

A DENOTATIONAL ENGINEERING OF PROGRAMMING LANGUAGES

to make software systems reliable
and user manuals clear, complete and unambiguous

A book in statu nascendi

(A working version)

Andrzej Jacek Blikle

in cooperation with Piotr Chrzastowski-Wachtel

*It always seems impossible until
it's done.*

Nelson Mandela

Warsaw, September 21st, 2021



„A Denotational Engineering of Programming Languages” by Andrzej Blikle in cooperation with Piotr Chrzastowski-Wachtel has been licensed under a Creative Commons: Attribution — NonCommercial — NoDerivatives 4.0 International. [For details see: https://creativecommons.org/licenses/by-nc-nd/4.0/legalcode](https://creativecommons.org/licenses/by-nc-nd/4.0/legalcode)

About the current versions of the book

The book has been initially written in two language versions — in Polish and in English. However, since September 2019, only the English version is publicly available. This is because, at that time, I have started to prepare presentations for my lectures, which I was giving in the summer semester 2020 and 2021 at the Department of Mathematics, Informatics, and Mechanics of Warsaw University. During this work, many corrections have been introduced into the English version, and therefore the Polish translation has been temporarily hidden from public access. Consecutive actualizations of the English version can be downloaded in PDF format from my website:

<http://www.moznainaczej.com.pl/what-has-been-done/the-book>

On the same site, there are a few articles associated with the book. All are also available from my accounts on ResearchGate, academia.edu, and arXiv.org, although there — due to a rather complicated process of manuscript replacement — the version of the book needs not to be the newest one.

I very warmly invite all my readers to send their remarks and questions about all aspects of the book. I am certainly aware of the fact that my English requires a lot of improvements, and therefore, I shall very much appreciate all linguistic corrections and suggestions as well. You may write to me on andrzej.blikle@moznainaczej.com.pl.

All interested persons are also invited to join the project *Denotational Engineering*. For more details about that project see:

<http://www.moznainaczej.com.pl/an-invitation-to-the-project>

A tag ??? in the book's text means that the tagged issue has to be rethought or completed.

Acknowledgments to the Polish version

Since June 2018, Polish version of my book has been made available to selected readers, which resulted in a flow of remarks. Below is the list of my colleagues whose observations contributed to the improvement of the content of the book. The order is chronological by the dates that comments were communicated to me. I express my heartfelt thanks to all contributors listed below.

Stanisław Budkowski, Antoni Mazurkiewicz, Marek Ryćko, Bogusław Jackowski, Ryszard Kubiak, Paweł Urzyczyn, Marek Bednarczyk, Wiesław Pawłowski, Krzysztof Apt, Jarosław Deminet.

My special thanks go to:

- Piotr Chrzastowski-Wachtel, who contributed with numerous remarks the earliest versions of the book,
- Stefan Sokołowski, who pointed out some inconsistencies of my earlier treatment of *assertions* and *invariants*,
- Jan Madey, who came with many bibliographical comments,
- Andrzej Tarlecki, who very carefully read the first half of the book, and passed me several valuable remarks, especially about the mechanisms of types and procedures.

Since March 2019, the Polish version of the book has been frozen and hidden from public access.

Acknowledgments to the English version

During winter 2020, Piotr Chrzastowski-Wachtel and Janusz Jabłonowski kindly agreed to go with me through my lectures, which I was preparing for a course at Warsaw University. Numerous discussions with them significantly contributed to the content and the clarity (I hope) of the book.

Similar acknowledgments are addressed to Katarzyna Wielgosz, Marcin Stańczyk, and Albert Cenkier from Poznań who spend with me several hours in listening to my lecture and discussing the details of my model.

Krzysztof Apt contributed with numerous substantial remarks and suggestions to the program-correctness model described in sections 7 and 8.

Radosław Waśko pointed out some inconsistencies in the definitions of quantifiers for yokes.

A technical remark to the reader of the “Word version” of the book

To protect the layout of formulas, set tabulators to 0,5 cm. Tabulator’s setting is a local parameter of a document which you set in the section

Main tools / Paragraph

of the tools panel, where at the lower right corner of that panel, you should click a small arrow.

Nelson Mandela’s quotation on the front page has been taken from

https://www.brainyquote.com/authors/nelson_mandela

Contents

Foreword.....	11
1 INTRODUCTION	13
1.1 Reverse the traditional order of things	13
1.2 What is in the book?.....	15
1.3 What is this book not offering?.....	16
1.4 What is original in this approach?.....	17
1.5 Lingua from bird's-eye view	18
1.5.1 Notational conventions	18
1.5.2 Data and (their) types	18
1.5.3 Abstract errors.....	21
1.5.4 Expressions	21
1.5.5 Instructions.....	22
1.5.6 Variable- and type declarations.....	24
1.5.7 Procedure declarations.....	24
1.5.8 Typological procedures	25
1.5.9 Object programming.....	26
1.5.10 SQL programming.....	27
1.5.11 Programs	27
1.5.12 Validating programming.....	27
2 METASOFT AND ITS MATHEMATICS.....	29
2.1 Basic notational conventions of MetaSoft.....	29
2.1.1 General mathematical notation	29
2.1.2 Sets	30
2.1.3 Functions.....	31
2.1.4 Tuples	33
2.2 Partially ordered sets	35
2.3 Chain-complete partially-ordered sets.....	36
2.4 A CPO of formal languages.....	38
2.5 Equational grammars.....	39
2.6 A CPO of binary relations	41
2.7 A CPO of denotational domains	44
2.8 Abstract errors.....	46
2.9 A three-valued propositional calculus.....	47
2.10 Data algebras.....	50
2.11 Many-sorted algebras	52
2.12 Abstract syntax and reachable algebras.....	56
2.13 Ambiguous and unambiguous algebras.....	59
2.14 Algebras and grammars.....	61

3	General remarks about denotational models	68
3.1	How did it happen?	68
3.2	From denotations to syntax.....	71
3.3	Languages of the Lingua family	71
3.4	Why do we need denotational models of programming languages?.....	72
3.5	Five steps to a denotational model.....	72
3.6	Notational conventions of our metalanguage	75
4	LINGUA-A — AN APPLICATIVE LAYER OF LINGUA	76
4.1	Lingua as a strongly-typed language.....	76
4.2	The general idea of the model of types	77
4.3	From data to values	78
4.3.1	Data.....	78
4.3.2	Bodies	82
4.3.3	Composites.....	86
4.3.4	Yokes	90
4.3.5	Types	96
4.3.6	Values	97
4.4	Expression denotations.....	100
4.4.1	Memory states	100
4.4.2	The algebra of denotations of Lingua-A.....	102
4.4.3	Denotations of data expressions.....	102
4.4.4	Denotations of body-, trace, yoke- and type expressions	107
4.4.5	Seven steps on the way to the algebra of expression denotations.....	109
4.5	Algebras of the syntax of expressions.....	111
4.5.1	Abstract syntax of Lingua-A.....	111
4.5.2	Concrete syntax of Lingua-A.....	114
4.5.3	Colloquial syntax of Lingua-A	118
4.5.3.1	A general rule for the layout of syntax	118
4.5.3.2	Boolean data-expressions	118
4.5.3.3	Numeric data-expressions.....	118
4.5.3.4	Array data-expressions.....	119
4.5.3.5	Record data-expressions.....	120
4.5.3.6	Transfer and yoke expressions.....	120
4.5.3.7	Type expressions	121
4.6	A sketch of the semantics of Lingua-A	121
4.7	Two forms of a manual.....	124
4.8	Main milestones on the way to language implementation	124
5	LINGUA-1 — AN IMPERATIVE LANGUAGE WITHOUT PROCEDURES.....	126
5.1	Denotations	126
5.1.1	Denotational domains.....	126
5.1.2	Conservative denotations.....	127
5.1.3	Programs	127
5.1.4	Declarations	127

5.1.4.1	Declarations of data variables	128
5.1.4.2	Declarations of body constants	129
5.1.4.3	Declarations of type constants	129
5.1.4.4	Trivial declaration	129
5.1.4.5	Structured declarations	129
5.1.5	Instructions	130
5.1.5.1	Sorts of instructions	130
5.1.5.2	Assignment instruction	130
5.1.5.3	Yoke-replacement instruction	131
5.1.5.4	Trivial instruction	131
5.1.5.5	Structured instructions	131
5.2	Syntax	133
5.2.1	Abstract syntax	133
5.2.2	Concrete syntax	134
5.2.3	Colloquial syntax	134
5.2.4	An example of a simple program	135
5.3	Semantics	136
6	LINGUA-2 — PROCEDURES	138
6.1	An introduction to a model of procedures	138
6.1.1	Procedures from a historical perspective	138
6.1.2	Procedures versus structured programming	138
6.1.3	Imperative procedures in a denotational framework	139
6.2	Communication between imperative procedures and programs	141
6.2.1	How does it work?	142
6.2.2	Constructors of parameter denotations	143
6.2.3	The compatibility of parameter-lists	144
6.2.4	Passing actual parameters to a procedure	146
6.2.5	Returning reference-parameters to a program	147
6.3	Imperative procedures with single recursion	148
6.3.1	Constructor of procedures	148
6.3.2	Procedure declaration	150
6.3.3	Recursion — how does it work?	150
6.3.4	Instruction of a procedure call	151
6.4	Imperative procedures with mutual recursion	152
6.4.1	Multiprocedures and their components	152
6.5	Functional procedures	153
6.5.1	The structure of a functional-procedure declaration	154
6.5.2	Functional procedures denotationally	154
6.5.3	Constructors of functional-procedure-denotation contents	155
6.5.4	The constructor of a functional procedure	155
6.5.5	The expressions of functional-procedure calls	156
6.5.6	The declaration of a functional procedure	157
6.5.7	Typological procedures ???	157
6.6	Procedures as parameters of procedures	157

6.7	Programs	158
6.8	Syntax and semantics	158
6.8.1	The signature of the algebra of denotations	158
6.8.1.1	The carriers of the algebra of denotations of Lingua-2	158
6.8.1.2	New constructors of the algebra of denotations	158
6.8.2	Concrete syntax	160
6.8.3	Colloquial syntax	161
6.8.4	Semantics	161
7	SEMANTIC CORRECTNESS OF PROGRAMS	164
7.1	Historical remarks	164
7.2	A relational model of nondeterministic programs	165
7.3	Iterative programs	166
7.4	Procedures and recursion	168
7.5	Three concepts of program correctness	169
7.6	Partial correctness	173
7.6.1	Sequential composition and branching	173
7.6.2	Recursion and iteration	174
7.7	Weak total correctness	178
7.7.1	Sequential composition and branching	178
7.7.2	Recursion and iteration	179
8	LINGUA-2V — VALIDATING PROGRAMMING	184
8.1	The structure of a validating language	184
8.2	Conditions	185
8.2.1	Generalities about conditions	185
8.2.2	Data-oriented conditions	187
8.2.3	Declaration-oriented conditions	187
8.2.4	Algorithmic conditions	188
8.3	Specified instructions	189
8.4	Propositions	191
8.4.1	Syntactic propositions	192
8.4.2	Metaconditions	192
8.4.3	Metainstructions	195
8.4.4	Metaprograms	195
8.4.5	Jaco de Bakker paradox in Hoare's logic	197
8.5	Constructing correct metaprograms	198
8.5.1	The role of declarations in the derivation of correct metaprograms	198
8.5.2	Basic rules	199
8.5.3	Imperative procedures	203
8.5.4	Recursive procedures	206
8.5.5	Functional procedures	207

8.5.6	Invariants versus assertions	209
8.6	Transformational programming	211
8.6.1	First example.....	211
8.6.2	Changing data-types	217
8.6.3	Adding a register identifier	219
9	RELATIONAL DATABASES INTUITIVELY	222
9.1	Preliminary remarks	222
9.2	Simple data	222
9.3	Creating tables	225
9.4	The subordination relation for tables	227
9.5	Instructions of table modification	228
9.6	Transactions	229
9.7	Queries.....	231
9.8	Aggregating functions	233
9.9	Views.....	233
9.10	Cursors.....	234
9.11	The client-server environment	235
10	Lingua-SQL.....	236
10.1	General assumptions about the model.....	236
10.2	Data	236
10.3	Subordination of tables.....	237
10.4	Bodies	239
10.5	Composites	241
10.5.1	Signatures of composite constructors.....	242
10.5.2	Constructors of simple composites.....	243
10.5.3	Constructors of row composites.....	243
10.5.4	Row constructors of table composites	244
10.5.5	Column constructors of table composites	248
10.5.6	A referential constructor of table composites	251
10.6	Yokes.....	253
10.6.1	A row transfer	253
10.6.2	Table yokes	254
10.7	Types	255
10.8	Values.....	255
10.8.1	Simple values, row values, and table values.....	255
10.8.2	Database values	257
10.9	The algebra of denotations.....	257
10.9.1	States and denotational domains	257
10.9.2	Data-expression denotations	258

10.9.3	Type-expression denotations.....	259
10.9.4	Denotations of type-constant declarations.....	260
10.9.5	Denotations of data-variable declarations.....	261
10.9.6	Instructions.....	261
10.9.6.1	Categories of SQL instructions.....	261
10.9.6.2	Row instructions.....	262
10.9.6.3	Two universal constructors of table assignment.....	262
10.9.6.4	Transactions.....	265
10.9.6.5	Global table instructions.....	269
10.9.6.6	Local table instructions.....	270
10.9.6.7	Queries.....	270
10.9.6.8	Transfer-replacement instructions.....	270
10.9.6.9	Cursors.....	271
10.9.6.10	Views.....	271
10.9.6.11	Database instructions.....	271
10.10	Concrete syntax.....	273
10.11	Colloquial syntax.....	276
10.12	The rules of correct-program constructions.....	278
11	LINGUA-OO — OBJECT-ORIENTED PROGRAMMING (WORK IN PROGRESS).....	279
12	WHAT REMAINS TO BE DONE.....	280
12.1	Foundations.....	280
12.1.1	The extension of Lingua model.....	280
12.1.2	The completion of the Lingua model.....	280
12.1.3	The principles of writing user manuals.....	280
12.2	Implementation.....	280
12.2.1	Tools for language developers.....	281
12.3	Tools for programmers.....	281
12.4	Manuals.....	281
12.5	Programming experiments.....	281
12.6	Building a community of Lingua supporters.....	281
13	ANNEXE 1 — Generalized trees.....	283
14	ANNEXE 2 — About user manuals.....	283
15	REFERENCES.....	284
16	INDICES AND GLOSSARIES.....	288
16.1	Index of terms and authors.....	288
16.2	Index of notations.....	291
16.3	Glossary of algebras and domains.....	292

FOREWORD

When in 1990, I decided to run my family business¹ “for a while” — which took me two decades — I already had a plan for my book on denotational models of programming languages. It was the result of my research for nearly thirty years, starting in 1962 after I graduated from The Department of Mathematics and Physics of Warsaw University. I began my work in a group of young researchers who planned to build mathematical tools for software engineering. At that time there were only a few such groups in Poland and maybe 20-30 in the World. Although our approaches were technically different from each other, we were sharing mostly the same opinion about state of the art in software engineering. Let me try to sum it up now in a few lines.

In each engineering — except software engineering — the designing process of a new product starts with a blueprint supported by mathematical calculations. Both provide a mathematical warranty that the future functionality of the product will satisfy the expectations of the designer and the future user.

In the IT industry, the situation was different. In the place of a blueprint and calculations, programmers (i.e., producers) were given an informal description of the future product in a natural language, like plain English or Polish. As a consequence, a bulk of the budget for product-development was spent on testing, i.e., removing errors introduced at the stage of coding. Since testing may only discover errors but never gives a guarantee of their absence, the remaining bugs were passed on to the user to be removed later under the name of “maintenance”. In some cases, these situations were leading to spectacular catastrophes. Here are a few examples:

- the death of six patients in US hospitals as a result of a wrong computer-computations of radiation dosage (1985),
- the catastrophe of an American lander of the Venus planet (the 1980-ties),
- the catastrophe of an oil platform in a Norwegian fiord (1991),
- Airbus crash in Warsaw (1993)²,
- an overlooking of Lothar hurricane by German meteorological services (1999),
- a rounding error in Intel’s microprocessor (1995).

That was the situation in the past. And how is it today? Today software products are a few orders of magnitude bigger, and the number of their users grows exponentially. However, the problems mentioned above have not disappeared. The following statistics concerns software products of a total value of 250 billion USD (see [1]):

- 88% of projects exceeded the planned realization time and/or budget,
- the average overrun of the assumed budget was 189%,

¹ I was borne in a family of Warsaw’s confectioners who’s firm was established in 1869. The business survived two world wars and 45 years of communist time, hence when our country became independent again in 1989, I decides to develop our family business according to the European standards. My father passed away many years ago and my son was too young to take the business over. My preliminary plan was to stay in the business for a few years only and then to come back to my beloved research. The life turn out, however, more difficult than I expected.

² In this case, although the cause of the accident had its origin in the software, this error was not due to programmers, but to the aircraft engineers, who did not anticipate certain specific aerodynamic conditions that may occur during the landing of the aircraft. In effect, they passed a wrong specification to programmers. For this information, I am thankful to Jarosław Deminet.

- the average overrun of assumed realization-time was 222%.

It is also a well-known fact that every user of a software application has to accept a disclaimer. Here is a typical example dating from 2018:

There is no warranty for the program, to the extent permitted by applicable law. Except when otherwise stated in writing the copyright holders and/or other parties provide the program "as is" without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the program is with you. Should the program prove defective, you assume the cost of all necessary servicing, repair, or correction.

Is it thinkable that a producer of a car, a dishwasher, or a building could request such a disclaimer from his client? Why then is the software industry an exception?

In my opinion, the cause of this situation is a lack of such mathematical models and tools for software engineers that would guarantee the functional reliability of products based on the way they have been designed and manufactured. The lack of mathematical models for programming languages also affects user-manuals of these languages, which again contributes to a low quality of programs.

In the field of user manuals, I do not see progress either. A published in 1960 report on Algol 60 (see [5]) — a language, which largely influenced the development of several generations of programming languages — far surpassed today's manuals regarding not only the precision and the completeness of language descriptions but also their compactness³.

First, their syntax was described by generative Chomsky's grammars rather than — as today — by (usually unclear) examples.

Second, their semantics, although defined without any mathematical tools (they were not known at that time) was described with the use of well-defined technical concepts such as *variable*, *block*, *variable-visibility*, *procedure*, *procedure-parameter*, *recursion*. Ten years later, the manual of Pascal [56] was written in a similar style⁴.

Unfortunately, one cannot say the same about today's manuals, where the authors do not distinguish expressions from instructions and instructions from declarations.

The described situation is common not only for programming languages but also for many applications such as e.g., Content Management System Joomla! and Drupal. Their poor descriptions cause the growing popularity of support forums, where desperate users exchange their own experiences. Manuals are rarely used because they are not only imprecise and incomplete but highly unreadable due to their language lacking conceptual apparatus, and to their volume. For instance, Algol 60 manual contained 237 pages and Pascal manual — 166 pages, whereas in the case of Python [69] we have 696 pages, for Access [80] — 952 pages and the manual of Delphi that was supposed to become — as some of its predecessors — the universal language of programming for all times, exceeds 2000 pages.

The users' forums are therefore filled up with questions like "Hey, does anyone know how to ...?", to which most frequently nobody answers. From my practice, for three questions asked by me, two remain unanswered. I only find related questions asked by others, which convince me that I am not alone with my problem.

3 Similar remarks can be made about a Polish manual [71] of Algol 60.

4 Similar remarks are true for a Polish manual [63] of Pascal.

1 INTRODUCTION

1.1 Reverse the traditional order of things

The problem of mathematically-provable program-correctness appeared for the first time in a work of Alan Turing [78] published in conference-proceedings *On High-Speed Calculating Machines*, which took place at Cambridge University in 1949. Later, for several decades, that subject was investigated usually under the name of *proving program correctness*, but the developed methods never became standard tools of software engineers. Finally, all these efforts were abandoned what has been commented in 2016 by the authors of a monography *Deductive Software Verification* [2]:

For a long time, the term formal verification was almost synonymous with functional verification. In the last years, it became more and more clear that full functional verification is an elusive goal for almost all application scenarios. Ironically, this happened because of advances in verification technology: with the advent of verifiers, such as KeY, that mostly cover and precisely model industrial languages and that can handle realistic systems, it finally became obvious just how difficult and time-consuming the specification of the functionality of real systems is. Not verification, but specification is the real bottleneck in functional verification.

In my opinion, the failure of constructing a practical system for proving or making programs correct has two sources.

The first lies in the fact that in building a programming language, we start from syntax and only later — if at all — define its semantics. The second source is somehow similar but concerns programs: we first write a program and only then try to prove it correct.

To build a logic of programs for a programming language, one must first define its semantics on a mathematical ground. Since 1970-ties it was rather clear for mathematicians that such semantics to be “practical” must be compositional, i.e., the meaning of a whole must be a composition of the meanings of its parts. Later such semantics were called *denotational* — the meaning of a program is its *denotation* — and for about two decades, researchers investigated the possibilities of defining denotational semantics for existing programming languages. Two most complete such semantics were written in 1980 for Ada [15] and for CHILL [36] in using a metalanguage VDM [14]. A little later, but in the same decade, a minor exercise in this field was a denotational semantics of a subset of Pascal written by myself in MetaSoft [24], the latter based on VDM.

Unfortunately, none of these attempts resulted in the creation of software-engineering tools that would be widely accepted by the IT industry. In my opinion, that was unavoidable since, for the existing programming languages, a full denotational semantics simply cannot be defined (see Sec. 3). That was, in turn, the consequence of the fact that historically syntaxes were coming first, and only later, researchers were trying to give them a mathematical meaning. In other words — the decision of how to describe preceded a reflection of what to describe.

In addition to that, two more issues were complicating denotational models of programming languages. They were related to two mechanisms considered necessary in 1960-ties but ten years later almost totally abandoned. One was a common *jump instruction goto*, the other — specific procedures that may take themselves as parameters (Algol 60, see [5]). The former had led to the *continuations* (see [75]), the latter to *reflexive domains* (see [74]). Both contributed to the technical complexity of denotational models, which was discouraging not only for practitioners but also for mathematicians.

The second group of problems followed from an implicit assumption that in the development of mathematically correct programs, the development of programs should precede the proofs of their correctness. Although

this order is quite evident in mathematics — first a theorem (a hypothesis) and then its proof — it is somewhat unusual for engineers who should first perform all necessary calculations (the proof) and only then build their bridges or airplanes.

The idea “first a program and its correctness proof later” seems not only irrational but also practically rather unfeasible for two reasons.

First reason follows from the fact that a proof of a theorem is usually longer than the theorem itself. Consequently, proofs of program correctness should contain thousands, if not millions of lines. It makes “hand-made proofs” somewhat unrealistic. In turn, automated proofs were not possible due to the lack of formal semantics of existing programming languages.

Even more critical seem, however, the fact that programs that are supposed to be proved correct are usually incorrect! Consequently, correctness proofs are regarded as a method of detecting errors in programs. It means that we are first doing things wrong to correct them later. Such an approach does not seem very rational, either.

As an attempt to cope with the mentioned problems, I am showing in the book a mathematical method of designing programming languages with denotational semantics. To illustrate this method, an exemplary programming language **Lingua** is developed from denotations to syntax (first publication of that method in [25]). In this way, the decision of what to do (denotations) precedes the choice of how to express it (syntax).

Mathematically both the denotations and the syntaxes constitute many-sorted algebras (Sec. 2.11), and the associated semantics is the homomorphism from syntax to denotations. As turns out, there is a simple method — to a large extent, algorithmizable — of deriving syntax from (the description of) denotations and the semantics from both of them.

At the level of data structures, **Lingua** covers booleans, numbers, texts, records, arrays, and their arbitrary combinations plus SQL databases. It is also equipped with a relatively rich mechanism of types, e.g., covering SQL-like integrity constraints and with tools allowing the users to define their own types structurally. At the imperative level, this language contains structured instructions, type definitions, procedures with recursion and multi-recursion, and some preliminaries of object programming.

The issue of concurrency is not tackled in the book since the development of a “fully” denotational semantics for concurrent programs (if at all possible) would require separate research⁵.

Of course, **Lingua** is not a real language since otherwise, the book would become unreadable. It is only supposed to illustrate the method which (hopefully) may be used in the future to design and implement a real language of sequential programming.

Once we have a language with denotational semantics, we can define program-construction rules that guarantee the correctness of programs. This method was for the first time sketched in my paper [21], and in this book is described in Sec. 7 and Sec. 8⁶. It consists in developing so-called *metaprograms*, which are programs that syntactically include their specifications. The method guarantees that if we compose two or more correct programs into a new program or if we transform a correct program, we get a correct program again. The correctness proof of a program is hence implicit in the way the program has been developed.

Basic mathematical tools used in the book are the following:

1. fixed-point theory in partially ordered sets,
2. the calculus of binary relations,
3. formal-language theory and equational grammars,

⁵ There exist mathematical semantics of concurrency which can be said to be only “partially denotational”. An example of such a solution is a “component-based semantics” (cf. [13]), where the denotations of programs’ components are assigned to programs in a compositional way (i.e. the denotation of a whole is a composition of the denotations of its parts), but the denotations themselves are so called *fucons* whose semantics is defined operationally.

⁶ The philosophy of developing correct programs rather than proving programs correct, although not based on denotational semantics, was elaborated nearly a half century ago by Edsger W. Dijkstra on the ground of his method of weakest preconditions (see [44] and [45]).

4. fixed-point domain-equations based on so-called *naive denotational semantics* (cf. [33]),
5. many-sorted algebras,
6. abstract errors as a tool for the description of error-handling mechanisms,
7. three-valued predicate calculi of McCarthy and Kleene,
8. the theory of total correctness of programs with clean termination.

These tools are described in Sec. 2 and Sec. 7, which should make the book self-contained. The reader is only expected to be familiar with the preliminaries of set theory and mathematical logic and to have basic experience in programming.

In constructing **Lingua**, I assumed three priorities regarding the choice of programming mechanisms:

- the priority of the simplicity of the model — the simplicity of denotations, syntax, and semantics; e.g., the resignation from **goto** instruction and self-applicative procedures,
- the priority of the simplicity of metaprogram construction rules; e.g., the assumption that the declarations of variables, types, and procedures should always be placed at the beginning of a program,
- the priority of protection against “oversight errors” of a programmer; e.g., the resignation of global variables in imperative procedures and of side-effects in functional procedures.

All these commitments forced me to give up some programming constructions which — although denotationally definable — would lead to complicated descriptions and even more complicated program-construction rules.

It is worth mentioning in this place that the priority of simplicity is not new in the history of programming languages. For that very reason, programming-language designers abandoned **goto**-s (see [43]) as well as self-applicative procedures.

1.2 What is in the book?

I am deeply convinced that one can talk about programming in a precise and transparent way. I also believe that taking responsibility for their products by software engineers should be possible in the same way as it is in the case of the engineers of cars, bridges, or airplanes. On the other hand, I am aware that the existing tools for software engineers do not allow for the realization of any of these goals.

As I mentioned already in the Foreword, the book contains many thoughts developed in the years 1960-1990 that later have been abandoned. One of the few teams developing these ideas was working in the Institute of Computer Science of the Polish Academy of Sciences, and I had the pleasure to chair it. At that time, we were developing a semi-formal metalanguage called **MetaSoft** dedicated to formal definitions of programming languages (cf. [24]). This language is used in the book.

The book starts with a short description of **Lingua**, which is later developed and described throughout the book.

Sec.2 is devoted to the introduction of all mathematical tools that are listed in Sec. 1.1 except the program-correctness issue.

Sec.7 includes a general theory of partial and total correctness of programs. These concepts are formulated on the ground of binary relations, which allows concentrating on the main subject without technical details of programming languages.

The remaining part of the book is devoted to the construction of denotational models for successive programming mechanisms in the **Lingua** series.

Sec.3 contains a general discussion of algebraic and denotational models of programming languages that are later exploited in the subsequent sections of the book.

Sec. 4 is devoted to the development of a general model of data structures and types that can be used to describe data- and type-mechanisms of a sufficiently large class of programming languages. In this model, a type is a pair that consists of a *body* that describes the structure of a data, e.g., a list of records, and a *yoke* that describes other properties of data, e.g., that in each of these records the sum of numbers assigned to attributes **salary** and **commission** should be less than 10.000. Such yokes are typical in SQL-based languages, although they are not named in this way there. A language covering these mechanisms is called **Lingua-A** (A stands for “applicative”). It consists of expressions only, i.e., contains neither declarations nor instructions. It is not a prototype of an applicative language but only an applicative layer of a general-purpose programming language.

Sec.5 contains a model of **Lingua-1** that covers the whole **Lingua-A** plus structured instructions, variable and type- declarations, and some mechanism allowing programmers to build types in a bottom-up way. Types may be given names that are later stored in memory.

In Sec.6, **Lingua-1** is enriched to **Lingua-2** by introducing procedures both imperative and functional. Recursion and multi-recursion are covered as well.

Sec.8 is devoted to the idea and techniques of *validating programming*, which I investigated in the years 1970/80. As was already explained in Sec.1.1, it consists in building metaprograms by using constructors that guarantee metaprogram correctness. The language for validating programming in **Lingua-2** is called **Lingua-V2** (V for “validating”).

Sec.9 and 10 are devoted to the extension of **Lingua-2** by some tools typical for SQL-based languages. That version of **Lingua** is called **Lingua-SQL**.

I am aware of the fact that the content of the book represents a very restricted part of the world of today’s programming languages. However, something had to be chosen to begin with. **Lingua** contains, therefore, only a selection of programming tools that have been known for many years, and that are still in use. In the future, I shall try to complete my models with those vehicles that my readers will consider necessary. I also hope that maybe some of you will undertake this challenge. Please feel invited to cooperate.

1.3 What is this book not offering?

As I explained in the Foreword and in Sec.1.1, the reason why I have written this book is the lack of mathematical tools that would allow software producers to take such responsibility for their products as is usual in many other industries such as, e.g., automotive or aircraft manufacturing or in the sector of civil engineering. It does not mean, however, that the book offers a tool ready to be used today by IT industry. What I am trying to offer is only a suggestion of where to research for such tools and an associated mathematical framework.

To better explain what I mean, let me refer to the concept of *product quality* as understood in the field of *Total Quality Management*. By the quality of a product, we mean the degree of the satisfaction of its user. Product quality is usually measured by the number of faults in the product — the fewer faults, the higher the quality — where an error is any such product’s property that the user “has the right not to expect”. E.g., if we order a beer, we have the right not to expect it to be warm, unless we are requesting a mulled beer.

The quality of a product is therefore not an immanent property of a product, but rather a relation between a product and the expectations of its user. Paradoxically we can increase the quality of a product without changing the product itself when we honestly describe all its faults. Unfortunately, this approach is not a usual practice since it would lower the chances of selling the product.

In the case of software, user expectations are described by a specification that a program should fulfill. The quality of a program consists therefore in:

1. the compatibility of program specification with the expectations of its user,
2. the compatibility of the program itself with the specification.

In my book, I am tackling only the second aspect. My choice is not caused by the fact that the first problem is less important, or that it has already been solved, but only because the second problem was the main subject of my research for two decades and therefore I dare to talk about it now⁷.

In the end, I have to very strongly emphasize again that the virtual language **Lingua** is not regarded as a practical programming language, although maybe such a language will grow from **Lingua** in the future. At present, it only offers a platform where to explain the constructions and the models discussed in the book. I have tried to cover in **Lingua** the majority common mechanisms that are present in languages that are known to me today.

1.4 What is original in this approach?

By “this approach”, I understand the ideas and techniques described in my early papers from [18] to [28] published in the years 1972-2020, which have been summarised and extended in the present book. All these ideas are based on concepts known for a long time. In the list below I give references to the earliest papers on a given subject, and to major contributions that followed.

- denotational semantics of D. Scott’s and Ch. Strachey’s ([75] 1971, [74] 1977),
- generative grammars of N. Chomsky ([37] 1956, [39] 1957, [40] 1959, [41] 1962, [49] 1966, [16] 1971),
- C.A.R Hoare’s logic of programs (the founding paper [55] 1969, and surveys [4] 1981, [5] 2020, [6] 2020),
- many-sorted algebras introduced to the mathematical foundations of computer science by J. A Goguen, J.W, Thatcher, E.G. Wagner and J.B Wright ([51] 1977),
- three-valued propositional calculus of J. McCarthy (cf. [65] 1967),
- abstract errors in program’s semantics originally introduced by Joe Goguen ([51] 1978, [23] 1981, [11] 1984, [24] 1987 [77] 1988, [26] 1988)

What, I believe, is — in this or another way — original in the presented approach, is the following:

1. Programming language design and development:

- 1.1. A formal, and to a large extent, an algorithmic method of systematic derivation of syntax from denotations and a denotational semantics from both of them ([25] 1987, [27] 1989).
- 1.2. The idea of a colloquial syntax which allows making syntax user-friendly without damaging a denotational model ([27] 1989).
- 1.3. The systematic use of error-elaboration in programs supported by a three-valued predicate calculus ([23] 1981, [26] 1988).
- 1.4. Denotational model based on set-theory rather than on D. Scott’s reflexive domains, which makes the model much simpler and easier to be formalized ([33] 1983).
- 1.5. A model of data-types that covers not only structured and user-defined types but also SQL-like integrity constraints (this book).

2. The development of correct programs

- 2.1. A method of systematic development of correct programs with their specifications, seen as an alternative, to proving the correctness of (earlier developed) programs ([21] 1979, [22] 1981)
- 2.2. The use of three-valued predicates to enrich Hoare’s logic by a clean termination property ([23] 1981).

⁷ I am deeply convinced that the first problem is equally fascinating as the second. I would very much welcomed any initiative of a cooperation in this field.

3. General mathematical tools

- 3.1. Equational grammars applied in defining the syntax of programming languages ([18] 1972).
- 3.2. A three-valued calculus of predicates applied in designing programming languages and in defining sound program constructors for such languages ([21] 1979, [22] 1981).

1.5 Lingua from bird's-eye view

To structure my book, **Lingua** is built layer-by-layer, as explained in Sec.1.2. Below I show a condensed and only half-formalized description of the language without entering into technical details. I also refrain from describing the process of language development and concentrate on its target version. This section is addressed to the readers who wish to grasp the idea of **Lingua** before they proceed to its technical details.

1.5.1 Notational conventions

Below I shall use the following notation (full description and justification in Sec. 2.1):

- $a : A$ means that a is an element of the set A ; according to the denotational dialect “sets” are most frequently called “domains”,
- $f.a$ denotes $f(a)$, and $f.a.b.c$ denotes $((f(a))(b))(c)$; intuitively f takes a as an argument and returns the value $f(a)$ which is a function which takes b as an argument and returns the value $(f(a))(b)$, which is again a function...
- $A \rightarrow B$ denotes the set of all *partial functions* from A to B , i.e., functions possibly undefined for some elements of A ,
- $A \mapsto B$ denotes the set of all *total functions* from A to B , i.e., functions undefined for all elements of A ; of course, each total function is a particular case of a partial function,
- $A \Rightarrow B$ denotes the set of all function from A to B defined for only finite subsets of A ; such functions are called *mappings*, and of course, each mapping also is a particular case of a partial function,
- $A|B$ denotes the set-theoretic union of A and B ,
- $A \times B$ denotes the Cartesian product of A and B ,
- tt and ff denote logical values „true” and „false” respectively,
- many-character symbols like `dom`, `bod`, `com` denote metavariables running over domains and, if written with parentheses as `'abdsr'` denote themselves, i.e., metaconstants.

In order to distinguish between meta-level of phrases written in **MetaSoft** and the level of phrases written in **Lingua**, the former level will be typeset in `Arial` and the latter in `Courier New`.

1.5.2 Data and (their) types

So far data in **Lingua** may be split into three groups:

- *simple data* including booleans, numbers, and words (finite strings of characters),
- *structural data* including list, many-dimensional arrays, records, and their arbitrary combinations,
- *SQL-data*, including rows and tables that carry simple data and databases that carry tables.

Structural data may „carry” simple data as well as other structural data. That means that we may build “deep” data structures, e.g., records that carry lists of arrays. Lists and tables carry data of the same type, whereas in records, data assigned to attribute may be of different types.

Arrays are formally one-dimensional, but since their elements may be other arrays, we may construct arrays of arbitrary dimensions.

Databases are — simplifying a little — records of tables, i.e., finite functions from identifiers to tables, tables are — simplifying again — one-dimensional arrays of rows and rows are records that carry simple data.

Lingua has been equipped with a mechanism of types that covers the typical mechanism of programming languages. By a “mechanism of types,” I understand programming tools that allow programmers to define their types for future use either in declaring new types or in declaring variables. This mechanism is described in Sec. 4.3.5, and in Sec. 4.4.4.

Types are pairs consisting of a *body* and a *yoke*. With every type, we associate a set of data called the *clan* of this type.

Intuitively a body describes the “internal structure of a data” — e.g., that a data is a number, a list, or a record — and formally is a combination of tuples and mappings. The bodies of simple data are one-element tuples of words: (`'boolean'`), (`'number'`) or (`'word'`). The bodies of lists and arrays are respectively of the form (`'L'`, body) or (`'A'`, body) where body is shared by all the elements of a list/array and where the *initials* `'L'` and `'A'` indicate that we are dealing with a list type or with an array type respectively. A record body is of the form (`'R'`, body-record) where body-record is a record of bodies such as, e.g.:

```

Ch-name      : ('word'),
fa-name      : ('word'),
birth-year   : ('number'),
award-years  : ('A', ('number')),
salary       : ('number'),
bonus        : ('number')

```

(1.5-1)

The words on the left-hand-side of colons are identifiers called *attributes*. The first three attributes and the last two have simple bodies, whereas the fourth one — an array body. For the sake of further discussions, this record-body will be referred to as **employee**.

With each body **bod**, we associate the set of data denoted by **CLAN-Bo.bod**. The function **CLAN-Bo** is defined inductively over the structure of bodies. E.g., the set **CLAN-Bo.employee** contains records with numbers, words, and one-dimensional number arrays assigned to the attributes.

Next important concept from the “world” of data and types is a *composite* that is a pair (**dat**, **bod**) consisting of a data and its body, i.e., such that:

```
dat : CLAN-Bo.bod
```

Having defined composites, we can define *transfers* and *yokes*. Transfers are one-argument functions that transform composites into composites, and *yokes* are transfers with boolean composites as values. By a *boolean composite*, we mean (**tt**, (`'boolean'`)) or (**ff**, (`'boolean'`)). Transfers may also assume abstract errors as values (see later).

Mathematically yoks are close to one-argument predicates on composites⁸. An example of a yoke expression that describes a property of composites whose body is **employee** is the following⁹:

```
record.salary + record.bonus < 10000,
```

The yoke which is the denotation of this expression is satisfied whenever its argument is a record composite with (at least) the attributes `salary` and `bonus`, and the data corresponding to these attributes satisfy the corresponding inequality. In this example

```
record.salary + record.bonus
```

⁸ They “are closed to predicates” rather than simply “are predicates” since they assume as values composites and abstract errors rather than just boolean values **tt** and **ff**. Consequently their logical constructors **and**, **or** and **not** are not the classical constructors but three-valued constructors of a calculus defined by John McCarthy (Sec. 2.9).

⁹ Here we anticipate future notational conventions according to which syntax of **Lingua** is typeset in *Courier New*.

is a transfer that is not a yoke. It transforms record composites into number composites.

Yokes understood in our way appear in SQL and are called *integrity constraints*. As a matter of fact, they have been introduced into our model in order to cope with SQL data, although now they seem to may also have other applications.

Transfers have merely a technical role. We need them only to define an algebra where yokes may be built. With every yoke we associate its clan:

$$\text{CLAN-Yo.yok} = (\text{com} \mid \text{yok.com} = (\text{tt}, (\text{'boolean'})))$$

which consists of composites that satisfy that yoke.

A pair that consists of a body and a yoke is called a *type*. With every type $\text{typ} = (\text{bod}, \text{yok})$ we associate its *clan*, which is the set of data whose body is *bod*, and such that (dat, bod) satisfies *yok*. Therefore:

$$\text{CLAN-Ty.}(\text{bod}, \text{yok}) = \{\text{dat} \mid \text{dat} : \text{CLAN-Bo.bod} \text{ and } (\text{dat}, \text{bod}) : \text{CLAN-Yo.yok}\}$$

The last concept associated with data and types is a *value*, also called *typed data*. A value is a pair (dat, typ) , such that $\text{dat} : \text{CLAN-Ty.typ}$. A value is, therefore, a pair $(\text{dat}, (\text{bod}, \text{yok}))$, which can be written as $((\text{dat}, \text{bod}), \text{yok})$, i.e., (com, yok) or as $(\text{dat}, \text{bod}, \text{yok})$. In other words, value may be regarded either as a pair *data-type* or as a pair *composite-yoke*, or as a triple.

Values are created in the course of data-expressions evaluation (see a little later). All data operations in **Lingua** are defined as operations on values equipped with a mechanism of checking if the arguments “delivered” to an operation have appropriate bodies. E.g., if we try to put a word on a list of numbers, the corresponding operation will generate an error message.

Values are assigned in memory states to the identifiers of variables and are passed to procedure calls as actual parameters. An assignment instruction — i.e., an instruction that assigns a value to a variable — may only change the data assigned to a variable but never its type. Yokes may only be changed by a special yoke-oriented instruction.

Let us sum up the list of mathematical entities associated with the concepts of data and their types:

- *data* are primary mathematical beings processed by programs,
- *bodies* are finitistic objects that describe “internal structures” of data,
- *composites* are pairs (dat, bod) , where $\text{dat} : \text{CLAN-Bo.bod}$,
- *transfers* are one-argument functions on composites and errors, and *yokes* are transfers that return boolean composites or abstract errors as their values,
- *types* are pairs that consist of a body and a yoke; as we are going to see later, in memory states types will be assigned to *type constants*, and *type expressions* will evaluate to types,
- *values* are pairs consisting of a data and (its) type; in states, data are assigned to *variable identifiers*, and data expressions evaluate to values.

Similarly, as in many programming languages (although not in all of them) types in **Lingua** have been introduced for four reasons:

1. to define a type of a variable when it is declared, and to assure that this type remains unchanged during program executions,
2. to ensure that a data which is assigned to a variable by an assignment is of the type consistent with the type of that variable,
3. to ensure that a similar consistency takes place when sending actual parameters to a procedure or when returning reference parameters by a procedure,
4. to ensure that in evaluating an expression, an error message is generated whenever data “delivered” to a data constructors are of inappropriate types, e.g., when we try to add a word to a number.

1.5.3 Abstract errors

An essential feature of **Lingua** is the inclusion of error messages in its model. For this purpose, the domains (sets) of bodies, composites, and types are “equipped” with elements that are called *errors*. Mathematically errors may be anything, but in **Lingua** they are words, e.g.

‘division-by-zero’ or
‘record-expected’

that intuitively describe the cause of an error. All operations on composites, bodies, and types are also defined on errors, and the majority of them are *error-transparent*. The latter means that if an argument of an operation is an error, then the resulting value is the first error that appears in the course of a computation. Intuitively the appearance of an error means that program execution aborts, and an error message is displayed on a monitor. It may also happen that an error causes the execution of an *error handling procedure* (see Sec. 5.1.5.5 and Sec.10.9.6.4).

A special case of error-handling operations are boolean operations (Sec.2.9) that handle errors in accordance with McCarthy’s propositional calculus. For instance:

ff and $ee = ff$
 ee and $ff = ee$

where ee represents an error or a non-terminating computation. The arguments of conjunction are evaluated from left to right, and if the first argument evaluates to ff , then the evaluation of the second argument is skipped. In this way, we maybe avoid an error message or a non-terminating evaluation. E.g. the boolean expression

$x \neq 0$ **and** $1/x > 10$

assumes the value ff for $x=0$ even though $1/x > 10$ would generate an error or would loop infinitely. In McCarthy’s calculus whenever $x = 0$, the evaluation of $1/x > 10$ is postponed.

Particular cases where errors are signaled are overflows. Formally for every domain of data, a predicate is defined that “reacts” to overflows.

1.5.4 Expressions

Expressions are syntactic element and their *denotations*, i.e. their semantic meanings, are functions from states to values (*data expressions*) or from states to types (*type expressions*). In order to define these concepts we have to start with the definition of a domain of *states*. This domain is defined by so called *domain equations*:

State	= Env x Store	(state)
Env	= TypEnv x ProEnv	(environment)
Store	= Valuation x (Error {'OK'})	(store)
Valuation	= Identifier \Rightarrow Value	(valuation)
TypEnv	= Identifier \Rightarrow Type	(type environment)
ProEnv	= Identifier \Rightarrow Procedure Function	(procedure environment)

where **Error** is some fixed set of errors. As we see, states are storing values, types, procedures, and functions (functional procedures) assigned to identifiers as well as errors stored in a “dedicated register”. If a state does not carry an error then this register stores the word ‘OK’.

The *denotations of data expressions* and the *denotations of type expressions* are the elements of an *algebra of expression denotations* from which a syntactic *algebra of expressions* may be derived. A function from expressions to their denotation is called the *semantics of expressions*¹⁰.

Data-expression denotations are partial functions from states to composites or error messages:

¹⁰ This “function” is in fact a homomorphism from the algebra of expressions into the algebra of expression denotations.

$\text{DatExpDen} = \text{State} \rightarrow \text{Value} \mid \text{Error}$

For every operation on data, there is an operation on composites, and for every operation on composites, there is a constructor of data-expression denotations. E.g., the denotation of the expression

$x + y$

is a function that for a given state sta first successively checks the following conditions:

- Is sta carrying no error?
- Are there any values assigned to identifiers x and y in sta ?
- Are these values numbers?
- Is their sum less than the largest number representable in the current implementation?

If all these checks terminate positively, then the function creates a value $(\text{dat}, (\text{'number'}, \text{TT}))$, where dat is the sum of the numbers assigned to x and y , and TT is a yoke which is always satisfied. If some of these checks do not terminate successively then an appropriate error message is generated, e.g.,

`'number-expected'`

and the computation terminates. In particular, if the input state carries an error, then this error becomes the result of the computation.

Notice that data expressions represent partial functions since they may call functional procedures whose executions may loop infinitely.

Contrary to data expressions, the denotations of type-expressions are total functions from states to types or error messages:

$\text{TypExpDen} = \text{State} \mapsto \text{Type} \mid \text{Error}$

The constructors of such denotations are defined similarly as for data expressions, although now they base on type operations rather than on data operations. E.g., the denotation of the following type expression:

```
record-type
  Ch-name      as word,
  fa-name      as word,
  birth-year   as number,
  award-years  as number-array ee
  salary       as number
  bonus        as number
ee
```

is a function on states that creates a record type or generates an error. This expression refers to two built-in types `word` and `number` and one user-defined type `number-array` (arrays of numbers). A typical case of a type-expression evaluation generating an error may be an operation of the removal of an attribute of a record-type if this attribute does not appear in the record.

Data-expression denotations and type-expression denotation, together with their constructors constitute an *expression-denotation algebra* (Sec. 4.4). From that algebra, we derive its syntactic counterpart — an *expression algebra* (Sec. 4.5).

1.5.5 Instructions

Expressions belong to the *applicative part* of our language. Their denotations take states as arguments but neither create them nor change. Such tasks are performed by *instructions*, and by *variable-*, *type-* and *procedure declarations*. All of them belong to the *imperative layer of the language*.

Instruction denotations are partial functions from states to states, which means that their domain is the following:

InsDen = State \rightarrow State

Contrary to expression denotations, which may generate an error, instruction denotations write an error to the error register. The denotations of the majority of instructions are *transparent* relative for error-carrying states, i.e., they do not change such states but only pass them to the subsequent part of a program. However, an error may also cause an error-handling action.

The basic atomic instruction is, of course, the *assignment* of a value to a variable identifier. Syntactically assignment instructions are of the form:

```
identifier := data-expression
```

The denotation of an assignment changes an input state into an output state in the following steps:

1. checking if the input state does not carry an error, and if this is the case, then the input state becomes the output state, and the execution terminates; in the opposite case
2. checking if the identifier has been declared, i.e., if in the input state it is bound to a value or to a pseudo value (see later); if this is not the case then an error message 'identifier-undeclared' is put to the error register; in the opposite case
3. trying to evaluate the data expression; if this attempt generates an error then this error is put to the error register; in the opposite case
4. checking if the value computed from the expression has a body conformant with the body of the identifier's type, and if it is not the case then an error message is put to the error register; in the opposite case
5. checking if the value computed from the expression has a composite that satisfies the yoke of the type of the variable, and if it is not the case then an error message is loaded to the error register; in the opposite case
6. the computed value is assigned to the identifier with the yoke of the variable (not of the value!) remaining unchanged.

The remaining instructions belong to one of the following six categories where the first three are *atomic instructions*, and the remaining three are *structural instructions*, i.e., instructions composed of other instructions and expression:

1. the replacement of a yoke assigned to a variable by another one,
2. the activation of an error-handling procedure,
3. the call of an imperative procedure,
4. the sequence of instructions,
5. the conditional composition of instructions of the form **if-then-else-fi**,
6. the loop **while-do-od**.

Of all these instructions, only procedure calls have to be explained.

When a procedure is called, it gets two lists of *actual parameters*: *value parameters* and *referential parameters*, the values of which are assigned to the corresponding *formal parameters*, which may also be value- and referential. After the execution of a procedure body, the values of formal referential parameters are passed to the corresponding actual referential parameters.

The mechanism of parameter passing is the only communication channel between a procedure body and its hosting program. Inside a procedure body, only local variables are available, and these variables are not available outside the procedure. It is to be emphasized that this decision has not been "forced" by the fact that we are building a denotational model. It has been taken — like many others — for pragmatic reasons that I shall try to justify when it comes to their more technical discussion.

Contrary to variables, all types and procedures declared in the declarative part of a program (see later) have a global character, i.e., they are visible inside all procedure bodies. In a procedure body, one may also declare local variables, procedures, and types that are not available outside the procedure body. It is again a pragmatic (engineering) decision rather than a denotational necessity.

For imperative procedures, there is a mechanism of both a *direct recursion* (a procedure calls itself) and an *indirect recursion* (procedure A calls procedure B, which calls procedure C which calls... procedure A).

The denotations of data- and type expressions, of instructions, of variable-, type-, and procedure declarations, and of programs (see sections that follow) constitute a many-sorted algebra of denotations which is described in Sec. 5 (without procedures) and in Sec. 6 (with procedures). In these sections also the corresponding syntactic algebras are described.

1.5.6 Variable- and type declarations

Variable-declaration denotations are total functions that map states into states:

$$\text{VarDecDen} = \text{State} \mapsto \text{State}$$

assigning types to identifiers and leaving their data undefined. More formally, they assign *pseudo-values*, which are pairs of the form (Ω, typ) , where Ω is an abstract element called a *pseudo-data*. Syntactically a single declaration is of the form:

```
let identifier be type-expression tel
```

Variable declarations are similar to assignments with the difference that in the former case, an error is signaled whenever the identifier is bound in the input state to a pseudo-value, a value, a type, or a procedure. In each such case, the error message ‘*identifier-not-free*’ is generated. It means that a variable may be declared in a program only once. Subsequently, its value may be changed only by changing its composite and possibly the yoke. Bodies may be changed only in the case of database tables and only if we add new attributes or if we remove existing attributes (an engineering decision).

The denotations of type declarations are similar to those of variable declarations with the difference that they assign types rather than pseudo-values to identifiers.

$$\text{TypDecDen} = \text{State} \mapsto \text{State}$$

An identifier that is bound to a type in a state is called a *type constant*. Notice that “constant” rather than “variable” since the type assigned to it, cannot be changed in the future (an engineering decision).

Similarly, as in the case of assignments, also type definitions, and variable declarations may be combined sequentially using a semicolon.

1.5.7 Procedure declarations

Procedures may be imperative or functional. The former are functions that receive two lists of actual parameters — value parameters and reference parameters — and return partial functions on stores¹¹. Functional procedures take only value parameters and return partial functions on stores:

$$\begin{aligned} \text{ipr} &: \text{ImpPro} = \text{AcPaDe} \times \text{AcPaDe} \mapsto \text{Store} \rightarrow \text{Store} \\ \text{fpr} &: \text{FunPro} = \text{AcPaDe} \mapsto \text{Store} \rightarrow (\text{Value} \mid \text{Error}) \end{aligned}$$

In these equations, *AcPaDe* is a domain of *actual-parameter lists*. Notice that we do not talk here about procedure denotations but about procedures as such since they are “purely denotational” concepts. Consequently, they do not have syntactic counterparts. At the level of syntax, we have only *procedure declarations*

¹¹ The fact that procedures transform stores rather than states is a technical trick that allows to avoid a self-application of a function, i.e. a situations where a function takes itself as an argument. More about that problem in Sec. 3.1. Of course, procedure calls are instructions and therefore transform states into states.

and *procedure calls*, which, of course, have their denotations. Syntactically an imperative-procedure declaration is of the form:

```
proc Identifier (val for-val-par ref for-ref-par)
  program
end proc
```

where `program` is a program (see later). Both parameter lists are lists of variable identifiers each followed by a type expression, e.g.

```
(val age, weight as number, name as word
  ref patient as patient-record)
```

Expressions, except single-identifier expressions, are not allowed as value parameters since this would complicate the model and program-construction rules (an engineering decision).

If we want to declare a group of mutually recursive procedures, then we use a *multiprocedure declaration* of the form:

```
begin multiproc
  DekPro-1;
  DekPro-2;
  ...
  DekPro-n
end multiproc
```

Functional procedures are partial functions that transform stores into composites. They also do not have syntactic counterparts, and their declarations are of the form:

```
fun identifier (for-parameters)
  program
return expression as type-expression
```

The call of such a procedure first executes the program and then evaluates the expression in the output state of that program. The value generated by that expression becomes the result of the procedure call. Such a call has no *side-effects*, i.e., it never modifies a state (an engineering decision). In a particular case, the program may be trivial “doing nothing”, and the expression may be reduced to a single identifier.

Procedures discussed above accept as parameters only variable identifiers, i.e., identifiers that bind values. All types and procedures defined in a hosting program of a procedure call are visible in the procedure body as global entities, and therefore they do not need to be passed as parameters (an engineering decision).

In the version of **Lingua** described above, procedures cannot take other procedures as parameters. It is shown in (Sec. 6.6) how to overcome this restriction by constructing a hierarchy of procedures that can take as parameters only procedures of a lower rank than themselves. This decision protects procedures from taking themselves as parameters since this leads to models that are not denotational in our sense¹².

1.5.8 Typological procedures

Denotationally *typological procedures* are similar to functional procedures, but instead of composites return types, and their bodies are type expressions.

Since typological procedures are procedures, they do not appear on the level of syntax, where we only have *typological procedure declarations* and *typological procedure calls*. An example of a declaration may be as follows:

```
type list-of-employees (employee)
  list-of.employee
```

¹² Formally speaking this decision is forced more by set-theoretical argument than by the denotationality of our model (see Sec. 3.1).

end type

Here we declare a typological procedure `list-of-employees` whose single formal parameter is the identifier `employee`, which represents a future type of employee. The keyword **list-of** denotes a language-based constructor of list types. If somewhere later in the program, we declare a type and give it the name `accountant`:

```

set accountant as
  record-type
    ch-name      as string,
    fa-name      as string,
    birth-year   as number,
    award-years as array-of number ee
ee
tes

```

then we may declare a type of lists of accountants by calling our typological procedure with an actual parameter `accountant`:

```

set list-of-accountants as list-of-employees(accountant) ee

```

Here the call of a typological procedure plays the role of a type expression.

In order to include typological procedures in our model, we have to define two constructors:

1. a constructor of typological-procedure declarations,
2. a constructor of typological-procedure calls.

The first of them, given an identifier and typological-procedure components, returns a typological-procedure declaration, which belongs to the domain of declaration denotations:

$$\text{declare-typ-pro} : \text{Identifier} \times \text{TypPar} \times \text{TypExpDen} \mapsto \text{DecDen}$$

or in an “unfolded” form:

$$\text{declare-typ-pro} : \text{Identifier} \times \text{TypPar} \times \text{TypExpDen} \mapsto \text{State} \mapsto \text{State}$$

The denotation of a typological-procedure declaration, given:

1. an identifier `ide`,
2. a list of formal parameters `tpa`,
3. a type-expression denotation `ted`,
4. a type environment `tye`,

first builds a typological procedure out of its components (2.-4.), and then assigns it to `ide` in the environment.

The second constructor of our the algebra of denotations corresponds to a call of a typological procedure:

$$\text{call-typ-pro} : \text{Identifier} \times \text{TypPar} \mapsto \text{TypExpDen} \quad \text{i.e.}$$

$$\text{call-typ-pro} : \text{Identifier} \times \text{TypPar} \mapsto \text{State} \mapsto \text{TypeE}$$

After all necessary checks, the typological procedure assigned to `ide` in the procedure environment is applied to the list of actual parameters and to the (call-time) type environment of the current state.

The calls of typological procedures belong to type-expression denotations. In other words, calling a typological procedure is one of the possible ways of creating a type.

1.5.9 Object programming

Work on a denotational model of object-oriented programs is in progress at the moment. This section will be completed later.

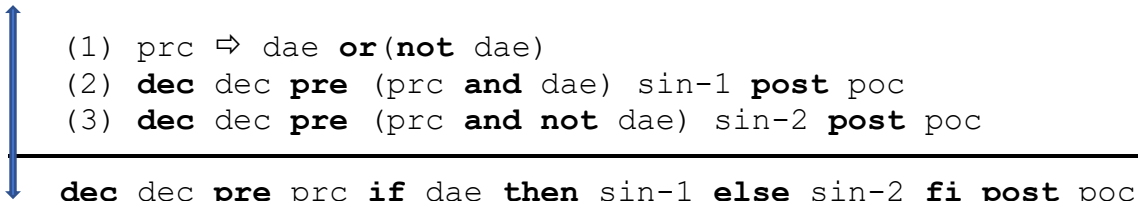

```

asr z > 0 and x ≥ 0
  while z2 ≤ n do z:=2*z od;
  x := 0;
  while z > 1
  do
    z := z/2;
    if (x+z)2 < n then x:=x+z else x:=x fi
  od
rsa
post x = isrt(n) and z = 1                                     (postcondition)

```

The part of the program between **asr** con and **rsa** is called the *range of condition* con. If our metaprogram is correct, then the condition is satisfied by all intermediate states in its range.

Correct-program construction starts from simple programs whose correctness is proved in a traditional way. The following programs are constructed from already existing programs in using construction rules that guarantee the correctness of the resulting program. Here is an example of such a rule that is used to construct a program consisting of a conditional instruction:



```

(1) prc ⇔ dae or(not dae)
(2) dec dec pre (prc and dae) sin-1 post poc
(3) dec dec pre (prc and not dae) sin-2 post poc
-----
dec dec pre prc if dae then sin-1 else sin-2 fi post poc

```

This rule we read as follows:

If

- (1) every state that satisfies the precondition prc satisfies either the data expression dae or its negation (this assumption means that if the precondition is satisfied, then the branching data-expression dae evaluates to a boolean value, and therefore neither loops nor generates an error message),
- (2) metaprogram (2) is correct
- (3) metaprogram (3) is correct

then

the metaprogram below the line is correct.

This rule allows for the construction of a correct metaprogram starting from two correct metaprograms. Observe that in the classical predicate calculus, the metacondition (1) would be a tautology, but in our case — due to the third logical value — it is not.

Die Mathematiker sind eine Art Franzosen: Redet man zu ihnen, so übersetzen sie es in ihre Sprache, und dann ist es alsobald ganz etwas anders.¹³

Johann Wolfgang von Goethe

2 METASOFT AND ITS MATHEMATICS

When in the years 1970 to 1990 I was lecturing mathematical foundations of computer science to IT practitioners, I frequently heard an objection that there is definitely much too much of this mathematics that software engineers have to swallow. Bosses of IT departments expected that their teams could be “trained” in that new mathematics within one weekend and maximally two. Then I was trying to bring to their attention the fact that future engineers attend two to five semesters of mathematics during their university studies. The majority of this mathematics was, however created at the borderline between XIX and XX century and is oriented towards physics, astronomy, and classical engineering rather than informatics.

When at the beginning of the second half of XX century mathematicians started to think about mathematical theories for computer science, some of the branches of mathematics earlier considered as “unpractical” — such as set theory, mathematical logic, or abstract algebras — became their convenient tools. A little later, new branches started to emerge: theory of abstract automata and formal languages, logics of programs, models of concurrent systems, and many others. Today mathematical foundations of computer science are large and still fast-growing new branches of applied mathematics.

Of course, in this section, I do not pretend to present even a sketch of that mathematics. I present only these tools, which I shall use in the books. I am conscious of the fact that for some readers going through Sec.2 may be quite a challenge. However, becoming familiar with **MetaSoft** will allow them to describe complex programming constructions in a relatively simple way and — what is especially important — complete and unambiguous.

2.1 Basic notational conventions of MetaSoft

MetaSoft is a semi-formal (i.e., not fully formalized) mathematical notation oriented towards formal descriptions of programming languages. Since typically formulas that appear in such descriptions oversize everything we know from traditional mathematics, some new notational conventions will be introduced. In particular, when it comes to defining models of programming languages (starting from Sec. 3) instead of using single-letter symbols like **a**, **b**, **c** many-letter symbols are used such as **sta** (for “state”), **den** (for “denotation”), etc. To distinguish **MetaSoft** formulas in the text, they are printed in *Arial* font. At the end of the book, there is a list of most frequently used symbols (at this moment in may be incomplete).

2.1.1 General mathematical notation

Logical operators are given traditional names: **and**, **or**, **not**, **tt**, **ff**. The two last are logical constants “true” and “false”. For quantifiers, I shall use:

\forall — *general quantifier* (for all)

¹³ Mathematicians are [like] a sort of Frenchmen; if you talk to them, they translate it into their own language, and then it is immediately something quite different.

— Johann Wolfgang von Goethe, *Maximen und Reflexionen*, 2006 (Helmut Koopmann, ed.)

\exists — *existential quantifier* (there exists)

Instead of $i = 1, \dots, n$ I frequently write $i = 1;n$. By “iff” I mean “if and only if”.

2.1.2 Sets

Symbol $\{ \}$ denotes the empty set and

$$\{a_1, \dots, a_n\} \text{ or } \{a_i \mid i = 1;n\}$$

denotes a finite n -element set. The fact that a is (or is not) an element of A shall be written as

$$a : A \quad (a \notin A)$$

and the inclusion of sets shall be written as

$$A \subseteq B$$

By

$$A \cup B \text{ and } A \cap B$$

we denote the union and the intersection of sets A and B . If Fam is a family of sets then

$$\bigcup \text{Fam}$$

denotes the union of that family. By

$$A \times B$$

I denote the Cartesian product of sets. The expression:

$$A \times B \times C \times D$$

denotes the set of tuples of the form (a, b, c, d) The expression:

$$A \times (B \times C) \times D$$

denotes the set of tuples of the form $(a, (b, c), d)$, and analogously for other combinations of parentheses. For every $n \geq 0$ the n -th Cartesian power A^{cn} of a set A is the set of all tuples of the elements of A , i.e.:

$$A^{c0} = \{()\} \quad \text{— the only element of that set is an empty tuple}$$

$$A^{cn} = \{(a_1, \dots, a_n) \mid a_i : A\} \quad \text{— for } n > 0$$

Using Cartesian power, we define two other operations:

$$A^{c+} = \bigcup \{A^{cn} \mid n > 0\}$$

$$A^{c*} = A^{c0} \cup A^{c+}$$

The set of all subsets of A and respectively of all finite subsets of A is denoted by

$$\text{Sub}.A$$

$$\text{FinSub}.A$$

The following notations shall be used for sets of relations and functions:

$\text{Rel.}(A, B)$ — the set of all binary relations between A and B ; i.e., the set of all subsets of $A \times B$; more about binary relations in Sec.2.6,

$A \rightarrow B$ — the set of all *partial functions* from A to B , i.e., functions that do not need to be defined for all elements of A ,

$A \mapsto B$ — the set of all *total functions* from A to B , i.e., functions that are defined for all elements of A ; notice that each total function is a partial function but not vice-versa,

$A \Rightarrow B$ — the set of all *mappings* from A to B , i.e., functions defined for only a finite subset of A .

Following this notation by

$$f : A \mapsto B$$

we mean that f is an element of the set $A \mapsto B$, i.e. is a total function from A to B and analogously for other operators creating sets of functions. This rule also explains why the traditional $a \in A$ is written as $a : A$.

2.1.3 Functions

For practical reasons, the value of a function shall be written as $f.a$ rather than $f(a)$. Why this is practical will be seen a little later. The expression

$$f.a = ? \tag{2.1-1}$$

means that f is not defined for a . It does not mean that “?” is anything like an “undefined element”. The expression $f.a = ?$ stands for

$$\mathbf{not} (\exists b)(f.a=b)$$

Analogously

$$f.a = !$$

stands for $(\exists b)(f.a=b)$. For an arbitrary function

$$f : A \rightarrow B$$

and an arbitrary set C by the *truncation of f to C* we shall mean:

$$f \text{ trun } C = \{(a, f.a) \mid a : A \cap C\}.$$

The *domain* of f is the set where f is defined, i.e.

$$\text{dom}.f = \{a \mid a : A \mathbf{and} f.a = !\}$$

In the sequel we shall also use the notation

$$f [a/?] = f \text{ trun } (\text{dom}.f - \{a\})$$

Another notation that will be used frequently comes from Haskell Curry and concerns many-argument functions whose arguments are taken successively one after another, e.g.

$$f : A \rightarrow (B \rightarrow (C \rightarrow (D \rightarrow E))) \tag{2.1-2}$$

The value of such a function should be formally written as

$$((((f.a).b).c).d)$$

but Curry writes

$$f.a.b.c.d$$

which intuitively means that

- function f takes a as an argument and returns as value a function $f.a$ that belongs to the set $B \rightarrow (C \rightarrow (D \rightarrow E))$ and next
- function $f.a$ takes as an argument an element b and returns as a value function $f.a.b$ that belongs to the set $C \rightarrow (D \rightarrow E)$, etc.

This notation allows not only to avoid many parentheses but also to define function of “mixed” types like e.g.

$$f : A \rightarrow (B \mapsto (C \rightarrow (D \Rightarrow E))) \quad \text{or} \tag{2.1-3}$$

$$f : (A \rightarrow B) \mapsto (C \rightarrow (D \Rightarrow E))$$

Another simplifying convention allows to write

$$f : A \rightarrow B \mapsto C \rightarrow D \Rightarrow E \quad (2.1-4)$$

instead of

$$f : A \rightarrow (B \mapsto (C \rightarrow (D \Rightarrow E))) \quad (2.1-5)$$

The expression

$$f : \mapsto A \quad (2.1-6)$$

means that f is a zero-argument function with only one value that belongs to A . That value is denoted by

$$f.()$$

About formulas from (2.1-2) to (2.1-6) we say that they describe *types* or *signatures* of corresponding functions. For instance we say that the function in (2.1-5) *is of the type*

$$A \rightarrow B \mapsto C \rightarrow D \Rightarrow E$$

For every (possibly partial) function

$$f : A \rightarrow A,$$

by its n -th iteration where $n = 0, 1, 2, \dots$ we shall mean the function

$$f^n : A \rightarrow A$$

defined in the following way:

f^0 is an *identity function* on A , i.e. $f.a = a$ for every $a : A$,

$f^n.a = f.(f^{n-1}.a)$ for $n > 0$.

In mathematical definitions of programming languages, we shall frequently use many-level *conditional definitions of functions* with the following scheme:

$$\begin{aligned} f.x = & \\ & p_1.x \rightarrow g_1.x \\ & p_2.x \rightarrow g_2.x \\ & \dots \\ & p_n.x \rightarrow g_n.x \end{aligned} \quad (2.1-7)$$

where each p_i is a classical predicate, i.e., a total function with logical values **tt** or **ff**, and each g_i is just a function. The formula (2.1-7) is read as follows:

if $p_1.x$ is true, then $f.x = g_1.x$ and otherwise,

if $p_2.x$ is true, then $f.x = g_2.x$ and otherwise,

...

Intuitively speaking, the evaluation of this function goes line by line and stops at the first line where $p_i.x$ is satisfied. Of course, to make such a definition of function f unambiguous, the alternative of all predicates $p_i.x$ must evaluate to “true”, which means that all these predicates must exhaust all cases. To ensure that condition at the last line, we frequently write **true**, which stands for the predicate, which is always true. It can also be read as “in all other cases”.

In the scheme (2.1-7) we also allow the situation where, in the place of a $g_i.x$ we have the undefinedness sign “?” which means that for x that satisfies $p_i.x$, the function f is undefined. This convention allows for conditional definitions of partial functions.

In conditional definitions we also use a technique similar to defining local constants in programs. For instance if $f : A \times B \mapsto C$ we can write

$$\begin{aligned} f.x = & \\ & p_1.x \rightarrow g_1.x \end{aligned}$$


```

let
  (a, b) = x
p2.a → g2.x
p3.b → g3.x
...

```

which is read as: *let x be a pair of the form (a, b)*. We can also use **let** in the following way:

```

f.x =
  p1.x → g1.x
let
  y = h.x
p2.x → g2.y
p3.x → g3.y,
...

```

All these explanations are certainly not very formal, but the notation should be clear when it comes to applications in the sequel of the book. A finite total function $f : \{a_1, \dots, a_n\} \mapsto \{b_1, \dots, b_n\}$ defined by the formula:

```

f.x =
  x=a1 → b1
  x=a2 → b2
  ...
  x=an → bn

```

shall be written as

$[a_1/b_1, \dots, a_n/b_n]$ or alternatively as $[a_i/b_i \mid i = 1; n]$.

The empty function will be denoted by $[\]$. Let $f : A \rightarrow B$ and $g : C \rightarrow D$. The *overwriting of f by g* is a function denoted by

$f \blacklozenge g : A|C \rightarrow B|D$

and defined in the following way:

```

(f  $\blacklozenge$  g).x =
  g.x = ! → g.x
  g.x = ? → f.x

```

In particular this means that if $f.x=?$ and $g.x=?$, then $f \blacklozenge g.x=?$. A particular case of overwriting is an *update of a function* written as $f[a_1/b_1, \dots, a_n/b_n]$ and defined by the formula

```

(f[a1/b1, ..., an/bn]).x =
  x = a1 → b1
  ...
  x = an → bn
true → f.x

```

2.1.4 Tuples

An expression

(a_1, \dots, a_n) or alternatively $(a_i \mid i=1;n)$

denotes *n-tuple*. Consequently $()$ denotes an empty tuple. The difference between tuples and finite sets is such that the order of elements in a tuple is relevant and repetitions are allowed, which is not the case for sets. E.g.

$\{a, b, c, c\} = \{a, c, b\} = \{a, b, c\} = \dots$

$(a, b, c, c) \neq (a, c, c, b) \neq (a, b, c)$

where a, b and c are different with each other.

Tuples are used as mathematical models for several concepts in theoretical computer science and among others for pushdowns. In this area the following functions shall be used later on in the book:

$$\begin{aligned} \text{push.}(b, (a_1, \dots, a_n)) &= (b, a_1, \dots, a_n) \quad \text{for } n \geq 0 \\ \text{pop.}(a_1, \dots, a_n) &= (a_2, \dots, a_n) \quad \text{for } n \geq 2 \\ \text{pop.}(a) &= () \\ \text{pop.}() &= () \\ \text{top.}(a_1, \dots, a_n) &= a_1 \quad \text{for } n \geq 1 \\ \text{top.}() &= ? \end{aligned}$$

An important operation on tuples is a *Cartesian concatenation of tuples*¹⁴:

$$(a_1, \dots, a_n) \mathbb{C} (b_1, \dots, b_m) = (a_1, \dots, a_n, b_1, \dots, b_m).$$

We shall also use two predicates:

$$\text{are-repetitions.}(a_1, \dots, a_n) = \text{tt} \quad \text{iff there exist } i \neq j \text{ such that } a_i = a_j$$

$$\text{no-repetitions.}(a_1, \dots, a_n) = \text{tt} \quad \text{iff there are no } i \neq j \text{ such that } a_i = a_j$$

Tuples may also be regarded as functions from natural numbers into their elements i.e.

$$(a_1, \dots, a_n).i = a_i$$

Let now for a certain set A

$$\text{Tuple} = A^{c^*}$$

be the set of all tuples over A. For sets of tuples the following functions shall be used:

remove-repetitions : Tuple \mapsto Tuple

remove-repetitions.(a-1, ..., a-n) =

$$n = 0 \quad \rightarrow ()$$

$$n = 1 \quad \rightarrow (a_1)$$

$$a_1 : \{a_2, \dots, a_n\} \quad \rightarrow \text{remove-repetitions.}(a_2, \dots, a_n)$$

$$\text{true} \quad \rightarrow (a_1) \mathbb{C} \text{remove-repetitions.}(a_2, \dots, a_n)$$

join-without-repetition : Tuple x Tuple \mapsto Tuple

$$\text{join-without-repetition.}(tup_1, tup_2) = \text{remove-repetitions.}(tup_1 \mathbb{C} tup_2)$$

common-part : Tuple x Tuple \mapsto Tuple

common-part.((a₁, ..., a_n), (b₁, ..., b_m)) =

$$n = 0 \quad \rightarrow ()$$

$$m = 0 \quad \rightarrow ()$$

$$a_1 : \{b_1, \dots, b_m\} \quad \rightarrow (a_1) \mathbb{C} \text{common-part.}((a_2, \dots, a_n), (b_1, \dots, b_m))$$

$$\text{true} \quad \rightarrow \text{common-part.}((a_2, \dots, a_n), (b_1, \dots, b_m))$$

difference : Tuple x Tuple \mapsto Tuple

difference. ((a₁, ..., a_n), (b₁, ..., b_m)) =

$$n = 0 \quad \rightarrow ()$$

¹⁴ This should be not confused with a language-theoretic concatenation of words (see Sec. 2.4).

$$\begin{array}{ll}
m = 0 & \rightarrow (a_1, \dots, a_n) \\
a_1 : \{b_1, \dots, b_m\} & \rightarrow \text{difference.}((a_2, \dots, a_n), (b_1, \dots, b_m)) \\
\text{true} & \rightarrow (a_1) \text{ } \mathbb{C} \text{ difference.}((a_2, \dots, a_n), (b_1, \dots, b_m))
\end{array}$$

The last operation selects these elements of a tuple that satisfy a given predicate. Let then

$$p : A \mapsto \{\text{tt}, \text{ff}, \text{ee}\}$$

be a three-valued predicate. With every such predicate, we associate a *filtering function* that removes from a tuple all elements a that do not satisfy p , i.e., such that $p.a : \{\text{ff}, \text{ee}\}$.

$$\text{filter.p} : \text{Tuple} \mapsto \text{Tuple}$$

$$\begin{array}{ll}
\text{filter.p.}(a_1, \dots, a_n) = & \\
n = 0 & \rightarrow () \\
p.a_1 = \text{tt} & \rightarrow (a_1) \text{ } \mathbb{C} \text{ filter.p.}(a_2, \dots, a_n) \\
\text{true} & \rightarrow \text{filter.p.}(a_2, \dots, a_n)
\end{array}$$

2.2 Partially ordered sets

Let A be an arbitrary set and let

$$\sqsubseteq : \text{Rel}(A, A)$$

be a binary relation in that set. Relation \sqsubseteq is said to be a *partial order* in A if for any $a, b, c : A$ the following conditions are satisfied:

1. $a \sqsubseteq a$ *reflexivity*
2. if $a \sqsubseteq b$ and $b \sqsubseteq c$ then $a \sqsubseteq c$ *transitivity*
3. if $a \sqsubseteq b$ and $b \sqsubseteq a$ then $a = b$ *weak antisymmetry*

If only 1. and 2. are satisfied then \sqsubseteq is said to be *quasiorder*. In the sequel we shall deal most frequently with partial orders.

If $a \sqsubseteq b$, then we say that a is *smaller* than b or that b is *greater* than a . If additionally $a \neq b$, then we say that a is *significantly smaller* than b or that b is *significantly greater* than a .

A pair (A, \sqsubseteq) is called a *partially ordered set* (abbr. POS), and the set A is called its *carrier*. The word “partial” means that not any two elements of A are comparable with each other. If however,

for any a and b either $a \sqsubseteq b$ or $b \sqsubseteq a$,

then we say that this is a *total order*.

Of course, every linear order is partial, and every partial order is quasiorder but not vice versa. An example of a partial order which is not total is the inclusion of sets. Such POS is called *set-theoretic POS*.

Let B be a subset of a partially ordered A and let $b : B$. In that case

- b is called a *minimal element* in B , if there is no $a : B$ such that $a \sqsubseteq b$ and $a \neq b$
- b is called the *least element* in B , if for any $a : B$ holds $b \sqsubseteq a$,
- b is called a *maximal element* in B , if there is no $a : B$ such that $b \sqsubseteq a$ and $a \neq b$,
- b is called the *greatest element* in B , if for any $a : B$ holds $a \sqsubseteq b$.

There exist partially ordered sets without a minimal element and sets where there is more than one such element. However, if there is the least element in a set, then it is the unique minimal element and analogously for maximal and greatest elements.

An *upper bound* of \mathbf{B} is such an element of \mathbf{A} , which is greater than any element of \mathbf{B} . Notice that an upper bound of a set does not need to belong to that set, but if it does belong, then it is the greatest element of the set.

If the set of all upper bounds of \mathbf{B} has the least element, then this element is called the *least upper bound* of \mathbf{B} ¹⁵. If a two-element set $\{a, b\}$ has the least upper bound, then we denote it by

$$a \vee b$$

In a set-theoretic POS, the least upper bound of a family of sets is the set-theoretic union of that family. This, of course, also concerns a family of two sets.

2.3 Chain-complete partially-ordered sets

Let (A, \sqsubseteq) be a partially ordered set. By a *chain* in that set we mean any sequence of elements of A :

$$a_1, a_2, a_3, \dots$$

such that $a_i \sqsubseteq a_{i+1}$. If the set of all elements of a chain has the least upper bound, then it is called the *limit* of that chain and is denoted by:

$$\lim(a_i \mid i = 1, 2, \dots)$$

A POS is said to be *chain-complete partially ordered set* (abbr. CPO) if:

1. every chain in A has a limit,
2. there exists the least element in A .

This least element we shall denote by Φ .

A total function $f : A \mapsto A$ is said to be *monotone* if $a \sqsubseteq b$ implies $f.a \sqsubseteq f.b$ and we say that it is *continuous* if the following two conditions are satisfied:

1. for any chain $(a_i \mid i = 1, 2, \dots)$ the sequence $(f.a_i \mid i = 1, 2, \dots)$ is also a chain,
2. if the former has a limit, then the latter has a limit as well and

$$\lim(f.a_i \mid i = 1, 2, \dots) = f.[\lim(a_i \mid i = 1, 2, \dots)].$$

As is easy to see, every continuous function is monotone, which follows from the fact that

$$\text{if } a \sqsubseteq b \text{ then } \lim(a, b, b, b, \dots) = b.$$

Continuous functions satisfy a theorem — due to S.C. Kleene [57] — which we shall frequently use in our applications.

Theorem 2.3-1 *If f is continuous in a chain complete set, then the set of all solutions of the equation*

$$x = f.x \tag{2.3-1}$$

is not empty and contains the least element defined by the equation

$$Y.f = \lim(f^n.\Phi \mid n = 0, 1, 2, \dots) \blacksquare$$

Proof of that theorem is very simple:

$$f.(Y.f) = f.(\lim(f^n.\Phi \mid n = 0, 1, 2, \dots)) = \lim(f^n.\Phi \mid n = 1, 2, \dots) = \lim(f^n.\Phi \mid n = 0, 1, 2, \dots).$$

The last equality follows from the fact that $f^0.\Phi = \Phi$, hence adding $f^0.\Phi$ to the chain, does not change its limit.

■

The equation (2.3-1) is called a *fixed point equation* and its solution $Y.f$ — the *least fixed point of function* f . It is the least solution of the equation (2.3-1), but in the sequel I will call it simply *the solution* since other solutions will not be concerned.

¹⁵ The greatest lower bound is defined in an analogous way but that concept will not be used in the book.

The concept of a one-argument continuous function may be simply generalised to functions of many arguments. We say that

$$f : A^{cn} \mapsto A \quad (2.3-2)$$

is *continuous wrt*¹⁶ *to its first element*, if for any tuple (a_1, \dots, a_{n-1}) the function

$$g.a = f.(a, a_1, \dots, a_{n-1})$$

is continuous. In an analogous way we define the continuity of f with regard to any other of its arguments.

A many-argument function (2.3-2) is called *continuous* if it is continuous in all of its arguments.

As we are going to see soon, continuous functions are fundamental for our applications since due to Kleene's theorem we can recursively define sets and functions. Such definitions will most frequently have the form

$$x_1 = f_1.(x_1, \dots, x_n)$$

...

$$x_n = f_n.(x_1, \dots, x_n)$$

Of course, every such set of equations may be regarded as one equation

$$X = f.X$$

in a POS over a Cartesian product $A_1 \times \dots \times A_n$ where

$$f.(x_1, \dots, x_n) = (f_1.(x_1, \dots, x_n), \dots, f_n.(x_1, \dots, x_n))$$

and where the order is define component-wise, i.e.

$$(a_1, \dots, a_n) \sqsubseteq^n (b_1, \dots, b_n) \text{ iff } a_i \sqsubseteq b_i \text{ for } i = 1, \dots, n.$$

As is easy to show, if all A_i are chain-complete, then their Cartesian product is chain-complete wrt the above order. Besides, if each f_i is continuous, then f is continuous, as well.

As turns out, fixed-point sets of equations with continuous functions may be transformed (and reduced) in a way analogous to the case of algebraic equations. It is expressed by two theorems due to Hans Bekić [11] and Jacek Leszczyłowski [60].

Theorem 2.3-2 *If $f, g : A \times A \mapsto A$ are continuous, then the set of equations*

$$a = f.(a, b)$$

$$b = g.(a, b)$$

is equivalent to

$$a = f.(a, b)$$

$$b = g.(f.(a, b), b) \quad \blacksquare$$

Theorem 2.3-3 *If $f, g : A \times A \mapsto A$ are continuous, then the set of equations*

$$a = f.(a, b)$$

$$b = g.(a, b)$$

is equivalent to

$$a = h.b$$

$$b = g.(a, b)$$

¹⁶ This abbreviation means "with respect to".

where h is a function that to every b assigns the least fixed point of $f.(x, b)$ regarded as a one-argument function of x running over the set A . ■

As we are going to see, the theory of fixed-point equations in CPO is an important tool for writing recursive definitions of sets and of functions in denotational models. More investigations about continuous functions may be found in [24].

2.4 A CPO of formal languages

Grammars of *natural languages* such as English, Polish, French, etc. may be regarded as algorithms allowing to check which sentences are grammatically correct and which are not. In this spirit, Noam Chomsky has developed in early 1960. his model of *generative context-free grammars* or simply *context-free grammars* (see [37], [39], [40], [41]). Formal languages generable by such grammars have been called *context-free languages*.

Although this model turned out to be too simple for natural languages, it was successfully applied for programming languages. In the early years for Algol 60 and Pascal, later for ADA and CHILL and many others. This contributed to the rapid development of their theory. The first internationally recognized monography on that subject was written in 1966 by Seymour Ginsburg [49], and the first Polish monography in 1971 by myself [16]. A year later, I have published a paper on *equational grammars* [18], which are equivalent, in a sense, to context-free grammars.

This section contains a short introduction to context-free languages in the context of equational grammars.

Let A be an arbitrary finite set of symbols called an *alphabet*. By a *word* over A , we mean every finite string over A , including the empty string. Traditionally words are written as sequences of characters, e.g., $accbda$, and the *empty word* is denoted by ϵ .

If x and y are words, then by their *language-theoretic concatenations* or just a *concatenation* — which we denote by $x \odot y$ or simply by xy — we mean a sequential combination of these words. E.g.

$$abdaa \odot eaag = abdaaeaag$$

The function \odot is called *concatenation*, as well.

Note the difference between the Cartesian concatenation of tuples \oslash introduced in Sec. 2.1.4, and our concatenation of words. For instance

$$abdaa \oslash eaag = (abdaa, eaag)$$

where $abdaa$ and $eaag$ are regarded as one-element tuples. Note also that

$$\text{tup-1} \oslash \text{tup-2}$$

makes sense for tuples of arbitrary elements, e.g., of functions, whereas \odot is applicable only to words which are strings of characters.

Every set L of words over A is called a *formal language* (or simply a *language*) over A . By $\text{Lan}(A)$ we denote the family of all languages over A and \emptyset — the empty language (empty set). If P and Q are languages, then their *concatenation* is the language defined by the equation:

$$P \odot Q = \{p \odot q \mid p:P \text{ and } q:Q\}.$$

As we see by \odot we denote not only a function on words but also on languages. If it does not lead to ambiguities, $P \odot Q$ is written as PQ . Since concatenation is an associative operation, we can write PQL instead of $(PQ)L$ or $P(QL)$. I shall also assume that concatenation binds stronger than set-theoretic union, hence instead of

$$(P \odot Q) \mid (R \odot S)$$

I shall write

$$PQ \mid RS$$

It is also easy to see that concatenation is distributive over the union, i.e.

$$(P \mid Q) R = PR \mid QR.$$

The n -th *power of a language* P is defined recursively:

$$P^0 = \{ \varepsilon \}$$

$$P^n = P \odot P^{n-1} \text{ for } n > 0$$

We shall also use two operators called respectively *plus* and *star*:

$$P^+ = U.\{P^i \mid i > 0\}$$

$$P^* = P^+ \mid P^0$$

Hence for an alphabet A , the set A^+ is the set of all non-empty words over A , and A^* is the set of all words over A . Languages over A are subsets of A^* .

The inclusion of sets is, of course, a partial order in $\text{Lan}(A)$ and $(\text{Lan}(A), \subseteq)$ is a CPO with empty language as the least element. As is easy to show, all operations on languages, which are defined above, plus the union of languages, are continuous. For any two languages, P and Q , their least upper bound is their union $P \mid Q$, and the limit of a chain of languages is the union of all elements of the chain.

It should be emphasized that the Cartesian power of sets introduced in Sec. 2.1.2 is different from the power of languages. Notice that if P and Q are languages, then:

$$P \odot Q = \{ p \odot q \mid p : P \text{ and } q : Q \}$$

$$P \times Q = \{ (p, q) \mid p : P \text{ and } q : Q \}$$

The concatenation of languages is hence still a language, whereas the Cartesian product is not.

2.5 Equational grammars

Since all the operations on languages defined in Sec. 2.4 are continuous, they can be used in fixed-point equations (Sec. 2.3) regarded as grammars. This idea is elaborated below.

Consider a simple example of a set of equations that defines the set of identifiers of a programming language. We assume that identifiers always start from a letter:

$$\text{Letter} = \{a, b, \dots, z\}$$

$$\text{Digit} = \{0, 1, \dots, 9\}$$

$$\text{Character} = \text{Letter} \mid \text{Digit}$$

$$\text{Suffix} = \{ \varepsilon \} \mid \text{Character} \odot \text{Suffix}$$

$$\text{Identifier} = \text{Letter} \odot \text{Suffix}$$

Such sets of equations are called *equational grammars*, and their solutions (tuples of languages) are called *many-sorted languages*. In the above case, the defined many-sorted language is a tuple of five categories (sorts):

$$(\text{Letter}, \text{Digit}, \text{Character}, \text{Suffix}, \text{Identifier}).$$

The category *Suffix* has an auxiliary character since its only role is to express the fact that an identifier must start with a letter. Its equation can be eliminated in using the Theorem 2.3-2 and the Theorem 2.3-3. As is easy to prove

$$\text{Suffix} = \text{Character}^*$$

hence our grammar may be reduced to a more compact form

$$\text{Letter} = \{a, b, \dots, z\}$$

$$\text{Digit} = \{0, 1, \dots, 9\}$$

$$\text{Identifier} = \text{Letter} \odot (\text{Letter} \mid \text{Digit})^*$$

This grammar defines a many-sorted language, which consists of three categories — and therefore is different from the former — but defines the same set *Identifier*.

Let us now investigate equational grammars more formally (for details see [18]). Let A be an arbitrary non-empty finite alphabet and let

$$\text{Fam} \subseteq \text{Lan}(A)$$

be an arbitrary family of languages over A . Let $\text{Pol}(\text{Fam})$ denotes the least class of functions of the type:

$$p : \text{Lan}(A)^n \mapsto \text{Lan}(A) \text{ where } n \geq 0$$

which contains:

- (1) all projections, i.e. functions of the form $f.(X_1, \dots, X_n) = X_i$ for $i \leq n$,
- (2) all functions with constant values in Fam ,
- (3) the union and concatenation of languages

and is closed over the composition (superposition) of functions.

Functions in $\text{Pol}(\text{Fam})$ are called *polynomials over Fam*. Since all functions described in (1), (2) and (3) are continuous and composition preserves continuity, all polynomials are continuous.

By an *atomic language* over A we shall mean any one-element language $\{w\}$, where $w : A^*$. Polynomials over an arbitrary set of atomic languages are called *Chomsky's polynomials*¹⁷. Below a few examples of such polynomials:

$$p_1.(X, Y, Z) = \{b\}$$

$$p_2.(X, Y) = \{b\}$$

$$p_3.(X, Y, Z) = X$$

$$p_4.(X, Y, Z) = (\{d\}X\{b\}YY\{c\} \mid X) Z$$

Observe that for a complete identification of a polynomial we have to define its arity. This can be seen on the example of p_1 and p_2 .

Polynomials which do not “contain” union — e.g., such as p_1 , p_2 , and p_3 — are called *monomials*. Since concatenation is distributive over the union, every polynomial may be reduced to a union of monomials.

An *equational grammar* over an alphabet A is any fixed-point set of equations of the form:

$$X_1 = p_1.(X_1, \dots, X_n)$$

...

$$X_n = p_n.(X_1, \dots, X_n)$$

where all p_i are Chomsky's polynomials over A . Since polynomials are continuous, this set of equations has a unique least solution (L_1, \dots, L_n) . The languages L_1, \dots, L_n are said to be *defined by our grammar*. We also say that they are *equationally definable*.

As has been proved in [18], the class of equationally-definable languages is identical with the class of *context-free languages* in the sense of Chomsky¹⁸. Such a class remains the same if we allow the operations “*” and “+” in polynomials and if polynomials are built over arbitrary equationally-definable languages. For proofs of all these facts, see [18].

Due to these facts in the sequel of the book, equationally-definable languages will be called *context-free*.

¹⁷ Noam Chomsky — an American linguist, philosopher and political activist. Professor of linguistics at Massachusetts Institute of Technology, co-creator of the concept of transformational-generative grammars. Chomsky did not introduce the idea of Chomsky's polynomials but his grammars are very close to them.

¹⁸ Which means that for each equational grammar there exists an equivalent context-free grammar and vice versa.

2.6 A CPO of binary relations

Let A and B be arbitrary sets. Any subset of their Cartesian product $A \times B$ will be called a *binary relation* or just a *relation* between these sets. Hence

$$\text{Rel}(A,B) = \{R \mid R \subseteq A \times B\}$$

is the set of all binary relations between A and B . Instead of writing $(a,b) : R$, I shall usually write $a R b$.

If $A = B$, then instead of $\text{Rel}(A, A)$ we write $\text{Rel}(A)$. For every A we define an *identity relation*:

$$[A] = \{(a, a) \mid a:A\}$$

By \emptyset , we shall denote the *empty relation*¹⁹. Let now

$$\text{Boolean} = \{\text{tt}, \text{ff}\} \quad \text{— logical values}$$

$$p : A \rightarrow \text{Boolean} \quad \text{— a predicate}$$

With every predicate, we assign an identity relation defined by

$$\text{Id}(p) = \{a \mid p.a = \text{tt}\}$$

If $R : \text{Rel}(A,B)$, then

$$\text{dom}.R = \{a \mid (\exists b:B) a R b\} \quad \text{— the domain of } R$$

$$\text{cod}.R = \{b \mid (\exists a:A) a R b\} \quad \text{— the codomain of } R$$

Let $P : \text{Rel}(A,B)$ and $R : \text{Rel}(B,C)$. *Sequential composition* of P and R we call a relation

$$P \bullet R : \text{Rel}(A,C)$$

defined as follows:

$$P \bullet R = \{(a, c) \mid (\exists b:B) (a P b \ \& \ b R c)\}$$

For every two relations, their composition always exists, although it may be an empty relation. As is easy to check \bullet is associative i.e.

$$(P \bullet R) \bullet Q = P \bullet (R \bullet Q)$$

It is, therefore, legal to write $P \bullet R \bullet Q$. We shall also write PR instead of $P \bullet R$ whenever this does not lead to misunderstanding, and we shall assume that composition binds stronger than union, hence instead of

$$(P \bullet R) \mid (Q \bullet S)$$

we write

$$PR \mid QS.$$

In the sequel of the book, the sequential composition of relations will be most frequently applied in the particular case where the composed relations are function. In that case:

$$(P \bullet R).a = R.(P.a)$$

and therefore

$$(P \bullet R \bullet Q).a = (P \bullet (R \bullet Q)).a = Q.(R.(P.a))$$

which means that in a sequential composition of functions, the composed functions are “executed” from left to right one after another.

Similarly as for languages also for relations, the operations of iterations, i.e. of power and star are defined. In this case:

$$R^0 = [A] \quad \text{— identity relation in over } A$$

¹⁹ The same symbol was used for an empty set which is not an inconsistency since each relation is a set.

$$R^n = RR^{n-1} \text{ for } n > 0$$

$$R^+ = \bigcup \{R^n \mid n > 0\}$$

$$R^* = R^+ \mid R^0$$

The *converse relation* for R is defined as follows

$$a R^{-1} b \quad \text{iff} \quad b R a$$

A relation R is called a *function*, if \supseteq

for any a, b and c , if $a R b$ and $a R c$, then $b = c$.

If R and R^{-1} are functions, then R is said to be a *convertible function* or a *one-one function*. If P and R are functions, then PR is also a function and

$$(PR).a = P.(R.a)$$

hence the composition of functions is their superposition.

The set of relations $\text{Rel}(A,B)$ constitutes a CPO with ordering by set-theoretic inclusion and the empty relation as the least element. All of the defined operations on relations are continuous. The future we shall frequently refer to the following theorem:

Theorem 2.6-1 For any $P, Q : \text{Rel}(A)$ the least solutions of equations

$$X = P \mid QX \quad \text{and}$$

$$X = P \mid XQ$$

are respectively

$$X = Q^*P \quad \text{and}$$

$$X = P^*Q$$

Moreover, if both P and R are functions with disjoint domains, then both these solutions are also functions.

■

In this place, it is worth noticing that the set of partial functions

$$A \rightarrow B$$

constitutes a chain-complete subset of $(\text{Rel}(A,B), \subseteq)$ that is closed under the composition of arbitrary functions and union of functions with disjoint domains. Of course, both these operations are continuous.

Due to these facts, functions can be defined by fixed-point (recursive) equations. Since A and B are arbitrary, this is also true for functions of type

$$f : A_1 \rightarrow A_2 \rightarrow \dots \rightarrow A_n$$

provided that appropriate constructors are defined. As a first example, consider a recursive definition of a function of an n -th power of number 2, i.e.²⁰.

$$\text{power-of-two} : \text{Number} \rightarrow \text{Number} \quad \text{where } \text{Number} = \{0, 1, 2, \dots\}$$

$$\text{power-of-two}.n = 2^n \quad \text{for an integer } n \geq 0$$

A recursive definition of that function is as follows:

$$\begin{aligned} \text{power-of-two}.n = \\ n = 0 \quad \rightarrow 1 \end{aligned}$$

²⁰ Here I introduce a notational convention of VDM and MetaSoft where instead of using one-character symbols as in usual mathematics, I use many-character symbols for both sets and functions. As we are going to see later, this convention is practically a must in the case of denotational models where numbers of symbols goes into tens or hundreds.

$$n > 0 \rightarrow \text{power-of-two}.(n-1) * 2$$

This definition written as a fixed-point equation in the set-theoretic CPO

$$(\text{Number} \rightarrow \text{Number}, \subseteq, [\])$$

is as follows

$$\text{power-of-two} = \text{zero} \blacklozenge (\text{minus} \bullet \text{power-of-two}) \bullet \text{double}$$

where

$$\begin{aligned} \text{zero}.n &= [0/1] \\ \text{minus}.n &= n-1 \quad \text{for } n > 0 \\ \text{minus}.0 &= ? \\ \text{double}.n &= 2 * n \end{aligned}$$

Notice that all these functions are constants in our equation, hence the right-hand side of that equation represents a one-argument function in our CPO:

$$F.\text{fun} = \text{zero} \blacklozenge (\text{minus} \bullet \text{fun}) \bullet \text{double}$$

Since, as is easy to prove, \blacklozenge and \bullet are continuous on both arguments, our function F is continuous as well, and therefore — according to Kleene's theorem — the least solution of our equation is the limit (the union) of the following chain of functions:

$$\begin{aligned} F.\{ \} &= \text{zero} &&= [0/1] \\ F.\text{zero} &= \text{zero} \blacklozenge (\text{minus} \bullet \text{zero}) \bullet \text{double} &&= [0/1, 1/2] \\ F.(F.\text{zero}) &= \text{zero} \blacklozenge (\text{minus} \bullet F.\text{zero}) \bullet \text{double} &&= [0/1, 1/2, 2/4] \\ &\dots && \end{aligned}$$

Each element of that chain is a finite approximation of our function **power-of-two**.

Now let us consider a technically more complicated example of a two-argument function of power in the set of natural numbers:

$$\begin{aligned} \text{power} : \text{Number} \times \text{Number} &\rightarrow \text{Number} \\ \text{power}.n.m &= \\ m = 0 &\rightarrow 1 \\ m > 0 &\rightarrow n * \text{power}.n.(m-1) \end{aligned}$$

This definition can be expressed as a fixed-point equation in the CPO of binary relations:

$$\text{Rel}(\text{Number} \times \text{Number}, \text{Number})$$

To see that, let us construct a fixed-point equation whose solution is the function:

$$\text{power}.(n, m) = n^m$$

regarded as a relation in our CPO. Let us start from the definitions of a certain operation of composition of functions

$$F, Q : \text{Rel}(\text{A} \times \text{A}, \text{A}) \tag{2.6-1}$$

By the *composition of F and Q on the second argument*, we shall mean the relation

$$F \blacklozenge Q = \{((a, b), c) \mid (\exists d) ((a, b), d) : F \text{ and } ((a, d), c) : Q\}$$

If F and Q are functions then

$$[F \blacklozenge Q].(a, b) = Q.(a, F.(a, b))$$

The set of relations (2.6-1) is, of course, a CPO with set-theoretic inclusion. One can show that \blacklozenge is continuous on both arguments. Since the limit of a chain is in our case the set-theoretic union, it is sufficient to show that \blacklozenge is distributive over union on both arguments, which means that the following equalities hold (we assume that \blacklozenge binds stronger than the union):

$$(F_1 \mid F_2) \circledast Q = F_1 \circledast Q \mid F_2 \circledast Q \quad \text{and}$$

$$F \circledast (Q_1 \mid Q_2) = F \circledast Q_1 \mid F \circledast Q_2$$

Let then

$$((a, b), c) : (F_1 \mid F_2) \circledast Q$$

which means that there exists a d such that

$$((a, b), d) : (F_1 \mid F_2) \quad \text{and} \quad ((a, d), c) : Q$$

which means that there exist i and d such that

$$((a, b), d) : F_i \quad \text{and} \quad ((a, d), c) : Q$$

which means that there exists i such that

$$((a, b), c) : F_i \circledast Q$$

which means that

$$((a, b), c) : F_1 \circledast Q \mid F_2 \circledast Q$$

In this way, we have proved the inclusion

$$(F_1 \mid F_2) \circledast Q \subseteq F \circledast Q_1 \mid F \circledast Q_2$$

The proofs of the remaining three inclusions are analogous.

Since \circledast is continuous on both arguments the following fixed-point equation has the least solution:

$$\text{power} = \text{zero} \mid (\text{minus} \circledast \text{power}) \circledast \text{times} \quad (2.6-2)$$

where:

$$\begin{aligned} \text{zero}(n, 0) &= 1 \\ \text{minus}(n, m) &= m-1 \quad \text{for } m > 0, \text{ and for } m = 0 \text{ this function is undefined} \\ \text{times}(n, m) &= n * m \end{aligned}$$

Since the set-theoretic union and our composition are both continuous in the CPO of relations (2.6-1), Kleene's theorem implies that the solution of (2.6-2) is the limit of the chain of relation

$$P^0 \subseteq P^1 \subseteq P^2 \subseteq \dots \quad (2.6-3)$$

which are functions defined in the following way:

$$\begin{aligned} P^0 &= \text{zero} \\ P^{i+1} &= (\text{minus} \circledast P^i) \circledast \text{times} \quad \text{for } i \geq 0 \end{aligned}$$

This means that for every $i \geq 0$ function P^i is a partial function of power restricted to $m \leq i$:

$$\begin{aligned} P^i(n, m) &= \\ m \leq i &\rightarrow m^i \\ \text{true} &\rightarrow ? \end{aligned}$$

Since all these functions coincide on the common parts of their domains, the set-theoretic union of the chain (2.6-3) is a function, and it is the power function defined for arbitrary $n, m \geq 0$.

2.7 A CPO of denotational domains

One of the main tools of denotational models of software systems are sets traditionally referred to as *domains*. These domains are most frequently defined using equations — possibly fixed-point equations — based on functions that are listed below. Some of them have been already defined, but I recall their descriptions just to have a full list in one place:

- 1) $A \mid B$ — set-theoretic union

- 2) $A \cap B$ — set-theoretic intersection
- 3) $A \times B$ — Cartesian product
- 4) A^{cn} — Cartesian n-th power
- 5) A^{c+} — Cartesian +-iteration
- 6) A^{c*} — Cartesian *-iteration
- 7) $\text{FinSub}.A$ — the set of all finite subsets
- 8) $A \Rightarrow B$ — the set of all mappings including the empty mapping
- 9) $A - B$ — set-theoretic difference
- 10) $\text{Sub}.A$ — the set of all subsets
- 11) $A \rightarrow B$ — the set of all functions from A to B
- 12) $A \mapsto B$ — the set of all total functions from A to B
- 13) $\text{Rel}.(A,B)$ — the set of all relations from A to B

These operators may be used in “direct” equations, e.g.

$$\begin{aligned} \text{State} &= \text{Identifier} \Rightarrow \text{Data} \\ \text{Instruction} &= \text{State} \rightarrow \text{State} \end{aligned} \tag{2.7-1}$$

or in fixed point equations, e.g.:

$$\begin{aligned} \text{Record} &= \text{Identifier} \Rightarrow \text{Data} \\ \text{Data} &= \text{Number} \mid \text{Record} \end{aligned} \tag{2.7-2}$$

Whereas definition (2.7-1) does not raise any doubts, in the case of (2.7-2) the situation is different. Since this is obviously a fixed-point equation we have to prove the continuity of \Rightarrow and \mid , but the continuity where? What is the CPO of domains? Set-theoretic inclusion is clearly its partial order, but what is the carrier?

Potentially that carrier should include all domains that we shall define in the future, hence something like the set of all sets. Unfortunately — as it has been known since 1930-ties — such a set does not exist²¹. Despite this fact, our problem can be solved on the base of M.P. Cohn’s [42] construction. As he has shown, for any collection of sets \mathbf{B} (a collection does not need to be a set!) there exists a set of sets $\text{Set}.\mathbf{B}$ with the following properties:

1. all sets in \mathbf{B} belong (as elements) to $\text{Set}.\mathbf{B}$,
2. $\text{Set}.\mathbf{B}$ is closed under all our operations from 1) to 13),
3. $\text{Set}.\mathbf{B}$ is closed under unions of all denumerable families of its elements,
4. the empty set \emptyset belongs to $\text{Set}.\mathbf{B}$.

Following this construction, we choose as family \mathbf{B} , the set of all initial domains that we shall use in our model, such as **Boolean**, **Number**, **Identifier**, **Character**, etc. Since $(\text{Set}.\mathbf{B}, \subseteq)$ is a set-theoretic CPO, we can talk about the continuity of functions defined on sets in $\text{Set}.\mathbf{B}$. As is easy to show operations from 1) to 8) are continuous, the difference of sets is continuous only on the left argument, and the remaining functions are not continuous, and therefore they cannot appear in fixed-point equations²².

²¹ Formally speaking the attempt of constructing such a set leads to a contradiction. Indeed, let Z be the set of all sets. Let then Z_e be the set of all sets that are their own elements and Z_n — the set of all sets that are not their own elements. Since obviously $Z = Z_e \mid Z_n$, set Z_n must belong to either Z_e or Z_n . The first case must be excluded since in that case Z_n should belong to Z_n . The second case is impossible either, since then Z_n must not belong to itself. Intuitively speaking one can say that the collection of all sets is “too large to be a set”.

²² As an example let me show that the operator \rightarrow is not continuous. Let then $A_1 \subset A_2 \subset \dots$ be an arbitrary chain of mutually different sets, and let B be an arbitrary set. The sequence of domains $A_i \rightarrow B$ constitutes a chain but none of its elements contain a total function on the union $\cup A_i$, hence none of such functions belong to $\cup (A_i \rightarrow B)$, which means

As we see, therefore, the equation (2.7-2) has a solution (the least solution) defined by the theorem of Kleene (Sec.2.3). Records defined in that way may “carry” other records, but of a “lower-level” than themselves. At the end of that hierarchy, we have records carrying numbers. If however, we replace \Rightarrow by \rightarrow , then (2.7-2) would have no solution. A problem of precisely that type encountered mathematicians who, in the early 1970-ties, had been trying to define denotational semantics for Algol 60. More on that subject in Sec. 3.1.

As can be easily proved, among our functions on domains 1) – 8) are continuous on both arguments, 9) is continuous on the first argument only, and 10) – 13) are not continuous in both arguments.

The fact that non-continuous operators cannot be used in fixed-point domain equations does not mean however that they cannot be used in fixed-point equations “at all”. For instance, our two sets of equations (2.7-1) and (2.7-2) can be legally combined into one:

$$\begin{array}{llll} \text{Data} & = & \text{Number} & | & \text{Record} & & & (2.7-3) \\ \text{Record} & = & \text{Identifier} & \Rightarrow & \text{Data} & & & \\ \text{State} & = & \text{Identifier} & \Rightarrow & \text{Data} & & & \\ \text{Instruction} & = & \text{State} & \rightarrow & \text{State} & & & \end{array}$$

Although “as a whole” this is a fixed-point set of equations with one non-continuous operation, the recursion is present in only in the second and the third equation where the operators are continuous. This set of equations is therefore legal.

2.8 Abstract errors

For practically all expressions appearing in programs, their values in some circumstances cannot be computed “successfully”. Here are a few examples²³:

- expression x/y cannot be evaluated if the variables x or y have not been declared as numbers,
- expression x/y cannot be evaluated if the current value of y is zero,
- expression $x+y$ cannot be evaluated if its value exceeds the maximal number allowed in current implementation,
- the value of the array expression $a[k]$ cannot be computed if the variable a has not been declared as an array or if k is out of the domain of a ,
- the query “Has John Smith retired?” cannot be answered if John Smith is not listed in the database.

In all these cases, a well-designed implementation should stop the execution of a program and generate an error message.

To describe that mechanism formally, we introduce the concept of an *abstract error*. In a general case, abstract errors may be anything, but in our models, they are going to be words, such as, e.g. ‘division-by-zero’. They are closed in apostrophes to distinguish them from metavariables at the level of **MetaSoft**.

The fact that an attempt to evaluate the expression $x/0$ raises an error message can be now expressed by the equation:

$$x/0 = \text{'division-by-zero'}$$

In the general case with every domain **Data**, we associate a corresponding domain with abstract errors

$$\text{DataE} = \text{Data} | \text{Error}$$

that $U(A_i \rightarrow B) \neq UA_i \rightarrow B$. In an analogous way we may show the non-continuity of the operators $A \mapsto B$ and $\text{Rel.}(A,B)$. Notice, however, that $U(A_i \Rightarrow B) = UA_i \Rightarrow B$, and similarly for the right-hand-side argument which means that \Rightarrow is continuous on both arguments.

²³ Here I anticipate the future rule of typesetting syntactic elements in *Courier New* (see Sec. 2.12)

where **Error** is a universal set of all abstract errors that may be generated during the execution of our programs. This set will be regarded as a parameter of a programming language. Now, every partial operation

$$\text{op} : \text{Data}_1 \times \dots \times \text{Data}_n \rightarrow \text{Data},$$

whose partiality is *computable*,²⁴ is extended to a total operation

$$\text{ope} : \text{DataE}_1 \times \dots \times \text{DataE}_n \mapsto \text{DataE}$$

Of course **ope** should coincide with **op** wherever **op** is defined, i.e. if d_1, \dots, d_n are not errors and $\text{op}.(d_1, \dots, d_n)$ is defined, then $\text{ope}.(d_1, \dots, d_n) = \text{op}.(d_1, \dots, d_n)$.

An operation **ope** will be said to be *transparent for errors* or simply *transparent* if the following condition is satisfied:

$$\text{if } d_k \text{ is the first error in the sequence } d_1, \dots, d_n, \text{ then } \text{ope}.(d_1, \dots, d_n) = d_k$$

This condition means that arguments of **ope** are evaluated one-by-one from left to right, and the first error (if it appears) becomes the final value of the computation.

The majority of operations on data that will appear in our models will be transparent. An exception are boolean operations discussed in Sec. 2.9.

Error-handling mechanisms are frequently implemented in such a way that errors serve only to inform the user that (and why) program evaluation has been aborted. Such a mechanism will be called *reactive*. In some applications, however, the generation of an error results in an action, e.g., of recovering the last state of a database (Sec. 10.9.6.4). Such mechanisms will be called *proactive*.

As we shall see in the sequel of the book, a reactive mechanism may be quite simply enriched to a proactive one. Since, however, the latter is technically more complicated, in the development of our example-language **Lingua**, except **Lingua-SQL**, we shall most frequently apply a reactive model. Proactive constructions are discussed in Sec. 5.1.5.5 and Sec. 10.9.6.4.

A well-defined error-handling mechanism allows avoiding situations where programs hang up without any explanation, or even worse — when they generate an incorrect result without warning the user (see Sec. 10.9.6.4).

2.9 A three-valued propositional calculus

Tertium non datur — used to say ancients masters. Computers denied this principle.

In the Aristotelean logic, every sentence is either true or false. The third possibility does not exist. In the world of computers, however, the third possibility is not only possible but just inevitable. In evaluating a boolean expression such as, e.g., $x/y > 2$ an error (see Sec. 2.8) may appear.

To describe the error-handling mechanism of boolean expressions the basic domain of two boolean values “true” and “false”:

$$\text{Boolean} = \{\text{tt}, \text{ff}\}$$

must be enriched by a third element

$$\text{BooleanE} = \{\text{tt}, \text{ff}, \text{ee}\}$$

where **ee** stands for “error”, but in this case represents either an error or an infinite computation (a looping). We assume for simplicity that there is only one error. This assumption does not affect the generality of our

²⁴ Partiality of a function f is computable, if there is an algorithm which for every element x can detect if $f.x$ is defined or not. In the examples of this section all functions have computable partialities. It is a well-known fact, however, that in the general case the definedness of recursive functions is not computable. E.g. there is no algorithm which given a program will check whether the execution of this program will terminate. Therefore, we cannot assume that any undefinedness will be signaled by an error message.

model since, as we are going to see later, at the level of boolean expressions, all errors will be treated in the same way.

Now, notice that the transparency of boolean operators would not be an adequate choice. To see that consider a conditional instruction:

```
if x ≠ 0 and 1/x < 10 then x := x+1 else x := x-1 fi
```

We would probably expect that for $x=0^{25}$, one should execute the assignment $x := x-1$. If however, our conjunction would be transparent, then the expression

```
x ≠ 0 and 1/x < 10
```

would evaluate to ‘division-by-zero’, which means that our program would abort. Notice also that the transparency of **and** would imply

```
ff and ee = ee
```

which would mean that when an interpreter evaluates **p and q**, then it first evaluates both **p** and **q** — as in “usual mathematics” — and only later applies **and** to them. Such a mode is called an *eager evaluation*.

An alternative to it is a *lazy evaluation* where, if **p = ff**, then the evaluation of **q** is abandoned, and the final value of the expression is **ff**. In such a case:

```
ff and ee = ff
```

```
tt or ee = tt
```

A three-valued propositional calculus with lazy evaluation was described in 1961 by John McCarthy [65], who defined boolean operators as in Tab. 2.9-1.

or-m	tt	ff	ee	and-m	tt	ff	ee	not-m	
tt	tt	tt	tt	tt	tt	ff	ee	tt	ff
ff	tt	ff	ee	ff	ff	ff	ff	ff	tt
ee	ee	ee	ee	ee	ee	ee	ee	ee	ee

Tab. 2.9-1 Propositional operators of John McCarthy

To see the intuition behind McCarthy’s operators consider the expression **p or-m q** assuming that its arguments are computed from left to right²⁶:

- If **p = tt**, then we give up the evaluation of **q** (lazy evaluation) and assume that the value of the expression is **tt**. Notice that in this case, we maybe avoid an error message generated by **q** or entering an infinite computation.
- If **p = ff**, then we evaluate **q**, and its value becomes the value of the expression.
- If **p = ee**, then this means that the evaluation aborts at the evaluation of **p**, hence **q** will not be evaluated. As a consequence, the final value of our expression must be **ee**.

The rule for **and** is analogous. Notice that McCarthy’s operators coincide with classical operators on classical values (grey fields in the table). McCarthy’s implication is defined classically:

```
p implies-m q = (not-m p) or-m q
```

²⁵ Whereas x (in Courier) denotes a variable x (in Arial) denotes its value.

²⁶ The suffix “-m” stands for “McCarthy” and is used to distinguish McCarthy’s operators not only from classical ones but also from the operators of Kleene, which are discussed later.

As we are going to see, not all classical tautologies remain satisfied in McCarthy’s calculus. Among those that are satisfied we have²⁷:

- associativity of alternative and conjunction,
- De Morgan’s laws

and among the non-satisfied are:

- **or-m** and **and-m** are not commutative, e.g., **ff and-m ee = ff** but **ee and-m ff = ee**,
- **and-m** is distributive over **or-m** only on the right-hand side, i.e.
 $p \text{ and-m } (q \text{ or-m } s) = (p \text{ and-m } q) \text{ or-m } (p \text{ and-m } s)$ however
 $(q \text{ or-m } s) \text{ and-m } p \neq (q \text{ and-m } p) \text{ or-m } (s \text{ and-m } p)$ since
 $(tt \text{ or-m } ee) \text{ and-m } ff = ff$ and $(tt \text{ and-m } ff) \text{ or-m } (ee \text{ and-m } ff) = ee$
- analogously **or-m** is distributive over **and-m** only on the right-hand side,
- $p \text{ or-m } (\text{not-m } p)$ does not need to be true but is never false,
- $p \text{ and-m } (\text{not-m } p)$ does not need to be false but is never true.

On the ground of that calculus, we build in Sec. 7 a calculus of three-valued partial predicates²⁸, which are later used in program construction rules.

An alternative to McCarthy’s propositional calculus is that of Kleene with operators defined in the following way:

or-k	tt	ff	ee
tt	tt	tt	tt
ff	tt	ff	ee
ee	tt	ee	ee

and-k	tt	ff	ee
tt	tt	ff	ee
ff	ff	ff	ff
ee	ee	ff	ee

not-k	
tt	ff
ff	tt
ee	ee

Tab. 2.9-2 Propositional operators of Steven Kleene

In that case

$$tt \text{ or-k } ee = ee \text{ or-k } tt = tt$$

$$ff \text{ and-k } ee = ee \text{ and-k } ff = ff$$

This calculus can be said even “more lazy” than that of McCarthy, since now whenever any argument of **or-k** is **tt**, then the result is **tt**, and analogously for **and-k**. Due to this assumption, we gain commutativity of both operators, but at the implementations level we have to compute both arguments of our operators “in parallel”. This is the consequence of the fact that **ee** may correspond to an infinite execution and therefore McCarthy’s left-to-right execution of **ee or-k tt** would not let us “learn” that the second argument is **tt**.

Although Kleene’s calculus is not implementable in a sequential mode, it may be quite useful in describing the properties of data, for instance at the level of integrity constraints in SQL (Sec. 10) or when talking about program correctness (Sec. 8).

²⁷ It is true only in the case where we have one error element.

²⁸ The partiality of predicates is due to the use of functional-procedure calls in expressions.

2.10 Data algebras

Data types that are used in programs are usually described by sets of objects — such as numbers, booleans, strings, arrays, lists, etc. — and some operations on these objects. For instance, a data type of numbers may be described as a tuple²⁹:

$$\underline{\text{AlgNum}} = (\text{Number}, \text{make.no.1}, \text{plus}, \text{minus}, \text{times}, \text{divide}) \quad (2.10-1)$$

This tuple will be called the *algebra of numbers* where **Number** — called the *carrier of the algebra* — is the set of all real numbers and **make.no.1**, **plus**, **minus**, **times**, **divide** are functions on numbers called *constructors*. The following formulas define the domains and the codomains of constructors:

$$\begin{array}{ll} \text{make-no.1:} & \mapsto \text{Number} \\ \text{plus} & : \text{Number} \times \text{Number} \mapsto \text{Number} \\ \text{minus} & : \text{Number} \times \text{Number} \mapsto \text{Number} \\ \text{times} & : \text{Number} \times \text{Number} \mapsto \text{Number} \\ \text{divide} & : \text{Number} \times \text{Number} \rightarrow \text{Number} \end{array} \quad (2.10-2)$$

The zero-argument function **make-no.1** (make number one) represents a *constant* of our algebra. This function has no arguments, and its unique value is 1, hence:

$$\text{make-no.1}().() = 1$$

If our algebra were part of a model of a programming language, the presence of this constant would mean that number 1 may be expressed directly at the level of syntax by writing `make-no.1`. Notice that the number 2 cannot be expressed in this way. Instead, we have to write e.g.

$$\text{plus}().(\text{make-no.1}().(), \text{make-no.1}().())$$

Number 2 is thus created from two ones, whereas number 1 — from “nothing”. Both

$$\text{make-no.1} . ()$$

and

$$\text{plus}().(\text{make-no.1}().(), \text{make-no.1}().())$$

are examples of expressions written in so-called *abstract syntax* (see Sec. 2.12). Since such a syntax is not very user-friendly, it is in general modified to *concrete syntax* (see Sec. 3.5), where we would write respectively 1 and 1+1.

Notice that **divide** is a partial function since dividing by zero is not possible.

Our algebra of numbers is an example of *abstract algebra*, and the list of formulas (2.10-2) is called their *signature* (formal definitions in Sec.2.11). The word “abstract” expresses the fact that our algebra of numbers is not a branch of mathematics dedicated to solving algebraic equations, but an abstract mathematical object.

Of course, in programming languages that operate on numbers, we restrict the set of available numbers — i.e., the carrier of the algebra — to a finite set of decimal numbers *representable* in the arithmetic of our computer³⁰. If by **NumberR** we denote the set of such numbers, then the signature of our algebra may be the following:

$$\begin{array}{ll} \text{make-no.num} & : \quad \quad \quad \mapsto \text{NumberR} & \text{for num : NumberR} \\ \text{plus} & : \text{NumberR} \times \text{NumberR} \rightarrow \text{NumberR} \\ \text{minus} & : \text{NumberR} \times \text{NumberR} \rightarrow \text{NumberR} \\ \text{times} & : \text{NumberR} \times \text{NumberR} \rightarrow \text{NumberR} \\ \text{divide} & : \text{NumberR} \times \text{NumberR} \rightarrow \text{NumberR} \end{array}$$

²⁹ Names of algebras will be underlined.

³⁰ Notice that in user manuals the range of numbers is usually defined as an interval, e.g. from -2^{63} to $2^{63} - 1$ (see [46]) without mentioning that numbers with infinite or with too long binary representations will be truncated.

In this algebra we assume to have a finite family of zero-argument constructors indexed by representable numbers:

$$\{\text{make-no.num} \mid \text{num} : \text{NumberR}\}$$

Here `make-no` is a meta-constructor that is not a constructor of our algebra but is used to generate zero-argument constructors of that algebra.

Note that in this algebra, all constructors except `make-no.num` are partial functions since each of them may lead out of the domain of representable numbers. However, the use of partial functions as constructors in an algebra has two disadvantages:

- Mathematical disadvantage — in the theory of abstract algebras, all constructors are assumed to be total; the introduction of partial constructors is, of course, possible but would complicate the model.
- Informatical disadvantage — in **Lingua**, we want to have error-messages that warn the user about each situation when an operation can't be performed.

To cope with both these problems we introduce abstract errors as described in Sec.2.8 and replace the carrier `NumberR` by the carrier

$$\text{NumberE} = \text{NumberR} \mid \text{Error}$$

where the set `Error` contains all error messages that we shall need in our algebra. Now the signature of our algebra is as follows:

<code>make-no.num</code>	:	$\mapsto \text{NumberE}$	for <code>num</code> : <code>NumberR</code>
<code>plus</code>	:	$\text{NumberE} \times \text{NumberE} \mapsto \text{NumberE}$	
<code>minus</code>	:	$\text{NumberE} \times \text{NumberE} \mapsto \text{NumberE}$	
<code>times</code>	:	$\text{NumberE} \times \text{NumberE} \mapsto \text{NumberE}$	
<code>divide</code>	:	$\text{NumberE} \times \text{NumberE} \mapsto \text{NumberE}$	

Passing to another aspect of data-type algebras notice that in the majority of programming languages, with data-type *number* we associate not only arithmetic operations but also *predicates* such as e.g. `less` or `equal`, hence functions with numerical arguments and boolean values. To describe such structures we need an algebra with two carriers — `NumberE` and `BooleanE` — hence

$$\text{AlgNumBoo} = (\text{NumberE}, \text{BooleanE}, \\ \{\text{make-no.num} \mid \text{num} : \text{NumberR}\}, \text{plus}, \text{minus}, \text{times}, \text{divide}, \\ \text{less}, \text{equal}, \text{make-bo.tt}, \text{make-bo.ff}, \text{or}, \text{and}, \text{not})$$

The signature of this algebra in the following:

<code>make-no.num</code>	:	$\mapsto \text{NumberE}$	for <code>num</code> : <code>NumberR</code>	
<code>plus</code>	:	$\text{NumberE} \times \text{NumberE} \mapsto \text{NumberE}$		
<code>minus</code>	:	$\text{NumberE} \times \text{NumberE} \mapsto \text{NumberE}$		
<code>times</code>	:	$\text{NumberE} \times \text{NumberE} \mapsto \text{NumberE}$		
<code>divide</code>	:	$\text{NumberE} \times \text{NumberE} \mapsto \text{NumberE}$		
<code>less</code>	:	$\text{NumberE} \times \text{NumberE} \mapsto \text{BooleanE}$		(2.10-3)
<code>equal</code>	:	$\text{NumberE} \times \text{NumberE} \mapsto \text{BooleanE}$		
<code>make-bo.tt</code>	:	$\mapsto \text{BooleanE}$		
<code>make-bo.ff</code>	:	$\mapsto \text{BooleanE}$		
<code>or</code>	:	$\text{BooleanE} \times \text{BooleanE} \mapsto \text{BooleanE}$		
<code>and</code>	:	$\text{BooleanE} \times \text{BooleanE} \mapsto \text{BooleanE}$		
<code>not</code>	:	$\text{BooleanE} \mapsto \text{BooleanE}$		

An algebra with two carriers is said to be a *two-sorted algebra*. Sometimes signature of many-sorted algebras are visualizes graphically as in Fig. 2.10-1. For simplicity of the figure I included only some operation of the algebra and I use concrete-syntax names of operators `1`, `0`, `+`, `=`, etc.

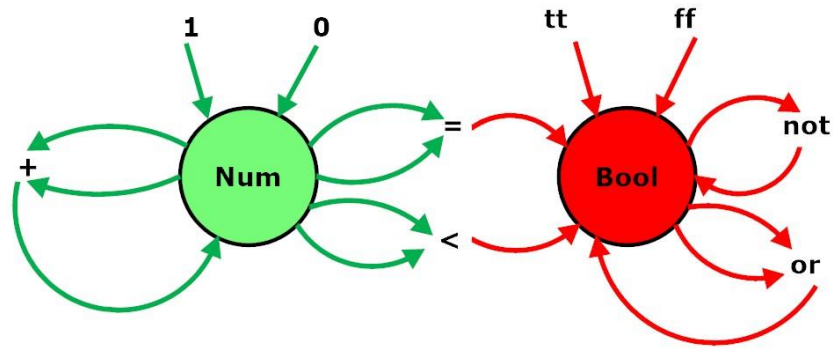


Fig. 2.10-1 Graphical representation of a two-sorted algebra

Of course, all operations of our algebra must not lead out of the set of representable numbers NumberR , which should be taken into account in their definitions. For example the operation of addition will be defined in the following form:

```

plus.(num-1, num-2) = 31
  num-1 : Error      → num-1
  num-2 : Error      → num-2
  not +.(num-1, num-2) : NumberR → 'overflow'32
  true              → +.(num-1, num-2)
    
```

where „+” is the arithmetical addition. Notice that plus is not commutative since

$$\text{plus}(\text{err-1}, \text{err-2}) \neq \text{plus}(\text{err-2}, \text{err-1})$$

if only $\text{err-1} \neq \text{err-2}$.

Note that our arithmetic operations are not associative, either. This is due to the limitation of the size of numbers. E.g., if 1000 is the maximal admissible number, then

$$(700 + 400) - 200 = \text{'overflow'} \quad \text{but} \\ 700 + (400 - 200) = 900$$

2.11 Many-sorted algebras

Our algebra AlgNumBoo is said to be *two-sorted* since it has two carriers: NumberE and BooleanE . In the sequel, we shall construct algebras with more than one carrier called *many-sorted algebras* or simply *algebras*. Formally a many-sorted algebra is the following tuple:

$$\text{Alg} = (\text{Sig}, \text{Car}, \text{Fun}, \text{car}, \text{fun})$$

where

- Sig = (Cn, Fn, ar, so) — is called the *signature of the algebra*,
- Cn — is a finite set of words called the *names of carriers*; the carriers themselves are called *sorts*,
- Fn — is a finite set of words called the *names of functions*; the functions themselves are called *constructors*

³¹ Here I anticipate a **MetaSoft** notation, that we are going to use later, where instead of writing num_1 we write num-1 , and similarly for analogous cases.

³² The negation operator **not** in this clause is not a constructor of our algebra, but a metaconstructor from the level of **MetaSoft**.

$ar : Fn \mapsto Cn^*$	— to every name of a function fn the function ar assigns a finite (possibly empty) sequence of sorts' names $ar.fn = (cn_1, \dots, cn_k)$ called the <i>arity</i> of fn ³³
$so : Fn \mapsto Cn$	— to every name of a function fn the function so assigns a carrier name $so.fn$ which is called <i>the sort of</i> fn ,
Car	— a finite set of <i>carriers</i> ,
Fun	— a finite set of total functions with arguments and values in carriers; these functions are called <i>constructors</i> ,
$car : Cn \mapsto Car$	— to every name cn of a carrier function car assigns a carrier $car.cn$,
$fun : Fn \mapsto Fun$	— to every function name fn such that $ar.fn = (cn_1, \dots, cn_k)$ $so.fn = cn$ the function fun assigns a total function $fun.fn : car.cn_1 \times \dots \times car.cn_k \mapsto car.cn$

The concepts of *arity* and *sort* are applied not only to function names but also to the corresponding functions themselves. Functions in the set Fun are traditionally called *constructors*. The tuple $((cn_1, \dots, cn_k), cn)$ that describes the arity and the sort of a constructor will also be called the *signature of that constructor*.

Zero-argument constructors, i.e., constructors whose arity is an empty sequence, are called *constants* of the algebra. If f is such a constant, then we write

$$f : \mapsto Carrier$$

and the unique value of f is written as

$$f.()$$

It should be emphasized that all constructors of an algebra are total functions. In our case, this is due to the use of abstract errors (Sec. 2.8).

As we are going to see, our formal definition of many-sorted algebras has been introduced to describe the derivation of syntax from denotations in the construction of a programming language. For concrete algebras, however, e.g., such as discussed in Sec.2.10, the signature is implicit in the set of formulas such as (2.10-3). Now consider two algebras:

$$\underline{Alg}_i = (Sig_i, Car_i, Fun_i, car_i, fun_i) \quad \text{for } i = 1, 2$$

with signatures

$$Sig_i = (Cn_i, Fn_i, ar_i, so_i) \quad \text{for } i = 1, 2$$

We say that Sig_2 is an *extension* of Sig_1 or that Sig_1 is a *restriction* of Sig_2 , if

$$1. Cn_1 \subseteq Cn_2 \text{ and } Fn_1 \subseteq Fn_2,$$

³³ The word „arity” comes from unary, binary, ternary etc.

2. functions ar_2, so_2 coincide with ar_1, so_1 on F_{n_1} .

We say that algebra \underline{Alg}_2 is an *extension of algebra* \underline{Alg}_1 , if

1. Sig_2 is an extension of Sig_1 ,
2. $car_{1.cn} \subseteq car_{2.cn}$ for every sort $cn : C_{n_1}$,
3. $fun_2.fn$ coincides with $fun_1.fn$ on the appropriate carriers for every $fn : F_{n_1}$.

In other words, each (nontrivial) extension of an algebra results from that algebra by adding new carriers and/or new constructors and/or new elements to the existing carriers.

Two many-sorted algebras are said to be *similar* if they have the same signature. In the future, we shall frequently define concrete algebras by defining their carriers and constructors but without showing their signatures explicitly. In that case, we shall say that two algebras are similar if it is possible to construct a common signature for them.

If \underline{Alg}_1 and \underline{Alg}_2 are similar, then we say that \underline{Alg}_1 is a *subalgebra* of \underline{Alg}_2 if:

1. the carriers of \underline{Alg}_1 are subsets of the corresponding carriers of \underline{Alg}_2 ,
2. the constructors of \underline{Alg}_1 coincide with constructors of \underline{Alg}_2 on the carriers of \underline{Alg}_1 .

Therefore every subalgebra of an algebra is a restriction of that algebra but not vice versa. By a *many-sorted homomorphism* from algebra \underline{Alg}_1 into a similar algebra \underline{Alg}_2 where

$$\underline{Alg}_i = (Sig, Car_i, Fun_i, car_i, fun_i) \quad \text{for } i = 1, 2$$

we call a family of functions

$$H = \{h.cn \mid cn : C_n\}$$

whose elements — called *the components of that homomorphism* — map the elements of \underline{Alg}_1 into the elements of \underline{Alg}_2 , hence

$$h.cn : car_{1.cn} \mapsto car_{2.cn} \quad \text{for all } cn : C_n$$

and where for every constructor name $fn : C_n$ such that

$$ar.fn = (cn_1, \dots, cn_n) \quad \text{where } n \geq 0$$

and every tuple of arguments

$$(a_1, \dots, a_n) : car_{1.cn_1} \times \dots \times car_{1.cn_n}$$

the following equality holds

$$h.cn.(fun_1.fn.(a_1, \dots, a_n)) = fun_2.fn.(h.cn_1.a_1, \dots, h.cn_n.a_n) \quad (2.11-1)$$

In other words a homomorphic image of the value of a function $fun_1.fn$ from the first algebra with arguments (a_1, \dots, a_n) equals the value of the corresponding function $fun_2.fn$ from the second algebra applied to the tuple of homomorphic images of the first tuple i.e. applied to $(h.cn_1.a_1, \dots, h.cn_n.a_n)$. Notice that for $n = 0$ the equality (2.11-1) has the form

$$h.cn.(fun_1.fn.()) = fun_2.fn.()$$

The fact that H is a homomorphism from \underline{Alg}_1 into \underline{Alg}_2 shall be written as:

$$H : \underline{Alg}_1 \mapsto \underline{Alg}_2$$

Our definition of homomorphism implies that if some carriers of the algebra \underline{Alg}_1 are empty, then the corresponding components of the homomorphism have to be empty as well. An algebra where all carriers are empty is called *an empty algebra*.

In the general case, homomorphisms do not map algebras onto algebras but into algebras, which means that not every element in \underline{Alg}_2 must be an image of an element from \underline{Alg}_1 . For instance an identity homomorphism from integers to numbers

$I2N : (\text{Integer}, 1, \text{plus}, \text{minus}) \mapsto (\text{Number}, 1, \text{plus}, \text{minus})$

is not “onto”, whereas a homomorphism from integers into even integers

$I2E : (\text{Integer}, 1, \text{plus}, \text{minus}) \mapsto (\text{Even}, 1, \text{plus}, \text{minus})$

defined by the equality $I2E.\text{int} = 2*\text{int}$ is “onto”. In the general case a homomorphism $H : \underline{Alg}_1 \mapsto \underline{Alg}_2$ is called:

- a *monomorphism* — if all its components are one-to-one functions; e.g., $I2N$ and $I2E$,
- an *epimorphism* — if all its components are “onto”; e.g., $I2E$
- an *isomorphism* — if it is both a monomorphism and an epimorphism; e.g., $I2E$.

Theorem 2.11-1 For every homomorphism $H : \underline{Alg}_1 \mapsto \underline{Alg}_2$, the image of \underline{Alg}_1 in \underline{Alg}_2 , i.e., the restriction of \underline{Alg}_2 to the images through H of \underline{Alg}_1 with the appropriate truncation of constructors of \underline{Alg}_2 constitutes a subalgebra of \underline{Alg}_2 . ■

Proof To prove our theorem, we have to show that the images in \underline{Alg}_2 of the carriers of \underline{Alg}_1 are closed under the operations of \underline{Alg}_2 . Let then (b_1, \dots, b_n) from \underline{Alg}_2 , be the image of (a_1, \dots, a_n) in \underline{Alg}_1 , i.e. let:

$$(b_1, \dots, b_n) = (h.cn_1.a_1, \dots, h.cn_n.a_n)$$

Let furthermore for some function name fn

$$fun_2.fn.(b_1, \dots, b_n) = b$$

We have to show that b has a coimage in \underline{Alg}_1 . It is indeed the case since on the ground of (2.11-1):

$$fun_2.fn.(b_1, \dots, b_n) = fun_2.fn.(h.cn_1.a_1, \dots, h.cn_n.a_n) = h.cn.(fun_1.fn.(a_1, \dots, a_n))$$

hence $h.cn.(fun_1.fn.(a_1, \dots, a_n))$ is the coimage of b in \underline{Alg}_1 . ■

An algebra, which is the image of a homomorphism, $\underline{Alg}_1 \mapsto \underline{Alg}_2$ is called *the kernel of the homomorphism* H in \underline{Alg}_2 .

All our investigations about homomorphisms can be generalized to the case where the signatures of two algebras

$$Sig_i = (Cn_i, Fn_i, ar_i, so_i) \quad \text{for } i = 1, 2$$

are not identical but are *similar* in the sense that there exist two reversible functions of similarity

$$S_n : Cn_1 \mapsto Cn_2$$

$$S_f : Fn_1 \mapsto Fn_2$$

such that if

$$S_f.fn_1 = fn_2$$

$$ar_1.fn_1 = cn_{11}, \dots, cn_{1p}$$

$$ar_2.fn_2 = cn_{21}, \dots, cn_{2m}$$

then

$$p = m$$

$$S_n.cn_{1i} = cn_{2i} \quad \text{for } i = 1;p$$

In other words, two signatures are similar if they have the same number of carrier names and function names, and the corresponding function names have identical arities and sorts up to the names of carriers.

Now we can generalize the notion of the similarity of algebras: two algebras shall be called *similar* if their signatures are similar. For any fixed functions, S_n and S_f the concept of homomorphism, and the corresponding theorems remain valid for the generalized similarity.

2.12 Abstract syntax and reachable algebras

Every signature

$$\text{Sig} = (\text{Cn}, \text{Fn}, \text{ar}, \text{so})$$

unambiguously determines a certain algebra with that signature and with formal languages as carriers. This algebra is called *abstract syntax over signature* Sig and will be denoted by $\text{AbsSy}(\text{Sig})$ ³⁴. The elements of its carriers are words of a many-sorted formal language

$$\{\text{Lan.cn} \mid \text{cn} : \text{Cn}\}$$

defined by an equational grammar (see Sec.2.5) in a way described below.

To every carrier name cn we associate a language denoted by Lan.cn . The tuple of all these languages is defined by an equational grammar where for every $\text{cn} : \text{Cn}$ we have the following equation³⁵:

$$\begin{aligned} \text{Lan.cn} = \{ \text{fn}_1 \} \circ \{ \} \circ \text{Lan.cn}_{11} \circ \{ , \} \circ \dots \circ \{ , \} \circ \text{Lan.cn}_{1n(1)} \circ \{ \} \mid \\ \dots \\ \{ \text{fn}_k \} \circ \{ \} \circ \text{Lan.cn}_1 \circ \{ , \} \circ \dots \circ \{ , \} \circ \text{Lan.cn}_{n(k)} \circ \{ \} \} \end{aligned} \quad (2.12-1)$$

Here fn_i for $i = 1;k$ are function names with

$$\text{so.fn}_i = \text{cn}$$

and

$$\text{ar.fn}_i = (\text{cn}_{i1}, \dots, \text{cn}_{in(i)}) \quad \text{for } i = 1;k$$

We assume that if for a carrier name cn there is no function name fn such that $\text{so.nf} = \text{cn}$, then the corresponding language is empty, i.e. its defining equation is:

$$\text{Lan.cn} = \emptyset$$

For every non-empty Lan.cn , its elements are words of the form

$$\text{fn}_i(\text{w}_{i1}, \dots, \text{w}_{in(i)})$$

i.e. of the form $\text{fn}_i \circ (\circ \text{w}_{i1} \circ \dots \circ \text{w}_{in(i)} \circ)$ where \circ is the concatenation of words and

$$\text{w}_{ik} : \text{Lan.cn}_k.$$

In this place, it is worth noticing that if there are no zero-argument functions' names (constants) in the signature, then all languages (carriers) of the corresponding abstract syntax are empty.

Since abstract syntaxes are generated from signatures, they may be associated with arbitrary algebras (through their signatures). If Alg is an algebra with signature Sig , then $\text{AbsSy}(\text{Sig})$ will be called *the abstract syntax of algebra* Alg . For instance, if AlgNumBoo is the two-sorted algebra described in Sec.2.10 than the carrier of its abstract syntax are defined by the following equational grammar, where NumExp and BooExp are languages of numeric expressions and boolean expressions respectively:

$$\begin{aligned} \text{NumExp} = 0 \mid 1 \mid \\ \text{plus}(\text{NumExp}, \text{NumExp}) \mid \text{minus}(\text{NumExp}, \text{NumExp}) \mid \\ \text{times}(\text{NumExp}, \text{NumExp}) \mid \text{divide}(\text{NumExp}, \text{NumExp}) \end{aligned} \quad (2.12-1)$$

$$\text{BooExp} = \text{tt} \mid \text{ff} \mid$$

³⁴ The idea of an abstract syntax regarded as a mathematical idealization of a syntax of a programming language appeared for the first time in papers of J. McCarthy [65] and P. Landin [59] but with abstract algebras was for the first time associated by J.A. Goguen, J.W. Thatcher, E.G. Wagner and J.B. Wright [51]. A little later I used that concept in an attempt to give a formal semantics to a subset of Pascal [24]. In this book abstract syntax is understood in a slightly different way (technically) but the idea is roughly the same.

³⁵ We assume, of course, that the commas “,” and the parentheses “(“ and “)” do not appear in the signature as constructors' names.

$$\text{less}(\text{NumExp}, \text{NumExp}) \mid \text{equal}(\text{NumExp}, \text{NumExp}) \mid \\ \text{or}(\text{BooExp}, \text{BooExp}) \mid \text{and}(\text{BooExp}, \text{BooExp}) \mid \text{not}(\text{BooExp})$$

In this grammar, I use four notational conventions that we shall assume as standards for future use:

1. words such as 0 , 1 , plus , $($, $)$ etc. that appear at the level of syntax are typeset with *Courier New*, whereas NumExp and BooExp are typeset in *Arial*, since they are metavariables from the level of **MetaSoft**,
2. if it does not lead to a confusion a one-element set $\{a\}$ is written as a ,
3. for each zero-argument constructor named kn , instead of $\text{kn}()$ I write kn , e.g., 1 instead of $1()$,
4. the concatenation sign \odot is omitted, e.g., I write ab instead of $a \odot b$,

Examples of a numeric and boolean abstract-syntax expression are the following:

- $\text{plus}(\text{plus}(\text{minus}(1, 0), 1), \text{plus}(1, 1))$
- $\text{or}(\text{less}(\text{plus}(\text{plus}(\text{minus}(1, 0), 1), \text{plus}(1, 1)), \text{plus}(1, 1)), \text{ff})$

As we see, the expressions of our languages do not contain variables and are written in a *prefix notation* where function symbols always precede their arguments. E.g., we write $\text{plus}(1, 1)$ instead of $(1 \text{ plus } 1)$. The latter style is called *infix-notation*.

In the syntactic algebra defined by our grammar, the elements of carriers are numeric and boolean expressions, respectively (without variables), and constructors correspond to constructor names from our signature. For instance, with a constructor name plus , we associate a constructor $[\text{plus}]$ of the algebra $\text{AbsSy}(\text{Sig})$ defined by the equation

$$[\text{plus}].[\text{num-exp}_1, \text{num-exp}_2] = \text{plus}(\text{num-exp}_1, \text{num-exp}_2)^{36}$$

This constructor, given two expressions num-exp_1 and num-exp_2 returns the expression of the form $\text{plus}(\text{num-exp}_1, \text{num-exp}_2)$. E.g. given $\text{times}(x, y)$ and $\text{plus}(z, y)$ returns

$$\text{plus}(\text{times}(x, y), \text{plus}(z, y))$$

Now we can formulate a theorem with fundamental importance for denotational models of programming languages.

Theorem 2.12-1 *For every many-sorted algebra Alg with a signature Sig there is exactly one homomorphism $H : \text{AbsSy}(\text{Sig}) \mapsto \text{Alg}$. ■*

Proof Every homomorphism $H : \text{AbsSy}(\text{Sig}) \mapsto \text{Alg}$ must (from the definition) satisfy the equation:

$$H.\text{cn}.[\text{fn}(w_1, \dots, w_n)] = \text{fun}.\text{fn}.[H.\text{cn}_1.w_1, \dots, H.\text{cn}_n.w_n]$$

where

$$\text{ar}.\text{fn} = (\text{cn}_1, \dots, \text{cn}_n)$$

$$\text{so}.\text{fn} = \text{cn}$$

$$w_i : \text{Lan}.\text{cn}_i \quad \text{for } i = 1; n$$

Since every word in abstract syntax is of a unique (for it) form $\text{fn}(w_1, \dots, w_n)$, the above equations (for all fn) define the family $\{H.\text{cn} \mid \text{cn} : \text{Cn}\}$ in an unambiguous way. In the case of empty carriers of $\text{AbsSy}(\text{Sig})$ the corresponding components of H are empty. ■

The unique homomorphism from $\text{AbsSy}(\text{Sig})$ to Alg will be called *the semantics of abstract syntax*. For instance, if by $\{\mathbf{N}, \mathbf{B}\}$ we denote the semantics of abstract syntax of AlgNumBoo , then this homomorphism maps boolean expression $\text{less}(\text{plus}(1, 1), \text{times}(1, 1))$ into the boolean value ff :

³⁶ The meta-parentheses “[” and “]” are introduced in order to distinguish them from parentheses that belong to the defined language.

$B.[less(plus(1,1), times(1,1))] =$
 $fun.less(N.[plus(1,1)], N.[times(1,1)]) =$
 $fun.less(fun.plus(N.[1], N.[1]), fun.times(N.[1], N.[1])) =$
 $fun.less(fun.plus(1,1), fun.times(1,1)) = ff$

On the ground of theorems 2.11-1 and 2.12-1, in every algebra \underline{Alg} , there is a unique subalgebra which is the kernel of the semantics of abstract syntax of \underline{Alg} . That algebra is called *the reachable subalgebra* of \underline{Alg} . This name expresses the fact that every element of that algebra can be constructed (reached) by using the constructors of the algebra. For instance, the reachable subalgebra of the algebra

(Number, 1, plus, divide)

is the algebra of positive rational numbers

(PosRat, 1, plus, divide)

since only such numbers can be constructed from 1 in using plus and divide. Notice that if we remove 1 from the algebra of numbers, then its reachable algebra becomes empty and consequently its algebra of abstract syntax will be empty as well.

An algebra is called *reachable* if it coincides with its reachable subalgebra. In particular, every algebra of abstract syntax is reachable. Reachable is also every empty algebra. Now we can formulate two important theorems.

Theorem 2.12-2 For any two similar algebras \underline{Alg}_1 and \underline{Alg}_2 , if \underline{Alg}_1 is reachable, then there is at most one homomorphism

$$H : \underline{Alg}_1 \mapsto \underline{Alg}_2,$$

and if this is the case, then the image of \underline{Alg}_1 in \underline{Alg}_2 is reachable. ■

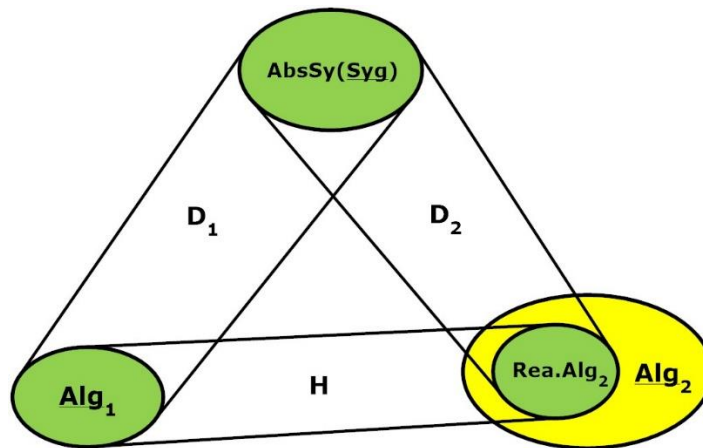


Fig. 2.12-1 Reachable algebras

Proof. The theorem and its proof are illustrated in Fig. 2.12-1. Since \underline{Alg}_1 and \underline{Alg}_2 are similar, they must have a common signature \underline{Sig} and a common abstract syntax $\underline{AbsSy}(\underline{Sig})$. Therefore — on the ground of Theorem 2.12-1 — there exist two unambiguously defined semantics of abstract syntaxes

$$D_1 : \underline{AbsSy}(\underline{Sig}) \mapsto \underline{Alg}_1 \text{ and}$$

$$D_2 : \underline{AbsSy}(\underline{Sig}) \mapsto \underline{Alg}_2$$

Now, if there exists a homomorphism $H : \underline{Alg}_1 \mapsto \underline{Alg}_2$, then the composition

$$D_1 \bullet H : \underline{AbsSy}(\underline{Sig}) \mapsto \underline{Alg}_2$$

defined as the composition of their components is a homomorphism. Since D_2 is the unique homomorphism between these algebras, we have

$$D_1 \bullet H = D_2,$$

and since $\underline{\text{Alg}}_1$ is reachable, the above equation defines H unambiguously, because otherwise, we could define another homomorphism from $\underline{\text{AbsSy}}(\text{Sig})$ into $\underline{\text{Alg}}_2$ which would contradict Theorem 2.12-1. This proves that the image of $\underline{\text{Alg}}_1$ in $\underline{\text{Alg}}_2$ is reachable. ■

As an immediate consequence of this theorem we have another theorem:

Theorem 2.12-3 For every nonempty algebra $\underline{\text{Alg}}$ over signature Sig the following claims are equivalent:

- (1) $\underline{\text{Alg}}$ is reachable,
- (2) every homomorphism of the type $H : \underline{\text{Alg}}_1 \mapsto \underline{\text{Alg}}$ (for an arbitrary $\underline{\text{Alg}}_1$) is onto,
- (3) the semantics of abstract syntax $D : \underline{\text{AbsSy}}(\text{Sig}) \mapsto \underline{\text{Alg}}$ is onto. ■

Proof Let $\underline{\text{Alg}}$ be reachable and let for some $\underline{\text{Alg}}_1$ similar to $\underline{\text{Alg}}$ there exist a homomorphism

$$H : \underline{\text{Alg}}_1 \mapsto \underline{\text{Alg}},$$

and let

$$D : \underline{\text{AbsSy}}(\text{Sig}) \mapsto \underline{\text{Alg}}_1$$

be the abstract-syntax semantics of $\underline{\text{Alg}}_1$. In that case

$$D \bullet H : \underline{\text{AbsSy}}(\text{Sig}) \mapsto \underline{\text{Alg}}$$

is the abstract-syntax semantics for $\underline{\text{Alg}}$, hence, since $\underline{\text{Alg}}$ is reachable, then $D \bullet H$ must be *onto*, and therefore also H must be *onto*. Hence (1) implies (2). Now (3) follows from (2) as its particular case, and (2) implies (1) by the definition of reachability. ■

At the end of this section, one more useful theorem:

Theorem 2.12-4 An algebra has a nonempty reachable subalgebra if and only if it contains at least one zero-argument constructor. ■

Proof If there is a constant in the algebra, then it belongs to its reachable part, and hence, this part is not empty. If, however, such a constant does not exist, then in the grammar corresponding to that algebra, there are no constant monomials, and therefore all the carriers of abstract syntax are empty. Therefore the reachable part of $\underline{\text{Alg}}$ is an empty algebra. ■

Abstract syntaxes are, in general, not very convenient in practical programming, and therefore they are usually replaced by more user-friendly syntaxes historically called *concrete syntaxes*. In such a case, elements of abstract syntax correspond to *parsing trees* of concrete expressions (see, e.g. [3]).

2.13 Ambiguous and unambiguous algebras

An algebra $\underline{\text{Alg}}$ with a signature Sig is said to be *unambiguous* if its abstract-syntax semantics

$$D : \underline{\text{AbsSy}}(\text{Sig}) \mapsto \underline{\text{Alg}}$$

is a monomorphism, i.e., if for every carrier Car.cn of $\underline{\text{Alg}}$ and every element e of that carrier there is at most one word $w : \text{Lan.cn}$ in the abstract syntax $\underline{\text{AbsSy}}(\text{Sig})$ such that

$$D.\text{cn}.w = e$$

Algebras which are not unambiguous are called *ambiguous*.

Algebras of denotations of programming languages are practically always ambiguous. For instance, the algebra $\underline{\text{AlgNum}}$ described in 2.10 (if supplemented with abstract errors to make their constructor total) is ambiguous since, e.g., two different words $\text{plus}(\text{plus}(1, 1), 1)$ and $\text{plus}(1, \text{plus}(1, 1))$ correspond to the same number 3.

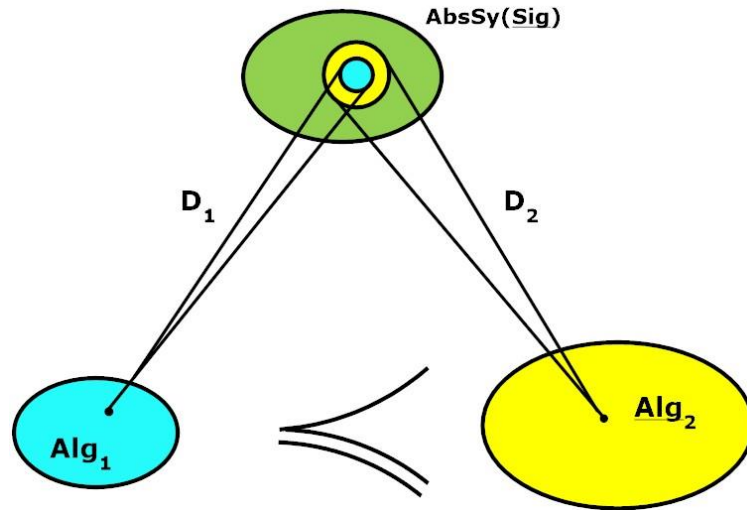


Fig. 2.13-1 Two ambiguous algebras

Now consider two algebras Alg_1 and Alg_2 with a common signature Sig hence also with a common abstract syntax $SkAbs(Sig)$. Let

$$D_1 : SkAbs(Sig) \mapsto Alg_1$$

$$D_2 : SkAbs(Sig) \mapsto Alg_2$$

be two corresponding abstract-syntax semantics. Algebra Alg_1 is said to be *less (or equally) ambiguous than* algebra Alg_2 , what we shall write as

$$Alg_1 \preceq Alg_2$$

if the homomorphism D_2 is gluing not more than D_1 (Fig. 2.13-1), i.e., if for any two words w_1 and w_2 in abstract syntax that belong to the same carrier $Car.cn$ the following implication holds:

$$\text{if } D_{1.cn}.w_1 = D_{1.cn}.w_2 \quad \text{then} \quad D_{2.cn}.w_1 = D_{2.cn}.w_2$$

Intuitively speaking, whenever an element of Alg_1 may be constructed in two different ways, the two ways lead to the same element in Alg_2 .

Ambiguous algebras play a certain role in the theory of programming languages since, for the majority of existing languages, their algebras of concrete syntax — if formally described — would turn out to be ambiguous. To explain this fact assume that $AbsSy(Sig)$ is defined by the grammar

$$NumExp = 0 \mid 1 \mid + (NumExp, NumExp),$$

Alg_1 is an algebra of infix expressions without parentheses defined by the grammar

$$NumExp = 0 \mid 1 \mid NumExp + NumExp$$

and Alg_2 is the algebra of integers. Let now D_1 replaces prefixes by infixes and removes parentheses.

Anticipating the considerations of Sec. 3, the algebra of numbers is the *algebra of denotations* (of meanings) for both our algebras of numeric expressions and the homomorphism D_2 is the *denotational homomorphism* (the *semantics*) of the algebra of abstract syntax. Now, we may raise a question, if there exists a denotational homomorphism

$$D_{12} : Alg_1 \mapsto Alg_2$$

from parentheses-free expressions into numbers.

To answer this question notice that for such algebras and their corresponding homomorphisms the following equalities hold:

$$\begin{aligned} D_1.[+ (+ (1, 1), 1)] &= 1+1+1 & D_2.[+ (+ (1, 1), 1)] &= 3 \\ D_1.[+ (1, + (1, 1))] &= 1+1+1 & D_2.[+ (1, + (1, 1))] &= 3 \end{aligned}$$

As we see D_1 is gluing not more than D_2 . In “practical mathematics”, hence also in programming languages, we frequently omit “unnecessary parentheses” whenever we deal with associative operations. The corresponding algebras are, in general, ambiguous, and therefore, the denotational homomorphism D_{12} need not exist. If however, they are not more ambiguous than the algebras of denotations, then such a homomorphism exist which follows from the following theorem:

Theorem 2.13-1 *If \underline{Alg}_1 and \underline{Alg}_2 are similar and \underline{Alg}_1 is reachable, then the (unique) homomorphism*

$D_{12} : \underline{Alg}_1 \mapsto \underline{Alg}_2$ *exists iff $\underline{Alg}_1 \preceq \underline{Alg}_2$. ■*

This unique homomorphism may be constructed as (intuitively speaking) the composition of the inverse of D_1 with D_2 , hence

$$D_{12} = D_1^{-1} \bullet D_2.$$

Although the inverse of D_1 maps the elements of \underline{Alg}_1 into sets of abstract expressions, yet all these expressions are mapped by D_2 into the same element of \underline{Alg}_2 . For formal proof of this theorem, see [27].

Of course, if D_1 is an isomorphism then \underline{Alg}_1 is “equally ambiguous” as \underline{Alg}_2 , and therefore the homomorphism D_{12} exists.

2.14 Algebras and grammars

The first step in the process of programming-language construction consists in defining an algebra of denotations from which we derive a unique algebra of abstract syntax. Since the latter is usually not user-friendly, we transform it into a concrete syntax (cf. Sec. 2.12) using a homomorphism that does not glue more than abstract-syntax semantics. Since in a user manual concrete syntax should be described by an equational grammar, we should raise a question, whether for any algebra of concrete syntax a corresponding grammar exists. To treat this problem formally, we need the concepts of *a skeleton function*.

A function f on words over an alphabet A is said to be a *skeleton function* if there exists a tuple of words $(w_1, \dots, w_k, w_{k+1})$ over A , called *the skeleton of this function* such that

$$f.(x_1, \dots, x_k) = w_1 x_1 \dots w_k x_n w_{k+1}$$

An example of a skeleton function may be

$$f.(exp-b, ins_1, ins_2) = \text{if } exp-b \text{ then } ins_1 \text{ else } ins_2 \text{ fi}$$

The skeleton of this function is (**if, then, else, fi**). Notice that the function

$$f.(exp-b, ins_1, ins_2) = \text{if } exp-b \text{ then } ins_2 \text{ else } ins_1 \text{ fi}$$

is not a skeleton function since the order of arguments on the left-hand side of our equation does not coincide with the order on its right-hand side.

In particular cases, a skeleton function may have more than one skeleton. E.g. the one-argument function

$$f : \{a\}^* \mapsto \{a\}^*$$

defined by equation

$$f.(x) = x a$$

has two skeletons (ϵ, a) and (a, ϵ) , since it may be equivalently defined by the equation

$$f.(x) = a x$$

However, if we change the type of the function f to $f : \{a, b\}^* \mapsto \{a, b\}^*$ without changing the defining equation, then this function has only one skeleton (ϵ, a) .

A many-sorted algebra will be called a *syntactic algebra* if it is a reachable algebra of words.

A syntactic algebra will be called a *context-free algebra* if all its constructors are skeleton functions. Of course, algebras of abstract syntax are context-free. As was shown in Sec. 2.12, for each such algebra, we can

build an equational grammar that defines its carriers and constructors. Similarly, we may assign an equational grammar for any context-free algebra.

Theorem 2.14-1 *For every context-free algebra, there is an equational grammar that generates its carriers.* ■

The following theorem is also true:

Theorem 2.14-2 *For every equational grammar there is a context-free algebra with carriers defined by that grammar.* ■

Proof Let

$$X_1 = \text{pol}_1.(X_1, \dots, X_n)$$

...

$$X_n = \text{pol}_1.(X_1, \dots, X_n)$$

be an equational grammar with the (unique) solution (L_1, \dots, L_n) . Assume that the polynomials of that grammar are expressed as unions of monomials. The corresponding algebra

$$\underline{\text{Alg}} = (\text{Sig}, \text{Car}, \text{Fun}, \text{car}, \text{fun}),$$

is defined in the following way:

- $\text{Sig} = (\text{Nc}, \text{Nf}, \text{ar}, \text{so})$
- $\text{Nc} = \{\text{cn}_1, \dots, \text{cn}_n\}$ — carriers' names are arbitrary, but the number of these names must be equal to the number of equations in the grammar,
- $\text{Nf} = \{\text{fn}_1, \dots, \text{fn}_m\}$ — function names are arbitrary, but the number of these names must be equal to the number of monomial occurrences in the grammar,
- ar and so are defined in that way, that they correspond to the arities and sorts of monomials in the grammar,
- $\text{Car} = \{L_1, \dots, L_n\}$,
- Fun — the set of all monomials in our grammar,
- $\text{car.cn}_i = L_i$ for $i = 1, \dots, n$

Notice now that every monomial in our grammar is (from the definition) a Chomsky's monomial (see Sec. 2.5), hence satisfies the equation:

$$\text{car.cn}_i(x_1, \dots, x_n) = \{s_1\} x_1 \dots \{s_k\} x_k \{s_{k+1}\}$$

This completes the definition of our algebra. Observe that the defined algebra is unique up to the names of carriers and constructors.

Now we have to show that the carriers of $\underline{\text{Alg}}$ are closed wrt all its constructors and that the algebra is reachable. For this proof see [27]. ■

Below is a simple example showing how to construct an algebra from a grammar. Consider the following grammar of a two-sorted language

$$\text{Number} = 1 \mid x \mid \text{Number} + \text{Number}$$

$$\text{Boolean} = \text{Number} < \text{Number} \mid \text{Boolean} \& \text{Boolean}$$

For simplicity, curly brackets for function names have been dropped. The operations of our grammar are defined by the following equations (the symbols of concatenation $\textcircled{\cdot}$ has been omitted as well) where $n\text{-exp}$ and $b\text{-exp}$ with indexes denote numerical and boolean expressions, respectively:

$$\text{one.}() = 1$$

$$\text{variable.}() = x$$

$$\text{plus.}(n\text{-exp}_1, n\text{-exp}_2) = n\text{-exp}_1 + n\text{-exp}_2$$

$$\text{less.}(n\text{-exp}_1, n\text{-exp}_2) = n\text{-exp}_1 < n\text{-exp}_2$$

$$\text{and.}(b\text{-exp}_1, b\text{-exp}_2) = b\text{-exp}_1 \ \& \ b\text{-exp}_2$$

An equational grammar is said to be *unambiguous* (resp. *ambiguous*) if the corresponding algebra is unambiguous (resp. ambiguous). Intuitively a grammar is ambiguous if there exists a word w that can be generated by that grammars in two different ways³⁷. These “different ways” are different elements of the abstract syntax that are coimages of w wrt the abstract-syntax semantics (see Sec. 2.12). For instance, the word $1+1+1$ may be generated in two different ways:

$$\text{plus}(1, \text{plus}(1, 1))$$

$$\text{plus}(\text{plus}(1, 1), 1)$$

As has been already mentioned, a concrete syntax of a programming language will be constructed as a homomorphic image of its abstract syntax. Since these syntaxes will be described by equational grammars, it is important to know which homomorphisms of syntactic algebras do not lead out of the class of context-free algebras.

Let us start with an example of a homomorphism that destroys the context-freeness of an algebra. Let Alg be a one-sorted algebra with the carrier $\{a\}^+$ and with two operations:

$$h.() = a$$

$$f.(x) = x a$$

This algebra is of course, context-free. Now consider a similar algebra with a carrier

$$\{a^n b^n c^n \mid n = 1, 2, \dots\}$$

and constructors

$$h.() = abc$$

$$f.(a^n b^n c^n) = a^{n+1} b^{n+1} c^{n+1}$$

This algebra is not context-free since its carrier is a well-known example of a not context-free language (see [49]), but it is isomorphic with our former algebra where the corresponding isomorphism is:

$$l.a^n = a^n b^n c^n \quad \text{for every } n \geq 1$$

As is easy to see this isomorphism is not a skeleton function.

A homomorphism H between two syntactic algebras is called a *skeleton homomorphism* (I recall that since syntactic algebra are reachable, such a homomorphism, if exists, is unique (Theorem 2.12-3)) if for every constructor fun.fn of the source algebra, for which

$$\text{so.fn} = cn$$

$$\text{ar.fn} = (cn_1, \dots, cn_n)$$

there exists a skeleton (s_1, \dots, s_{n+1}) , such that

$$H.\text{fn.}(\text{fun}_1.\text{fn.}(x_1, \dots, x_n)) = s_1 x_1 \dots s_n x_n s_{n+1}$$

In other words, a homomorphic image of every constructor of the source algebra is a skeleton constructor in the target algebra.

Theorem 2.14-3 For every syntactic algebra Alg the following facts are equivalent:

³⁷ The usability of ambiguous grammars also from the perspective of parsing was investigated in 1972 by A.V. Aho and J.D. Ullman in [3].

- (1) $\underline{\text{Alg}}$ is context-free,
- (2) every homomorphism into $\underline{\text{Alg}}$ is a skeleton homomorphism,
- (3) there exists a skeleton homomorphism into $\underline{\text{Alg}}$.

For proof, see [27].

Let us consider now a simple example of a process of constructing a syntactic algebra for a given algebra³⁸. Let the latter be a one-sorted algebra of numbers with three operations:

```

create-nu.1  :                               ↦ Number
plus        : Number x Number ↦ Number
times       : Number x Number ↦ Number

```

The corresponding abstract syntax, denote it by Syn-0 , is defined by the following grammar with only one equation, where Exp denotes a language of numerical expressions with constant values:

$$\text{Exp} = \text{create-nu.1.} () \mid \text{plus} (\text{Exp}, \text{Exp}) \mid \text{times} (\text{Exp}, \text{Exp})$$

The first step on our way to final syntax consists in:

- replacing `create-nu.1` by `1`,
- replacing `plus` and `times` by `+` and `*`,
- replacing prefix notation by infix notation.

This step corresponds to the following homomorphism:

$$\begin{aligned} \text{H}[\text{create-nu.1.} ()] &= 1 \\ \text{H}[\text{plus} (\text{exp}_1, \text{exp}_2)] &= (\text{H}[\text{exp}_1] + \text{H}[\text{exp}_2]) \\ \text{H}[\text{times} (\text{exp}_1, \text{exp}_2)] &= (\text{H}[\text{exp}_1] * \text{H}[\text{exp}_2]) \end{aligned}$$

This is of course a skeleton homomorphism and the corresponding context-free grammar is the following:

$$\text{Exp} = 1 \mid (\text{Exp} + \text{Exp}) \mid (\text{Exp} * \text{Exp})$$

In the second and the last step of syntax construction we would like to allow dropping out “unnecessary parentheses”, e.g. writing `1+1+1` instead of `(1+(1+1))` and analogously for multiplication. Unfortunately this turns out to be impossible since each homomorphism which removes parentheses has to satisfy the equations:

$$\begin{aligned} \text{H}[(\text{exp}_1 + \text{exp}_2)] &= \text{H}[\text{exp}_1] + \text{H}[\text{exp}_2] \\ \text{H}[(\text{exp}_1 * \text{exp}_2)] &= \text{H}[\text{exp}_1] * \text{H}[\text{exp}_2] \end{aligned}$$

but this would mean that it glues expressions with different denotations, e.g.

$$\text{H}[(1+1) * (1+1)] = \text{H}[((1+(1*1)) +1)] = 1+1*1+1$$

Although H is a skeleton homomorphism, which implies that its target grammar

$$\text{Exp} = 1 \mid \text{Exp} + \text{Exp} \mid \text{Exp} * \text{Exp}$$

is context-free, the corresponding algebra is more ambiguous than the algebra of numbers, hence a denotational semantics of this syntax into the algebra of numbers does not exist.

A known traditional way of solving this problem as e.g. in Algol ([5] and [71]) or in Pascal [56] consists in reconstructing the whole model of the language by introducing to the algebra of denotations and to the algebra of syntax three carriers **Com** (component), **Fac** (factor) and **Exp** (expression) and the following signature:

³⁸ In more general terms such processes will be discussed in Sec. 3.5.

c-to-e	: Com	\mapsto Exp	<i>component to expression identically</i>
+	: Exp + Com	\mapsto Exp	<i>addition</i>
f-to-c	: Fac	\mapsto Com	<i>factor to component identically</i>
*	: Fac * Com	\mapsto Com	<i>multiplication</i>
1	: Fac	\mapsto Fac	<i>the generation of 1 as a factor</i>
e-to-c	: Exp	\mapsto Fac	<i>expression to factor identically</i>

The corresponding grammar of abstract syntax is the following:

$$\begin{aligned} \text{Exp} &= \text{c-to-e}(\text{Com}) \mid +(\text{Exp}, \text{Com}) \\ \text{Com} &= \text{f-to-c}(\text{Fac}) \mid *(\text{Fac}, \text{Com}) \\ \text{Fac} &= 1 \mid (\text{Exp}) \end{aligned}$$

and for the first (isomorphic) transformed syntax:

$$\begin{aligned} \text{Exp} &= (\text{Com}) \mid (\text{Exp} + \text{Com}) \\ \text{Com} &= (\text{Fac}) \mid (\text{Fac} * \text{Com}) \\ \text{Fac} &= 1 \mid (\text{Exp}) \end{aligned}$$

In this grammar names of identity functions have been omitted, which, however, does not destroy the unambiguity of the grammar, since these names appear in elements of different carriers.

Now we can define a skeleton homomorphism that removes parentheses in each of three sorts of expressions:

$$\begin{aligned} E.[(\text{com})] &= \text{com} \\ E.[(\text{com} + \text{exp})] &= E.[\text{exp}] + S.[\text{com}] \\ C.[(\text{fac})] &= C.[\text{fac}] \\ C.[(\text{fac} * \text{com})] &= F.[\text{fac}] * C.[\text{com}] \\ F.[1] &= 1 \\ F.[(\text{exp})] &= (\text{exp}) \end{aligned}$$

This leads to the following context-free grammar

$$\begin{aligned} \text{Exp} &= \text{Com} \mid \text{Exp} + \text{Com} \\ \text{Com} &= \text{Fac} \mid \text{Fac} * \text{Com} \\ \text{Fac} &= 1 \mid (\text{Exp}) \end{aligned}$$

This grammar may be also written in a direct way in using the constructor of iteration:

$$\begin{aligned} \text{Exp} &= \text{Com} [+ \text{Com}]^* && \text{an expression is a sum of components} \\ \text{Com} &= \text{Fac} [* \text{Fac}]^* && \text{a component is a multiplication of factors}^{39} \\ \text{Fac} &= 1 \mid (\text{Exp}) && \text{a factor is a constant or an expression in parentheses} \end{aligned}$$

Observe that the parentheses-removal homomorphism is not an isomorphism, since it glues $(1+(1+))$ and $((1+1)+1)$ into $1+1+1$ and similarly for multiplication. However it does not glue “to much” since addition and multiplication are associative. On the other hand from expression $((1+1) * (1+1))$ it removes only external parentheses.

The denotational homomorphism for our grammar is now the following:

$$\begin{aligned} \text{Se}.[\text{com}] &= \text{Ss}.[\text{com}] \\ \text{Se}.[\text{exp} + \text{com}] &= \text{Se}.[\text{exp}] + \text{Sc}.[\text{com}] \\ \text{Ss}.[\text{fac}] &= \text{Sc}.[\text{fac}] \\ \text{Ss}.[\text{fac} * \text{com}] &= \text{Sc}.[\text{fac}] * \text{Ss}.[\text{com}] \end{aligned}$$

³⁹ Note the difference between the operation of multiplication $*$, e.g. as in $1*1$ and the operation of the iteration of languages $*$, e.g. as in $[+ \text{Com}]^*$.

$$\begin{aligned} \text{Sc.}[1] &= 1 \\ \text{Sc.}[(\text{exp})] &= \text{Se.}[\text{exp}] \end{aligned}$$

Notice that the above equations express the school rules of priority of multiplication over addition.

Commentary 2.14-1

The reader to whom I have promised that denotational models of programming languages will offer readable definitions may have some doubts at this moment. So far, the simple language of arithmetic expressions that is very well known to every ground-school student has been described in a rather complicated way and moreover using advanced mathematics. This, of course, requires a commentary.

First, what we can say to a student in a simple way, when “talking” to a computer, we have to express in a way appropriate for the interpreter. That “appropriate way” is a denotational homomorphism, which may be mapped one-to-one into a code of an interpreter.

Second, the discussed language serves only to illustrate the denotational method in an elementary example. The real advantage of the method will be appreciated (I hope) when we introduce more advanced programming mechanisms such as declarations, types, instructions, recursive procedures, objects, etc. whose definitions require advanced mathematical tools.

Third, in writing a user’s manual for our language, we may directly refer to our acquaintance with school mathematics by saying that numerical expressions can be written and are calculated in a “usual way”, which frees us from the necessity of showing a grammar. However, as we shall see in Sec. 3.5 there are better solutions to that problem called *colloquial syntax*.

Two following lessons may be learned from our exercise:

First, the description of the simple operation of dropping out unnecessary parentheses requires rather complicated and not very intuitive grammar. Such a grammar is necessary for the implementor but not for the user, who can be simply informed that numerical expressions are written and understood in a “usual way”.

Second, the idea of dropping parentheses came out only at the level of second syntactic algebra, when the two formers have already been defined. Therefore, to implement the parenthesis-free notation one has to restart the construction of the model from scratch. In our simple example, this does not lead to too much work, but in real situations, things may look different. To avoid such problems, one should think about syntax as early as on the level of the algebra of denotations. This, however, contradicts the philosophy “from denotations to syntax” and also ruins the principle that denotations should be constructed in a maximally simple way.

The above problems had been investigated in [25], [27], and [34]. A solution suggested there consists in assuming that the programmer’s syntax that will be called colloquial syntax does not need to be a homomorphic image of concrete syntax. In our example concrete syntax would be defined by the grammar:

$$\text{Exp} = 1 \mid (\text{Exp} + \text{Exp}) \mid (\text{Exp} * \text{Exp})$$

and colloquial syntax — which allows for (although it does not force) the omission of parentheses — would be defined by the grammar:

$$\text{Exp} = 1 \mid (\text{Exp} + \text{Exp}) \mid (\text{Exp} * \text{Exp}) \mid \text{Exp} + \text{Exp} \mid \text{Exp} * \text{Exp}$$

Observe that the algebra of colloquial syntax is not only not-homomorphic to the former but is even not similar since it has a different signature.

Note, however, that it is easy to define a transformation that would map our colloquial syntax “back” into concrete syntax by adding the “missing” parentheses. Such a transformation will be called a *restoring transformation*. In practice, this approach leads to a user manual that contains a formal definition of concrete syntax (a grammar) plus an informal rule which says, e.g., that parentheses may be omitted in the “usual way”⁴⁰.

⁴⁰ As we are going to see in Sec.4.5.3 the situation may a little more complicated.

In the general case, a restoring transformation may be described formally or informally according to the complexity of colloquialization. Its formal definition is, however, always necessary for implementors who have to write a procedure that converts each colloquial program into its concrete version.

More on colloquial syntax as such in Sec. 3.5, and on colloquialisms in **Lingua** in Sec. Sec. 4.5.3, 5.2.3, 6.8.3, and 10.11.

In the end, one methodological remark seems necessary. Languages discussed in this section covered only expressions without variables. Such a case has, of course, no practical value, and it was chosen only to make examples of algebras and corresponding grammars possibly simple. Starting from Sec. 3 we shall discuss methods of constructing denotational models for more realistic languages.

3 GENERAL REMARKS ABOUT DENOTATIONAL MODELS

This section introduces the reader to the general theory of denotational models based on abstract algebras. Later on, this model is used in the construction of two layers of a virtual programming language **Lingua**:

1. *an applicative layer* covering datalogical and typological expressions whose denotations are state-to-values and state-to-type functions, respectively,
2. *an imperative layer* covering instructions and declarations whose denotations are state-to-state functions.

3.1 How did it happen?

Mathematicians working on mathematical models of programming languages were usually assuming (as in mathematical logic) that a programming language should be described by three mathematical objects:

1. **Syn** — *syntax*, which in our model is a context-free syntactic algebra,
2. **Den** — *denotations*, which in our model is an algebra similar (the same signature, see Sec. 2.11) to the algebra of syntax,
3. **Sem** : **Syn** \mapsto **Den** — *semantics*, that associates denotations to syntactic elements and in our model is a many-sorted homomorphism between two mentioned algebras.

Intuitively speaking, a denotational semantics describes the meaning of every complex syntactic object as a composition of the meanings of its components. This property of semantics — called *compositionality* — allows for the description of complex objects by means of so-called *structural induction*.

It should be mentioned at this point that denotational (compositional) models of semantics — which for mathematicians have always been an obvious choice — have not been used in the first formal models of programming languages. Similarly to the prototypes of sewing machines that were mechanical arms repeated the movements of a tailor, and to the first steamboat engine driving oars, the early formal definitions of a programming language were the descriptions of a virtual computer executing programs⁴¹.

This model of semantics, called later *operational semantics*, was abandoned after a few years of experiments because the description of the virtual machine was not less complex than the code of a real compiler, and still it was not a description of a “real” machine⁴².

⁴¹ First metalanguage used to write such semantics in the 1970. was developed in IBM laboratory Vienna and was called Vienna Definition Language (VDL). Later some members of the IBM team have created a lab on the Danish Technical University in Lyngby with the aim of writing a denotational semantics in a metalanguage called Vienna Development Method (VDM) [13]. This language was used, among other applications, to describe the semantics of two programming languages Ada and Chill. In the case of the former, which was expected to become a universal programming language of all times, the process of writing its semantics resulted in repairing many inaccuracies of the language, and in developing first Ada compiler. Unfortunately, both Chill and Ada were excessively complex, and hence have been fairly quickly forgotten.

⁴² To be precise this remark is true for sequential programming only (without concurrent processes), i.e. such that we shall deal with in this book. An operational semantics for concurrent programs was developed by Plotkin [72].

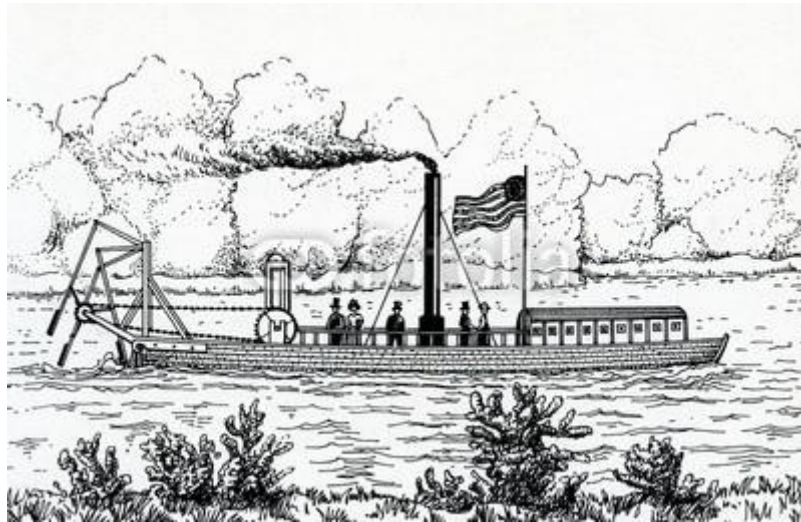


Fig. 3.1-1 Steamboat moving oars

The road to denotational semantics was, however, not simple either. As was already mentioned, early denotational models of programming languages were characterized by great mathematical complexity. Technically this was the consequence of the assumption that two following mechanisms were indispensable in high-level programming languages:

1. the jump instruction **goto** that transfers program execution from one line of code to another one; this mechanism was available in practically all programming languages in the years 1960/70, and was inherited from low-level languages, where it was the only tool for building logical structures of programs,
2. procedures that may take themselves as parameters; this construction was present in Algol 60 (see [5] and [71]) considered by academic community of 1960. as an indisputable standard.

The requirement of having **goto**'s has led to a technically rather complex model of *continuations*⁴³. That semantics was not only technically complex but above all quite far from programmers' intuition. Independently, at the turn of the 1960-ties to 1970-ties, IT professionals began to be aware of a risk imposed by **goto** instruction (see [43]). Programs with **goto**'s were difficult to understand, and therefore not always behave as expected by programmers. As a consequence **goto**'s were eliminated in favor of structured programming mechanisms (see Sec.7.2) such as **if-the-else**, **while** and similar.

The continuation model, although technically complex, was based on traditional mathematics. This cannot be said about the model of procedures that take themselves as parameters. Here we are not talking about recursive procedures that call themselves in their bodies — such procedures can be modeled by fixed-point equations (see Sec.6) — but about constructions of type $f(f)$, where a function takes itself as an argument. Such functions were not known to mathematicians, because they cannot be described on the ground of classical set theory, let alone that mathematicians never needed such functions.

In Algol 60 the construction $f(f)$ was implemented in such a way, that a procedure f was receiving as a parameter not exactly itself, but a copy of its own code, which was inserted into its body during compilation. Such an operation was called *copy rule*. Mathematicians of the decade of 1960. were fascinated by this construction because it was challenging the existing concept of a function. As a consequence, the theory of *reflexive domains* was created by Dana Scott and Christopher Strachey [75] and was later described in detail by J.E. Stoy in a monograph [74]⁴⁴. Although some mathematicians were investigating reflexive domains, for

⁴³ First author who introduced that concept — although under a different name of *tail functions* — was Antoni Mazurkiewicz [61]. Under the name of continuations it was introduced in [75] and later and popularized in [53].

⁴⁴ To my colleagues mathematicians I may explain that the idea of reflexive domains was in fact a "hidden realization" of copy rule. The authors of this model used the fact that functions definable by programs are computable, hence can be "numbered" with natural numbers — each function f may be given a unique number $n(f)$. In this model $f(f)$ meant

software engineers, they were even more difficult and less intuitive than continuations. Pretty soon it turned out also that the ability of “uploading” procedures to themselves as parameters lead to even greater dangers in programming than the use of `goto`'s. Consequently, in later programming languages, self-applicable procedures were abandoned. Unfortunately, some researchers decided that denotational semantics should be left as well.

The denotational model introduced in this book uses neither continuations nor reflexive domains.

In our model, the denotations of instructions are state-to-state functions where a state includes a component called a *valuation*. The concept of valuation has been well-known to mathematicians since the pioneering work of Alfred Tarski [76]. In those times the meanings of expressions were described as functions mapping valuations of variables

$$v : \text{Valuation} = \{x, y, z\} \rightarrow \text{Value}$$

into values. E.g., the meaning of an expression

$$2x+4y$$

is a function

$$F[2x+4y] : \text{Valuation} \rightarrow \text{Number}$$

such that

$$F[2x+4y].v = 2 * v(x) + 4 * v(y)$$

From there only one step to an observation that the meaning of an instruction

$$x := 2x + 4y$$

is such a transformation of valuations that the value of x in the new valuation is the value of the expression $2x+4y$ in the former. This idea was applied in my paper [17], published in 1971, where I described a prototype of a denotational semantics of a very simple programming language.

In turn, the inspiration to abandon the model of reflexive domains came to me from the book of Michael Gordon [53], where the author treats Scott's reflexive domains as “usual sets” with the following commentary on page 29:

“We shall not discuss the mathematics involved in Scott's theory at all; our approach to recursive equations⁴⁵ is similar to an engineering approach to differential equations, namely we assume they have solutions but don't bother with the mathematical justification.”

I have read Gordon's book in the year 1981 during a train ride from Copenhagen to Århus, where I was going to meet Peter Mosses, a strong proponent of the theory of Dana Scott. The book was, for me, a significant breakthrough since, for the first time, I was reading a semantics of a programming language with understanding not only its mathematics but also its IT content. The treatment of reflexive domains as “usual sets” was a real simplification. I also had the impression that this informal treatment did not lead to any mathematical problems. Only later, I realized that Gordon was actually not dealing with self-applicable functions.

The approach of Michael Gordon, although intuitively simple, was mathematically not quite acceptable since the assumption that reflexive domains are usual sets is simply not true. It wasn't, therefore, quite clear if his model did not lead to inconsistencies, which are undoubtedly critical when building a model to develop a logic of programs.

To cope with this problem, Andrzej Tarlecki and I published in 1983 a paper [34], in which we constructed a denotational model of a programming language, where the domains of denotations are sets, and the denotations of instructions are state-to-state transformations. This approach stimulated in 1980-ties the creation of a

$f(n(f))$ which can be modelled on the ground of classical set theory. That was in fact a mathematical application of copy rule since $n(f)$ may be regarded as the code of procedure f .

⁴⁵ M.Gordon is talking here about recursive domain-equations, which, in some case of non-continuous domain operators, lead to D.Scott's reflexive domains.

metalanguage **MetaSoft** [24] in the Institute of Computer Science of the Polish Academy of Sciences. And this is the approach that I have chosen to write the present book.

3.2 From denotations to syntax

All early works on the semantics of programming languages were devoted to building semantics for existing languages. That has led to a tacit assumption that in designing a language, the syntax should come first into the play. Of course, there is a certain logic in this way of thinking, since how can we build a model for something that does not yet exist? After all, astronomers were describing the mechanics of celestial bodies when the Sun and the planet were already there.

This way of thinking has, however, a particular vulnerability, since computer science cannot be compared to astronomy, physics, or biology, where we describe the world around us. Building a programming language is an engineering task, such as constructing a bridge or an airplane. Would any engineer ever think of first constructing a bridge basing on common sense and only then making all necessary calculations? Such a bridge would certainly collapse (cf. Sec. 1.1).

In my approach, I reverse the traditional order where we first build syntax and only later define the denotations. I will show how to build a language starting from an algebra of denotation from which syntax is derived then in such a way that a denotational semantics always exists.

A sample programming language built in this book is called **Lingua**. I have chosen this name to commemorate the circumstances under which, from October to December 1969, I wrote my first denotational semantics of a very simple programming language. This work was later published in *Dissertationes Mathematicae* [17] as my postdoctoral thesis. By three months as a scholar of the Italian Government, I was working in the Istituto di Elaborazione dell'Informazione in Pisa. I didn't yet know the works of Dana Scott or the concept of denotational semantics, and I constructed my language and its semantics on a model theory known in mathematical logic. Only eighteen years later, in the year 1987, I described (in [25]) the idea of deriving syntax from denotations.

3.3 Languages of the Lingua family

As has been announced in Sec.3.2, the method of building a denotational model of a programming language will be illustrated on the example of a virtual language **Lingua**. This language will be constructed layer-by-layer, starting with applicative mechanisms and enriching them by successive imperative constructions. Each successive layer will constitute an enrichment of the former by new tools:

Lingua-A	an applicative part of the future language including datalogical and typological expressions hence the models of data and of types,
Lingua-1	assignments, structural instructions, declarations of variables and definitions of types,
Lingua-2	imperative procedures with mutual recursion and functional procedures with simple recursion,
LinguaV-2	tools for building correct (validated) programs in Lingua-2 ,
Lingua-SQL	an application programming interface (API) for SQL databases,
Lingua-OO	object-oriented programming.

From an algebraic perspective, the algebra of denotations of each language from **Lingua-1** to **Lingua-SQL** will be an extension (in the sense as defined in Sec. 2.11) of the preceding algebra. In other words, each of the corresponding languages will be constructed from the former by adding new elements to the existing carriers, and/or new carriers, and/or new constructors. This scalability of algebras should lead to the scalability of possible implementations.

The model of **Lingua-OO** will differ from the former models by a more general concept of a state. The other constructions will be analogous.

In this place, I should emphasize that **Lingua** is not regarded as a future standard of a denotation-based language but only as a platform of experiments on which a possible future standard could be built.

3.4 Why do we need denotational models of programming languages?

A denotational model of a programming language serves as a starting point for the realization of three tasks:

1. building the implementation of the language, i.e., its parser and interpreter or compiler,
2. creating rules of building correct specified programs in this language,
3. writing a user manual.

When designing a language in this way, we should observe one fundamental (although not quite formal) principle:

The principle of simplicity

A programming language should be as simple to understand and easy to use as possible, although without damaging its functionality, mathematical clarity, and completeness of its description. The same applies to the manual of the language and to the rules of building correct programs.

This principle shall be fulfilled by:

1. making the syntax of the language as close as possible to the language of “usual” mathematics, e.g., whenever it is common, we allow infix notation and the omission of “unnecessary” parentheses,
2. making the structure of the language (i.e., program constructors) leading to possibly simple rules of constructing correct programs (Sec. 7 and Sec. 8),
3. making the semantics of the language easy to understand by the user rather than convenient for the implementor; for the latter, an equivalent implementation-oriented model may be written.

Particular attention should be given to point 2 because the simplicity of the rules of building correct programs leads to a better understanding of programs by programmers. This fact was realized already in the years 1970 and has led to the elimination of **goto** instructions. This decision resulted in a significant simplification of program structures, which increased their reliability. On the other hand, it did not limit the functionality of programming languages.

Following point 3, we will sometimes — as typical in mathematics — “forget” about the difference between syntax and denotations. E.g., we will talk about the value of an arithmetic expression $x + y$, rather than about the value generated by its denotation. We will say that the instruction $x := y + 1$ modifies variable x , instead of saying that the denotation of this instruction modifies the memory state at variable x , etc. Of course, at the model’s level, we shall precisely distinguish syntax from denotations.

3.5 Five steps to a denotational model

Building up **Lingua**, we refer to an algebraic model described in Sec. 2.10 to Sec. 2.14. This model corresponds to the diagram of three algebras shown in Fig. 3.5-1. We build it in such a way that the equation:

$$As = Co \bullet Cs$$

is satisfied, which guarantees the existence of a denotational semantics of our language.

The construction of a denotational model begins with an algebra of denotation Den. Its constructors unambiguously determine the reachable subalgebra ReDen. Now, from the signature of Den we unambiguously derive the *abstract syntax algebra* AbsSy. The first of these steps is creative since it comprises all the significant decisions about future language. Contrary to it, the second step can be performed algorithmically.

Since abstract syntax is usually not very convenient for programmers, we build a *concrete syntax* ConSy. In typical situations, we do it by replacing prefix notation by infix notation and introducing more intuitive names of constructors. In that case the corresponding homomorphism Co (*concretization*) is usually an isomorphism, and therefore there exist a unique homomorphism:

$$Cs : \underline{\text{ConSy}} \mapsto \underline{\text{ReDen}}$$

(*concrete semantics*), which is the semantics of concrete syntax. In this way, we create the main components of our denotational model.

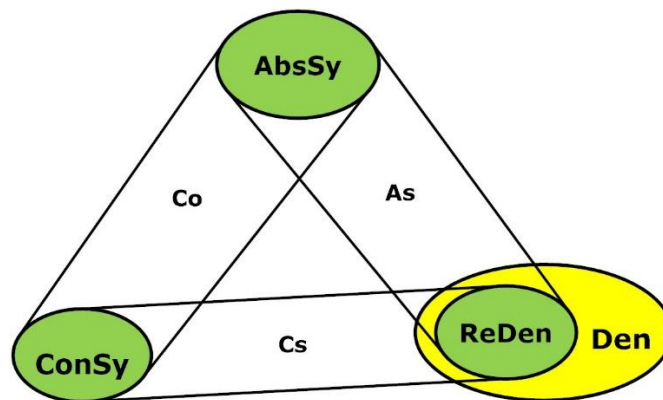


Fig. 3.5-1 An algebraic model of a programming language

The step from abstract syntax to concrete syntax is creative — although rather simple. For instance, instead of writing $+(a, b)$ we write $(a+b)$ and instead of writing $\text{if}(x>0, x:=x+1, x:=x-1)$ we write

```
if x>0 then x:=x+1 else x:=x-1 fi
```

The next step in building a user-friendly syntax consist in introducing so called *colloquialisms*. For instance instead of writing

```
(a+ (b+ (c*d) )
```

we shall write

```
a + b + c*d
```

assuming that multiplication binds stronger than addition. The introduction of colloquialisms into concrete syntax leads to *colloquial syntax* ColSy (Fig. 3.5-2), which most frequently has a different signature than concrete syntax, and therefore cannot be a homomorphic image of it. However, we make sur that there exists an implementable transformation

$$Rt : \underline{\text{ColSy}} \mapsto \underline{\text{ConSy}}$$

which removes colloquialisms, e.g., by adding the missing parentheses. Such a transformation is called a *restoring transformation*.

In a programmer's manual of a language with colloquialisms, concrete-syntax is defined by an equational grammar, and colloquialisms may be described informally. For instance, we explain that in writing arithmetic expressions, we can skip parentheses while maintaining the priority of multiplication and division over addition and subtraction.

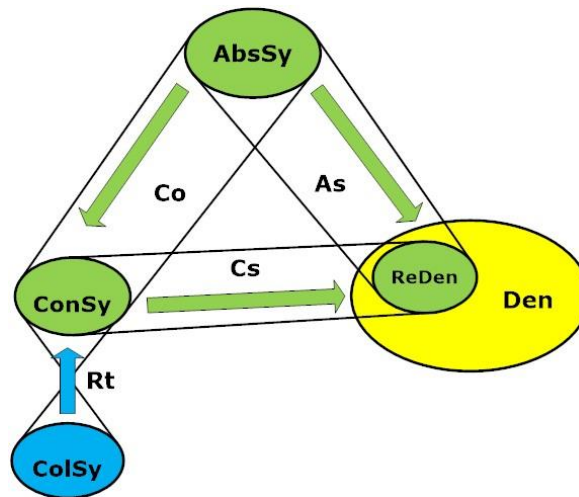


Fig. 3.5-2 An algebraic model of a language with colloquial syntax

In such a case, an implementor receives a standard denotational model of a language plus a formal definition (algorithm) of restoring transformation. The execution of a program consists then of two steps:

1. a pre-treatment of the source code by a restoring transformation,
2. an interpretation or compilation of concrete-syntax code.

Summing up our considerations, the construction of a denotational model of a programming language proceeds in five steps:

1. In the first step, we build an algebra of denotations Den that includes the denotations of the future syntax as well as their constructors. In that step, significant decisions are taken about the functionality of the language. A language designer must specify the repertoire of constructors of Den in such a way that the corresponding (unique) reachable subalgebra ReDen contains all the objects that we want to access through syntax. This will be illustrated on examples starting from Sec. 4.
2. The signature of algebra Den uniquely determines the algebra of abstract syntax AbsSy and the corresponding homomorphism (abstract semantics) As. The step from the signature of Den to AbsSy can be performed in a fully algorithmic way. From the perspective of a language designer, this step does not require any creativity and may be performed by a software tool.
3. The abstract syntax is usually not very user-friendly since it is restricted to prefix notation. We, therefore, build a concrete syntax ConSy, which is closer to programmers' syntax. Algebraically it is an isomorphic image of the abstract syntax which guarantees the existence of a denotational semantics (a homomorphism) $Cs : \underline{ConSy} \mapsto \underline{Den}$.
4. Now, the description of Cs can be algorithmically generated from the descriptions of As and Co.
5. In the last step, we introduce colloquialisms and describe the restoring transformation. This step is creative again.

As we see, the creative tasks of a language designer correspond to the first, third, and fifth steps. The second and fourth steps can be performed algorithmically.

Here a methodological remark is in order. Although Theorem 2.13-1 allows us to build ambiguous concrete syntaxes, in practice, we shall not use this opportunity. It turns out that practically it is much easier to construct an unambiguous concrete syntax, and then "shift" all ambiguities to the level of colloquial syntax, rather than to prove that the ambiguities are admissible. Notice that if we build an ambiguous syntax, then — similarly to the case of colloquial syntax — we have to define some sort of a restoring transformation that removes the ambiguities.

After having built a denotation model of a language, one can proceed to the definitions of correct-program constructors (Sec.8). This step corresponds to a historic task of developing programs' logic in Hoare's style.

In our case, however, we shall deal with a logic stronger than that of Hoare, since dealing with a so-called *clean termination*.

3.6 Notational conventions of our metalanguage

In the description of our sample language **Lingua**, we deal with three conceptual levels each associated with different fonts (cf. Sec. 2.12):

1. at the level of the concrete and colloquial syntax of **Lingua** we use `Courier New`,
2. at the level of the formal definition of our model, we use `Arial` with notational conventions coming from **MetaSoft** (Sec. 3.1),
3. at the level of informal descriptions and comments, we use `Times New Roman`.

Indices, which in traditional mathematics are written with reduced font size and at a lowered level like a_i , will be treated as arguments of functions by writing `a.i` or `a-i`, where `a` is regarded as a function and `i` — as its argument.

Due to a great variety of symbols occurring in the models of languages, instead of using one-character symbols as in usual mathematics, like `a`, `b`, `c`, `α`, `β`, `γ`... we use many-character symbols such as `ide`, `sta`, `sto`,... which, in turn, is a common technique in writing programs.

The names of sets always start with a capital letter, for example, `Number` or `InsDen` (instruction denotations) and the names of their elements with small letters.

Following the convention used in VDM (the Vienna Development Method; see [14]) the metavariables that run over domains are “announced” in the definitions of domains by writing, e.g.:

`ide` : Identifier = Letter © Character*

`val` : Valuation = Identifier \Rightarrow Data,

which means that `ide` runs over `Identifier` and `val` over `Valuation`. At the end of the book, there is a list of the most frequently used alphanumeric symbols.

As has been mentioned already in Sec. 2.8, the values that are strings of characters will be closed in apostrophes to distinguish them from metavariables. E.g., `ide` is a metavariable that runs over the domain of identifiers, and `'abcd'` is a particular word which consists of four letters.

In order to shorten certain conditional definitions of functions which in full version are written as a list of clauses:

`condition-1` \rightarrow `value-1`

...

`condition-n` \rightarrow `value-n`

we shall also allow a compact version:

`condition-i` \rightarrow `value-i` for $i = 1;n$

This will be clear later when it comes to examples.

4 LINGUA-A — AN APPLICATIVE LAYER OF LINGUA

4.1 Lingua as a strongly-typed language

In a manual of SQL ([46] p. 786), we can read the following sentence⁴⁶:

If we do not provide (...) correct values to functions, we should not expect consistent results.

Contrary to that philosophy, **Lingua** will be constructed in such a way that whenever a program will “send” incorrect values to a function, this function will generate an error message and/or initiate a recovery action. To achieve that goal, we equip **Lingua** with a type mechanism where:

1. A type of data describes the structure of that data, e.g., that it is a boolean, a number or an array, and possibly also some other properties of data, e.g., that each element of an array is an integer from an interval [0,100].
2. Each variable declared in a program has a fixed type assigned to it. This type is never changed during the execution of the program⁴⁷, and all data assigned to that variable must be of that type.
3. Programs operate on values that are triples including: a data, a description of its structure (called *body*), and a description of its other properties (called *yoke*). Values are assigned to identifiers in memory states, are passed to procedures as values of actual parameters (both value parameters and reference parameters), and when we evaluate an expression, we get a value as a result.
4. Type checking precedes the following actions:
 - a. assigning a value to a variable,
 - b. applying an operation to its arguments (to values),
 - c. passing actual parameters to a procedure or a function,
 - d. returning formal reference parameters from a procedure.
5. Types are stand-alone mathematical beings (rather than sets of data), but each type defines a unique set of data of that type called the *clan of the type*.
6. Types and their constructors constitute an algebra of types, which provides tools for the construction of user-defined types. At the language level we have type expressions that evaluate to types, and type declarations that are used to name types, i.e. assign them to identifiers, and save them in memory states for subsequent use.

In the end, it should be reemphasized that **Lingua-A**, which we are going to construct in Sec. 4, is not regarded as a prototype of a stand-alone applicative programming language, but only as an example of an applicative part of an imperative language. In **Lingua-A**, we have a mechanism of expression evaluation but not of state transformation (declarations and instructions). The latter mechanisms, called imperative, will be introduced in **Lingua-1** (Sec. 5).

⁴⁶ My own translations from a Polish edition of this book.

⁴⁷ This condition will be weakened in **Lingua-SQL** where we have operations that add (or remove) an attribute in a row or in a table (Sec. 10).

4.2 The general idea of the model of types

In early programming languages such as Fortran, Algol 60, Pascal, or Cobol, the concept of a data type was introduced in the first place to allocate appropriate memory space to variables. With boolean variables, single-bit registers were assigned, with numeric variables — many-bit registers and finally with array variables — a larger memory space depending on the size of an array⁴⁸. Over time it turned out, however, that assigning types to variables allows not only for better management of memory space but also contributes to a better understanding of the functionality of programs both by programmers and analysts, and allows one to capture several run time errors at the compile time.

Today, when memory management is no longer so critical (except in some special applications, e.g., databases), this second aspect remains relevant. It results not only for a better understanding of program behavior but also in the avoidance of type errors such as, e.g., sending inappropriate actual-parameters to a procedure, or inappropriate arguments to an operation, or assigning an inappropriate value to a variable. The realization of all these goals requires a type-tracking mechanism activated whenever we evaluate an expression or execute an instruction.

In the denotational model of **Lingua**, the type-tracking mechanism is implemented by assuming that variables are assigned in states to triples called *values*, which include three elements:

- data* — such as numbers, booleans, words, lists, arrays, records, etc.
- body* — which is a finitistic object describing the structure of a data, e.g., that it is a number, a list or a record⁴⁹,
- yoke* — which describes other properties of data, e.g., that all elements of an array of numbers must belong to the interval [0,100].

Every value (*data*, *body*, *yoke*) generated during program execution must be *well-typed* by which we mean that *data* must have the structure described by *body* and the pair (*data*, *body*), which we call a *composite* must satisfy *yoke*.

Bodies and types uniquely determine sets of data — with that body or of that type, respectively. This idea is formalized in the subsequent sections.

Our model of types allows programmers to build complex types step-by-step in a bottom-up way as in the following example (here we anticipate the future syntax of **Lingua**):

```
set years_register_type as
  array-type number where all-of-ar 2000 ≤ value ≤ 2100 ee
tes ;
```

This is a type declaration which assigns a type to an identifier (a type constant), in this case `years_register_type`. The declared type is a type of one-dimensional arrays of numbers (*body*), which belong to the interval [2000, 2100] (*yoke*).

Next type declaration defines a record type with four attributes `ch_name`, `fa_name`, `birthyear`, `award_years`, where the type of `award_years` is the predefined type `years_register_type`. The

⁴⁸ During the first course of programming in my university studies in the year 1960 we were writing programs for the first Polish computer named XYZ which was constructed in the Department of Mathematical Apparatuses of the Polish Academy of Sciences. The only memory of that computer was a RAM with the capacity of 1 K. Magnetic memories (types or discs) were not known at these times yet.

⁴⁹ Some inspiration for the introduction of this model was for me the idea used in the definition of programming language Ada written in a metalanguage VDM (see [14] and [15]). In that case however there were two semantics: a static semantics to compute types, and a dynamic semantics to compute data. The former was describing a type-checking mechanism activated at compile time, the latter — program execution. Such a model can be convenient for the implementor of a language, but seems rather far from programmer's perspective which we are trying to stick to in this book.

yoke of the new type is assumed to be trivial (always true), and therefore the clause **where** does not appear in the declaration.

```

set employee_type as
  record-type
    ch_name, fa_name of type word
    birthyear       of type integer,
    award_years     of type years_register_type
  ee
tes;

```

Having declared our two types, we now may declare two variables of these types:

```

let smith be employee_type ee
let awards_smith be years_register_type ee

```

4.3 From data to values

4.3.1 Data

The first phase in designing a programming language usually consists in two steps:

1. determining data that the future language will manipulate,
2. determining operations on these data — we shall call them *primary constructors* — that will be available in the language.

In a denotational framework data domains are defined by domain equations, whose solutions are usually larger than the sets of (reachable) data generated during the executions of future programs.

In turn, primary constructors constitute a base for future constructors of expression denotations. On a purely abstract ground they may be assumed to be “given ahead”, but on a practical ground they must be defined by using operations available on some implementation platform. Here we choose the latter perspective based on an exercise of developing an experimental implementation of **Lingua** by the listeners of a course that I gave at the Department of Mathematics, Informatics, and Mechanics of Warsaw University in the year 2020 (see [33]). Our implementation platform in this case was Objective Caml (OCaml) (see [37], and [61]).

Let us start with the domain of identifiers. This domain is not going to be a domain of data, but we shall need it in the definition of a domain of records. Let then

```

ide : Identifier = Letter (Letter | Digit | { _ })*   with len.ide ≤ 32
let : Letter    = {A, ..., Z, a, ..., z}
dig : Digit     = {0, 1, ..., 9}

```

An identifier is a finite string of letters, digits, and underscores, which starts with a letter and contains not more than 32 characters. Here **len.ide** denotes the number of characters in **ide**.

In defining a domain of identifiers we exclude from it some reserved keywords that will be used later in our syntax such as e.g., *true*, *false*, *list*, *push*, *while* etc. Let’s assume that this has been done.

Next auxiliary domain that we define is the domain of characters that we shall need in defining the domain of words. Let

```

car : Character = Letter | Digit | Sign
sig : Sign      = { +, -, *, @, _ }

```

In this place the elements of **Sign** are just typical examples. We assume also that it contains all punctuation marks, but does not contain apostrophes.

Now, we are ready to define our domains of data. Here again we follow the mentioned implementation of OCaml. Our reals correspond to OCaml's floating-point numbers.

```

int  : Integer    =  $[-2^{30}, 2^{30}-1]$ 
rea  : Real       =  $[-1,8 \times 10^{308}, 1,8 \times 10^{308}]$ 
boo  : Boolean    = {tt, ff}
wor  : Word       = {'}Character*{'}    with len.wor  $\leq 2^{24} - 5$ 
lis  : List       = Datac*
arr  : Array      = Integer  $\Rightarrow$  Data
rec  : Record     = Identifier  $\Rightarrow$  Data
dat  : Data       = Integer | Real | Word | List | Array | Record

```

A word is a string (possibly empty) of the elements of `Character` closed with apostrophes. The empty word is therefore `''`.

A list is a finite (possibly empty) sequence of arbitrary data.

Arrays and records are mappings, i.e., finite functions. Arrays are one-dimensional, but since their elements can be arrays, the domain `Array` contains arrays of arbitrary dimensions. Identifiers that appear in records will be called *record attributes*.

Lists, arrays, and records constitute *structured domains*, and their elements are called *structured data*. They may be tuples (lists) or mappings (arrays and records).

All domains which are defined above, except `Data` and `Identifier`, will be referred to as *data sorts*, e.g., *integer sort*, *word sort*, *array sort*, etc. At this stage list and arrays may be not-homogeneous, i.e., may include elements of different sorts. Further, domains of indices of arrays may be arbitrary finite sets of integers, rather than (as usual) sets of consecutive integers. Moreover, all our structured data may be "arbitrarily large".

Such "oversized" domains have been defined to make their definitions expressible by simple domain equations. Later the constructors of our algebras will assure that all reachable data will have "appropriate structures". This technique of defining "too large" domains whose reachable parts are appropriately "truncated" is typical for denotational models and will be frequently used in the sequel of the book. As we already know from Sec. 2.12, only reachable elements of algebras have their representations in syntax.

Having defined a family of data domains, we have to indicate a certain initial set of operations on data that we shall call *primary constructors*. As we have already mentioned, they must be definable by using constructors available on the implementation platform. Let's assume the following list of primary constructors:

Families of zero-argument constructors (constants)

```

create-id.ide  :  $\mapsto$  Identifier          for all ide : Identifier
create-in.int  :  $\mapsto$  Integer             for all int : IntegerS
create-re.rea  :  $\mapsto$  Real                for all rea : RealS
create-wo.wor  :  $\mapsto$  Word                for all wor : WordS

```

For instance:

```

create-id.size.() = size    where size : Identifier
create-in.127.() = 127

```

The presence of zero-argument constructors in our model means that programmers in **Lingua** may “enter from the keyboard” syntactic representations of identifiers and of the elements of **IntegerS**, **RealS**, and **WordS**. E.g. they may type 3 instead of $((1+1) +1)$.

The domains **IntegerS**, **RealS**, and **WordS** are subsets of **Integer**, **Real**, and **Word**, respectively, with syntactically representable elements. What are these sets will depend on the implementation of **Lingua**. At the general level, we only assume that they are finite. We don't assume, however, but also do not rule out, that the remaining constructors of data will generate data only from these sets.

The domain of identifiers contains, by definition, only the elements that can be typed from the keyboard, and therefore the suffix "S" is not needed.

In order to define the remaining constructors of data we introduce a certain universal set of abstract errors (cf. Sec. 2.8) that we denote by **Error** and assume that it contains all error messages (words) that may be generated during the executions of our programs. With every domain **Domain** we assign a corresponding domain that includes errors:

$$\text{DomainE} = \text{Domain} \mid \text{Error}$$

Now, the signatures of the remaining primary constructors of data are the following.

Comparison constructors

equal	: DataE x DataE	↦ BooleanE
less	: DataE x DataE	↦ BooleanE

Here we do not introduce logical connectives **and**, **or** and **not** in the domain **BooleanE** since, at the level of data, we do not define operations on booleans. They come only at the level of expression denotations in Sec. 4.4.2, where we use McCarthy's propositional calculus.

Integer number constructors

add-in	: IntegerE x IntegerE	↦ IntegerE
subtract-in	: IntegerE x IntegerE	↦ IntegerE
multiply-in	: IntegerE x IntegerE	↦ IntegerE
divide-in	: IntegerE x IntegerE	↦ RealE

Real number constructors

add-re	: RealE x RealE	↦ RealE
subtract-re	: RealE x RealE	↦ RealE
multiply-re	: RealE x RealE	↦ RealE
divide-re	: RealE x RealE	↦ RealE

Word constructors

glue	: WordE x WordE	↦ WordE
------	-----------------	---------

List constructors

create-li	: DataE	↦ ListE
push	: DataE x ListE	↦ ListE
top	: ListE	↦ DataE
pop	: ListE	↦ ListE

Array constructors

create-ar	: DataE	↦ ArrayE
put-to-ar	: DataE x ArrayE	↦ ArrayE

change-in-ar : ArrayE x IntegerE x DataE \mapsto ArrayE replace an element of an array
 get-from-ar : ArrayE x IntegerE \mapsto ArrayE

Record constructors

create-re : Identifier x DataE \mapsto RecordE
 put-to-re : DataE x RecordE x Identifier \mapsto RecordE
 get-from-re : RecordE x Identifier \mapsto DataE
 change-in-re : RecordE x Identifier x DataE \mapsto RecordE replace an element of a record

In order to define primary constructors, we need an auxiliary concept. As we know, many of these constructors cannot be applied to arbitrary arguments from their domains. E.g., we can't divide a number by zero, we can't add two numbers if their sum would be too large, or we cannot select the 11th element from an array whose range of indices is $[1, \dots, 10]$. In all such cases, we should protect **Lingua** operations from calling the corresponding operation of the implementation platform. To describe such a mechanism, for every primary constructor, e.g. an integer division

divide-in : IntegerE x IntegerE \rightarrow IntegerE

we define a function called *trust test*

trust.divide-in : IntegerE x IntegerE \mapsto Error | {'OK'}

such that whenever this test yields 'OK', the constructor yields a correct result, and otherwise it generates an appropriate error messages. For instance:

trust.divide-in.(int-1, int-2) =
 int-i : Error \rightarrow int-i for $i = 1, 2$
 int-2 = 0 \rightarrow 'division-by-zero'
not (int-1 / int-2) : $[-2^{30}, 2^{30}-1]$ \rightarrow 'overflow'
true \rightarrow 'OK'

where int/1 / int-2 denotes mathematical quotient of int-1 by int-2. Of course, the predicate

(int-1 / int-2) : $[-2^{30}, 2^{30}-1]$

must be implemented in such a way that it does not need to perform int-1 / int-2.

We assume here that all our trust tests will be transparent for errors, i.e., whenever one of their arguments is an error, then — whichever comes first on the way from left to right — this error becomes the result of the test. Using our test, we can define the primary operation of division as follows:

divide-in.(int-1, int-2) =
 trust.divide-in.(int-1, int-2) : Error \rightarrow trust.divide-in.(int-1, int-2)
true \rightarrow divide-in-IP.(int-1, int-2)

where divide-IP is the denotation of division provided by the implementation platform. Most frequently the IP-division of integers will return a real as a result. If, however, we want to have a division of integers that rounds the result to its integer part, we have add some rounding operation, and set:

divide-in-2-in.(int-1, int-2) =
 trust.divide-in.(int-1, int-2) : Error \rightarrow trust.divide-in.(int-1, int-2)
true \rightarrow round-2-in.(divide-in-IP.(int-1, int-2))

In these examples primary divisions may be regarded as a functional procedures that call the implementation-platform (IP) division. In the general case, however, all that we have to assume is that primary operations are definable in the terms of trust tests and some IP operations. E.g. we might define our primary operations on lists by calling IP operations on tuples. Here we skip formal definitions of primary operations assuming that they are the parameters of our model.

In the end, we have to explain why propositional connectives **and**, **or**, **not** have been not included among primary constructors. This is the consequence of the fact that at the level of data-expression denotations (Sec. 4.4.2), and of yokes (Sec. 4.4.4) the logical connectives have to satisfy McCarthy's philosophy of lazy evaluation (Sec. 2.9), and therefore we have to define them in each such level separately. This will be clear when we proceed to denotations.

4.3.2 Bodies

Having defined our domains of data we may proceed to *bodies*. As has been already said, bodies describe "internal structures" of data, and are used in building types. Bodies are defined as tuples, mappings (records), and their combinations. The domains of bodies are "derivative" to the corresponding domains of data.

$\text{bod} : \text{SimBody} = \{ ('boolean'), ('integer'), ('real'), ('word') \}$	simple bodies
$\text{bod} : \text{LisBody} = \{ 'L' \} \times \text{Body}$	list bodies
$\text{bod} : \text{ArrBody} = \{ 'A' \} \times \text{Body}$	array bodies
$\text{bod} : \text{RecBody} = \{ 'R' \} \times (\text{Identifier} \Rightarrow \text{Body})$	record bodies
$\text{bod} : \text{Body} = \text{SimBody} \mid \text{LisBody} \mid \text{ArrBody} \mid \text{RecBody}$	(4.3.2-1)
$\text{bod} : \text{BodyE} = \text{Body} \mid \text{Error}$	

Bodies of simple data are one-element tuples of words. Symbols 'L', 'A' and 'R' are called *body initials* and indicate sorts of structured bodies. E.g. ('A', ('integer')) is a body of arrays of integers, and ('L', ('A', ('real')))) is a body of lists, whose elements are arrays of reals.

In the case of a list-body ('L', bod) we say that bod is the *inner body* of the list-body and similarly for array-bodies. The elements of the domain

$$\text{bor} : \text{BodRec} = \text{Identifier} \Rightarrow \text{Body}$$

are called *body records*. Hence every record-body is of the form ('R', bor), where bor is a body record.

The definitions of body domains anticipate the principle that all elements of a list or of an array must have a common body.

Notice that an array body does not specify the number of array elements. The introduction of arrays with a fixed number of elements will be possible with the help of yokes (see Sec. 4.3.4).

To associate data with bodies, we assign to each body a set of data called the *clan* of this body. Formally, we define a function CLAN-Bo that with each body assigns its clan:

$$\text{CLAN-Bo} : \text{Body} \mapsto \text{Sub.Data}$$

This function is defined by structural induction

$$\begin{aligned} \text{CLAN-Bo.('boolean')} &= \text{Boolean} \\ \text{CLAN-Bo.('integer')} &= \text{Integer} \\ \text{CLAN-Bo.('real')} &= \text{Real} \\ \text{CLAN-Bo.('word')} &= \text{Word} \\ \text{CLAN-Bo.('L', bod)} &= (\text{CLAN-Bo.bod})^c \\ \text{CLAN-Bo.('A', bod)} &= \text{Integer} \Rightarrow \text{CLAN-Bo.bod} \\ \text{CLAN-Bo.('R', [ide-1/bod-1, \dots, ide-n/bod-n])} &= \\ &\{ [ide-1/dat-1, \dots, ide-n/dat-n] \mid \text{dat-}i : \text{CLAN-Bo.bod-}i \text{ for } i = 1;n \} \end{aligned}$$

As we see, the union of all clans does not exhaust the domain **Data**, which means that some data have no bodies. For example, a non-homogeneous list of numbers mixed with words, and booleans, like (123, 'abc', tt), has no body. In this way, using bodies we restrict the set of reachable data. Details will be seen later.

We shall also need an auxiliary function:

$$\text{sort-b} : \text{BodyE} \mapsto \{('boolean'), ('integer'), ('real'), ('word'), 'L', 'A', 'R'\} \mid \text{Error}$$

$\text{sort-b.bod} =$

$\text{bod} : \text{Error}$	$\rightarrow \text{bod}$
$\text{bod} = ('boolean')$	$\rightarrow ('boolean')$
$\text{bod} = ('integer')$	$\rightarrow ('integer')$
$\text{bod} = ('real')$	$\rightarrow ('real')$
$\text{bod} = ('word')$	$\rightarrow ('word')$
$\text{bod} : \{ 'L' \} \times \text{Body}$	$\rightarrow 'L'$
$\text{bod} : \{ 'A' \} \times \text{Body}$	$\rightarrow 'A'$
$\text{bod} : \{ 'R' \} \times (\text{Identifier} \Rightarrow \text{Body})$	$\rightarrow 'R'$

Since bodies in composites should always coincide with corresponding data, whenever we perform an operation on data, we have to perform “in parallel” a corresponding operation of their bodies. These operations become constructors of an algebra of bodies AlgBod. We assume that this algebra has two following carriers:

$\text{ide} : \text{Identifier}$ — defined in Sec. 4.3.1

$\text{bod} : \text{BodyE} = \text{Body} \mid \text{Error}$

Now, with every primary constructor we associate a body constructor. To show this association explicitly, with every name pco of a primary constructor, we assign a name bo-pco of a body constructor.

Zero-argument constructors

$\text{create-id.ide} : \mapsto \text{Identifier}$ for all $\text{ide} : \text{Identifier}$

In the case of identifiers, we skip the prefix bo- since the same (meta) constructor of identifiers will appear in many subsequently defined algebras.

$\text{bo-create-bo} : \mapsto \text{Body}$

$\text{bo-create-in} : \mapsto \text{Body}$

$\text{bo-create-re} : \mapsto \text{Body}$

$\text{bo-create-wo} : \mapsto \text{Body}$

Comparison constructors

$\text{bo-equal} : \text{BodyE} \times \text{BodyE} \mapsto \text{BodyE}$

$\text{bo-less} : \text{BodyE} \times \text{BodyE} \mapsto \text{BodyE}$

Arithmetic constructors for integers

$\text{bo-add-in} : \text{BodyE} \times \text{BodyE} \mapsto \text{BodyE}$

$\text{bo-subtract-in} : \text{BodyE} \times \text{BodyE} \mapsto \text{BodyE}$

$\text{bo-multiply-in} : \text{BodyE} \times \text{BodyE} \mapsto \text{BodyE}$

$\text{bo-divide-in} : \text{BodyE} \times \text{BodyE} \mapsto \text{BodyE}$

Arithmetic constructors for reals

$\text{bo-add-re} : \text{BodyE} \times \text{BodyE} \mapsto \text{BodyE}$

$\text{bo-subtract-re} : \text{BodyE} \times \text{BodyE} \mapsto \text{BodyE}$

$\text{bo-multiply-re} : \text{BodyE} \times \text{BodyE} \mapsto \text{BodyE}$

bo-divide-re : BodyE x BodyE \mapsto BodyE

Word constructors

bo-glue : BodyE x BodyE \mapsto BodyE

List constructors

bo-create-li : BodyE \mapsto BodyE

bo-push : BodyE x BodyE \mapsto BodyE

bo-top : BodyE \mapsto BodyE

bo-pop : BodyE \mapsto BodyE

Array constructors

bo-create-ar : BodyE \mapsto BodyE

bo-put-to-ar : BodyE x BodyE \mapsto BodyE

bo-check-in-ar : BodyE x BodyE \mapsto BodyE

bo-get-from-ar : BodyE x BodyE \mapsto BodyE

Record constructors

bo-create-re : Identifier x BodyE \mapsto BodyE

bo-put-to-re : BodyE x BodyE x Identifier \mapsto BodyE

bo-get-from-re : BodyE x Identifier \mapsto BodyE

bo-check-in-re : BodyE x Identifier x BodyE \mapsto BodyE

Below a few definitions as examples:

bo-create-id.ide.() = ide for all ide : Identifier

bo-create-boo.() = ('boolean')

bo-create-int.() = ('integer')

bo-equal.(bod-1, bod-2) =

bod-i : Error	\rightarrow bod-i	for i = 1,2
bod-1 \neq bod-2	\rightarrow 'compared-bodies-must-coincide'	
not comparable.bod-1	\rightarrow 'not-comparable'	
true	\rightarrow ('boolean')	

Here we introduce a metapredicate **comparable** to express the fact that only some data will be comparable with each other. E.g., numbers and words will be, but lists, arrays, and records will not.

bo-divide-in.(bod-1, bod-2) =

bod-i : Error	\rightarrow bod-i	for i = 1,2
bod-i \neq ('integer')	\rightarrow 'integer-expected'	for i = 1,2
true	\rightarrow ('number')	

bo-create-li.bod =

bod : Error	\rightarrow bod
true	\rightarrow ('L', bod)

```

bo-push.(bod-e, bod-l) =                                     push bod-e on list bod-l
  bod-i : Error      → bod-i                               for i = e,l
  sort-b.bod-l ≠ 'L' → 'list-expected'
  let
    ('L', bod) = bod-l
  bod-e ≠ bod      → 'conflict-of-bodies'
  true            → bod-l

bo-create-ar.bod =
  bod : Error      → bod
  true            → ('A', bod)

bo-put-to-ar.(bod-e, bod-a) =                               put bod-e to array bod-a
  bod-i : Error      → bod-i                               for i = a,e
  sort-b.bod-a ≠ 'A' → 'array-expected'
  let
    ('A', bod) = bod-a
  bod ≠ bod-e      → 'conflict-of-bodies'
  true            → bod-a

bo-create-rec.(ide, bod) =
  bod : Error      → bod
  true            → ('R', [ide/bod])

bo-put-to-rec.(bod-e, bod-r, ide) =                         put bod-e to record bod-r on attribute ide
  bod-i : Error      → bod-i
  sort-b.bod-r ≠ 'R' → 'record-expected'
  bod-r.ide = !      → 'attribute-already-exist'
  true            → ('R', bod-r[ide/bod-e])

bo-check-in-re.(bod-r, ide, bod-e) =                       check if new body coincides with the former
  bod-i : Error      → bod-i                               for i = r, e
  sort-b.bod-r ≠ 'R' → 'record-expected'
  let
    ('R', bod-rb) = bod-r                                 -rb for „record body”
  bod-rb.ide = ?    → 'no-such-attribute'
  let
    bod-ab = bod-rb.ide                                   -ab for “attribute body”
  bod-e ≠ bod-ab    → 'conflict-of-bodies'
  true            → bod-r

```

The last constructor anticipates the fact that if we replace a data assigned to a record attribute, the new data must have the same body as the former data. It will be used in Sec. 4.3.3 in the definition of a corresponding constructor of record composites.

In the end, in the class of body constructors we distinguish a subclass of *body-creating constructors* which include the following constructors:

```

bo-create-bo   :      ↦ BodyE
bo-create-in   :      ↦ BodyE
bo-create-re   :      ↦ BodyE
bo-create-wo   :      ↦ BodyE

```

bo-create-li	: BodyE	\mapsto BodyE
bo-create-ar	: BodyE	\mapsto BodyE
bo-create-re	: Identifier x BodyE	\mapsto BodyE
bo-put-to-re	: BodyE x BodyE x Identifier	\mapsto BodyE

These are all constructors that we shall refer to (call) in body expressions (Sec. 4.4.4), and consequently in body constant declarations (Sec. 5.1.4.2). The remaining constructors will be needed when we define the constructors of values (Sec. 4.3.6).

4.3.3 Composites

By a *composite*,⁵⁰ we shall mean a pair consisting of a data and its (!) body. We assume that such a body is not an error. The domain of composites is defined by the following equation::

$$\text{com} : \text{Composite} = \{(\text{dat}, \text{bod}) \mid \text{dat} : \text{CLAN-Bo.bod}\} \quad (4.3.3-1)$$

A composite (dat, bod) is said to *carry* the data dat and the body bod.

A composite that carries a simple data is called *simple composite* and analogously are understood *structured composites*. We shall also talk about *boolean composites*, *integer composites*, *list composites*, etc. Since boolean composites will play a special role, we introduce the corresponding domain:

$$\text{com} : \text{BooComposite} = \{(\text{boo}, ('boolean')) \mid \text{boo} : \{\text{tt}, \text{ff}\}\}$$

We also introduce two domains of composites and errors:

$$\begin{aligned} \text{com} : \text{CompositeE} &= \text{Composite} \quad | \quad \text{Error} \\ \text{com} : \text{BooCompositeE} &= \text{BooComposite} \quad | \quad \text{Error} \end{aligned}$$

Now we define an *algebra of composites* AlgCom with two carriers⁵¹:

$$\begin{aligned} \text{ide} : \text{Identifier} &\quad \text{— defined in Sec. 4.3.1} \\ \text{com} : \text{CompositeE} &\quad \text{— defined above} \end{aligned}$$

Now, we have to define constructors that will “call” the (already defined) constructors of data and bodies. To do that we expand the earlier introduced function `sort` (Sec. 4.3.1) onto composites and identifiers:

$$\begin{aligned} \text{sort-b}.\text{(dat, bod)} &= \text{sort-b.bod} \\ \text{sort-b.ide} &= \text{ide} \end{aligned}$$

Notice that for simple composites function `sort` coincides with `body`, but for structured composites, this is not the case. We also introduce two new selection functions:

$$\begin{aligned} \text{data}.\text{(dat, bod)} &= \text{dat} \\ \text{body}.\text{(dat, bod)} &= \text{bod} \\ \text{data.ide} &= \text{ide} \\ \text{body.ide} &= \text{ide} \end{aligned}$$

⁵⁰ In this place Andrzej Tarlecki asked a question, why I introduce bodies and composites. If every reachable data has a unique body, we could operate on data with implicitly assigned bodies. From a mathematical point of view that would be, of course, quite correct. However, I decided to show explicitly how the modification of data goes in parallel with the modification of their bodies. In this way I suggest a certain technique for the implementation of **Lingua**. This approach is also useful when we define types and type constructors (Sec. 4.3.5).

⁵¹ Why the algebra of composites has only two carriers whereas we defined many sorts of data, will be explained at the end of Sec. 5.1.4.1.

Now we can proceed to the constructors of AlgCom. For each primary constructor pco we define a composite constructor co-pco , that will refer to (call) pco and bo-pco . Note that all our constructors are total functions which is due to the fact that CompositeE includes abstract errors.

Zero-argument constructors

create-id.ide	$: \mapsto \text{Identifier}$	for all ide	$: \text{Identifier}$ ⁵²
co-create-bo.bo	$: \mapsto \text{CompositeE}$	for all boo	$: \text{Boolean}$
co-create-in.int	$: \mapsto \text{CompositeE}$	for all int	$: \text{IntegerS}$
co-create-re.rea	$: \mapsto \text{CompositeE}$	for all rea	$: \text{RealS}$
co-create-wo.wor	$: \mapsto \text{CompositeE}$	for all wor	$: \text{Words}$

Comparison constructors

co-equal	$: \text{CompositeE} \times \text{CompositeE}$	$\mapsto \text{CompositeE}$
co-less	$: \text{CompositeE} \times \text{CompositeE}$	$\mapsto \text{CompositeE}$

Arithmetic constructors for integers

co-add-in	$: \text{CompositeE} \times \text{CompositeE}$	$\mapsto \text{CompositeE}$
co-subtract-in	$: \text{CompositeE} \times \text{CompositeE}$	$\mapsto \text{CompositeE}$
co-multiply-in	$: \text{CompositeE} \times \text{CompositeE}$	$\mapsto \text{CompositeE}$
co-divide-in	$: \text{CompositeE} \times \text{CompositeE}$	$\mapsto \text{CompositeE}$

Arithmetic constructors for reals

co-add-re	$: \text{CompositeE} \times \text{CompositeE}$	$\mapsto \text{CompositeE}$
co-subtract-re	$: \text{CompositeE} \times \text{CompositeE}$	$\mapsto \text{CompositeE}$
co-multiply-re	$: \text{CompositeE} \times \text{CompositeE}$	$\mapsto \text{CompositeE}$
co-divide-re	$: \text{CompositeE} \times \text{CompositeE}$	$\mapsto \text{CompositeE}$

Word constructors

co-glue	$: \text{CompositeE} \times \text{CompositeE}$	$\mapsto \text{CompositeE}$
------------------	--	-----------------------------

List constructors

co-create-li	$: \text{CompositeE}$	$\mapsto \text{CompositeE}$
co-push	$: \text{CompositeE} \times \text{CompositeE}$	$\mapsto \text{CompositeE}$
co-top	$: \text{CompositeE}$	$\mapsto \text{CompositeE}$
co-pop	$: \text{CompositeE}$	$\mapsto \text{CompositeE}$

Array constructors

co-create-ar	$: \text{CompositeE}$	$\mapsto \text{CompositeE}$
co-put-to-ar	$: \text{CompositeE} \times \text{CompositeE}$	$\mapsto \text{CompositeE}$
co-change-in-ar	$: \text{CompositeE} \times \text{CompositeE} \times \text{CompositeE}$	$\mapsto \text{CompositeE}$
co-get-from-ar	$: \text{CompositeE} \times \text{CompositeE}$	$\mapsto \text{CompositeE}$

⁵² See the note about create-id in Sec. 4.3.2.

Record constructors

co-create-re	: Identifier x CompositeE	↦ CompositeE
co-put-to-re	: CompositeE x CompositeE x Identifier	↦ CompositeE
co-get-from-re	: CompositeE x Identifier	↦ CompositeE
co-change-in-re	: CompositeE x Identifier x CompositeE	↦ CompositeE

Now let us show a few examples of definitions of these constructors. All of them will be transparent for errors (see Sec. 2.8). Let us also recall that the “outputs” of these constructors, whenever they are not errors, must be composites, i.e., such pairs (dat, bod) that $\text{dat} : \text{CLAN-Bo.bod}$.

co-create-id.ide.() = ide for all ide : Identifier
 co-create-bo.boob.() = (boob, ('boolean')) for both boob : Boolean
 co-create-in.int.() = (int, ('number')) for all int : IntegerS
 etc.

All the remaining constructors will be defined according to a common scheme:

1. check if the argument composites are not errors, and if they are not then,
2. compute the resulting body, and if no error is signaled then,
3. compute the resulting data, and if no error is signaled then,
4. combine body and data into composite.

If in 1, 2 or 3 an error is signaled, then this error becomes the final result. Let us illustrate this scheme by an example of a division of integers:

co-divide-in : CompositeE x CompositeE ↦ CompositeE
 co-divide-in.(com-1, com-2) =
 com-i : Error → com-i for i = 1, 2
let
 (dat-i, bod-i) = com-i for i = 1, 2
 bod = bo-divide-in.(bod-1, bod-2)
 bod : Error → bod
let
 dat = divide-in.(dat-1, dat-2)
 dat : Error → dat
true → (dat, (bod))

This definition should be read as follows:

- If one of the arguments is an error, then that error — whichever comes first — is returned as a result.

Formally this part of our definition should be written as

com-1 : Error → com-1
 com-2 : Error → com-2

- Otherwise, we compute the resulting body, and if it is an error (if one of the arguments is not ('integer'), then an appropriate message is generated.

- Otherwise, we compute the resulting data, and if it is an error, then an appropriate message is generated.
- Otherwise, we combine the resulting data with the resulting body into the final composite.

Note that in this definition, we refer to (call) two previously introduced operations — a body operation **bo-divide-re**, and a primary operation **divide-in**. First of them checks if the arguments of **divide-in** are integers. The second is responsible for all other checks.

Depending on our definition of **divide-in** (see Sec. 4.3.1) the body constructor **bo-divide-in** returns ('integer') or ('real').

A few other examples of composite-constructor definitions are the following. In each of them we refer to (call) one body constructor and one primary constructor:

co-create-li : CompositeE \mapsto CompositeE

```

co-create-li.com =
  com : Error                 $\rightarrow$  com
  let
    (dat, bod) = com
    bod-l      = bo-create-li.bod
  bod-l : Error               $\rightarrow$  bod-l
  let
    list = create-li.dat
  true                       $\rightarrow$  (list, bod-l)

```

Here, of course, **create-li.dat** = (dat), where (dat) denotes a one-element list whose only element is dat. As we see, we can make a list composite out of any composite independently of its body. However, the bodies of all future elements of that list must coincide with the body of the first element. This rule is expressed in the definition of the next constructor, where the rule is explicit in the definition of **bo-push** (Sec. 4.3.2)

co-push : CompositeE x CompositeE \mapsto CompositeE

```

co-push.(com-e, com-l) =
  com-i : Error                 $\rightarrow$  com-i    for i = e, l      push com-e on list com-l
  let
    (dat-i, bod-i) = com-i      for i = e, l
    bod            = bo-push.(bod-e, bod-l)
  let
    new-li = push.(dat-e, dat-l)
  true                       $\rightarrow$  (new-lis, bod)

```

With this operation we guarantee that all reachable list-composites will be homogeneous. In an analogous way we restrict the class of reachable array-composites.

co-create-ar : CompositeE \mapsto CompositeE

```

co-create-ar.com =
  com : Error                 $\rightarrow$  com
  let
    (dat, bod) = com
    bod-a      = bo-create-ar.bod
  bod-a : Error               $\rightarrow$  bod-a
  let
    arr = create-ar.dat
  true                       $\rightarrow$  (arr, bod-a)

```

If at level of data we assume that

create-ar.dat = [1/dat],

then the first index of an array will be always 1. In a similar way the operation of putting a new element at the end of an array should guarantee that the domain of every array is of the form $\{1, \dots, n\}$.

$\text{co-put-to-ar} : \text{CompositeE} \times \text{CompositeE} \mapsto \text{CompositeE}$

```

co-put-to-ar.(com-e, com-a) =
  com-i : Error           → com-i           for i = e, a
  let
    (dat-i, bod-i) = com-i           for i = e, a
    bod             = bo-put-to-ar.(bod-e, bod-a)
  bod : Error           → bod
  let
    new-ar = put-to-ar.(dat-e, dat-a)
  true                 → (new-arr, bod)

```

where we assume that at the level of data we have

```

new-ar   = dat-a[new-ind/dat-e]
new-ind  = max.(dom.dat-a) + 1

```

At the end one more definition which “inherits” a decision from the level of bodies:

$\text{co-change-in-re} : \text{CompositeE} \times \text{Identifier} \times \text{CompositeE} \mapsto \text{CompositeE}$

```

co-change-in-re.(com-r, ide, com-e) =
  com-i : Error           → com-i           for i = r, e
  let
    (dat-i, bod-i) = com-i           for i = r, e
    bod             = bo-check-in-re.(bod-r, ide, bod-e)
  bod : Error           → bod
  let
    new-rec = change-in-re.(dat-r, ide, dat-d)
  true                 → (new-rec, bod-r)

```

Here we assume that the corresponding data-constructor is the following:

```

change-in-re.(dat-r, ide, dat-d) = dat-r[ide/dat-d]

```

The inherited decision concerns the fact that if we assign new data to an attribute of a record, then the new body must be identical with the previous one. Consequently the body of the record does not change.

4.3.4 Yokes

The concept of a body allows expressing such properties of data, which in many programming languages exhaust the concept of a type, e.g., the type of booleans, numbers, lists, arrays, etc. Some languages, however, offer a higher expressiveness of types. For instance, in SQL (see Sec. 10), one may declare types of such tables (arrays of records), where a given column has no repetitions or types of such databases that satisfy a subordination relation between tables.

To ensure such types in the model of **Lingua**, we introduce a sort of⁵³ predicates on composites, which we shall call *yokes*. The domain of yokes is defined by:

$\text{yok} : \text{Yoke} = \text{CompositeE} \mapsto \text{BooCompositeE}$

By the *clan of a yoke*, we mean the set of composites that satisfy this yoke. Formally we define a function:

$\text{CLAN-Yo} : \text{Yoke} \mapsto \text{Sub.Composite}$

$\text{CLAN-Yo.yok} = \{(dat, bod) \mid dat : \text{CLAN-Bo.bod} \text{ and } yok.com = (tt, ('boolean'))\}$

⁵³ They are “sort of predicates” rather than just “predicates”, because their values are boolean composites rather than booleans.

Now we define an algebra of yokes AlgYok with three carriers:

`ide` : Identifier = ...
`tra` : Transfer = CompositeE \mapsto CompositeE
`yok` : Yoke = CompositeE \mapsto BooCompositeE

The carrier of *transfers* is necessary to make the domain of reachable yokes “sufficiently rich”. This will be seen in the examples shown below.

Notice that the algebra of yokes is “one level up” wrt composites since its elements are constructors of composites. There is also a particular singularity in our algebra since Yoke is a subset of Transfer.

Our algebra does not contain errors but contains yokes and transfers that may return errors as their values. We say that a composite `com` *satisfies a yoke* `yok` if

`yok.com = (tt, ('boolean'))`

Anticipating future concrete syntax of **Lingua** the yoke expression

`value < 10`

represents a yoke that is satisfied whenever the input composite carries a number, and that number is less than 10. In this expression, `value` is not a variable identifier, but a transfer expression which corresponds to an identity transfer `pass.()` (see later). Another example may be a yoke expression

`value + 2 < 10`

which is satisfied if the data carried by the current composite incremented by 2 is less than 10. Here `value+2` is a transfer expression. In turn, the yoke expression:

`record.salary + record.commission < 7000`

is satisfied if its argument composite carries a record with numeric attributes `salary` and `commission` whose sum is less than⁵⁴ 7000. Our three examples explain why we need transfers in our algebra.

All constructors of transfers will be defined in such a way that all reachable transfers will be transparent for errors, i.e., will satisfy the equation:

`tra.err = err` for every `err` : Error

Of course, the same rule concerns yokes, since yokes are just a particular case of transfers.

The majority of transfer- and yoke constructors will be derived from composite constructors, although not necessarily from all of them. Which composite constructors we bring to the level of transfers is, of course, an engineering decision. Here we assume that transfers may fully “elaborate” only numerical and boolean composites, whereas, for structured data, we shall make available only selection operations such as, e.g., `co-get-from-ar`. These are, of course, engineering decisions.

We may also have constructors that do not correspond to constructors of composites. As a matter of example, we introduce one new operation, and two new predicates on sequences of integers, which we shall use to build constructors of transfers and yokes respectively.

`sum-in` : Integer^{c+} \mapsto Integer the sum of Integers on a list
`unique` : Integer^{c+} \mapsto Boolean no repetition on a list
`increasing-in` : Integer^{c+} \mapsto Boolean increasingly ordered list of integers

By $Tc[cco]$ we denote a transfer constructor associated with a composite constructor `cco` from AlgCom. Below is the list of constructors of AlgYok split into six groups:

⁵⁴ From a pure mathematical viewpoint we could omit the keywords in the syntax of composites, e.g. writing simply „< 10” or „+2<10”, but such syntax would be rather far from intuition.

(1) Constructors of identifiers

create-id.ide : \mapsto Identifier for ide : Identifier

(2) Identity transfer

pass : \mapsto Transfer

(3) Constructors of transfers based on simple-composite operations

Here and in (4) and (5) $Tc[cco]$ denotes a constructor of transfers (including yokes, of course), which is derived from a composite constructor cco . The metaconstructor Tc (transfers constructor) is defined a little later.

$Tc[co-create-bo.bo]$:	\mapsto Yoke	for boo : Boolean
$Tc[co-create-in.int]$:	\mapsto Transfer	for int : IntegerS
$Tc[co-create-re.rea]$:	\mapsto Transfer	for rea : RealS
$Tc[co-create-wo.wor]$:	\mapsto Transfer	for wor : WordS

$Tc[co-add-in]$: Transfer x Transfer \mapsto Transfer

$Tc[co-subtract-in]$: Transfer x Transfer \mapsto Transfer

$Tc[co-multiply-in]$: Transfer x Transfer \mapsto Transfer

$Tc[co-divide-in]$: Transfer x Transfer \mapsto Transfer

$Tc[co-add-re]$: Transfer x Transfer \mapsto Transfer

$Tc[co-subtract-re]$: Transfer x Transfer \mapsto Transfer

$Tc[co-multiply-re]$: Transfer x Transfer \mapsto Transfer

$Tc[co-divide-re]$: Transfer x Transfer \mapsto Transfer

$Tc[co-glue]$: Transfer x Transfer \mapsto Transfer

(4) Constructors of transfers based on selection operations for list, arrays and records

$Tc[co-top]$: Transfer \mapsto Transfer

$Tc[co-get-from-ar]$: Transfer x Transfer \mapsto Transfer

$Tc[co-get-from-re]$: Transfer x Identifier \mapsto Transfer

(5) Constructors of transfers specific for transfers algebra

yo-sum : Transfer \mapsto Transfer

(6) Constructors of yokes based on predicates

$Tc[co-equal]$: Transfer x Transfer \mapsto Yoke

$Tc[co-less]$: Transfer x Transfer \mapsto Yoke

yo-unique : Transfer \mapsto Yoke
 yo-increasing-in : Transfer \mapsto Yoke

(7) Constructors of yokes based on Kleene's operators

yo-and : Yoke x Yoke \mapsto Yoke
 yo-or : Yoke x Yoke \mapsto Yoke
 yo-not : Yoke \mapsto Yoke
 all-of-li : Yoke \mapsto Yoke
 exists-in-li : Yoke \mapsto Yoke
 all-of-ar : Yoke \mapsto Yoke
 exists-in-ar : Yoke \mapsto Yoke

Let us now give a few examples of definitions of our constructors. The constructors of identifiers are well-known from the former algebra. The identity transfer maps a composite or error onto itself:

$\text{pass}().\text{com} = \text{com}$

It has a technical character which will be shown a little later. The metaconstructor Tc is defined by the equation:

$$\text{Tc}[\text{cco}].(\text{tra-1}, \dots, \text{tra-n}).\text{com} = \text{cco}.\text{com}(\text{tra-1}.\text{com}, \dots, \text{tra-n}.\text{com}) \quad (4.3-2)$$

where $n \geq 0$ and tra-i 's are either transfers or identifiers, and where we assume that $\text{ide}.\text{com} = \text{ide}$. For instance:

$$\begin{aligned} \text{Tc}[\text{co-get-from-ar}].(\text{tra-ar}, \text{tra-no}).\text{com} &= \text{co-get-from-ar}.\text{com}(\text{tra-ar}.\text{com}, \text{tra-no}.\text{com}) \\ \text{Tc}[\text{co-get-from-re}].(\text{tra-re}, \text{ide}).\text{com} &= \text{co-get-from-re}.\text{com}(\text{tra-re}.\text{com}, \text{ide}) \end{aligned}$$

In the first case, $\text{tra-ar}.\text{com}$ must be an array composite and $\text{tra-no}.\text{com}$ must be a number composite. Otherwise an error will be signaled from the cco level. Similar observations apply to the second case. Notice that in (4.3-2), if all tra-i are transparent, then

$$\text{Tc}[\text{dco}].(\text{tra-1}, \dots, \text{tra-n})$$

is transparent as well. The scheme (4.3-2) applies also to zero-argument constructors, e.g.,

$$\begin{aligned} \text{Tc}[\text{co-create-bo.tt}] : &\mapsto \text{Yoke} \\ \text{Tc}[\text{co-create-bo.tt}].().\text{com} = & \\ \text{com} : \text{Error} &\rightarrow \text{com} \\ \text{true} &\rightarrow (\text{tt}, (\text{"boolean"})) \end{aligned}$$

This yoke will be denoted by TT . Now consider the following examples of transfer expressions:

- A. 10 ,
- B. **value**,
- C. **value** + 2,
- D. **record**.salary + **record**.commission.

Their respective denotations are the following transfers (for simplicity in these examples we assume that com is not an error):

(A) $\text{Tc}[\text{co-create-in.10}].().\text{com} =$

```
co-create-in.10.() =
(10, ('integer'))
```

(B) `pass().com = com`

(C) `Tc[co-add-in].(pass.(), Tc[co-create-in.2]).com =`
`co-add-in.(pass.().com. Tc[co-create-in.2].com) =`
`co-add-in.(com, (2, ('integer')))`

Here we use `pass` to get two arguments for `co-add-in` out of one composite.

(D) `Tc[co-add-in].(Tc[co-get-from-re], (pass.(), salary),`
`Tc[co-get-from-re], (pass.(), commission)).com =`
`co-add-in.(co-get-from-re.(pass.(), salary).com,`
`co-get-from-re.(pass.(), commission).com) =`
`co-add-in.(co-get-from-re.(com, salary),`
`co-get-from-re.(com, commission))`

Note that by combining transfers generated by constructors of group (4), we may select data stored on an arbitrary depth of structured data. E.g., if `com` carries a list of records, then, to get an element assigned to attribute `age` in a record which is on the top of the list, we have to evaluate an expression whose denotation is the following:

```
Tc[co-get-from-re].(Tc[co-top].(pass.(), age).com =
co-get-from-re.(co-top.com, age)
```

The corresponding expression in concrete syntax will look as follows:

```
get-from-re(top(value), age)
```

The summation constructor (group (5)) is defined as follows:

```
yo-sum.tra.com =
com : Error      → com
let
  com-t = tra.com
com-t : Error    → com-t
let
  (dat-t, bod-t) = com-t
sort-b.bod-t ≠ 'L' → 'list-expected'
let
  ('L', bod-l) = bod-t
bob-l ≠ ('integer') → 'integers-expected'
let
  int = sum.dat-t
oversized.int      → 'overflow'
true              → (num, ('number'))
```

where `oversized` is a predicate that checks if the resulting sum is not too large for a current implementation. Constructors of group (6) are rather obvious.

Constructors of the last group (7) refer to Kleene's propositional calculus (see Sec. 2.9) rather than to that of McCarthy, as it will be the case for data expressions (Sec. 4.4.2). The conjunction of yokes is defined as follows:

```
yo-and : Yoke x Yoke  $\mapsto$  Yoke
yo-and.(yok-1, yok-2).com =
  com : Error            $\rightarrow$  com
  let
    com-i = yok-i.com           for i = 1, 2
    com-1 = (ff, ('boolean'))  $\rightarrow$  (ff, ('boolean'))
    com-2 = (ff, ('boolean'))  $\rightarrow$  (ff, ('boolean'))
    com-i : Error            $\rightarrow$  com-i           for i = 1, 2
    sort-b.com-i  $\neq$  ('boolean')  $\rightarrow$  'boolean expected'   for i = 1, 2
  true                        $\rightarrow$  (tt, ('boolean'))
```

As we see, to falsify this conjunction, is enough that at least one of its arguments carry `ff`. If this is not the case, then the result is either an error or a composite carrying `tt`. Constructor `yo-not` is the same as in McCarthy's case, and `yo-or` is defined in a way that guarantees the satisfaction of De Morgan's law.

As we are going to see, McCarthy's calculus will be assumed for data-expression denotations, because there — due to functional procedures (Sec. 6.5) — expressions may generate infinite executions. Since we do not allow functional procedures in transfers, we can assume a “more lazy” Kleene's calculus. This calculus has also been assumed in SQL standards (Sec. 10). In this calculus, alternative and conjunction are commutative (except where both arguments are errors), whereas, in the calculus of McCarthy's, they are not.

The general-quantifier constructors for lists and arrays are defined in the following way (also in Kleene's spirit):

```
all-of-li : Yoke  $\mapsto$  Yoke
all-of-li.yok.com =
  com : Error            $\rightarrow$  com
  sort-b.com  $\neq$  'L'      $\rightarrow$  'list expected'
  data.com = ( )          $\rightarrow$  (tt, ('boolean'))
  let
    (dat-1,...,dat-n) = data.com
    ('L', bod)       = body.com           list elements have all the same body
    com-i             = yok.(dat-i, bod)   for i = 1;n
    (there exists 1  $\leq$  i  $\leq$  n) com-i = (ff, ('boolean'))  $\rightarrow$  (ff, ('boolean'))
    (for all 1  $\leq$  i  $\leq$  n) com-i = (tt, ('boolean'))       $\rightarrow$  (tt, ('boolean'))
  true                        $\rightarrow$  'never-false'
```

This definition may be said to be consistent with the Kleene's definition of conjunction in the sense that

`ff and ee = ee and ff = ff`

The existential quantification is defined in an analogous way:

```
exists-in-li : Yoke  $\mapsto$  Yoke
exists-in-li.yok.com =
  com : Error            $\rightarrow$  com
  sort-b.com  $\neq$  'L'      $\rightarrow$  'list expected'
  data.com = ( )          $\rightarrow$  (ff, ('boolean'))
  let
```

$(\text{dat-1}, \dots, \text{dat-n}) = \text{data.com}$ $(\text{'L'}, \text{bod}) = \text{body.com}$ $\text{com-i} = \text{yok.}(\text{dat-i}, \text{bod})$ $(\text{there exists } 1 \leq i \leq n) \text{ com-i} = (\text{tt}, (\text{'boolean'})) \rightarrow (\text{tt}, (\text{'boolean'}))$ $(\text{for all } 1 \leq i \leq n) \text{ com-i} = (\text{ff}, (\text{'boolean'})) \rightarrow (\text{ff}, (\text{'boolean'}))$ true \rightarrow 'never-true'	list elements have all the same body for $i = 1;n$
--	---

Similarly this definition may be seen as consistent with the Kleene's alternative where

tt and $\text{ee} = \text{ee}$ and $\text{tt} = \text{tt}$

Quantifiers for arrays are defined in an analogous way. Why we assume Kleene's calculus for yokes, rather than the calculus of McCarthy, may be justified by an example of an array $\mathbf{a} = [1/0, 2/1]$ and a yoke (in an anticipated syntax):

$\text{exists-in-ar.}(1/(\mathbf{a}.i) > 0) \text{ iff } 1/0 > 0 \text{ or } 1/1 > 0$

In McCarthy's calculus, the value of this yoke would be an error, which is certainly not what we would expect.

4.3.5 Types

As was already said, *types* describe properties of data. This rule includes the description of their structures by bodies and the description of some structure-independent properties by yokes. The domain of types is defined in the following way:

$\text{typ} : \text{Type} = \text{Body} \times \text{Yoke}$

Types of the form (bod, TT) are called *yokeless*. We assume that all boolean types, i.e., types of the form $(\text{'boolean'}, \text{yok})$, will be yokeless. Only such boolean types will be generated in the course of type- and data-expression evaluation. With every type, we associate a set of data called the *clan of that type*. To do that we define a function:

$\text{CLAN-Ty} : \text{Type} \mapsto \text{Sub.Data}$

$\text{CLAN-Ty.}(\text{bod}, \text{yok}) = \{\text{dat} \mid \text{dat} : \text{CLAN-Bo.bod} \text{ and } (\text{dat}, \text{bod}) : \text{CLAN-Yo.yok}\}$

A type whose clan is empty will be called an *empty type*. Now we may define an *algebra of types* named AlgTyp with five carriers:

$\text{ide} : \text{Identifier} = \dots$

$\text{bod} : \text{BodyE} = \dots$

$\text{tra} : \text{Transfer} = \dots$

$\text{yok} : \text{Yoke} = \dots$

$\text{typ} : \text{TypeE} = \text{Type} \mid \text{Error}$

and five groups of constructors:

(1) Constructors of identifiers

$\text{create-id.ide} : \mapsto \text{Identifier} \quad \text{for ide} : \text{Identifier}$

(2) Selected constructors of bodies

$\text{bo-create-bo} : \mapsto \text{BodyE}$

$\text{bo-create-in} : \mapsto \text{BodyE}$

$\text{bo-create-wo} : \mapsto \text{BodyE}$

$\text{bo-create-li} : \text{BodyE} \mapsto \text{BodyE}$

$$\begin{aligned} \text{bo-create-ar} & : \text{BodyE} && \mapsto \text{BodyE} \\ \text{bo-create-re} & : \text{Identifier} \times \text{BodyE} && \mapsto \text{BodyE} \\ \text{bo-put-to-re} & : \text{BodyE} \times \text{BodyE} \times \text{Identifier} && \mapsto \text{BodyE} \end{aligned}$$

(3) All constructors of the algebra of yokes

See Sec. 4.3.4

(4) One type constructor

$$\text{create-ty} \quad : \text{BodyE} \times \text{Yoke} \mapsto \text{TypeE}$$

Constructors of groups (1), (2), and (3) are already known. In group (2), we include only these constructors, which build “new” bodies. E.g., we omit constructors of selection, and such constructors as `bo-add`, `bo-and`, `bo-push`. The only constructor which yields types is defined in the following way:

$$\begin{aligned} \text{create-ty}(\text{bod}, \text{yok}) & = \\ \text{bod} : \text{Error} & \rightarrow \text{bod} \\ \text{true} & \rightarrow (\text{bod}, \text{yok}) \end{aligned}$$

Note that this constructor can be used to build empty types. However, as we are going to see, the mechanisms of creating values (Sec. 4.3.6) or of assigning them to variables (Sec. 5.1.5.2) will raise error messages in such cases. E.g., if we try to push a value on a list whose type is empty, then the yoke of this list will generate an error when checking if it is satisfied for the pushed value.

One methodological comment is needed at the end. Originally (historically), yokes have been introduced in **Lingua** to build **Lingua-SQL** (Sec. 10), where yokes correspond to integrity constraints. It seems, however, quite reasonable to think of other applications of yokes. Imagine a situation where in an array of numbers we store results of some process of physical measurements. Suppose further that we are designing a software that should take action whenever a current measurement comes out of a specified fixed interval $[p, q]$. In that case, it would be enough to declare an array-type variable whose yoke describes the required condition.

4.3.6 Values

As was announced in Sec. 4.1, values are *well-typed* data of the form (dat, typ) where `dat` is of the type `typ`. The domain of values is, therefore, the following:

$$\text{val} : \text{Value} = \{(\text{dat}, \text{typ}) \mid \text{dat} : \text{CLAN-bo.typ}\}$$

We also define

$$\text{val} : \text{ValueE} = \text{Value} \mid \text{Error}$$

Values may also be regarded as triples $(\text{dat}, \text{bod}, \text{yok})$ or as pairs (com, yok) . We shall use all three forms according to the need. Values will play an important role in our model:

- they will be assigned to variables in memory states,
- expressions will evaluate to values,
- values will be passed to procedure calls as actual parameters and will be return as actual reference parameters.

Values of the form $(\text{dat}, \text{bod}, \text{TT})$ will be called *yokeless values*. Values whose composite is boolean will be called *boolean values*. Following our assumption about boolean types (Sec. 4.3.5) we shall make sure that only two boolean values will be generable by our programs:

$$(\text{tt}, (\text{'boolean'}, \text{TT})) \text{ and } (\text{ff}, (\text{'boolean'}, \text{TT})).$$

A value constructor

$$\text{va-con} : \text{ValIde-1} \times \dots \times \text{ValIde-n} \mapsto \text{ValueE}$$

where each ValIde-i is either ValueE or Identifier will be said to be *transparent for errors* if

$$\text{va-con}(\text{arg-1}, \dots, \text{arg-n}) = \text{arg-k}$$

whenever arg-k is the first argument from the left that belongs to Error . On values we expand the predicate checking the volume of a data:

$$\text{oversized} : \text{Value} \mapsto \{\text{tt}, \text{ff}\}$$

Now we may define the algebra of values AlgVal which will constitute a fundament for the future algebra of expression denotation. This algebra has four carriers:

$$\text{ide} : \text{Identifier} = \dots$$

$$\text{tra} : \text{Transfer} = \dots$$

$$\text{yok} : \text{Yoke} = \dots$$

$$\text{val} : \text{ValueE} = \dots$$

Constructors of this algebra may be split into three groups:

1. all constructors of yokes and transfers from the algebra of yokes,
2. value constructors derived from all composite constructors,
3. specific value constructors.

Constructors of the first group have been defined in Sec. 4.3.5. Constructors of the second group will “call” corresponding composite constructors but will also process yokes. For the latter reason, they can’t be — as was the case for transfers — defined with the use of one universal metaconstructor. Let us show the signatures of value constructors of categories 2., 3., and 4. (va- stands for “value”):

Zero-argument constructors

$$\text{create-id.ide} : \mapsto \text{Identifier} \quad \text{for all ide} : \text{Identifier}$$

$$\text{va-create-bo.bo} : \mapsto \text{ValueE} \quad \text{for all bo} : \text{Boolean}$$

$$\text{va-create-in.int} : \mapsto \text{ValueE} \quad \text{for all int} : \text{IntegerS}$$

$$\text{va-create-re.rea} : \mapsto \text{ValueE} \quad \text{for all rea} : \text{RealS}$$

$$\text{va-create-wo.wor} : \mapsto \text{ValueE} \quad \text{for all wor} : \text{WordS}$$

Comparison constructors

$$\text{va-equal} : \text{ValueE} \times \text{ValueE} \mapsto \text{ValueE}$$

$$\text{va-less} : \text{ValueE} \times \text{ValueE} \mapsto \text{ValueE}$$

Integer number constructors

$$\text{va-add-in} : \text{ValueE} \times \text{ValueE} \mapsto \text{ValueE}$$

$$\text{va-subtract-in} : \text{ValueE} \times \text{ValueE} \mapsto \text{ValueE}$$

$$\text{va-multiply-in} : \text{ValueE} \times \text{ValueE} \mapsto \text{ValueE}$$

$$\text{va-divide-in} : \text{ValueE} \times \text{ValueE} \mapsto \text{ValueE}$$

Real number constructors

$$\text{va-add-re} : \text{ValueE} \times \text{ValueE} \mapsto \text{ValueE}$$

$$\text{va-subtract-re} : \text{ValueE} \times \text{ValueE} \mapsto \text{ValueE}$$

$va\text{-multiply-re} : ValueE \times ValueE \mapsto ValueE$
 $va\text{-divide-re} : ValueE \times ValueE \mapsto ValueE$

Word constructor

$va\text{-glue} : ValueE \times ValueE \mapsto ValueE$

List constructors

$va\text{-create-li} : ValueE \mapsto ValueE$
 $va\text{-push} : ValueE \times ValueE \mapsto ValueE$
 $va\text{-top} : ValueE \mapsto ValueE$
 $va\text{-pop} : ValueE \mapsto ValueE$

Array constructors

$va\text{-create-ar} : ValueE \mapsto ValueE$
 $va\text{-put-to-ar} : ValueE \times ValueE \mapsto ValueE$
 $va\text{-change-in-ar} : ValueE \times ValueE \times ValueE \mapsto ValueE$
 $va\text{-get-from-ar} : ValueE \times ValueE \mapsto ValueE$

Record constructors

$va\text{-create-re} : Identifier \times ValueE \mapsto ValueE$
 $va\text{-put-to-re} : Identifier \times ValueE \times ValueE \mapsto ValueE$
 $va\text{-get-from-re} : ValueE \times Identifier \mapsto ValueE$
 $va\text{-change-in-re} : ValueE \times Identifier \times ValueE \mapsto ValueE$

About the constructors of numeric, word, and boolean constants, we shall assume that they build yokeless constants. For instance, for every $num : Number$ we set:

$va\text{-create-in.int.}() = (co\text{-create-in.int.}(), TT) = (int, ('integer'), TT)$

All remaining constructors will be transparent for errors and will be defined according to the following scheme: if

$cco : Comlde \times \dots \times Comlde \mapsto CompositeE$

is a composite constructor, then

$va\text{-cco} : ValldE \times \dots \times ValldE \mapsto ValueE$

(where $ValldE$ is either $ValueE$ or $Identifier$) the corresponding value constructor is defined according to the following (half formal) scheme:

$va\text{-cco}(\text{arg-1}, \dots, \text{arg-n}) =$
 $\text{arg-i} : Error \quad \rightarrow \text{arg-i} \quad \text{for } i = 1;n$
let
 $c\text{-arg-i} =$ the i -th argument of cco
 $\text{arg-i} : Identifier \rightarrow \text{arg-i}$
 $\text{true} \quad \rightarrow \text{com-i where } \text{arg-i} = (\text{com-i}, \text{yok-i})$
 $\text{new-com} = cco(c\text{-arg-1}, \dots, c\text{-arg-n})$
 $\text{new-com} : Error \quad \rightarrow \text{new-com}$
let
 $\text{new-yoke} = \dots$ here a specific for $va\text{-cco}$ construction of the new yoke (engineering decision)
 $\text{boo-com} = \text{new-yok.new-com}$

```

boo-com : Error   → boo-com
boo-com = (ff, ('boolean')) → 'resulting-yoke-not-satisfied'
true      → (new-com, new-yoke)

```

Let us explain this scheme on two examples.

```

va-add.(val-1, val-2) =
  val-i : Error           → val-i           for i = 1,2
  let
    (com-i, yok-i) = val-i           for i = 1,2
    com            = co-add.(com-1, com-2)
  com : Error           → com
  true              → (com, TT)

```

Here we assume that the resulting yoke is TT independently of the argument yokes. This does not seem harmful, since, as we are going to see in Sec. 5.1.5.2, yokes of numeric values will come to the play where we define the assignment instruction. There, in order to assign a value to a variable, the composite of the new value must satisfy the yoke assigned to the variable by its declaration.

Sometime, however, the choice of a new yoke may be determined by the nature of a constructor as in the case of `va-push`:

```

va-push.(val-e, val-l) =                                     push val-e on list val-l
  val-i : Error           → val-i           for i = e,l
  let
    (dat-i, bod-i, yok-i) = val-i   for i = e,l
    com = co-push.((dat-e, bod-e), (dat-l, bod-l))
  com : Error           → com
  let
    boo-com = yok-l.com
  boo-com : Error       → boo-com
  boo-com = (ff, ('boolean')) → 'resulting-yoke-not-satisfied'
  true              → (com, yok-l)

```

In this case, we have decided that the resulting yoke should be the yoke of the value, which carries a list. This is again just an example, but it seems to make sense that if we set a yoke of a list-value, then we probably wish to keep that yoke satisfied during the life-time of the list.

At the end one comment about the manipulation of yokes at the level of values. At this level we only “compute” yokes of values created by value constructors, but we have no constructors that would change a yoke of a given value. We do not have nothing like a constructor

```
change-yoke : ValueE x Yoke → ValueE
```

As we are going to see in Sec. 4.4.2, we are not going to have such mechanisms at the level of data expressions either.

More yoke-manipulating mechanisms will be introduced at the level variable declarations (Sec. 5.1.4.1), where we assign an initial yoke to a variables, and at the level of yoke-replacement instructions (Sec. 5.1.5.3), where we can change a yoke assigned to a variable.

All these decisions are, of course, of an engineering character.

4.4 Expression denotations

4.4.1 Memory states

As was already mentioned in Sec. 3.1, to define functions plying the role of the denotations of expressions, of declarations and of instructions, one has to define the concept of a *state*. In a simple programming language,

states might be just valuations, i.e., mappings from identifiers to values, whatever the latter are. However, in the majority of programming languages, states may save more than just values. They may save:

- values and pseudovalues (see later),
- types as independent beings,
- procedures⁵⁵.

By a *pseudovalue*, we mean a triple of the form $(\Omega, \text{bod}, \text{typ})$, where Ω is a special element called a *pseudo-data*. We assume that

$$\begin{aligned} \Omega & : \text{CLAN-Bo.bod} && \text{for every bod : Body} \\ (\Omega, \text{bod}) & : \text{CLAN-Bo.typ} && \text{for every bod : Body and typ : Type} \end{aligned}$$

Intuitively Ω may be regarded as a “place for a data of an arbitrary type”. As we are going to see in Sec. 5.1.4.1, pseudovalues will be assigned to variables by variable declarations. Pairs (Ω, bod) will be called *pseudocomposites*. We introduce a domain

$$\text{val} : \text{PsValue} = \{(\Omega, \text{typ}) \mid \text{typ} : \text{Type}\}$$

In the sequel, whenever we say “a value”, we shall mean “a value or a pseudovalue”. A *proper value* is a value that is not a pseudovalue. Memory states will save arbitrary values assigned to identifiers, but data expressions will evaluate to proper values only (or errors). Our *states* will correspond to that part of computer memory where we save:

- values assigned to the *identifiers of data variables*,
- types assigned to the *identifiers of type constants*,
- procedures (including functional procedures) assigned to the *identifiers of procedure names*.

The domain of *states* is defined by the following domain equations:

$$\begin{aligned} \text{sta} : \text{State} & = \text{Env} \times \text{Store} && \text{state} \\ \text{env} : \text{Env} & = \text{TypEnv} \times \text{ProEnv} && \text{environment} \\ \text{sto} : \text{Store} & = \text{Valuation} \times (\text{Error} \mid \{\text{'OK'}\}) && \text{store} \\ \text{vat} : \text{Valuation} & = \text{Identifier} \Rightarrow (\text{Value} \mid \text{PsValue}) && \text{valuation}^{56} \\ \text{tye} : \text{TypEnv} & = \text{Identifier} \Rightarrow (\text{Type} \mid \text{Body}) && \text{type environment} \\ \text{pre} : \text{ProEnv} & = \text{Identifier} \Rightarrow \text{Procedure} && \text{procedure environment} \\ \text{prc} : \text{Procedure} & = \text{ImpProc} \mid \text{FunPro} && \text{procedures}^{57} \end{aligned}$$

The split of a state into two pairs in the place of regarding it as one four-tuple is not accidental. It will be justified on the ground of our model of procedures (Sec. 6). The domain **Procedure** will be defined there too.

A *store* is that component of a state which saves values and pseudovalues by binding them to identifiers in valuations. Observe that valuations do not save errors.

An error message, when generated, becomes a component of a store and, since then, is passed to all subsequent states. Otherwise, the store is carrying ‘OK’ (no error). If the message is different from ‘OK’, then we say that the *state (store) carries an error*.

For now, we will ensure that all imperative denotations (i.e., denotations that transform states, do not change states that carry an error (transparency)), and that all applicative denotations, (i.e., denotations that

⁵⁵ In the case of object-oriented languages this model may be even more complex (Sec. 11)

⁵⁶ The metavariable that runs over Valuation has been called “vat” rather than “val”, since the latter has been already reserved for the domain Value.

⁵⁷ Procedures may be imperative or functional.

transform states into values, or types), generate an error whenever a state carries an error. This principle shall not be observed when we introduce error-handling mechanisms (Sec. 5.1.5.5 and Sec. 10.9.6.4).

Environments constitute these components of states which store user-defined bodies, types, procedures, and functions (functional procedures).

In order to describe the mechanism of errors at the level of states, we introduce four auxiliary functions:

$\text{error} : \text{State} \mapsto \text{Error} \mid \{\text{'OK'}\}$ error-selection operator

$\text{error}(\text{env}, (\text{vat}, \text{err})) = \text{err}$

$\text{is-error} : \text{State} \mapsto \text{Boolean}$ error-detection predicate for states

$\text{is-error.sta} =$
 $\text{error.sta} \neq \text{'OK'} \rightarrow \text{tt}$
 $\text{true} \rightarrow \text{ff}$

$\text{is-error} : \text{Store} \mapsto \text{Boolean}$ error-detection predicate for stores

$\text{is-error}(\text{vat}, \text{err}) =$
 $\text{err} \neq \text{'OK'} \rightarrow \text{tt}$
 $\text{true} \rightarrow \text{ff}$

$\blacktriangleleft : \text{State} \times \text{Error} \mapsto \text{State}$ error-insertion operator

$(\text{env}, (\text{vat}, \text{err})) \blacktriangleleft \text{err-1} =$
 $(\text{env}, (\text{vat}, \text{err-1}))$

4.4.2 The algebra of denotations of Lingua-A

The *algebra of expression denotations* — which we shall denote by AlgExpDen — contains six carriers:

ide	: Identifier	= ...	defined earlier
ded	: DatExpDen	= State \rightarrow ValueE	data-expression denotations
bed	: BodExpDen	= State \mapsto BodyE	type-expression denotations
tra	: TraExpDen	= Transfer	transfer-expression denotations
yok	: YokExpDen	= Yoke	yoke-expression denotations
ted	: TypExpDen	= State \mapsto TypeE	type-expression denotations

The denotations of transfer expressions and yoke expression are not functions on states since we assume that transfers and yokes are not storable. This is, of course, an engineering decision.

Below we define constructors of the denotations of data expressions, body expressions, and type expressions. Constructors of transfers and yokes have been already defined in Sec. 4.3.4.

4.4.3 Denotations of data expressions

The partiality of data-expression denotations follows from the fact that in the future (Sec. 6.5), data expressions will include procedure calls, which may generate infinite executions. Since program-termination problem is not decidable (see Sec. 7.5), we cannot expect that instead of an infinite execution, an error signal will be generated. It is to be emphasized, however, that the result of a data-expression evaluation will never be a pseudo-value.

A denotation of a data expression is said to be *transparent* for errors, if for any state that carries an error it evaluates to that error, i.e.

$$\text{ded.}(\text{env}, (\text{vat}, \text{err})) = \text{err} \quad \text{for } \text{err} \neq \text{'OK'}.$$

At the level of implementation, the appearance of an error means that this error is displayed on the monitor and program execution halts.

A constructor of data-expression denotations is said to be *diligent* if it builds transparent denotations. All our constructors of data expression denotations will be *diligent*. Consequently, all reachable expression denotations will be transparent. Their constructors split into four categories:

1. one constructor of variables,
2. constructors derived from non-boolean value constructors; for each such constructor vco we define a constructor of denotations $\text{Cdd.}[\text{vco}]$ (Cdd stands for *constructor of data-expression denotations*),
3. boolean constructors,
4. one constructor that corresponds to conditional expressions.

The list of our constructors is, therefore, the following:

A constructor of variables

$$\text{ded-variable} \quad : \text{Identifier} \quad \mapsto \text{DatExpDen}$$

Zero-argument constructors

$$\text{Cdd.}[\text{va-create-id.ide}] \quad : \quad \mapsto \text{Identifier} \quad \text{for } \text{ide} \quad : \text{Identifier}$$

$$\text{Cdd.}[\text{va-create-bo.bo}] \quad : \quad \mapsto \text{DatExpDen} \quad \text{for } \text{boo} \quad : \text{Boolean}$$

$$\text{Cdd.}[\text{va-create-in.int}] \quad : \quad \mapsto \text{DatExpDen} \quad \text{for } \text{int} \quad : \text{IntegerS}$$

$$\text{Cdd.}[\text{va-create-wo.wor}] \quad : \quad \mapsto \text{DatExpDen} \quad \text{for } \text{wor} \quad : \text{WordS}$$

Comparison constructors

$$\text{Cdd.}[\text{ded-equal}] \quad : \text{DatExpDen} \times \text{DatExpDen} \quad \mapsto \text{DatExpDen}$$

$$\text{Cdd.}[\text{ded-less}] \quad : \text{DatExpDen} \times \text{DatExpDen} \quad \mapsto \text{DatExpDen}$$

Integer number constructors

$$\text{Cdd.}[\text{va-add-in}] \quad : \text{DatExpDen} \times \text{DatExpDen} \quad \mapsto \text{DatExpDen}$$

$$\text{Cdd.}[\text{va-subtract-in}] \quad : \text{DatExpDen} \times \text{DatExpDen} \quad \mapsto \text{DatExpDen}$$

$$\text{Cdd.}[\text{va-multiply-in}] \quad : \text{DatExpDen} \times \text{DatExpDen} \quad \mapsto \text{DatExpDen}$$

$$\text{Cdd.}[\text{va-divide-in}] \quad : \text{DatExpDen} \times \text{DatExpDen} \quad \mapsto \text{DatExpDen}$$

Real number constructors

$$\text{Cdd.}[\text{va-add-re}] \quad : \text{DatExpDen} \times \text{DatExpDen} \quad \mapsto \text{DatExpDen}$$

$$\text{Cdd.}[\text{va-subtract-re}] \quad : \text{DatExpDen} \times \text{DatExpDen} \quad \mapsto \text{DatExpDen}$$

$$\text{Cdd.}[\text{va-multiply-re}] \quad : \text{DatExpDen} \times \text{DatExpDen} \quad \mapsto \text{DatExpDen}$$

$$\text{Cdd.}[\text{va-divide-re}] \quad : \text{DatExpDen} \times \text{DatExpDen} \quad \mapsto \text{DatExpDen}$$

Word constructors

$$\text{Cdd.}[\text{va-glue}] \quad : \text{DatExpDen} \times \text{DatExpDen} \quad \mapsto \text{DatExpDen}$$

List constructors

$$\text{Cdd.}[\text{va-create-li}] \quad : \text{DatExpDen} \quad \mapsto \text{DatExpDen}$$

Cdd.[va-push]	: DatExpDen x DatExpDen	↦ DatExpDen
Cdd.[va-top]	: DatExpDen	↦ DatExpDen
Cdd.[va-pop]	: DatExpDen	↦ DatExpDen

Array constructors

Cdd.[va-create-ar]	: DatExpDen	↦ DatExpDen
Cdd.[va-put-to-ar]	: DatExpDen x DatExpDen	↦ DatExpDen
Cdd.[va-change-in-ar]	: DatExpDen x DatExpDen x DatExpDen	↦ DatExpDen
Cdd.[va-get-from-ar]	: DatExpDen x DatExpDen	↦ DatExpDen

Record constructors

Cdd.[va-create-re]	: Identifier x DatExpDen	↦ DatExpDen
Cdd.[va-put-to-re]	: DatExpDen x DatExpDen x Identifier	↦ DatExpDen
Cdd.[va-get-from-re]	: DatExpDen x Identifier	↦ DatExpDen
Cdd.[va-change-in-re]	: DatExpDen x Identifier x DatExpDen	↦ DatExpDen

Boolean constructors

ded-and	: DatExpDen x DatExpDen	↦ DatExpDen
ded-or	: DatExpDen x DatExpDen	↦ DatExpDen
ded-not	: DatExpDen	↦ DatExpDen

Conditional-expression constructor

when	: DatExpDen x DatExpDen x DatExpDen	↦ DatExpDen
------	-------------------------------------	-------------

The first constructor builds *data-variable denotations* (ded- stands for “data-expression denotations”):

```

ded-variable : Identifier ↦ DatExpDen
ded-variable.ide.sta =
  is-error.sta  → error.sta
  let
    (env, (vat, 'OK')) = sta
    vat.ide = ?    → 'undeclared-variable'
  let
    (dat, bod, typ) = vat.ide
    dat = Ω       → 'uninitialized-variable'
  true         → (dat, bod, typ)

```

The calculations of the value of a variable (note that variable is an expression) starts from checking if its identifier *ide* has been declared and initialized. If that is not the case, then appropriate error signals are generated. In the opposite case the value assigned to *ide* becomes the final result.

Constructors of the second category are all derived in a uniform way from non-boolean value constructors. Let *vco* be a value constructor, and let *Cdd*.[*vco*] be its counterpart. Then

```

Cdd.[vco] : DedExpDenIde-1 x ... x DatExpDenIde-n ↦ DatExpDen
Cdd.[vco].(arg-1,...,arg-n).sta =
  is-error.sta  → error.sta
  for i = 1;n
  do
    (arg-i != Identifier) and arg-i.sta = ? → ?

```

(4.4.2-1)


```

let
  val-i =
    arg-i : Identifier      → arg-i
    true      → arg-i.sta
od
true      → vco.(val-1, ..., val-n)

```

Note that in this scheme the value constructor `vco` “takes care” about all cases where an error message may be generated from the level of values. Let us see it on three examples:

`Cdd[va-create-in.int] : \mapsto DatExpDen`

```

Cdd[va-create-in.int].().sta =
  is-error.sta → error.sta
  true        → va-create-in.int.()

```

We recall that

`va-create-in.int.() = (num, ('number'), TT)`

In the case of numerical division we have the following definition

`Cdd.[va-divide-in] : DatExpDen x DatExpDen \mapsto DatExpDen`

```

Cdd.[va-divide-in].(ded-1, ded-2).sta =
  is-error.sta → error.sta
  ded-i.sta = ? → ?      for i = 1,2
let
  val-i = ded-i.sta      for i = 1,2
true      → va-divide-in.(val-1, val-2)

```

To see a case where one of the arguments is an identifier consider putting a new attribute to a record:

`Cdd.[va-put-to-re] : Identifier x DatExpDen x DatExpDen \mapsto DatExpDen`

```

Cdd.[va-put-to-re].(ide, ded-e, ded-r).sta =      put attribute ide with value ded-e to record ded-r
  is-error.sta → error.sta
  ded-i.sta = ? → ?      for i = e, r
let
  val-i = ded-i.sta      for i = e, r
true      → va-put-to-re.(ide, val-e, val-r)

```

Boolean constructors do not refer (call) boolean constructors of values since they have to cope with a lazy-evaluation style (McCarthy’s) not only for errors but also for undefinedness (looping).

`dad-and : DatExpDen x DatExpDen \mapsto DatExpDen`

```

ded-and.(ded-1, ded-2).sta =
  is-error.sta → error.sta
  ded-1.sta = ? → ?
let
  val-1 = ded-1.sta
  val-1 : Error → val-1
let
  (dat-1, bod-1, typ-1) = val-1
  dat-1 = ff → ff 58
  bod-1 ≠ ('boolean') → 'boolean-expected'
  ded-2.sta = ? → ?
let

```

(4.4.2-2) (*)

⁵⁸ Notice that if `dat-1=ff` then we do need to check if `bod-1 = ('boolean')` since `val-1` is a value, and therefore it must be well-typed.

```

    val-2 = ded-2.sta
val-2 : Error      → val-2
let
  (dat-2, bod-2, typ-2) = val-2
bod-2 ≠ ('boolean') → 'boolean-expected'
true                → (dat-2, ('boolean'), TT)

```

Notice that the computation starts from an attempt of computing the value of the first argument and if this value carries `ff`, then the computation terminates with this value (clause `(*)`). In this way we avoid the computation of the second argument which might loop indefinitely or generate an error message. Informally we may say that `dad-end` is not “transparent for undefinedness” of the second argument.

Since `ded-not` must be “transparent for undefinedness”, we can define it in a standard way, i.e. by calling the corresponding value constructor:

```

Cdd.[va-not] : DatExpDen ↦ DatExpDen
Cdd.[va-not].ded.sta =
  is-error.sta → error.sta
  ded.sta = ? → ?
let
  val = ded.sta
true → va-not.(val)

```

The constructor corresponding to alternative is defined in a way which guarantees the satisfaction of De Morgan’s law:

```

ded-or.(ded-1, ded-2) = Cdd.[va-not].( ded-and.( Cdd.[va-not].ded-1 , Cdd.[va-not].ded-2 ) )

```

The unique constructor of the fourth category corresponds to conditional expressions⁵⁹:

```

when : DatExpDen x DatExpDen x DatExpDen ↦ DatExpDen
when.(ded-1, ded-2, ded-3).sta =
  is-error.sta      → error.sta
  ded-1.sta = ?     → ?
let
  val-1 = ded-1.sta
val-1 : Error      → val-1
let
  (dat-1, bod-1, yok) = val-1
bod-1 ≠ ('boolean') → 'boolean-expected'
dat-1 = tt          → ded-2.sta
dat-1 = ff          → ded-3.sta

```

Note that two last clauses cover the case, where the computation of `ded-2.sta` or `ded-3.sta` does not terminate. In this case, we have to do with lazy evaluation since in evaluating one of `ded-i`’s we do not need to care if the evaluation of the other is infinite or if it terminates with an error message⁶⁰.

Note that the evaluation of a conditional expression may lead to a situation where the type of the result depends on the property of input data, as, e.g., in the example:

```

if x > 0 then x+2 else 'abcd' fi

```

(here `fi` is a closing parenthesis of `if`).

⁵⁹ We call it “**when**” rather than “**if**” since the latter is reserved for conditional instructions.

⁶⁰ The acceptance of lazy evaluation in this place is a significant decision of language constructor, since it allows for the use of partial functions without the risk of error messages or infinite computations. Notice that if `sqrt(x)` denotes square root of `x`, then the expression `if x>0 then sqrt(x) else sqrt(-x) fi` evaluated eagerly would generate an error signal for every `x` different from 0.

4.4.4 Denotations of body-, trace, yoke- and type expressions

In **Lingua** type expressions are used in three situations:

1. in type-constant declarations (Sec. 5.1.4.2), where we save types in environments for later use in data-variable declarations and in procedure declarations,
2. in data-variable declarations (Sec. 5.1.4.1), where they indicate types of values assignable to these variables,
3. in procedure declarations (Sec. 6.3.2), where they indicate types of values that may be passed to these procedures as parameters.

Our algebra of *type-expression denotations* includes five carriers:

ide	: Identifier	= ...	defined earlier
bed	: BodExpDen	= State \mapsto BodyE	body-expression denotations
tra	: TraExpDen	= Transfer	transfer-expression denotations
yok	: YokExpDen	= Yoke	yoke-expression denotations
ted	: TypExpDen	= State \mapsto TypeE	type-expression denotations

The fact that the denotations of transfer expressions and yoke expressions are not functions on states, is the consequence of an engineering decision that transfers and yokes will not be saved in computer memory. The only way we can get them is by an evaluation of appropriate expressions. In turn, the denotations of body expressions and type expressions are functions on states which results from another engineering decision that bodies and types may be assigned to identifiers in environments. Both decisions will be commented at the end of this section.

The first constructor of body-expression denotations to be defined builds *body constants*, which are analogous to data variables. In this case, we are talking about “constants” rather than “variables” since we assume that bodies assigned to identifiers, once established, are never changed during program execution. Below **bod-** stands for body-expression denotation.

bod-constant : Identifier \mapsto BodExpDen

```

bod-constant.sta =
  is-error.sta       $\rightarrow$  error.sta
  let
    ((tye, pre), sto) = sta
  tye.ide = ?       $\rightarrow$  'body-constant-undefined'
  not tye.ide : Body  $\rightarrow$  'body-expected'
  true            $\rightarrow$  tye.ide

```

Notice that unlike for data variables, in this case, we do not have a situation where a constant has been defined but not initialized. It is the consequence of the fact that body declarations (Sec. 5.1.4.2) always assign concrete bodies to identifiers.

The remaining constructors of body-expression denotations correspond to body building constructors (see Sec. 4.3.2) and are defined in a way analogous to the definitions of data-expression denotations. Let **Cbd**, which stands for *constructor of body-expression denotations*, denotes a constructor which builds constructors of expression denotations from constructors of bodies. If

bco : BodIde-1 x ... x BodIde-n \mapsto BodyE

is a body constructor where BodIde-i stands for BodyE or Identifier, then

Cbd.[bco] : BodExpDenIde-1 x ... x BodExpDenIde-n \mapsto BodExpDen

```

Cbd.[bco].(arg-1, ..., arg-n).sta =
  is-error.sta  $\rightarrow$  error.sta
  let

```

$\text{bod-}i = \text{arg-}i.\text{sta}$ for $i = 1;n$ where $\text{ide.sta} = \text{ide}$
true $\rightarrow \text{bco.}(\text{bod-}1, \dots, \text{bod-}n)$

where bco “cares for errors”. In this group we include the following constructors:

$\text{Cbd.}[\text{bo-creat-}bo]$:	$\mapsto \text{BodExpDen}$
$\text{Cbd.}[\text{bo-creat-in}]$:	$\mapsto \text{BodExpDen}$
$\text{Cbd.}[\text{bo-creat-wo}]$:	$\mapsto \text{BodExpDen}$
$\text{Cbd.}[\text{bo-creat-ar}]$:	$\text{BodExpDen} \mapsto \text{BodExpDen}$
$\text{Cbd.}[\text{bo-creat-re}]$:	$\text{BodExpDen} \times \text{Identifier} \mapsto \text{BodExpDen}$
$\text{Cbd.}[\text{bo-put-to-re}]$:	$\text{Identifier} \times \text{BodExpDen} \times \text{BodExpDen} \mapsto \text{BodExpDen}$

Now let’s see two exemplary definitions of our constructors. Analogously to the case of data-expression denotations, in this case each of such constructors “calls” a corresponding constructor of bodies.

$\text{Cbd.}[\text{bo-creat-in}] : \mapsto \text{BodExpDen}$

$\text{Cbd.}[\text{bo-creat-in}]().\text{sta} =$
 $\text{is-error.sta} \rightarrow \text{error.sta}$
true $\rightarrow \text{bo-creat-in}().$

$\text{Cbd.}[\text{bo-put-to-re}] : \text{BodExpDen} \times \text{Identifier} \times \text{BodExpDen} \mapsto \text{BodExpDen}$

$\text{Cbd.}[\text{bo-put-to-re}] (\text{bed-e}, \text{ide}, \text{bed-r}).\text{sta} =$ e for “element”, r for “record”
 $\text{is-error.sta} \rightarrow \text{error.sta}$
let
 $\text{bod-}i = \text{bed-}i.\text{sta}$ for $i = e, r$
 $\text{bod-}i : \text{Error} \rightarrow \text{bod-}i$ for $i = e, r$
true $\rightarrow \text{bo-put-to-re.}(\text{bed-e}, \text{ide}, \text{bed-r})$

Constructors of transfers and yokes are known from Sec. 4.3.4, and the two remaining constructors of type-expression denotations are the following:

$\text{typ-constant} : \text{Identifier} \mapsto \text{TypExpDen}$

$\text{typ-constant.ide.sta} =$
 $\text{is-error.sta} \rightarrow \text{error.sta}$
let
 $((\text{tye}, \text{pre}), \text{sto}) = \text{sta}$
 $\text{tye.ide} = ? \rightarrow \text{‘type-constant-undefined’}$
 $\text{tye.ide} : \text{Body} \rightarrow \text{‘type-expected’}$
true $\rightarrow \text{tye.ide}$

$\text{create-ty} : \text{BodExpDen} \times \text{YokExpDen} \mapsto \text{TypExpDen}$

$\text{create-ty.}(\text{bed}, \text{yok}).\text{sta} =$
 $\text{is-error.sta} \rightarrow \text{sta}$
let
 $\text{bod} = \text{bed.sta}$

true \rightarrow `create-ty.(bod, yok)`

The second constructor allows for the construction of types with empty clans. However, as we are going to see in Sec. 5.1.5.2, the mechanisms of assigning values to variables by assignment instructions will raise an error message ‘yoke-not-satisfied’ whenever we try to assign a value to a variable with empty-clan type.

As we see, the only way we can “get a type” is either by “reading” it from a type environment, or by creating it from a body and a yoke. We do not allow the construction of structured types from earlier defined types, as it is the case with bodies. This an engineering decision that follows from the way we are going to use types in the denotations of assignment instructions and in the mechanisms of passing parameters to procedures. In each of these cases, we shall compare the type assigned to a variable, say (bod-v, yok-v), with the type of a value, say (dat, (bod-d, yok-d)) which is going to be assigned to that variable. In such cases, we shall only check if:

bod-v = bod-d and
 (dat, bod-d) satisfies yok-v

If we had structured type of say arrays of lists of records with yokes assigned to bodies at each of these levels, then the comparison of such types, or even of their “hidden bodies”, would hardly be implementable.

Summarizing these remarks, and anticipating concrete syntax of **Lingua-A**, at the level of syntax a type expression may be of only one of the two forms:

Identifier or
create-type BodExp **with** YokExp **ee**

4.4.5 Seven steps on the way to the algebra of expression denotations

As has been already mentioned a few times, the design process leading from the selection of data domains to the algebra of expression denotations shows a certain way of building an interpreter of our future language. Let us now sum up this process. The arrows in Fir.4.4-1 indicate the way we proceed from one stage of our construction to another. Red arrows indicate additionally that the non-boolean constructors of a target algebra are defined with the help of a metaconstructor that transforms source-algebra constructors into target-algebra constructors.

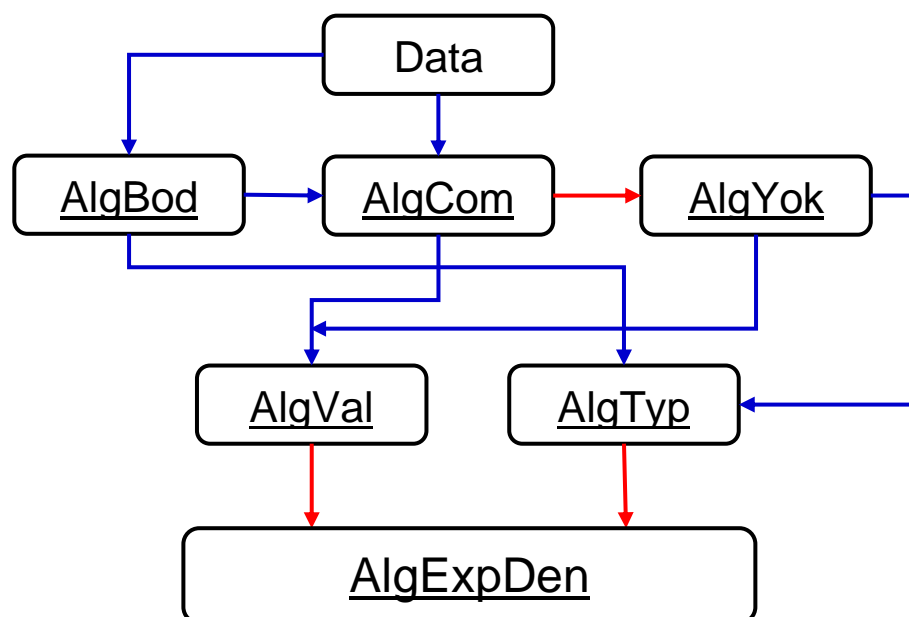


Fig. 4.4-1 Seven steps from data to denotations

1. We start by defining the domains of data. It is one of the most essential and creative steps in the process of designing a language. We decide here what are the sorts of objects that our language will manipulate, and what are the operations that will be used for this manipulation. At this stage, we also choose primary data constructors regarded as “given ahead” by a future implementation platform.
2. In this step, we define the algebra AlgBod of bodies. This step is technically rather simple, but here we take the first essential decisions about future reachable data. E.g., we decide whether we shall allow, or not, nonhomogeneous lists and arrays.
3. In the third step, we define the algebra of composites AlgCom. Its constructors are defined in a certain uniform way by referring to (calling) primary constructors and body constructors. Here we define the significant part of error-detection mechanisms of future expressions and, in this way, decide about the field of reachable data.
4. Given composites, we may construct an algebra AlgYok of yokes, which describes properties of composites. To make the carrier of yokes rich enough, we define transfers which are functions from composites to composites. In the definitions of their constructors, we use a metaconstructor Tc which given a composite constructor cco returns a transfer constructor Tc[cco]. Yokes and transfers are going to be denotations of future transfer- and yoke expression of **Lingua-A**.
5. Over the algebras of composites and yokes, we build an algebra AlgVal of values with four carriers: Identifier, Transfer, Yoke, ValueE. This algebra inherits all constructors of transfers and yokes, and additionally, to them, we define constructors derived from composite constructors — one for each. Although the latter refer to (call) composite constructors, we cannot define a universal metaconstructor, like Tc for transits, since each of value constructors builds not only a new composite (by calling a composite constructor) but also a new yoke. The “added value” of the algebra of values is the mechanism of the creation of yokes along with the creation of composites.
6. Over the algebras of yokes and bodies, we construct an algebra of types AlgTyp. This algebra includes all constructors of bodies, all constructors of yokes, and transfer, and one constructor which combines a body with a yoke. Theoretically, we may combine an arbitrary body with any yoke, but practically some of such combinations will lead to types with empty clans. Algebra of types constitutes a fundament for the future carrier and constructors of the algebra of type-expression denotations.
7. Over the algebras of values and of types, we construct an algebra AlgExpDen of expression denotations. Its carrier DatExpDen is derived from the algebra of values, and the corresponding constructors are derived with the help of a metaconstructor Cdd which calls constructors of values. Analogously its carrier BodExpDen and its constructors are built with the help of a metaconstructor Cbd which calls constructors of bodies.

It is worth commenting, in the end, a particular position of boolean constructors in these seven steps. They appear only among the constructors of yokes (Kleene’s constructors) and the constructors of data-expression denotations (McCarthy’s constructors). Of course, theoretically, we could have defined them in the remaining algebras, but we could not use them later in the boolean constructors of yokes and data-expression denotation. This is due to the fact that in both cases they are not error-transparent, and therefore in each case, we have to describe their specific mechanism of laziness “from the scratch”. For instance, at the level of data expressions, we have to express the fact that in evaluating an and-expression, if one of its arguments evaluates to **ff**, then the evaluation of the other is skipped.

4.5 Algebras of the syntax of expressions

4.5.1 Abstract syntax of Lingua-A

According to the five-step method of building a denotational model of an applicative part of a programming language (see Sec. 3.5), what we have to do now, is to construct abstract, concrete, and colloquial syntax for **Lingua-A**.

As we already know from Sec. 2.12, starting from the signature of an algebra of denotations (in this case AlgExpDen), we can “algorithmically generate” a corresponding abstract-syntax algebra (in this case AlgExpA), and from that algebra, we can generate (Sec. 2.14) an equational grammar of abstract syntax. Below we skip building an algebra of abstract syntax of expressions, and we directly build the corresponding grammar.

carriers of denotations	carriers of syntaxes	description of syntax
Identifier	IdentifierA	identifier expressions
DatExpDen	DatExpA	abstract data expressions
BodExpDen	BodExpA	abstract body expressions
TraExpDen	TraExpA	abstract transfer expressions
YokExpDen	YokExpA	abstract yoke expressions
TypExpDen	TypExpA	abstract type expressions

Tab. 4.5-1 The carriers of the syntactic algebra

To each of five carriers of the algebra of denotations, we assign a corresponding carrier of abstract syntax (Tab. 4.5-1).

The equational grammar which describes our abstract-syntax will be written with notational conventions introduced in Sec. 2.14. For each syntactic category, there is one domain equation of our grammar, and for each constructor, there is one component (one line) of such an equation. According to the assumed convention, phrases that belong to syntax are typeset in *Courier New*. The first equation defines the domain of identifier expressions:

Identifiers

```
ide : IdentifierA =
  {Cdd. [va-create-id.ide]. () | ide : Identifier}}
```

Here `create-id.ide.()` is an element of abstract syntax, and `ide`, `IdentifierA`, and `Identifier` are metavariables that belong to the level of *MetaSoft* (Sec. 2). If we would like to write this equation formally in standard notation of abstract-syntax grammars as in Sec. 2.12, we had to explicitly list all (a finite number of) identifiers acceptable in a given implementation, i.e.:

```
ide : IdentifierA = create-id.a.() | create-id.b.() | create-id.ab.() | ...
```

Since this would not be very practical, on a formal level we use the above abbreviation, and on manual’s level — where we define concrete syntax (see Sec. 4.5.2) — we only list characters admissible in identifiers, e.g., `{a, b, c, ..., y, z, 0, 1, ..., 9}`, and we indicate the maximal length of an identifier. On the implementation level, we write a simple program checking if a given identifier is not too long and if it does not contain prohibited characters e.g., `#, ?, !, ...`

Here it should also be pointed out that whereas at the level of concrete syntax identifiers are just arbitrary strings (of restricted length) of admissible characters, such as e.g.

```
birth-date
```

a counterpart of such an identifier at the level of abstract syntax is a phrase

```
Cdd.[create-id.birth-date].()
```

which stands for itself.

The following four equations define the languages of data expressions, transfer expressions, yoke expressions, and type expressions. In the first three lines of the first equation `boo` (in *Courier New*) is a syntactic representation of boolean data `boo` (in *Arial*), and similarly for `num` and `wor`.

Data expressions

`dae : DatExpA =`

constants

```
{Cdd[create-bo.boo].() | boo : Boolean} |
{Cdd[create-in.int].() | num: NumberS} |
{Cdd[create-wo.wor].() | wor : WordS} |
```

variables

```
dat-variable.(Identifier) |
```

comparison expressions

```
Cdd[va-equal].(DatExpA, DatExpA) |
Cdd[va-less].(DatExpA, DatExpA) |
```

integer number expressions

```
Cdd[va-add-in].(DatExpA, DatExpA) |
Cdd[va-subtract-in].(DatExpA, DatExpA) |
Cdd[va-multiply-in].(DatExpA, DatExpA) |
Cdd[va-divide-in].(DatExpA, DatExpA) |
```

real number expressions

```
Cdd[va-add-re].(DatExpA, DatExpA) |
Cdd[va-subtract-re].(DatExpA, DatExpA) |
Cdd[va-multiply-re].(DatExpA, DatExpA) |
Cdd[va-divide-re].(DatExpA, DatExpA) |
```

word expressions

```
Cdd[va-glue].(DatExpA, DatExpA) |
```

list expressions

```
Cdd[va-create-li].(DatExpA) |
Cdd[va-push].(DatExpA, DatExpA) |
Cdd[va-top].(DatExpA) |
Cdd[va-pop].(DatExpA) |
```

array expressions

```
Cdd[va-create-ar].(DatExpA) |
Cdd[va-put-to-ar].(DatExpA, DatExpA) |
Cdd[va-change-in-ar].(DatExpA, DatExpA, DatExpA) |
Cdd[va-get-from-ar].(DatExpA, DatExpA) |
```

record expressions


```

Cdd[va-create-re] . (Identifier, DatExpA) |
Cdd[va-put-to-re] . (Identifier, DatExpA, DatExpA) |
Cdd[va-get-from-re] . (DatExpA, Identifier) |
Cdd[va-cut-from-re] . (Identifier, DatExpA) |
Cdd[va-change-in-re] . (DatExpA, Identifier, DatExpA) |

```

Boolean expressions

```

ded-and . (DatExpA , DatExpA) |
ded-or . (DatExpA , DatExpA) |
ded-not . (DatExpA) |

```

conditional expressions

```

when (DatExpA , DatExpA , DatExpA)

```

The equations of transfer expressions and yoke expressions are derived from the signature of the algebra of yokes (Sec.4.3.4), since its constructors are at the same time the constructors of denotations. We omit the constructors of identifiers, since the corresponding equation is already in our grammar.

Transfer expressions

```

tre : TraExpA =
{Tc[co-create-in.int].() | num : NumberS} |
{Tc[co-create-wo.wor].() | wor: WordS} |
pass.() |
Tc[co-add-in] . (TraExpA, TraExpA) |
Tc[co-subtract-in] . (TraExpA, TraExpA) |
Tc[co-multiply-in] . (TraExpA, TraExpA) |
Tc[co-divide-in] . (TraExpA, TraExpA) |
Tc[co-add-re] . (TraExpA, TraExpA) |
Tc[co-subtract-re] . (TraExpA, TraExpA) |
Tc[co-multiply-re] . (TraExpA, TraExpA) |
Tc[co-divide-re] . (TraExpA, TraExpA) |
yo-sum . (TraExpA) |
yo-max . (TraExpA) |
Tc[co-glue] . (TraExpA, TraExpA) |
Tc[co-top] . (TraExpA) |
Tc[co-get-from-ar] . (TraExpA, TraExpA) |
Tc[co-get-from-re] . (TraExpA, Identifier) |

```

Yoke expressions

```

yoe : YokExpA =
Tc[co-create-bo.tt].() |
Tc[co-create-bo.ff].() |
Tc[co-equal] . (TraExpA, TraExpA) |
Tc[co-less] . (TraExpA, TraExpA) |
yo-unique . (TraExpA) |
yo-increasing-in . (TraExpA) |
yo-and . (YokExpA, YokExpA) |
yo-or . (YokExpA, YokExpA) |
yo-not . (YokExpA) |

```

```

all-of-li. (YokExpA)           |
all-of-ar. (YokExpA)

```

Body expressions

```

bex : BodExpA =
  Cbd. [bo-create-bo]. ()      |
  Cbd. [bo-create-in]. ()     |
  Cbd. [bo-create-wo]. ()     |
  bod-constant. (Identifier)  |
  Cbd. [bo-create-li]. (BodExpA) |
  Cbd. [bo-create-ar]. (BodExpA) |
  Cbd. [bo-create-re]. (Identifier, BodExpA) |
  Cbd. [bo-put-to-re]. (BodExpA, Identifier, BodExpA) |

```

Type expressions

```

tex : TypExpA =
  type-constant. (Identifier) |
  create-type. (BodExpA, YokExpA)

```

4.5.2 Concrete syntax of Lingua-A

As has been explained in Sec. 2.14 and in Sec. 3.5, *concrete syntax* was historically meant as a syntax which was used by programmers. In our approach concrete syntax constitutes kind of a “denotational approximation” of programmer’s syntax, i.e. such a syntax for which a denotational semantics exists. The final programmer’s syntax is the result of introducing notational conventions called *colloquialisms* (Sec. 3.5). Along with colloquial syntax, we define a function called *restoring transformation* that maps colloquial syntax into concrete syntax (see Fig. 3.5-2 in Sec. 3.5).

The present section contains an equational grammar of concrete syntax of **Lingua-A**. The corresponding *expression algebra* will be denoted by **AlgExp**. Its carriers are defined explicitly by the grammar, which is below, and its constructors are implicit in the equations of that grammar.

The modifications of the abstract syntax described below correspond to a homomorphism **Co** (see Fig. 3.5-2), which in our case, is an isomorphism, i.e., a one-to-one many-sorted function. Main changes on the way from abstract syntax to concrete syntax are the following:

1. As boolean constants, we take `true` and `false`.
2. Numeric constants are written with a colon as a separator between the integer part and the fractional parts, e.g. `12,473`,
3. Word constants are closed in apostrophes, e.g., `'salary'`.
4. In the case of data variables and type constants instead of `variable-dat(abc)` and `type-constant(abc)` we write `abc` in both cases. This gluing is not harmful (is isomorphic) since the glued expressions belong to different carriers of the algebra.
5. Type expressions for simple yokeless types are written `boolean`, `number`, `word`.
6. Arithmetic operators and predicates are written with infix notation and with “common” symbols `+`, `/`, `<`, hence we write, e.g. `(x + y)` and `(x < y)` instead of `add(x, y)` and `less(x, y)`. The “superfluous” parenthesis shall be dropped only at the level of colloquial syntax since such a transformation is not homomorphic.

7. For boolean constructors, we use common names **or**, **and**, **not** written in infix notation. In the context of data expressions, they denote McCarthy's operators and in the context of transfer expressions — Kleene's operators. This situation does not lead to inconsistency since context always indicates the appropriate meaning. The use of boldface typesetting (here and below) is only a "syntactic sugar", which means that it helps humans to read programs, but is grammatically and denotationally not significant.
8. Conditional expressions are written with an infix notation:
if DatExp **then** DatExp **else** DatExp **fi**,
 and similar conventions are assumed for list-, array- and record constructors (see the grammar below).
9. Data- and type expressions, if written with infix notations, are closed with the parenthesis **ee**, which stands for *end-of-expression*.
10. None of the keywords `true`, `false`, **if**, **then**, ... can be used as an identifier. However, since this assumption is hardly expressible by a grammar, it is usually assured by lexical analyzer.

Our new grammar is described below. In this case, the names of syntactic categories are written without any suffix (formerly it was A), since now we have to do with a grammar addressed to users, who do not need to know about abstract syntax at all.

```
ide : Identifier =
    ide | ...                               for every syntactically acceptable ide
```

Data expressions

```
dae : DatExp =
```

constants

```
    true | false |
    num          | (for every num : NumberS)
    wor          | (for every wor : WordS)
```

variables

```
    Identifier | (constructor's name is omitted)
```

comparison expressions

```
    (DatExp = DatExp) |
    (DatExp < DatExp) |
```

integer number expressions

```
    (DatExp + DatExp) |
    (DatExp - DatExp) |
    (DatExp * DatExp) |
    (DatExp / DatExp) |
```

real number expressions

```
    (DatExp + . DatExp) |
    (DatExp - . DatExp) |
    (DatExp * . DatExp) |
    (DatExp / . DatExp) |
```

word expressions

(DatExp © DatExp) |

list expressions**list** DatExp ee |**push** DatExp **on** DatExp ee |**top** (DatExp) |**pop** (DatExp) |**array expressions****array** DatExp ee |**put-to-arr** DatExp **new** DatExp ee |**change-arr** DatExp **at** DatExp **by** DatExp |**array** DatExp **at** DatExp ee |**record expressions****record** Identifier **value** DatExp ee |**add-attr** Identifier **value** DatExp **to** DatExp ee |**record** DatExp **at** Identifier ee |**remove-attr** Identifier **from** DatExp ee |**change-rec** DatExp **at** Identifier **by** DatExp ee |**boolean expressions**(DatExp **and** DatExp) |(DatExp **or** DatExp) |**(not** DatExp) |**conditional expression****if** DatExp **then** DatExp **else** DatExp **fi**

In the last clause, we use (a common) **if** rather than **when** (as in abstract syntax). This will not lead to any confusion with conditional instructions **if-then-else-fi** (Sec.5.1.5.5) since expressions and instructions belong to different carriers (syntactic categories). Note that at the level of denotations, we had to use different names for the corresponding constructors, since they are different functions.

Transfer expressions

tre : TraExp =

num | for every num : NumberS

wor | for every wor : Words

value | (in the place of pass . ())

(TraExp + TraExp) |

(TraExp - TraExp) |

(TraExp * TraExp) |

(TraExp / TraExp) |

(TraExp + . TraExp) |

(TraExp - . TraExp) |

(TraExp * . TraExp) |

(TraExp / . TraExp) |

sum (TraExp) |

```

max (TraExp)           |
  (TraExp @ TraExp)    |
top (TraExp)          |
array TraExp at TraExp ee |
record TraExp at Identifier ee

```

The fact that transfer expressions

```

top (TraExp)
array TraExp at TraExp ee
record TraExp at Identifier ee

```

have the same structure and keywords as in the case of data expression is not harmful for the unambiguity of our grammar (i.e. does not destroy the isomorphism of our transformation from abstract syntax to concrete syntax), because data expression and trace expressions belong to two different carriers of our algebras of syntax.

Yoke expressions

```

yoe : YokExp =
  true | false           |
  (TraExp = TraExp)     |
  (TraExp < TraExp)     |
  unique (TraExp)       |
  increasing-in (TraExp) |
  (YokExp and YokExp)   |
  (YokExp or YokExp)    |
  (not YokExp)         |
  all-of-li YokExp ee   |
  all-of-ar YokExp ee

```

Boolean operators do not need to be marked with **-yo**, since the context of an expression will always indicate whether this is a data expression, where McCarthy's operators are used, or a yoke expression, where we use Kleene's calculus.

Body expressions

```

bex : BodExp =
  boolean                |
  number                 |
  word                   |
  Identifier              |
  list BodExp ee         |
  array BodExp ee       |
  record Identifier as BodExp ee |
  add-attr Identifier value BodExp to BodExp ee |

```

Type expressions

```

tex : TypExp =

```

Identifier |
type BodExp with YokExp ee

4.5.3 Colloquial syntax of **Lingua-A**

The definition of a colloquial syntax is an important step in the process of language design since it makes this language more user-friendly. We gain on clarity without losing anything of mathematical precision.

We shall assume that colloquial syntax includes all concrete syntax, which means that the use of colloquialisms is optional. Formally, each colloquialism expands our grammar of concrete syntax by a new clause, or — equivalently — our concrete-syntax algebra, by a new constructor. The algebra of colloquial syntax is, therefore, not similar to AlgDen, and therefore it cannot have denotational semantics in AlgDen.

Below I show some examples of possible colloquialisms in **Lingua-A**. They are not necessarily the best solutions since the aim here is to show the method rather than to construct a real language. All colloquialisms are defined informally based on examples, which, however, should indicate a way to both — grammatical clauses and a restoring transformation.

4.5.3.1 A general rule for the layout of syntax

We allow spaces, tabulators, carriage returns, boldface or underlining in any place, and assume that they do not change the (denotational) meaning of syntax. They all constitute “syntactic sugar”, and at the level of implementation should be removed by a restoring transformation. Boldface print is most frequently used to distinguish between identifiers and keywords. Of course, at the level of implementation, we must have a lexical analyzer that would protect programmers against using constructor names as identifiers.

We allow comments in programs which should be closed in parentheses # (left parenthesis) and \$ (right parenthesis).

4.5.3.2 Boolean data-expressions

For boolean expressions, we allow the omission of the “unnecessary” parentheses and assume the priority of conjunction over alternative. E.g.

- instead of writing $(x \text{ or } (y \text{ or } z))$ we write $x \text{ or } y \text{ or } z$ and
- instead of writing $(x \text{ or } (y \text{ and } z))$ we write $x \text{ or } y \text{ and } z$

In the first case, the restoring transformation may add parentheses in an arbitrary way, which is due to the associativity of the alternative. In the second — the assumed priority has to be observed.

4.5.3.3 Numeric data-expressions

The case of numeric expressions is a little more complicated since in real situations the addition and the multiplication are not associative, which is due to the effect of overloading. E.g., if the maximal size of a number in our implementation is 10, then

$$((-4 + 9) + 2) = 7 \quad \text{but}$$

$$(-4 + (9 + 2)) = \text{'overflow'}$$

A usual practice is therefore that parentheses-free expressions are evaluated from left to right in using the priorities between operations. E.g., the expression:

$$x + y + z + x*y$$

is restored to

$$(((x + y) + z) + (x*z))$$

4.5.3.4 Array data-expressions

In this category we are going to have four colloquialisms. The first of them concerns the constructor of an array. For instance, the colloquial expression

```
array [x, x+y, 3*y]
```

describes the construction of an array with three numeric elements, and thus unfolds to the concrete expression:

put-to-arr	(add the value of $3*y$ to the array)
put-to-arr	(add the value of $x+y$ to the array)
array x ee	create one-element array with the value of x)
new x+y ee	
new 3*y ee	

The second colloquialism allows to write

```
measurement-data.[x+1]
```

where measurement-data is an array variable, instead of

```
array measurement-data at x+1 ee
```

and

```
measurement-data.[x+1].[y-1]
```

instead of:

```
array
  array measurement-data at x+1 ee
  at y-1
ee
```

The case of adding new elements to an array may be treated analogously. We can write

```
put-to-arr measurement-data new [x, x + y, 3*y] ee
```

instead of

```
put-to-arr
  put-to-arr
    put-to-arr measurement-data new x ee
    new x+y
  ee
  new 3*y
ee
```

and in the case of array modification (here we introduce a new symbol „<=“):

```
change-arr measurement-data by
  s   <= x,
  s+1 <= x+y,
  3*p <= z-1
ee
```

which unfolds to:

```
change-arr
  change-arr
    change-arr measurement-data at s by x ee
    at s+1 by x+y ee
  at 3*p by z-1 ee
```

4.5.3.5 Record data-expressions

Examples for records may be similar to these for arrays. For instance, we may assume that a colloquial expression:

```

record
  ch-name      := 'John',
  fa-name      := 'Smith',
  birth-date   := 1968,
  award-years := award-years-Smith
ee

```

corresponds to the concrete:

```

add-attr award-years value award-years-Smith to
  add-attr birth-date value 1968 to
    add-attr fa-name value 'Smith' to
      record ch-name value 'John' ee
    ee
  ee
ee

```

and a colloquial expression

```
employee.(fa-name)
```

corresponds to the concrete:

```
record employee at fa-name ee
```

Notice that despite a similarity between selection expression from an array and from a record, there is no ambiguity since array indices are closed in bracket parenthesis and record indices in ordinary parenthesis.

4.5.3.6 Transfer and yoke expressions

Similarly, as for data expressions, we introduce school rules for dropping parentheses with corresponding priorities between operations. For instance, in the place of:

```
(2 + value) < 10
```

we write

```
2 + value < 10
```

In the place of

```
array value at 5 ee
```

which means that if the input composite carries an array, then we take the fifth element of that array, we colloquially write

```
array. [5]
```

It is to be recalled that in this case, **array** is not an array variable — as, e.g., in the expression `measurement-data.[x+1]` — but a keyword. We can also write

```
array. [5]. [7]
```

instead of

```

array
  array value at 5 ee
  at 7
ee

```


which means that in an input array, which should be an array or arrays, we choose the fifth element, and then the seventh element of the former element.

Analogously we may construct a transfer expression that selects the value of a record attribute where that record is a top element of a list. A concrete-syntax expression

```
record top(value) at age ee
```

could be colloquialized to

```
top. (age)
```

4.5.3.7 Type expressions

Here we give only one example. A colloquial type expression

```
type
  record
    name      as string,
    birth-year as number
  with
    birth-year > 2000
ee
```

unfolds to the following concrete form

```
type
  add-attr birth-year value number
  to
    record
      name as string
    ee
  with
    record value at birth-year ee > 2000 ee
ee
```

4.6 A sketch of the semantics of Lingua-A

Let us recall that AlgExp and AlgExpDen denote respectively the algebras of concrete syntax and of denotations of **Lingua-A**. Since in our case the former is isomorphic with the latter, there is a unique homomorphism:

$$Cs : \underline{\text{AlgExp}} \mapsto \underline{\text{AlgExpDen}}$$

with five components:

```
Sid   : Identifier  $\mapsto$  Identifier
Sde   : DatExp     $\mapsto$  DatExpDen
Stre  : TraExp     $\mapsto$  TraExpDen
Syoe  : YokExp     $\mapsto$  YokExpDen
Sbe   : BodExp     $\mapsto$  BodExpDen
Ste   : TypExp     $\mapsto$  TypExpDen
```

Below some examples of the definitions of these components. With *Courier*, we write not only concrete syntactic elements but also corresponding metavariables, as, e.g., *ide* or *dae-i*. This convention is, of course, not very formal but hopefully will improve the readability of our closes. We recall that $\text{Cdd}[\text{vco}]$ denotes a constructor of data expression denotations, which corresponds to a constructor vco of values.

Identifiers

$Sid : Identifier \mapsto Identifier$

$Sid.[ide] = \text{create-id.}ide.()$ for every ide

algebraic form

$Sid.[ide] = ide$ for every ide

direct form

Data expressions

$Sde : DatExp \mapsto DatExpDen$ i.e.

$Sde : DatExp \mapsto State \rightarrow ValueE$

$Sde.[true] = Cdd[va-create-bo.tt].()$

algebraic form

$Sde.[true].sta =$

direct form

$is-error.sta \rightarrow error.sta$

$true \rightarrow (tt, ('boolean'), TT)$

$Sde.[ide] = ded-variable.(Sid.[ide])$ for $ide : Identifier$

$Sde.[ide].sta =$

$is-error.sta \rightarrow error.sta$

let

$(env, (vat, 'OK')) = sta$

$ide = Sid[ide]$

$vat.ide = ? \rightarrow 'undeclared-variable'$

let

$(dat, bod, yok) = vat.ide$

$dat = \Omega \rightarrow 'uninitialized-variable'$

$true \rightarrow (dat, bod, yok)$

$Sde.[(dae-1 / dae-2)] = Cdd[va-add].(Sde.[dae-1], Sde.[dae-2])$

$Sde.[(dae-1 / dae-2)].sta =$

$is-error.sta \rightarrow error.sta$

$Sde.[dae-i].sta = ? \rightarrow ?$

for $i = 1, 2$

let

$(dat-i, bod-i, yok-i) = Sde.[dae-i].sta$

for $i = 1, 2$

$bod-i \neq ('real') \rightarrow 'real-expected'$

for $i = 1, 2$

let

$rea = divide-re.(dat-1 + dat-2)^{61}$

$rea : Error \rightarrow rea$

$true \rightarrow (rea, ('real'), TT)$

Transfer expressions

⁶¹ Here we use the fact that composites are well-structured, hence if $bod-l = ('number')$ for $l = 1, 2$, then $dat-i : Number$ for $l = 1, 2$.

It is to be recalled that transfer denotations are transfers themselves, which, in turn, are functions from composites and errors to composites and errors.

$\text{Stre} : \text{TraExp} \mapsto \text{Transfer}$

$\text{Stre} : \text{TraExp} \mapsto \text{CompositeE} \mapsto \text{CompositeE}$

$\text{Stre}[\text{value}] = \text{pass}()$

$\text{Stre}[\text{value}].\text{com} = \text{com}$

$\text{Stre}[(\text{tre-1} / \text{tre-2})] = \text{Tc}[\text{co-divide}](\text{Stre}[\text{tre-1}], \text{Stre}[\text{tre-2}])$ i.e.

$\text{Stre}[(\text{tre-1} + \text{tre-2})].\text{com} =$

$\text{com} : \text{Error} \rightarrow \text{com}$

let

$\text{com-i} = \text{Stre}[\text{tre-i}].\text{com}$ for $i = 1,2$

$\text{com-i} : \text{Error} \rightarrow \text{com-i}$ for $i = 1,2$

let

$(\text{dat-i}, \text{bod-i}) = \text{com-i}$ for $i = 1,2$

$\text{bod-i} \neq (\text{'number'}) \rightarrow \text{'number-required'}$ for $i = 1,2$

$\text{dat-2} = 0 \rightarrow \text{'division-by-zero'}$

let

$\text{num} = \text{round}(\text{dat-1} \div \text{dat-2})$

$\text{oversized.int} \rightarrow \text{'overflow'}$

true $\rightarrow (\text{num}, (\text{'number'}))$

Yoke expressions

$\text{Syoe}[\text{true}] = \text{create-tr-bo.tt}()$ i.e.

$\text{Syoe}[\text{true}].\text{com} =$

$\text{com} : \text{Error} \rightarrow \text{com}$

true $\rightarrow (\text{tt}, (\text{'boolean'}))$

$\text{Syoe}[\text{all-li tre satisfy yoe ee}] = \text{all-of-li}(\text{Stre}[\text{tre}], \text{Syoe}[\text{yoe}])$ i.e.

$\text{Syoe}[\text{all-li tre satisfy yoe ee}].\text{com} =$

$\text{com} : \text{Error} \rightarrow \text{com}$

let

$\text{com-l} = \text{Stre}[\text{tre}].\text{com}$

$\text{com-l} : \text{Error} \rightarrow \text{com-l}$

$\text{sort.com-l} \neq \text{'L'} \rightarrow \text{'list-expected'}$

let

$(\text{dat-1}, \dots, \text{dat-n}) = \text{data.com-l}$

$(\text{'L'}, \text{bod}) = \text{com-l}$

$\text{com-i} = \text{Syoe}[\text{yoe}].(\text{dat-i}, \text{bod})$ for $i = 1;n$

$\text{com-i} : \text{Error} \rightarrow \text{com-i}$

$(\forall i = 1;n) \text{com-i} = (\text{tt}, (\text{'boolean'})) \rightarrow (\text{tt}, (\text{'boolean'}))$

true $\rightarrow (\text{ff}, (\text{'boolean'}))$

-l for "list"

Type expressions

The denotations of type expressions refer to the types saved in type environments.

$$\text{Sty} : \text{TypExp} \mapsto \text{TypExpDen}$$

$$\text{Sty} : \text{TypExp} \mapsto \text{State} \mapsto \text{TypeE}$$

$$\text{Sty}.\text{[ide]} = \text{type-constant.ide} \quad \text{i.e.}$$

$$\text{Sty}.\text{[ide]}.sta =$$

$$\text{is-error.sta} \rightarrow \text{error.sta}$$

let

$$((\text{tye}, \text{pre}), \text{sto}) = \text{sta}$$

$$\text{tye.ide} = ? \rightarrow \text{'type-constant-undefined'}$$

$$\text{true} \rightarrow \text{tye.ide}$$

The remaining definitions are left to the reader.

4.7 Two forms of a manual

A denotational model of a programming language is a starting point not only for the development of implementation but also for writing a user manual. Since manuals written in such a way do not exist yet, there are no practical experiences in this field. It seems, however rather evident that such manuals should describe a language in three following steps in the given order (this is experimentally elaborated in [29]):

1. the concrete syntax described by equational grammar and illustrated by examples,
2. the colloquial syntax illustrated by examples of restoring transformations (e.g., as in Sec. 4.5.3),
3. the semantics of concrete syntax, i.e., the association of concrete programs to their denotations.

In describing the semantics of a language, one has to choose between two forms of definitions (cf. Sec. 4.7):

- A. definitions that refer to (“call”) earlier defined constructors as in the majority of cases in Sec. 4.3 and Sec. 4.4; such definitions will be referred to as *algebraic*,
- B. definitions that describe constructors explicitly, as in Sec. 4.6; such definitions will be called *direct*.

Which of these definitions we choose depends on its addressee.

For implementators, the algebraic form seems more convenient. The definitions of denotation constructors may be written as mutually recursive procedures with procedural parameters (cf. Sec. 4.4.5) and the definition of semantics as a mutually recursive set of procedures that call the former procedures.

In turn, for a language user (a programmer) direct semantics seems more convenient since the meaning of each syntactic constructions is describe explicitly and totally in one definition.

More on a **Lingua** manual in [29].

4.8 Main milestones on the way to language implementation

Once the designers of a language complete five steps leading to a denotational model of the language (see Sec. 3.5), implementors may go into the play.

The creation of language implementation consists in writing a procedure — or rather a system of mutually recursive procedures — that each sequence of characters from colloquial syntax, let it be colloquial-program, will elaborate in three steps:

1. restoring transformation that transforms colloquial syntax into concrete syntax,
2. parsing that transforms concrete syntax into abstract syntax, and at the same time checks if the elaborated program is syntactically correct,
3. interpretation (or compilation) which corresponds to turning abstract syntax into denotations, i.e., into executable code.

The first step performs a relatively simple transformation from colloquial program `colloquial-program` to concrete program `concrete-program`. Of course, during this transformation, error messages may be raised.

The second step is performed by a *syntax analyzer*, also called a *parser*, that constructs the co-image of `concrete-program` in the abstract syntax. Since the concrete syntax is isomorphic to abstract syntax, this transformation is unambiguously defined. From this perspective, the abstract syntax of our program constitutes its parsing tree.

If an attempt to building a parsing tree fails, then the user is informed that the elaborated program contains syntactic errors, which means that it does not belong to the language defined by the concrete grammar. Parsing procedure should also indicate the place (syntactic context) of the generated error.

The third step corresponds to program execution, which means that the program is either executed by an interpreter or that it is compiled and the compiled code is executed by an implementation platform. In the sequel of the book, we shall talk about interpreters, since they are intuitively closer to our denotational model, but most remarks about interpreters will equally concern compilers.

The implementor of a programming language has, therefore, to create three essential software tools:

1. a syntax analyzer that transforms colloquial syntax into concrete syntax,
2. a parser of concrete syntax into abstract syntax,
3. an interpreter (or compiler) of abstract syntax.

The second and third tasks should be performed by an algorithm whose inputs are the grammar of concrete syntax and the definitions of denotation constructors.

A language should be constructed in such a way that as many as possible of potential errors are detectable at the level of syntax analysis since it is much faster than program execution. We try, therefore, to describe possibly many language features at the syntactic level, which is done by creating sufficiently many carriers in the algebra of denotations. For instance, in a well-constructed language, its parser should detect a syntactic error in the expression

```
if y > 0 then y+1 else list-type number ee fi
```

where **else** is followed by a type expression rather than by a data expression⁶². On the other hand, on the syntactic level, we are not able to check if a given variable is of a given type. Consequently, this analysis must be performed at the level of execution, i.e., of semantics⁶³.

⁶² In some languages, e.g. in C, such a construction is acceptable.

⁶³ As a matter of fact type errors may be detected on the level of so called *static semantics*, where we compute only types (in our case bodies) without computing values. Such a solution was applied in the semantics of programming language Ada [15] in the framework of VDM methodology (Vienna Development Method) [13]. More about Ada in a foot note of Sec. 3.1.

5 LINGUA-1 — AN IMPERATIVE LANGUAGE WITHOUT PROCEDURES

Starting from this section, we shall develop successive languages from the **Lingua** series by extending each of them with new mechanisms. **Lingua-1** emerges from **Lingua-A** by adding the mechanisms of type- and variable declarations and instructions. Procedures will be discussed in Sec. 6.

5.1 Denotations

5.1.1 Denotational domains

Denotational domains of **Lingua-1** correspond to its future syntactic categories:

1. identifiers,
2. data expressions,
3. body expressions,
4. transfer expressions,
5. yoke expressions,
6. type expressions,
7. instructions,
8. declarations,
9. programs.

Carriers of the future algebra of denotations are the following:

<code>ide</code>	<code>: Identifier</code>	<code>= ...</code>	(5.1.1-1)
<code>ded</code>	<code>: DatExpDen</code>	<code>= State → ValueE</code>	data-expression denotations
<code>bed</code>	<code>: BodExpDen</code>	<code>= State → BodyE</code>	body-expression denotations
<code>tra</code>	<code>: TraExpDen</code>	<code>= Transfer</code>	transfer-expression denotations
<code>yok</code>	<code>: YokExpDen</code>	<code>= Yoke</code>	yoke-expression denotations
<code>ted</code>	<code>: TypExpDen</code>	<code>= State ↦ TypeE</code>	type-expression denotations
<code>ded</code>	<code>: DecDen</code>	<code>= State ↦ State</code>	declaration denotations
<code>ind</code>	<code>: InsDen</code>	<code>= State → State</code>	instruction denotations
<code>prd</code>	<code>: ProDen</code>	<code>= State → State</code>	program denotations

As was already mentioned earlier, denotations of data expressions are partial functions. Although in **Lingua-1** reachable denotations of data expressions are total function (due to the mechanism of errors), in **Lingua-2** they may be partial since expression may include functional procedures whose executions may enter infinite loops. In the case of instructions and programs partiality is always there, since **while** instructions and recursive procedures may generate infinite executions.

The first six domains cover applicative denotations and have been discussed in Sec. 4.4. The remaining concern *imperative denotations* and are discussed below. Their elements are built employing three groups of constructors: declaration-denotation constructors, instruction-denotation constructors, and program-denotation constructors. We shall start with the last one.

5.1.2 Conservative denotations

An imperative denotation dim is said to be *conservative* if two following conditions are satisfied:

1. dim is *error-state transparent*, which means that if is-error.sta then $\text{dim.sta} = \text{sta}$,
2. dim does not change bodies of values or pseudovalues assigned to variables, which means that for any state sta which does not carry an error and any variable identifier ide declared in that state if
 - dim.sta is defined and does not carry an error, and
 - $\text{Sde}[\text{ide}].\text{sta} = (\text{dat-1}, (\text{bod-1}, \text{yok-1}))$, and
 - $\text{Sde}[\text{ide}].(\text{dim.sta}) = (\text{dat-2}, (\text{bod-2}, \text{yok-2}))$
 then
 - $\text{bod-1} = \text{bod-2}$.

As we shall see, all reachable imperative denotations of **Lingua-1** and **Lingua-2** that do not involve error handling will be conservative⁶⁴. It is, of course, an engineering decision.

A constructor of imperative denotations is said to be *decent* if it transforms conservative denotations into conservative denotations. In the sequel, we shall make sure that all our constructors are decent.

5.1.3 Programs

In all languages of the **Lingua** family, a program consists of a declaration followed by an instruction. We have, therefore, only one constructor of programs:

$$\begin{aligned} \text{create-program} &: \text{DecDen} \times \text{InsDen} \mapsto \text{ProDen} \\ \text{create-program}(\text{ded}, \text{ind}) &= \text{ded} \bullet \text{ind} \end{aligned}$$

As we are going to see, both components of a program may be trivial (doing nothing), atomic, or composed. Trivial declarations will be allowed in the bodies of imperative procedures, and trivial instructions in the bodies of functional procedures⁶⁵. Of course, a stand-alone program with a trivial declaration and non-trivial instruction will generate an error 'identifier-not-declared'.

5.1.4 Declarations

Declarations modify environments. In **Lingua-1** we have four types of *atomic declarations*:

1. data-variable declarations (Sec. 5.1.4.1),
2. body-constant declarations (Sec. 5.1.4.2),
3. type-constant declarations (Sec. 5.1.4.2),
4. trivial declaration (Sec. 5.1.4.4).

⁶⁴ This situation will be changed in **Lingua-SQL** were we introduce instructions which may add a new column to a database table or remove a column from such a table (see Sec. 10)

⁶⁵ Both these solutions, although in a slightly different form, have been suggested to me by Andrzej Tarlecki.

Besides them, we have *structured declarations* that are built from atomic declarations by sequential composition.

We also introduce an operator that will serve to ensure that an identifier declared in a valuation cannot be at the same time declared in an environment and vice-versa. Denotationally, such an assumption is not necessary since — as we are going to see — every reference to an identifier will explicitly point to the state component where the identifier should be found. From a programmer’s view, however, allowing an identifier to point to more than one object, may contribute to errors in programs.

declared : Identifier \mapsto State \mapsto BooleanE

declared.ide.((tye, pre), (vat, err)) =
 err \neq ‘OK’ \rightarrow err
 tye.ide = ! **or** pre.ide = ! **or** vat.ide = ! \rightarrow tt
true \rightarrow ff

The predicate **declared** is satisfied for an identifier in a state (which does not carry an error), if this identifier has been bound in that state with a value, a type or a procedure.

5.1.4.1 Declarations of data variables

As we already know (Sec. 4.4.2) *data variables* or simply *variables* are identifiers with values or pseudo-values assigned to them in valuations. Variable declarations assign pseudo-values to an identifiers, i.e. assign types leaving data temporarily undefined. Values are assigned to variables by assignment instructions (Sec. 5.1.2).

declare-dat-var : Identifier x TypExpDen \mapsto DecDen

declare-dat-var.(ide, ted).sta =
 is-error.sta \rightarrow sta
 declared.ide.sta \rightarrow sta \blacktriangleleft ‘variable-declared’
let
 (env, (vat, ‘OK’)) = sta
 typ = ted.sta
 typ : Error \rightarrow sta \blacktriangleleft typ
true \rightarrow (env, (vat[ide/(Ω , typ)], ‘OK’))

If a state carries an error, then the declaration does not change the state. Otherwise, if in the current state the identifier has been already declared, then an error signal is raised. This means that no identifier can be declared twice (redeclared) in a program.

If our type expression generates an error, then this error is passed to the state. Otherwise, the valuation is modified by assigning a pseudo-value (Ω , typ) to **ide**. As we shall see in the sequel, variable declarations are the only imperative constructs that introduce pseudo-values to states. A variable with assigned value or pseudo-value (**dat**, typ) is said *to be of type typ*.

An identifier that is bound in the valuation of a state, to a value, or pseudo-value is said to be a *declared variable* in that state.

In the end, we can finally explain why in the algebra of composites all potential sorts of composites — such as the composites of numbers, booleans, words, lists, etc. — were integrated into one sort **Composite**. The cause of that decision can be seen only at the level of the algebra of denotations. If in that algebra we would introduce separate carriers of data-expression denotations for numbers, booleans, words, lists, etc., then we would need to add a different variable constructor for each of these carriers. Consequently, at the level of syntax, we would have to somehow “label” variables with sorts. Technically this is possible, but would be rather unpractical and probably has never been applied in modern programming languages⁶⁶. As a

⁶⁶ Except for some very early languages of the decade of 1950.

consequence, since we decided to put all data-expression denotations into one sort, there was no reason to assume different sorts in the algebra of composites.

5.1.4.2 Declarations of body constants

Body constants are identifiers with bodies assigned to them in type environments. We call them *constants* rather than *variables* since a body, once assigned to an identifier remains unchanged during the whole execution of a program. In the case of variables, their bodies do not change, but their yokes and values may be modified.

The following constructor creates the denotation of a body constant declaration:

$\text{declare-bod-con} : \text{Identifier} \times \text{BodExpDen} \mapsto \text{DecDen}$

```

declare-bod-con.(ide, bed).sta =
  is-error.sta      → sta
  declared.ide.sta  → sta ◀ 'identifier-not-free'
  let
    bod              = bed.sta
    ((tye, pre), sto) = sta
  bod : Error       → sta ◀ bod
  true            → ((tye[ide/bod], pre), sto)

```

As we see, body declarations modify only type environments, and, of course, may generate an error message.

An identifier that is bound in the type environment of a state to a body is said to be a *declared body constant* in that state.

5.1.4.3 Declarations of type constants

Type constants are identifiers with types assigned to them in type environments. Similarly to body constants, they are also called *constants* since a type once assigned to a constant remains unchanged during the whole execution of a program. In the case of variables, their yokes and values may be modified.

The following constructor creates the denotation of a type constant declaration:

$\text{declare-typ-con} : \text{Identifier} \times \text{TypExpDen} \mapsto \text{DecDen}$

```

declare-typ-con.(ide, ted).sta =
  is-error.sta      → sta
  declared.ide.sta  → sta ◀ 'identifier-not-free'
  let
    typ              = ted.sta
    ((tye, pre), sto) = sta
  typ : Error       → sta ◀ typ
  true            → ((tye[ide/typ], pre), sto)

```

An identifier that is bound in the type environment of a state to a type is said to be a *declared type constant* in that state.

5.1.4.4 Trivial declaration

Trivial declaration transforms a state into itself. We shall need it as an option in the bodies of imperative procedures (Sec. 6.3).

$\text{create-trivial-dec}().\text{sta} = \text{sta}$

5.1.4.5 Structured declarations

A *structured declaration* is a sequential composition of atomic declarations. The only constructor of structured declarations is, therefore, the following:

$\text{sequence-dec}(\text{ded-1}, \text{ded-2}) = \text{ded-1} \bullet \text{ded-2}$

5.1.5 Instructions

5.1.5.1 Sorts of instructions

Instructions modify stores. Similarly to declarations, they may be atomic or structured. In **Lingua-1** we have three sorts of *atomic instructions*:

1. assignment instructions (Sec. 5.1.5.2),
2. yoke-replacement instructions (Sec. 5.1.5.3),
3. trivial instruction (Sec. 5.1.5.4).

Structured instructions are built from atomic instructions using four constructors that correspond to:

1. sequential composition,
2. **if-then-else-fi** instructions,
3. **while-do-od** instructions,
4. error-handling instructions.

5.1.5.2 Assignment instruction

Now we are ready to define a constructor corresponding to assignment instructions which are fundamental for the imperative part of **Lingua**.

$\text{assign} : \text{Identifier} \times \text{DatExpDen} \mapsto \text{InsDen}$

```

assign.(ide, ded).sta =
  is-error.sta          → sta
  let
    ((tye, pre), (vat, 'OK')) = sta
  vat.ide = ?          → sta ◀ 'identifier-not-declared'
  ded.sta = ?          → ?                                     an infinite execution
  ded.sta : Error      → sta ◀ ded.sta
  let
    (dat-f, (bod-f, yok-f)) = vat.ide                          f – former value
    (dat-n, (bod-n, yok-n)) = ded.sta                          n – new value
  com                       = yok-f.(dat-n, bod-n)
  com : Error                → sta ◀ com
  bod-n ≠ bod-f              → sta ◀ 'inconsistent-bodies'
  com ≠ (tt, ('boolean')) ) → sta ◀ 'yoke-not-satisfied'
  let
    val-n = (dat-n, (bod-f, yok-f))
  true                       → ((tye, pre), (vat[ide/val-n], 'OK'))

```

Assignment instruction assigns a new value to a data variable in a state. This instruction generates an error in three following cases:

1. if the variable has not been declared (but it does not need to be initialized),
2. if the body of variable's type does not coincide with the body of the assigned value,
3. if the assigned value does not satisfy the yoke assigned to the variable.

Note that the yoke of the variable yok-f remains unchanged independently of the yok-n of the new value. However, to prove that the new value satisfies the yoke of the variable it is enough to prove that yok-n implies yok-f .

As we also see, assignment cannot change variable's type. This is, of course, an engineering decision. It has been taken to assure that every modification of variable's yoke must be explicit in a program. In **Lingua**, to make such a modification, we have to use a yoke-replacement instruction described in Sec. 5.1.5.3.

5.1.5.3 Yoke-replacement instruction

Yoke replacement is similar to assignment with the difference that this time we do not change a composite but yoke. Similarly to assignment instructions also yoke-replacement instructions belong to the category of atomic instructions.

```

replace-yo : Identifier x Yoke  $\mapsto$  InsDen
replace-yo.(ide, yok-n).sta =
  is-error.sta  $\rightarrow$  sta n – new
let
  (env, (vat, 'OK')) = sta
  vat.ide = ?  $\rightarrow$  'identifier-not-declared'
let
  ((com-f, yok-f) = vat.ide f – former
  yok-n.com-f  $\neq$  (tt, ('boolean'))  $\rightarrow$  'yoke-not-satisfied'
let
  val-n = (com-f, yok-n)
true  $\rightarrow$  (env, vat[ide/val-n], 'OK')
```

New value has an old composite and a new yoke. The latter must be satisfied by the old composite, since otherwise (com-f, yok-n) would not be a value. This instruction has been introduced mainly for the sake of **Lingua-SQL** (Sec. 10.9.6)⁶⁷.

It is worth noticing in this place that in the algebra of types, we have a similar constructor **ty-replace-yo**, which, however, is a constructor of types rather than of instruction denotations. We need both of them. The **ty-replace-yo** is required to build types statically in declarations, whereas **replace-yo** is necessary to change the type of variables dynamically in instruction.

5.1.5.4 Trivial instruction

Trivial instruction is an identity transformation of a state into itself. As we are going to see, it will be useful in defining the declarations of functional procedures (Sec. 6.5). The denotation of this instruction is created by the following constructor:

```

create-trivial-ins :  $\mapsto$  InsDen
create-trivial-ins().sta = sta
```

5.1.5.5 Structured instructions

Structured instructions are built from atomic instructions using four constructors mentioned in Sec.5.1.5.1. In the present section, we shall define these constructors.

The simplest constructor of structured instructions corresponds to their sequential composition, and is defined as follows:

```

sequence-ins : InsDen x InsDen  $\mapsto$  InsDen
sequence-ins.(ind-1, ind-2) = ind-1 • ind-2
```

Sequentially composed instructions are executed one after another. Conditional composition is defined as follows:

```

if : DatExpDen x InsDen x InsDen  $\mapsto$  InsDen
if.(ded, ind-1, ind-2).sta =
  is-error.sta  $\rightarrow$  sta
  ded.sta = ?  $\rightarrow$  ?
```

⁶⁷ This very general form of yoke-replacement has been chosen for the sake of simplicity. In real situations one should think about more specific replacements, as e.g. by conjunctively adding a new yoke to the former.

```

ded.sta : Error    → sta ◀ ded.sta
let
  (dat, (bod, yok)) = ded.sta
bod ≠ ('boolean') → sta ◀ 'boolean-expected'
dat = tt          → ind-1.sta
true           → ind-2.sta

```

It is to be emphasised that due to **while** loops (see below) the execution of both component instructions may be infinite, which means that the states `ind-1.sta` or `ind-2.sta` may be undefined. If `dat = tt` and `ind-1.sta` is undefined then the terminal state of the conditional instruction is undefined as well, and in the opposite case the final state is undefined if `ind-2.sta` is undefined.

while : $\text{DatExpDen} \times \text{InsDen} \mapsto \text{InsDen}$

```

while.(ded, ind).sta =
  is-error.sta    → sta
  ded.sta = ?     → ?
  ded.sta : Error → sta ◀ ded.sta
let
  (dat, (bod, yok)) = ded.sta
bod ≠ ('boolean') → sta ◀ 'boolean-expected'
dat = ff          → sta
true           → (ind • [while.(ded, ind)]).sta

```

In this definition we have to do with a fixed-point equation. Notice however that the unique (least) solution of this equation is not **while** constructor, but the effect of its application to its arguments, i.e. `while.(ded, ind)`.

Due to this construction, the denotations of instructions can be partial functions. In the sequel, where imperative and functional procedures are introduced, the partiality of `while.(ded, ind)` may take place in three different situations:

1. the boolean expression corresponding to `ded` includes a functional procedure, the call of which generates an infinite execution,
2. the instruction represented by `ind` generates an infinite execution,
3. the “main loop” runs infinitely.

Comment 5.1.5.5-1 In the definition of **while** we have to do with a fixed-point equation in **InsDen**, which is a CPO of partial functions (Sec. 2.6). For any pair **(ded, ind)** the solution of our fixed-point equation is a state-to-state function:

while.(ded, ind) : State → State

Of course, for every such a pair **(ded, ind)** we have to do with a different equation. To be sure that solutions of such equations exist, we have to prove that the right-hand sides of such equations are continuous in the CPO of partial functions **State → State**. To do that, let us introduce the following notations:

```

NotOK = {(sta, sta) | (1) satisfied}
ExpEr = {(sta, sta ◀ ded.sta) | (3) not (1) and (2)}
NotBoo = {(sta, sta ◀ 'boolean-expected') | (4) not (1) and (2) and (3)}
FF     = {(sta, sta) | (5) not (1), (2), (3) and (4)}
TT     = {(sta, sta) | not (1), (2), (3), (4) and (5)}

```

Now, our definition maybe written as a fixed-point equation:

X = NotOK | ExpEr | NotBoo | FF | TT • ind • X

Since the operators `|` and `•` are continuous, the least solution of that equation exists, and since the coefficients of that equations have mutually disjoint domains, from Theorem 2.6-1 we may conclude that its solution is a function and has the form:

X = (TT • Din)* • (NotOK | ExpEr | NotBoo | FF)

The last structured constructor concerns the *error-handling mechanism*. It allows building an error-handling mechanism into programs, a mechanism that is activated whenever a particular error message is generated.

$\text{if-error} : \text{DatExpDen} \times \text{InsDen} \rightarrow \text{InsDen}$

```

if-error.(ded, ind).sta =
  let
    (env, (vat, err)) = sta
    err = 'OK'      → sta
  let
    sta-1 = (env, (vat, 'OK'))
    ded.sta-1 = ?  → ?
  let
    val = ded.sta-1
    val : Error   → sta ◀ val © 'error-handling-not-executed'
  let
    (dat, (bod, yok)) = val
    bod ≠ ('word')  → sta ◀ 'word-expected' © 'error-handling-not-executed'
    dat ≠ err       → sta ◀ dat © 'error-handling-not-executed'
    ind.sta-1 = ?   → ?
  let
    sta-2 = ind.sta-1
    is-error.sta-2 → sta ◀ error.sta-2 © 'error-handling-not-executed'
  true      → sta-2

```

If the input-state does not carry an error, then it becomes the output state, since there is no cause to handle an error.

In the opposite case, a temporary state **sta-1** is created by the removal of the error from **sta**. In this state, we compute the value of the expression **ded** whose value should be the handled error. If this computation does not terminate, then the execution of the whole instruction does not terminate either. Otherwise, if the result of that computation is an error or a value that does not carry a word, then an appropriate error message is generated together with the additional message 'error-handling-not-executed'.

In the opposite case, if the word carried by **val** — the error to be handled — is different from the initial error, then the output state is loaded with an appropriate message.

In the opposite case, the error-handling instruction **ins** is executed in the temporary state **sta-1**. If during this execution, an error is generated, then it is signaled together with the information 'error-handling-not-executed'.

In the opposite case the “handled” state becomes the output state.

As we see, the expression that appears in an error-handling instruction must evaluate to a word value. If that word coincides with the current error message, then the “internal” instruction is executed.

It is to be stressed that the above constructor should be regarded only as an example showing that error-handling mechanisms may be described in our model. In no way, it should be considered as a pattern for error handling. Other examples of such mechanisms are shown in sections 6.3.2 and 10.9.6.4.

5.2 Syntax

Since **Lingua-1** is being built as an extension of **Lingua-A**, we shall describe only these elements of the syntax of **Lingua-1** that do not appear in **Lingua-A**.

5.2.1 Abstract syntax

Programs

```
prg : ProgramA =
  create-program (DeclarationA , InstructionA)
```

Declarations

```
dec : DeclarationA =
  declare-dat-var (Identifier , TypExpA) |
  declare-bod-con (Identifier , BodExpA) |
  declare-typ-con (Identifier , TypExpA) |
  create-trivial-dec.() |
  sequence-dec. (DeclarationA, DeclarationA)
```

Instructions

```
ins : InstructionA =
  assign (Identifier , DatExpA) |
  replace-yo (Identifier , YokExpA) |
  create-trivial-ins () |
  if (DatExpA , InstructionA , InstructionA) |
  if-error (DatExpA , InstructionA) |
  while (DatExpA , InstructionA) |
  sequence-ins (InstructionA , InstructionA)
```

5.2.2 Concrete syntax

Programs

```
prg : Program =
  (Declaration ; Instruction)
```

Declarations

```
dec : Declaration =
  let Identifier be TypExp te1 |
  set-body Identifier as BodExp tes |
  set-type Identifier as TypExp tes |
  (Declaration ; Declaration) |
  skip-d
```

Instructions

```
ins : Instruction =
  Identifier := DatExp |
  yoke Identifier := YokExp ekoy |
  skip-i |
  if DatExp then Instruction else Instruction fi |
  if-error DatExp then Instruction fi |
  while DatExp do Instruction od |
  (Instruction ; Instruction)
```

5.2.3 Colloquial syntax

Here are some examples of possible colloquialisms. First concerns sequential composition of:

1. a declaration with an instruction (to create a program),
2. two declarations,
3. two instructions

In all these cases we can skip parentheses.

Second, in type declarations and variable declarations with trivial yokes `true`, we can skip the yoke part of the expression. For instance instead of writing

```
set-type list-of-names
  as
    type
      list string ee
      with true
    ee
tes
```

we may write

```
set-type list-of-names
  as
    type
      list string ee
tes
```

and analogously for variable declarations.

Third, variables-declarations of the same type may be grouped into one declaration with many variables, e.g. instead of writing

```
let x be number tel;
let y be number tel;
let z be number tel
```

we write

```
let x, y, z be number tel
```

and analogously for body and type declarations.

Comment 5.2.2-1 In building concrete syntax for **Lingua-1**, I have applied some notational conventions known to me from the “old times”, which in my opinion improve the clarity of programs and thus contribute to a less number of errors made by programmers. They are the following:

1. For an assignment, I use „:=” rather than an equality „=” as in some other languages. The equality symbol is reserved for a comparison predicate.
2. I use closing parentheses like e.g., **fi** for **if** and **od** for **do** since my experience shows that this contributes to better clarity of programs.
3. Hierarchical carriage returns and spaces help in exposing the structure of programs, however using them as parentheses (as, e.g., in Phyton) may be error-prone resulting from an erroneous use of the Del-key. As a mathematician, I also cannot accept the fact that a hidden formatting-sign is an element of syntax. It is, however, convenient to use carriage returns and indentations freely, i.e., without interfering with the meanings of programs. In **Lingua**, they are removed by the restoring transformation.

5.2.4 An example of a simple program

Here is an example of a simple program that creates a record. Under each part of that program, I give an explanation of its meaning.

```
set-body register_body as
  array number ee
tes
```

Identifier `register_body` is declared as an array-body constant with body `(‘A’, (‘number’))`.

```

set-body employee_body as
  record
    ch_name, fa_name as string,
    birth_year as number,
    awards as register_body
  ee
tes;

```

Identifier `employee_body` is declared as a body constant with record body

```

(‘R’, [‘ch_name’ / (‘word’), ‘fa_name’/(‘word’), ‘birth_year’/(‘number’), ‘awards’/(‘A’, (‘number’))])

```

Having declared body constants we can declare corresponding type constants.

```

set-type register_type as
  body register_body
  with all-of-arr value < 2010 ee
tes

```

Identifier `register_type` has been declared as a type constant with a body assigned to `register_body` and a yoke which requires that each element of the array is less than 2010.

```

set-type employee_type as
  body employee_body
  with (record value at birth_year ee < 2000) ee
tes

```

Identifier `employee_type` is declared analogously as the former. Having declared types we can declare variables of these types

```

let salesman be employee_type ee
let awards_Smith be register_type ee

```

Now, we can initialize our variables

```

awards_Smith := array [1995, 1999, 2007]

salesman :=
  record
    ch-name      := ‘John’
    fa-name      := ‘Smith’
    birth-year   := 1968
    awards       := awards_Smith
  ee

```

5.3 Semantics

The definition of **Lingua-1** semantics consists of the definition of the semantics of **Lingua-A** (Sec. 4.6) extended by definitional clauses for the imperative part of the language:

```

Sde : Declaration  $\mapsto$  DecDen
Sin : Instruction  $\mapsto$  InsDen
Spr : Program  $\mapsto$  ProDen

```

The definitions of these semantic functions are given in an algebraic form.

Declarations

$\text{Sde} : \text{Declaration} \mapsto \text{DecDen}$ i.e.

$\text{Sde} : \text{Declaration} \mapsto \text{State} \mapsto \text{State}$

$\text{Sde}.[\text{let } \text{id} \text{ be } \text{tex}] = \text{data-variable.}(\text{Sid}.[\text{id}], \text{Sty}.[\text{tex}])$

$\text{Sde}.[\text{set } \text{id} \text{ as } \text{tex}] = \text{declare-typ-con.}(\text{Sid}.[\text{id}], \text{Sty}.[\text{tex}])$

$\text{Sde}.\text{skip-d} = \text{create-trivial-dec.}()$

$\text{Sde}.[(\text{dec-1}; \text{dec-2})] = \text{sequence-vde.}(\text{Sde}.[\text{dec-1}], \text{Sde}.[\text{dec-2}])$

Instructions

$\text{Sin} : \text{Instruction} \mapsto \text{InsDen}$ i.e.

$\text{Sin} : \text{Instruction} \mapsto \text{State} \rightarrow \text{State}$

$\text{Sin}.[\text{id} := \text{dae}] = \text{assign.}(\text{Sid}.[\text{id}], \text{Sw}.[\text{dae}])$

$\text{Sin}.[\text{if } \text{dae} \text{ then } \text{ins-1} \text{ else } \text{ins-2} \text{ fi}] = \text{if.}(\text{Sw}.[\text{dae}], \text{Si}.[\text{ins-1}], \text{Si}.[\text{ins-2}])$

$\text{Sin}.[\text{if-error } \text{dae} \text{ then } \text{ins-1} \text{ fi}] = \text{if-error.}(\text{Sw}.[\text{dae}], \text{Si}.[\text{ins-1}])$

$\text{Sin}.[\text{while } \text{dae} \text{ do } \text{ins} \text{ od}] = \text{while.}(\text{Sw}.[\text{dae}], \text{Si}.[\text{ins}])$

$\text{Sin}.[(\text{ins-1}; \text{ins-2})] = \text{sequence-ins.}(\text{Si}.[\text{ins-1}], \text{Si}.[\text{ins-2}])$

Programs

$\text{Spr} : \text{Program} \mapsto \text{ProDen}$ i.e.

$\text{Spr} : \text{Program} \mapsto \text{State} \rightarrow \text{State}$

$\text{Spr}.[(\text{dec}; \text{ins})] = \text{create-program.}(\text{Sde}.[\text{dec}], \text{Sin}.[\text{ins}])$

6 LINGUA-2 — PROCEDURES

6.1 An introduction to a model of procedures

6.1.1 Procedures from a historical perspective

The concept of a procedure appeared in programming languages in the decade of 1950. Initially, procedures were just lists of instructions communicating with a hosting program through global variables. Later, to increase the universality of procedures, they were equipped with parameter-passing mechanisms and with a mechanism of local-variable declarations. Procedures understood in this way are named *imperative procedures*, and correspond to state-to-state functions. Consequently, the calls of such procedures belong to the category of instructions.

Another type of procedures was introduced under the name of *functional procedures* or just *functions*. The calls of these procedures belong to the category of expressions since they return values rather than states.

The most popular high-level language of the decades 1950/1960 was Fortran. In this language, procedures could call other procedures but not themselves. Procedures that may call themselves were introduced in Algol 60 under the name of *recursive procedures*. The creators of Algol 60 went even one step further, allowing procedures to take other procedures — and even themselves (!) — as parameters (cf. Sec. 3.1). The self-applicability of procedures as parameters was, however, abandoned rather quickly, and did not appear later in programming languages. On the other hand, recursion turned out to be an advantageous vehicle and today is present in many languages. In some of them, procedures may take other procedures as parameters, but not themselves (see Sec. 6.6).

It is worth mentioning in this place that at the turn of decades 1950 and 1960, Polish scientists have developed and implemented a programming language SAKO — System Automatycznego Kodowania (A System of Automatic Coding) — that was similar to Fortran. Its compiler was implemented on a Polish computer XYZ constructed in Zakład Aparatów Matematycznych PAN (Department of Mathematical Apparatuses) — research unite of Institute of Mathematics of Polish Academy of Sciences. That was the first computer that I learned to program as a student. Its first version was equipped with an operational memory of 1024 bytes, i.e., 1 KB (disks were not known yet) and was later expanded by a magnetic drum with — as far as I can remember — 5 KB.

At that time, programmers were instructed that to “intellectually” control the behavior of a program, the latter should not exceed one paper-sheet of A4 size. In the case of larger programs, that principle was implemented by writing procedures that were calling other procedures. That style was later called *structured programming* (cf. Sec. 3.1).

Structured programming was introduced not only to help programmers in a better understanding of their programs but also to prove program-correctness by induction based on program structure. So far, however, this technique is rather far for a full practical realization (cf. Sec. 7.1).

6.1.2 Procedures versus structured programming

In programming languages with procedures, procedures may call other procedures or even themselves. This mechanism allows building programs in the following structural way:

1. The main program calls one *main procedure*, which calls *subprocedures of the first level*.

2. Subprocedures of the first level call *subprocedures of the second level*.
3. ...

The number of successive levels is theoretically arbitrary.

In the simplest case — which appears most frequently — procedures constitute a tree-like structure (Fig. 6.1-1). The main procedure MP calls subprocedures SP1 and SP2 the first calls in turn SP3, SP4, and SP5.

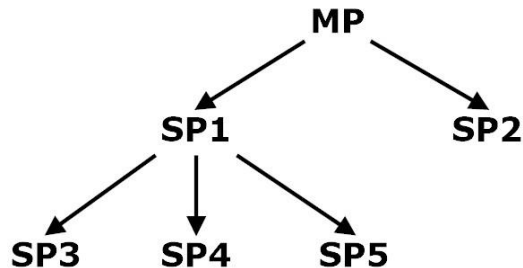


Fig. 6.1-1 A tree of procedures without recursion

It may also be the case that a procedure calls a higher-level procedure or itself. Such a situation is illustrated in Fig. 6.1-2.

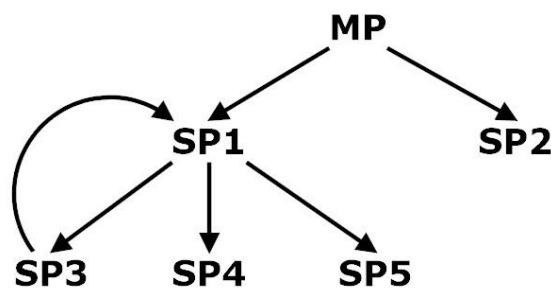


Fig. 6.1-2 A graph of procedures with recursion

If in the body of a procedure an interpreter encounters a call of this procedure, then basically two types of reactions are possible:

- an error message 'procedure-undeclared' is generated,
- a copy of the called procedure is activated.

The second case, which is today rather common in programming languages, is known as *recursive call* of a procedure. If a procedure calls itself directly, i.e., in its own body, then we have to do with a *simple recursion*. If, however, SP1 calls SP3 and SP3 calls SP1, then we have to do with *mutual recursion*. Of course, a cycle of procedures calling one another may have more than just two elements.

6.1.3 Imperative procedures in a denotational framework

Although recursion is today a rather common standard in high-level programming languages, its technical details may differ from one language to another. For the sake of our investigations, we assume a certain more-or-less universal model chosen in such a way that it leads to relatively simple correction-proof-rules.

Procedures in our model split into two categories: *imperative procedures* or just *procedures* and *functional procedures* or just *functions*. The calls of the former belong to the category of instructions, the calls of the latter — to the category of expressions. Let us concentrate first on imperative procedures. Functional procedures will be discussed in Sec. 6.5.

Imperative procedures may be seen as named instructions with additional mechanisms that allow to use them repeatedly in many different contexts:

- they may be saved in procedure-environments,
- they may use local variables, types, and procedures that are *not visible* outside of a *procedure-body*,
- they may receive lists of values that are used to initialize local variables; this mechanism is known as *called-by-value actual parameters* or as *actual value-parameters*,
- they may receive lists of variables known as *called-by-reference actual parameters* or as *actual reference-parameters*; the initial values of these parameters are passed to procedures, and their terminal values are exported back to the hosting program.

Besides all these operational features of procedures, there is one essential difference between procedures and instructions — procedures have no syntactic counterparts. In commonly known programming languages, procedures do not appear as syntactic objects, and even more — they do not appear as independent concepts at all. The authors of manuals talk about procedure declarations and procedure calls but not about procedures as such. This awkward situation is caused by the fact that manuals are usually concentrated on syntax⁶⁸.

However, talking about bodies, declarations, and calls of “beings” that have not been defined not only conflicts with mathematical good-practice but may also lead to a poor understanding of language mechanisms.

In **Lingua-2** procedures constitute a separate domain which, however, is not a carrier of our algebra of denotations. It is why procedures have no syntactic counterparts. Consequently, we talk about procedures “as such”, rather than about procedure denotations.

In order to include imperative procedures in our denotational model, we define three new categories of denotational beings:

<i>imperative procedures</i>	—	total functions that given parameters return partial functions that modify stores,
<i>the denotations of imp.-procedure declarations</i>	—	total functions that modify states by assigning a (just declared) procedure to an identifier in the environment,
<i>the denotations of imp.-procedure calls</i>	—	partial functions that modify states by executing a procedure that modifies the store of the input state of the call.

To start the extension of our model with imperative procedures we define two new domains (of which only the second will become a carrier of our algebra of denotations):

$$\begin{array}{ll} \text{ipr} : \text{ImpPro} = \text{AcPaDe} \times \text{AcPaDe} \mapsto \text{Store} \rightarrow \text{Store} & \text{imperative procedures} \\ \text{apd} : \text{AcPaDe} = \text{Identifier}^c & \text{list of actual-parameter denotations} \end{array}$$

To avoid a self-applicability of procedures we define them as functions which given actual-parameter denotations return store-to-store functions.

As we are going to see a little later, declarations of imperative procedures will take two lists of formal parameters called respectively *formal value-parameters*, and *formal reference-parameters*. The domain of formal-parameter denotations is defined as follows⁶⁹:

⁶⁸ From a denotational perspective this situation is, however, not as awkward as one could expect. Notice that other storable objects such as values and types do not have syntactic counterparts either. At the level of syntax they are represented by expressions. Analogously to that, procedures are represented by calls.

⁶⁹ Note that here we talk about formal-parameter denotations — rather than just about formal parameters — which means that they will have their syntactic counterparts. These counterparts will be tuples of pairs consisting of an

$\text{fpd} : \text{FoPaDe} = (\text{Identifier} \times \text{TypExpDen})^{\text{c}*}$ list of formal-parameter denotations

Formal-parameters denotations include type-expression denotations since, in the declarations of procedures, we indicate the types of their future actual-parameters.

In the sequel, we shall require that formal parameters of a procedure do not include repetition. For that sake, we define the following predicate:

$$\begin{aligned} \text{repetitions-in-for-par} : ((\text{ide-1}, \text{ted-1}), \dots, (\text{ide-n}, \text{ted-n})) = \\ (\exists i, j)(\text{ide-i} = \text{ide-j}) &\rightarrow \text{tt} \\ \text{true} &\rightarrow \text{ff} \end{aligned}$$

The domain of declaration denotations remains unchanged:

$$\text{ded} : \text{DecDen} = \text{State} \mapsto \text{State}$$

we shall only enrich the list of its constructors, and consequently its reachable subdomain.

For the simplicity of our model, and especially for the simplicity of program construction rules, we have assumed that actual parameters of procedures must be identifiers rather than arbitrary expressions. As we shall see later, expressions will include functional-procedure calls, which in turn may contain calls of imperative procedures, and which, themselves, may be recursive. If actual parameters could be arbitrary expressions, then one could write a procedure that calls itself recursively when calculating its own parameters. Denotationally, this is (probably?) feasible, but a corresponding program-construction rule would become pretty complicated. More on that issue in Sec. 6.5.1

After having defined domains for procedures, we can proceed to the definitions of their constructors. We assume that all constructors defined in **Lingua-1** are available in **Lingua-2**.

According to our general rule about the series of **Lingua** languages (Sec. 3.3), **Lingua-2** emerges from **Lingua-1** by adding new carriers and new constructors to the algebra of denotations. Additionally, the carrier of instruction denotations gets new reachable elements that correspond to procedure calls, and the carrier of declarations gets new reachable elements that correspond to procedure declarations⁷⁰.

At the end of this section, notice that in the definition of **ImpPro**, we do not have an illegal fixed-point recursion since procedures do not take states as arguments but only stores⁷¹. It is why stores have been introduced as a separate component of a state. If we had assumed the equation

$$\text{ImpPro} = \text{AcPaDe} \times \text{AcPaDe} \mapsto \text{State} \rightarrow \text{State}$$

then together with the equations

$$\text{State} = (\text{TypEnv} \times \text{ProEnv}) \times \text{Store}$$

$$\text{ProEnv} = \text{Identifier} \Rightarrow \text{ImpPro}$$

we would have an illegal fixed-point equation since the operators „ \mapsto ” and „ \Rightarrow ” are not continuous (Sec. 2.7). As we are going to see, our model of procedures allows recursion but does not allow procedures that take themselves as parameters. Procedures that can take other procedures as parameters — but not themselves — are discussed in Sec. 6.6

6.2 Communication between imperative procedures and programs

In the descriptions of procedural mechanisms, we shall use some concepts referring to the fact that procedures are created when they are declared and are executed when they are called. In respect to that, we shall talk

identifier and a type expression. Such tuples will be called *formal parameters*. A similar situation takes place for actual parameters, although now parameter denotations are identical with their syntax and are just identifiers.

⁷⁰ The carriers themselves do not change since they remain the $\text{State} \rightarrow \text{State}$ domain. The same will concern expression denotations when we introduce functional-procedure calls.

⁷¹ That solution has been introduced by Andrzej Tarlecki and myself in 1983 (see [33]).

about *declaration-time states* and *call-time states*, respectively⁷². Traditionally by a *procedure body*, we shall mean a program that is executed when a procedure is called. In turn, by a *procedure content*, we shall mean a procedure body equipped with two parameter-passing mechanisms:

1. passing the values of actual parameters to formal parameters before the execution of the body,
2. returning the values of formal reference parameters to formal parameters after the execution of the body.

As I have already announced, in **Lingua-2**, there will be no global variables in procedures. It is not a mathematical necessity but an engineering decision. The intention is that the head of a procedure-call describes explicitly and completely the communication mechanisms between a procedure and the hosting program. That solution may seem restrictive but — in my opinion — guarantees a better understanding of program functionality by programmers and also simplifies program-construction rulers.

6.2.1 How does it work?

An execution of a procedure call may be symbolically split into five stages illustrated in Fig. 6.2-1. (technical details in Sec. 6.3).

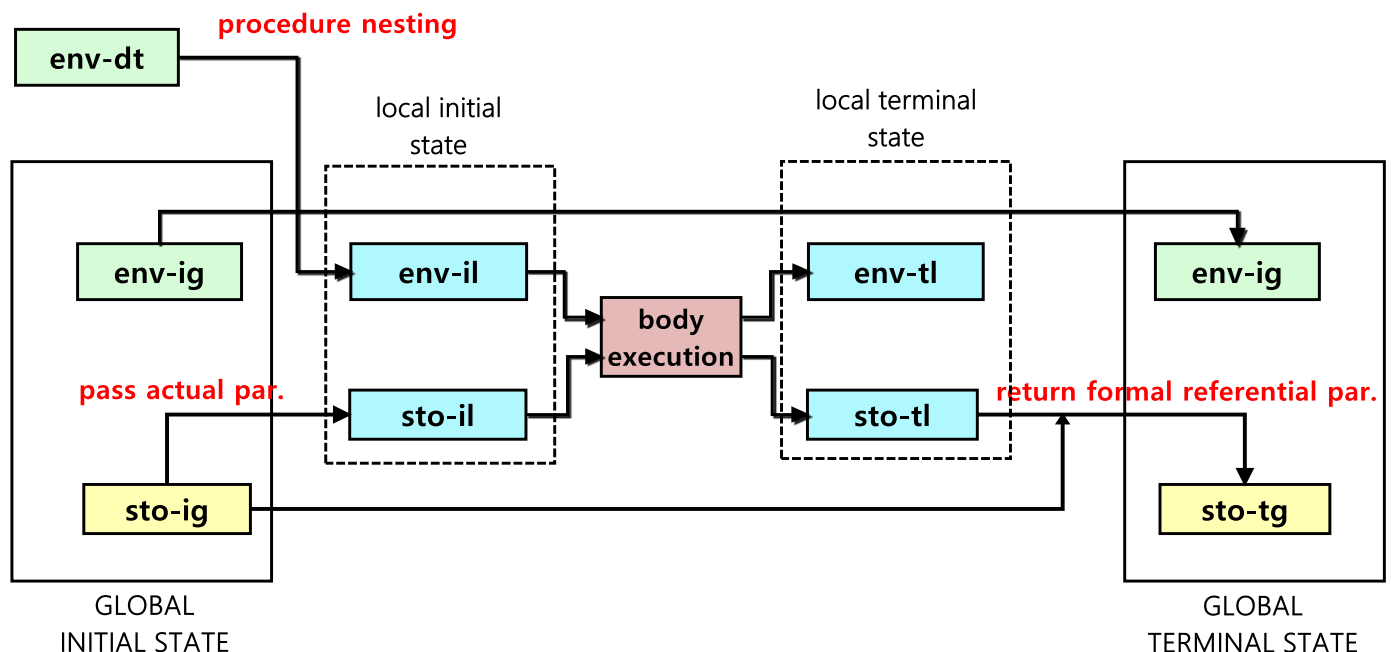


Fig. 6.2-1 The execution of a procedure content

1. **Getting the called procedure from the environment.** We check if the procedure identifier (procedure's name) is assigned in the global initial-environment to an imperative procedure.
2. **Checking the compatibility of actual parameters with formal parameters.** The initial global state consists of:
 - a. an *initial global environment* `env-ig`,
 - b. an *initial global store* `sto-ig = (vat-ig, err)`

If `err` \neq 'OK', then the initial global state is returned by the procedure call, i.e., it becomes the terminal global state. In the opposite case, actual parameters are checked for compatibility with formal parameters, and their values are passed to the local initial state.

3. **The creation of an *initial local state*** — that state consists of:

⁷² These ideas, similarly to a few others, have been borrowed from M. Gordon [53].

- a. *initial local environment* env-il created from the declaration-time environment by nesting in it the called procedure; this nesting is necessary to enable recursive calls (Sec. 6.3.2),
 - b. *initial local valuation* vat-il carrying only formal parameters with assigned values of corresponding actual parameters; to get these values, we refer to initial global valuation val-ig .
4. **The transformation of the local initial state** by executing the procedure body (a program). If this execution terminates, then the local terminal state consists of:
- a. *terminal local environment* env-tl ,
 - b. *terminal local store* $\text{sto-tl} = (\text{val-tl}, \text{err-tl})$.
- If $\text{err-tl} \neq \text{'OK'}$, then a global terminal state is created from the initial global state by loading to it err-tl . Notice that in this case, the terminal local-environment and terminal local-store are “abandoned”. Otherwise, the terminal global state is created.
5. **The creation of the terminal global state** — that state consists of:
- a. *initial global environment* env-ig ; notice that terminal local environment env-tl is “abandoned”,
 - b. *terminal global store* sto-tg created from initial global store sto-ig by passing to it the values of formal referential parameters (stored in sto-tl) and assigning them to the corresponding actual referential parameters.

Notice that the initial local environment “inherits” all types and procedures from the declaration-time environment. Procedure body may create its private local types, variables, and procedures, but after the completion of the call, they cease to exist since the hosting program continues its execution with the initial global environment.

It is to be stressed that the procedure body may access only that part of the environment, which was created before the procedure declaration.

Of a similar character is the local valuation that is created only for procedure-execution time; however, in this case, the values or reference-parameters stored in it are eventually returned to the terminal global valuation.

Summarising visibility rules concerning procedure call:

1. the only variables visible during the execution of a procedure-body are formal parameters plus variables local to the body (declared in it),
2. the only types and procedures visible in procedure-body are declaration-time types and procedures plus locally declared ones,
3. variables, types, and procedures declared in procedure-body are not visible outside the procedure call.

All these choices are not mathematical necessities but pragmatic engineering decisions dictated by the intention of making our model relatively simple. This decision should contribute to the simplicity of proof rules and a better understanding of the program’s behavior by language users.

6.2.2 Constructors of parameter denotations

Since procedure constructors will take lists of parameters as arguments, we have to define constructors of the denotations of parameter lists in the first place. Lists of actual-parameter denotations are tuples of identifiers (possibly empty), and therefore they can be built by three following constructors:

$\text{create-empty-act-par-den} : \mapsto \text{AcPaDe}$

$\text{create-empty-act-par-den}().() = ()$

$\text{create-single-act-par-den} : \text{Identifier} \mapsto \text{AcPaDe}$
 $\text{create-single-act-par-den.}(\text{ide}) = (\text{ide})$

$\text{add-act-par-den} : \text{AcPaDe} \times \text{Identifier} \mapsto \text{AcPaDe}$
 $\text{add-act-par-den.}(\text{apd}, \text{ide}) = \text{apd} \mathbb{C} (\text{ide})$

where \mathbb{C} denotes a Cartesian concatenation of tuples (see Sec. 2.1.4), and (ide) denotes a one-element tuple that consists of ide . Analogously we define constructors of the lists of formal-parameter denotations:

$\text{create-empty-for-par-den} : \mapsto \text{FoPaDe}$
 $\text{create-empty-for-par-den.}() = ()$

$\text{create-single-for-par-den} : \text{Identifier} \times \text{TypExpDen} \mapsto \text{FoPaDe}$
 $\text{create-single-for-par-den.}(\text{ide}, \text{ted}) = ((\text{ide}, \text{ted}))$

$\text{add-for-par-den} : \text{FoPaDe} \times (\text{Identifier} \times \text{TypExpDen}) \mapsto \text{FoPaDe}$
 $\text{add-for-par-den.}(\text{fpd}, (\text{ide}, \text{ted})) = \text{fpd} \mathbb{C} (\text{ide}, \text{ted})$

At the end of this section, a technical explanation seems necessary. It concerns the definitions of both domains AcPaDe and FoPaDe , but we shall discuss only the first of them since the argument for the second is analogous.

The issue to be discussed concerns an observation that to generate AcPaDe , we do not need the constructor $\text{create-single-act-par-den}$. Instead of using it we can use the first and the third constructor in assuming that

$() \mathbb{C} \text{apd} = \text{apd}$

and therefore

$() \mathbb{C} (\text{ide}) = (\text{ide})$

Such a solution, however, leads to a problem, if at the level of concrete syntax, we want to explicitly mark an empty list, e.g., by a keyword `empty-ap`. In that case, we cannot generate a one-identifier list but necessarily a two-element list, e.g. $(\text{empty-ap}, \text{size})$. Another solution could be that in concrete syntax, we replace `empty-ap` by empty word ϵ , but in that case, we violate our principle that all decisions of a programmer must be explicit in the program's syntax.

6.2.3 The compatibility of parameter-lists

When an imperative procedure is called its formal parameters receive the values (typed data) of actual parameters, and in this way a local valuation is created. However, in order to make such a parameter-passing possible, the list of actual-parameter denotations of a procedure call must be compatible with the lists of formal-parameter denotations in the corresponding procedure declaration both as to their numbers and types. And of course, the identifiers which are used as actual parameters must be declared as data variables. In order to formalize these requirements we define two functions.

$\text{statically-compatible} : \text{FoPaDe} \times \text{FoPaDe} \times \text{AcPaDe} \times \text{AcPaDe} \mapsto \text{Error} \mid \{\text{'OK'}\}$
 $\text{statically-compatible.}(\text{fpd-v}, \text{fpd-r}, \text{apd-v}, \text{apd-r}) =$


```

let(for n, m, k, p ≥ 0)
  fpd-v   = ((ide-fv.i, ted-fv.i) | i=1;k)           list of formal value-parameter denotations
  fpd-r   = ((ide-fr.i, ted-fr.i) | i=1;n)           list of formal reference-parameter denotations
  apd-v   = (ide-av.i | i=1;p)                       list of actual value-parameter denotations
  apd-r   = (ide-ar.i | i=1;m)                       list of actual reference-parameters denotations
  are-repetitions.[(ide-fr.i | i=1;n) ⋈
                    (ide-fv.i | i=1;k)] → 'formal-par-repetitions'73
  are-repetitions.apd-r      → 'actual-reference-par-repetitions'
  n ≠ m or k ≠ p           → 'incompatible-numbers-of-parameters'
  true                     → 'OK'

```

In other words, lists of formal and actual parameter denotations of a procedure call are statically compatible if:

1. no formal parameter appears twice on a combined list of both sorts (value- and reference) parameters; a similar property of actual value-parameters is, of course, not required,
2. no actual reference parameter appears twice on the list of actual reference parameters,
3. the mutually corresponding lists of formal and actual parameter denotations are of the same lengths.

The defined property is called *static* since it can be checked at compilation-time, i.e., before program execution. Notice that “statically” does not mean “syntactically”! Static compatibility cannot be described by a grammar.

The next compatibility function refers to valuations and type environments and therefore is dynamic since its execution is possible only during the execution of a program. Also here we compare the denotations of formal parameters with the denotations of actual parameters.

```

dynamically-compatible : FoPaDe x FoPaDe x AcPaDe x AcPaDe ↦
                                                                TypEnv x Valuation ↦ Error | {'OK'}

```

```

dynamically-compatible.(fpd-v, fpd-r, apd-v, apd-r).(tye, vat) =
let
  message = statically-compatible.(fpd-v, fpd-r, apd-v, apd-r)
  message : Error      → message
let (for n, m, k, p ≥ 0)
  fpd-v   = ((ide-fv.i, ted-fv.i) | i=1;k)           list of formal value-parameter denotations
  fpd-r   = ((ide-fr.i, ted-fr.i) | i=1;n)           list of formal reference-parameter denotations
  apd-v   = (ide-av.i | i=1;p)                       list of actual value-parameter denotations
  apd-r   = (ide-ar.i | i=1;m)                       list of actual reference-parameter denotations
  checking if actual value-parameters have been declared
  vat.(ide-av.i) = ?      → 'value-parameter undeclared'      for i = 1;p
  checking if actual reference-parameters have been declared
  vat.(ide-ar.i) = ?      → 'reference-parameter undeclared'   for i = 1;m
  computing the types of formal value-parameters
let
  sta      = ((tye, [ ]), (vat, 'OK'))                explanation below
  typ-fv.i = ted-fv.i.sta   for i = 1;k              types of formal-value-parameters
  typ-fr.i = ted-fr.i.sta   for i = 1;n              types of formal-reference-parameters
  typ-fv.i : Error          → typ-fv.i for i = 1;k
  typ-fr.i : Error          → typ-fr.i for i = 1;n
let

```

⁷³ Function **are-repetitions** (Sec. 2.1.4) has been defined for tuples, therefore its argument in this definition is a Cartesian composition ‘⋈’ of formal-reference and formal-value parameter-lists.

```

(bod-fv.i, tra-fv.i) = typ-fv.i  for i = 1;k
(bod-fr.i, tra-fr.i) = typ-fr.i  for i = 1;n
(∃ i) bod-fv.i ≠ bod-av.i → 'incompatible-bodies-of-value-parameters'
(∃ i) bod-fr.i ≠ bod-ar.i → 'incompatible-bodies-of-reference-parameters'
(∃ i) (tra-fv.i).((dat-av.i, bod-av.i) ≠ (tt, ('boolean'))) → 'yoke-not-satisfied-by-val'
(∃ i) (tra-fr.i).((dat-ar.i, bod-ar.i) ≠ (tt, ('boolean'))) → 'yoke-not-satisfied-by-ref'
true → 'OK'

```

Lists of formal and actual parameters are considered *dynamically compatible*, if:

1. they are statically compatible,
2. all actual parameters are declared; note that they do not need to be initialized⁷⁴,
3. all type expressions assigned to formal parameters of both categories evaluate to non-errors,
4. all bodies of mutually corresponding formal- and actual-parameter values of both categories are identical; the formal-parameter type is defined by a type expression in procedure declaration, and actual-parameter type is defined in the call-time valuation,
5. all composites carried by actual parameters satisfy the yokes of corresponding formal parameters; notice that the yokes of actual parameters are not considered at all.

The computation of the types of formal parameters required a certain technical trick. Since these types are defined by type expressions, to compute them, the type expression denotations have to be applied to a state. Here is a problem since the function

```
dynamically-compatible.(fpd-v, fpd-r, apd-v, apd-r)
```

gets as an argument, not the whole state ((tye, pre), (vat, err)) but only two of its elements: tye and vat. To cope with this problem, a “temporary” state is created

```
((tye, [ ]), (vat, 'OK'))
```

where [] is an empty procedure-environment. As a matter of fact, this environment might be quite arbitrary since type expression denotations do not depend on it.

Notice at the end that each of the numbers n, m, k, and p may be zero, i.e., each of the corresponding parameter lists may be empty.

6.2.4 Passing actual parameters to a procedure

This function is activated by a procedure call and creates local initial valuation (Fig. 6.2-1). The only identifiers bound in this valuation are formal parameters and their initial values are the current values of the corresponding actual parameters.

```
pass-actual : FoPaDe x FoPaDe x AcPaDe x AcPaDe ↦
```

```
TypEnv x Valuation ↦ Valuation | Error
```

```
pass-actual.(fpd-v, fpd-r, apd-v, apd-r).(tye, vat) =
```

```
let
```

```
message = dynamically-compatible.(fpd-v, fpd-r, apd-v, apd-r).(tye, vat)
```

```
message ≠ 'OK' → message
```

```
let (for n, k ≥ 0)
```

```
((ide-fv.i, ted-fv.i) | i=1;k) = fpd-v
```

```
list of formal-value parameter denotation
```

```
((ide-fr.i, ted-fr.i) | i=1;n) = fpd-r
```

```
list of formal-reference parameter denotation
```

```
(ide-av.i | i=1;k) = apd-v
```

```
list of actual-value parameter denotation
```

⁷⁴ The assumption that actual parameters do not need to be initialized has been introduced mainly for reference variables. For value variables this assumption probably does not have much of a practical value but it is not harmful either.

$(ide-ar.i \mid i=1;n) = apd-r$	list of actual-reference parameter denotation
$val-v.i = vat.(ide-av.i) \quad \text{for } i=1;k$	list of values of actual-value parameter denotation
$val-r.i = vat.(ide-ar.i) \quad \text{for } i=1;n$	list of values of actual-reference parameter denotation
<i>creating list of initial local valuation</i>	
$vat-v = [ide-fv.i/val-v.i \mid i=1;k]$	initial local valuation of value-parameters
$vat-r = [ide-fr.i/val-r.i \mid i=1;n]$	initial local valuation of reference-parameters
$vat-il = vat-v \blacklozenge vat-r$	list of initial local valuation ⁷⁵
true $\rightarrow vat-il$	

The defined operator checks the compatibilities of parameters, and then creates local initial valuation to be later executed by procedure-body:

- formal value-parameters receive the values (or pseudovalues) of actual value-parameters; the definedness of these values and the compatibility of their types is checked by the function `dynamically-compatible`.
- formal reference-parameters receive the values (or pseudovalues) of actual reference-parameters; the definedness of these values and the compatibility of their types has been checked by the function `dynamically-compatible`.

Similarly, as in the former definitions, empty lists of parameters are allowed.

Notice that the described mechanism of creating initial local valuations does not offer a possibility of using global variables, i.e. variables visible both outside and inside procedure-body. The only communication channel of procedure call between its external and internal worlds are reference parameters that pass their values according to the following scheme:

```

fpd-v   := the values of apd-v
fpd-r   := the values of apd-r
[procedure-body execution]
apd-r   := value of fpd-r

```

6.2.5 Returning reference-parameters to a program

Whereas formal value-parameters play the role of local variables since they are visible only inside procedure body, formal reference-parameters play the role of global variables whose values are modified by procedure bodies.

`return-referential : FoPaDe x AcPaDe \mapsto TypEnv x Valuation x Valuation \mapsto Valuation | Error`

`return-referential.(fpd-r, apd-r).(tye, vat-tl, vat-ig) =`

```

let
  message = dynamically-compatible.((), fpd-r, (), apd-r).(tye, vat-ig)76
  message  $\neq$  'OK'  $\rightarrow$  message
let
  (ide-ar.i  $\mid$  i=1;n) = apd-r           list of actual reference-parameter denotations
  ((ide-fr.i, typ-fr.i)  $\mid$  i=1;k) = fpd-r   list of formal reference-parameter denotations
  vat-tl.(ide-fr.i) = ?  $\rightarrow$  'value-of-reference-parameter-undeclared' for i = 1;k
let

```

⁷⁵ Local valuation is created as an overwriting of local reference-valuation by local value-valuation. Since their sets of identifiers are disjoint, the resulting valuation is a simple expansion of one function by another. The overwriting operation \blacklozenge has been defined in Sec. 2.1.3..

⁷⁶ By “()” we denote empty tuples of parameters. Due to this trick we can apply a four-argument function to two lists of parameters. Notice that in principle we do not need to check here the adequacy of parameters since this is checked in passing actual parameters to procedure-body (Sec. 6.2.4). However, removing this check would make our definition incorrect.

$\begin{aligned} \text{val-fr.i} &= \text{vat-tl}.\text{(ide-fr.i) for } i=1;n \\ \text{vat-tg} &= \text{vat-ig}[\text{ide-ar.i/val-fr.i} \mid i=1;n] \\ \text{true} &\quad \rightarrow \text{vat-tg} \end{aligned}$	terminal values of formal ref-parameters terminal global valuation
--	---

After procedure-body has been executed, the values of formal reference-parameters in `vat-tl` are passed to the corresponding actual reference parameters in `vat-ig`. This operation transforms `vat-ig` into `vat-tg`.

As has already been mentioned, this communication mechanism might be described by two symbolic assignment-instructions. Before the execution of the body:

`fpd-r := the values of apd-r`

and after its execution

`apd-r := the value of fpd-r.`

If we read these assignments literally, they mean that actual-parameter values are copied to some memory-space allocated for procedure execution. If a parameter value is a small object like, e.g., a number, then, of course, such an implementation is quite acceptable, but if it is a large object, e.g., a database, such a solution would be somewhat absurd.

In the majority of programming languages, this problem is solved by passing references rather than values to actual-reference parameters. These references provide access (i.e., a memory address) to the values of formal parameters. From a functional point of view, such a solution is equivalent to ours, but if we would like to describe it formally, we had to introduce addresses in our model to bind identifiers with addresses and addresses with data (as in [53] by M. Gordon). The choice between the two alternatives depend upon the addressees of our model — are they language user or language implementors. According to the philosophy assumed in this book, we address our model to users rather than to implementors, and therefore we have not introduced addresses.

6.3 Imperative procedures with single recursion

In this section, we shall investigate a model described jointly by Andrzej Tarlecki and myself in [34]. As has been already announced, in this model, procedures are purely denotational objects. Consequently, we do not talk about procedure denotations but about procedures as such. On the syntactical side, we have only procedure declarations and procedure calls.

6.3.1 Constructor of procedures

Intuitively speaking, an imperative-procedure is created from a program, frequently called the *body of the procedure*, plus two lists of formal parameters — value parameters and referential parameters — which are used by the mechanisms of passing and returning parameters.

The three elements — two lists of parameters, and a body — will be referred to as the denotations of *imperative-procedure contents* or simply as *procedure contents*. As already announced in Sec.6.2.1, given a declaration-time environment and a denotation of a procedure content, we shall create a procedure, i.e., a function which, given a pair of actual parameters, returns a partial function from stores to stores. In our algebraic framework the following domain of procedure-content denotations will constitute a new carrier of the algebra of denotations:

$$\text{icd} : \text{lprConDen} = \text{FoPaDe} \times \text{FoPaDe} \times \text{ProDen}$$

The only constructor of that domain is

$$\text{create-imp-con} : \text{FoPaDe} \times \text{FoPaDe} \times \text{ProDen} \mapsto \text{lprConDen}$$

whose definition looks a little strange:

$$\text{create-imp-con}.\text{(fpd-v, fpd-r, prd)} = \text{(fpd-v, fpd-r, prd)}$$

At first glance, this constructor seems to be an identity function. However, on our algebraic ground, it takes three arguments from three carriers of our algebra and returns the result to a fourth carrier. Without that constructor, the reachable part of carrier lprConDen would be empty. Why we need that carrier, and as a consequence, its strange constructor, will become better seen when we come to multirecursion in Sec. 6.4.1.

Now, we can define an auxiliary constructor of procedures that, given two lists of parameter denotations, and a program denotation, returns a function, which given an environment, returns a procedure:

$\text{create-imp-proc} : \text{lprConDen} \mapsto \text{Env} \mapsto \text{ImpPro}$ i.e.

$\text{create-imp-proc} : (\text{FoPaDe} \times \text{FoPaDe} \times \text{ProDen}) \mapsto \text{Env} \mapsto \text{AcPaDe} \times \text{AcPaDe} \mapsto \text{Store} \rightarrow \text{Store}$

The environment that appears in this definition is a declaration-time environment (we shall denote it by env-dt) since our procedure will be created by a procedure declaration, hence in a declaration time⁷⁷. Our constructor is an auxiliary function that will not become a constructor of our algebra. Its definition explicitly shows how our procedure will elaborate on the initial global store sto-ig when given actual parameters (apd-v , apd-r):

```

create-imp-proc.(fpd-v, fpd-r, prd).env-dt.(apd-v, apd-r).sto-ig =
  is-error.sto-ig      → sto-ig
  let
    (vat-ig, 'OK')     = sto-ig
    (tye-dt, pre-dt)  = env-dt                                declaration-time environment
    par                = (fpd-v, fpd-r, apd-v, apd-r)
    vat-il             = pass-actual.par.(tye-dt, vat-ig)      initial local valuation
    vat-il : Error    → (vat-ig, vat-il)                       here vat-il is an error
  let
    sta-il = (env-dt, (vat-il, 'OK'))                          initial local state
    prd.sta-il = ?      → ?
  let  procedure-body execution
    (env-tl, (vat-tl, err)) = prd.sta-il                       the execution of proc. body creates terminal local state
    err ≠ 'OK'             → sto-ig ◀ err                      (*)
  let
    vat-tg = return-referential.(fpd-r, apd-r).(tye-dt, vat-tl, vat-ig)
    vat-tg : Error        → (vat-ig, vat-tg)                  (**)                               here vat-tg is an error
  true                   → (vat-tg, 'OK')

```

In the first step we check if the initial global store carries an error and if this is the case, then this store becomes the terminal global store.

In the opposite case, we create the *initial local valuation* vat-il , where the execution of the procedure body will start (see Fig. 6.2-1). It is created from *initial global valuation* vat-ig by passing values of actual parameters to formal parameters (the argument par). We recall (Sec. 6.2.4) that since the operator pass-actual checks the adequacy of parameter-lists, this part of procedure-execution may terminate with an error message. If that is the case, the terminal store consists of the initial local valuation vat-il and an error, which in our definition, is represented by the metavariable vat-il .

If an error message does not appear, we create an *initial local store* sto-il with the 'OK' message, and we create the *initial local environment* env-il by taking the declaration-time environment.

The declaration-time environment with an initial local store constitutes an *initial local state* sta-il .

In the next step, procedure-body (represented by a program-declaration prd) is executed in sta-il and — if this execution terminates — then its terminal state sta-tl becomes the *local terminal state*.

⁷⁷ Observe that if procedures were supposed to be executed in call-time environments, then they had to be functions of the type $P : \text{AcPaDe} \mapsto \text{Env} \rightarrow \text{Store}$, i.e. they could take themselves as arguments, and this leads to a not set-theoretical model.

If that state carries an error, then the terminal store consists of the initial global valuation and the current error.

Otherwise, we select the terminal local valuation vat-tl from that state and use it in returning the current values of formal reference-parameters to actual reference-parameters. If this results with an error than the terminal store consists of the initial global valuation and the current error.

In the opposite case, the terminal global-store (vat-tg , 'OK') consists of terminal global-valuation and 'OK' message.

It is worth noticing here that the execution of our procedure involves two non-trivial error-handling clauses (*) and (**). In both cases, an error message causes not only the interruption of program execution but also the recovery of the initial global-store. Of course, this is just one possible choice of an error-handling strategy in this place.

6.3.2 Procedure declaration

An imperative-procedure declaration assigns a procedure to an identifier in the current environment. This procedure is built from a procedure content and declaration time environment. The corresponding constructor is therefore the following:

$$\begin{aligned} \text{declare-imp-pro} &: \text{Identifier} \times \text{lprConDen} \mapsto \text{DecDen} && \text{i.e.} \\ \text{declare-imp-pro} &: \text{Identifier} \times \text{FoPaDe} \times \text{FoPaDe} \times \text{ProDen} \mapsto \text{State} \mapsto \text{State} \\ \text{declare-imp-pro.}(\text{ide}, \text{fpd-v}, \text{fpd-r}, \text{prd}).\text{sta} &= \\ \text{is-error.sta} & \rightarrow \text{sta} \\ \text{ide} : \text{declared.sta} & \rightarrow \text{sta} \leftarrow \text{'identifier-declared'} \\ \mathbf{let} & \\ & ((\text{tye}, \text{pre}), \text{sto}) = \text{sta} \\ \mathbf{ipr} = \text{create-imp-proc.}(\text{fpd-v}, \text{fpd-r}, \text{prd}, (\text{tye}, \text{pre}[\text{ide}/\text{ipr}])) & \text{fixed-point equation} \\ \mathbf{true} & \rightarrow ((\text{tye}, \text{pre}[\text{ide}/\text{ipr}]), \text{sto}) \end{aligned}$$

Observe that procedure ipr is defined here by means of a fixed-point equation. Such a construction allows for recursive calls of the declared procedure⁷⁸.

Note also that the constructor create-imp-proc receives the declaration-time environment, which means that this environment is “remembered” in ipr for later use when ipr is called. Theoretically, we could save in ipr only a procedure environment allowing a procedure call to get call-time type environment, thus setting

$$\text{ipr} : \text{AcPaDe} \times \text{AcPaDe} \times \text{TypEnv} \mapsto \text{Store} \rightarrow \text{Store}$$

However, from a practical point of view, it would be rather awkward to expect that programmers, when declaring a procedure, will anticipate types that will be declared later.

6.3.3 Recursion — how does it work?

The mechanism of recursion usually involves a stack mechanism that handles successive calls of a procedure. It can be seen from Fig. 6.2-1, where the initial global store must be saved to be referred to at the end of the call. In our case, however, we do not refer to a stack mechanism since recursion in **Lingua** is described in using the recursion in **MetaSoft**. Formally, the procedure ipr from Sec. 6.3.2 is defined by the fixed-point equation:

$$\mathbf{ipr} = \text{create-imp-proc.}(\text{fpd-v}, \text{fpd-r}, \text{prd}, (\text{tye}, \text{pre}[\text{ide}/\text{ipr}]))$$

⁷⁸ This construction has been suggested by Andrzej Tarlecki in our common paper [34]. In this way recursion in **Lingua** is expressed by the recursion in **MetaSoft** without involving a usual stack mechanism, and also without involving the problem of self-applicable functions.

By Kleene's theorem (Sec. 2.3) this means that ipr is the limit — in this case, a set-theoretic union — of an infinite chain of partial functions:

$$\begin{aligned}\text{ipr}.0 &= \text{create-imp-proc}(\text{fpd-v}, \text{fpd-r}, \text{prd}, (\text{tye}, \text{pre}[\text{ide}/[]])) \\ \text{ipr}.(k+1) &= \text{create-imp-proc}(\text{fpd-v}, \text{fpd-r}, \text{prd}, (\text{tye}, \text{pre}[\text{ide}/\text{ipr}.k]))\end{aligned}$$

where $[]$ denotes an empty state-to-state function. Here $\text{ipr}.[k+1]$ is a function which calls itself $k+1$ times. To see how it works in a concrete case take as an example a recursive procedure that computes n^m :

```
proc power (val n,m as nn_integer ref p as nn_integer),
  if m=0 then p:=1 else m:=m-1; call power(val n,m ref p); p:=p*n fi
endproc
```

where `nn_integer` is a user-defined type corresponding to non-negative integers. Let `abort` be an instruction whose denotation is $[]$. Now, the elements of our chain we may be symbolically (and not quite formally) described in the following way:

$$\begin{aligned}\text{power}.0.(n, m, p) &= \text{if } m=0 \text{ then } p:=1 \text{ else } m:=m-1 \bullet \text{abort} \bullet p:=p*n \text{ fi} \\ &= \text{if } m=0 \text{ then } p:=1 \text{ else } \text{abort} \text{ fi} \\ &= m=0 \rightarrow p=1=n^0 \\ &\quad m>0 \rightarrow ? \\ \text{power}.1.(n, m, p) &= \text{if } m=0 \text{ then } p:=1 \text{ else } m:=m-1 \bullet \text{power}.0.(n, m, p) \bullet p:=p*n \text{ fi} \\ &= \text{if } m=0 \text{ then } p:=1 \text{ else } m:=m-1; \text{if } m=0 \text{ then } p:=1 \text{ else } \text{abort} \text{ fi} \bullet p:=p*n \text{ fi} \\ &= m=0 \rightarrow p=1=n^0 \\ &\quad m=1 \rightarrow p=n=n^1 \\ &\quad m>1 \rightarrow ? \\ \text{power}.2.(n, m, p) &= m=0 \rightarrow p=1=n^0 \\ &\quad m=1 \rightarrow p=n=n^1 \\ &\quad m=2 \rightarrow p=n^2 \\ &\quad m>2 \rightarrow ?\end{aligned}$$

The limit of this chain is, of course, `power`.

In the end, assume that a recursive procedure, call it `Main`, includes a declaration of a local procedure, say `DoIt`. When `Main` is called, it is “given” its declaration-time environment `env-dt`, which it uses to create its local environment `env-lo`. In this environment, `DoIt` is declared. Somewhere during the execution of `Main`, its recursive call gets again `env-dt`, where it declares `DoIt`, thus creating the same `env-lo` as before. Notice that if a procedure call would receive a call-time environment, then an attempt to redeclare `DoIt` would cause an error message ‘`identifier-declared`’. Of course, the same conclusion applies to local types of a procedure.

6.3.4 Instruction of a procedure call

Calling an imperative procedure consists in executing three steps (cf. Sec. 6.2.1 and Fig. 6.2-1):

1. getting the called procedure from an environment,
2. applying to it actual parameters (tuples of identifiers) in order get a store-to-store transformation; this transformation is responsible for (see the definition of `create-imp-proc` in Sec. 6.3.1):
 - a. creating an initial local state and passing values of actual parameters to formal parameters in its store,
 - b. executing the body of the procedure (a program) in the initial local state thus getting a terminal local-state
 - c. returning values of formal reference parameters to actual reference parameters thus getting a terminal global-store,

3. combining the terminal global-store with initial global-environment thus getting a terminal global state.

Formally, we define a constructor

$\text{call-imp-proc} : \text{Identifier} \times \text{AcPaDe} \times \text{AcPaDe} \mapsto \text{InsDen}$ i.e.

$\text{call-imp-proc} : \text{Identifier} \times \text{AcPaDe} \times \text{AcPaDe} \mapsto \text{State} \rightarrow \text{State}$

The resulting denotation of procedure-call instruction is defined in the following way:

```

call-imp-proc.(ide, apd-v, apd-r).sta-ig =
  is-error.sta-ig           → sta-ig
  let
    (env-ig, sto-ig) = sta-ig           initial global state of the call
    (tye-ig, pre-ig) = env-ig
    pre-ig.ide = ?                     → sta-ig ◀ 'procedure-unknown'
    pre-ig.ide : FunPro79             → sta-ig ◀ 'procedure-not-imperative'
  let
    ipr = env-ig.ide                   the called imperative procedure
    ipr.(apd-v, apd-r).sto-ig = ?     → ?                               infinite computation
  let
    sto-tg = ipr.(apd-v, apd-r).sto-ig terminal global store of the call
    (vat-tg, err) = sto-tg
    is-error.sto-tg                   → sta-ig ◀ error.sto-tg
  true                                 → (env-ig, sto-tg)

```

If the call-time state does not carry an error message and the identifier `ide` is bound in the environment to an imperative procedure, then we apply this procedure to actual parameters getting in this way a partial function on stores:

$\text{pro}(\text{apd-r}, \text{apd-v}) : \text{Store} \rightarrow \text{Store}$

This function is applied to the initial global store `sto-ig`. Notice that since our procedure carries declaration-time environment, the corresponding procedure-body is executed in the state

$(\text{env-dt}, \text{vat-ig}, \text{'OK'})$

where

`env-dt` — declaration-time environment

`sto-ig` — initial global store

If the terminal store is not defined, then the result of the procedure call is not defined either. If the execution of the procedure body raises an error message, then this message is loaded into the initial global state of the call. In the opposite case, the terminal global-store of the call `sto-tg` becomes the component of the *terminal global-state of the call* $(\text{env-ig}, \text{sto-tg})$. The initial environment remains unchanged.

Notice that in this definition, we have neither parameter-adequacy check nor parameter passing since these operations are included in the procedure itself (Sec. 6.3.1).

6.4 Imperative procedures with mutual recursion

6.4.1 Multiprocedures and their components

The model of recursion described so far does not cover the case where procedure `P` calls procedure `Q`, and procedure `Q` calls procedure `P`. Of course, at the syntactic level, we cannot exclude such situations, but at the

⁷⁹ Mathematically the metapredicate of checking if a procedure is imperative or functional is rather evident since the domains `FunPro` and `ImpPro` are disjoint. Implementationally a possible solution may consist in labeling identifiers in procedure environment with e.g. `I` (for Imperative) and `F` (for Functional).

denotational (and implementational) level, if a procedural mechanism is defined as in Sec. 6.3 then mutual recursion will cause an error message ‘procedure-not-declared’. Indeed, if the declaration of P precedes the declaration of Q , then the call of Q in the body of P , and thus in the declaration-time environment of P would not find Q .

To solve this problem, P and Q must be defined jointly by one set of fixed-point equations of the form

$$P = F(P,Q)$$

$$Q = G(P,Q)$$

and, of course, analogously for a larger number of mutually recursive procedures.

In order to incorporate mutual recursion into our model, we have to modify the constructor of declarations in such a way that now instead of taking an identifier and a procedure content, it will take a tuple of such pairs. This leads to the following domain of components of multiprocedures:

$$\text{mcd} : \text{MprComDen} = (\text{Identifier} \times \text{IprConDen})^{c^+} \quad \text{multiprocedure-component denotations}$$

Note the wording: imperative-procedure content denotations are included in multiprocedure component denotations. Now we need two constructors to generate the elements of this domain. The first of them makes a one-element tuple of pairs:

$$\text{create-mcd} : \text{Identifier} \times \text{IprConDen} \mapsto \text{MprComDen}$$

$$\text{create-mcd}(\text{ide}, \text{icd}) = ((\text{ide}, \text{icd}))$$

The second constructor joins two tuples into one:

$$\text{join-mcd} : \text{MprComDen} \times \text{MprComDen} \mapsto \text{MprComDen}$$

$$\text{join-mcd}(\text{mcd-1}, \text{mcd-2}) = \text{mcd-1} \uplus \text{mcd-2}$$

Now, the definition of the new constructor of declaration denotations is the following:

$$\text{declare-imp-mul-pro} : \text{MprComDen} \mapsto \text{DecDen} \quad \text{i.e.}$$

$$\text{declare-imp-mul-pro} : \text{MprComDen} \mapsto \text{State} \mapsto \text{State}$$

$$\text{declare-imp-mul-pro.mcd.sta} =$$

$$\text{is-error.sta} \quad \rightarrow \text{sta}$$

let

$$((\text{ide-1}, \text{icd-1}), \dots, (\text{ide-n}, \text{icd-n})) = \text{imcd}$$

$$(\text{fpd-v-i}, \text{fpd-r-i}, \text{prd-i}) = \text{icd-i} \quad \text{for } i=1;n$$

$$(\text{env}, \text{sto}) = \text{sta}$$

$$(\text{tye}, \text{pre}) = \text{env}$$

$$\text{are-repetitions}(\text{ide-1}, \dots, \text{ide-n}) \rightarrow \text{sta} \leftarrow \text{‘procedure-names-are-repeated’}$$

$$\text{ide-i} : \text{declared.sta} \rightarrow \text{sta} \leftarrow \text{‘identifier-ide-i-declared’} \quad \text{for } i=1;n$$

let

$$\text{ipr-1} = \text{create-imp-proc}(\text{icd-1}, (\text{tye}, \text{pre}[\text{ide-1}/\text{ipr-1}, \dots, \text{ide-n}/\text{ipr-n}]))$$

...

$$\text{ipr-n} = \text{create-imp-proc}(\text{icd-n}, (\text{tye}, \text{pre}[\text{ide-1}/\text{ipr-1}, \dots, \text{ide-n}/\text{ipr-n}]))$$

$$\text{true} \rightarrow ((\text{tye}, \text{pre}[\text{ide-1}/\text{ipr-1}, \dots, \text{ide-n}/\text{ipr-n}]), \text{sto})$$

Notice that if $n=1$, then our definition coincides with the case of a single recursion.

6.5 Functional procedures

The difference between imperative- and functional procedures is that the result of an imperative-procedure call is a state, whereas, for functional procedures, it is a value. Imperative procedures may be regarded, therefore, as instructions with parameters and functional procedures — as expressions with parameters. Functional-procedure calls belong in **Lingua-2** to the domain of expressions.

6.5.1 The structure of a functional-procedure declaration

Even though functional procedures correspond to expressions, the bodies of their declarations will consist of a program — may be trivial, i.e., consisting of a trivial declaration and a trivial instruction — and an expression. A program transforms an input state and passes the resulting state to the expression, which computes a value. This value is exported by the call. Below an example of a declaration of a procedure which computes the absolute value of a power n^m :

```

fun absolute-power(n, m integer)
  let p be integer;
  p := 1 ;
  while m > 0 do p := p*n ; m:=m-1 od
  return if p ≤ 0 then -p else p fi as integer
endfun

```

In particular, the program that precedes **return** may be trivial, i.e., of the form **skip-d**; **skip-i**, and the expression that follows **return** may be reduced to a single variable⁸⁰. The expression following **return** will be referred to as an *exporting expression*.

In some languages (e.g., in Pascal [56]), global variables are admitted in functional procedures, which means that they can change states. It is frequently called a *side-effect*. In **Lingua-2**, I deliberately give up this option since, in my opinion, each covert action of programs may contribute to programming errors. As a matter of fact, the authors of Pascal — although they allow side-effects — at the same time, they strongly discouraged programmers from using them (see [56] page 79). One may wonder why they have not eliminated that option from their language?

In this place, we can return to the question of why actual parameters were assumed to be identifiers rather than arbitrary expressions (cf. Sec. 6.1.3)? Notice that in the opposite case, actual parameters could be functional-procedure calls, which would lead to a new model of recursion and would certainly complicate construction rules for procedures (see Sec. 7).

A possible technical solution to that problem might be an assumption that actual parameters may be expressions but could not include procedure calls. Mathematically this is possible, but on the algebraic level, it leads to two sorts of expressions (with and without procedure calls) and again complicates proof rules.

6.5.2 Functional procedures denotationally

In the case of functional procedures, similarly as for imperative procedures, we deal with three categories of mathematical beings:

- *functional procedures* — store-to-value functions
- *denotations of declarations of functional procedures* — state-to-state functions,
- *denotations of functional-procedure calls* — state-to-value functions

Since the calls of functional procedures will belong to the domain of expression denotations, and their declarations to the domain of declarations, we introduce only one new domain:

$$\text{fpr} : \text{FunPro} = \text{AcPaDe} \mapsto \text{Store} \rightarrow \text{ValueE} \quad \text{functional procedures}$$

Similarly to imperative procedures, and for the same reason this domain will not become a carrier of our algebra of denotations. Consequently functional procedures will not have syntactic representations.

At the same time, the domain of expression denotations is enriched by the denotations of functional-procedure calls. It is a substantial change in our language since now expression denotations need access to

⁸⁰ This universal form of a functional-procedure declaration was suggested to me by Andrzej Tarlecki.

procedure-environments from which they will take functional procedures. Notice that although expressions were formally defined in **Linguga-1** on states, they effectively were “reaching” only stores.

Contrary to imperative procedures that take value parameters and referential parameters, functional procedures have only value parameters. It is another engineering decision, which means that functional procedures have no side-effects.

6.5.3 Constructors of functional-procedure-denotation contents

The domain of *the denotations of functional-procedure contents* is defined in the following way:

$$\text{fcd} : \text{FprConDen} = \text{FoPaDe} \times \text{ProDen} \times \text{DatExpDen} \times \text{TypExpDen}$$

Compared with imperative procedures, functional-procedure contents include only one list of formal parameters (value parameters) but, on the other hand, include the denotation of an exporting expression and of a type expression. The latter defines the expected type of the result of the call.

Analogously to the case of imperative procedures, also with this domain, we assign only one constructor:

$$\text{create-fup-con} : \text{Identifier} \times \text{FoPaDe} \times \text{ProDen} \times \text{DatExpDen} \times \text{TypExpDen} \mapsto \text{FprConDen}$$

$$\text{create-fup-con}(\text{ide}, \text{fpd}, \text{prd}, \text{ded}, \text{ted}) = (\text{ide}, \text{fpd}, \text{prd}, \text{ded}, \text{ted})$$

See comment to the corresponding constructor in Sec. 6.3.1.

6.5.4 The constructor of a functional procedure

A functional procedure may be regarded as a sequential composition of three components:

1. a function that passes actual parameters to the call-time state of the procedure
2. an instruction,
3. an exporting expression.

We start with the definition of an auxiliary exportation function:

$$\text{export} : \text{DatExpDen} \times \text{TypExpDen} \mapsto \text{State} \mapsto \text{ValueE}$$

$$\text{export}(\text{ded}, \text{ted}).\text{sta} =$$

$$\text{is-error.sta} \quad \rightarrow \text{error.sta}$$

let

$$\text{val} = \text{ded.sta}$$

$$\text{typ} = \text{ted.sta}$$

exported value
expected type of value

$$\text{typ} : \text{Error} \quad \rightarrow \text{typ}$$

$$\text{val} : \text{Error} \quad \rightarrow \text{val}$$

let

$$(\text{dat-v}, \text{bod-v}, \text{yok-v}) = \text{val}$$

$$(\text{bod-t}, \text{yok-t}) = \text{typ}$$

$$\text{bod-t} \neq \text{bod-v} \quad \rightarrow \text{'bodies-inconsistent'}$$

$$\text{yok-t}(\text{dat-v}, \text{bod-v}) \neq (\text{tt}, \text{'boolean'}) \quad \rightarrow \text{'yoke-not-satisfied'}^{81}$$

$$\text{true} \quad \rightarrow \text{val}$$

This operator returns a value `val` computed by the denotation `ded` of the exporting expression under the condition that:

1. the body of `val` equals the body of the type `typ` computed by `ted`,
2. the composite of `val` satisfies the yoke of `typ`.

⁸¹ Notice that `yok.(dat-v, bod-v) ≠ (tt, 'boolean')` means that either `yok.(dat-v, bod-v) = (ff, 'boolean')` or `yok.(dat-v, bod-v) : Error`.

Now we are ready to define the constructor of functional procedures:

$$\begin{aligned} \text{create-fun-pro} &: \text{Identifier} \times \text{FprConDen} \times \text{Env} \mapsto \text{FunPro} \quad \text{i.e.} \\ \text{create-fun-pro} &: \text{Identifier} \times (\text{FoPaDe} \times \text{ProDen} \times \text{DatExpDen} \times \text{TypExpDen}) \times \text{Env} \mapsto \\ & \quad (\text{AcPaDe} \mapsto \text{Store} \rightarrow \text{ValueE}) \end{aligned}$$

Similarly to the case of imperative procedures, this constructor is an auxiliary function, i.e., is not a constructor in the algebra of denotations.

$$\begin{aligned} \text{create-fun-pro.}(\text{ide}, (\text{fop-v}, \text{prd}, \text{ded}, \text{ted}), \text{env-dt}).\text{apd-v.sto-ig} = & \\ \text{is-error.sto-ig} \quad \rightarrow \text{error.sto-ig} & \quad \text{initial global store} \\ \mathbf{let} & \\ \quad (\text{vat-ig}, \text{'OK'}) = \text{sto-ig} & \\ \quad (\text{tye-dt}, \text{pre}) = \text{env-dt} & \quad \text{declaration time environment} \\ \quad \text{vat-il} = \text{pass-actual.}(\text{fop-v}, (), \text{apd-v}, ()) . (\text{tye-dt}, \text{vat-ig}) & \quad \text{initial local valuation} \\ \text{val-il} : \text{Error} \quad \rightarrow \text{vat-il} & \\ \mathbf{let} & \\ \quad \text{sta-il} = (\text{env-dt}, (\text{vat-il}, \text{'OK'})) & \\ \text{prd.sto-il} = ? \quad \rightarrow ? & \\ \mathbf{let} & \\ \quad \text{sta-tl} = \text{prd.sto-il} & \quad \text{terminal local store} \\ \text{is-error.sta-tl} \quad \rightarrow \text{error.sta-tl} & \\ \mathbf{true} \quad \rightarrow \text{export.}(\text{ded}, \text{ted}).\text{sta-tl} & \end{aligned}$$

Using parameter-passing operator, we create a local initial state that is passed to the program included in the procedure body. The exporting expression is evaluated in the output state of that program, and the resulting value is the result of the procedure call. The body of this value must be of the type indicated by the type expression, which is checked by the exporting operator. The empty tuples $()$ that appear among the arguments of this operator correspond to formal and actual referential parameters respectively.

6.5.5 The expressions of functional-procedure calls

A functional procedure is a function which given actual value-parameters, returns a data-expression denotation. A call of such a procedure is performed in four steps:

1. getting the procedure from an environment,
2. computing the values of its actual parameters,
3. applying the procedure to parameters to get a data-expression denotation,
4. applying this denotation to the actual state, which — if the computation terminates — returns a value or an error message.

The corresponding constructor is of the type:

$$\begin{aligned} \text{call-fun-pro} &: \text{Identifier} \times \text{AcPaDe} \mapsto \text{DatExpDen} \quad \text{i.e.} \\ \text{call-fun-pro} &: \text{Identifier} \times \text{AcPaDe} \mapsto \text{State} \rightarrow \text{ValueE} \end{aligned}$$

The expression denotation that is created in this way is defined as follows:

$$\begin{aligned} \text{call-fun-pro.}(\text{ide}, \text{apd}).\text{sta} = & \\ \text{is-error.sta} \quad \rightarrow \text{error.sta} & \\ \mathbf{let} & \\ \quad ((\text{tye}, \text{pre}), \text{sto}) = \text{sta} & \\ \text{pre.ide} = ? \quad \rightarrow \text{'procedure-not-declared'} & \\ \text{pre.ide} : \text{ImpPro} \quad \rightarrow \text{'procedure-not-functional'} & \\ \mathbf{let} & \end{aligned}$$

<code>fpr = pre.ide</code>			functional procedure
<code>fpr.apd.sto = ?</code>	$\rightarrow ?$		
<code>true</code>	\rightarrow <code>fpr.apd.sto</code>		

If the initial state does not carry an error, and a functional procedure has been declared under the name `ide` in the environment, then this procedure is applied to the current actual-parameters and the current store. If the application terminates, then its result is the result of the call. It may be a value or an error.

6.5.6 The declaration of a functional procedure

In this case, the corresponding constructor is a function of the type:

`declare-fun-pro` : `FprConDen` \mapsto `DecDen`

hence

`declare-fun-pro` : `Identifier` x `FoPaDe` x `ProDen` x `DatExpDen` x `TypExpDen` \mapsto `State` \mapsto `State`

The definition of this constructor is analogous as in the imperative case:

<code>declare-fun-pro.(ide, fpd, prd, ded, ted).sta =</code>			
<code>is-error.sta</code>	\rightarrow <code>error.sta</code>		
<code>ide : declared.sta</code>	\rightarrow <code>sta</code> \blacktriangleleft 'variable-declared'		
let			
<code>((tye, pre), sto) = sta</code>			
<code>fpr = create-fun-pro.(fpd, prd, ded, ted, (tye-dt, pre-dt[ide/fpr])</code>			fixed-point equation
true	\rightarrow <code>((tye, pre[ide/fpr]), sto)</code>		

The fixed-point character of the definition of `fpr` allows for recursive calls similarly as in the case of imperative procedures.

6.5.7 Typological procedures ???

This section will be written along the lines sketched in Sec. 1.5.8. Or, maybe, not?

6.6 Procedures as parameters of procedures

As we already know, the attempt to define procedures that can take other procedures as parameters leads in the general case to a non-denotational domain recursion of the type:

`Procedure` = `Parameter` \mapsto `Store` \rightarrow `Store`

`Parameter` = `Value` | `Procedure`

A mechanism of that sort had been implemented in Algol 60, but a mathematical description of its construction has led to non-denotational models or at least non-denotational on the ground of classical set-theory (cf. [75]).

However, the fact that in "usual" set theory, a function cannot take itself as an argument does not mean that it cannot take other functions in this way. To construct a denotation model of procedures with procedural parameters, we have to construct a hierarchy of procedural domains (see [33] for more details):

`Procedure.0` = `Parameter.0` \mapsto `Store` \rightarrow `Store`

`Parameter.0` = `AcPaDe` x `AcPaDe`

For $n \geq 0$:

`Parameter.(n+1)` = `Parameter.0` | ... | `Parameter.n`

`Procedure.n` = `Parameter.n` \mapsto `Store` \rightarrow `Store`

In this model, a procedure may take as procedural arguments only procedures of a lower level than its own. To keep the description of **Lingua** of a reasonable size, this model shall not be developed further in the book.

6.7 Programs

In **Lingua-1**, programs consist of a declaration followed by an instruction where, of course, both of them may be structured. In **Lingua-2**, we keep this principle unchanged, but declarations may now include procedure declarations of all types. It is again a technical assumption that will make proof-rules simpler.

6.8 Syntax and semantics

6.8.1 The signature of the algebra of denotations

As we remember from Sec. 3.2 and Sec. 5.2.2, concrete syntax of a language is derived from abstract syntax, which in turn is derived from the signature of the algebra of denotations. Let us start, therefore, from that signature. To a large extent, it is implicit in the definitions of denotation-constructors; however, we have to remember that not all constructors defined in Sec. 6 are constructors of our algebra of denotations. Some of them were introduced only to define other constructors. On this list, we have functions that describe communication between imperative procedures and their hosting programs (Sec. 6.2) and all three constructors of procedures (Sec. 6.3, 6.4, and 6.5).

An analogous observation concerns the domains themselves.

6.8.1.1 The carriers of the algebra of denotations of Lingua-2

The list below covers all **Lingua-2** carriers, including **Lingua-1** carriers. New carriers are labeled by NEW.

ide	: Identifier		identifiers
ded	: DatExpDen		data-expression denotations including the calls of functions
tra	: TraExpDen		transfer-expression denotations
yok	: YokExpDen		yoke-expression denotations
typ	: TypExpDen		type-expression denotations
fpd	: FoPaDe	NEW	list of formal parameter denotations
apd	: AcPaDe	NEW	list of actual parameter denotations
icd	: lprConDen	NEW	imperative-procedure-content denotations
mcd	: MprComDen	NEW	multiprocedure-component denotations
fcd	: FprConDen	NEW	functional-procedure content denotations
din	: InsDen		instruction denotations including the calls of procedures
ded	: DecDen		declaration denotations
prd	: ProDen		program denotations

6.8.1.2 New constructors of the algebra of denotations

PARAMETERS

Actual parameters

create-empty-act-par-den :	\mapsto AcPaDe
create-single-act-par-den : Identifier	\mapsto AcPaDe
add-act-par-den : AcPaDe x Identifier	\mapsto AcPaDe

Formal parameters

create-empty-for-par-den :	\mapsto FoPaDe
create-single-act-par-den : Identifier x TypExpDen	\mapsto AcPaDe
add-for-par-den : FoPaDe x Identifier x TypExpDen	\mapsto FoPaDe

Definitions in Sec. 6.2.2

IMPERATIVE PROCEDURES**Denotations of imperative-procedure contents**

create-imp-con : FoPaDe x FoPaDe x ProDen \mapsto IprConDen

Definition in Sec. 6.3.1

Procedure declarations

declare-imp-pro : Identifier x IprConDen \mapsto DecDen

Definition in Sec. 6.3.2

Procedure calls

call-imp-proc : Identifier x AcPaDe x AcPaDe \mapsto InsDen

Definition in Sec. 6.3.4

MULTIPROCEDURES**Multiprocedure contents**

create-mcd : IprConDen \mapsto MprComDen

join-mcd : MprComDen x MprComDen \mapsto MprComDen

Definitions in Sec. 6.4.1

Multiprocedure declarations

declare-imp-mul-pro : ImprConDen \mapsto DecDen

Definition in Sec. 6.4.1

Multiprocedure calls

The calls of multiprocedures are just procedure calls, and therefore neither a new domain nor a new constructor are necessary.

FUNCTIONAL PROCEDURES**Functional-procedure contents**

create-fup-con : FoPaDe x ProDen x DatExpDen x TypExpDen \mapsto FprConDen

Definition in Sec. 6.5.3

Function calls

call-fun-pro : Identifier x AcPaDe \mapsto DatExpDen

Definition in Sec. 6.5.5

Function declarations

declare-fun-pro : FprConDen \mapsto DecDen

Definition in Sec. 6.5.6.

6.8.2 Concrete syntax

In the process of concrete-syntax creation, we skip the stage of abstract syntax since it has an algorithmic character and has already been described in detail for **Lingua-A** (Sec. 5.2.1). In doing so, we act similar to a mathematician who constructs proofs of theorems in an intuitive way rather than formally derives sequences of formulas by deduction-rules. However, in both cases, we have to make sure that there exists a theoretical fundament that guarantees the mathematical correctness of our constructions.

Contrary to Sec. 6.8.1.2, where only new constructors have been listed, here we show all syntactic categories of **Lingua-2**, although without explicitly repeating these clauses which have been taken from **Lingua-A** and **Lingua-1**.

ide : Identifier = (as in **Lingua-A**)

tre : TraExp = (as in **Lingua-A**)

yoe : YokExp = (as in **Lingua-A**)

tex : TypExp = (as in **Lingua-A**)

dae : DatExp = (as in **Lingua-A**) | Identifier (ActPar)

ins : Instruction = (as in **Lingua-1**) | **call** Identifier (**val** ActPar **ref** ActPar)

acp : ActPar = **empty-ap** | Identifier | (Identifier, ActPar)

fop : ForPar = **empty-fp** | Identifier **as** TypExp **sa** | (Identifier **as** TypExp **sa** , ForPar)

ico : lprCon = ((**val** ForPar **ref** ForPar) Program)

mprCon : MprCon = (Identifier, lprCon) | (MprCon , MprCon)

fprCon : FprCon = Identifier (ForPar) Pro **return** DatExp **as** TypExp)

dec : Declaration =

(variable declaration and type declarations as in Lingua-1)	
proc Identifier lprCon endproc	
mulproc MprCon endmulproc	
fun Identifier FprCon endfun	

prg : Program = (as in **Lingua-1**)

Now, one comment is necessary about our grammar. It concerns actual parameters where `empty-ap` is a keyword whose denotation is an empty list of actual parameters. Notice that our grammar allows the generation of “awkward” lists of parameters that start with `empty-ap`, e.g.

```
empty-ap, x, y, z
```

It is the price that we pay for the simplicity of our grammar. If we wanted to avoid such situations, we had to use a grammar with two equations:

```
ActPar          = empty-ap | NotEmptyActPar
NotEmptyActPar  = Identifier | NotEmptyActPar , Identifier
```

First clause permits for empty lists of actual parameters, the second — permits skipping `empty-ap`, if we want to declare an empty list of parameters. Such a grammar leads to a syntactic algebra which is not similar to our algebra of denotations. Of course, we could change the latter to make it similar, but this would mean that at the level of denotations, we think about syntax, and this is what we actually want to avoid. We accept, therefore, our compromise grammar. Notice in this place, that our grammar allows for the generation of the list as we wish to have, e.g.,

```
x, y, z
```

and on the other hand, each “awkward” list has a sound denotational meaning.

There is one more issue with actual parameters, which has to be explained. It concerns the fact that a procedure call, where a list of parameters is empty, must explicitly indicate this fact. For instance, we cannot write

```
call inventory (ref x, y)
```

but instead, we have to write

```
call inventory (val empty-ap ref x, y)
```

This sort of discipline may be seen tedious, but it protects programmers against oversight errors. In **Lingua**, every action must be explicit. It is why we have no global variables in imperative procedures and no side-effects in functional procedures. Analogous remarks apply, of course, to formal parameters.

6.8.3 Colloquial syntax

In **Lingua-2**, we allow all the colloquialisms of **Lingua-1**, and we add one concerning formal parameters in procedure declarations of both types. We allow grouping parameters into lists of variables associated with a common type as in the following example:

```
proc name (val w, z as number ref x, y as number a, b, c as employee)
```

where `employee` is a user-defined type.

6.8.4 Semantics

Since **Lingua-2** semantically coincides with **Lingua-1**, wherever both languages coincide syntactically, in this section, we consider these constructions that are not in **Lingua-1**. We write `ide` instead of `Sid.[ide]` since the semantics of identifiers is an identity function. We shall use the algebraic style of semantics (see Sec. 4.7).

Actual parameters

$S_{ap} : LisActPar \mapsto AcPaDe$ i.e.

$S_{ap} : LisActPar \mapsto Identifierc^*$

$S_{ap}.[empty-ap] = ()$

$S_{ap}.[ide] = (ide)$

$S_{ap}.[apd-1 , apd-2] = S_{ap}.[apd-1] \odot S_{ap}.[apd-2]$

Formal parameters

$Sfpa : LisForPar \mapsto FoPaDe$ i.e.
 $Sfpa : LisForPar \mapsto (Identifier \times TypExpDen)^c$
 $Sfpa.[empty-fp] = ()$
 $Sfpa.[ide \textbf{ as } tex] = ((ide, Ste.[tex]))$
 $Sfpa.[lap-1 , lap-2] = Sfpa.[lap-1] \odot Sfpa.[lap-2]$

Data expressions: functional-procedure calls

$Sde : DatExp \mapsto DatExpDen$
 $Sde.[ide (lap)] = call-fun-pro.(ide, Sapa.[lap])$

Instructions: imperative-procedure calls

$Sin : Instruction \mapsto InsDen$
 $Sin.[call \textit{ ide } (\textbf{ref } apd-r \textbf{ val } apd-v)] =$
 $call-imp-pro.(ide, Sapa.[apd-r], Sapa.[apd-v])$

Imperative procedure contents

$Sipc : lprCon \mapsto lprConDen$
 $Sipc.[ide (\textbf{val } acp-1 \textbf{ ref } acp-2) \textit{ prg}] = (ide, Sfpa.[acp-1], Sfpa.[acp-2], Spr.[prg])$

The semantics of procedure content yields a tuple of denotations which constitute a procedure-content denotation.

Imperative-procedure declarations

$Sipd : ImpDec \mapsto lprDecDen$
 $Sipd.[ico] = declare-imp-pro.(Sipc.[ico])$

The semantics of procedure declarations first turn a procedure declaration into the denotation of a procedure-content and then applies to it the constructor of declaration denotations.

Multiprocedure contents

$Simpc : ImprCon \mapsto ImprConDen$
 $Simpc.[ico] = Sipc.[ico]$
 $Simpc.[imc-1, imc-2] = Smpc.[imc-1] \odot Smpc.[imc-2]$

Multiprocedure declarations

$Smpd : MultiProcDec \mapsto MprDecDen$
 $Smpd.[\textbf{begin multiproc } imc \textbf{ end multiproc}] =$
 $declare-mul-pro.(Smpc.[imc])$

In a more intuitive form (although not quite formally), e.g. if we address our definition to programmers rather than to compiler designers, we can write this clause in the following unfolded form

$Smpd.[\textbf{begin multiproc}$
 $($
 $ide.1(\textbf{val } fpd-v.1 \textbf{ ref } fpd-r.1) \textit{ pro.1} ,$
 \dots
 $ide.n(\textbf{val } fpd-v.n \textbf{ ref } fpd-r.1) \textit{ pro.n}$
 $)$

```
end multiproc ] =  
  declare-mul-pro.((ide-i, Sfpa.[fpd-v.i], Sfpa.[fpd-r.i], Spr.[pro.i]) | i=1;n)
```

Functional-procedure contents

```
Sfpc : FprCon  $\mapsto$  FprConDen  
Sfpc.[ ide (acp) prg return dae as tex ] =  
  (ide, Sfpa.[acp], Spr.[prg], Sde.[dae], Ste.[tex])
```

Functional-procedure declarations

```
Sfpc : FprDec  $\mapsto$  FprDecDen  
Sfpc.[fco] =  
  declare-fun-pro.(Sfpc.[fco]) =
```

7 SEMANTIC CORRECTNESS OF PROGRAMS

7.1 Historical remarks

Semantic correctness of programs, historically called *program correctness*, was a subject of investigations from the very beginning of the computer's era. The earliest paper in this field—today practically forgotten—has been published by a British mathematician Alan Turing⁸² in 1949 [78]. Nearly twenty years later, in the year 1967, the same ideas were investigated again by an American scientist Richard Floyd [47]. In 1978 Association of Computing Machinery established annual Turing Price “for outstanding achievements in informatics”. One of the first winners of that price in 1978 was... Richard Floyd.

As far as I know, it has never been established if Floyd knew Turing's work. In the 1980-ties I wrote on that subject to Cambridge University. The only answer that I received was a very firm advise that I should not try to build “yet another myth about Turing”.

The work of Floyd introduced a fundamental concept of *an invariant of a program* and was dedicated to programs represented by graphical forms called *flow diagrams*. Two years later, a British scientist C.A.R Hoare (also a Turing Price winner), published a paper concerning Floyd's ideas applied to *structured programs*, i.e., programs constructed with the help of sequential composition, *if-then-else* branching, and while loops. This approach called later *Hoare's Logic* had given rise to a large field of research. See also Edsger W. Dijkstra [44], and two extensive monographs by K. Apt [4] and by K. Apt and H.R. Olderog [5].

Research devoted to program correctness was also developed in Poland. The first paper on that subject (although in an approach different to Hoare's) was published in 1971 by Antoni Mazurkiewicz [61]. A year later, during the first conference in a series of conferences on *Mathematical Foundations of Computer Science*⁸³, Antoni and I have presented a joint paper [32] on a similar subject based on an algebra of binary relations and covering recursive programs and nondeterminism. Nearly ten years later, I have published a paper [23] with a complete model of program-correctness rules for programs corresponding to arbitrary flow-diagrams without procedures and recursion. Contrary to many papers in this field and in particular to papers developing Hoare's logic, I assumed that program failure might correspond not only to infinite computations but also to program abortion, which in this book is modelled by issuing a state which carries an error. This also forces the use of three-valued predicate calculus.

In this place, it would also be appropriate to mention two fields of research developed at Warsaw University. The first one was a formalized approach to program correctness based on a so-called algorithmic logic [10] where programs appear in logical formulas. The second [61] was much more engineering-oriented and splits into three areas: *grammatical deduction*, *performance-analysis of computing systems*, and *formal specification of software requirements*. An interesting application of the second approach is described in a paper

⁸² Alan Turing (1912-1954) was one of the creators of the theory of computability. His model known today as *Turing machine* is regarded as one of fundamental concepts of this theory. Due to his work "On Computable Numbers, With an Application to the Entscheidungsproblem" Turing was considered as one of the greatest mathematicians of the world. Unfortunately he was also subject to a homophobic discrimination. When in 1952 police has learned about his homosexuality he was forced to choose between prison or hormonal therapy. He has chosen the latter but committed a suicide.

⁸³ This conference was organized in 1972 by a group of young researchers from the Institute of Computer Science of the Polish Academy of Sciences and the Department of Mathematics and Mechanics of Warsaw's University. Next year a similar conference was organized in Czechoslovakia which gave rise to a long series of MFCS conferences. Since 1974 proceedings of these conferences have been published by Springer Verlag in the series Lecture Notes in Computer Science.

by D.L. Parnas, G.J.K. Asmis, J. Madey [70] devoted to a safety assessment of software for a shutdown systems of the Darlington Nuclear Power Generating Station (Canada).

The idea of proving programs correct — despite its undoubted scientific importance — was never widely applied in software engineering. In my personal opinion, this situation was due to the implicit assumption that programs come first and their proofs are built later. This order is natural in mathematics, where a theorem precedes its proof but is somewhat unusual in engineering. Imagine an engineer who first constructs a bridge and only later performs all the necessary calculations. Such a bridge would probably collapse before its construction was completed, and this is what happens with programs. The first version of code usually does not work as expected. Hence a large part of the program development budget is spent on testing and “debugging”, i.e., on removing bugs introduced at the stage of writing the code. It is a well-known fact that all bugs can never be identified and removed by testing. Hence the remaining bugs are removed on the user’s expense under the name of “maintenance”.

In this book (Sec. 7 and Sec. 8), I am trying to develop ideas sketched earlier in my papers [21] and [22] where instead of proving programs correct, a programmer develops correct programs using rules that guarantee the correctness. In such a framework, a software engineer can work as an engineer who builds bridges, cars, or airplanes, where products are built from correct components by using rules that guarantee the correctness of the result.

Since the rules for the development of correct programs are derived from the rules of proving programs correct, we shall start from the latter. The discussion is carried on the ground of an algebra of binary relations since this leads to a relatively simple model where many technicalities of programming languages can be hidden. Of course, to apply these rules in a practical environment, they have to be expressed on the ground of a mathematical model of a programming language. A language **Lingua-2V** (V for “validation”) with such a model is constructed in Sec. 8.

7.2 A relational model of nondeterministic programs

Each program and each of its imperative components defines a specific *input-output relation* (I-O relation) that describes the transformations of input states into output states. Of course, in a deterministic case, this relation is a function. As we are going to see, in the general relational model, we can express quite a few ideas associated with program correctness. Although programs in **Lingua**, as described in this book, are deterministic, the discussion of a more general case seems worthwhile, especially that it does not complicate the model.

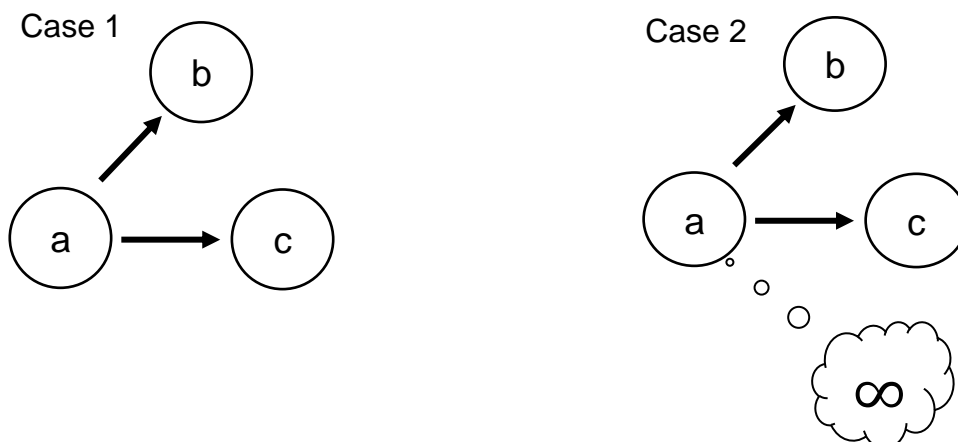


Fig. 7.2-1 Two nondeterministic cases

Let S be an arbitrary, possibly infinite, set of elements called *states*. In **Lingua**, states are mappings that assign values, types, and procedures to identifiers, but in the abstract case, we do not need to assume anything about them. In the relational model programs are represented by binary relations over S , i.e., elements of the set:

$$\text{Rel}(S, S) = \{R \mid R \subseteq S \times S\}$$

The fact that

$$a R b \quad \text{for } a, b : S$$

means that there exists an execution of program R that starts in a and terminates in b . In a non-deterministic case, there may be more than one execution that starts in a . Some may terminate with another state, say c (Case 1 of Fig. 7.2-1), some others may be infinite (Case 2 of Fig. 7.2-1). In our model, the difference between Case 1 and Case 2 cannot be expressed. In both cases, we can only say that

$$a R b \text{ and } a R c.$$

Note that due to the use of states which may carry errors, abortion of a computation from a to b means that b carries an error. This also means that if R is a function then the non-existence of a state b such that $a R b$ means that a starts an infinite execution.

If we want to deal with infinite executions explicitly, we need a different concept of program denotations. Two such models were analysed in [19]. One uses so-called δ -relations, where $a R \delta$ means that there exists an infinite computation that starts in a ⁸⁴. In that model, however, we cannot describe the fact that there are two or more different infinite computations that start from the same state. Such issues can be handled on the ground of yet another model, where program denotations are sets of finite or infinite sequences of states called *bundles of computations*. Both approaches can be used in building denotational models of programming languages.

7.3 Iterative programs

In “prehistoric” informatics of the years 1940/1950, programs were written as lists of labeled instructions executed sequentially one after another unless a *jump instruction goto* interrupted that flow. With jump instruction and conditional instruction *if-then*, one could build an arbitrary graph of elementary instructions called a *flow-diagram*. Early papers on program correctness were devoted to such programs later called *iterative programs*.

A general relational model of an iterative program is the following fixed-point set of so called left-linear equations⁸⁵:

$$\begin{aligned} X_1 &= R_{11} X_1 \mid \dots \mid R_{1n} X_n \mid E_{1n} \\ &\dots \\ X_n &= R_{n1} X_1 \mid \dots \mid R_{nn} X_n \mid E_{nn} \end{aligned} \tag{7.2-1}$$

that corresponds to a graph whose nodes are numbers $1, \dots, n$, each relation R_{ij} labels the edge between i and j , and each E_{in} (exit relation) is a “dangling edge” that start on i , but does not point to any other node. The code of such a program may be written as an arbitrarily ordered⁸⁶ sequences of labelled instructions of the form:

i : **do** R_{ij} **goto** j **and**
 i : **do** E_{in} .

If there is no instruction between i and j , then the relation R_{ij} is assumed to be empty which means that there are no runs between i and j . Since the atomic instructions R_{ij} and E_{in} are not necessarily functions, such a

⁸⁴ In this model each δ -relation is a union of three set of pairs $R \subseteq S \times S$, $D \subseteq S \times \{\delta\}$ and $\{(\delta, \delta)\}$, where S and D may be empty.

⁸⁵ They are called so because coefficients of variables X_i stand on their left-hand side. A symmetric model of right-linear equations of the general form $X = XR \mid Q$ has been analysed in [23].

⁸⁶ The execution of such a program does not depend on the order of its instructions since every instruction points to the instruction which should be executed as the next one.

program may have a non-deterministic character. For (7.2-1) to be deterministic, two conditions must be satisfied:

- all R_{ij} and E_{in} must be functions,
- for every i , all R_{i1}, \dots, R_{in} and E_{in} must have disjoint domains.

As has been proved in [23], if (P_1, \dots, P_n) is the least solution of (7.2-1), then P_i is the input-output relation on the path from node i to node n . Therefore, if we assume that 1 represents the initial node, and n is the final node, then P_1 is the input-output relation (the denotation) of our program. The class of iterative programs understood in that way, together with their correctness-proof rules, had been investigated in [19] and [23]. It is worth mentioning in this place that P_i 's correspond to A. Mazurkiewicz *tail functions* [64] or D. Scott and Ch. Strachey *continuations* [75]. Both models were published in 1971.

Programmers of the decade 1950/1960 were competing with each other in building more and more complicated flowchart programs that usually nobody except them was able to understand. Unfortunately, quite frequently, the authors themselves were not able to predict the behavior of such programs.

As a reaction to these problems, first algorithmic programming languages such as Fortran and Algol-60 were created. They were offering tools for *structured programming* such as sequential composition, *if-then-else*, and *while*⁸⁷. Such programs were much easier to understand and also allowed for significant simplification of program-correctness proof rules.

In the sequel, we shall restrict our discussion to only three primary *structural constructors* since they allow for the construction of any flowchart:

1. sequential constructor denoted by a semicolon “;”,
2. conditional constructor *if-then-else-fi*,
3. loop constructor *while-do-od*.

The sequential composition is the composition of relations (functions) as defined in Sec. 2.6. To define the remaining constructors, we have to introduce additional concepts. Since in our case the denotations of boolean expressions are three-valued partial functions, each of them will be represented by two disjoint set of states:

$$C = \{s \mid p.s = tt\}$$

$$\neg C = \{s \mid p.s = ff\}$$

Of course, if p is a two-valued total predicate, then $C \mid \neg C = S$, and therefore only one set is necessary to represent it. Notice also that our model does not distinguish between the two cases:

$$p.s : \text{Error}$$

$$p.s = ?$$

In both of them $s : S - (C \mid \neg C)$. If we wanted to distinguish between these cases, we had to represent predicates by three disjoint sets:

$$C = \{s \mid p.s = tt\}$$

$$\neg C = \{s \mid p.s = ff\}$$

$$eC = \{s \mid p.s : \text{Error}\}$$

where $S - (C \mid \neg C \mid eC)$ would include states where the evaluation of p is indefinite. We are not going to do so, since in constructing correct programs we equally care about the avoidance of abortion and infinite computations, and therefore we can identify these two cases in our relational model. However, in the denotational model of **Lingua** the case of abortion has been distinguished from infinite looping, because the latter is not decidable.

⁸⁷ The author who introduced the term “structured programming” was a Dutch computer scientist Edsger Dijkstra (see [43] and [44]).

It may be interesting to see, how on the ground of our relational model, we can express the difference between McCarthy's and Kleene's operators of propositional calculus. E.g.

$$(A, \neg A) \text{ and } (B, \neg B) = (A \cap B, \neg A \mid A \cap \neg B) \quad \text{— McCarthy}$$

$$(A, \neg A) \text{ and } (B, \neg B) = (A \cap B, \neg A \mid \neg B) \quad \text{— Kleene}$$

Now, let P and Q represent arbitrary programs and the pair of disjoint sets of states $(C, \neg C)$ — an arbitrary three-valued partial predicate. Our three structural constructors may be defined as particular cases of the universal set of equations (7.2-1). We recall that for any set of states A

$$[A] = \{(a, a) \mid a : A\}$$

is a subset of identity relation (function).

Sequential composition; $P ; Q$

$$X = P Y$$

$$Y = Q$$

Therefore by theorem 2.3-2:

$$X = P Q$$

Conditional composition; $\text{if } (C, \neg C) \text{ then } P \text{ else } Q \text{ fi}$

$$X = [C] Y \mid [\neg C] Z$$

$$Y = P$$

$$Z = Q$$

where $[C]$ and $[\neg C]$ are identity functions (see Sec. 2.6). Therefore by theorem 2.3-2::

$$X = [C] P \mid [\neg C] Q$$

Loop; $\text{while } (C, \neg C) \text{ do } P \text{ od}$

$$X = [C] P X \mid [\neg C]$$

Therefore by theorem 7.3-1

$$X = ([C] P)^* [\neg C]$$

Summarizing our definition:

1. $P ; Q$ = $P Q$
2. **if** $(C, \neg C)$ **then** P **else** Q **fi** = $[C] P \mid [\neg C] Q$
3. **while** $(C, \neg C)$ **do** P **od** = $([C] P)^* [\neg C]$

At the end one methodological remark is necessary. Although in **Lingua** all programs are deterministic, hence correspond to functions rather than relations, in the relational theory of program correctness we shall mainly talk about arbitrary relations (with an exception of **while** loops), since in these cases determinism does not contribute to the simplification of proof rules.

7.4 Procedures and recursion

The next step towards the development of structured-programming techniques was the introduction of procedures and, in particular — recursive procedures. On the ground of the algebra of relations mutually recursive procedures may be regarded as components of a vector of relations (R_1, \dots, R_n) which is the least solution of a set of fixed-point *polynomial equations* of the form:

$$X_1 = \Psi_1.(X_1, \dots, X_n)$$

...

$$X_n = \Psi_n.(X_1, \dots, X_n)$$

In these equations, each $\Psi_i(X_1, \dots, X_n)$ is a polynomial, i.e., a combination of variables and constants by composition and union, e.g., $AXYB \mid XXC$. Such sets of equations may be regarded as single fixed-point equations in a CPO of relational vectors ordered content-wise, i.e., in the CPO over the carrier:

$$\text{Rel}(S, S)^{cn} = \{(R_1, \dots, R_n) \mid R_i : \text{Rel}(S, S)\}$$

Every such set of polynomial equations defines a vectorial function:

$$\Psi : \text{Rel}(S, S)^{cn} \mapsto \text{Rel}(S, S)^{cn}$$

$$\Psi.(R_1, \dots, R_n) = (\Psi_1.(R_1, \dots, R_n), \dots, \Psi_n.(R_1, \dots, R_n))$$

If each Ψ_i is continuous in all its variables, then Ψ is continuous as well, and therefore Kleene's theorem holds (Sec. 2.3).

Since the correctness problem for recursive procedures is much more complicated than in the iterative case (see [5]), we shall investigate in Sec. 7.6.2 and Sec. 7.7.2 a simple scheme of a recursive procedure with only one procedural call that corresponds to an equation of the form:

$$X = HXT \mid E \tag{7.4-2}$$

where $H, T, E : \text{Rel}(S, S)$ are relations called the *head* the *tail* and the *exit* of the procedure, respectively. Although this is certainly not a general scheme for a recursive procedure, it seems quite common in practice. This scheme will be referred to as a *simple recursion*.

Notice that (7.4-2) covers the case of the iterative instruction *while-do-od* with $H = [C]P$, $T = [S]$ and $E = [\neg C]$.

7.5 Three concepts of program correctness

To express the property of program correctness on the ground of binary relations, we shall use two operations of a composition of a relation with a set. Both are similar to sequential compositions of relations, as defined in Sec. 2.6. In the sequel A, B, C, \dots will denote subsets of the set of states S and P, Q, R, \dots will denote relations in $\text{Rel}(S, S)$. Both operations are denoted by the same symbol “ \bullet ”, which has also been used for a composition of functions:

$$A \bullet R = \{s \mid (\exists a:A) a R s\} \quad \text{— left composition; the image of } A \text{ by } R$$

$$R \bullet B = \{s \mid (\exists b:B) s R b\} \quad \text{— right composition; the coimage of } B \text{ by } R.$$

In the sequel, the symbol of composition “ \bullet ” will be omitted; hence we shall write AR and RA . Intuitively speaking (see Fig. 7.5-1):

- AR is the set of all final states of executions of R that start in A ; notice however that some of them may be at the same time final states of executions that start outside A ,
- RB is the set of all initial states of executions of R that terminate in B , but if R is not a function, then some of them may at the same time generate executions that terminate outside B or do not terminate at all.

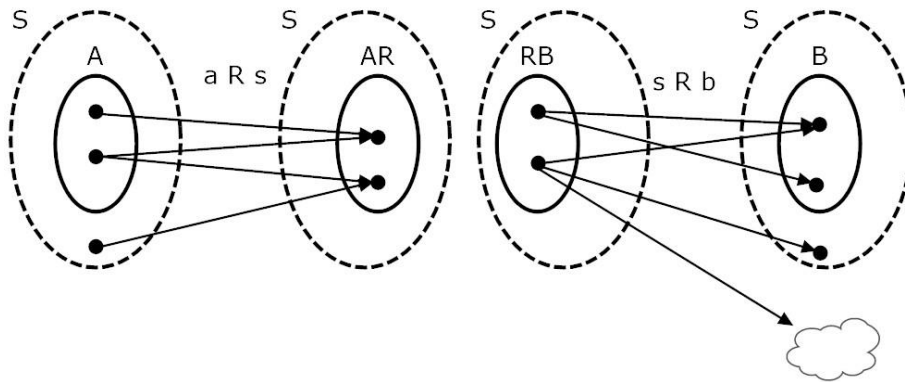


Fig. 7.5-2 Left- and right composition of a set with a relation

Both compositions of a relation with a set have properties similar to that of the composition of two relations. For instance, they are associative:

$$A(RQ) = (AR)Q$$

$$(RQ)B = R(QB)$$

and distributive over unions of sets and relations:

$$(A \mid B) R = (AR) \mid (BR)$$

$$A (R \mid Q) = (AR) \mid (AQ)$$

...

They are also monotone in each argument:

$$\text{if } A \subseteq B \text{ then } AR \subseteq BR$$

$$\text{if } R \subseteq Q \text{ then } AR \subseteq AQ$$

and analogously for right-hand-side composition. In fact, both operations are continuous in each argument. In the sequel, we shall assume that composition binds stronger than union hence we shall write

$$AR \mid BR \text{ instead of } (AR) \mid (BR)$$

Now let us recall (Sec. 2.6) that

$$[A] = \{(a, a) \mid a:A\}$$

denotes a subset of identity relation (i.e., function) on sets restricted to A

Lemma 7.5-1 For any $A, B, C \subseteq S$, and $R : \text{Rel}(S, S)$ the following equalities hold:

1. $[A]B = A \cap B$
2. $A[B] = A \cap B$
3. $(A \cap B)R = A [B] R$
4. $R(A \cap B) = R [A] B$
5. $(A \cap B)R \subseteq C$ is equivalent to $A[B]R \subseteq C$
6. if $A \subseteq [B]RC$ then $(A \cap B) \subseteq RC$ ■

Proofs are left to the reader.

Now we are ready to define three fundamental concepts concerning the correctness of programs: *partial correctness*, *weak total correctness*, and *clean total correctness*. All these concepts express the fact that if an input state of a program satisfies certain conditions, then the output state has expected properties. For instance, we may expect that a list-sorting program, when given an appropriate list (precondition), will return a sorted list (postcondition).

With every property of states, we can unambiguously associate a set of states with that property. As consequence correctness of a program R wrt a precondition A and postcondition B may be easily expressed in the algebra of relations and sets:

$AR \subseteq B$ — *partial correctness* of R wrt precondition A and postcondition B ;
 $(\forall a:A) \text{ if } (\exists b) aRb \text{ then } b:B$

$A \subseteq RB$ — *weak total correctness*⁸⁸ of R wrt precondition A and postcondition B ;
 $(\forall a:A) (\exists b) aRb \text{ and } b:B$

Partial correctness means that every execution that starts in A , if it terminates, then it terminates in B . Set A is called *partial precondition*, and B is called *partial postcondition*. If B does not contain error-carrying states then we talk about *clean partial correctness*.

Weak total correctness means that for every state a in A , there exists an execution that starts in a and terminates in B . Set A is called *weak total precondition*, and B is called *weak total postcondition*. The adjective “weak” expresses the fact that the existence of an execution from a to B does not exclude that other executions starting with a may terminate outside B or do not terminate at all. Similarly as in the former case, if B does not contain error-carrying states then we talk about *weak clean total correctness*.

Both defined concepts of program correctness were historically introduced for deterministic programs, i.e., for the case where R was a function. In such cases, the inclusion $A \subseteq RB$ means that each execution of R that starts in A terminates in B . That property will be called *total correctness* or *clean total correctness* respectively.

As is easy to see, in the non-deterministic case, none of the partial and total correctness is stronger than the other. Indeed, partial correctness does not imply termination, and the existence of one terminating execution from a to B does not mean that any terminating execution starting in a will terminate in B .

In the deterministic case, however, total correctness obviously implies partial correctness. i.e., for any partial function $F : S \rightarrow S$,

$$A \subseteq FB \text{ implies } AF \subseteq B \quad (7.5-1)$$

The following implication is also true:

$$\text{if } AF \subseteq B \text{ and for every } a : A, F.a \text{ is defined, then } A \subseteq FB \quad (7.5-2)$$

Both observations lead to the following theorem:

Theorem 7.5-1 *If F is a function then for any $A, B \subseteq S$ the following facts are equivalent:*

- $A \subseteq FB$ — *total correctness of F wrt A and B*
- $AF \subseteq B$ and $A \subseteq FS$ — *partial correctness of F wrt A and B , plus termination of F on A*

Clean termination of a deterministic program F on A means that F is a total function of A , and $F.a$ never carries an error.

We say that a deterministic program has a *halting property* in A , if no execution of that program that starts in A is infinite.

For many “practical programs”, the halting property may be so obvious that it does not need a formal proof. For instance, the program:

```
pre n, m > 0
  x := 1;
  y := m;
  while x < n
```

⁸⁸ In the earlier versions of the book the weak total correctness of relations was called just total correctness. Krzysztof Apt convinced me that such wording may lead to misunderstanding. He also pointed out that in [6] written by him and two other authors the notion of weak total correctness is used in a slightly different way. It is used in the context of distributed programs and combines partial correctness with absence of failures and divergence freedom.

```

do;
  x := x+1;
  y := y*m
od
post y = m^n

```

obviously halts for every n . However, there are cases where the halting property may be far from evident. One such program is displayed on the front of Warsaw University Library:

```

x := n;
while x > 1
do
  if x mod 2 = 0 then x := x/2 else x := 3x + 1 fi
od

```

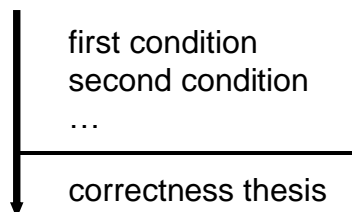
Under the program we see the following question: “Why for every $n > 0$ this program stops?”. This question is, however, not quite adequate, since today we do not know if this program has a halting property. It is a well-known *Collatz hypothesis* formulated in 1937 and not answered until today. At the date, I am writing these words (March 2021), it has been proved⁸⁹ only that the hypothesis is true for all $n < 5 \cdot 2^{90}$.

A similar situation concerns *Fermat's theorem*⁹¹ that was announced in the year 1637 and proved only in 1994 by a British mathematician Andrew Wiles. His proof is 100 pages long and uses an advanced topological theory of elliptic curves. Fermat theorem can be also formulated as a halting problem.

On the ground of the theory of computability, it has been proved (Alan Turing) that there is no algorithm which for every program and every input state could effectively — i.e., in a finite number of steps — decide whether this program stops for this input state.

Theorem 7.5-2 *In the general case, the termination property of programs is not decidable. ■*

In the sequel, proof rules for program correctness will be expressed by showing in which way the correctness of composed programs may be proved by proving the correctness of their components. In the most general case such rules will be written in the following form:



where the arrow shows the direction of implication. In some rules, we have both-sided arrows, which means that the implication is of the *iff*-type. As we shall see in Sec. 8.5, these rules indicate ways of building correct programs from correct components.

⁸⁹ One could (naïvely) expect that this result was proved by a simple checking by means of a ultra-fast computer. However, as is easy to check, if we assume that the execution of Collatz program for any $n < 5 \cdot 2^{89}$ takes on the average 1 nanosecond then such a check would take a time longer then 10^{65} times the age of the universe.

⁹⁰ I once fell victim to this hypothesis, when I was referring my work on total correctness of programs at the University of Saarbrücken. When I said that with my method one can easily prove the termination of the program, someone in the room asked me to illustrate this in a very simple example, and he gave me the Collatz program. I did not know this example, so I wrote the program on the board and proceeded to analyze it. Since I was not able to solve the problem off hand, I said I'd think about it this evening. But I still haven't had that proof in the evening. What a shame — such a simple program, and I cannot cope with it. After returning to Warsaw I showed the problem to my colleagues and then I was enlightened that I was not the only one who was not able to solve the Collatz's problem. I was truly relieved.

⁹¹ This theorem claims that for no integer $n > 2$ there exist three positive integers x, y, z that satisfy the equality $x^n + y^n = z^n$. That theorem had been written in 1637 by Fermat on the margin of a book together with a commentary that he found a “marvellously simple proof” of the theorem which was however too long to fit to the margin. The theorem has been proved by Andrew Wiles in 1993.

It should be emphasized in this place that in our approach to program correctness, we are not building any “logic of programs” in Hoare’s style (cf. [55], [4], and [5]). We only construct a set-theoretical (denotational) model of programs where the latter are represented by binary relations (or functions). On the ground of this model, program correctness is expressed by inclusions of the form $AP \subseteq B$ or $AP \subseteq B$. Then, we formulate and prove some lemmas which may be used in proving programs correctness. For short, these lemmas are called *Proof rules*.

In the end, one comment about the use of single sets of states $A, B, \dots \subseteq S$ as pre- and post-conditions, rather than pairs $(C, \neg C)$, which represent three-valued predicates. As a matter of fact A, B, \dots also correspond to three-valued predicates, but if we use them as pre- and post-conditions, we are only interested in their “domains of satisfaction”, i.e., in the first element of each pair $(C, \neg C)$. For instance, in proving the correctness of a program with a precondition:

$$1/x > 2 \tag{*}$$

we are only interested in the behavior of the program whenever our precondition is satisfied. We do not care about that behavior in all other cases. If, however, condition (*) appears as a boolean expression of an *if-then-else-fi* instruction, we must not forget that for $x = 0$ the value of that condition is neither true or false. It will be better seen in Sec. 8.

7.6 Partial correctness

Although our primary concern is total correctness of programs, the methods of proving partial correctness are of interest too since in the deterministic case, proof of total correctness may be reduced to a proof of partial correctness plus a proof of termination (Theorem 7.5-1). In turn, although in the general case termination property is not decidable, in many practical cases, it may be quite easy to prove.

7.6.1 Sequential composition and branching

When defining program correctness proof rules, it is worth distinguishing between two classes of program constructors: *simple constructors* that do not introduce repetition mechanisms and *recursive constructors* which introduce such mechanisms. The former are defined by composition and union of relations; the latter require fixed-point equations. From this perspective, iteration is a particular case of recursion.

The most frequently used simple constructors of programs are sequential composition and branching.

Rule 7.6.1-1 Partial correctness of a sequential composition

For arbitrary $A, D \subseteq S$ and $P, Q : \text{Rel}(S, S)$ the following rule is satisfied:

$$\begin{array}{l} \uparrow \text{there exist conditions } B \text{ and } C \text{ such that:} \\ (1) AP \subseteq B \\ (2) CQ \subseteq D \\ (3) B \subseteq C \\ \hline (4) A(PQ) \subseteq D \\ \downarrow \end{array}$$

Proof From (1), (2) and the monotonicity of composition

$$(AP)Q \subseteq CQ \subseteq D$$

hence from the associativity of composition

$$A(PQ) \subseteq D.$$

To prove the bottom-to-top implication is sufficient to set

$$B = C = AP$$

Hence $AP \subseteq B$ and $BQ = APQ \subseteq D$ ■

Rule 7.6.1-2 Partial correctness of if-then-else-fi

For arbitrary $A, D, C, \neg C \subseteq S$ and $P, Q : \text{Rel}(S, S)$, if $C \cap \neg C = \emptyset$, then the following rule is satisfied:

$$\begin{array}{l} \uparrow \\ (1) (A \cap C)P \quad \subseteq B \\ (2) (A \cap \neg C)Q \quad \subseteq B \\ \hline (3) A \text{ if } (C, \neg C) \text{ then } P \text{ else } Q \text{ fi} \subseteq B \\ \downarrow \end{array}$$

The proof is obvious.

In the end, three more rules which follow directly from the monotonicity of composition of a set with a relation.

Rule 7.6.1-3 Strengthening a partial precondition

For every $P : \text{Rel}(S, S)$ and any $A, B, C \subseteq S$ the following rule holds:

$$\begin{array}{l} | AP \subseteq B \\ | C \subseteq A \\ \hline \downarrow CP \subseteq B \end{array}$$

Rule 7.6.1-4 Weakening a partial postcondition

For every $P : \text{Rel}(S, S)$ and any $A, B, C \subseteq S$ the following rule holds:

$$\begin{array}{l} | AP \subseteq B \\ | B \subseteq C \\ \hline \downarrow AP \subseteq C \end{array}$$

Rule 7.6.1-5 The conjunction of pre- and postconditions

For every $P : \text{Rel}(S, S)$ and any $A, B, C, D \subseteq S$ the following rule holds:

$$\begin{array}{l} | AP \subseteq B \\ | CP \subseteq D \\ \hline \downarrow (A \cap C)P \subseteq B \cap D \end{array}$$

In the present section we skip the problem of proving properties of atomic components of programs such as, e.g., assignments or variable declarations. It is because in the model of abstract binary relations such rules cannot be expressed. This issue will be discussed in Sec. 8 where **Lingua-2V** enters the game.

7.6.2 Recursion and iteration

In order to formulate proof rules for mutually recursive procedures, we generalize the operation of composition of relations with relations and with sets to the case of vectors of respectively relations and sets:

$$(P_1, \dots, P_n) (R_1, \dots, R_n) = (P_1 R_1, \dots, P_n R_n)$$

and analogously for the composition of a relation with sets. In an obvious way, we can also generalize the inclusion of sets to the inclusion of vectors:

$(A_1, \dots, A_n) \subseteq (B_1, \dots, B_n)$ means $A_1 \subseteq B_1$ and ... and $A_n \subseteq B_n$

For simplicity, the inclusion between vectors of sets is denoted by the same symbol as the inclusion of sets. In the sequel vectors of sets and relations as well as operations on them will be written with boldface characters.

A vector of relations \mathbf{R} is said to be *partially correct* wrt the vectors of sets \mathbf{A} and \mathbf{B} (with appropriate numbers of elements) iff $\mathbf{A} \mathbf{R} \subseteq \mathbf{B}$. The notion of a continuous function is generalized to the case of vectorial functions in an obvious way.

Now we can formulate partial-correctness proof rule in the general case of fixed-points of continuous functions on vectors of relations. Although this case is restricted to polynomial functions, this assumption does not contribute to the simplicity of the rule. For concrete, simple polynomials, such rules will be shown a little later in this section.

Rule 7.6.2-1 Partial correctness of a vector of relations defined by a fixed-point equation

For every continuous function $\Psi : \text{Rel}(S, S)^{cn} \mapsto \text{Rel}(S, S)^{cn}$, if \mathbf{R} is the least solution of the equation $X = \Psi.X$, then for any $\mathbf{A}, \mathbf{B} : S^{cn}$ the following rule holds, where $\emptyset = (\emptyset, \dots, \emptyset)$ is a n -element vector of empty relations:

$$\begin{array}{l}
 \uparrow \text{ there exists a family of (vectors of) preconditions } \{\mathbf{A}_i \mid i \geq 0\} \\
 \text{ and a family of (vectors of) postconditions } \{\mathbf{B}_i \mid i \geq 0\} \text{ such that} \\
 (1) (\forall i \geq 0) \mathbf{A} \subseteq \mathbf{A}_i \\
 (2) (\forall i \geq 0) \mathbf{A}_i \Psi^i.\emptyset \subseteq \mathbf{B}_i \\
 (3) \bigcup \{\mathbf{B}_i \mid i \geq 0\} \subseteq \mathbf{B} \\
 \hline
 (4) \mathbf{A} \mathbf{R} \subseteq \mathbf{B} \\
 \downarrow
 \end{array}$$

Proof Form Kleene’s theorem (Sec. 2.3)

$$\mathbf{R} = \mathbf{U} \{\Psi^i.\emptyset \mid i \geq 0\}$$

Adding the components of (1) sidewise we obtain

$$\mathbf{U} (\mathbf{A}_i \{\Psi^i.\emptyset \mid i \geq 0\}) \subseteq \mathbf{U} \{\mathbf{B}_i \mid i \geq 0\}$$

hence from (1) and (3), we have (4). To prove the bottom-up implication, we assume

$$\mathbf{B}_i = \mathbf{A} (\Psi^i.\emptyset) \text{ for } i \geq 0 \text{ and}$$

$$\mathbf{A}_i = \mathbf{A} \blacksquare$$

From this rule, we obtain immediately a rule for single recursion, i.e., where $n = 1$:

Rule 7.6.2-2 Partial correctness of a relation defined by a fixed-point equation

For every continuous function $\Psi : \text{Rel}(S, S) \mapsto \text{Rel}(S, S)$, if R is the least solution of the equation $X = \Psi.X$, then for any $A, B \subseteq S$ the following rule holds:

$$\begin{array}{l}
 \uparrow \text{ there exists a family of preconditions } \{A_i \mid i \geq 0\} \\
 \text{ and a family of postconditions } \{B_i \mid i \geq 0\} \text{ such that} \\
 (1) (\forall i \geq 0) A_i \Psi^i.\emptyset \subseteq B_i \\
 (2) (\forall i \geq 0) A \subseteq A_i \\
 (2) \bigcup \{B_i \mid i \geq 0\} \subseteq B \\
 \hline
 (3) AR \subseteq B \\
 \downarrow
 \end{array}$$

We can also formulate more specific rules for each particular polynomial function, e.g., for the simple-recursion constructor as defined in Sec. 7.4. Below two versions of such a rule:

Rule 7.6.2-3 Partial correctness of a relation defined by simple recursion (version 1)

For any $H, T, E : \text{Rel}(S, S)$, if the relation R is the least solution of the equation

$$X = HXT \mid E$$

then for any $A, B \subseteq S$ the following rule holds:

$$\begin{array}{l} \uparrow \\ \text{there exists a family of preconditions } \{A_i \mid i \geq 0\} \\ \text{and a family of postconditions } \{B_i \mid i \geq 0\} \text{ such that} \\ (1) (\forall i \geq 0) A_i H_i E T_i \subseteq B_i \\ (2) (\forall i \geq 0) A \subseteq A_i \\ (2) \bigcup \{B_i \mid i \geq 0\} \subseteq B \\ \hline (3) AR \subseteq B \\ \downarrow \end{array}$$

The proof follows immediately from Rule 7.6.2-2 and from the fact that, as is easy to prove,

$$R = \bigcup \{H^i E T^i \mid i \geq 0\} \blacksquare$$

The following top-down-implication rule with a stronger assumption may be useful as well:

Rule 7.6.2-4 Partial correctness of a relation defined by simple recursion (version 2)

For any $H, T, E : \text{Rel}(S, S)$, if the relation R is the least solution of the equation

$$X = HXT \mid E$$

then for any $A, B \subseteq S$ the following rule holds:

$$\begin{array}{l} \downarrow \\ (1) (\forall Q) (AQ \subseteq B \text{ implies } A(HQT) \subseteq B) \\ (2) AE \subseteq B \\ \hline (3) AR \subseteq B \end{array}$$

Proof From (1) and (2) we can prove by induction that for every $i \geq 0$:

$$A (H^i E T^i) \subseteq B$$

and, therefore, by side-wise summation, we get (3). \blacksquare

As Andrzej Tarlecki pointed to me, this rule may also be written in an alternative way:

Rule 7.6.2-5 A Partial correctness of a relation defined by simple recursion (version 3)

For any $H, T, E : \text{Rel}(S, S)$, if the relation R is the least solution of the equation

$$X = HXT \mid E$$

then for any $A, B \subseteq S$ the following rule holds:

$$\begin{array}{l} \downarrow \\ (1) AH \subseteq A \\ (2) AE \subseteq B \\ (3) BT \subseteq B \\ \hline (4) AR \subseteq B \end{array}$$

Proof The three inclusions (1), (2), and (3) imply that for any $i > 0$, we have

$$A (H^i E T^i) \subseteq A E T^i \subseteq B T^i \subseteq B. \blacksquare$$

Now let us denote by

while (C, $\neg C$) **do** P **od**

the least solution of the equation

$$X = [C]PX \mid [\neg C].$$

Setting $H = [C]P$, $T = [S]$ and $E = [\neg C]$ from both general rules we can draw rules for while-do-od iteration:

Rule 7.6.2-6 Partial correctness of while-do-od loop (version 1)

For every relation $P : \text{Rel}(S,S)$, any disjoint $C, \neg C \subseteq S$, and any $A, B \subseteq S$ the following rule holds:

$$\begin{array}{l} \uparrow \text{there exists a family of postconditions } \{B_i \mid i \geq 0\} \text{ such} \\ \text{that} \\ (1) (\forall i \geq 0) A ([C]P)^i [\neg C] \subseteq B_i \\ (2) \bigcup \{B_i \mid i \geq 0\} \subseteq B \\ \hline (3) A \text{ while } (C, \neg C) \text{ do } P \text{ od} \subseteq B \end{array}$$

Rule 7.6.2-7 Partial correctness of while-do-od loop (version 2)

For every relation $P : \text{Rel}(S,S)$, any disjoint $C, \neg C \subseteq S$, and any $A, B \subseteq S$ the following rule holds:

$$\begin{array}{l} (1) (\forall Q) A Q \subseteq B \text{ implies } A [C]QP \subseteq B \\ (2) A [\neg C] \subseteq B \\ \hline (3) A \text{ while } (C, \neg C) \text{ do } P \text{ od} \subseteq B \end{array}$$

■

In the literature, the following rule is also well known, although it is usually formulated for the case of two-valued predicates, i.e. where $C \mid \neg C = S$

Rule 7.6.2-8 Partial correctness of while-do-od loop (version 3)

For every relation $P : \text{Rel}(S,S)$, for any disjoint $C, \neg C \subseteq S$, any $A, B \subseteq S$, the following rule is satisfied:

$$\begin{array}{l} \uparrow \text{there exists } N \subseteq S \text{ (called loop invariant) such that:} \\ (1) (N \cap C) P \subseteq N \\ (2) A \subseteq N \\ (3) N [\neg C] \subseteq B \\ \hline (4) A \text{ while } (C, \neg C) \text{ do } P \text{ od} \subseteq B \end{array}$$

■

Proof Let (1) – (3) be satisfied. Since

$$(N \cap C) P = N [C] P$$

from (1) we can prove by induction:

$$N([C]P)^i \subseteq N \text{ for all } i \geq 0$$

Therefore and from (2)

$$A([C]P)^i \subseteq N \text{ for all } i \geq 0$$

hence from (3)

$$A([C]P)^i[\neg C] \subseteq N[\neg C] \subseteq B \text{ for all } i \geq 0$$

In summing these inclusions sidewise, we get (4). Now assume that (4) is satisfied and let us set:

$$(5) N = A([C]P)^*$$

Therefore and from (4) we get $N[\neg C] \subseteq B$, hence (3). In turn (5) is equivalent to

$$N = A \mid A([C]P)^+,$$

hence (2). To prove (1) notice that:

$$(N \cap C)P = N[C]P = A[C]P \mid A([C]P)^+[C]P = A([C]P)^+ \subseteq N \quad \blacksquare$$

7.7 Weak total correctness

Rules for weak total correctness are used to prove that if an input state of a program satisfies a precondition, then at least one execution of that program will terminate with postconditions being satisfied. If our program is deterministic, then weak total correctness coincides with clean total correctness which means that the unique execution of a program terminates with a state satisfying a postcondition.

7.7.1 Sequential composition and branching

Rule 7.7.1-1 Weak total correctness of a composition

For any $A, D \subseteq S$ and $P, Q : \text{Rel}(S, S)$ the following rule holds:

$$\begin{array}{l} \uparrow \text{there exist conditions } B \text{ and } C \text{ such that} \\ (1) A \subseteq PB \\ (2) C \subseteq QD \\ (3) B \subseteq C \\ \hline (4) A \subseteq (PQ)D \\ \downarrow \end{array}$$

■

Proof. From (1), (2) and (3) we immediately have:

$$A \subseteq PB \subseteq PC \subseteq P(QD) = (PQ) D.$$

Now assume that $A \subseteq (PQ)D$, which means that $A \subseteq P(QD)$. Assuming $B = C = QD$ we get (1) and (2). ■

Rule 7.7.1-2 Weak total correctness of if-then-else⁹²

For any $A, B, C, \neg C \subseteq S$ and $P, Q : \text{Rel}(S, S)$, if $C \cap \neg C = \emptyset$, then the following rule is satisfied:

$$\begin{array}{l} \uparrow (1) A \cap C \subseteq PB \\ (2) A \cap \neg C \subseteq QB \\ (3) A \subseteq C \mid \neg C \\ \hline (4) A \subseteq \text{if } (C, \neg C) \text{ then } P \text{ else } Q \text{ fi } B \\ \downarrow \end{array}$$

Proof. Let (1) – (3) be satisfied. Then:

⁹² Notice that in case of two-valued predicates, condition (3) is not necessary, since $C \mid \neg C = S$.

$$[C] (A \cap C) \subseteq [C] PB$$

$$[\neg C] (A \cap \neg C) \subseteq [\neg C] QB$$

Adding the inclusions sidewise:

$$[C] (A \cap C) \mid [\neg C] (A \cap \neg C) \subseteq [C] PB \mid [\neg C] QB = ([C]P \mid \mid [\neg C] Q) B$$

The following equalities are also true

$$[C] (A \cap C) = A \cap C$$

and analogously for $\neg C$. Hence and from (3)

$$[C] (A \cap C) \mid [\neg C] (A \cap \neg C) = (A \cap C) \mid (A \cap \neg C) = A$$

and finally

$$(4) A \subseteq [C] PB \mid [\neg C] QB$$

In turn, (4) implies $A \subseteq C \mid \neg C$, and from (4) and the fact that C and $\neg C$ are disjoint, follow (1) and (2). ■

Observe the assumption (3) in our rule. In the case of classical predicates where $C \mid \neg C = S$, this condition is a tautology. Therefore, in Hoare's logic, which is built for classical predicates, this condition is simply omitted. In that case, however, we can prove the total correctness of a program, that aborts. For an explanation and an example, see Sec. 8.5.2 and Rule 8.5.2-7.

In the end, three more rules for pre- and postconditions analogous to the respective rules for partial correctness.

Rule 7.7.1-3 The strengthening of a weak total precondition

For every $P : \text{Rel}(S,S)$ and any $A,B,C \subseteq S$ the following rule holds:

$$\frac{\begin{array}{l} A \subseteq PB \\ C \subseteq A \end{array}}{C \subseteq PB}$$

Rule 7.7.1-4 The weakening of a weak total postcondition

For every $P : \text{Rel}(S,S)$ and any $A,B,C \subseteq S$ the following rule holds:

$$\frac{\begin{array}{l} A \subseteq PB \\ B \subseteq C \end{array}}{A \subseteq PC}$$

Rule 7.7.1-5 The conjunction of conditions

For every $P : \text{Rel}(S,S)$ and any $A,B,C,D \subseteq S$ the following rule holds:

$$\frac{\begin{array}{l} A \subseteq PB \\ C \subseteq PD \end{array}}{A \cap C \subseteq P(B \cap D)}$$

7.7.2 Recursion and iteration

Similarly, as in the case of partial correctness, we start from the case of a general recursive operator.

Rule 7.7.2-1 Weak total correctness of a vector defined by a general fixed-point equation

For every continuous function $\Psi : \text{Rel}(\mathbf{S}, \mathbf{S})^{\text{cn}} \mapsto \text{Rel}(\mathbf{S}, \mathbf{S})^{\text{cn}}$, if \mathbf{R} is the least solution of $\mathbf{X} = \Psi.\mathbf{X}$, then the following rule holds, where $\emptyset = (\emptyset, \dots, \emptyset)$:

$$\begin{array}{l}
 \uparrow \\
 \text{there exists a family of preconditions } \{\mathbf{A}_i \mid i \geq 0\} \\
 \text{and a family of postconditions } \{\mathbf{B}_i \mid i \geq 0\} \text{ such that} \\
 (1) (\forall i \geq 0) \mathbf{A}_i \subseteq (\Psi^i.\emptyset)\mathbf{B}_i \\
 (2) \mathbf{A} \subseteq \mathbf{U}\{\mathbf{A}_i \mid i \geq 0\} \\
 (3) (\forall i \geq 0) \mathbf{B}_i \subseteq \mathbf{B} \\
 \hline
 (4) \mathbf{A} \subseteq \mathbf{R}\mathbf{B} \\
 \downarrow
 \end{array}$$

Proof If \mathbf{R} is the least fixed point of Ψ , then from the continuity of Ψ

$$(4) \mathbf{R} = \mathbf{U}\{\Psi^i.\emptyset \mid i \geq 0\}$$

Adding sidewise inclusions (1) we have

$$\mathbf{U}\{\mathbf{A}_i \mid i \geq 0\} \subseteq \mathbf{U}\{(\Psi^i.\emptyset)\mathbf{B}_i\}$$

Hence from (2) and (3), we have (4). Now assume that $\mathbf{A} \subseteq \mathbf{R}\mathbf{B}$ which means that

$$\mathbf{A} \subseteq \mathbf{U}\{\Psi^i.\emptyset \mid i \geq 0\} \mathbf{B}$$

Let for $i \geq 0$

$$\mathbf{A}_i = (\Psi^i.\emptyset) \mathbf{B} \text{ and}$$

$$\mathbf{B}_i = \mathbf{B}$$

Then obviously (1), (2), and (3) are satisfied. ■

From this rule for $n = 1$, we immediately have

Rule 7.7.2-2 Weak total correctness of a relation defined by a general fixed-point equation

For every continuous function $\Psi : \text{Rel}(\mathbf{S}, \mathbf{S}) \mapsto \text{Rel}(\mathbf{S}, \mathbf{S})$, if \mathbf{R} is the least solution of an equation $\mathbf{X} = \Psi.\mathbf{X}$, then the following rule holds:

$$\begin{array}{l}
 \uparrow \\
 \text{there exists a family of preconditions } \{\mathbf{A}_i \mid i \geq 0\} \\
 \text{and a family of postconditions } \{\mathbf{B}_i \mid i \geq 0\} \text{ such that} \\
 (1) (\forall i \geq 0) \mathbf{A}_i \subseteq (\Psi^i.\emptyset)\mathbf{B}_i \\
 (2) \mathbf{A} \subseteq \mathbf{U}\{\mathbf{A}_i \mid i \geq 0\} \\
 (3) (\forall i \geq 0) \mathbf{B}_i \subseteq \mathbf{B} \\
 \hline
 (4) \mathbf{A} \subseteq \mathbf{R}\mathbf{B} \\
 \downarrow
 \end{array}$$

As we know well, a call of a recursive procedure may generate an infinite execution. Therefore, it is worth asking a question, where our rule tackles the halting problem? Of course, formally, the halting property is guaranteed by (4). But where is it hidden in the proof of that fact, i.e., above the line? In fact, it is expressed by the conjunction of (1) and (2). If $\mathbf{a} : \mathbf{A}$, then by (2), there exist an $i \geq 0$ such that $\mathbf{a} : \mathbf{A}_i$. This fact guarantees by (1) the existence of an execution that starts in \mathbf{a} and terminates in \mathbf{B}_i after exactly i recursive calls of our procedure.

Of course, an analogous argument applies to Rule 7.7.2-1, but its interpretation in the case of a single procedure may be easier to appreciate.

Rule 7.7.2-3 Weak total correctness of a relation defined by simple recursion (version 1)

If relation R is the least solution of the equation $X = H X T \mid E$ then the following rule holds:

$$\begin{array}{l}
 \uparrow \text{there exists a family of preconditions } \{A_i \mid i \geq 0\} \\
 \text{and a family of postconditions } \{B_i \mid i \geq 0\} \text{ such that} \\
 (1) (\forall i \geq 0) A_i \subseteq (H^i E T^i) B_i \\
 (2) A \subseteq \bigcup \{A_i \mid i \geq 0\} \\
 (3) (\forall i \geq 0) B_i \subseteq B \\
 \hline
 (4) A \subseteq RB \\
 \downarrow
 \end{array}$$

Proof Define

$$\Psi.X = H X T \mid E$$

In this case

$$\Psi^0.\emptyset = E$$

$$\Psi^1.\emptyset = \Psi.(\Psi^0.\emptyset) = H (\Psi^0.\emptyset) T \mid E = H E T \mid E$$

$$\Psi^2.\emptyset = \Psi.(\Psi^1.\emptyset) = H (\Psi^1.\emptyset) T \mid E = H (H E T \mid E) T \mid E = H^2 E T^2 \mid H^1 E T^1 \mid E$$

Therefore, by induction, for any $n \geq 0$

$$\Psi^i.\emptyset = \bigcup \{ H^i E T^i \mid i=1,2,\dots,n \} \mid E = \bigcup \{ H^i E T^i \mid i=0,1,\dots,n \}$$

Now, by (1) and the monotonicity of composition of a relation with a set, we have for every $i \geq 0$

$$A_i \subseteq H^i E T^i B_i \subseteq (\bigcup \{ H^i E T^i \mid i=0,\dots,n \}) B_i \subseteq (\Psi^i.\emptyset) B_i$$

From this inclusion together with (2), (3) and Rule 7.7.2-2, we conclude

$$A \subseteq RB$$

In turn, if the inclusion is satisfied, then we set

$$A_i = (\Psi^i.\emptyset) B$$

$$B_i = B$$

With this settings (1) and (3) are obviously satisfied, and (2) is satisfied because

$$A \subseteq RB \subseteq \bigcup \{ \Psi^i.\emptyset \mid i \geq 0 \} B = \bigcup \{ (\Psi^i.\emptyset) B \mid i \geq 0 \} = \bigcup \{ A_i \mid i \geq 0 \} \blacksquare$$

Rule 7.7.2-4 Clean total correctness of a function defined by simple recursion (version 2)

If F is the least solution of the equation $X = HXT \mid E$ where H , T , and E are functions and the domains of H and E are disjoint, then the following rule holds:

$$\begin{array}{l}
 (1) (\forall Q) (AQ \subseteq B \text{ implies } A(HQT) \subseteq B) \\
 (2) AE \subseteq B \\
 (3) A \subseteq FS \\
 \hline
 (3) A \subseteq FB \\
 \downarrow
 \end{array}$$

Proof As is easy to prove, for any H , T , and E the least solution of our equation is

$$U\{ H^n E T^n \mid n \geq 0 \}$$

and if additionally H , T , and E are functions and the domains of H and E are disjoint, then this solution is a function. Now, by (1), (2) and the Rule 7.6.2-4, $AF \subseteq B$, i.e., F is partially correct wrt A and B . Since (3) means that F is total on A , by Theorem 7.5-1 we can claim that it is totally correct wrt A and B . ■

From Rule 7.7.2-3 we can immediately derive our first rule about while-do-od instruction based on the observation that **while** (C , $\neg C$) **do** P **od** is the least solution of the equation

$$X = [C]PX \mid [\neg C].$$

Let then R be the least solution of this equation, i.e.,

$$R = ([C]P)^*[\neg C].$$

Rule 7.7.2-4 Clean total correctness for nondeterministic while-do-od

↑	<i>there exists a family of preconditions</i> $\{A_i \mid i \geq 0\}$
↑	<i>and a family of postconditions</i> $\{B_i \mid i \geq 0\}$ <i>such that</i>
↑	(1) $(\forall i \geq 0) A_i \subseteq ([C]P)^i[\neg C] B_i$
↑	(2) $A \subseteq \bigcup \{A_i \mid i \geq 0\}$
↑	(3) $(\forall i \geq 0) B_i \subseteq B$
↓	(4) $A \subseteq RB$

The most commonly known version of a rule for **while-do-od** concerns a deterministic case, and does not require the construction of two infinite families of conditions. It is also based on a well-known method of proving the halting property of a loop. First, we introduce two auxiliary concepts.

We say that a function $F : S \rightarrow S$ has *limited replicability property* in a set $N \subseteq S$ if there exists no infinite sequence of the form $s, F.s, F.(F.s), \dots$ in N .

A partially ordered set $(U, >)$ is said to be *well-founded*, if there is no infinite decreasing sequence in it, i.e., a sequence $u_1 < u_2 < \dots$. The following obvious lemma is useful in proving the limited replicability of a function $F : S \rightarrow S$.

Lemma 7.7.2-1 If there exists a well-founded set $(U, <)$ and a function $K : N \mapsto U$ such that for any $a : N$, $F.a = !$, $F.a : N$ and

$$K.a > K.(F.b)$$

then F has limited replicability in N . ■

Now we can formulate our rule.

Rule 7.7.2-4 Clean total correctness of a deterministic while-do-od loop

For any function $F : S \rightarrow S$, any $A, B, N \subseteq S$, and any disjoint $C, \neg C \subseteq S$

↑	(1) $A \subseteq N$
↑	(2) $N \subseteq C \mid \neg C$
↑	(3) $N \cap \neg C \subseteq B$
↑	(4) $N \cap C \subseteq FN$ (clean total correctness of F)
↑	(5) $[C]F$ has limited replicability in N
↓	(6) $A \subseteq \mathbf{while} (C, \neg C) \mathbf{do} F \mathbf{od} B$

Proof Assume that (1), (2), (3) are satisfied but the inclusion

$$N \subseteq ([C]F)^*[\neg C]S.$$

does not hold. In that case, there exists $s_0 : \mathbf{N}$, that does not belong to

$$([\mathbf{C}]\mathbf{F})^*[\neg\mathbf{C}]\mathbf{S} = ([\mathbf{C}]\mathbf{F})^+[\neg\mathbf{C}]\mathbf{S} \mid \neg\mathbf{C},$$

and therefore s_0 does not belong to $\neg\mathbf{C}$. From there, by (3), $s_0 : \mathbf{N} \cap \mathbf{C}$, and therefore by (4), there exists s_1 such that $[\mathbf{C}]\mathbf{F}.s_0 = s_1$ and $s_1 : \mathbf{N}$. Therefore by (3)

$$s_1 : \mathbf{C} \mid \neg\mathbf{C}.$$

Now, s_1 cannot belong to $\neg\mathbf{C}$, since then s_0 would belong to

$$[\mathbf{C}]\mathbf{F}[\neg\mathbf{C}]\mathbf{S}$$

which is a subset of $([\mathbf{C}]\mathbf{F})^*[\neg\mathbf{C}]\mathbf{S}$. Reasoning in this way, we could prove by induction that for any $n \geq 0$ there exists a sequence $s_i : i = 0, 1, \dots, n$ such that $s_0 : \mathbf{N}$ and

$$s_i [\mathbf{C}]\mathbf{F} s_{i+1} \text{ and } s_i : \mathbf{N} \text{ for } i = 0, 1, \dots, n$$

Since \mathbf{F} is a function, this implies the existence of an infinite sequence

$$s_i [\mathbf{C}]\mathbf{F} s_{i+1} \text{ and } s_i : \mathbf{N} \text{ for } i = 0, 1, \dots$$

which contradicts (5). ■

8 LINGUA-2V — VALIDATING PROGRAMMING

By *validating programming*, we shall mean a programming technique that guarantees the total-correctness of programs wrt program specification, the latter being created in parallel with the program’s code. This technique was already mentioned in Sec. 1.1 and its abstract mathematical foundations are described in Sec. 7. The present section is devoted to general rules of equipping a language from the **Lingua** family with validating tools. These rules are illustrated by examples referring to **Lingua-2**.

The general idea of validating programming was sketched (without procedures) in my papers [20], [21] and [22] published at the turn of the decades 1970 and 1980. On that ground, I came to the conclusion that in order to create a language with rules that guarantee program correctness, one has to build a mathematical model of such a language. The next few years till the end of the decade of 1980. I devoted to the investigations of such models and the following 23 years (1990-2013) to run my family business (see Foreword). Therefore only in 2013, I have returned to my project, and the present book was the first step of it.

This chapter gained in clarity (I hope) due to a fruitful discussion (in 2018) with my former MetaSoft-teammate Stefan Sokołowski. Another teammate Ryszard Kubiak also contributed to this subject.

8.1 The structure of a validating language

Very briefly, a language of validating programming is a language of propositions called *metaprograms*. Each metaprogram is composed of two mutually nested layers:

1. *a programming layer* which is a program as defined earlier,
2. *a descriptive layer* which consists of pre- and post-conditions plus assertions (see Sec. 8.3) that are “nested” in-between instructions.

A metaprogram is said to be *correct* if its programming layer is totally correct (see Sec. 7.5) wrt its pre- and post-condition.

Validating programming consists in deriving correct programs from correct programs where the “initial” programs have to be proved correct in a traditional way. This situation is analogous to a formalised theory where we “derive” theorems from earlier proved theorems employing deduction rules. However, in deriving correct programs from correct programs, we sometimes change programs’ functionality (denotation) in preserving correctness. This phenomenon will be seen in Sec. 8.5.6, where we discuss the idea of *transformational programming*.

For every *source language* **Lingua-n** we may construct a corresponding language **Lingua-nV** of validating programming which contains all mechanisms of the source language plus four followings (syntactic) categories of its descriptive layer:

1. *Conditions* — the denotations of which are three-valued partial predicates on states, i.e., they may evaluate to tt, ff, an error, or maybe undefined (the case of looping).
2. *Specified instructions* — the denotation of which are partial functions on states (like the denotations of instruction) and where the descriptive layer describes the properties of the programming layer.
3. *Specified programs* — which are specified instructions preceded by a declaration.
4. *Propositions* — the denotations of which are classical boolean values tt and ff; propositions are split into three subcategories:

- 4.1. *properties* that express syntactic properties of programs, e.g., that a given procedure declaration appears in a declaration,
- 4.2. *metaconditions* that express the semantic properties of conditions, e.g., that a given condition is never false but may be undefined,
- 4.3. *metaprograms* that express total-correctness properties of programs which they include.

Propositions are assumed to be closed under classical boolean operators and classical quantifiers. It means that in constructing correct programs, and in talking about program correctness, we remain in the domain of classical logic.

Contrary to our philosophy *from denotations to syntax*, in constructing a language of validating programming, we proceed from syntax to denotations. This is the consequence of the fact that this time our starting point is an “already existing” syntax of a source-language, which has to be a subset — or better a layer — of the corresponding validating language.

8.2 Conditions

8.2.1 Generalities about conditions

To avoid too many technicalities, our conditions will not be defined in details. Instead, we shall only assume some of their properties. The description of these properties should show a way of building the category of *conditions* for each particular language from the **Lingua** family.

Classes of conditions will be described and illustrated with the help of their (anticipated) concrete syntax. In defining their semantics, we shall use the following notation for boolean (yokeless) values:

$$vt = (tt, (('boolean'), TT))$$

$$vf = (ff, (('boolean'), TT)).$$

The syntactic category of conditions is defined by the following clause:

$$\text{con} : \text{Condition} =$$

DatCon		data-oriented conditions
DecCon		declaration-oriented conditions
SpecInstruction @ Condition		algorithmic conditions
(Condition and Condition) (Condition or Condition) (not Condition)		
(\forall Identifier: Condition) (\exists Identifier: Condition)		

Data-oriented conditions, called simply data conditions, declaration-oriented conditions, and algorithmic conditions will be discussed in details in the subsequent sections. At the general level we shall assume that the denotations of conditions are partial functions from states to (boolean) values or errors, which means that their semantics is a function:

$$\text{Sco} : \text{Condition} \mapsto \text{State} \rightarrow \text{ValueE}$$

The partiality of the denotations of conditions is due to the fact that conditions may include data expressions. In the sequel we shall use the following notational conventions:

$$[\text{con}] = \text{Sco}.\text{[con]}$$

$$\{\text{con}\} = \{\text{sta} \mid [\text{con}].\text{sta} = tt\}$$

Why this is convenient will be seen a little later.

To gain commutativity of conjunction and alternative we assume that boolean constructors are defined in the Kleene’s style (see Sec. 2.9). Consequently, quantifiers will be defined accordingly:

\forall : Identifier \times Condition \mapsto Condition

```

[( $\forall$ ide: con)].sta =
  is-error.sta                                 $\rightarrow$  error.sta
  let
    (env, (vat, 'OK')) = sta
  for every val : Value, [con].(env, (vat[ide/val], 'OK')) = vt  $\rightarrow$  vt
  there is val : Value, [con].(env, (vat[ide/val], 'OK')) = vf  $\rightarrow$  vf
  true                                          $\rightarrow$  'never-false'

```

The message 'never-false' — which is formally regarded as an abstract error — is generated in situations where the value

$[con].(env, (vat[ide/val], 'OK'))$

is never vf, but at the same time is not always vt, i.e. if it is:

- either vt,
- or an error,
- or undefined⁹³,

and for at least one value val it is not equal vt. The existential quantifier is defined in the following way (which is to satisfy De Morgan's laws)

\exists : Identifier \times Condition \mapsto Condition

```

[( $\exists$ ide: con)].sta =
  is-error.sta                                 $\rightarrow$  error.sta
  let
    (env, (vat, 'OK')) = sta
  there is val : Value, [con].(env, (vat[ide/val], 'OK')) = vt  $\rightarrow$  vt
  for every val : Value, [con].(env, (vat[ide/val], 'OK')) = vf  $\rightarrow$  vf
  true                                          $\rightarrow$  'never-true'

```

Notice that the equality

$[(\forall ide: con)].sta = vf$

holds even if for some value val, the value of con is an error and analogously in the situation where

$[(\exists ide: con)].sta = vt$.

These facts mean that quantifiers are defined according to Kleene's philosophy rather than to that of McCarthy. In the case of boolean expressions, which are evaluated during program execution, Kleene's philosophy was not acceptable since it would lead to the necessity of parallel computations (cf. Sec. 2.9). However, in the case of conditions, which are not executed, Kleene's calculus is not only acceptable but — in the case of quantifiers — even better. This claim may be justified by the example of a condition:

$(\exists x: 1/x > 2)$

the value of which in the calculus of McCarthy would be undefined, since it is undefined for $x = 0$. More on the consequences of that choice in [26] and [58].

⁹³ The assumption that 'never-false' is issued in the case of undefinedness does not mean that undefinedness has to be computable. This is due to the fact that condition will never be executed by programs. They will only be "proved".

8.2.2 Data-oriented conditions

Data-oriented conditions describe properties of data assigned to identifiers in states. They split into two groups.

In the first group, we have boolean expressions of the source language. Of course, we assume that the semantics of this group of conditions coincides with the semantics of data expressions of the source language.

In the second group, we have expressions that correspond to predicates not available in the programming layer of the language but needed to express some specific properties of our programs. For instance, we may wish to claim that after the execution of a quick-sort, the resulting list is lexicographically ordered. This group of expressions will depend on the expected area of the application of our language.

We shall assume that in all **Lingua-nV** the second group will include conditions which express the equality of two values:

$$\text{dae-1} = \text{dae-2}$$

Here both $\text{dae-}i$ are arbitrary data expressions. This does not mean, however, that at the level of **Lingua** we allow the comparisons of arbitrary values. Such an equality may appear only in the descriptive layer of **Lingua-nV**, which means that $\text{dae-1} = \text{dae-2}$ is a condition, but not necessarily a boolean data-expression!

8.2.3 Declaration-oriented conditions

The category of *declaration-oriented conditions* splits into two categories. In the first category we have conditions which are satisfied if a given identifier is not bound in a state⁹⁴:

$$[\text{is-free}(\text{ide})].\text{sta} = \text{not declared.ide.sta},$$

where the function `declared`, has been defined in the preamble to Sec. 5.1.4.

Conditions of the second category describe properties of output states of four sorts of declarations: of data variables, type constants, imperative procedures, and functional procedures. We describe these conditions by semantic clauses, where we also show their syntax.

```
[ide is tex].sta =
  is-error.sta      → error.sta
  let
    (env, (vat, 'OK')) = sta
    vat.ide = ?      → vf
    Ste.[tex].sta : Error → Ste.[tex].sta
  let
    (dat, typ) = Sde.[ide].sta
    typ-e      = Ste.[tex].sta
    typ = typ-e      → vt
    typ ≠ typ-e     → vf
```

This condition describes the fact that the identifier `ide` has been declared as a data variable of type defined by the type expression `tex`. An example of such a condition may be:

```
length is real
```

or

```
employee is employee-record-type
```

An analogous condition of the form

```
ide is-type tex
```

⁹⁴ We have one such condition for every identifier.

for instance

```
employee is-type employee-record-type
```

expresses the fact that `ide` has been declared as a type constant assigned to a type described by `tex`. At the level of colloquial syntax, we allow grouping similar conditions into one, as e.g., in

```
x, y, z is number    or
a, b, c is-type employee-record-type
```

The third class of conditions expresses the fact that a given identifier `ide` has been declared as an imperative procedure with declaration `ipd` in a given state `sta`. Formally we define the following condition:

```
[ ide proc-with ipd ].sta = vt
```

iff

- (1) `sta` does not carry an error,
- (2) `ipd` is a declaration of `ide`, which means that `ipd` is of the form

```
proc ide (val LisForPar ref LisForPar) Program endproc,
```
- (3) there exists a (initial) state `sta-ini` such

```
sta = Sipd.[ipd].sta-ini.
```

We assume that otherwise

```
[ ide proc-with ipd ].sta = vf
```

Assumption (3) implies that any state of an execution of the instruction part of a program whose declaration includes `ipd` satisfies our condition. In Sec. 8.5.3 this condition will be used in building a correctness rule for procedure calls.

For functional procedures, we introduce an analogous condition of the syntactic form:

```
ide fun-with fpd
```

A formal definition of its denotation is left to the reader.

The last class of conditions corresponds to the function *dynamically-compatible* defined in Sec. 6.2.3 and concerning the compatibility of the denotations of formal and actual parameters. Let then `fpd-v`, `fpd-r`, `apd-v`, `apd-r` be the list of formal-parameter denotations (value- and reference-) and the corresponding actual-parameter denotations (value- and reference-):

```
[ conformant (acp-v, acp-r, lap-v, lap-r) ].sta =
  is-error.sta                                → error.sta
let
  ((tye, pre), (vat, 'OK')) = sta
  fpd-v                      = Sfpa[fpd-v]
  fpd-r                      = Sfpa[fpd-v]
  apd-v                      = Sapa[apd-v]
  apd-r                      = Sapa[apd-r]
  dynamically-compatible.(fpd-v, fpd-r, apd-v, apd-r).(tye, vat) = 'OK' → vt
true                                → vf
```

8.2.4 Algorithmic conditions

*Algorithmic conditions*⁹⁵ are sequential combinations of a declaration `dec`, a specified instruction `sin` (see Sec. 8.3) and a condition `con`, and is of the syntactic form

⁹⁵ Algorithmic conditions have been introduced in *algorithmic logic* developed at Warsaw University in the years 1970-1990 (see [10]).

```
dec ; sin @ con
```

Its semantics is the following:

$$[\text{dec ; sin @ con}] = \text{Sde}.[\text{dec}] \bullet \text{Ssi}.[\text{sin}] \bullet [\text{con}]$$

As we see, the logical value of a condition `dec ; sin @ con` in a state `sta` equals the value of `con` in the state $\text{Sde}.[\text{dec}] \bullet \text{Ssi}.[\text{sin}].\text{sta}$, i.e. in the terminal state of the execution of `sin` that starts with `sta`. As follows from investigations of Sec. 7.5, `sin @ con` is the weakest precondition that guarantees that the execution of `sin` terminates with a state that satisfies `con`. Notice also that `con` is an arbitrary condition, and, therefore, may be an algorithmic condition as well.

Algorithmic conditions, similarly as data-conditions, may evaluate to non-boolean values, and for some states may be undefined, which correspond to infinite evaluations.

8.3 Specified instructions

Intuitively speaking *specified instructions* or just *specinstructions* are instructions with nested assertions. They later describe properties of states intermediate in the executions of these instructions. Their syntax is defined by the following clause

```
sin : SpecInstruction =
  Instruction |
  asr Condition rsa |
  if DatExp then SpecInstruction else SpecInstruction fi |
  if-error DatExp then SpecInstruction fi |
  while DatExp do SpecInstruction od |
  SpecInstruction ; SpecInstruction
```

As we see, specinstructions contain all “usual” instructions and additionally instructions that are called *assertion*, and are of the form

```
asr con rsa,
```

where `con` is an arbitrary condition.

The denotations of specinstructions belong to the same domain as the denotations of instructions, hence their semantics is a function:

$$\text{Ssi} : \text{SpecInstruction} \mapsto \text{State} \rightarrow \text{State}$$

This function is defined in the following way:

$$\text{Ssi}.[\text{ins}] = \text{Sin}.[\text{ins}] \quad \text{for every instruction } \text{ins} : \text{Instruction}$$

```
Ssi.[asr con rsa].sta =
  is-error.sta    → sta
  [con].sta = ?   → ?
  [con].sta = vt  → sta
  true           → sta ◀ ‘assertion-not-satisfied’
```

The semantics of specinstructions, which are instructions, coincide with the semantics of instruction. In the case of assertions, if their conditions hold, then the state remains unchanged, and otherwise, an error message is generated. Notice that the error message ‘assertion-not-satisfied’ will appear in two situations:

1. when the value of the condition is `vf`,
2. when the value of the condition is an error.

For the remaining four clauses, our semantics is defined analogously to the semantics of instructions.

Notice that if a metaprogram is correct, then all assertions (if any) that appear in the program must be satisfied in the course of any execution whose initial state satisfies the precondition. This statement follows from two facts⁹⁶:

1. every non-satisfied assertion generates an error message,
2. every condition evaluated in an error-carrying state generates an error message.

The described syntax and semantics of specinstructions constitute a fundament for the definitions of program-construction rules. In this context, assertions describe properties of states that appear in the course of program execution and may indicate which program transformations are applicable to that program.

Sometimes assertions may be satisfied on a certain “interval” of successive *atomic instructions* (i.e. assignments and procedure calls), possibly with the exclusion of a certain subinterval of this interval. In such a case, in order to avoid repeating the same assertion many times between successive instructions, we use two colloquial notational abbreviations of the form

```

asr con: sin rsa (*)
off sin on (**)
```

The first of them corresponds to an instruction resulting from `sin` by the insertion of `asr con rsa` between any two atomic instructions with the exclusion of each exclusion-interval and each error-handling instruction. The specinstruction (*) will be called the *on-range of con*, and we shall also say that in `sin` the condition `con` has been *set-on*.

The colloquialism (**) intuitively indicates that if `sin` is a part of an on-range of a (larger) specinstruction, then within `sin` all previously set-on conditions do not need to be satisfied⁹⁷.

Summing up, both (*) and (**) are not specinstructions but just notational conventions. They will be formalized as colloquialisms, i.e., by an appropriate restoring transformation. Consequently, they do not appear in concrete syntax. Notice that the specinstruction

```
asr con: sin rsa
```

cannot be given a denotational semantics, since for example two following specinstructions:

```

asr x > 0:
  x := x
rsa
asr x > 0:
  x := -x ; x := -x
rsa
```

have different denotations, although the denotations of their conditions and instructions are identical.

In this situation (*) and (**) have to be treated as a colloquialism. The corresponding restoring transformation is an identity function for all specinstructions without on-ranges of conditions and otherwise is defined by structural induction.

We start from the case where an on-range is an (ordinary) instruction. Since that case requires a structural induction again, we start from an assignment:

```

RT.[ asr con: ide := dae rsa ] =
  asr con rsa: ide := dae; asr con rsa
```

For yoke-assignments and procedure calls, the transformation is defined analogously. Next case is an error-handling instruction where the rule is similar to the former:

⁹⁶ These two facts follow from the semantics of conditions which has been designed under the assumption that **Lingua-2V** we do not deal with error elaboration.

⁹⁷ For the sake of simplicity I assume that in the **off-on** region all previously activated conditions do not need to be satisfied. An alternative would be, of course, a **off-on** clauses which indicates a particular condition to be off, but this more flexible form is left for future investigations.

```
RT.[ asr con: if-error dae then ins fi rsa]=
  asr con rsa; if-error dae then ins fi ; asr con rsa
```

The remaining subcases with ordinary instruction are defined in the following way:

```
RT.[asr con: if dae then ins-1 else ins-2 fi rsa]=
  asr con rsa;
  if dae
    then RT.[asr con ins-1 rsa]
    else RT.[asr con ins-2 rsa]
  fi;
  asr con rsa
```

```
RT.[asr con: while dae do ins od rsa]=
  asr con rsa;
  while dae do RT.[asr con: ins rsa] od;
  asr con rsa
```

```
RT.[asr con: ins-1 ; ins-2 rsa]=
  asr con rsa;
  RT.[asr con: ins-1 rsa];
  asr con rsa;
  RT.[asr con: ins-2 rsa]
  asr con rsa
```

```
RT.[asr con: if-error dae then ins fi rsa]=
  asr con rsa;
  if-error dae then RT.[asr con: ins rsa] fi;
  asr con rsa;
```

Now we have to consider the case where the on-range is a specinstruction which is not an instruction.

```
RT.[asr con off ins on rsa]= ins
```

As we see the assertion does not “penetrate” the instruction closed by the exclusion-brackets.

```
RT.[asr con-1: asr con-2 rsa rsa]=
  asr con-1 and con-2 rsa

RT.[asr con-1: asr con-2: sin rsa rsa]=
  RT.[asr con-1 and con-2: sin rsa]
```

The remaining cases connected to structured specinstructions are defined in a way analogous to the corresponding ordinary structured instructions.

The denotation of a specified program is defined as a sequential composition of the denotation of its declaration with the denotation of its specified instruction:

```
Ssp : SpecProgram  $\mapsto$  State  $\rightarrow$  ValueE
Ssp.[dec ; sin] = Sde.[dec] • Ssi.[sin]
```

8.4 Propositions

Propositions split into four classes:

- *syntactic propositions* — which describe properties of the syntax of programs and of their components,
- *metaconditions* — which describe semantic properties of conditions,
- *metainstructions* — which describe semantic properties of instructions,
- *metaprograms* — which describe the semantic properties of programs.

Contrary to conditions, whose values are boolean composites or errors, or may be undefined, the values of propositions may be only `tt` and `ff`. In executing programs, we evaluate three-valued partial predicates, whereas, in the descriptions of programs, we remain at the level of classical logic.

Similarly, as for conditions — and for the same reasons — the model of propositions is built from syntax to denotations.

8.4.1 Syntactic propositions

Syntactic propositions describe the syntactic properties of programs and their components. Below a few typical examples of such propositions.

- `IS-CORRECT` (`dec`) — no identifier has been declared twice in `dec`,
- `ipd IS-PRO-DEC-OF ide IN dec` — `ipd` is a procedure's declaration of `ide` in `dec`
- `fpd IS-FUN-DEC-OF ide IN dec` — `fpd` is a function's declaration of `ide` in `dec`
- `ide NOT-IN dec` — `ide` has not been declared in `dec`

The names of predicates that correspond to syntactic proposition are written in capital letters to distinguish them from operators corresponding to conditions. E.g. whereas

`ipd IS-PRO-DEC-OF ide IN dec`

is simply true or false, the value of the condition

`ide proc-with ipd`

may be `vt` or `vf`, depending on a state where it is evaluated.

8.4.2 Metaconditions

Metaconditions describe such properties of conditions that refer to their denotations. They do not belong to the syntax of **Lingua-2V** but to the level of **MetaSoft**, where we talk about programs. In order to define them we introduce a new notation:

$$\{\text{con}\} = \{\text{sta} : [\text{con}].\text{sta} = \text{vt}\}$$

Metaconditions are created by means of four constructors which we shall call *metapredicates*:

$$\Rightarrow, \sqsubseteq, \Leftrightarrow, \equiv : \text{Condition} \times \text{Condition} \mapsto \text{Proposition}$$

The denotations of metaconditions are classical logical values `tt` and `ff` and metapredicates correspond to binary relations between conditions:

- | | | | | | |
|--------------------|-------------------|--------------------|-----|---|---------------------------------------|
| <code>con-1</code> | \Rightarrow | <code>con-2</code> | iff | $\{\text{con-1}\} \sqsubseteq \{\text{con-2}\}$ | weaker/stronger than; metaimplication |
| <code>con-1</code> | \sqsubseteq | <code>con-2</code> | iff | $[\text{con-1}] \sqsubseteq [\text{con-2}]$ | less/more defined than |
| <code>con-1</code> | \Leftrightarrow | <code>con-2</code> | iff | $\{\text{con-1}\} = \{\text{con-2}\}$ | weakly equivalent |
| <code>con-1</code> | \equiv | <code>con-2</code> | iff | $[\text{con-1}] = [\text{con-2}]$ | strongly equivalent |

In the first case we also say that con-2 is *weaker than* con-1 and in the second that con-2 is *wider defined than* con-1 . The following rather obvious relations hold between metapredicates⁹⁸:

$\text{con-1} \equiv \text{con-2}$	is equivalent to	$(\text{con-1} \sqsubseteq \text{con-2} \text{ and } \text{con-2} \sqsubseteq \text{con-1})$
$\text{con-1} \Leftrightarrow \text{con-2}$	is equivalent to	$(\text{con-1} \Rightarrow \text{con-2} \text{ and } \text{con-2} \Rightarrow \text{con-1})$
$\text{con-1} \equiv \text{con-2}$	implies	$\text{con-1} \Leftrightarrow \text{con-2}$
$\text{con-1} \equiv \text{con-2}$	implies	$\text{con-1} \sqsubseteq \text{con-2}$
$\text{con-1} \Leftrightarrow \text{con-2}$	implies	$\text{con-1} \Rightarrow \text{con-2}$

It is important to understand the difference between three implications that belong to three different logical levels

1. **implies** : Condition x Condition \mapsto Condition — syntactic constructor,
2. \Rightarrow : Condition x Condition \mapsto {tt, ff} — metaimplication,
3. **implies** : {tt, ff} x {tt, ff} \mapsto {tt, ff} — MetaSoft implication

The first of them, given two conditions con-1 and con-2 constructs a third condition written as con-1 **implies** con-2 .

The second is used to describe the relationship between the two conditions.

The third belongs to the metalevel of **MetaSoft**, where we can talk about the properties of metaconditions, and about the relationships between them.

Despite substantial differences in the natures of these three implications, there is a certain relationship between them:

$$(\text{con-1} \text{ **implies** } \text{con-2}) \equiv \mathbf{TT} \text{ **implies** } \text{con-1} \Rightarrow \text{con-2}.$$

Indeed let $\text{sta}:\{\text{con-1}\}$, which means that $[\text{con-1}].\text{sta} = \text{vt}$. If now $[(\text{con-1} \text{ **implies** } \text{con-2})].\text{sta} = \text{vt}$ and $[\text{con-1}].\text{sta} = \text{vt}$, then $[\text{con-2}].\text{sta} = \text{vt}$ which means that $\text{sta}:\{\text{con-2}\}$. On the other hand:

$$\text{con-1} \Rightarrow \text{con-2} \text{ does not imply } (\text{con-1} \text{ **implies** } \text{con-2}) \equiv \mathbf{TT}.$$

Indeed, despite that the metaimplication $\sqrt[2]{x} > 4 \Rightarrow x > 3$ holds, the condition

$$\sqrt[2]{x} > 4 \text{ **implies** } x > 3$$

is undefined for $x < 0$.

It is important to note that using metaimplication, we can easily express the property of total-correctness of an instruction ins wtr a precondition pre and a postconditions post .

$$\text{pre} \Rightarrow \text{ins} @ \text{post} \quad \text{— total correctness}$$

We can also describe the concept of a *strong invariant* of an instruction:

$$\text{con} \Rightarrow \text{ins} @ \text{con} \quad \text{— strong invariant}$$

Strong invariants are used in correctness-proofs of total correctness of **while** loops. Now let us examine a few examples⁹⁹:

$$\begin{aligned} x > 0 \text{ **and** } \sqrt[2]{x} > 2 &\equiv x > 4 \\ \sqrt[2]{x} > 2 &\Leftrightarrow x > 4 \quad \text{but} \equiv \text{does not hold} \\ \sqrt[2]{x} < 2 &\sqsubseteq x < 4 \quad \text{but neither} \equiv \text{nor} \Leftrightarrow \text{holds} \end{aligned}$$

⁹⁸ It is worth noticing that on the ground of our non-classical calculus of conditions we have two concepts of satisfiability — *strong satisfiability* ($\text{con} \equiv \mathbf{TT}$) con is always true, and *weak satisfiability* ($\text{con} \sqsubseteq \mathbf{TT}$) con is never false. Readers interested in logics based on these concepts are referred to [58].

⁹⁹ We assume that the square root of a negative number is undefined.

$$\sqrt[2]{x} > 4 \Leftrightarrow x > 3 \quad \text{but neither } \Leftrightarrow \text{ nor } \sqsubseteq \text{ holds}$$

Metapredicates play an important role in our future techniques of the development of correct metaprograms. For instance (anticipating the investigations of Sec. 8.4.3), if in a correct metaprogram we replace:

- its pre- and postconditions, and its assertions by weakly equivalent ones,
- its boolean expressions by strongly equivalent ones,

then the new program is correct as well. The following lemmas are useful in program development (proofs and more investigations in [24]).

Lemma 8.4.2-1 *Relations \equiv and \Leftrightarrow are both equivalences, i.e., they are reflexive, symmetric, and transitive.*

■

Lemma 8.4.2-2 *Strong equivalence is a congruence wrt **and**, **or** and **not**, i.e., the replacement of a subcondition of a condition by a strongly equivalent one result a condition strongly equivalent to the initial one.* ■

Lemma 8.4.2-3 *Weak equivalence is a congruence wrt **and** and **or**.* ■

Weak equivalence is not a congruence wrt negation since

$$\text{con-1} \Leftrightarrow \text{con-2} \quad \text{does not imply} \quad \text{not con-1} \Leftrightarrow \text{not con-2}$$

For instance, although

$$\sqrt[2]{x} > 2 \Leftrightarrow x > 4$$

is satisfied, the metacondition

$$\sqrt[2]{x} \leq 2 \Leftrightarrow x \leq 4$$

is not, since for $x = -1$ the right-hand-side equation evaluates to **vt**, but on the left-hand side, we have an error.

Lemma 8.4.2-4 *The operators **and** and **or** are strongly associative, i.e.*

$$\begin{aligned} (\text{con-1} \text{ and } \text{con-2}) \text{ and } \text{con-3} &\equiv \text{con-1} \text{ and } (\text{con-2} \text{ and } \text{con-3}) \\ (\text{con-1} \text{ or } \text{con-2}) \text{ or } \text{con-3} &\equiv \text{con-1} \text{ or } (\text{con-2} \text{ or } \text{con-3}) \end{aligned}$$

Of course, they are also weakly associative since strong equivalence implies weak equivalence.

Lemma 8.4.2-5 *The operator **and** is strongly left-hand-side distributive wrt **or** and vice versa, i.e..*

$$\begin{aligned} \text{con-1} \text{ and } (\text{con-2} \text{ or } \text{con-3}) &\equiv \text{con-1} \text{ and } \text{con-2} \text{ or } (\text{con-1} \text{ and } \text{con-3}) \\ \text{con-1} \text{ or } (\text{con-2} \text{ and } \text{con-3}) &\equiv \text{con-1} \text{ or } \text{con-2} \text{ and } (\text{con-1} \text{ or } \text{con-3}) \end{aligned}$$

However, both operators are not strongly right-hand-side distributive. Indeed (not quite formally written):

$$\begin{aligned} (\text{vt} \text{ or } \text{ee}) \text{ and } \text{vf} = \text{vf} \quad \text{but} \quad (\text{vt} \text{ and } \text{vf}) \text{ or } (\text{ee} \text{ and } \text{vf}) = \text{ee} \\ (\text{vf} \text{ and } \text{ee}) \text{ or } \text{vt} = \text{vt} \quad \text{but} \quad (\text{vf} \text{ or } \text{vt}) \text{ and } (\text{ee} \text{ or } \text{vt}) = \text{ee} \end{aligned} \tag{8.4.2-1}$$

Lemma 8.4.2-6 *The operator **and** is weakly left-hand-side distributive wrt **or** i.e.*

$$(\text{con-1} \text{ or } \text{con-2}) \text{ and } \text{con-3} \Leftrightarrow (\text{con-1} \text{ and } \text{con-3}) \text{ or } (\text{con-2} \text{ and } \text{con-3})$$

However, **or** is not even weakly left-hand-side distributive wrt **and** which can be seen in (8.4.2-1).

Lemma 8.4.2-7 *The de Morgan's laws for **and** and **or** and for the negation of quantifiers are satisfied with strong equivalence.* ■

Lemma 8.4.2-8 *Conjunction is weakly commutative.*

$$\text{con-1} \text{ and } \text{con-2} \Leftrightarrow \text{con-2} \text{ and } \text{con-1} \quad \blacksquare$$

However, conjunctions are not strongly commutative, and the alternative is not even weakly commutative, since:

vt or ee = vt but ee or vt = ee

Lemma 8.4.2-9 If

con-1 \Leftrightarrow con-2

then

con-1 and con-2 \equiv con-1 ■

Besides the two-argument metapredicates, we also define three-argument metapredicates which will be used in the development of correct programs:

con-1 \equiv con-2 **whenever** con **means** con **and** con-1 \equiv con **and** con-2

con-1 \Leftrightarrow con-2 **whenever** con **means** con **and** con-1 \Leftrightarrow con **and** con-2

In both cases, we say that con constitutes a *logical context* or simply a *context* for the equivalence which it follows. We shall also say that the *equivalence* con-1 \equiv con-2 *is satisfied under the condition* con and analogously for a weak equivalence. The following metapredicates are satisfied:

$n > x^2 \equiv \sqrt[3]{n} > x$ **whenever** ($n \geq 0$ **and** $x \geq 0$)

$n > x^2 \Leftrightarrow \sqrt[3]{n} > x$ **whenever** $x \geq 0$

The context is usually a condition in whose range we want to replace one condition by another one.

All presented above considerations were published by myself in the decade 1980 in [22] and [26], and the development of these ideas towards three-valued deductive theories was investigated in a paper [58] written together with Beata Konikowska and Andrzej Tarlecki.

8.4.3 Metainstructions

Metainstructions describe the properties of instructions. At the moment we introduce just one metainstruction of the form

if dat **then** sin **fi limited-replicability in** con

which is satisfied iff

[{dat}] Ssi.[sin] has limited replicability in {con}.

For “limited replicability,” see Sec. 7.7.2.

8.4.4 Metaprograms

Metaprograms are propositions with the following syntax:

mpr : MetaProgram =

pre Condition :

Declaration ;

SpecInstruction

post Condition¹⁰⁰

Metaprograms express total correctness of programs (as defined in Sec. 7.7). The semantics of metaprograms is a function:

Smp : MetaProgram \mapsto {tt, ff}

defined as follows:

¹⁰⁰ Notice that in this syntax the keywords **pre** and **post** and the colon play the role of separators.

$$\text{Smp.}[\mathbf{pre} \text{ } prc : \text{dec} ; \text{sin } \mathbf{post} \text{ } poc] = \text{tt}$$

iff

$$\{prc\} \subseteq \text{Sde.}[\text{dec}] \bullet \text{Ssi.}[\text{sin}] \bullet \{poc\} \quad \text{or equivalently} \quad (8.4.3-1)$$

$$prc \Leftrightarrow \text{dec} ; \text{sin} @ \text{poc}$$

Notice that both, the precondition, and the postcondition may be arbitrary conditions, i.e. in particular algorithmic conditions.

If the denotation of a metaprogram is tt, then we say that this metaprogram is *totally correct* or simply *correct*.

It should be emphasized that our notion of correctness corresponds to so-called *total correctness with clean termination* as defined in [23]. In contrast to total correctness considered by other authors — such as, e.g., in [8], [44], [45] or [55], where programs never generate errors (which is not very realistic) — in our case errors may happen, but in correct programs, they do not.

We will say that under a condition con a data expression dae *evaluates cleanly*, if the following metaimplication is satisfied:

$$\text{con} \Leftrightarrow \text{dae} = \text{dae}$$

This metaimplication says that for every state that satisfies con, the evaluation of dae terminates, and does not generate error.

In the sequel of the book, “correctness” means “total correctness”, and whenever we want to talk about partial correctness, we will express it explicitly. In the latter case, we shall use the following syntax:

$$\mathbf{par-pre} \text{ } prc : \text{dec} ; \text{sin } \mathbf{par-post} \text{ } poc,$$

which means that

$$\{prc\} \bullet \text{Sde.}[\text{dec}] \bullet \text{Ssi.}[\text{sin}] \subseteq \{poc\}$$

In this case, there is no equivalent formulation in terms of metaconditions since we have nothing like a “left-sided” @ operation.

A few simple but useful lemmas may be formulated about the correctness of metaprograms.

Lemma 8.4.3-1 *If*

$$\mathbf{pre} \text{ } prc : \text{dec} ; \text{sin } \mathbf{post} \text{ } poc$$

is correct then in any execution of dec ; sin that starts with a state which satisfies prc:

1. *neither dec nor sin nor poc generates an error,*
2. *states in {prc} do not bind (refer to) identifiers that are (going to be) declared in dec,*
3. *all states are adequate for dec,*
4. *all assertions in sin are satisfied,*
5. *the terminal state does not carry an error.*

Statement 5. follows from the fact that if a state carries an error, then the evaluation of a condition in such a state generates an error.

Lemma 8.4.3-2 *If*

$$\mathbf{pre} \text{ } \text{con-pr} : \text{dec} ; \text{sin } \mathbf{post} \text{ } \text{con-po}$$

is correct and sin-1 has been created from sin by the removal of an arbitrary number of assertions or assertion-declarations, then the program

pre con-pr : dec ; sin-1 **post** con-po

is correct as well. ■

Lemma 8.4.3-3 *The replacement in a correct metaprogram its pre- or post-condition or a condition in an assertion by a weakly equivalent condition, does not violate the correctness of the program.* ■

The proof for pre- and post-conditions is obvious. For assertions the proof follows from the fact that if

$$\text{con-1} \Leftrightarrow \text{con-2} \text{ i.e. } \{\text{con-1}\} = \{\text{con-2}\}$$

then

$$[\text{con-1}].\text{sta} = \text{vt} \quad \text{iff} \quad [\text{con-2}].\text{sta} = \text{vt}$$

In particular, this lemma implies that on the level of conditions (but not of boolean expressions of the programming layer!) we can apply all the lemmas of Sec. 8.4.2 that concern weak equivalence.

Lemma 8.4.3-4 *The replacement in a correct metaprogram any boolean expression dae in an instruction or assertion by a boolean expression dae-1 that is stronger defined (i.e., such that $\text{dae} \sqsubseteq \text{dae-1}$) does not violate the correctness of the program.* ■

If the source program is correct, then none of its boolean expressions generates an error, and wherever dae is defined dae-1 is also defined and has the same value. ■

Now we assume a convention, which is similar as in “everyday mathematics”. Whenever we write a proposition of the form

pre con-pr : dec ; sin **post** con-po

we mean that this proposition is satisfied, i.e. the corresponding metaprogram is correct. This convention is analogous to writing $x > 2$, whenever we want to say the expression “ $x > 2$ ” is satisfied.

8.4.5 Jaco de Bakker paradox in Hoare’s logic

As was noticed by Jaco de Bakker (p. 108, Sec. 4 in [5]) and later commented by Krzysztof Apt in [4], on the ground of Hoare’s logic one can prove the formula:

pre true: a[a[2]] := 1 **post** a[a[2]] = 1

which for some arrays \mathbf{a} is not true. To show that consider an array:

$\mathbf{a} = [2, 2]$.

In that array

$\mathbf{a}[2] = 2$

hence the execution of the assignment

$\mathbf{a}[\mathbf{a}[2]] := 1$

means the execution of

$\mathbf{a}[2] := 1$

which means that the new array is $\mathbf{a} = [2, 1]$, and therefore $\mathbf{a}[\mathbf{a}[2]] = \mathbf{a}[1] = 2$.

Let us observe, however, that Hoare's problem results neither from having arrays in a language nor from the admission of expressions like $a[a[2]]$, but from a tacit assumption that whenever such an expression appears on the left-hand-side of an assignment, it should be treated as a variable. As a matter of fact, for many years, programmers used to talk about "subscripted variables" (in Algol 60 [71]) or about "indexed variables" (in Pascal [56]).

De Bakker's problem with Hoare's logic lies in the imperfect understanding of the meaning (semantics) of array variables¹⁰¹. In our language de Bakker's paradox does not appear since the instruction of the form:

```
a.(a.2) := 1
```

would be syntactically incorrect. In that place, we write

```
a := change-arr a at a.2 by 1 ee
```

or colloquially

```
a := change-arr a by a.2 := 1 ee
```

Now on the ground of constructions rules of Sec. 8.5.2 we can easily derive the following correct metaprogram:

```
pre a is arr-type number and a.1=2 and a.2=2
  a := change-arr a by a.2 := 1 ee
post a.1=2 and a.2=1
```

8.5 Constructing correct metaprograms

8.5.1 The role of declarations in the derivation of correct metaprograms

A derivation of a correct metaprogram of the form

```
pre prc: dec; sin post poc
```

can be split into the derivation of two metaprograms:

```
pre prc:
  dec;
  skip-i
post prc and poc-dec
```

```
pre prc and poc-dec:
  skip-d;
  sin
post poc
```

If *dec* is correct (no identifier is declared twice in it, see Sec. 8.4.1) then the derivation of the first program is trivial since the condition *poc-dec* should be simply a conjunction of declaration-oriented conditions (see Sec. 8.2.3) of the form:

```
ide is tex
ide is-type tex
ide proc-with ipd
ide fun-with fpd
```

¹⁰¹ In the denotational model described by M. Gordon in [53] array-variables or indexed-variables are admitted on the cost of a rather substantial complication of the model by distinguishing between left-values of expressions (locations) and right-values of expressions (values). In states values are assigned to locations and locations to identifiers.

The fact that the precondition `prc` appears in the postcondition of the first program follows from the simple fact that `prc` cannot include identifiers declared in the declaration. Indeed, if that were the case, the execution of `dec` would generate an error, because the satisfaction of a condition that includes `ide` implies that `ide` must have been declared “earlier”, and no identifier may be declared twice.

In any case, since the first of the two program-derivation steps is trivial, in the description of program construction rules we can concentrate on the case of programs with trivial declarations.

Note also that `poc-dec` may be implicit in `prc`. For instance, if condition `x>0` is satisfied in a given state, where `>` denotes a relation on integers, then `x` must be bound in that state to an integer value, and therefore

`x is integer`

must be satisfied as well. Consequently:

`x > 0 ⇒ x is integer`

and therefore

`(x is integer and x>0) ⇔ x>0.`

It means that ‘`x is integer`’ in a precondition, postcondition or assertion may be replaced by ‘`x>0`’ without violating the validity of a program.

Note that strong equivalence does not hold in this place since if `x` has not been declared as an integer, then `x is integer` is false, whereas `x>0` raises an error.

8.5.2 Basic rules

In the rules that follow, we tacitly assume that all metavariables such as `prc`, `dec`, `sin`, `poc`, etc. are bound by general quantifiers that stand before metaimplications denoted by the diagrams. Since we shall restrict our investigations to programs with trivial declarations, we shall omit writing `skip-d` in our metaprograms.

Let us start with simple rules that follow immediately as consequences of Rule 7.7.1-3, Rule 7.7.1-4, and Rule 7.7.1-5. We recall that our rules are, in fact, implications from the level of **MetaSoft** rather than proof rules, as understood in Hoare’s logics (cf. [55], [4], [5], [6]).

Rule 8.5.2-1 Strengthening precondition

$$\frac{\begin{array}{l} \text{pre } prc : \text{ sin } \text{ post } poc \\ prc-1 \Leftrightarrow prc \end{array}}{\text{pre } prc-1 : \text{ sin } \text{ post } poc}$$

Rule 8.5.2-2 Weakening postcondition

$$\frac{\begin{array}{l} \text{pre } prc : \text{ sin } \text{ post } poc \\ poc \Leftrightarrow poc-1 \end{array}}{\text{pre } prc : \text{ sin } \text{ post } poc-1}$$

Rule 8.5.2-3 Conjunction of conditions

<pre>pre prc-1 : sin post poc-1 pre prc-2 : sin post poc-2</pre>	<pre>pre (prc-1 and prc-2) : sin post (poc-1 and poc-2)</pre>
--	---

Now we can pass to our main rules concerning assignments and structured instructions.

Rule 8.5.2-5 Assignment¹⁰²

The following program is correct

```
pre (ide:=dae) @ poc :
  ide:=dae
post poc
```

This rule is, in fact, a tautology. Notice that if a state satisfies the precondition

$$(ide:=dae) @ poc$$

then, by the definition of @, the execution of

$$ide:=dae$$

must terminate without an error, and the terminal state must satisfy poc .

Despite its “tautological simplicity”, our rule is quite useful, and in fact necessary, in deriving correct programs. Consider the following example of a program-generation on the ground of this rule. First, we generate a tautological metaprogram:

```
pre x:=y+1 @ 2*x<10
  x := y+1
post 2*x<10
```

Such a program is, of course, quite trivial, but we can change it into a “less trivial” form by observing that the following weak equivalence is satisfied¹⁰³:

$$x:=y+1 @ 2*x<10 \Leftrightarrow 2*(y+1)<10$$

Basing on this equivalence, we can apply Lemma 8.4.3-3 to claim the correctness of the program

```
pre 2*(y+1)<10
  x := y+1
post 2*x<10
```

thus eliminating x from precondition¹⁰⁴.

¹⁰² In a Hoare’s style (cf. [5] p.6) the precondition of this rule is usually written in the form $poc[ide/dae]$ which denotes dae with all free occurrences of ide replaced by dae . With such a formulation we have to inductively define the replacement of an identifier by an expression. In our approach we “shift” this task to the stage of program construction.

¹⁰³ Here we in fact have a strong equivalence but to perform our transformation we need only a weak equivalence.

¹⁰⁴ Here we see an example of a replacement of x in $2*x<10$ by $y+1$.

Rule 8.5.2-6 Sequential composition of a program with an instruction¹⁰⁵

<div style="display: flex; align-items: center; justify-content: center;"> <div style="border-left: 1px solid black; height: 100px; margin-right: 5px;"></div> <div style="margin-left: 5px;"> <pre>pre prc-1: sin-1 post poc-1 pre prc-2: sin-2 post poc-2 poc-1 ⇔ prc-2</pre> <hr style="border: 0.5px solid black;"/> <pre>pre prc-1: sin-1; sin-2 post poc-2 pre prc-1: sin-1; asr poc-1 rsa; sin-2 post poc-2 pre prc-1: sin-1; asr prc-2 rsa; sin-2 post poc-2</pre> </div> </div>
--

Our rule follows directly from Rule 7.7.1-1. It is also to be pointed out that the vertical arrow goes only top-down since we have skipped existential quantification of `poc-1` and `pre-2` (cf. Rule 7.7.1-1), which in the case of conditions (syntactic entities), rather than sets of states, would not have much practical sense.

Another important observation is that to construct each of our target programs, we have to perform two program constructions and one proof of a metaimplication. As we are going to see, this is a typical situation. Such metaconditions are, on the one hand (usually) not very “sophisticated” mathematically, but on the other — may include quite a lot of variables. The first property makes them provable by automatic provers, and the second means that proving them “by hand” may be practically unfeasible. In other words, an automatic theorem prover should be an element of a programmer’s environment in each **Lingua-nV**.

Rule 8.5.2-7 Conditional branching if-then-else-fi

<div style="display: flex; align-items: center; justify-content: center;"> <div style="border-left: 1px solid black; height: 100px; margin-right: 5px;"></div> <div style="margin-left: 5px;"> <pre>pre (prc and dae) : sin-1 post poc pre (prc and not dae) : sin-2 post poc prc ⇔ (dae or (not dae))</pre> <hr style="border: 0.5px solid black;"/> <pre>pre prc : if dae then sin-1 else sin-2 fi post poc</pre> </div> </div>

This rule follows directly from Rule 7.7.1-2. In this case, the implication is two-directional since we do not need to construct any intermediate assertion. It is also worth noticing that the metacondition

$$\text{prc} \Leftrightarrow (\text{dae} \text{ or } (\text{not dae}))$$

means that whenever `prc` is satisfied, the data expression `dae` is either $\forall t$ or $\forall f$, i.e.

1. is defined (does not loop),
2. does not generate an error,
3. generates a boolean value.

Notice that in two-valued predicate calculus, this metaimplication would be a tautology. Consequently, when one uses Hoare’s logic of programs [55] which is based on the classical two-valued logic, this metaimplication is omitted, which leads to incorrect conclusions¹⁰⁶. To illustrate the problem consider the following program;

```
pre x ≥ 0 :
  if 1/x > 0 then x:=x else x:=-x fi
post x > 0
```

which aborts for $x=0$, although in Hoare’s logic it can be proved totally correct.

¹⁰⁵ In this rule declaration appears only in the first program since it must always appear at the beginning of a program.

¹⁰⁶ In the historical paper [55] of C.A.R Hoare a rule for conditional branching is not considered, but it is implicit in a paper by K. Apt [4]. Originally it is formulated there for partial correctness on p. 433, but is later followed by the following comment on p. 441: “It is clear that the only proof rule of Hoare which introduces a possibility of nontermination is the while rule; so to deal with total correctness that rule has to be changed”.

Rule 8.5.2-8 Loop while-do-od

```

(1) pre inv and dae : sin post inv
(2) asr dae rsa ; sin limited-replicability-in inv
(3) prc  $\Leftrightarrow$  inv
(4) inv  $\Leftrightarrow$  (dae or (not dae))
(5) inv and (not dae)  $\Leftrightarrow$  poc

```

pre prc : **while** dae **do** sin **od** **post** poc

This rule follows from Rule 7.7.2-5. To use it in the process of program construction, we have to:

1. construct a correct metaprogram (1) that includes the future instruction `sin` of the loop; this also requires inventing the invariant `inv` of the loop,
2. prove the limited replicability (4) of `asr dae rsa ; sin`; this requires inventing a well-founded set and a function that maps states into the elements of that set.
3. prove three metaimplications (3), (4), and (5); notice that (5) is absent in Hoare's logic,

The metaprogram (3) expresses the fact that the satisfaction of the invariant in conjunction with `dae` in the precondition guarantees clean termination of `ins`. Of course, in proving the halting property of `ins`, we may use Lemma 7.7.2-1 or another similar vehicle.

In the end of this section let us consider an example of a derivation of the following metaprogram, where `nnint` denotes a predefined type non-negative integers:

```

pre n,m is nnint and x=n and k=1:   — precondition prc
  while x≠0                               — data expression dae
  do
    k := k*m ;                             — the beginning of sin
    x := x-1                               — the end of sin
  od ;
post k=m^n                               — postcondition poc

```

It is implicit in the precondition that `n`, `m`, `x`, and `k` have been declared as numeric variables. We should also note that in our metaprogram, the values of `n` and `m` do not change during program execution. We shall call them, therefore, *constants*. Using these variables, we can describe the relationship between the initial values of `n` and `m`, and the terminal value of `k`.

To derive our metaprogram using Rule 8.5.2-4, we have in the first place to come up with a loop invariant. In this case, it is an easy job. We set it as

```
n,m, is nnint and k=m^(n-x)
```

Now we have to prove correct, or derive, metaprograms (1), and to prove (2) to (5):

```

(1) pre inv and dae : sin post inv
(2) asr dae rsa ; sin limited-replicability-in inv
(3) prc  $\Leftrightarrow$  inv
(4) inv  $\Leftrightarrow$  (dae or (not dae))
(5) inv and (not dae)  $\Leftrightarrow$  poc

```

In the general case, steps (1) and (2) are most difficult since, in (1), we have to “invent” an invariant of our loop, and in (2), we have to prove halting property which requires the choice of a well-founded set. In our example, however, both steps are fairly easy.

ad (1). The construction of the program

```
pre inv and x≠0: k:=k*m; x:=x-1 post inv
```

can be done by combining sequentially (Rule 8.5.2-6) two obviously correct programs

```
pre n,m, is nnint and k=m^(n-x) and x≠0:
```

```
  k:=k*m
```

```
post n,m, is nnint and k=m^(n-x+1)
```

```
post n,m, is nnint and k=m^(n-x+1):
```

```
  x:=x-1
```

```
post n,m, is nnint and k=m^(n-x)
```

ad (2). The termination of the loop is obvious because the loop runs for x from n to 0 . Formally we can apply Lemma 7.7.2-1, and as a well-formed set, assume the set of nonnegative integers with strict inequality “ $<$ ”.

ad (3). The metaimplication

```
n,m is nnint and x=n and k=1  $\Rightarrow$  n,m, is nnint and k=m^(n-x)
```

is obvious

ad (4). The metaimplication

```
n,m, is nnint and k=m^(n-x)  $\Rightarrow$  (x=0 or x≠0)
```

follows from the fact that the satisfaction of $k=m^{(n-x)}$ guarantees that x has been declared as a number.

(5) The satisfaction of $(n,m, \text{is nnint and } k=m^{(n-x)} \text{ and } x=0) \Rightarrow k=m^n$ is obvious.

8.5.3 Imperative procedures

So far, our construction rules were used to build programs with expected properties. To be more specific, given some expectation about future program expressed by a precondition prc and a postcondition poc , we had to build an instruction ins such that $\text{pre prc: ins post poc}$ is correct.

In the case of procedures our task is more complicated. Now, the given expectations are expressed by a metaprogram with a procedure call:

```
pre prc-call :
  call DoIt(val acp-v ref acp-r)
post poc-call
```

(8.5.3-1)

and our task consists in building a declaration

```
proc DoIt(val fop-v ref fop-r)
  body
end proc
```

(8.5.3-2)

such that the metaprogram with the call is correct. This means in turn that we have to build a metaprogram of the form

```
pre prc-body:
  body
post poc-body.
```

such that (8.5.3-1) will be satisfied. In this situation, let us try to figure out what relationship should occur between the specification of the call, i.e., prc-call and poc-call on the one hand, and the specification of the body, i.e., prc-body and poc-body , on the other.

First, observe that the precondition of the call, which describes the future programming context of the call, should guarantee that DoIt has been declared in the hosting program of the call. This requirement can be expressed by the metaimplication

```
prc-call  $\Rightarrow$  DoIt proc-with ipd.
```

The precondition of the call must also guarantee that passing actual parameters to formal parameters will be successful, i.e.:

$$\text{prc-call} \Leftrightarrow \mathbf{conformant}(\text{fop-v}, \text{fop-r}, \text{acp-v}, \text{acp-r})$$

The next question is, what should we expect about `prc-body` and `poc-body` to make (8.5.3-1) correct? Here we have another two metaimplications.

The first expresses the fact that whenever the precondition of the call is satisfied, the precondition of the body is satisfied, provided that the values of actual parameters are passed to formal parameters. Formally:

$$\text{prc-call} \Leftrightarrow \text{prc-body}[\text{fop-v}/\text{acp-v}, \text{fop-r}/\text{acp-r}]$$

Here $\text{prc-body}[\text{fop-v}/\text{acp-v}, \text{fop-r}/\text{acp-r}]$ denotes the condition `prc-body`, where the identifiers of formal parameters were replaced by the identifiers of actual parameters¹⁰⁷.

In turn, the satisfaction of the postcondition of the body must guarantee that, after the execution of the call, its postcondition is satisfied, provided that the values of formal reference-parameters were passed to actual reference-parameters:

$$\text{poc-body} \Leftrightarrow \text{poc-call}[\text{acp-r}/\text{fop-r}]$$

Putting all our arguments together, we have the following rule:

Rule 8.5.3-1 Building a declaration of an imperative procedure

(1)	<code>pre prc-body : body post poc-body</code>						
(2)	<code>prc-call \Leftrightarrow DoIt proc-with ipd</code>						
(3)	<code>prc-call \Leftrightarrow conformant(fop-v, fop-r, acp-v, acp-r)</code>						
(4)	<code>prc-call \Leftrightarrow prc-body[fop-v/acp-v, fop-r/acp-r]</code>						
(5)	<code>poc-body \Leftrightarrow poc-call[acp-r/fop-r]</code>						
<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="border-left: 1px solid black; padding-left: 5px;"><code>pre prc-call</code></td> <td></td> </tr> <tr> <td style="border-left: 1px solid black; padding-left: 5px;"><code> call DoIt(val acp-v ref acp-r)</code></td> <td></td> </tr> <tr> <td style="border-left: 1px solid black; padding-left: 5px;"><code>post poc-call</code></td> <td></td> </tr> </table>		<code>pre prc-call</code>		<code> call DoIt(val acp-v ref acp-r)</code>		<code>post poc-call</code>	
<code>pre prc-call</code>							
<code> call DoIt(val acp-v ref acp-r)</code>							
<code>post poc-call</code>							

Here a comment is needed since our rule seemingly does not include the case where `DoIt` may be recursive. Formally there is no such reference indeed, but if `DoIt` is called in `body` (whether directly or indirectly), then in proving (1) we shall have to cope with recursion. This issue is investigated in Sec. 8.5.4.

Now, let us analyse an example of an application of our rule. Let `nnint` is a predefined yokeless type of non-negative integers, and assume that our goal consists in building a declaration of a procedure `Power`, whose call should satisfy the following proposition:

<code>pre Power proc-with ipd and a,b,c \geq 0:</code>	– <code>prc-call</code>	
<code> call Power(val a,b ref c)</code>		(*)
<code>post c=b^a.</code>	– <code>poc-call</code>	

Notice that since `a` and `b` are value parameters, the postcondition

$$c=b^a$$

¹⁰⁷ A formal definition of this transformation requires a rather laborious construction by structural induction wrt the grammar of conditions, which I omit at that stage. It is worth noticing in this place that if we would let in **Lingua** actual parameters to be arbitrary data expressions, rather than variables, then the definition of the transformation would be even more complicated.

describes the value of c after the execution of the call where b and c have values assumed before the call. In contrast to the example which follows Rule 8.5.2-8, now we do not need to introduce any constants to describe the relationship between the input state and the output state of the call.

As a starting point in the development of future procedure declaration, we take the program developed in Sec. 8.5.2:

```
pre m, n ≥ 0 and x = n and k = 1:
  while x ≠ 0 do k := k * m; x := x - 1 od
post k = m^n
```

(**)

Variables n and m are obvious candidates for formal value parameters, k is a candidate for formal reference parameter, and x may play the role of a local variable. All four variables may be given the same type `nnint` (non-negative integer). The header of our procedure should be, therefore,

```
Power(val m, n nnint, ref k nnint)
```

Now, we have to modify (**) in such a way that its precondition `pre-call` metaimplies the precondition of the future body with a, b, c replaced by n, m, k , respectively. This modification leads to `pre-body` of the form

$$m, n, k \geq 0$$

which is due to the metaimplication

$$(\text{Power } \text{proc-with } \text{ipd } \text{and } m, n, k \geq 0) \Rightarrow (m, n, k \geq 0)$$

This metaimplication will guarantee the satisfaction of (4) of the rule. Since the **while** loop of the body must get a state where

$$m, n \geq 0 \text{ and } x = n \text{ and } k = 1$$

we transform (**) into the following program:

```
pre m, n, k ≥ 0 :
  let x be nnint tel;
  x := n; k := 1;
  asr n, m ≥ 0 and x = n and k = 1 rsa ;
  while x ≠ 0 do k := k * m; x := x - 1 od
post k = m^n
```

The correctness of this program — hence the satisfaction of (1) of our rule — follows from the correctness of (**), and the (obvious) correctness of

```
pre m, n, k ≥ 0 :
  let x be nnint tel;
  x := n; k := 1;
post m, n ≥ 0 and x = n and k = 1 rsa
```

and from Rule 8.5.2-6 about the composition of a program with an instruction. This fact leads us to the following declaration (where we omit assertion):

```
proc Power(val m, n nnint, ref k nnint)
  let x be nnint tel;
  x := n; k := 1;
  while x ≠ 0 do k := k * m; x := x - 1 od
endproc
```

It is easy to check that the remaining metaconditions (2), (3), and (5) of our rule are satisfied, which guarantees that (*) is a correct metaprogram.

8.5.4 Recursive procedures

Sec. 8.5.3 includes a construction rule which we can use in the process of building a procedure declaration adequate for a given procedure call. This process includes two steps:

1. first, we derive a correct metaprogram that includes a candidate for (a preliminary version of) the body of our future procedure,
2. next we modify this metaprogram in such a way that its components satisfy propositions (1) – (5) of Rule 8.5.3-1.

In the case of recursion, the situation is significantly different since in that case the correctness of a body

```
pre prc-body :
  body
post poc-body
```

depends on the correctness of the call

```
pre prc-call
  call DoIt (val acp-v ref acp-r)
post poc-call
```

and, of course, vice versa. In other words, now we cannot — as before — construct a correct call from a correct body. Instead, in one step, we have to prove that both correctness hypotheses are true.

Let us analyze such a case on a simple example. Let `RecPower` be the name of a future recursive procedure with the same functionality as `Power` of Sec. 8.5.3. This means that we expect the following program to be correct:

```
pre RecPower proc-with ipd and a,b,c ≥ 0 :
  call RecPower (val a,b ref c)
post c=a^b
```

(8.5.4-1)

Now, as a “candidate declaration” of our procedure we take

```
proc RecPower (val m,n nnint ref k nnint)
  let x be nnint tel;
  x:=n; k:=1;
  if x≠0
    then x:=x-1 ; call RecPower (val m,x ref k); k:=k*m
    else skip-i
  fi
end-proc
```

(8.5.4-2)

Observe that in this case, we do not derive our declaration from some previously derived programs, but “invent it from scratch”.

Our mathematical task is now to prove the correctness of the call expressed by (8.5.4-1), provided that the declaration of our procedure is described by (8.5.4-2). To do that we shall prove by induction on N the following program-correctness proposition:

```
pre RecPower proc-with ipd and a,b,c ≥ 0 and b=N:
  call RecPower (val a,b ref c)
post RecPower proc-with ipd and a,b,c ≥ 0 and b=N and c=a^b
```

(8.5.4-3)

Here N is a metaexpression that denotes a concrete number. Note that in this way, we construct two infinite families of conditions indexed by nonnegative integers.

In the first step of our proof we set $N=0$. In this case our hypothesis (8.5.4-3) becomes the following correctness statement

```
pre RecPower proc-with ipd and a,b,c ≥ 0 and b=0:
```

```

    call RecPower(val a, 0 ref c)
  post RecPower proc-with ipd and a,b,c ≥ 0 and b=0 and c=a^0

```

which can be “unfolded” to

```

pre RecPower proc-with ipd and a,b,c ≥ 0 and b=0:
  let x be nnint tel;
  x:=0; c:=1;
  if x≠0
    then x:=x-1 ; call RecPower(val a,x ref c); c:=c*a
    else skip-i
  fi
post RecPower proc-with ipd and a,b,c ≥ 0 and b=0 and c=a^b

```

which is equivalent to

```

pre RecPower proc-with ipd and a,b,c ≥ 0 and b=0:
  let x be nnint tel;
  x:=0; c:=1
post RecPower proc-with ipd and a,b,c ≥ 0 and b=0 and c=1

```

which is, of course, true.

Now assume, that our hypothesis is true for some $N \geq 0$ and consider the following inductive hypothesis which we already write in an “unfolded” form:

```

pre RecPower proc-with ipd and a,b,c ≥ 0 and b=N+1:
  let x be nnint tel;
  x:=N+1; c:=1;
  if x≠0
    then x:=x-1 ;
    asr RecPower proc-with ipd and a,b,c ≥ 0 and x=N rsa;
    call RecPower(val a,N ref c);
    asr RecPower proc-with ipd and a,b,c ≥ 0 and b=N and c=a^N rsa;
    c:=c*a;
    else skip-i
  fi
post c=a^(N+1)

```

The read part of this thesis corresponds to our inductive assumption expressed by means of two assertions that describe the properties of states before and after the execution of the recursive call.

Note that the halting property of our program (8.5.4-3) follows from the fact that the execution of the call terminates for $a=0$, and that if it terminates for $a=N$, then it terminates also for $a=N+1$. In our proof, we have implicitly applied Rule 7.7.2-3, where proposition (1) has been proved by induction on index i .

In the end, a personal note is in order. I am aware (I, Andrzej Blikle) of the fact the presented proof is only half formal. What is not quite formal are the steps where formal parameters are replaced by actual parameters. Although intuitively fairly clear, these steps are not justified by any formal rule, and they do not take into account the fact that the local initial state of an execution of a call inherits a declaration-time environment of the called procedure (Sec.6.3.4). In our simple example, this informality did not matter much, because no other procedures or user-defined types were referred to, but a formal rule should cover such cases. It is clear that some research is required in this case, and I would encourage the readers to undertake this challenge.

8.5.5 Functional procedures

Analogously to imperative procedures, correctness statements about functional procedures describe the properties of their calls. Let us start with an example of a functional procedure with the following declaration:

```

fun RecPowerFun (m,n)
  let k is nnint tel
  call RecPower(val m,n ref k)
  return 3*k+1
endfun

```

Here `RecPower` is the procedure analyzed in Sec. 8.5.4. There are two possible forms of correctness statements about a procedure. In our example, they are the following:

```

pre RecPowerFun fun-with fpd and a,b ≥ 0:
  RecPowerFun (a,b)
post-exp 3*(a^b)+1

```

or

```

pre RecPowerFun fun-with fpd and a,b ≥ 0:
  RecPowerFun (a,b)
post-yoke value > 1

```

The first statement expresses the relationship between the input values of actual parameters a , and b , and the value exported by the call. The second — expresses a property of the exported value. In the first case, we have a precondition and a *postexpression*, in the second — a precondition and a *postyoke*.

To define the semantics of both forms of correctness statements, we shall generalize these statements by setting an arbitrary data expression dae in the place of a procedure call. Recall that calls of functional procedures are expressions. This first form of a new correctness statement is

```

pre con
  dae
post-exp p-dae

```

This statement is satisfied if the following metaimplication is true:

$$\text{con} \Rightarrow \text{dae} = \text{p-dae}$$

Note that $\text{dae} = \text{p-dae}$ also means that in every state that satisfies con , the evaluation of both expressions will terminate cleanly. The second statement is of the form

```

pre con
  dae
post-yoke yok

```

and is satisfied if the following metaimplication is true:

$$\text{con} \Rightarrow \text{dae} \square \text{yok}$$

Here \square is a new operator that builds a condition from an expression and a yoke. The denotation of $\text{dae} \square \text{yok}$ is defined in the following way:

```

[dae □ yok].sta =
  is-error.sta      → error.sta
  Sde.[dae].sta = ? → ?
let
  val = Sde.[dae].sta
  val : Error      → val
let
  (com, yok-v) = val
  y-val        = Syoe.[yok].com
true          → y-val

```

Condition $\text{dae} \square \text{yok}$ is satisfied in a given state if the composite of the value of dae in that state satisfies the yoke yok .

Concerning the issue of proving correctness statements about functional procedures, if they involve procedures with non-trivial programs in their declarations, we should use rules for proving programs correct. Otherwise, we have to do with “usual” mathematical formulas, where no specific proof vehicles are needed.

8.5.6 Invariants versus assertions

From a philosophical viewpoint, invariants and assertions, as they have been defined in this book, are close to invariants in the sense of R. Floyd [47] and C.A.R Hoare [55]. Formally they are, however, not only quite different from each other but also belong to different linguistic categories¹⁰⁸.

Mathematically invariants are just conditions (Sec. 8.2) but “to be an invariant”, concerns a relationship between a condition and an instruction. We say that a *condition* `con` is a *partial resp. total invariant of an instruction* `ins` if it satisfies one of two metacondition:

$$\begin{aligned} \{\text{con}\} \bullet \text{Sin}.[\text{ins}] &\subseteq \{\text{con}\} && \textit{partial invariant} \\ \{\text{con}\} &\subseteq \text{Sin}.[\text{ins}] \bullet \{\text{con}\} && \textit{total invariant} \end{aligned}$$

Total invariants may be also defined by metaimplications:

$$\text{con} \Rightarrow \text{ins} @ \text{con}$$

whereas partial cannot, since in **Lingua-2V** we have not introduced a left-hand-side composition of a condition with an instruction.

Yet another concept is a *loop-invariant* for **while**, which appears in the rule 8.5.2-8:

<pre> there exists a condition inv (an invariant) such that : pre inv and dae: sin post inv pre inv: while dae do sin od post TT prc ⇔ inv inv ⇔ (dae or (not dae)) inv and (not dae) ⇔ poc </pre>	<pre> pre prc : while dae do sin od post poc </pre>
--	---

In this case `inv` is a loop-invariant in the instruction

```
while  dae  do  sin  od ,
```

if it satisfies all the metaconditions above the line.

In all three cases, *to be an invariant* is a property of a condition relativized in this or another way to an instruction.

The situation with assertions is different. In the first place, they are not conditions, but specinstructions built up of conditions. A specinstruction

```
asr  con  rsa
```

„behaves” as a filter which does not change a state if the condition `con` is satisfied, and which changes a state by writing an error into its error register in the opposite case.

Whereas invariants are used in program-correctness proofs, assertions are used when we transform correct metaprograms into (optimized) correct metaprograms.

Assertions describe local properties of programs expressed by the properties of states intermediate in program executions. The use of assertion in program-transformations bases on the observation that if a given metaprogram is correct, then its assertions must be satisfied in every execution of that program that starts from

¹⁰⁸ This section has been written as a reaction to some important remarks of Stefan Sokolowski.

a state that satisfies the precondition of the program. This observation allows us to decide which transformation rules may be applied to a given program¹⁰⁹.

Together with assertions, we have two derivative concepts that allow to decree the satisfaction of a given condition on a given range of an instruction:

```
asr con; sin rsa
```

```
off sin on.
```

These concepts have been defined as colloquialisms, and thus they belong neither to the level of concrete syntax and denotations (as assertions) nor to the meta-level (as conditions).

¹⁰⁹ In the examples of Sec. 0 assertions were applied only in transformations concerning register-identifiers. Time will show if they may have a larger scope of application.

8.6 Transformational programming

8.6.1 First example

In the previous section, we were dealing with rules for constructing correct metaprograms from correct components. An analogy in the automotive industry would be the construction of tools for assembly lines. In the present section, we shall consider rules that transform programs to “enrich” their functionality. In the following examples, we show the applications of rules introduced earlier as well as some new rules that are going to be formalized in Sec. 8.6.2. Let us start with an example of two obviously correct metaprograms.

```

pre x,n is nnint :
  x := 0;
  while (x+1)2 ≤ n
  do
    x := x+1
  od
post x = isrt(n)

pre x,n,m is nnint
  x := 0;
  while (x+1)*m ≤ n
  do
    x := x+1
  od
post x = iqt(n,m)

```

Each of these programs goes number-by-number through the set of nonnegative integers in seeking the expected result. Returning to our automotive metaphor, we may say that both programs are driven by the same while-engine:

```

P1: pre x,k is nnint:
  x := 0;
  while x+1 ≤ k
  do
    x := x+1
  od
post x = k

```

Now, we can use this universal engine to drive two different appliances: an integer square root `isrt(n)` or an integer quotient `iqt(n,m)`. In each of these cases, we change the functionality of a program but preserve its correctness. Let us show a simple universal method that can justify the correctness of the resulting metaprogram.

First observe that the correctness of P1 implies the correctness of P2.

```

P2: pre x,n is nnint :
  x := 0;
  asr x,n is nnint:
  while x+1 ≤ isrt(n)
  do
    x := x+1
  od
  rsa
post x = isrt(n)

```

Here and in the sequel, new or changed elements of a program will be marked in red. It is to be clarified that introducing an on-region (Sec. 8.3) of assertion

```
x,n is nnint
```

is not the result of an application of a general rule, but a step the soundness of which has to be proved, although in this case the proof is, of course, straightforward.

So far, our metaprogram looks a bit pointless since it refers to `isrt(n)` to compute it. We shall, therefore, eliminate that expression from the programming layer on the strength of a strong equivalence¹¹⁰:

$$x+1 \leq \text{isrt}(n) \equiv (x+1)^2 \leq n \textbf{ whenever } x, n \textbf{ is } \text{nnint}$$

and applying Lemma 8.4.3-3, which allows replacing a boolean expression by a strongly equivalent one. In our case, this equivalence holds only in the context specified by the **whenever** clause, and this context is assured within the on-range of our assertion.

As a result of the described transformation, we end up with a final program P3 where the assertion (now not necessary) has been removed.

```
P3: pre x, n is nnint :
    x := 0;
    while (x+1)2 ≤ n
    do
        x := x+1
    od
post x = isrt(n)
```

The instruction of the derived program does not refer to `isrt(n)`, and therefore may be said to be “more practical” than P2.

Nevertheless, it very slow. If we want to speed it up, we have to install a “faster engine” to drive it. Let us start from the construction of a universal searching engine that searches for its targets in logarithmic time.

Let `po2.k` denote a condition which is satisfied if `k` is a nonnegative power of 2, i.e., if there exists a nonnegative `m` such that:

$$k=2^m$$

Let `mag.k` (the magnitude of `k`) denotes a function with values in the set of powers of 2 such that

$$\text{mag.k} \leq k < 2*\text{mag.k}$$

For instance, `mag.11 = 3` since

$$2^3 \leq 11 < 2^4$$

Now, it is easy to prove the total correctness of the two following programs:

```
Q1: pre x, k, z is nnint :
    z := 1
    asr x, k, z is nnint and po2.z :
        while z ≤ k do z:=z*2 od
    rsa
post x, k, z is nnint and z = 2*mag.k
```

```
Q2: pre x, k, z is nnint and z = 2*mag.k:
    x := 0
    while z > 1
    do
        z := z/2;
        if x+z ≤ k then x:=x+z else skip-i fi
    od
```

¹¹⁰ This equivalence may be formally proved on the ground of the following definition of `isrt(n)`: it is the unique integer `k` such that $k^2 \leq n < (k+1)^2$.

```
post x = k and z = 1
```

The first program computes the successive powers of 2 until it reaches $2^{\text{mag.k}}$, and the second returns from $2^{\text{mag.k}}$ to 1 through successive powers 2^m and on its way summarises these powers of 2 that correspond to 1 in the binary representations of k . For instance, since

$$11 = 0 \cdot 16 + 1 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 1 \cdot 1$$

the second program, while given $2^{\text{mag.11}} = 16$, will perform the following summation

$$8 + 2 + 1 = 11.$$

In this way, the target value of k is reconstructed in logarithmic time, compared to a linear time of program P3. Now observe that the following proposition is true:

$$z \leq \text{mag.k} \equiv z \leq k \text{ whenever } x, n, z \text{ is nnint and po2.z}$$

Due to that equivalence, we can replace the boolean expression in **while** of the first program by strongly equivalent $z \leq k$. If we join both programs on the ground of Rule 8.5.2-3, we get our target program that finds the value of k in logarithmic time. In the same step, we move the initialization of x at the beginning of the program.

```
Q3: pre z, x, k is nnint :
    z := 1
    x := 0
    asr x, k, z is nnint and po2.z :
        while z ≤ k do z:=2*z od;
        while z > 1
            do
                z := z/2;
                if x+z ≤ k then x:=x+z fi
            od
    rsa
    post x = k and z = 1
```

Here and in the sequel

```
if dae then ins fi
```

means

```
if dae then ins else skip-i fi
```

If in this program we replace the expression k by the expression $\text{isrt}(n)$, then we have a program that computes $\text{isrt}(n)$ but refers to it. We eliminate this expression by using two strong conditional equivalences:

$$\begin{aligned} z \leq \text{isrt}(n) &\equiv z^2 \leq n && \text{whenever } z, n \text{ is nnint} \\ x+z \leq \text{isrt}(n) &\equiv (x+z)^2 \leq n && \text{whenever } z, x, n \text{ is nnint} \end{aligned}$$

In this way we get

```
Q4: pre z, x, n is nnint:
    z := 1;
    x := 0
    asr x, k, z is nnint and po2.z :
        while z2 ≤ n do z:=2*z od ;
        while z > 1
            do
                z := z/2;
                if (x+z)2 ≤ n then x:=x+z fi
```

```

    od
  rsa
  post x = isrt(n) and z = 1

```

This program computes $\text{isrt}(n)$, but can be optimized further by restricting the number of performed operations (time).

Let us start from the observation that in each run of the first loop, the program recalculates the value of z^2 , which is not optimal. Of course, in our simple case, it is not very relevant, but if we would repeat the multiplication of large matrices, the optimization may be worth the effort.

To optimize Q4 we introduce a new variable q , and we enrich our program in such a way that the condition $q=z^2$ is always satisfied. In this case, q is called a *register identifier* and z^2 — a *register expression*. This technique is discussed in detail in Sec. 8.6.3.

```

Q5: pre z, x, n, q is nnint:
  z := 1;
  x := 0;
  q := 1;
  asr z, x, n is nnint and po2.z and q = z2
  while q ≤ n
  do
    off z:=2*z; q:=4*q on
  od
  while z > 1
  do
    off z:=z/2; q:=q/4 on
    if x2+2*x*z+q ≤ n then x:=x+z fi
  od
  rsa
  post x=isrt(n) and z = 1 and q=z2

```

Notice the double-use of **off-on** is necessary since each time when the first assignment destroys the satisfaction of $q=z^2$, the second recovers it. Now we proceed to further transformations:

1. we use the equivalence $z>1 \equiv q>1$ **whenever** ($z>0$ and $q=z^2$) to modify boolean expression in the second loop,
2. we introduce two new variables y and p with the conditions $y=n-x^2$ and $p = x*z$,
3. we use the equivalence

$$x^2 + 2*x*z + q \leq n \equiv 2*p+q \leq y \text{ whenever } (y=n-x^2 \text{ and } p=x*z)$$

Using the corresponding transformations, we get the following program

```

Q6: dec let be number
  pre z, x, n, q, y, p is nnint:
  z := 1;
  x := 0;
  q := 1;
  asr z, x, n is nnint and q = z2 :
  while q ≤ n
  do
    off z:=2*z; q:=4*q on
  od
  y := n;
  p := 0;
  asr y=n-x2 and p = x*z :

```

```

    while q > 1
    do
      off z:=z/2; q:=q/4; p:=p/2; on
      if 2*p+q ≤ y then x:=x+z; p:=p+q; y:=y-2p-q fi
    od
  rsa
  rsa
  post x=isrt(n) and z=1 and q=z2 and y=n-x2 and p=x*z

```

Contrary to the former introduction of a new variable which was clearly justified, now it not quite clear why p and y have been introduced. The answer to this question follows from a well-known truth that in programming, like in playing chess, we sometimes have to predict a few moves in advance. These moves will be shown a little later.

In the next transformation, we prepare our program for the removal of variable z . For that sake, we perform the following changes:

1. we apply the equivalence $q=z^2 \Leftrightarrow \text{isrt}(q)=z$ **whenever** $z>0$ to change the assertion,
2. we use the condition $\text{isrt}(q)=z$ to replace z by $\text{isrt}(q)$ everywhere except the left-hand side of the assignment,
3. we make obvious changes based on the equality $z=1$.

The resulting program is the following:

```

Q7: pre z, x, n, q, y, p is nnint:
  z := 1;
  x := 0;
  q := 1;
  asr z, x, n is nnint and isrt(q)=z :
    while q ≤ n
    do
      off z:=2*isrt(q); q:=4*q on
    od
  y := n;
  p := 0;
  asr y = n-x2 and p = x*isrt(q) :
    while q > 1
    do
      off z:=isrt(q)/2; q:=q/4; p:=p/2 on
      if 2*p+q ≤ y then x:=x+isrt(q); p:=p+q; y:=y-2p-q fi
    od
  rsa
  rsa
  post x=isrt(n) and z=1 and q=1 and p=x and y=n-x2

```

Now notice that in Q7 the variable z does not appear neither in boolean expressions nor on the right-hand sides of assignment that do not change z . Since we do not care about the terminal value of z , we can remove that variable from our program together with the corresponding assignment (general rule will be described in Sec. 8.6.2). In this way we get:

```

Q8: pre x, n, q, y, p is nnint :
  q := 1;
  x := 0;
  asr x, n is nnint :
    while q ≤ n
    do

```

```

        q:=4*q
    od
y := n;
p := 0;
asr y = n-x2 and p = x*isrt(q) :
    while q > 1
    do
        off q:=q/4; p:=p/2 on
        if 2*p+q≤y then x:=x+isrt(q); p:=p+q; y:=y-2p-q fi
    od
rsa
rsa
post x=isrt(n) and q=1 and p=x and y=n-x2

```

Now we use the equivalence

$$x=\text{isrt}(n) \equiv p=\text{isrt}(n) \text{ whenever } p=x$$

to modify the postcondition which makes variable x not necessary anymore. Therefore, we can remove it with all expressions, and assertions, where it appears.

```

Q9: pre n, q, y, p is nnint:
    q := 1;
    while q ≤ n do q:=4*q od
    y := n;
    p := 0;
    while q > 1
    do
        if 2*p+q≤y then p:=p+q; y:=y-2p-q fi
    od
post p=isrt(n) and q=1

```

In the last step we replace the instruction

```
p:=p/2; if 2*p+q≤y then p:=p+q; y:=y-2p-q else x:=x fi
```

by an equivalent instruction

```
if p+q≤y then p:=p/2+q; y:=y-p-q else p:=p/2 fi
```

As a result, we get the final version of our program:

```

Q10: pre n, q, y, p is nnint :
    q := 1;
    while q ≤ n do q:=4*q od
    y := n;
    p := 0;
    while q > 1
    do
        q:=q/4;
        if p+q≤y
        then p:=p/2+q; y:=y-p-q else p:=p/2
        fi
    od
post p = isrt(n)

```

This program had been written by a well-known Norwegian computer-scientist Ole-Johan Dahl in 1970. Its value can be seen in the fact that the arithmetic operations used in the program are easily implementable in a binary arithmetic.

I do not know in what way Dahl has built this program but we may suppose that he performed an optimisation similar to ours, although without formalised rules.

At the end of this section, one pragmatic remark. Programmers who develop hundreds of thousands or millions of lines of code will probably regard the discussed example with a certain scepticism. Indeed, the volume of our program is not very impressive, and the shown optimization is not very irrelevant for the majority of applications. If, however, we build microprograms that are implemented in hardware and executed hundreds of millions of times by hundreds of millions of devices, then its correctness as well as time- and space-consumption may be worth an effort. Our example also shows a specific general method — although not universal — of building programs in three steps:

1. writing a program-engine that searches through a specific set of data,
2. installing an application on that engine which implements the expected functionality,
3. optimizing the program.

As we are going to see in Sec. 8.6.2, program optimization may also be used in changing the types of data elaborated by a program.

8.6.2 Changing data-types

Another application of register-identifier technique may serve in the replacement of one data-type by another one. In this section we show how to transform program Q10 from Sec. 8.6.1 into a program that operates on binary representations of numbers. Let **Binary** be the set of binary representations of integers, i.e. a set of zero-one tuples starting from 1.

$$\text{bin} : \text{Binary} = \{(1)\} \odot \{(0), (1)\}^*$$

On this set we define a few functions and relations:

$$\begin{array}{ll} \text{sl} : \text{Binary} \mapsto \text{Binary} & \text{shift left} \\ \text{sl.bin} = & \\ \text{bin} = (0) & \rightarrow 0 \\ \text{true} & \rightarrow \text{bin} \odot (0) \end{array}$$

$$\begin{array}{ll} \text{sr} : \text{Binary} \mapsto \text{Binary} & \text{shift right} \\ \text{sr.bin} = & \\ \text{bin} = (0) & \rightarrow 0 \\ \text{true} & \rightarrow \text{pop.bin} \end{array}$$

$$\begin{array}{ll} + : \text{Binary} \mapsto \text{Binary} & \text{addition} \\ - : \text{Binary} \mapsto \text{Binary} & \text{subtraction} \\ < : \text{Binary} \mapsto \{\text{tt}, \text{ff}\} & \text{earlier} \\ \leq : \text{Binary} \mapsto \{\text{tt}, \text{ff}\} & \text{earlier or equal} \end{array}$$

The addition and the subtraction of tuples are denoted by the same symbols as for numbers and we assume that they are defined in such a way that the equations (5) and (6) below are satisfied. The orderings are lexicographic and again correspond to their numeric counterparts.

$$\begin{array}{ll} \text{b2n} : \text{Binary} \mapsto \text{Number} & \text{binary to number; conversion function} \\ \text{n2b} : \text{Number} \mapsto \text{Binary} & \text{number to binary; conversion function} \end{array}$$

All these functions and relations are defined in such a way that they satisfy the following equations:

$$\begin{array}{ll} (1) \text{ b2n}.\text{(n2b.lic)} & = \text{num} \\ (2) \text{ n2b}.\text{(b2n.bin)} & = \text{bin} \\ (3) \text{ n2b}.\text{(num*2)} & = \text{sl}.\text{(n2b.num)} \end{array} \quad \text{where num} : \text{Number}$$

- (4) $n2b.(num/2) = sr.(n2b.num)$ where „/” denotes the integer part of division
 (5) $n2b.(num1 + num2) = n2b.int1 + n2b.num2$
 (6) $n2b.(num1 - num2) = n2b.num1 - n2b.num2$
 (7) $n2b.num1 < n2b.num2$ iff $num1 < num2$
 (8) $n2b.num1 \leq n2b.num2$ iff $num1 \leq num2$

Now we transform program Q10 by introducing to it three new variables and three corresponding register-conditions:

```
Q = n2b(q)
Y = n2b(y)
P = n2b(p)
```

At the same time we introduce a new type `binary` into our language. We introduce the assertions into it and we shift all initialisations to the beginning of our next program:

```
Q11: pre n, q, y, p is nnint and Q, Y, P is binary and  $n \geq 1$ 
  q := 1; Q := (1);
  y := n; Y := n2b(n);
  p := 0; P := (0);
  asr Q = n2b(q) and Y = n2b(y) and P = n2b(p) :
    while q ≤ n
    do
      off q:=4*q ; Q = sl(sl(Q)) on
    od
    while q > 1
    do
      off q:=q/4; p:=p/2;
      Q:=sr(sr(Q)); P:=sr(P); on
      if p+q≤y
      then off p:=p/2+q; y:=y-2p-q;
      P:=sr(P)+Q; Y:=Y-sl(P)-Q on
      else off p:=p/2; P:=sr(P) on
      fi
    od
  asr
  post p = isrt(n) and q = 1
```

Now we use four conditional equivalences in order to replace boolean numeric expressions by boolean binary ones:

$q \leq n$	$\equiv Q \leq n2b(n)$	whenever $Q=n2b(q)$
$q > 1$	$\equiv (1) < Q$	whenever $Q=n2b(q)$
$p+q \leq y$	$\equiv P+Q \leq Y$	whenever $Q=n2b(q)$ and $Y=n2b(y)$ and $P=n2b(p)$
$p=isrt(n)$	$\equiv P=n2b(isrt(n))$	whenever $P=isrt(p)$

Next we remove from our program all numeric variables except `n` with the corresponding assignments and the `on`-clause. Since the `on`-range reaches the end of the program, we can modify the postcondition in an appropriate way.

```
Q12: pre  $n \geq 1$  and Q, Y, P is binary
  Q := (1);
  Y := n2b(n);
  P := (0);
  while Q ≤ N do Q = sl(sl(Q)) od;
  while (1) < Q
  do
    Q:=sr(sr(Q)); P:=sr(P)
```

```

    if P+Q≤Y
      then P:=sr(P)+Q; Y:=Y-sl(P)-Q
      else P:=sr(P)
    fi
  od
post P = n2b(isrt(n)) and Q = (1)

```

8.6.3 Adding a register identifier

This section is devoted to a transformation of a metaprogram by adding to it a new identifier *ide-r* which satisfies an assertion of the form:

$$\text{ide-r} = \text{dae-r}. \quad (*)$$

Such transformation was applied in Sec. 8.6.1 in passing from Q4 to Q5 and in Sec. 8.6.2 in passing from Q10 to Q11.

An identifier *ide-r* that satisfies the condition $\text{ide-r}=\text{dae-r}$ on a certain range is called a *register-identifier* or just a *register*; the expression *dae-r* is called a *register-expression* and the condition $\text{ide-r}=\text{dae-r}$ — *register-condition*.

Let us start from an obvious generalization of the meaning of @ (Sec. 8.2.4) which now will compose instructions not only with conditions but also with data expressions:

$$\text{Sde.}[\text{sin} \ @ \ \text{dae}] = \text{Ssi.}[\text{sin}] \bullet \text{Sde.}[\text{dae}]$$

Now, let us consider a metaprogram that we assume to be correct:

```

P:pre prc
  ins-h; head (possibly empty)
  asr con rsa ;
  asr con : ins ; rsa
  ins-t tail (possibly empty)
post poc

```

Let *ide-r* be an identifier which does not appear in P, and let *dae-r* be a data expression such that

```
pre con: ide-r := dae-r post TT
```

which simply means that *con* guarantees the execution of $\text{ide-r} := \text{dae-r}$ without an error or looping. Under these assumptions a transformation that enriches P by introducing *ide-r* with a register-condition $\text{ide-r}=\text{dae-r}$ yields a program:

```

Q:pre prc and ide-r is tex
  ins-h ;
  ide-r := dae-r ;
  asr con and ide-r=dae-r :
    $(ins, ide-r=dae-r) enriched instruction (see below)
  rsa ;
  ins-t
post poc

```

where $\$(\text{ins}, \text{ide-r}=\text{dae-r})$ denotes such an enrichment of *ins* which makes Q correct, provided that P was correct. The assertion *asr con rsa* has been dropped from Q (although we could have left it there), since it only served to guarantee, that in its location the value of *dae-r* was defined.

The syntactic operation \$ is defined by structural induction, wrt the structure of *ins*. Let us start from *ins* which is an assignment

```
ide := dae
```

where obviously ide is different from $ide-r$, since we have assumed that $ide-r$ does not appear in P .

If ide does not appear in $dae-r$, then the execution of this assignment does not cause any change in the value of $dae-r$, and therefore we do not need to add any actualization.

If, however, this is not the case, then directly after $ide:=dae$, we have to add an assignment which recovers the satisfaction of the condition $ide-r=dae-r$. In such a case

$$\$(ide:=dae, ide-r=dae-r) = \text{off } ide:=dae; ide-r:=dae-r \text{ on}$$

where equality sign '=' denotes the equality of syntactic objects. An off-clause is necessary here since ide appears in $dae-r$. Consequently, the alteration of the value of ide may cause the alteration of the value of $dae-r$ and the falsification of our condition. In the case of the transformation of Q4 to Q5 with a register condition $q=z^2$ this has led to the enrichment of

$$\text{asr } q=z^2 \text{ rsa ; } z:=2*z$$

into:

$$\text{asr } q=z^2 \text{ rsa ; off } z:=2*z ; q:=z^2 \text{ on}$$

The assertion has been left in the resulting instruction since we shall need it a little later. Now, our instruction may be changed into an equivalent one (note the inverse order of assignments):

$$\text{asr } q=z^2 \text{ rsa ; off } q:=((z:=2*z) @ z^2) ; z:=2*z \text{ on}$$

In this instruction, we can eliminate $@$, by transforming the expression $(z:=2*z) @ z^2$ to a standard form:

$$\text{asr } q=z^2 \text{ rsa ; off } q:=4*z^2 ; z:=2*z \text{ on}$$

Now, since the assertion $q=z^2$ holds "just before" the assignment $q:=z^2$, we can replace our instruction by:

$$\text{asr } q=z^2 \text{ rsa ; off } q:=4*q ; z:=2*z \text{ on}$$

which makes the modification of q independent of z , and therefore — in our example — allows for the elimination of z from the program. In the general case, these transformations are as follows. First the instruction

$$\text{off } ide:=dae ; ide-r:=dae-r \text{ on}$$

is replaced by an equivalent one

$$\text{off } ide-r:=((ide:=dae) @ dae-r) ; ide:=dae \text{ on}$$

Further on, the expression $((ide:=dae) @ dae-r)$ is transformed to a standard form, and then we try to change it in such a way that the identifier ide can be eliminated due to the register-condition $ide-r=dae-r$. This action completes the transformation.

The second "atomic" case to be investigated is a procedure call:

$$\text{call } ide(\text{val } acp-v \text{ ref } acp-r)$$

Let us assume that our procedure call appears in the program in the same context as the assignment in the former case. We again have two subcases to be considered.

If none of the actual referential parameters appears in $dae-r$, then we keep the instruction unchanged. In the opposite case, we replace it with the instruction

$$\text{off call } ide(\text{ref } acp-r \text{ val } acp-v); ide-r:=dae-r \text{ on.}$$

This completes the first step of structured instruction. The remaining steps are rather obvious:

$$\begin{aligned} &\$((ide-1 ; ide-2), ide-r=dae-r) = \\ &\quad \$(ide-1, ide-r=dae-r) ; \$(ide-2, ide-r=dae-r) \end{aligned}$$

```
$(if dae-b then ins-1 else ins-2 fi, ide-r=dae-r) =
  if dae-b then $(ins-1, ide-r=dae-r) else $(ins-1, ide-r=dae-r) fi
```

```
$(while dae-b do ins od, ide-r=dae-r) =
  while dae-b do $(ins, ide-r=dae-r) od
```

In short, after each assignment or a procedure call that changes the value of a register condition, we add a recovering assignment. The generalization of $\$$ on specinstruction is rather evident.

In the end, let us point out a methodological difference between $\@$ and $\$$. The former is a character in the syntax of **Lingua-2V**, and on the denotational side corresponds to a sequential composition of an instruction denotation with a data-expression denotation. Therefore:

$$\mathbf{Sde}.[ins \ @ \ dae] = \mathbf{Sin}.[ins] \bullet \mathbf{Sde}.[dae]$$

In turn, $\$$ is a constructor of syntaxes (from the level of **MetaSoft**)

$$\mathbf{\$} : \text{Instruction} \times \text{RegisterCondition} \mapsto \text{Instruction}$$

where

$$\text{RegisterCondition} = \text{Identifier} = \text{DatExp}^{111}$$

¹¹¹ Notice that the first sign of equality belongs to MetaSoft and denotes the equality of formal languages, whereas the second – typed in Courier New – is a character in the syntax of Lingua.

9 RELATIONAL DATABASES INTUITIVELY

9.1 Preliminary remarks

Section 10 is devoted to an extension of **Lingua-2** by selected database tools of SQL (Structured Query Language). Since I don't expect the reader to be familiar with SQL, the present section contains an informal description of these SQL-mechanisms that will be given denotational definitions in Sec. 10. Some notions that I introduce below do not appear in standard SQL manuals, and therefore they will be labeled by "MON" which stands for "my own notion".

This section refers to several sources since one manual is usually not enough to determine the meaning of an SQL mechanism. The book of Lech Banachowski [9] contains a model of *Relational Databases* and a nice description of SQL standard, but some issues are missing (e.g., three-valued predicates), and some others are only sketched. On the other end of the scale of clarity is a thick volume of Paul DuBois [46]. I quote some descriptions from that book just to show the scale of problems that one has to tackle in building a denotational model for SQL. Between these two extremes, but certainly closer to DuBois, are four other books [48], [54], [66], and [72].

Since all mentioned books were published some time ago, some mechanisms described there may look today differently. However, it doesn't seem too much of a problem, since in any case, all our SQL-constructions must be defined independently. Of course, we should make them as close as possible to SQL standard, and, of course, applicable to SQL databases created by existing applications.

Lingua-SQL, whose draft description is given in Sec. 10, may be regarded as a sort of API (Application Programming Interfaces) or a CLI (Call Level Interfaces)¹¹². APIs have been created for such programming languages as C, PHP, Perl, Python, and CLIs — for ANSI, C, C#, VB.NET, Java, Pascal, and Fortran¹¹³. In each of these cases, a language is equipped with mechanisms allowing to run functionalities of an independently constructed SQL engine. In the case of **Lingua-SQL**, the situation is different. Our language, if ever implemented, must base on our own SQL engine "equipped" with a denotational model. Such an approach is necessary if we want to provide a credible denotational model for **Lingua-SQL**.

9.2 Simple data

Only one data-type — the type of tables (Sec. 9.3) — appears explicitly in SQL-manuals mentioned above. Several other types appear only implicitly. They include *simple data* (MON) that appear in the fields of database tables and *structured types* (MON) such as rows and columns of databases and the databases themselves.

Simple data constitute probably one of the least standardised areas of SQL. The sorts and the types of data differ not only between different applications but also between different implementations of the same application.

In the present section, I base mainly¹¹⁴ on [72], whose authors declare the compatibility with the standard ANSI SQL-2011¹¹⁵. The SQL syntax is printed in Arial Narrow.

¹¹² CLI refers to the standard ANSI SQL (see [72] p. 359)

¹¹³ Access has not been mentioned on these lists since it is available only together with Microsoft Basic Access.

¹¹⁴ „Mainly” but not „totally” since this manual also contains gaps.

¹¹⁵ ANSI is an acronym of American National Standard Institute, and SQL-2011 is a standard accepted by ANSI in December 2011.

Database-tables can carry four sorts of data which, except booleans, split into several types:

- **Numbers** split into three subsorts: *total numbers*, *decimal numbers*, and *floating-point numbers*. Each of them splits again into several types differing with each other on the range of values (in our approach can be described by yokes), e.g., INTEGER, SMALLINT, BIGINT or DECIMAL(p, s), where p (precision) denotes the maximal number of digits and s (scale) — the maximal number of digits after the decimal point.
- **Logical values** are handled as in the three-valued predicate calculus of Kleene (Sec. 2.9), and in [72] they are denoted by TRUE, FALSE, and NULL whereas in [46] by 0, 1, and NULL. Sometimes, e.g., in [54], instead of NULL we have UNKNOWN.
- **Strings** are in principle words in our sense, but, similarly to numbers, they are split into types according to a maximal accepted number of characters. For instance, CHARACTER(n) is the type of words of the length n. The type of a string with varying length limited to n is called in [72] CHARACTER VARYING(n), and the type of a string of an unlimited length (whatever it means) is called BLOB. There exist also binary strings, and text-strings called TEXT.
- **Times** are tuples of three types: DATE — (year, month, day), TIME — (hour, minute, second), DAY-TIME — (year, month, day, hour, minute, second).

Although this is nowhere explicitly said, one may guess (cf. [72]) that all sorts of data contain NULL that plays the role of an abstract error. The majority of constructors, except boolean constructors, seem to be transparent for that error.

The constructors of simple data may be split into five following groups¹¹⁶:

1. Arithmetic operations: +, −, *, /.
2. String operations: CONCAT, UPPER, LOWER, SUBSTR, LENGTH.
3. Time operations: GETDATE, DAYNAME, DAYOFMONTH,
4. Basic predicates: =, <>, <, <=, >, >=, IS NULL, BETWEEN, LIKE.
5. Logical connectives: NOT, OR, AND.

The first group apparently seems quite obvious. It turns out, however, that this is the case only in typical situations: $2+3=5$, but if we try to add a number to a string (which is possible!), or to add two numbers whose sum exceeds the maximal allowed value, then the expected result is not clear. The source [72] does not comment on such cases at all, and in [46] p. 786, we can read the following¹¹⁷:

If we do not provide (...) correct values to functions, we should not expect reasonable results.

In another place of the same manual (p. 754) we read:

(...) expressions that contain big numbers may exceed the maximal range of 64-bits computations in which case they return unpredictable values (my emphasis).

It is to be pointed out that in the definitions of arithmetic operations, NULL does not appear, although it could be used as an abstract error. In this place, the worst possible solution has been chosen: instead of an error message, we have an “unpredictable result” which means that the computation does not abort, but generates a false result without warning the user.

Especially many unclarities are associated with default rules for type-conversion. For instance ([46] p. 753) the following rule concerns the addition operation in the context of words as arguments:

¹¹⁶ The descriptions of 1 to 4 are from [72] (pp. 129 and 180) and of 5 and 6 from [54] (pp. 191 and 201). The terminology is mine.

¹¹⁷ My own translation from a Polish version of the book.

... '+' is not an operator for the concatenation of texts, as it is the case in some programming languages. Instead, before the performance of the operation, textual strings are converted into numbers. Strings that do not look like numbers (my emphasis) are converted to 0.

This rule has been illustrated with the following examples:

'43bc' + '21d' = 64

'abc' + 'def' = 0

It has not been explained, if, e.g., '43ab2c' "looks like a number", and if it does, is it converted to 43 or 432? It has not been explained either, whether these rules apply to other arithmetic operations.

Fortunately [72] treats conversion a little more seriously — although still informally — introducing four types of conversions:

1. strings to numbers,
2. numbers to strings,
3. strings to dates and times,
4. dates, and times to strings.

String-operators offer fewer ambiguities but still are defined only for typical situations. For instance, I did not find information about what happens if the concatenation of two strings exceeds an accepted length.

Time-operators offer further examples of inconsistencies between different SQL-applications that concern both the syntax and the types of operators. We not further analyse this problem since the involved operators are easy to formalise.

Predicates are typologically ambiguous since, in the majority of cases, they apply to all four sorts of data. E.g., the operators '=' and BETWEEN may be used for numbers and strings and probably also for dates. Their definitions are rather vague. E.g., in [72] p. 130, we can read:

If in a query, we use the (=) operator, the compared values must be identical, and in the opposite case, the condition is not satisfied.

It has not been explained in "not satisfied" means "false" or "not true". E.g. should we regard the value of the boolean expression $12 = abc$ as false or undefined?

The operator BETWEEN takes three arguments and checks if the first is between the second and the third in some default ordering.

The operator LIKE takes two string-arguments and checks if the first coincides with the pattern described by the second. Patterns are described using letters and digits and two special symbols:

% — an arbitrary string of characters (possibly empty)

_ — an arbitrary character

The only source where I found complete definitions of logical operators in [54], where a table-definition is given on page 191. In our notation, this table would be as in Fig. 9.2-1.

OR	tt	ff	ee	AND	tt	ff	ee	NOT	
	tt	tt	tt		tt	ff	ee		tt
	ff	tt	ff		ff	ff	ff		ff
	ee	tt	ee		ee	ff	ee		ee

Fig. 9.2-1 Boolean operators in SQL

Despite the existence of the NOT operator, special negated versions are introduced for all predicates, e.g., NOT NULL and NOT BETWEEN.

In the case of all non-boolean operators, we have a situation typical for software-manuals. Within the area of standard ranges of arguments, everything is clear. If, however, we go beyond that, we can hardly predict what happens. With a high degree of certainty, we may expect that in each implementation, we shall encounter a different surprise.

One more remark at the end. Simple data may be assigned in SQL to table fields only but not to variables.

9.3 Creating tables

An important SQL-concept is a *table*. On the ground of our denotational model, tables are close to¹¹⁸ one-dimensional arrays of records that carry simple data. In SQL manuals, records included in tables are called *rows*, the attributes of these records — *column-names*, and the intersections of rows and columns — *table fields*.

Tables in SQL — and precisely speaking the corresponding values (Sec. 4.3.6) — are (probably?) the only sort of SQL data that may be assigned to variables. In the sequel, variables carrying tables are called *table-variables* (MON). To declare a table variable, we use operator CREATE TABLE, which to a variable identifier assigns a *table type* and (we can guess) some sort of an *empty table* (MON). In our terms, empty tables will correspond to pseudo-data (Sec. 4.4.1)¹¹⁹.

The table type is a record-body supplemented by some properties of attributes that may be split into two groups: *yoks* as defined in Sec. 4.3.4 and *default values*, which go a little beyond our model, but it may be easily introduced into it. Here is an example of two such declarations which are cited with only small modifications after [4] p. 14¹²⁰:

```
CREATE TABLE Departments
(
  Department_ID      Number(3) PRIMARY KEY,
  Department_name    Varchar(20) NOT NULL      UNIQUE
  City               Varchar(50)
);
```

```
CREATE TABLE Employees
(
  Employee_ID       Number(6) PRIMARY KEY,
  Name              Varchar(20) NOT NULL,
  Position          Varchar(9)  DEFAULT NULL,
  Manager           Number(6) ,
  Employment_date   Date,
  Salary            Number(8,2),
  Bonus             Number(8,2),
  Department_ID     Number(3) REFERENCES Departments,
  CHECK (Bonus < Salary)
```

¹¹⁸ In Sec. 10 they are defined in a slightly different way.

¹¹⁹ I did not find that concept in the literature on SQL, neither any information about what sort of an object is assigned to a table variable by its declaration.

¹²⁰ In Sec. 10 we shall frequently refer to this example and also to some other examples from [4]. In all cases we keep the original notation, where Number(p) denotes a type of total numbers with p digits, and Number(p, s) denotes the type of decimal numbers of the total number of digits equal to p and the number of digits after decimal point equal to s. In turn Varchar(n) denotes the type of strings of length not exceeding n.

)

The tabulation in this example shows a certain universal structure of a declaration:

- in the first column we see column names, i.e., the attributes that are common to all the records (rows) constituting a table,
- the remaining columns carry information about data stored in columns; in our model, they will be expressed by bodies and yoks,
- a special case is a piece of information expressed by REFERENCES Departments, which describes a subordination relation between tables (Sec. 10.3),
- what we see in the last row of the second declaration is a condition concerning an expected relation between the values of the fields Salary and Bonus in each row of the future table; the bonus cannot be higher than the salary; in our terminology, this is a yok expression using a general quantification.

The elements of a table declaration, except column names, define so-called *integrity constraints*. Their meanings are as described below. According to a convention assumed earlier in this book, whenever we say that an error message is raised, we mean that at the same time that our program aborts.

1. Number(3) — The type of data in the column.
2. DEFAULT — The default value.
3. NOT NULL — All fields in the column must not be empty, i.e., none of them may be NULL. An attempt of a violation of this constraint should generate an error signal.
4. UNIQUE — No two identical data may appear in the column. If that happens, an error message should be raised.
5. PRIMARY KEY — This column is indicated as a *primary key*. Each primary key must be an *unambiguous key*, which means that the value of that key in a row identifies that row unambiguously. The attribute *primary key* may be assigned to more than one column. The database engine should react with an error message for each violation of the unambiguity of a primary key.
6. REFERENCES Departments — The field Department_ID in table Employees is related to the field of the same name in the table Departments. Relations between tables are used to modify tables and to set queries (see later).
7. CHECK(Bonus<Salary) — Whenever we add a new row to a table, or we modify an existing row in a way that violates this condition, an error message is generated.

As we see from this example, when we declare a table variable, we simultaneously define its type, i.e., its body and yok¹²¹. This type defines five groups of properties of the future table:

1. the names of columns,
2. the types of values in all fields of a given column, e.g., Number(6),
3. restrictions concerning columns as a whole, e.g., PRIMARY KEY, NOT NULL or UNIQUE,
4. relationships between values in each row, e.g., CHECK(Bonus<Salary),
5. relationships between tables by indicating related columns in tables, e.g., REFERENCES Departments.

As was already said, the properties of columns describe by 2. to 5. are called *integrity constraints*. They are, however, not the only examples of such constraints. Another example of an integrity constraint may be the requirement that some operations on balance sheets must not change the balance-sheet total (an example in Sec. 9.5).

¹²¹ It seems that SQL lacks a mechanism that would allow to define a table type independently of a variable declaration.

9.4 The subordination relation for tables

Subordination relations in SQL define links between tables that are used when we perform operations on several tables that are linked together. In our terms, relations may be regarded as yoks that define properties of databases, where databases are sort of records that carry tables.

The mechanism of establishing relations between tables appears in SQL literature in several versions. All of them are based on a common idea, although their implementations may be different. Below I try to describe this common idea.

Consider the tables `Departments` and `Employees` from Sec. 9.3. In `Employees`, we have a column `Department_Id` which defines the association of an employee to a department. In its declaration we have the constraint `REFERENCES Departments` expressing the fact that in the table `Departments` we may find information about the department where the employee is employed. Instead of storing in the table `Employees` the information about the department where he/she works, we only show the ID of that department that identifies the appropriate row in the table `Departments`. Now, for this construction to have a practical sense, our two tables must satisfy three conditions:

1. the column `Department_ID` must appear in both tables,
2. every ID of a department which is in the table `Employees` must also appear in the table `Departments`,
3. in the table `Departments` the attribute `Department_ID` must be an unambiguous *key*.

If these conditions are satisfied, then we say that:

*the attribute `Department_ID` links the tables `Departments` and `Employees`
with a one-to-many relation (abbr. 1-M)*

With every department, there is associated a set (possibly empty) of employees, whereas, with every employee, there is associated exactly one department.

In the pair of tables, `Departments`, and `Employees`, the table `Departments` is a *parent* table or a *superior table*, whereas `Employees` is a *child* table or a *subordinated* table. The attribute `Department_ID` is a *primary key* in the table `Departments` and a *foreign key* in the table `Employees`.

If an employee's row `ER` and a department's row `DR` have the same value in the field `Department_ID`, then we say that the `ER` *points* to the `DR` (MON).

By (1-M), we shall denote a ternary relation being a set of triples defining a relationship between two tables with a common attribute:

$$(1-M) \subseteq \text{Table} \times \text{Attribute} \times \text{Table}$$

We assume that

$$(\text{tab-1}, \text{atr}, \text{tab-2}) : (1-M) \quad \text{iff} \quad \text{tab-1 is a parent of tab-2 with a primary key atr.}$$

In our example, the triple (`Departments`, `Department_ID`, `Employees`) is, therefore, an element of such a relation. In that case, the attribute `Department_ID` is called a *linking key* of our tables.

Observe now that this relation may be broken by the modification of one or both tables, e.g., whenever:

- we remove a row from `Departments` that is pointed by a row from `Employees`,
- we insert a row to `Employees` with department's ID that does not exist in `Departments`,
- in one of our tables we rename the attribute `Department_ID`,
- we insert to `Departments` a new row with an ID equal to the ID of another row, and in that way, we spoil the unambiguity of the key `Department_ID`.

The fact that two tables are in the relation (1-M) may be used when we generate reports or create new tables. However, checking each time, if two given tables are in the (1-M) relation, would not be very practical. It is

much better to declare in advance that such a relation should hold, and then make sure that the database engine does not allow to violate that declaration.

In our example, the declaration of (1-M) relationship between Departments and Employees is implicit in the declarations of the corresponding table-variables:

- in the declaration of Departments, the attribute Department_ID is declared as a PRIMARY KEY; recall that every primary key has to be unambiguous,
- in the declaration of Employees, the attribute Department_ID is linked to the table Departments by the constrain REFERENCES Departments.

The establishment of a relation (1-M) between tables has consequences for operations on these tables. For instance:

- Introducing an employee who has been employed in a non-existent department is impossible. The database-engine will force the programmer to introduce the new department in the first place.
- A department's record cannot be removed from a table until there are employees employed in that department. An alternative solution is that in such a case, all employees of the deleted department are "automatically" removed.
- One can request the generation of a table with three columns that combine information from both linked tables, e.g., with columns Name, Department_name, City.

A particular case of a (1-M) relation is a (1-1) relation, where for every record in a parent table, there is at most one record in the corresponding child table. Notice that "at most one" rather than "exactly one", which means that (1-1) relation does not need to be symmetric. Consequently, one of these tables is a parent and another — a child.

To formalize the investigation on parent-child relations, we introduce the concept of a *parent-child graph* (MON) which is an arbitrary finite (possibly empty) set of triples of identifiers:

$$\text{pcg} : \text{ParChiGra} = \text{FinSub}(\text{Identifier} \times \text{Identifier} \times \text{Identifier})$$

The elements of this set are called *parent-child edges*. Intuitively every edge (ide-c, ide, ide-p) corresponds in a database to a relation, which holds between the tables named ide-c (child), ide-p (parent) with the primary key ide.

9.5 Instructions of table modification

Tables that have been declared or made accessible (see Sec. 10.9.6.11) may be modified using a large class of instructions. Below a few examples:

Entering a new column to a table:

```
ALTER TABLE Employees
  ADD COLUMN ID_number CHAR(11) DEFAULT NULL
```

We add a column to a table, and we indicate a default value for that column.

Deleting a column from a table

```
ALTER TABLE Departments
  DROP COLUMN Department_ID CASCADE (or RESTRICT)
```

If this instruction is executed with the option CASCADE, then the deletion of a column results in the deletion of all objects of a database (tables, perspectives,...) that refer to that column. In the case of RESTRICT, the instruction is not executed whenever such objects exist in the database.

Notice that the instructions from the group ALTER TABLE modify not only the content (the data) of a table but also its type. There are other examples of instructions altering tables ([54] p. 49):

- ALTER COLUMN — column-type is modified by SET DEFAULT or DROP DEFAULT, which sets or drops a default value.
- ADD — new constraint is added to an existing column.
- DROP CONSTRAINT — the removal of a constraint from an indicated column. With this instruction, RESTRICT or CASCADE must be set.

Another group of table-modifying instructions changes the content of a table without modifying its type. Some typical examples are:

The insertion of a new record (row):

```
INSERT INTO Departments
VALUES (095, 'Marketing', 'London')
```

This instruction may also be written in a form where column names are explicit (cf. [46], p. 73)

```
INSERT INTO Departments (Department_ID, Dep_name, City)
VALUES (095, 'Marketing', 'London')
```

The modification of all data in one column. E.g., the increase of salaries of all salesmen by 10%:

```
UPDATE Employees
SET Salary = Salary * 1,1
WHERE Position = 'salesman'
```

The removal of all rows that satisfy a given condition. E.g., the removal of all employees who have no position:

```
DELETE FROM Employees
WHERE Position IS NULL
```

A particular situation takes place if we drop a row with a primary key which is a foreign key in a child-table, e.g.:

```
DELETE FROM Departments
WHERE Dep_name = 'production'
```

If in the child table Employees the key Department_ID is — as in our case — a foreign key and there exist rows which point to the rows that are supposed to be deleted from Departments, then the operation is not executed and an error message is generated. However, the operation:

```
DELETE FROM Departments
WHERE Dep_name = 'production' CASCADE
```

will be executed, and additionally, in the table Employees, all rows that point to the row, which is deleted from Departments, are deleted as well¹²².

9.6 Transactions

By a *transaction*, we mean a sequence of instructions closed (or not) in some parentheses such as, e.g., BEGIN TRANSACTION and COMMIT TRANSACTION¹²³. The mechanism of transactions that we shall call a *recovery mechanism* (MON) stops the execution of a transaction whenever:

- the execution would violate some integrity constraints, or
- the execution is impossible, e.g., we search for a non-existing element in a table.

¹²² There is a certain inconsistency in SQL compared with the deletion columns. In the case of rows option RESTRICT is set by the system without the possibility of choosing another option by the user.

¹²³ These parentheses may differ between applications (some manuals are not mentioning them at all). Here we use the notation of Bena Forty ([48], p. 175) which is a standard for Microsoft SQL Server.

In all such cases, the implementation returns to the initial database state of the transaction, a state called the *roll-back value of the database*¹²⁴.

Five following instructions are used to control the recovery mechanism of transactions in SQL-programs:

SAVEPOINT	— save rollback-value of a database
RELEASE SAVEPOINT	— delete rollback-value
ROLLBACK	— call-of transaction
IF	— a conditional activation of a rollback
COMMIT TRANSACTION	— accept transaction.

The instruction

```
SAVEPOINT savepoint-name
```

assigns the actual database value to a temporary user-defined variable `savepoint-name`. The instruction

```
RELEASE SAVEPOINT savepoint-name
```

deletes the variable `savepoint-name` (and its value) from the state. The instruction

```
ROLLBACK savepoint-name
```

brings the database to its rollback-value and deletes the variable `savepoint-name`. This instruction may also appear without a parameter, in which case the database is (probably?) rolled back to the value initial of transaction-execution¹²⁵. In such cases, the execution of a transaction should start with a default SAVEPOINT, which saves database value to some system variable. It also seems that ROLLBACK aborts program execution and generates an error message.

To make the execution of ROLLBACK dependent on an error message, one may use the conditional IF constructor. Ben Forta ([48] p. 179) shows the following example:

```
IF @@ERROR <> 0 ROLLBACK savepoint-name
```

It is explained there that `@@ERROR` is a system-variable whose value equals 0 if there is no error message, and (I guess) equals an error message in the opposite case.

This example suggests — although this has not been explicitly written — that the condition of IF might be of the form

```
@@ERROR = error-message
```

with a specific error message. Such a solution would allow making the execution of ROLLBACK dependent on the type of an error.

The execution of COMMIT results in saving the result of the transaction and deleting all earlier declared rollback-variables.

For instance, in a database carrying data about bank customers, the transaction that moves 1000 \$ from one account to another may have the following form:

```
BEGIN TRANSACTION
  SAVEPOINT start
  UPDATE Accounts
    SET Balance = Balance - 1000
    WHERE ClientID = 1250 ;
    IF @@ERROR <> 0 ROLLBACK start ;
  UPDATE Accounts
```

¹²⁴ I have to warn the reader that in all known to me manuals, transactions are described in an exceptionally unclear and incomplete way, and therefore my understanding of this construction is based more on guesses than on facts.

¹²⁵ The parameter less version of this instruction appears in the majority of manuals known to me.

```

SET Balance = Balance+ 1000
WHERE ClientID = 1260 ;
IF @@ERROR <> 0 ROLLBACK start
COMMIT TRANSACTION

```

The first ROLLBACK takes place if there is no customer in the database with ID equal to 1250, or if its balance-value is less than 1000. The second ROLLBACK is activated if the first is not, but there is no customer in the database with ID equal 1260.

Notice that after the execution of the first UPDATE, the actual sum of all deposits is not equal to the bank-balance of deposits, which means that the integrity constraints are violated. The second UPDATE “removes” this violation, but if it can’t be performed because of the lack of 1260-customer, then the transaction would end with an inconsistent database. The second ROLLBACK prevents such a situation.

9.7 Queries

Queries are used to collect information from databases, and more precisely — from one or more database tables. The execution of a query results in the generation of a table and possibly in displaying it on a monitor. Queries are constructed by several variants of operator SELECT. Below a few typical examples:

The selection of indicated columns of a table:

```

SELECT Name, Salary, Position
FROM Employees

```

As a result of this query, a monitor displays a three-column table with columns indicated by the parameters of SELECT.

The selection of columns combined with the filtering of rows:

```

SELECT Name, Salary, Position
FROM Employees
WHERE Department_ID = 10

```

In WHERE clause, we may have boolean expressions with operators on simple data described in Sec. 9.2.

Queries may be composed of other queries using operators called by Banachowski [9] “algebraic operators on queries”. These operators may be applied to more than one table. For instance:

```

SELECT Department_ID
FROM Departments
EXCEPT
SELECT Department_ID
FROM Employees

```

This query generates a one-column table of the IDs of these departments that appear in the table Departments but that do not appear in the table Employees. i.e., the IDs of departments with no employees.

A specific group of queries allows reaching more than one table. In such a case, we say that queries use the *joins of tables*. Below we see an example of a query that selects data from two tables — Employees and Departments.

```

SELECT Employee_ID, Name, Department_ID
FROM Employees, Departments
WHERE Employees.Department_ID = Departments.Department_ID
AND Departments.City = 'London'

```

This query generates a three-column table where each row contains the ID of an employee, his/her name, and the name of the department where he/she is employed. The condition in WHERE-clause is called a *joint predicate*. In our case, it returns only such rows where employees are employed in departments located in London.

In WHERE-clauses, we may use boolean expressions exploring basic predicates on simple data (Sec. 9.2), e.g.:

```
SELECT Employee_ID, Name, Salary
FROM Employees
WHERE Salary > 1000 AND Salary <= 2000
```

or set-theoretic operators. For instance, the query:

```
SELECT Employee_ID, Name, Position, Salary
FROM Employees
WHERE Position IN ('cashier', 'salesman', 'manager').
```

generates a table with cashiers, salesmen, and managers. The query:

```
SELECT Employee_ID, Name, Position, Salary
FROM Employees
WHERE Salary > ALL
(
  SELECT Salary
  FROM Employees
  WHERE Position = 'cashier'
)
```

generates a table that shows employees with salaries higher than the salaries of all cashiers. In this case, we have to do with a *nested query*, where the inner SELECT generates a column with the salaries of all cashiers. Let us denote:

sae : SalEmp — the set of values in the column Salary of the table Employees,
sac : SalCas — the subset of SalEmp that contains the salaries of cashiers,
shc : SalHigCas — the subset of SalEmp that contains salaries higher than the salaries of cashiers

In that case:

$$\text{SalHigCas} = \{ \text{sae} \mid \text{sae} : \text{SalEmp} \text{ and } (\forall \text{sac} : \text{SalCas}) \text{sae} > \text{sac} \}$$

therefore:

$$\text{SalHigCas} = \{ \text{sae} \mid \text{sae} : \text{SalEmp} \text{ and } \underline{\text{all}}.(\text{SalCas}, >).\text{sae} = \text{tt} \}$$

where $>$ is a predicate that compares numeric values and assumes **ee** whenever at least one of its arguments is not a numeric value.

The transparency of $>$ implies that the set **SalHigCas** contains numbers only, although it may be empty as well. In particular, it is empty, if **SalCas** contains at least one not-number.

In none of the bibliographic sources, I found information about what happens, if inequality $\text{sae} > \text{sac}$ generates an error. Will it interrupt a program and generate an error, or the query will generate some “unexpected” table, maybe empty?.

Let us consider now a query that results from the former if ALL is replaced by EXISTS, i.e., that generates the table of employees with salaries higher than the salary of at least one cashier¹²⁶:

```
SELECT Employee_ID, Name, Position, Salary
FROM Employees
WHERE Salary > EXISTS
(
```

¹²⁶ In this case I use a syntax which is — maybe — not compatible with SQL. I used it, however, to keep the similarity with the ALL example, whose syntax (although not the example itself) has been taken from [72] p. 139.


```

SELECT Salary
  FROM Employees
 WHERE Position = 'cashier'
)

```

Denote:

$\text{shs} : \text{SalHigSomCas}$ — salaries higher than some salaries of cashiers.

In that case:

$$\text{SalHigSomCas} = \{ \text{sea} \mid \text{sea} : \text{SalEmp} \text{ and } (\exists \text{sac} : \text{SalCas}) \text{ sea} > \text{sac} \}$$

hence:

$$\text{SalHigSomCas} = \{ \text{sea} \mid \text{sea} : \text{SalEmp} \text{ and } \underline{\text{exists}}.(\text{SalCas}, >).\text{sac} = \text{tt} \}$$

In that case, contrary to the former, if SalCas contains not-numbers, then the set SalHigSomCas does not need to be empty.

Notice now that whenever the evaluation of $\text{sae} > \text{sac}$ for some sac , generates an error, then

$$\underline{\text{exists}}.(\text{SalCas}, >).\text{sac} = \text{ff}$$

If, however, we replace EXISTS by SOME, then ee may appear. This replacement does not change the table generated by our query but affects error generation.

Quantifiers may also appear in the context of joining tables. The query shown below generates the table of departments where at least one employee is employed.

```

SELECT Department_ID
  FROM Departments
 WHERE Department_ID = EXISTS
  (
    SELECT Department_ID
    FROM Employees
  )

```

As was mentioned in Sec. 9.2, for every simple operator, there exists its negated version, e.g., $=$ and $<>$, LIKE and NOT LIKE, etc. Similarly, we have NOT IN. In the case of set-theoretic quantifiers, I have found only NOT EXISTS and only in [72] p. 147 and in [46] p. 242. Of course, none of these sources concerns the case where EXISTS generates an error.

From a denotational perspective, queries may be regarded as expressions since they generate a value (a table) without changing a state.

9.8 Aggregating functions

The *aggregating functions* SUM, MAX, MIN, AVG take as arguments one-column tables that are the results of queries and return a number. If the argument-table is empty, then the value of an aggregating function is NULL ([54] p. 148).

Function COUNT takes an arbitrary one-column table and returns the number of these rows where NULL does not appear. In turn, COUNT(*) takes an arbitrary table and counts all rows, including duplicates ([72] p. 155).

9.9 Views

If we want to use a query more than once, we may declare it as a procedure. Such procedures are called *views*. Below we see an example of a view-declaration:

```
CREATE VIEW Officials
```

```
(Employee_ID, Name, Salary)
AS SELECT Employee_ID, Name, Salary
FROM Employees
WHERE Position = 'official'
```

This view is named `Officials` and creates a three-column table by selecting columns from `Employees` and rows with `'official'` that stands in the column `Position`.

Since views are procedures, they have no counterparts in syntax (cf. Sec. 6.1.3). At the syntactic level, we only have *view declarations* `CREATE VIEW` and *view calls* (`MON`) that refer to the names of views.

View calls may be used in queries in the same way as tables and, of course, a view is executed in the call-time state rather than in the declaration time state. In SQL-manuals, views are, therefore, referred to as *virtual tables*. Views may also be called in instructions that create or modify tables. Consider the following view-declaration:

```
CREATE VIEW Salesmen
AS SELECT * FROM Employees
WHERE Department_ID = 20
```

In this declaration, the star “*” means that we chose all columns, and the number `20` is the ID of the sales department. Calling the view `Salesmen` we can create an instruction that modifies the table `Employees` by increasing the salaries of all salesmen by 10%:

```
UPDATE Salesmen
SET Salary = Salary * 1.1
```

In the case of using views for the modifications of tables, each SQL engine has its specific restrictions. E.g., MySQL requires that in `SELECT`-clauses, only column names may appear.

A special case are *views with check option* which force the checking of a condition when views are used in instructions. Banachowski [9] shows an example of such a view:

```
CREATE VIEW Employees_on_not-paid_holiday
AS SELECT *
FROM Employees
WHERE Salary = 0 OR Salary IS NULL
WITH CHECK OPTION
```

If this view is used in the instruction:

```
UPDATE Employees_on_not-paid_holiday
SET Salary = 1000
WHERE Name = 'Smith'
```

then it is not executed if the salary of Smith is 0 or NULL.

9.10 Cursors

Cursors are used to assign selected rows of tables to data variables. This mechanism allows for processing database data using programs written in user-interface programming languages such as API or CLI (see Sec. 9.1). A cursor points to a row in an indicated table and allows us to get data from that row. Tables indicated by cursors are defined using queries. As a matter of fact, we should not talk about a cursor as such, but about a *cursor of a table*, or maybe about a *cursor of a query*.

Cursors are created using *cursor declarations*, which assign a cursor to a *cursor name* (an identifier). Such declarations are of the form¹²⁷:

```
DECLARE cursor_name IS
```

¹²⁷ The syntax of a cursor-declaration depends upon application. Here I use the syntax of ORACLE ([72] p. 352).

SELECT ...

After a cursor has been declared, it is not yet ready for use. To make it ready, we have to apply an *opening instruction* of the form:

```
OPEN cursor_name.
```

This instruction causes the execution of SELECT, which appears in the declaration and (I guess) in the setting of the so-called *cursor grasp* at the “position” preceding the first row of the generated table. The operation of getting data from a table is:

```
FETCH NEXT cursor_name INTO variable
```

The NEXT means getting the data of the row next to the grasp and moving the grasp one row further. It seems, therefore, that OPEN sets the cursor before the first row.

The FETCH NEXT instruction is usually applied in a program loop, which means that when a grasp reaches the last row of a table, it cannot be moved further, I have found only one comment on that issue in [72] p. 353 (my translation from the text in Polish):

In every implementation of databases, cursors are implemented in a slightly different way, but each of them enables a correct cursor-closing without an unnecessary generation of errors.

If a cursor is temporarily not needed, we close it by instruction:

```
CLOSE cursor_name
```

This instruction leaves the cursor structure for reopening.

9.11 The client-server environment

So far, when talking about SQL-systems, we were assuming tacitly that the user has a database to his/her excluded disposal. However, this is usually not the case. In general, there is more than one user, which means that we need tools to give them and deny access to databases. Here is an instruction scheme which sets a lock on a given table:

```
LOCK TABLE table_name
  IN [SHARE | EXCLUSIVE]
  [NOWAIT]
```

where the options in square-brackets mean the following:

- SHARE — the lock applies to all users,
- EXCLUSIVE — the lock applies to all users except the one who sets the lock,
- NOWAIT — do not wait for lock setting, if it cannot be set at the moment.

Locks are removed by instructions COMMIT or ROLLBACK. An example of an instruction which gives permissions to a given user may be:

```
GRANT SELECT, UPDATE (Salary)
  ON Employees
  TO Smith
```

This instruction grants the permission of performing SELEC and UPDATE in the table Employees to the user Smith.

These mechanisms of SQL may differ between the application, but since they are relatively simple to describe, I shall not discuss them later. For that reason, they will not be included in **Lingua-SQL**.

10 LINGUA-SQL

10.1 General assumptions about the model

As was already explained, **Lingua-SQL** does not rely on any existing SQL-engine but on its own database-operations. The denotational model of that language will be built as an extension of **Lingua-2** model by adding:

1. new domains of data including specific SQL simple data, rows, tables and databases,
2. the corresponding new domains of bodies, composites, values, and denotations,
3. new operations defined on new domains.

Technically the SQL-part of our model will be built in a slightly different way than in the case of **Lingua-2**. This difference concerns the applicative part of the model, where we skip constructors of composites and proceed straight to constructors of value. We proceed in this way since operations on rows and tables involve yokes, which are not available at the level of composites.

Similarly, as in the previous versions of **Lingua**, we do not pretend here to build a practical repertoire of SQL-tools. Our goal is just to show a denotational framework for databases, rather than to build a real API. Hopefully, this framework will allow building a real language in some future.

10.2 Data

SQL data are separated from the data of **Lingua-2** in the sense that lists, records, and arrays of **Lingua-2** do not carry rows, tables, and databases of **Lingua-SQL** and table fields of **Lingua-SQL** do not contain lists, records, and arrays of **Lingua-2**. At the same time, however, the extended repertoire of SQL simple-data is available for the constructors of lists, records, and arrays.

The only new simple data in **Lingua-SQL** are associated with time, i.e., with calendars and clocks, and are the following:

dat : Date = Year x Month x Day
 tim : Time = Hour x Minute x Second
 dti : DateTime = Date x Time

where:

yea : Year = {0,...,9999} (just an example)
 mon : Month = {1,...,12}
 day : Day = {1,...,31}
 hou : Hour = {0,...,23}
 min : Minute = {0,...,59}
 sek : Second = {0,...,59}

The remaining data domains of **Lingua-SQL** are defined by the following equations:

dat	: SimData = Boolean Integer Real Word Date Time DateTime { Θ }	simple data
dat	: NdbData = SimData List Array Record	non-database data ¹²⁸
lis	: List = NdbData ^{c*}	
arr	: Array = Number \Rightarrow NdbData	
rec	: Record = Identifier \Rightarrow NdbData	
dat	: SqlData = Row Table	SQL data
row	: Row = Identifier \Rightarrow (SimData { Θ })	
tab	: Table = Row ^{c*}	
dat	: Data = NdbData SqlData	

The symbol Θ represents an empty field of a row and is called an *empty data*¹²⁹. Empty data will never appear as values of expressions and will never be assigned to variables in states. Of course, since tables are lists of rows, empty data will appear in tables.

Databases do not appear at the level of data. They will be defined at the level of values (Sec. 0), where we can define integrity constraints represented by yokes and subordination relations.

In **Lingua-SQL**, primary operations on data include all primary operations of **Lingua-A** (Sec. 4.3.1) appropriately extended to a new set of simple data, plus operations corresponding to time-oriented data, rows, and tables. Operations on time-oriented data are assumed to be parameters of our model.

The subcategories of numbers such as INTEGER, SMALLINT, BIGINT, DECIMAL(p, s), or of words CHARACTER(n), CHARACTER VARYING(n), BLOB, may be described by types with appropriate yokes

The relation *equal* introduced in Sec. 4.3.1 is extended to new simple data in a natural way.

At the level of domain equations, tables may contain rows of different lengths and different attributes. Of course, such tables will not be reachable. A table with an empty tuple of rows is called an *empty table* and is denoted by $()$.

Notice that rows and tables, similarly to list, arrays, and records, carry data rather than composites or values.

10.3 Subordination of tables

Subordination relations describe binary relationships that can hold between tables. Let then **A** and **B** be tables and let *ide* be an attribute that appears in both of them. Let **A.ide** and **B.ide** be the corresponding columns in these tables.

We say that **A** is *subordinate to B at ide* or that **B** is *superior to A at ide* — alternatively, we say that **A** is a *child* and **B** is a *parent* — that we write as

A sub[ide] B

if the following three conditions are satisfied:

1. an *ide*-column appears in both tables,

¹²⁸ They are called “non-database” rather than “non-sql” since time-oriented data are coming from SQL.

¹²⁹ Note that Θ , which is assignable to fields of rows is different from Ω which is assigned to a variable at declaration-time.

2. column **B.ide** is repetition-free which means that each of its elements unambiguously identifies the row, where it belongs,
3. column **A.ide** contains only the data that appear in **B.ide**, which — together with 2. — means that each row of **A** unambiguously points to a row in **B**.

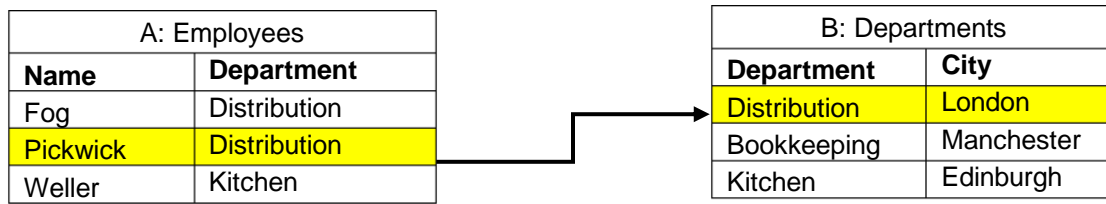


Fig. 10.3-1 Employees is a subordinate to Departments at Department

On Fig. 10.3-1, we see an example where the following relation holds:

Employees sub[Department] Departments

The attribute *ide* is called a *subordination indicator* (MON) for **A** and **B**. The column **A.ide** is said to be a *subordinated column* for **B.ide**. If in column **B.ide** there is an element which appears in **A.ide** more than once (more than one employee is employed in the same department), then we say that our subordination relation is of type (1-M), read *one-to-many*. In the opposite case, we say that it is of (1-1) type. Notice that in both cases, there may be some elements in the superior column that do not appear in the subordinated column (departments with no employees). It means that a (1-1) relationship does not need to be symmetric.

Notice that the subordination relation concerns tables rather than table composites, which means that to decide if that relation holds, we do not need to compare bodies.

As one may easily check, subordination relations between tables (they may be more than one) may be spoiled in four following cases:

- A. if we remove a column assigned to a subordination indicator (condition 1),
- B. if we add such a row to a parent table **B** that introduces repetition to the indicator-column (condition 2),
- C. if from a parent table **B** we remove a row pointed to by a row of the child table **A** (condition 3),
- D. if to a child table **A**, we add a row that introduces to the indicator column an element that does not appear in the indicator column of the parent table (condition 3).

States of programs operating on databases have to carry information about declared subordination relations between tables. To include this mechanism in our model, we use the concept of a *subordination graph* (MON) defined as a set of triples of identifiers:

$$sgr : \text{SubGra} = \text{Sub}(\text{Identifier} \times \text{Identifier} \times \text{Identifier})^{130}$$

Each tuple (*ide-c*, *ide*, *ide-p*) in *sgr* is called an *edge of the subordination graph*, where *ide-c* (child) and *ide-p* (parent) play the role of graph nodes, and *ide* is a label of the edge. In the context of a given state, each edge expresses the fact that a subordination relation holds between the tables named *ide-c* and *ide-p*, where *ide* is the subordination indicator.

About the subordination graphs, we assume only that *ide-c* ≠ *ide-p*, although cycles are allowed. Notice also that there may be many edges starting in one node (one child may have many parents), and many edges may end in one node (many children may have a common parent).

¹³⁰ Notice that since the set **Identifier** is finite, each subordination graph is finite as well. This is why we use here the operator **Sub** rather than **FinSub**.

10.4 Bodies

The domains of bodies in **Lingua-SQL** are defined by the following equations, where q marks body initials related to SQL:

$\text{bod} : \text{SimBody} = \{('boolean'), ('number'), ('word'), ('date'), ('time'), ('date-time')\}$ simple bodies

$\text{bod} : \text{NdbBody} = \text{LisBody} \mid \text{ArrBody} \mid \text{RecBod}$ non-database bodies

$\text{bod} : \text{LisBody} = \{ 'L' \} \times \text{NdbBody}$ list bodies

$\text{bod} : \text{ArrBody} = \{ 'A' \} \times \text{NdbBody}$ array bodies

$\text{bod} : \text{RecBody} = \{ 'R' \} \times \text{BodRec}$ record bodies

$\text{bod} : \text{RowBody} = \{ 'Rq' \} \times \text{BodRow}$ row bodies

$\text{bod} : \text{TabBody} = \{ 'Tq' \} \times \text{Row} \times \text{RowBody}$ table bodies

$\text{bod} : \text{Body} = \text{NdbBody} \mid \text{RowBody} \mid \text{TabBody}$

$\text{bod} : \text{BodyE} = \text{Body} \mid \text{Error}$

where:

$\text{bor} : \text{BodRec} = \text{Identifier} \Rightarrow \text{NdbBody}$ body records

$\text{bor} : \text{BodRow} = \text{Identifier} \Rightarrow \text{SimBody}$ body rows

Rows contained in table bodies carry information about default data for columns. Default data may be empty, i.e., equal to Θ . Of course, the list of attributes of the default-data row must coincide with the list of the attributes of the row body. This property will be insured by table-body constructors. It is worth noticing in this place that that bodies of tables include data (rows).

Since $\text{BodRow} \subseteq \text{BodRec}$, we use the same metasymbol **bor** to denote the elements of both.

The function **CLAN-Bo** from **Lingua-2** is extended in an obvious way on row bodies. In the case of table bodies, we assume that for in each table in **CLAN-bo**.('Tq', row, bod):

1. all rows must belong to **CLAN-bo.bod**, which implies that **bod** must be a row body,
2. all rows must carry non-empty data in the fields whose default data (indicated by **row**) are non-empty, although they do not need to be the default data.

Default data will be used when adding a new row (Sec. 12.2.6) or a new column (Sec. 12.2.7) to a table.

We assume that empty table () belongs to the clan of every table body. The function **sort** (Sec. 4.3.3) is extended in an obvious way to new bodies.

Body constructors in **Lingua-SQL** include all body constructors of **Lingua-2**, which are appropriately extended to cover new simple data. Additionally we have constructors for row bodies and table bodies.

Here we are coming to a certain singularity of the SQL applicative layer. Whereas in **Lingua-A** we have defined algebras of bodies, composites, transfers and yokes successively one after another, now some of row

constructors will take composites as arguments, and composite constructors will take transfers and yokes. Formally this means that we define one algebra of bodies, composites, transfers and yokes.

Row constructors

bo-create-ro	: Identifier x BodyE	\mapsto BodyE	create a one-attribute row
bo-add-to-ro	: Identifier x BodyE x BodyE	\mapsto BodyE	add an attribute to a row
bo-cut-from-ro	: Identifier x BodyE	\mapsto BodyE	remove an attribute from a row
bo-get-from-ro	: Identifier x BodyE	\mapsto BodyE	select a body from a row
bo-check-in-ro	: Identifier x BodyE x BodyE	\mapsto BodyE	check if bodies coincide

All these constructors except bo-cut-from-ro are defined analogously to the constructors of record bodies (Sec. 4.3.2). The cut-constructor will be used by the operation of removing a column from a table (Sec. 10.5.5). At the level of rows it removes an attribute from a row.

Table constructors

bo-create-empty-table	: CompositeE	\mapsto BodyE
bo-add-ro-to-tb	: BodyE x BodyE	\mapsto BodyE
bo-get-ro-from-tb	: BodyE	\mapsto BodyE
bo-join-tb	: BodyE x BodyE	\mapsto BodyE

Below we give the definitions of table-body constructors. The definitions of remaining constructors are left to the reader.

bo-create-empty-table : CompositeE \mapsto BodyE

```
bo-create-empty-table.com =
  com : Error     $\rightarrow$  com
  sort.com  $\neq$  'Rq'  $\rightarrow$  'row-expected'
let
  (row, bod) = com
true       $\rightarrow$  ('Tq', row, bod)
```

This constructor builds a table body out of a composite whose row becomes the row of default data and whose body becomes the common body of all future rows of the table. The fact that we start from a composite guarantees that row : CLAN-bo.bod.

bo-add-ro-to-tb : BodyE x BodyE \mapsto BodyE

```
bo-add-ro-to-tb.(bor-r, bod-t) =
  bod-i : Error     $\rightarrow$  bod-i    for i = r, t
  sort.bod-r  $\neq$  'Rq'  $\rightarrow$  'row-expected'
  sort.bod-t  $\neq$  'Tq'  $\rightarrow$  'table-expected'
let
  ('Rq', bod-r)      = row-r
  ('Tq', row-d, bod-rt) = bod-t
  bod-r  $\neq$  bod-rt     $\rightarrow$  'bodies-not-compatible'
true               $\rightarrow$  bod-t
```

This constructor does not create a new body but only checks if first argument is a row body, second argument is a table body, and if the row body is compatible with the table body. This constructor anticipates the fact that adding a row to a table does not change the body of that table.

bo-get-ro-from-tb : BodyE \mapsto BodyE

```
bo-get-ro-from-tb.bod =
  bod : Error     $\rightarrow$  bod
```



```

sort.bod ≠ 'Tq' → 'table-expected'
let
  ('Tq', row, bod-r) = bod
true → bod-r

```

If we select a row from a table, then the body of this row equals the row body of the table.

```

bo-join-tb : BodyE x BodyE → BodyE
bo-join-tb.(bod-1, bod-2) =
  bod-i : Error → bod-i          for i = 1,2
  sort.bod-i ≠ 'Tq' → 'table-expected' for i = 1,2
bod-1 ≠ bod-2 → 'bodies-not-compatible'
true → bod-1

```

This constructor only checks if the argument bodies are table bodies, and if they are equal.

10.5 Composites

The domain of composites in **Lingua-SQL** is defined in the same way as in **Lingua-A** (Sec. 4.3.3), i.e., by domain equations:

```

com : Composite = {(dat, bod) | dat : CLAN-Bo.bod}
com : CompositeE = Composite | Error

```

This definition means, in particular, that the fields of rows and tables carry data, rather than composites. For technical reasons we introduce an auxiliary domain of simple composites:

```

com : SimCom = {(dat, bod) | (dat, bod) : CompositeE and bod : SimBody}

```

We also assume that for every simple body `bod`

```

Θ : CLAN-Bo.bod

```

i.e., that (Θ, bod) is a composite.

SQL constructors of composites will be defined in a way analogous to that described in Sec. 4.3.3. Of course, they should satisfy the rule that whenever a SQL operation generates an error, then the corresponding **Lingua-SQL** constructor of composites should also generate an error. At the same time, whenever in a SQL API's "one cannot expect a meaningful result" (cf. Sec. 9.2), our constructor should generate an error as well.

Before we proceed to the definitions of SQL-constructors of composites, we have to take an engineering decision of choosing one of the two following strategies:

1. for every future operation available at the level of syntax, we create an individual composite constructor, or
2. we define some basic composite constructors, and later all remaining constructors are defined as their combinations.

For instance, a replacement of a data in a table may be described as one table-to-table constructor or as a combination of the replacement of a data in a row and a row in a table.

The first option seems closer to SQL tradition. It leads, however, to long lists of constructors "one for each case" and may result in an imperfect understanding of language semantics. We choose, therefore, the second option which — hopefully — will contribute to:

1. a simpler description of the language,
2. a shorter list of program-building rules,
3. the restriction of interpreter's source-code for **Lingua-SQL** to basic constructors, and the definition of other constructors as procedures from the level of our language.

10.5.1 Signatures of composite constructors

We assume that all composite constructors of **Lingua-A** are available in **Lingua-SQL**. We also assume that SQL-constructors of new simple composites (composites that correspond to simple data) are regarded as parameters of our model, and therefore we skip their definition. The remaining SQL constructors are split into groups corresponding to categories of data.

Unlike in **Lingua-A**, some of our constructors take bodies, transfers, and yokes as arguments. Of course, the domains of transfers and yokes will be extended to the new domain of composites.

A constructor of empty composites

empty : BodyE \mapsto CompositeE

The constructors of row composites

co-create-ro	: Identifier x CompositeE	\mapsto CompositeE
co-add-to-ro	: Identifier x CompositeE x CompositeE	\mapsto CompositeE
co-cut-from-ro	: Identifier x CompositeE	\mapsto CompositeE
co-get-from-ro	: Identifier x CompositeE	\mapsto CompositeE
co-change-in-ro	: Identifier x CompositeE x Transfer x Yoke	\mapsto CompositeE

Row constructors of table composites

co-create-empty-table	: CompositeE	\mapsto CompositeE
co-add-ro-to-tb	: CompositeE x CompositeE	\mapsto CompositeE
co-cut-ro-from-tb	: Yoke x CompositeE	\mapsto CompositeE
co-get-ro-from-tb	: Yoke x CompositeE	\mapsto CompositeE
co-exclude-ro-from-tb	: CompositeE x CompositeE	\mapsto CompositeE
co-filter-ro-in-tb	: Yoke x CompositeE	\mapsto CompositeE
co-join-tb	: CompositeE x CompositeE	\mapsto CompositeE
co-intersect-tb	: CompositeE x CompositeE	\mapsto CompositeE

Column constructors of table composites

co-add-co-to-tb	: Identifier x CompositeE x CompositeE	\mapsto CompositeE
co-cut-co-from-tb	: Identifier x CompositeE	\mapsto CompositeE
co-filter-co-from-tb	: AcPaDe x CompositeE	\mapsto CompositeE
co-change-co-in-tb	: Identifier x CompositeE x Yoke	\mapsto CompositeE
co-get-co-from-tb	: Identifier x CompositeE	\mapsto ColumnE

Table constructor creating a derivative table (Sec. 10.5.6)

co-create-der-tb : CompositeE x CompositeE x Identifier x Yoke \mapsto CompositeE

10.5.2 Constructors of simple composites

We assume that the constructors of simple composites in **Lingua-SQL** cover:

- all constructors of simple composites from **Lingua-2**,
- the zero-argument composites generating new simple composites,
- some repertoire of operations and predicates on such composites whose examples were shown in Sec. 9.2.

This set of constructors is regarded as a parameter of our model. We only assume that it contains a special constructor that to each body assigns a composite with the empty data (it corresponds to an empty field of a row or of a table).

$\text{empty} : \text{BodyE} \mapsto \text{CompositeE}$

```
empty.bod =
  bod : Error           → bod
  not bod : SimpleBod → 'simple-body-expected'
  true                 → (Θ, bod)
```

Since we have assumed earlier that Θ belongs to the clan of each body, each (Θ, bod) is a correct composite.

10.5.3 Constructors of row composites

SQL row constructors, although close to record constructors (Sec. 4.3.3), differ from them in two ways:

1. they allow for the construction of only such rows, whose attributes carry simple data,
2. an attribute may carry the empty data Θ .

In the second case, we have to do with an empty field, which may be later filled with a data of an appropriate sort.

Below three examples of definitions.

Add an attribute to a row

$\text{co-add-to-ro} : \text{Identifier} \times \text{CompositeE} \times \text{CompositeE} \mapsto \text{CompositeE}$

```
co-add-to-ro.(ide, com-s, com-r) = (s – simple, r – row)
  com-i : Error           → com-i    for i = s, r
  let
    (dat-s, bod-s) = com-s
    (dat-r, bod-r) = com-r
    bod-nr        = bo-add-to-ro.(ide, bod-s, bod-r) (–nr – new row)
  bod-nr : Error           → bod-nr
  let
    new-com = (dat-r[ide/dat-s], bod-nr)
  oversized.new-com → 'overflow'
  true         → new-com
```

Adding an attribute to a row composite extends both — the row (data) and its body — which guarantees that the new composite is well-formed. We assume that body constructor **bo-add-to-ro** makes all necessary checks, and if they do not raise errors, extends body **bod-r** by new attribute **ide**.

Get a data from a row

$\text{co-get-from-ro} : \text{Identifier} \times \text{CompositeE} \mapsto \text{CompositeE}$

```

co-get-from-ro.(ide, com) =
  com : Error    → com
  let
    (dat, bod) = com
    bod-a      = bo-get-from-ro.(ide, bod)           body assigned to attribute ide
  bod-a : Error  → bod-a
  dat.ide =  $\Theta$  → 'empty-field'
  let
    ('Rq', bor) = bod
  true          → (dat.ide, bod-a)

```

We assume that bo-get-from-ro performs all necessary checks. Note that we do not need to check if dat.ide is defined because this follows from the fact that bod-a is not an error, and $(\text{dat}, ('Rq', \text{bor}))$ is a composite.

Change a data in a row conditionally

$\text{co-change-in-ro} : \text{Identifier} \times \text{CompositeE} \times \text{Transfer} \times \text{Yoke} \mapsto \text{CompositeE}$

```

co-change-in-ro.(ide, com, tra, yok) =
  com : Error    → com
  tra.com : Error → tra.com
  yok.com : Error → yok.com
  let
    (dat-r, bod-r) = com
    (dat-t, bod-t) = tra.com
    bod-c          = bo-check-in-ro.(ide, bod-r, bod-d)
  bod-c : Error  → bod-c
  (dat-y, bod-y) = yok.com
  let
    new-com =
      dat-y = tt → (row-r[ide/dat-t], bod-r)
      true      → com
  oversized.new-com → 'overflow'
  true              → new-com

```

The new data dat-t that is assigned to ide in the row of com is created by the application of transfer tra to the row composite com . The assignment takes place under the condition that the row composite satisfies yok . Before new data is inserted into the row, it is checked if its body is compatible with the body assigned in the row to the identifier ide .

10.5.4 Row constructors of table composites

Table constructors are used in the definitions of table transformations, views and queries. These constructors are split into two groups: *row constructors* and *column constructors*. To define them some auxiliary concept are needed.

We say that a *row body* bod is *compatible with a table body* $(\text{'Tq'}, \text{row}, \text{bod-r})$ if $\text{bod} = \text{bod-r}$. Take two rows with the same set of attributes:

$\text{row-1} = [\text{ide-1}/\text{dat-11}, \dots, \text{ide-n}/\text{dat-1n}]$

$\text{row-2} = [\text{ide-1}/\text{dat-21}, \dots, \text{ide-n}/\text{dat-2n}]$

We define a function

$$\text{fill-in.}(\text{row-2}, \text{row-1}) = [\text{ide-1}/\text{dat-31}, \dots, \text{ide-n}/\text{dat-3n}]$$

where for $i = 1;n$:

$$\begin{aligned} \text{dat-3i} &= \\ \text{dat-2i} \neq \Theta &\rightarrow \text{dat-2i} \\ \text{dat-2i} = \Theta &\rightarrow \text{dat-1i} \end{aligned}$$

This means that each empty value in dor-2 is replaced by a default value from dat-1 . This function describes a rule of adding a new row row-2 to a table whose default row is row-1 .

Create an empty table

$$\text{co-create-empty-table} : \text{CompositeE} \mapsto \text{CompositeE}$$

$$\text{co-create-empty-table.com} =$$

$$\begin{aligned} &\mathbf{let} \\ &\quad \text{bod} = \text{bo-create-empty-table.com} \\ \text{bod} : \text{Error} &\rightarrow \text{row} \\ \mathbf{true} &\rightarrow ((), \text{bod}) \end{aligned}$$

An empty table is created from a row composite whose row becomes the row of default values of the table and whose body indicates bodies assigned to attributes.

Add a row to a table

$$\text{co-add-ro-to-tb} : \text{CompositeE} \times \text{CompositeE} \mapsto \text{CompositeE}$$

$$\text{co-add-ro-to-tb.}(\text{com-r}, \text{com-t}) =$$

$$\text{com-i} : \text{Error} \rightarrow \text{com-i} \quad \text{for } i = r, t$$

$$\mathbf{let}$$

$$\begin{aligned} (\text{row}, \text{bod-r}) &= \text{com-r} \\ (\text{tab}, \text{bod-t}) &= \text{com-t} \\ \text{bod} &= \text{bo-add-ro-to-tb.}(\text{bod-r}, \text{bod-t}) \end{aligned}$$

$$\text{bod} : \text{Error} \rightarrow \text{bod}$$

$$\mathbf{let}$$

$$\begin{aligned} ('Rq', \text{bod-r}) &= \text{bod-r} \\ ('Tq', \text{row-d}, \text{bod-rt}) &= \text{bod-t} && (\text{rt} - \text{row-body of the table}) \end{aligned}$$

$$\text{row-fi} = \text{fill-in.}(\text{row}, \text{row-d})$$

$$\text{new-tab} = \text{tab} \odot (\text{row-fi})$$

$$\text{are-repetitions.new-tab} \rightarrow \text{'redundant-row'}$$

$$\mathbf{let}$$

$$\text{new-com-t} = (\text{new-tab}, \text{bod-t})$$

$$\text{oversized.new-com-t} \rightarrow \text{'overflow'}$$

$$\mathbf{true} \rightarrow \text{new-com-t}$$

The body of the added row must be compatible with the body of the table. Additionally, if the value of an attribute in the added row is empty, then in this place, we put the value of that attribute in the row of default values (which may be empty as well). Of course, the operation of adding a row to a table does not change the body of the table.

The table which is extended by a new row may be empty. In adding a row to a table, we also make sure that the new row is not redundant, i.e., equal to a row, which is already in the table.

Remove a row from a table

```

co-cut-ro-from-tb : Yoke x CompositeE  $\mapsto$  CompositeE
co-cut-ro-from-tb.(yok, com) =
  com : Error  $\rightarrow$  com
  sort.com  $\neq$  'Tq'  $\rightarrow$  'table-expected'
  let
    (tab, ('Tq', row-d, bod)) = com
    (row-1, ..., row-n) = tab
  n = 1  $\rightarrow$  'unique-row-cannot-be-removed'
  yok.(row-i, bod) = (tt, ('boolean'))  $\rightarrow$  (tab[i/?], bod) for i = 1;n
  true  $\rightarrow$  'no-such-row'

```

This constructor removes the first row of the table, which satisfies the yoke `yok`. If there is no such row, an error message is generated. In this definition, `tab[i/?]` denotes the tuple of rows `tab` after the removal of its i -th element (see Sec. 2.1.3)¹³¹. Note that in this case, we do not refer to any constructor of bodies because removing a row, we do not modify the table's body.

Get a row from a table

```

co-get-ro-from-tb : Yoke x CompositeE  $\mapsto$  CompositeE
co-get-ro-from-tb.(yok, com) =
  com : Error  $\rightarrow$  com
  let
    (tab, bod-t) = com
    bod-r = bo-get-ro-from-tb.bod-t
  bod-r : Error  $\rightarrow$  bod-r
  let
    (row-1, ..., row-n) = tab
  yok.(row-i, bod-r) = (tt, ('boolean'))  $\rightarrow$  (row-i, bod-r) for i = 1;n
  true  $\rightarrow$  'no-such-row'

```

First, for every row `row-i`, we create a row composite `(row-i, bod)` that consists of that row and the row body `bod` of the table. Next, we select the first of such composites that satisfies the yoke `yok`. If there is no such composite, then an error message is generated. However, if in the course of searching for a row for some index i , `yok.(row-i, bod)` turns out to be an error, then the search continues. Notice also that the constructor `bo-get-ro-from-tb` guarantees that `bod-r` is a row body. The yoke that is used to select the row will be called a *selection yoke*.

Exclude rows from a table

```

co-exclude-ro-from-tb : CompositeE x CompositeE  $\mapsto$  CompositeE
co-exclude-ro-from-tb.(com-1, com-2) =
  com-i : Error  $\rightarrow$  com-i for i = 1,2
  sort.com-i  $\neq$  'Tq'  $\rightarrow$  'table-expected' for i = 1,2
  let
    (tab-i, bod-i) = com-i for i = 1,2

```

¹³¹ Users familiar with SQL are aware of the fact that the removal of a row from a table may be either blocked by integrity constraints (subordination relation) or lead to a cascade removal of rows from subordinated tables. These mechanisms will be described on the level of denotation constructors where we can talk about subordination relations.

```

bod-1 ≠ bod-2    → 'bodies-not-compatible'
let
  new-tab  = difference.(tab-1, tab-2)
  new-com  = (new-tab, bod-1)
true         → new-com

```

(see Sec. 2.1.4)

This constructor removes all rows from the first table that belong to the second table. The result may be an empty table. Here again, we do not use any body-constructor.

The next constructor generates a table consisted of all rows of a given table that satisfy a given yoke.

Filter rows in a table

$\text{co-filter-ro-in-tb} : \text{Yoke} \times \text{CompositeE} \mapsto \text{CompositeE}$

```

co-filter-ro-in-tb.(yok, com) =
  com : Error          → com
  sort.com ≠ 'Tq'     → 'table-expected'
let
  ((row-1,...,row-n), ('Tq', row, bod-r)) = com
  fi-tuple-com-row                        = filter.yok.((row-1, bod),..., (row-n, bod))  (fi – final)
  fi-tuple-com-row = () → 'no-row-satisfies-this-condition'
let
  ((row-fi-1, bod),..., (row-fi-m, bod)) = fi-tuple-com-row
true                                → ((row-fi-1,...,row-fi-m), ('Tq', row, bod-r))

```

A tuple of row composites created from the table is filtered using a universal operation on tuples **filter** defined in Sec. 2.1.4. Rows appearing in composites of this tuple are used to create the new table. Of course, this operation does not change the table's body.

Join two tables

$\text{co-join-tb} : \text{CompositeE} \times \text{CompositeE} \mapsto \text{CompositeE}$

```

co-join-tb.(com-1, com-2) =
  com-i : Error          → com-i          for i = 1,2
let
  (tab-i, bod-i) = com-i    for i = 1,2
  bod            = bo-join-tb.(bod-1, bod-2)
  bod : Error          → bod
let
  new-tab  = join-without-repetition.(tab-1, tab-2)
  new-com  = (new-tab, bod)
  oversized.new-com → 'overflow'
true    → new-com

```

The joining of two tables results in adding to the first table all these rows of the second that do not lead to repetitions. The tables that are put together must have identical bodies. The operation on tuples **join-without-repetition** is defined in Sec 2.1.4.

Intersect two tables

$\text{co-intersect-tb} : \text{CompositeE} \times \text{CompositeE} \mapsto \text{CompositeE}$

```

co-intersect-tb.(com-1, com-2) =
  com-i : Error → com-i          for i = 1,2
  let
    (tab-i, bod-i) = com-i          for i = 1,2
    bod            = bo-join-tb.(bod-1, bod-2)
  bod : Error → bod
  let
    new-tab = common-part.(tab-1, tab-2)
    new-com = (new-tab, bod)
  true     → new-com

```

The resulting table contains only those rows that are common to both tables. The intersected tables must have identical bodies, and to check that, we use body constructor `bo-join-tb`. The operation on tuples `common-part` is defined in Sec. 2.1.4.

10.5.5 Column constructors of table composites

By a *column*, we shall mean a non-empty tuple of simple composites. We assume that columns do not contain errors, but the domain of columns does. Therefore:

$\text{col} : \text{ColumnE} = \text{SimCom}^{c^+} \mid \text{Error}$

Notice that we do not define columns composites, analogously to table composites, but columns as such. They are introduced as a technical vehicle to describe five column-oriented operations on tables.

The first four of these operations are associated with columns only implicitly since none of them neither takes a column as an argument nor returns it as a value. The fifth constructor returns columns as values but has an auxiliary character, and therefore will have no syntactic counterpart. All of them are defined in three steps according to a common rule:

1. the decomposition of a table composite into a tuple of row composites,
2. a modification of every row composite by an appropriate constructor of row composites,
3. the composition of the resulting tuple of row composites into a new table composite (constructors `add`, `cut`, `change`) or into a column (constructor `get`).

Add a column to a table

$\text{co-add-co-to-tb} : \text{Identifier} \times \text{CompositeE} \times \text{CompositeE} \mapsto \text{CompositeE}$

```

co-add-co-to-tb.(ide, com-s, com-t) =                               (s – simple, t – table)
  com-i : Error → com-i          for i = s, t
  sort.com-t ≠ 'Tq' → 'table-expected'
  let
    (tab, ('Tq', row-d, bod-r)) = com-t                               (d – default)
    (row-1, ..., row-n)         = tab
    com-j                       = (row-j, bod-r)                    for j = 1;n          (1)
    com-d                       = (row-d, bod-r)                    (2)
    new-com-j                   = co-add-to-ro.(ide, com-s, com-j)   for j = 1;n          (3)
    new-com-d                   = co-add-to-ro.(ide, com-s, com-d)
    new-com-j : Error → new-com-j   for j = 1;n
    new-com-d : Error → new-com-d
  let
    (new-row-j, new-bod-r) = new-com-j   for j = 1;n

```


$$(\text{new-row-d}, \text{new-bod-r}) = \text{new-com-d} \quad (4)$$

$$\text{new-tab} = (\text{new-row-1}, \dots, \text{new-row-n}) \quad (4)$$

$$\text{new-com-t} = (\text{new-tab}, ('Tq', \text{new-row-d}, \text{new-bod-r})) \quad (5)$$

$$\text{oversized.new-com-t} \rightarrow \text{'overflow'}$$

$$\text{true} \rightarrow \text{new-com-t}$$

This constructor adds a new column to a table by extending all its rows, including the row of default values, with the same data taken from the simple composite `com-s`. This task is done in the following steps:

1. After two routine checks, we construct a family of row composites $\{\text{com-j} \mid j=1;n\}$. Each of these composites includes one row of the table, and (shared) body-row `bod-r` of the table.
2. We construct a composite `com-d` that corresponds to the row of default values `row-d`.
3. Each of the constructed composites is extended by a new attribute in using row constructor `co-add-to-row` (Sec. 10.5.3). This constructor also checks if `com-s` is a simple composite and if `ide` does not appear in the set of attributes of the table. Composites constructed in this way have a common row body `new-bod-r`.
4. `new-tab` includes all new rows.
5. The new table composite consists of the new table and the new table body $(('Tq', \text{new-row-d}, \text{new-bod-r}))$.

Of course, this algorithm does not need to be literally performed by a future implementation of our constructor. It only defines its functionality.

Cut a column from a table

This constructor is defined analogously to the former, but this time we use the constructor `co-cut-from-ro` (announced but not defined in Sec. 10.5.1), which also checks if `ide` is an attribute of the table and if it is not the unique attribute. Of course, this time, we do not need to check for an overflow.

$$\text{co-cut-co-from-tb} : \text{Identifier} \times \text{CompositeE} \mapsto \text{CompositeE}$$

$$\text{co-cut-co-from-tb}(\text{ide}, \text{com-t}) =$$

$$\text{com-t} : \text{Error} \rightarrow \text{com-t}$$

$$\text{sort.com-t} \neq 'Tq' \rightarrow \text{'table-expected'}$$

let

$$(\text{tab}, ('Tq', \text{row-d}, \text{bod-r})) = \text{com-t}$$

$$('Rq', \text{bor}) = \text{bod-r}$$

$$(\text{row-1}, \dots, \text{row-n}) = \text{tab}$$

$$\text{com-j} = (\text{row-j}, \text{bod-r}) \quad \text{for } j = 1;n$$

$$\text{com-d} = (\text{row-d}, \text{bod-r})$$

$$\text{new-com-j} = \text{co-cut-from-ro}(\text{ide}, \text{com-j}) \quad \text{for } j = 1;n$$

$$\text{new-com-d} = \text{co-cut-from-ro}(\text{ide}, \text{com-d})$$

$$\text{new-com-j} : \text{Error} \rightarrow \text{new-com-j} \quad \text{for } j = 1;n$$

$$\text{new-com-d} : \text{Error} \rightarrow \text{new-com-d}$$

let

$$(\text{new-row-j}, \text{new-bod-r}) = \text{new-com-j} \quad \text{for } j = 1;n$$

$$(\text{new-row-d}, \text{new-bod-r}) = \text{new-com-d}$$

$$\text{new-tab} = (\text{new-row-1}, \dots, \text{new-row-n})$$

$$\text{new-com} = (\text{new-tab}, ('Tq', \text{new-row-d}, \text{new-bod-r}))$$

$$\text{true} \rightarrow \text{new-com}$$

Filter the indicated columns of a table (remove the not-indicated)

$$\text{co-filter-co-from-tb} : \text{AcPaDe} \times \text{CompositeE} \mapsto \text{CompositeE}$$

```

co-filter-co-from-tb.(apd, com) =
  apd = ()           → 'choose-an-attribute'
  com : Error       → com
  sort.com ≠ 'Tq'   → 'table-expected'
  let
    (ide-1,...,ide-n) = apd
    (tab, ('Tq', row, ('Rq', bor))) = com
  bor.ide-i = ?     → 'no-attribute-ide-i'   for i = 1;n
  let
    tab-fi = tab trun {ide-1,...,ide-n}
    row-fi = row trun {ide-1,...,ide-n}
    bor-fi = bor trun {ide-1,...,ide-n}
  true           → (tab-fi, ('Tq', row-fi, ('Rq', bor-fi)))

```

In this definition, we refer to the domain of actual parameters **AcPaDe** (Sec. 6.1.3), although now it plays a different role. Our constructor removes all columns except those whose attributes are on the list of parameters. The operator `trun` of the truncation of a function has been defined in Sec. 2.1.3. Notice that repetitions in the list of parameters do not affect the performance of our constructor.

Change a column in a table conditionally

$$\text{co-change-co-in-tb} : \text{Identifier} \times \text{CompositeE} \times \text{Transfer} \times \text{Transfer} \mapsto \text{CompositeE}$$

```

co-change-co-in-tb.(ide, com, tra) =
  com : Error       → com
  sort.com ≠ 'Tq'   → 'table-expected'
  let
    (tab, ('Tq, row, bod) = com
    (row-1,...,row-n)     = tab
    com-j                 = (row-j, bod)                       for j = 1;n
    new-com-j            = co-change-in-ro.(ide, com-j, tra, yok) for j = 1;n
  new-com-j: Error     → new-com-j                           for j = 1;n
  let
    (new-row-j, bod)     = new-com-j                         for j = 1;n
    new-tab              = (new-row-1,...,new-row-n)
    new-com              = (new-tab, ('Tq', row, bod))
  true                 → new-com

```

This constructor applies `co-change-in-ro` to each row of the table. This application does not change the table body but may generate an error message in the case of non-compatibility of bodies. A particular application of this constructor corresponds to the instruction:

```

UPDATE Employees
  SET Salary = Salary * 1,1
  WHERE Position = 'salesman'

```

The last constructor of this group selects a column from a table. Although there is probably no such constructor in the SQL standard, we introduce it for later use in the definition of yokes for tables. Its denotational counterpart does not belong to the signature of the algebra of denotations, and therefore is not represented at the level of syntax either.

Get a column from a table

$\text{co-get-co-from-tb} : \text{Identifier} \times \text{CompositeE} \mapsto \text{ColumnE}$

```

co-get-co-from-tb.(ide, com-t) =
  com-t : Error           → com-t
  sort.com ≠ 'Tq'        → 'table-expected'
  let
    (tab, ('Tq', row-d, bod-r)) = com-t
    (row-1, ..., row-n)         = tab
    com-j                       = (row-j, bod-r)           for j = 1;n
  chosen-com-j                 = co-get-from-ro.(ide, com-j) for j = 1;n
  chosen-com-j : Error → new-com-j                       for j = 1;n
  true                         → (chosen-com-1, ..., chosen-com-n)

```

This constructor creates a tuple of simple composites that correspond to attribute *ide* in each of table rows except the row of default values. Consequently, the resulting column does not contain a default composite. It will be referred to in the definition of a yoke (Sec. 10.6.2), which checks the repetition freeness of a column.

10.5.6 A referential constructor of table composites

This constructor allows for the composition of two tables into a third one. In typical applications, the source tables will be linked by a subordination relation. Still, in the definition of our constructor, we shall not use this fact, since subordination graphs will be available only at the level of denotations¹³². We start from an auxiliary constructor that gets three arguments:

1. a row composite *com-r*,
2. a subordination indicator *ide*,
3. a table composite *com-t*,

and returns a row composite that corresponds to first such a row in *com-t* that carries on *ide* the same data as is carried on *ide* by *com-r*.

The selected row of table composite *com-t* will be called a *superior row* for row composite *com-r*. Such a relation between rows is shown in Fig. 10.3-1. If the row of *com-r* belongs to a table that is subordinated to *com-t*, then a superior row always exists and is unique. In the opposite case, it may be no such row, or there may be more than one.

A table constructor indicating a superior row

$\text{indicate-sup-ro} : \text{CompositeE} \times \text{Identifier} \times \text{CompositeE} \mapsto \text{CompositeE}$

```

indicate-sup-ro.(com-r, ide, com-t) =
  com-i : Error           → com-i           for i = w, t
  sort.com-r ≠ 'Rq'      → 'row-expected'
  sort.com-t ≠ 'Tq'      → 'table-expected'
  let
    (row, ('Rq', bor-w)) = com-r
    (tab, ('Tq', row-d, ('Rq', bor-t))) = com-t
  bor-i.ide = ?          → 'unknown-attribute' for i = w, t

```

¹³² An alternative to that solution might be a third carrier in the algebra of composites — the domain of subordination graphs. I did not choose such a solution to avoid the modification of the algebra, although, frankly speaking, I am not sure which solution is better.

let

```

(row-1,...,row-n) = tab
row-j.ide = row.ide  → (row-j, ('Rq', bor-t))    for j = 1;n
true                → 'no-such-row'

```

After all necessary checks, the source-table **tab** is inspected row-by-row in searching for a row that carries in **ide** the same data as the source row of **com-r**. The first such row — if it exists — becomes the result of the computation. Otherwise, an error message is generated.

Now we are ready to define our target constructor, which gets four arguments:

1. a table composite **com-c** that “plays the role of a child table,
2. a table composite **com-p** that “plays the role” of a parent table,
3. an indicator **ide** shared by both tables,
4. a row yoke **yok**

and creates a table of such rows of **com-c** that indicate the rows of **com-p** that satisfy **yok**. The table created by this constructor will be called the *derivative table* of the two source tables.

The table constructor of derivative tables

create-der-tb : CompositeE x CompositeE x Identifier x Transfer \mapsto CompositeE

This constructor is defined by induction on the number of rows of child table. We start, therefore, from a table with one row only. For that case we define a separate constructor:

```

create-der-tb-1w.(com-c, com-p, ide, yok) =
com-i Error                → com-i    for i = c, p
sort.com-i ≠ 'Tq'         → 'table-expected'
let
  ((row), ('Tq', row-d, bod-r)) = com-c
  com-r                          = (row, bod-r)                composite created from a row
  com-rs                         = indicate-sup-ro.(com-r, ide, com-p)
com-rs : Error                → com-rs
yok.com-rs = (tt, ('boolean')) → com-c
yok.com-rs : Error            → yok.com-rs
true                         → ((), ('Tq', row-d, bod-r))

```

The earlier defined constructor **indicate-sup-ro** indicates composite **com-rs** that carries a row superior for the unique row **com-r** of the table **com-c**. If the superior row **com-rs** satisfies the yoke **yok** then the current table **com-c** becomes the result of the constructor. Otherwise:

- if the application of **yok** leads to an error then this error becomes the result,
- otherwise, i.e., if **yok** generates **(ff, ('boolean'))**, then the result is an empty table with the body identical to the body of **com-c**.

The second inductive step is the following:

```

create-der-tb.(com-c, com-p, ide, yok) =
com-i Error                → com-i    for i = p, n
sort.com-i ≠ 'Tq'         → 'table-expected'

```

```

let
  (tab, bod-r) = com-c
tab = ()           → 'empty-subordinated-table'
let
  ((row-1,...,row-k), bod-r) = com-c
  com-c-1                    = ((row-1), bod-r)
  com-rs -1                  = create-der-tb-1w.(com-c-1, ide, com-p)
com-rs -1 : Error          → com-rs-1
k = 1                      → com-rs-1
let
  com-res      = ((row-2,...,row-k), bod-r)      (res – residuum)
  com-rs -res = create-der-tb.(com-res, ide, com-p)
com-rs -res : Error → com-rs-res
true          → co-join-tb.(com-rs-1, com-rs-ind)

```

After all necessary checks, the resulting table `com-rs-1` is created for the table `com-c-1` that results from `com-c` by reducing it to only one row.

If the table `com-c` has only one row, then the computation terminates. In the opposite case, we recursively apply our constructor to the residuum of the table `com-c`, and the resulting table is “glued” using `join-tb` to the table resulting from the first row.

Notice that this constructor is defined for an arbitrary pair of source tables, i.e., not necessarily for tables linked by a subordination relation.

10.6 Yokes

Types — as understood in this book — are mentioned in SQL manuals only in the context of simple data and even in that case in a very unclear and incomplete way. Types of tables are implicit in table declarations, and types of rows, columns, and databases are absent. In table declarations, descriptions of bodies are mixed with the descriptions of yokes, and with database instructions, and are called *integrity constraints* (Sec. 9.3).

Unfortunately, in none of the manuals listed in Sec. 9, I have found a complete description of integrity constraints. Although all of them have a non-empty common part, besides that part, each manual offers different ideas. In this situation, I decided to construct such a model of SQL types that would cover a “sufficiently large” spectrum of types appearing in SQL manuals.

Since in **Lingua-SQL**, there are no database composites, there will be no database yokes either. The properties of databases will be described by:

- yokes that describe their tables,
- subordination graphs “visible” at the level of denotations.

We assume that in **Lingua-SQL**, we have earlier defined transfer constructors and yoke constructors. New constructors will generate transfers on new simple data, that we may regard as the parameters of our model, plus row- and table-transfers that are defined below.

10.6.1 A row transfer

In this group we have only one constructor which is analogous to the selection constructor for records:

$Tc[\text{co-get-from-ro}] : \text{Identifier} \mapsto \text{Transfer}$

$Tc[\text{co-get-from-ro}].\text{ide.com} = \text{co-get-from-ro}.\text{(ide, com)}$

The transfer constructed in this way selects a data and its body (a composite) assigned to an identifier in a row. For such a transfer not to generate an error, `com` must be a row composite.

The remaining constructors, including boolean constructors, may be used to create row transfers and yokes in the same way as for records.

10.6.2 Table yokes

Table yokes are split into two classes. The first contains *quantified table-yokes*, which describe table properties by row yokes that should be satisfied by all rows of a table. The second class contains column yokes.

In the first case, we have a situation analogous to the creation of a list yoke using **all-of-li**. The name of this constructor does not have the form $Tc[dco]$ since it does not refer to any data constructor.

Quantified table-yoke

all-in-tb : Yoke \mapsto Yoke

all-in-tb.yok.com =

com : Error	\rightarrow com	
sort.com \neq 'Tq'	\rightarrow 'table-expected'	
let		
((row-1,...,row-n), ('Tq', row-d, bod)) = com		
com-i	= (row-i, bod)	for i = 1;n
yok.com-i : Error	\rightarrow yok.com-i	for i = 1;n
sort.(yok.com-i) \neq ('boolean')	\rightarrow 'yoke-expected'	
(\forall com-i) yok.com-i = (tt, ('boolean'))	\rightarrow (tt, ('boolean'))	
true	\rightarrow (ff, ('boolean'))	

Notice that **yok** does not need to be satisfied by the row of default values **row-d**. This decision is due to the fact that some fields of **row-d** may be empty.

Although quantified table-yokes express properties of table rows explicitly, they express implicitly — due to quantifiers — some properties of columns. These properties may also be expressed by yokes satisfied by all the elements of a column. This technique does not allow, however, to express properties of columns regarded as a whole, e.g., that the column is ordered or that it does not contain repetitions. To express such properties, we need special column-dedicated constructors. Here is one example of such a constructor:

no-repetitions-tb : Identifier \mapsto Yoke

no-repetitions-tb.ide.com =

com : Error	\rightarrow com	
sort.com \neq 'Tq'	\rightarrow 'table-expected'	
let		
col = co-get-co-from-tb.(ide, com)		(see Sec. 10.5.5)
col : Error	\rightarrow col	
true	\rightarrow (no-repetitions.col, ('boolean'))	

We create a tuple of composites **col**, which represents the column of the attribute **ide**, and then we check if this tuple satisfies the universal predicate **no-repetitions** (Sec. 2.1.4). It is to be recalled that the created column does not contain an element from the row of default values.

Since we have boolean constructors among constructors of yokes (Sec. 4.3.4), we can use them to construct yokes that express properties of several columns of a table and all of its rows. Notice that contrary to the SQL standard, the properties of columns and rows may be combined by arbitrary boolean constructor rather than by conjunction only¹³³.

In the end, it should be pointed out that subordination relations do not appear at the level of table yokes since the subordination of one table to another one is not a property of tables but of databases. Consequently, as we are going to see in Sec. 10.11, a SQL-like declaration of a table variable will correspond in our case to

¹³³ To say the truth I am not sure if such a generalisation has a practical value.

a colloquial declaration “unfolding” in the concrete syntax to a sequence of a table-variable declaration and a database instruction.

10.7 Types

The algebra of types of **Lingua-SQL** contains six carriers:

Identifier
CompositeE
BodyE
Transfer
Yoke
TypeE

and results from the algebra of types of **Lingua-A** by extending it by a new carrier **CompositeE**, and by four groups of constructors:

1. all row-composite constructors of Sec. 10.5.3; we need them to build table types which include rows of default bodies,
2. all transfer and yoke constructors described in Sec. 10.6,
3. two constructors of row bodies (defined in Sec. 10.4),
4. one constructor of table bodies (defined in Sec. 10.4)
5. one type constructor described below.

New body constructors for rows and tables that we shall need in constructing types are the following.

bo-create-ro	: Identifier x BodyE	↦	BodyE
bo-add-to-ro	: Identifier x BodyE x BodyE	↦	BodyE
bo-create-empty-table	: CompositeE	↦	BodyE

Similarly as in Sec. 4.3.5, we define only one type constructor:

```
create-ty : BodyE x Yoke ↦ TypeE
create-ty.(bod, yok) =
  bod : Error   → bod
  true         → (bod, yok)
```

For explanations and comments see Sec. 4.3.5.

10.8 Values

10.8.1 Simple values, row values, and table values

So far, values in **Lingua** consisted of a composite and a yoke. This principle is kept in **Lingua-SQL** for values carrying simple data, rows, and tables. Still, in the case of databases, values are going to be records of tables supplemented by graphs of subordination relations (Sec. 0).

Simple values, row values, and table values in **Lingua-SQL** are understood in the same way as in the previous versions of **Lingua**, i.e., as pairs consisting of a data, and a type or — equivalently — of a composite and a yoke, where composite satisfies yoke.

In **Lingua-SQL**, we include all values and all value constructors of **Lingua-2** plus the constructors of values specific for SQL.

First constructor builds empty and yokeless values out of simple bodies:

va-empty: BodyE \mapsto ValueE

va-empty.bod =

bod : Error \rightarrow bod
not bod : SimpleBod \rightarrow 'simple-body-expected'
true \rightarrow (\emptyset , (bod, TT))

The remaining constructors have the following signatures:

Constructors of row values

va-create-ro : Identifier x ValueE \mapsto ValueE
va-add-to-ro : Identifier x ValueE x ValueE \mapsto ValueE
va-cut-from-ro : Identifier x ValueE \mapsto ValueE
va-get-from-ro : Identifier x ValueE \mapsto ValueE
va-change-in-ro : Identifier x ValueE x Yoke \mapsto ValueE

Row constructors of table values

va-create-empty-table : ValueE \mapsto ValueE
va-add-ro-to-tb : ValueE x ValueE \mapsto ValueE
va-cut-ro-from-tb : Yoke x ValueE \mapsto ValueE
va-get-ro-from-tb : Yoke x ValueE \mapsto ValueE
va-exclude-ro-from-tb : ValueE x ValueE \mapsto ValueE
va-filter-ro-in-tb : Yoke x ValueE \mapsto ValueE
va-join-tb : ValueE x ValueE \mapsto ValueE
va-intersect-tb : ValueE x ValueE \mapsto ValueE

Column constructors of table values

va-add-co-to-tb : ValueE x ValueE x ValueE \mapsto ValueE
va-cut-co-from-tb : ValueE x ValueE \mapsto ValueE
va-filter-co-from-tb : AcPaDe x ValueE \mapsto ValueE
va-change-co-in-tb : ValueE x ValueE x Yoke x Yoke \mapsto ValueE

Since all these constructors should be made transparent for errors, their definitions should be written according to the following scheme already known from Sec. 4.3.6:

1. checking if arguments are not errors,
2. performing the corresponding composite constructor on composite parts of values,
3. building a yoke of the resulting value,
4. checking if the resulting composite satisfies the resulting yoke.

The only creative part of that task involving engineering decisions is associated with defining resulting yokes (point 3.). We omit this issue to avoid going too deep into SQL mechanisms.

10.8.2 Database values

Database values are constructed in a way different from the constructors of other values. Intuitively a database consists of a record of tables, i.e., a mapping from identifiers to tables, plus a subordination relation between these tables. We shall assume that every state can carry several databases, but only one may be active at a time, i.e., it may be accessible to database operations.

To describe such a mechanism, two new notions are necessary. In the first place, we introduce the domain of *table values*:

$$\text{val: TabVal} = \{(com, yok) \mid \text{sort.com} = \text{'Tq'} \text{ and } yok.com = (tt, (\text{'boolean'}))\}$$

Of course, this domain is a subset of domain **Value**, and therefore its elements are built by constructors defined in Sec. 10.8.1. By a *database record* we shall mean a mapping that maps identifiers into table values:

$$\text{dbr : DatBasRec} = \text{Identifier} \Rightarrow \text{TabVal}$$

Of course, database records are not records in the sense of Sec. 4.3.1 but only in a set-theoretic sense.

We say that a database record *dbr* *satisfies the subordination relation* described by a subordination graph *sgr*, in symbols

dbr satisfies *sgr*,

if for every edge (*ide-c*, *ide*, *ide-p*) of this graph, the tables assigned to *ide-c* and *ide-p* are defined, and if

$$(com-c, yok-c) = \text{dbr.ide-c}$$

$$(com-p, yok-p) = \text{dbr.ide-p}$$

then the subordination relation holds, i.e.

$$com-c \text{ sub}[ide] com-p$$

By a *database value* we mean a pair consisting of a database record and a subordination graph that describes the subordination relations satisfied by that record:

$$\text{dbv : DbVal} = \{(dbr, sgr) \mid \text{dbr} \text{ satisfies } sgr\}$$

Intuitively we may say that in database values, the role of a yoke is played by the predicate satisfies. Notice, however, that since a database record carries table values, the tables of a database satisfy yokes assigned to them.

Constructors of database values will be described at the level of instructions in Sec. 10.9.6.

10.9 The algebra of denotations

As was already announced, **Lingua-SQL** will offer all programming mechanisms of **Lingua-2**, and additionally, some constructors corresponding to selected tools of SQL. In the present section, these new constructors are briefly described. Many technical details are omitted for the sake of simplicity.

10.9.1 States and denotational domains

The concept of a state in **Lingua-SQL** is analogous as before with the only difference that now the domain of values includes new sorts:

1. simple SQL values,
2. row values,
3. table values,

4. database values.

In every state, several database values may be stored, i.e., assigned to identifiers, but only one database may be active at a time. A database is said to be *active in a state* if its tables are assigned to identifiers in the valuation of that state. We assume further that every state carries four system identifiers:

- `sb-graph` — that binds the subordination graph of the active base in the environment,
- `copies` — that binds a finite set of table names (identifiers) in the valuation,
- `monitor` — that binds one table in the valuations; this table is displayed on a monitor,
- `check` — that binds words ‘yes’ or ‘no’ in valuations.

Their role will be explained later. So far, we only assume that these identifiers cannot be used as identifiers of variables, of type constants, and of procedures. The identifier `check` is called the *security flag*. If its value is ‘yes’, then we say that the *flag is up*. Otherwise, the flag *is down*.

The signature of the algebra of denotations of **Lingua-SQL** is an extension of the signature of **Lingua-2** by new constructors. There are no new carriers, but the old carriers are getting new elements.

10.9.2 Data-expression denotations

Following the rule described in Sec. 4.4.2 constructors of data-expression denotations of **Lingua-SQL** split into four categories:

1. one constructor of variables,
2. constructors derived from non-boolean value constructors; for each such constructor `vco`, we define a constructor of denotations `Cdd.[vco]` (`Cdd` stands for *constructor of data-expression denotations*),
3. boolean constructors,
4. one constructor that corresponds to conditional expressions.

Constructors of groups 1., 3., and 4. plus these constructors of group 2. that correspond to non-SQL value constructors remain the same as in **Lingua-2**. Therefore, all we have to define are constructors derived from SQL-value constructors except database constructors, that will appear at the level of instructions.

Specific SQL constructors of expression denotations will be derived from specific SQL constructors of values in a way similar — but not quite analogous — to that described in Sec. 4.4.2., i.e., according to the scheme (4.4.2-1). The “not quite analogous” means that in table expressions which take table values as arguments, tables must be represented by identifiers, rather than by expressions. For instance, with the constructor of table values:

$$\text{va-add-ro-to-tb} : \text{ValueE} \times \text{ValueE} \mapsto \text{ValueE}$$

we associate the following constructor of data-expression denotations:

$$\text{Cdd}[\text{va-add-ro-to-tb}] : \text{DatExpDen} \times \text{Identifier} \mapsto \text{DatExpDen}$$

$$\begin{aligned} \text{Cdd}[\text{va-add-ro-to-tb}].(\text{ded}, \text{ide}).\text{sta} = \\ \text{va-add-ro-to-tb}.\text{(ded.sta, dat-variable.ide.sta)} \end{aligned}$$

where the source row is `ded.sta`, and the source table is `dat-variable.ide.sta`. This decision is of engineering character and is assumed to simplify syntax analysis. It also seems conformant with SQL standards.

Since the definitions of all these constructors correspond to the scheme (4.4.2-1), we shall not repeat them here. We only show signatures of constructors, which we shall need to generate the syntax of **Lingua-SQL**.

The constructor of empty-value expression denotations

$$\text{Cdd}[\text{va-empty.bod}] : \text{BodExpDen} \mapsto \text{DatExpDen}$$

Constructors of row-expression denotations

$\text{Cdd}[\text{va-create-ro}]$: Identifier x DatExpDen	\mapsto DatExpDen
$\text{Cdd}[\text{va-add-to-ro}]$: Identifier x DatExpDen x DatExpDen	\mapsto DatExpDen
$\text{Cdd}[\text{va-cut-from-ro}]$: Identifier x DatExpDen	\mapsto DatExpDen
$\text{Cdd}[\text{va-get-from-ro}]$: Identifier x DatExpDen	\mapsto DatExpDen
$\text{Cdd}[\text{va-change-in-ro}]$: Identifier x DatExpDen x Transfer	\mapsto DatExpDen

Row constructors of table-expression denotations

$\text{Cdd}[\text{va-create-empty-table}]$: DatExpDen x Transfer	\mapsto DatExpDen
$\text{Cdd}[\text{va-add-ro-to-tb}]$: DatExpDen x Identifier	\mapsto DatExpDen
$\text{Cdd}[\text{va-cut-ro-from-tb}]$: Transfer x Identifier	\mapsto DatExpDen
$\text{Cdd}[\text{va-get-ro-from-tb}]$: Transfer x Identifier	\mapsto DatExpDen
$\text{Cdd}[\text{va-exclude-ro-from-tb}]$: Identifier x Identifier	\mapsto DatExpDen
$\text{Cdd}[\text{va-filter-ro-in-tb}]$: Transfer x Identifier	\mapsto DatExpDen
$\text{Cdd}[\text{va-join-tb}]$: Identifier x Identifier	\mapsto DatExpDen
$\text{Cdd}[\text{va-intersect-tb}]$: Identifier x Identifier	\mapsto DatExpDen

Column constructors of table-expression denotations

$\text{Cdd}[\text{va-add-co-to-tb}]$: Identifier x DatExpDen x Identifier	\mapsto DatExpDen
$\text{Cdd}[\text{va-cut-co-from-tb}]$: Identifier x Identifier	\mapsto DatExpDen
$\text{Cdd}[\text{va-filter-co-from-tb}]$: AcPaDe x Identifier	\mapsto DatExpDen
$\text{Cdd}[\text{va-change-co-in-tb}]$: Identifier x Identifier x Transfer x Transfer	\mapsto DatExpDen

The constructor of the denotation of an expression that creates a derivative table

$\text{Cdd}[\text{va-create-der-tb}]$: Identifier x Identifier x Identifier x Transfer	\mapsto DatExpDen
---------------------------------------	---	---------------------

10.9.3 Type-expression denotations

Similarly as in **Lingua-2** (Sec.4.4.2) the algebra of type-expression denotations of **Lingua-SQL** contains five carriers:

ide	: Identifier	= ...	
bed	: BodExpDen	= State \mapsto BodyE	body-expression denotations
tra	: TraExpDen	= Transfer	transfer-expression denotations
yok	: YokExpDen	= Yoke	yoke-expression denotations
ted	: TypExpDen	= State \mapsto TypeE	type-expression denotations

According to the rules described in Sec.4.4.2, the denotations of transfer- and yoke-expressions are simply transfers and yokes. It is so because transfers and yokes are not saved in states. To the constructors of **Lingua-2** we add:

1. constructors of SQL transfers and yokes defined in Sec. 10.6,

2. selected row constructors of SQL data-expression denotations (Sec. 10.9.2); they are needed to creates rows of default values,
3. constructors of specific SQL type-expression denotations derived from SQL type-constructors (Sec. 10.7).

These assumptions lead to the following list of new constructors in the algebra of denotations of **Lingua-SQL**.

The constructors of denotations of transfer- and yoke expressions

Tc[co-get-from-ro] : Identifier \mapsto TraExpDen
 all-in-tb : YokExpDen \mapsto YokExpDen
 no-repetitions-tb : \mapsto YokExpDen

The constructors of denotations of body expressions

I recall that Cbd is a meta-constructor which transforms body-constructors into constructors of the denotations of body expressions.

Cbd[bo-create-ro] : BodExpDen x Identifier \mapsto BodExpDen
 Cbd[bo-add-to-ro] : BodExpDen x Identifier x BodExpDen \mapsto BodExpDen
 Cbd[bo-create-empty-table] : DatExpDen \mapsto BodExpDen

10.9.4 Denotations of type-constant declarations

New declarations of type constants in **Lingua-SQL** are the declarations that refer to new type constructors (Sec. 10.7) and to new type of declarations related to the modifications of subordination graphs.

Since the declarations of the first group coincide with the general scheme described in Sec. 5.1.4.2, we shall not repeat them here.

The constructors related to the subordination graphs do not belong to that group since they do not create any type constant but only change the subordination graph assigned to the system identifier **sb-graph**. These constructors will appear only on the level of database instructions in Sec. 10.9.6.11.

The only constructor related to subordination graphs is therefore the following:

add-sub-type : Identifier x Identifier x Identifier \mapsto TypDecDen

add-sub-type.(ide-c, ide, ide-p).sta =

is-error.sta \rightarrow sta

ide-c = ide-p \rightarrow sta \leftarrow 'reflexivity-not-allowed'

let

((tye, pre), skl) = sta

sgr = tye.sb-graph

(ide-c, ide, ide-p) : sgr \rightarrow 'redundant-subordination'

let

new-sgr = sgr | {(ide-c, ide, ide-p)}

true \rightarrow ((tye[sb-graph/new-sgr], pre), sto)

This constructor builds a denotation that extends the subordination graph and updates the type environment. At the stage of type declarations I do not introduce the constructor of removing edges from a graph since this is an operation from the level of database instructions. It will appear, therefore in Sec. 10.9.6.11.

10.9.5 Denotations of data-variable declarations

Variables in **Lingua-SQL** may be bound to all values of **Lingua-2** and additionally to four groups of specific SQL-values:

1. simple SQL-values,
2. row values,
3. table values,
4. database values.

The declarations of variables of first two groups coincide with the general scheme of such declarations in **Lingua-2** (Sec. 5.1.2). In **Lingua-SQL** there are no declarations of database variables, but instead, we have a specific instruction of database archivation by assigning it to an indicated identifier (Sec. 10.9.6.11). The constructor of table-variable declarations is defined in a way slightly different than in **Lingua-2**:

$\text{declare-tab-var} : \text{Identifier} \times \text{TypExpDen} \mapsto \text{VarDecDen}$

$\text{declare-tab-var}.\text{(ide, ted).sta} =$

$\text{is-error.sta} \quad \rightarrow \text{sta}$

$\text{declared.ide.sta} \quad \rightarrow \text{sta} \leftarrow \text{'variable-declared'}$

let

$\text{typ} = \text{ted.sta}$

$\text{typ} : \text{Error} \quad \rightarrow \text{sta} \leftarrow \text{typ}$

let

$\text{(bod, yok)} = \text{typ}$

$\text{sort.bod} \neq \text{'Tq'} \quad \rightarrow \text{sta} \leftarrow \text{'table-type-expected'}$

let

$\text{val} \quad = \text{(()}, \text{typ})$

$\text{(env, (vat, 'OK'))} \quad = \text{sta}$

true $\quad \rightarrow \text{(env, (vat[ide/val], 'OK'))}$

The difference of this declaration from the standard of Sec. 5.1.2 consists in the fact that in the present case, a variable is bound to an empty table $((), \text{typ})$, rather than to a pseudo value (Ω, typ) . And, of course, we also check if typ is a table type.

10.9.6 Instructions

10.9.6.1 Categories of SQL instructions

The carrier of instruction denotations in the algebra of denotations of **Lingua-SQL** is enriched with new constructors of specific SQL instructions of three categories;

1. row assignments,
2. table assignments,
3. database instructions.

All constructors of **Lingua-2** are still available and applicable to the extended carrier of instruction denotations. This rule concerns, in particular, the constructor of yoke replacement and the constructors of structured

instruction, i.e., sequential composition, branching, and loop. The constructors of procedure declaration and procedure call remain unchanged as well, although now they are defined on extended domains.

A particular role in SQL is played by a large group of table assignments where we distinguish two categories:

1. *table-modification instruction* where on both sides of the assignment, we have the name of the same table,
2. *table-creation instruction* where on the left-hand side of the instruction we may have a different table name (the name of the table that is being created) than on the right-hand side.

From a mathematical perspective, the first category may be regarded as a particular case of the second. Still, denotationally they correspond to two different constructors of the algebra of denotations hence also to different constructors of the algebra of syntax. The reason for using two different constructors will be explained later.

Independently of the categorization described above, table assignments are split into two further categories according to two ways of using subordination constraints (cf. Sec. 9.5):

1. *conformist instructions* where an execution terminates with an error message whenever it would lead to a violation of subordination constraints; this category corresponds to the option RESTRICT,
2. *correcting instructions* which in the described situation introduce such changes to a hosting database that guarantee the protection of subordination constraints; this category corresponds to the option CASCADE.

If I understood it correctly from the manuals quoted at the beginning of Sec. 9.1, the first option is (most frequently?) the default option whereas the second has to be declared explicitly and is available only for a group of chosen instructions, e.g., when a row is removed from a table.

10.9.6.2 Row instructions

Row instructions create and modify row values assigned to identifiers in states. We build them using the assignment constructor defined in Sec. 5.1.2 and the constructor of data expressions denotations that return row values described in Sec. 10.9.2.

10.9.6.3 Two universal constructors of table assignment

In **Lingua-SQL**, we have two assignment constructors that correspond respectively to assigning a table to a table-variable and to assigning a table to the system-identifier monitor.

In the first case, we could use the general constructor defined in Sec. 5.1.2 unless the assignment modifies an existing table in a way that violates the subordination relation. To cope with the latter case, we have to introduce a database-oriented constructor of assignments. As we are going to see, it will become a convenient tool for the definitions of many other table assignments. Since, however, there is no such constructor in the SQL standard, the issue of making it available at the level of the syntax of **Lingua-SQL** I leave open so far.

The second universal constructor of table assignments will be used in the definitions of query denotations.

To define the first constructor, we introduce two auxiliary functions called *violation-control functions*. The first of them checks if a given identifier points in the current state to a table that violates one of the declared subordination relations.

violated-sr : Identifier x State \mapsto {tt, ff} | Error

violated-sr.(ide, sta) =

is-error.sta \rightarrow error.sta

let

((tye, pre), (vat, 'OK')) = sta

```

    sgr = tye.sb-graph
  vat.ide = ?           → 'no-such-table'
  sort.(vat.ide) ≠ 'Tq' → 'table-expected'
  (∃ (ide-c, ide-id, ide-p) : sgr)
    [vat.ide-i = ! and sort.(vat.ide-i) = 'Tq'] for i = p, n and
    ( (ide = ide-c and not vat.ide sub[ide-id] vat.ide-c) or
      (ide = ide-p and not vat.ide-c sub[ide-id] vat.ide) )
      → tt
true           → ff

```

This function returns `tt` if the table `vat.ide` does not satisfy the subordination-condition indicated by the edge `(ide, ide-id, ide-p)` or by `(ide-c, ide-id, ide)`.

The second function is similar to the first one and is used to check if a given composite satisfies a given table yoke. Notice that the checking may be deactivated by setting the flag `check` to `'not'`. In this case, we implement the mechanism of a temporary deactivation of integrity constraints described by yokes. I wish to emphasize that I have not introduced such an option for integrity constraints described by subordination relation¹³⁴.

`violated-yo : Composite x Yoke x State ↦ {tt, ff} | Error`

`violated-yo.(com, yok, sta) =`

`is-error.sta` → `error.sta`

let

`(env, (vat, 'OK')) = sta`

`vat.check = 'not'` → `ff` (check is a system identifier; Sec. 10.2)

`yok.com = (tt, ('boolean'))` → `ff`

true → `tt`

As we see, checking if a yoke has been violated is performed only if the flag is up (set into `'yes'`). In fact, this is the only role played by the state argument of this yoke constructor. Notice also that this function returns `tt` (signals the violation of the yoke) also, if the checking result `tra.com` is an error. Now we are ready to define the assignment constructor for tables.

`assign-tb : Identifier x DatExpDen ↦ InsDen`

`assign-tb.(ide, ded).sta =`

`is-error.sta` → `sta`

`vat.ide = ?` → `sta` ◀ `'undeclared-identifier'`

`ded.sta = ?` → `?`

¹³⁴ I assume that if the violation of the subordination relation is in danger, e.g. between the deletion of one column and the insertion of another, then the programmer should introduce two instructions into the program that modify the relation accordingly.

```

ded.sta : Error                → sta ◀ ded.sta
let
  ((tye, pre), (vat, 'OK')) = sta
  (dat-n, bod-n, yok-n)     = ded.sta                (n – new)
  (dat-f, bod-f, yok-f)     = vat.ide                (f– former)
sort.bod-n ≠ 'Tq'             → sta ◀ 'table-expected'
sort.bod-f ≠ 'Tq'             → sta ◀ 'table-expected'
not bod-n coherent bod-f    → sta ◀ 'no-coherence'
let
  sta-n = (env, (vat[ide/(dat-n, bod-n, yok-n)], 'OK'))
violated-yo.((dat-n, bod-n), yok-f, sta-n) → sta ◀ 'table-yoke-violated'
violated-sr.(com-n, sta-n)                → sta ◀ 'subordination-relation-violated'
true                                     → sta-n

```

As a result of the execution of such an assignment, the identifier `ide` is bound to a new value $((\text{dat-n}, \text{bod-n}), \text{yok-f})$ under the condition that:

1. both `mon-f` and `com-n` are table composites and are mutually coherent,
2. new composite satisfies in the new state the inherited (former) yoke `yok-f` unless the check-flag is set to 'not',
3. new composite does not violate in the new state the current subordination relation.

Notice that the violation of a subordination relation may only happen if the assignment modifies an existing table.

The second specific assignment constructor corresponds to the situation when a table which is defined by a table expression is assigned to the system identifier `monitor`, which physically means that it is displayed on a monitor. In that case, the new table is not restricted by any yoke, which is expressed by the fact that its yoke is `TT`.

```

assign-mo : DatExpDen ↦ InsDen
assign-mo.ded.sta =
  is-error.sta    → sta
let
  (env, (vat, 'OK')) = sta
  val                = ded.sta
val : Error        → sta ◀ com
sort.val ≠ 'Tq'    → 'table-expected'
let
  (com, yok) = val
true          → (env, (vat[monitor/(com, TT)], 'OK'))

```


As we see, in this case, we do not expect the identifier `monitor` to be declared. As a system identifier, it is always available and may be bound to an arbitrary table value. If, at the time of the execution of the described assignment, some value is already assigned to the `monitor`, then it is overwritten by the new value.

10.9.6.4 Transactions

Transactions, similarly to instructions, are state transformations, but contrary to the former, they are total functions since they do not contain loops and procedure calls. Moreover, they do not create new tables but only modify the existing ones. Their domain is, therefore, the following:

$$\text{trd} : \text{TrnDen} = \text{State} \mapsto \text{State}$$

Transactions are regarded as a separate carrier of our algebra to avoid the use of arbitrary table instructions in the contexts of transactions.

The largest group of transactions are *table modifications* which in a traditional syntax could have the form:

$$\text{ide} := \text{table-expression}(\text{ide})$$

where on both sides, we have the same table named `ide`. The denotations of such assignments are created as combinations of a table assignment (Sec. 10.9.6.3) and some denotation of a table expression or a transfer expression. Below there is a list of such transactions that are related to data expressions described in Sec. 10.9.2. First one corresponds to adding a row to a table:

$$\text{add-ro} : \text{Identifier} \times \text{DatExpDen} \mapsto \text{TrnDen}$$

$$\text{add-ro}(\text{ide}, \text{ded-r}) =$$

$$\text{assign-tb}(\text{ide}, \text{Cdd}[\text{va-add-ro-to-tb}](\text{dat-variable.ide}, \text{ded-r}))$$

The execution of this constructor creates a transaction-denotation, which to the table carried by the identifier `ide` adds a row generated by the denotation `ded-r`. Let us read that definition in details:

The table assignment constructor `assign-tb` receives as its first argument the identifier of the table that is being modified and as the second — the expression denotation generated by the constructor `Cdd[va-add-ro-to-tb]`. The arguments of the latter are two expression denotations:

- the denotation of the variable `dat-variable.ide`, which identifies the modified table,
- the denotation of a row expression `ded-r`, which generates the row which is to be added to the table.

In an analogous way we may define constructors related to table-modifications:

$$\text{cut-ro} : \text{Identifier} \times \text{Transfer} \mapsto \text{TrnDen}$$

$$\text{cut-ro}(\text{ide}, \text{tra}) =$$

$$\text{assign-tb}(\text{ide}, \text{Cdd}[\text{va-cut-ro-from-tb}](\text{tra}, \text{dat-variable.ide}))$$

$$\text{exclude-ro} : \text{Identifier} \times \text{Identifier} \mapsto \text{TrnDen}$$

$$\text{exclude-ro}(\text{ide-1}, \text{ide-2}) =$$

$$\text{assign-tb}(\text{ide-1}, \text{Cdd}[\text{va-exclude-ro-from-tb}](\text{dat-variable.ide-1}, \text{dat-variable.ide-2}))$$

$$\text{add-co} : \text{Identifier} \times \text{DatExpDen} \times \text{Identifier} \mapsto \text{TrnDen}$$

$$\text{add-co}(\text{ide-c}, \text{ded}, \text{ide-t}) =$$

(c - column, t - table)

$$\text{assign-tb}(\text{ide-t}, \text{Cdd}[\text{va-add-co-to-tb}](\text{ide-c}, \text{ded}, \text{dat-variable.ide-t}))$$

cut-co : Identifier x Identifier \mapsto TrnDen
 cut-co.(ide-c, ide-t) =
 assign-tb.(ide-t, Cdd[va-cut-co-from-tb].(ide-c, dat-variable.ide-t))

filter-co : AcPaDe x Identifier \mapsto TrnDen
 filter-co.(apd, ide) =
 assign-tb.(ide, Cdd[va-filter-co-from-tb].(apd, dat-variable.ide))

change-co : Identifier x Identifier x Transfer \mapsto TrnDen
 change-co.(ide-c, ide-t, tra) =
 assign-tb.(ide-t, Cdd[va-change-co-in-tb].(ide-c, dat-variable.ide-t, tra))

The second group of transactions consist of *protection commands* used to protect a table against destruction.

Create a security copy

create-security-copy: Identifier \mapsto State \mapsto State
 create-security-copy.sta =
 is-error.sta \rightarrow sta
 let
 (env, (vat, 'OK')) = sta
 base = vat trun {ide | vat.ide : TabVal}
 sgr = tye.sb-graph
 copy-register = vat.copies | {ide}
 vat.ide = ! \rightarrow 'variable-declared'
 true \rightarrow (env, (vat[ide/(base, sgr), copies/copy-register], 'OK'))

This function creates a database that consists of:

- all table values that appear in the current valuation,
- and of the current subordination graph,

and assigns this database to the identifier *ide*. This identifier is added to the register of copies assigned to the system identifier *copies*.

I recall that *trun* denotes the truncation of a function to a subset of its domain (Sec. 2.1.3).

Remove the security copy

remove-security-copy : Identifier \mapsto TrnDen
 remove-security-copy.ide.sta =
 is-error.sta \rightarrow sta
 let

```

(env, (vat, 'OK')) = sta
copy-register    = vat.copies - {ide}
vat.ide = ?      → 'unknown-identifier'
not vat.ide : DbVal → 'database-expected'
true         → (env, (vat[ide/? , copies/copy-register], 'OK'))

```

The copy of the base is removed from the valuation, and its name is removed from the copy register.

Recover the security copy

```

recover-security-copy : Identifier ↦ TrnDen
recover-security-copy.ide.sta =
  is-error.sta      → sta
let
  (env, (vat, 'OK')) = sta
  vat.ide = ?        → 'unknown-identifier'
  not vat.ide : DbVal → 'database-expected'
let
  (dbr, sgr)    = vat.ide
  copy-register = vat.copies - {ide}
true         → (env, (vat[ide/? , copies/copy-register] ♦ dbr, 'OK'))

```

The database `dbr` carried by the identifier `ide` is a mapping that assigns database values to identifiers. This mapping overwrites the current valuation from which we have removed the base carried by `ide`. The name of the removed copy is also removed from the copy register.

Recover the security-copy conditionally

```

recover-security-copy-if : DatExpDen x Identifier ↦ TrnDen
recover-security-copy-if.(ded, ide) = if-error.(ded, 'recover-security-copy'.ide)

```

If `ded` generates an error, then the recovery procedure is executed. In this case, we use the constructor `is-error` (Sec. 5.1.5.4). This use is not entirely formal since the second argument of `is-error` should be an instruction denotation, whereas in our case, this is a transaction denotation. However, since set-theoretically transaction denotations belong to the domain of instruction denotations, our definition makes sense.

The following two constructors are used to set the flag assigned to the system identifier check.

Set the security-flag down

```

security-flag-down : ↦ TrnDen
security-flag-down.().sta =
  is-error.sta      → sta

```

let

```

(env, (vat, 'OK')) = sta
vat.check = 'not' → 'security-flag-is-down'
true → (env, (vat[check/'not'], 'OK'))

```

This constructor generates an error if the flag is already down. This solution is, of course, not a mathematical necessity but an engineering decision that protects a programmer from committing a mistake. If he/she wants to set down a flag that is already down, then maybe he/she does not quite understand the functionality of his program. The second constructor of this group sets the flag up.

Set the security-flag up

```

security-flag-up : ↦ TrnDen
security-flag-up().sta =
  is-error.sta → sta
let
  (env, (vat, 'OK')) = sta
  vat.check = 'yes' → 'security-flag-is-up'
true → (env, (vat[check/'yes'], 'OK'))

```

Similarly as for instructions also for transactions we may apply a sequential composition:

```

sequence-trn : TrnDen x TrnDen ↦ TrnDen
sequence-trn.(trd-1, trd-2) = trd-1 • trd-2

```

The last constructor related to transactions creates an instruction from a transaction. For its definition, we shall need a function that removes all security copies. Its definition is, of course, recursive:

```

remove-all-security-copies : ↦ TrnDen
remove-all-security-copies().sta =
  is-error.sta → sta
let
  ((ten, pre), (vat, 'OK')) = sta
  sgr = ten.sb-graph
  sgr = ∅ → sta
  {ide} = sgr → remove-security-copy.ide.sta
  {ide-1, ..., ide-n} = sgr → remove-security-copy.ide • remove-all-security-copies.()

```

The constructor which transforms a transaction into an instruction is defined as follows:

$\text{trn-into-ins} : \text{TrnDen} \mapsto \text{InsDen}$

$\text{trn-into-ins.trd.sta} =$

$\text{is-error.sta} \rightarrow \text{sta}$

$\text{true} \rightarrow [\text{remove-all-security-copies}(). \bullet \text{security-flag-up}().] . \text{sta}$

This constructor is used to transform a block of transactions (maybe a one-element block) into an instruction. This constructor also removes all security copies and sets the security flag up.

As we see, the mechanism of transactions is used to the executions of such table modifications that allow for a temporary deactivation of integrity checks and of the mechanism of security copies.

10.9.6.5 Global table instructions

A *global table-instruction* is an instruction which when modifying a table modifies at the same time other tables to protect integrity constraints of a database. E.g. in SQL-standard if we remove a row from a parent table in the CASCADE mode (Sec. 9.5) then this may cause the removal of all rows from a child table which point to the removed row in the parent table. In that case “cascade” means that if a child table is a parent table for other tables, then this may result in the removals of rows from the other tables. The scheme of a definition of such a constructor is shown below:

$\text{cut-ro-cas} : \text{Identifier} \times \text{Transfer} \mapsto \text{InsDen}$

$\text{cut-ro-cas}(\text{ide}, \text{tra}).\text{sta} =$

$\text{is-error.sta} \rightarrow \text{sta}$

$\text{vat.ide} = ? \rightarrow \text{sta} \blacktriangleleft \text{'undeclared-identifier'}$

let

$(\text{env}, (\text{vat}, \text{'OK})) = \text{sta}$

$(\text{com-t}, \text{yok}) = \text{vat.ide}$

$\text{sort.com-t} \neq \text{'Tq'} \rightarrow \text{'table-expected'}$

let

$\text{com-n} = \text{co-cut-ro-from-tab}(\text{com-t}, \text{tra})$

$\text{com-n} : \text{Error} \rightarrow \text{sta} \blacktriangleleft \text{com-n}$

let

$\text{sta-n} = (\text{env}, (\text{vat}[\text{ide}/(\text{com-n}, \text{yok})], \text{'OK'}))$

$\text{violated-yo}(\text{com-n}, \text{yok}, \text{sta-n}) \rightarrow \text{sta} \blacktriangleleft \text{'table-yoke-violated'}$

$\text{violated-sr}(\text{com-n}, \text{sta-n}) \rightarrow \text{remove-integrity-violations.sta-n}$

true $\rightarrow \text{sta-n}$

This instruction removes a row from a table by using the composite constructor `co-cut-ro-from-tab`. Then if table yoke has not been violated, but the integrity constraints has, then it activates the procedure `remove-integrity-violations`. I do not define this procedure explicitly and regard it as a model parameter. Its definition would lead to technical considerations on searching procedures of subordination graphs, which would lead out of the scope of this book.

10.9.6.6 Local table instructions

The instructions of this group change only the table they concern. They either create a new table or modify an existing one using the universal table assignment (Sec. 10.9.6.3) and the denotations of table expressions (Sec. 10.9.2). In principle, we could avoid introducing such instructions into our model by allowing table assignments in the language. Since, however, there are no such assignments in SQL (which does not mean that we have to exclude them from **Lingua-SQL**), I give below some examples of the constructors of local table instructions.

$\text{add-ro} : \text{DatExpDen} \times \text{Identifier} \mapsto \text{InsDen}$

$\text{add-ro}(\text{ded}, \text{ide},) =$
 $\text{assign-tb}(\text{ide}, \text{Cdd}[\text{co-add-ro-to-tb}].(\text{ded}, \text{ide}))$

$\text{join} : \text{Identifier} \times \text{Identifier} \times \text{Identifier} \mapsto \text{InsDen}$

$\text{join}(\text{ide-n}, \text{ide-1}, \text{ide-2}) =$ (n – new table)
 $\text{assign-tb}(\text{ide-n}, \text{Cdd}[\text{co-join-tb}].(\text{dat-variable.ide-1}, \text{dat-variable.ide-2}))$

$\text{intersect} : \text{Identifier} \times \text{Identifier} \times \text{Identifier} \mapsto \text{InsDen}$

$\text{intersect}(\text{ide-p}, \text{ide-1}, \text{ide-2}) =$
 $\text{assign-tb}(\text{ide-p}, \text{Cdd}[\text{co-intersect-tb}].(\text{dat-variable.ide-1}, \text{dat-variable.ide-2}))$

$\text{create-ref} : \text{Identifier} \times \text{Identifier} \times \text{Identifier} \times \text{Identifier} \times \text{Transfer} \mapsto \text{InsDen}$

$\text{create-ref}(\text{ide-n}, \text{ide-t1}, \text{ide-t2}, \text{ide-c}, \text{tra}) =$ (c – column)
 $\text{assign-tb}(\text{ide-n}, \text{Cdd}[\text{create-der-tb}].(\text{dat-variable.ide-t1}, \text{dat-variable.ide-t2}, \text{ide-c}, \text{tra}))$

$\text{change-co} : \text{Identifier} \times \text{CompositeE} \times \text{Yoke} \times \text{Yoke} \mapsto \text{CompositeE}$

$\text{change-co}(\text{ide}, \text{val}, \text{yok-1}, \text{yok-2}) =$
 $\text{assign-tb}(\text{ide}, \text{Cdd}[\text{va-change-co-in-tb}].(\text{ide}, \text{val}, \text{yok-1}, \text{yok-2}))$

10.9.6.7 Queries

Queries are similar to simple instructions with the difference that they always create a new table assigned to the system-identifier **monitor**. Consequently, we apply simplified assignments **assign-mo** that never violates any constraints since the transfer of the new value is **TT**.

10.9.6.8 Transfer-replacement instructions

The definition of the constructor of that group, that has been defined in Sec. 5.1.5.3, is

$\text{replace-yo} : \text{Identifier} \times \text{TraExpDen} \mapsto \text{InsDen},$

and applies directly to the SQL case without any changes. Of course, we have to extend the domain of transfer-expression denotations.

10.9.6.9 Cursors

Cursors (Sec. 9.10) are mechanisms used to get row-by-row from tables. In our model, that can be easily defined, e.g., by adding a column to a table that enumerates its rows.

10.9.6.10 Views

Views are virtually procedures that call table instructions. They may be introduced to our model either as predefined instruction or by providing programming mechanisms of procedures that operate on tables.

10.9.6.11 Database instructions

I assume that in **Lingua-SQL**, an initial valuation of program execution may carry some variables assigned to database values.

I assume additionally that in every initial state of program execution, the system identifiers are bound to the following default values:

```

tye.sb-graph   = ∅
vat.copies    = ∅,
vat.monitor   = Ω                               (interpreted as no data to be displayed)
vat.check     = 'yes'

```

With these assumptions, each database program in **Lingua-SQL** that is supposed to operate on tables either has to create its own tables — and a database thereof — or to import an already existing database. In **Lingua-SQL**, we have, therefore, only two database instructions that operate on tables and besides two instructions that modify a subordination graph. Their constructions are defined below in a simplified form to avoid too many technical details.

Database activation

activate : Identifier \mapsto InsDen

activate.ide.sta =

is-error.sta \rightarrow sta

let

((tye, pre), (vat, 'OK')) = sta

tye.sb-graph = ! \rightarrow sta \leftarrow 'active-base-already-exists'

vat.ide = ? \rightarrow sta \leftarrow 'unknown-variable'

not vat.ide : DbVal \rightarrow sta \leftarrow 'database-expected'

let

(dbr, sgr) = vat.ide

true \rightarrow ((tye[sb-graph/sgr], pre), (vat \blacklozenge dbr, 'OK'))

This instruction overwrites the current valuation by a database record, which means that it stores in it table identifiers assigned to table values, and to the system variable **sb-graph** assigns the subordination graph of the activated base. Of course, it also checks whatever has to be checked. It does not allow us to create two databases at the same time (an engineering decision). I recall that **DbVal** is the domain of database values defined in Sec. 0.

The remaining database instruction writes all current table values in the database of the given name and removes from valuation all values except database values.

```

archive : Identifier  $\mapsto$  InsDen
archive.ide.sta =
  is-error.sta       $\rightarrow$  sta
  let
    ((tye, pre), (vat, 'OK')) = sta
  tye.sb-graph = ?  $\rightarrow$  sta  $\leftarrow$  'no-base-to-be-archived'
  vat.ide = !       $\rightarrow$  sta  $\leftarrow$  'variable-declared'
  dbr           = tables-only.vat
  new-vat       = remove-non-database.vat
  dbv           = (dbr, tye.sb-graph)
  true          $\rightarrow$  ((tye, pre), (new-vat[ide/dbv], 'OK'))

```

In this definition, I use two auxiliary functions `tables-only` and `remove-non-database`, whose obvious definitions are omitted. I also assume that the instruction does not allow to overwrite an existing database by a new database. It is, of course, an engineering decision.

In this definition, one might include a principle that tables which are considered as “temporary” are not subject to archiving. To do that, we could assume that, e.g., their identifiers are somehow labelled.

Notice that database archiving that assigns a database to an identifier does not require that this identifier has been declared.

Constructors that generate instructions that modify subordination graphs correspond to adding and removing an edge of a graph.

```

declare-subordination : Identifier x Identifier x Identifier  $\mapsto$  InsDen
declare-subordination.(ide-c, ide, ide-p).sta =
  is-error.sta       $\rightarrow$  sta
  let
    ((tye, pre), env, (vat, 'OK')) = sta
  vat.ide-i = ?       $\rightarrow$  'no-such-table'                for i = c, p
  let
    (com-i, tra-i) = vat.ide-l                            for i = c, p
  sort.com-i  $\neq$  'Tq'  $\rightarrow$  'table-expected'
  let
    ((tab-i, ('Tq', row-d-i, ('Rq', ror-i)), yok-i) = vat.ide-i for i = c, p
  ror-i.ide = ?       $\rightarrow$  'no-such-column'            for i = c, p
  let
    sgr = tye.sb-graph
  (ide-c, ide, ide-p) : sgr  $\rightarrow$  'redundant-declaration'
  com-c Sub[ide] com-n  $\rightarrow$  sgr | {(ide-c, ide, ide-p)}

```


true → 'subordination-not-satisfied'

Before adding a new edge to a subordination graph, this instruction checks if the subordination holds. If the concerned tables are large, then this check may be computationally expensive. This action, however, cannot be avoided if we want to protect database integrity.

The second operation does not require such a check since it only removes an edge from a subordination graph.

call-off-subordination : Identifier x Identifier x Identifier \mapsto InsDen

call-off-subordination.(ide-c, ide, ide-p).sta =

is-error.sta → sta

let

((tye, pre), (vat, 'OK')) = sta

sgr = vat.sb-graph

(ide-c, ide, ide-p) : sgr → sgr – {(ide-c, ide, ide-p)}

true → 'no-such-subordination'

10.10 Concrete syntax

For a reader who reached this section designing a concrete syntax of **Lingua-SQL** should be relatively easy. Therefore I restrict further investigations to grammatical clauses related to SQL. The syntax, which is described below, is probably not very optimal since it contains rather long keywords. My goal is, however, not to build a „practical” language but only to show a method of building such a language. For the same reason, my concrete syntax is not very close to the SQL standard. Long keywords correspond directly to the names of constructors, which should help the reader to understand their meaning.

It is worth noticing that compared to **Lingua-2**, we now have a new syntactic category of transactions. The keywords **ed**, **et**, and **ei** are read respectively as „end of declaration”, „end of transaction” and „end of instruction”.

Data expressions

dae : DatExp =

...

(here stand are all clauses of **Lingua-2**)

Expressions generating empty composites

empty-bool |

empty-number |

empty-word |

...

Row expressions

row Identifier val DatExp ee |

expand-row DatExp at Identifier by DatExp ee |

reduce-row DatExp at Identifier ee |

```

row DatExp at Identifier ee |
change-row DatExp at Identifier by DatExp ee |

```

Row table expressions

```

table DatExp at Identifier ee |
add-row DatExp to Identifier ee |
delete-row TraExp from Identifier ee |
remove Identifier from Identifier ee |
clear Identifier with TraExp ee |
intersect Identifier with Identifier ee |
union Identifier with Identifier ee |

```

Column table expressions

```

add-column Identifier with DatExp to Identifier ee |
remove-column Identifier from Identifier ee |
filter-columns AcPaDe from Identifier ee |
remove-column Identifier from Identifier ee |
update-column Identifier in Identifier with TraExp
                                where TraExp ee |

```

Expression creating derivative table

```

table Identifier with Identifier at Identifier
                                where TraExp ee |

```

Transfer expressions

```

wtr : TraExp =
... (here stand all clauses of Lingua-2)
row . Identifier |
unique |
all TraExp ee

```

Type expressions

```

wyt : TypExp =
... (here stand all clauses of Lingua-2)
row-type Identifier as TypExp ee |
expand-row-type TypExp by Identifier as TypExp ee |
table-type DatExp as TraExp ee

```

Type constant declarations

There are no new clauses in this group. Of course, the “former clauses” refer to new type expressions.

Data-variable declarations

```
vde :VarDec =
  ...                               (here stand both clauses of Lingua-2)
  create table Identifier as TypExp ed
```

Transactions

```
trn : Transaction =
  add DatExp to Identifier et      |
  delete DatExp from Identifier et |
  exclude Identifier from Identifier et |
  add column Identifier with DatExp to Identifier et |
  drop column Identifier from Identifier et |
  select columns AcPaDe from Identifier et |
  update Identifier at Identifier with TraExp et |
  savepoint Identifier et         |
  release savepoint Identifier et  |
  rollback Identifier et          |
  rollback Identifier if DatExp et |
  constraints off                 |
  constraints on                  |
  Transaction ; Transaction
```

Instructions

```
ins : Instruction =
  ...                               (here stand all clauses of Lingua-2)
```

Table instructions

```
delete cascade TraExp from Identifier ei |
add row DatExp to Identifier ei         |
union Identifier with Identifier into Identifier ei |
intersect Identifier with Identifier into Identifier ei |
create Identifier from Identifier and Identifier col Identifier
```

```

                                where TraExp ei |
modify column Identifier in Identifier by TraExp
                                where TraExp ei |

```

Database instructions

```

activate Identifier |
archive as Identifier |
set reference of Identifier et Identifier to Identifier ei |
clear reference of Identifier et Identifier to Identifier ei |

```

Queries

Queries are assignments (hence instructions) that assign a created table to the system identifier `monitor` and do not check anything since there is no type assigned to that monitor. Consequently, their denotations are slightly different from corresponding instructions, which means that their syntaxes must differ accordingly. I assume that they are created from corresponding instruction by adding a prefix **show**

```

que : Query =
  show Identifier |
  show union Identifier with Identifier into Identifier ei |
  show intersect Identifier with Identifier into Identifier ei |
  show create Identifier from Identifier and Identifier
                                col Identifier where TraExp ei

```

In the end, one methodological remark. In **Lingua-SQL**, we have all constructors of data expression denotations of **Lingua-2**. In particular, we have all the table expressions. We also have assignments where such expressions may appear. All these tools are rather far from SQL standard and may lead — with complex expressions — to hardly readable programs and difficult to formulate proof rules.

An alternative solution may consist in allowing only **Lingua-2** expressions and row expressions, in disposing of table expressions, and in using table instruction for step-by-step construction of tables. This solution does not mean, however, that at the model level, we cannot introduce constructors of table-expression denotations. However, when designing the syntax, we may take an engineering decision that some of these constructors are not included in the signature of the algebra of denotations. They may be treated as auxiliary functions used only at the level of the model. In such a case, their syntactic counterparts will not appear in syntax.

10.11 Colloquial syntax

The majority of new syntactic constructions of **Lingua-SQL** does not seem to require the introduction of colloquialisms. They may be made more user-friendly at the level of concrete syntax. However, the introduction of colloquialisms may be worthwhile in the case of table-variable declarations to make them closer to a typical SQL-syntax. Let us consider an example of such a declaration written in an SQL style (cf. Sec.9.3):

```

create table Employees with

```

```

Name          Varchar(20)    NOT NULL,
Position      Varchar(9),
Salary        Number(5)      DEFAULT 0,
Bonus         Number(4)      DEFAULT 0,
Department_Id Number(3)      REFERENCES Departments,
CHECK (Bonus < Salary)

```

ed

The restoring transformation would change this declaration into a sequential composition of a table-variable declaration and a database instruction:

```

create table Employees as
  table-type dat_exp with yok_exp ee
ed ;

set reference of Employees et Department_Id to Departments ei

```

where `dat_exp` and `tra_exp` represent a type expression and a yoke expression, respectively.

Restoring the data expression by means of row-creation and row-expansion constructors and the transfer expression with transfer-expression constructors we get the following concrete version of our colloquial declaration:

```

create table Employees as                                the beginning of the declaration
  table-type                                              the beginning of type expression
    expand-row                                           the beginning of data expression
      expand-row
        expand-row
          expand-row
            row Name val empty-word ee
          by Position val empty-word ee
        by Salary val 0 ee
      by Bonus val 0 ee
    by Department_Id by empty-number ee                the end of data expression
  with                                                    the beginning of transfer expression (yoke expression)
    all
      varchar(20) (row.Name)                               and
      not-null(row.Name)                                    and
      varchar(9) (row.Position)                            and
      number(5) (row.Salary)                               and
      number(4) (row.Bonus)                                and
      number(3) (row.Department_Id)                       and
      row.Bonus < row.Salary
    ee                                                    the end of transfer expression (yoke expression)

```

ee*the end of type expression***ed ;***the end of declaration***set reference of** Employees **et** Department_Id **to** Departments **ei**

Of course `varchar(20)`, `varchar(9)`,... are the names of appropriate predicates. Notice that in this example one “syntax unite” from the colloquial lever is transformed into a sequential composition of a declaration and an instruction.

10.12 The rules of correct-program constructions

The enrichment of the former versions of **Lingua** to **Lingua-SQL** consists basically on the extension of data- and type-algebras whereas new instructions are table modifications that on the denotational level refer to the generalised assignment. For the author of validation rules, this means the necessity of defining new conditions and new properties (Sec. 8.2 and Sec. 8.4.1). This should be postponed, however, until some practical version of **Lingua-SQL** is created.

11 LINGUA-OO — OBJECT-ORIENTED PROGRAMMING (WORK IN PROGRESS)

Work on this section is in progress.

12 WHAT REMAINS TO BE DONE

Even though the book is already of a considerable volume, the majority of subjects have only been sketched. What remains to be done is enough for a few more books and also as a research and development area for many researchers and developers. Below I suggest a preliminary list of subjects which is certainly not complete. It considers both research problems as well as programming (implementational) tasks.

12.1 Foundations

12.1.1 The extension of **Lingua** model

All currently described languages from the **Lingua** family — maybe except **Lingua-3** (object programming) — cover mainly traditional programming tools developed in the years 1960-1980. Since they are present today in the majority of programming languages, it was somewhat natural to start with them, which does not mean, however, that the model of **Lingua** should not be developed further. In my opinion, the next step should be the extension of our model by newer mechanisms, e.g., by script languages of HTML type or concurrency based on Mazurkiewicz and/or Petri model.

A few minor research problems have been mentioned in the central part of the book.

12.1.2 The completion of the **Lingua** model

The development of a complete (a practical) model for **Lingua** covering not only denotations, syntax, and semantics but also sound program-construction rules. In the last area, a closer look at assertions (Sec. 8.3) may be worthwhile since, so far, this issue has only been sketched.

12.1.3 The principles of writing user manuals

Denotational models should provide an opportunity for the revision of current practices seen in the manuals of programming languages. New practices should, on the one hand, base on denotational models, but on the other — do not assume that today's readers are experts in this field. A manual should, therefore, provide some basic knowledge and notation needed to understand the definition of a programming language written in a new style. At the same time — I firmly believe in that — it should be written for professional programmers rather than for amateurs. The role of a manual is not to teach the skills of programming. Such textbooks are, of course, necessary, but they should tell the readers what the programming is about rather than the technicalities of a concrete language. Unfortunately, the current practice usually contradicts this principle.

12.2 Implementation

First attempt to build an implementation of **Lingua** has been undertaken by a small group of two teachers (me and Aleksy Schubert) and three of our students of the Department of Mathematics, Informatics, and Mechanics of Warsaw University (see [33]) during the Spring Semester of the year 2020. To tell the truth, my role was limited in this case to checking if the developed implementation was compatible with the model of **Lingua**, as described in this book. The programming language of implementation was OCaml.

12.2.1 Tools for language developers

1. A tool generating abstract-syntax grammar from a signature (a meta-definition) of the algebra of denotations.
2. A tool supporting the development of a concrete-syntax grammar from an abstract-syntax grammar.
3. A tool supporting the generation of a restoring application from colloquial syntax into a concrete syntax.
4. An editor supporting the writing of the definitions of denotation constructors.
5. A generator of semantic clauses from a concrete-syntax grammar and the definitions of denotation constructors.
6. A generator of an interpreter/compiler code from semantic clauses.

12.3 Tools for programmers

An editor supporting program-development using correct-metaprogram development rules must be developed.

12.4 Manuals

To provide a practical value for the methodology which is contained in **Lingua**, there must be user manuals that follow that methodology. And, of course, they have to base on principles mentioned in Sec. 12.1.3. As a matter of fact, both these tasks should be developed in parallel. To describe rules for writing manuals, some experiments in writing manuals should take place, and experimental manuals must follow the developed general rules.

12.5 Programming experiments

For our idea of correct-program development to be noticed by the IT community, some convincing applications must be shown. In my opinion, an adequate field for such applications may be microprograms because:

1. microprograms contain a relatively small number of the lines of code,
2. their correctness is highly critical,
3. highly critical is also the memory- and time-optimisation of such programs.

Each experimental program developed within our framework must be independently tested by usual industrial tests.

12.6 Building a community of Lingua supporters

Our methods of designing programming languages and constructing programs may be assessed positively or negatively, but one seems to be evident — they are indeed quite far from current practices. What the book offers is a far-going change, and such changes always provoke springing up groups of opponents and supporters. The former should be convinced, and the latter must be kept. And of course, one has to start from the first task.

To realize that task one has to give the potential supporters some, may be very simple, still sufficiently practical, version of **Lingua** or — as an alternative — encourage them to build their own version. The first solution seems somewhat unrealistic since it would require finding an investor for a strange and utterly unknown product. The other way that remains means that an experimental **Lingua** is built by volunteers and for volunteers, as in the case of Linux, Joomla! or Drupal. However, such a product, although freely available, should not be an open-source product since this might lead to mathematically incorrect solutions and consequently to unsound program-construction rules.

The community of **Lingua** builders must, therefore, elaborate rules of accepting new members and of giving them rights for joining implementation teams.

13 ANNEXE 1 — GENERALIZED TREES

To be translated from the Polish version of the book.

14 ANNEXE 2 — ABOUT USER MANUALS

To be translated from the Polish version of the book.

15 REFERENCES

- [1] Aalst Wil van der, Hee Kees van, *Workflow management: models, methods, and systems (Cooperative Information Systems)*, MIT Press 2004
- [2] Ahrent Wolfgang, Beckert Bernhard, Bubel Richard, Hähle Reiner; Schmitt Peter H., Ulbrich Matias (Eds.), *Deductive Software Verification — The KeY Book; From Theory to Practice*, Lecture Notes in Computer Science 10001, Springer 2016
- [3] Aho A.V., Ullman J.D., *The Theory of Parsing, Translation, and Compilation, volume 1, Parsing*, Prentice-Hall, Englewood Cliffs, NJ 1972
- [4] Apt K.R., *Ten Years of Hoare's Logic: A Survey - Part 1*, ACM Trans. Program. Lang. Syst. 3(4): 431-483 (1981)
- [5] Apt Krzysztof R., Olderog Ernst-Rüdiger, *Fifty years of Hoare's Logic*, Springer 2020
- [6] Apt Krzysztof R., Boer (de) Frank, S., Olderog Ernst-Rüdiger, *Verification of Sequential and Concurrent Programs*, Third, Extended Edition, Springer 2020
- [7] Backus J.W., Bauer F.L., Green J., Katz C., McCarthy J., Naur P. (Editor), Perlis A.J., Rutishauser H., Samelson K., Vauquois B., Wegstein J.H., Van Wijngaarden A., Woodger M., *Report on the algorithmic language ALGOL 60*, Numerische Mathematik 2, 106--136 (1960)
- [8] Bakker Jaco (de), *Mathematical Theory of Program Correctness*, Prentice/Hall International 1980
- [9] Banachowski Lech, *Bazy danych. Tworzenie aplikacji*, Akademicka Oficyna Wydawnicza PLJ, Warszawa 1998
- [10] Banachowski Lech, Kreczmar Antoni, Mirkowska Grażyna, Rasiowa Helena, Salwicki Andrzej, *An introduction to Algorithmic Logic — Metamathematical Investigations of Theory of Programs*, T. 2: Banach Center Publications. Warszawa PWN, 1977, s. 7-99, series: Banach Center Publications, vol.2
- [11] Barringer H., Cheng J.H., Jones C.B., *A logic covering undefinedness in program proofs*, Acta Informatica 21 (1984), pp. 251-269
- [12] Bekić Hans, *Definable operations in general algebras and the theory of automata and flowcharts* (manuscript), IBM Laboratory, Vienna 1969
- [13] Binsbergen L. Thomas van, Mosses Peter D., Sculthorpe C. Neil, *Executable Component-Based Semantics*, Preprint submitted to JLAMP, accepted 21 December 2018
- [14] Bjørner Dines, Jones B. Cliff, *The Vienna development method: The metalanguage*, Prentice-Hall International 1982
- [15] Bjørner Dines, Oest O.N. (ed.), *Towards a formal description of Ada*, Lecture Notes of Computer Science 98, Springer Verlag 1980
- [16] Blikle Andrzej, *Automaty i gramatyki — wstęp do lingwistyki matematycznej*, (Automata and Grammars — An Introduction to Mathematical Linguistics) PWN 1971
- [17] Blikle Andrzej, *Algorithmically definable functions. A contribution towards the semantics of programming languages*, Dissertationes Mathematicae, LXXXV, PWN, Warszawa 1971
- [18] Blikle Andrzej, *Equational Languages*, Information and Control, vol.21, no 2, 1972
- [19] Blikle Andrzej, *Analysis of programs by algebraic means*, Mathematical Foundations of Computer Science, Banach Center Publications, vol.2, Państwowe Wydawnictwa Naukowe, Warszawa 1977

- [20] Blikle Andrzej, *Toward Mathematical Structured Programming*, Formal Description of Programming Concepts (Proc. IFIP Working Conf. St. Andrews, N.B Canada 1977, E.J Neuhold ed. pp. 183-2012, North Holland, Amsterdam 1978
- [21] Blikle Andrzej, *On Correct Program Development*, Proc. 4th Int. Conf. on Software Engineering, 1979 pp. 164-173
- [22] Blikle Andrzej, *On the Development of Correct Specified Programs*, IEEE Transactions on Software Engineering, SE-7 1981, pp. 519-527
- [23] Blikle Andrzej, *The Clean Termination of Iterative Programs*, Acta Informatica, 16, 1981, pp. 199-217.
- [24] Blikle Andrzej, *MetaSoft Primer — Towards a Metalanguage for Applied Denotational Semantics*, Lecture Notes in Computer Science, Springer Verlag 1987
- [25] Blikle Andrzej, *Denotational Engineering or from Denotations to Syntax*, red. D. Bjørner, C.B. Jones, M. Mac an Airchinnigh, E.J. Neuhold, *VDM: A Formal Method at Work*, Lecture Notes in Computer Science 252, Springer, Berlin 1987
- [26] Blikle Andrzej, *Three-valued Predicates for Software Specification and Validation*, first published in VDM'88, *VDM: The Way Ahead*, Proc. 2nd, VDM-Europe Symposium, Dublin 1988, Lecture Notes of Computer Science, Springer Verlag 1988, pp. 243-266, later republished in *Fundamenta Informaticae*, January 1991
- [27] Blikle Andrzej, *Denotational Engineering*, Science of Computer Programming 12 (1989), North Holland
- [28] Blikle Andrzej, *Why Denotational — Remarks on Applied Denotational Semantics*, *Fundamenta Informaticae* 28, 1996, pp. 55-85
- [29] Blikle Andrzej, *An Experiment with a user manual based on denotational semantics*, preprint 2019, DOI: 10.13140/RG.2.2.23355.67366
- [30] Blikle Andrzej, *An Experiment with denotational semantics*, SN Computer Science, (2020) 1: 15. <https://doi.org/10.1007/s42979-019-0013-0>, Springer
- [31] Blikle Andrzej, Jarosław Deminet, *Komputerowa edycja dokumentów dla średnio zaawansowanych*, (Computer-assisted edition of documents for medium-advanced authors), Helion 2020
- [32] Blikle Andrzej, Mazurkiewicz Antoni, *An algebraic approach to the theory of programs, algorithms, languages and recursiveness*, Proc. International Symposium and Summer School on Mathematical Foundations of Computer Science, Warsaw-Jabłonna, 1972.
- [33] Blikle Andrzej in cooperation with Schubert Aleksander, Dziubiak Marian, Kamas Tomasz, *Lingua-WU Report and a diary of the development of its implementation*, a manuscript in statu nascendi Blikle Andrzej, Tarlecki Andrzej, *Naïve denotational semantics*, Information Processing 83, R.E.A. Mason (ed.), Elsevier Science Publishers B.V. (North-Holland), © IFIP 1983
- [34] Blikle Andrzej, Tarlecki Andrzej, *Naïve denotational semantics*, Information Processing 83, R.E.A. Mason (ed.), Elsevier Science Publishers B.V. (North-Holland), © IFIP 1983
- [35] Blikle Andrzej, Tarlecki Andrzej, Thorup Mikkel, *On conservative extensions of syntax in system development*, Theoretical Computer Science 90 (1991), 209-233
- [36] Branquart Paul, Luis Georges, Wodon Pierre, *An Analytical Description of CHILL, the CCITT High-Level Language*, Lecture Notes in Computer Science vol. 128, Springer-Verlag 1982
- [37] Chailloux Emmanuel, Manoury Pascal, Pagano Bruno, *Developing Applications With Objective Caml*, Editions O'REILLY, <http://www.editions-oreilly.fr>
- [38] Chomsky Noam, *Three models for the description of language*, IRE Transactions of Information Theory, IT2, 1956

- [39] Chomsky Noam, *Syntactic Structures*, Hague 1957
- [40] Chomsky Noam, *On certain formal properties of grammars*, Information and Control, 2, 1959
- [41] Chomsky Noam, *Context-free grammar and pushdown storage*, MIT Research Laboratory Electrical Quarterly Progress Reports 65, 1962
- [42] Cohn P.M., *Universal Algebra*, D. Reidel Publishing Company 1981
- [43] Dijkstra Edsger, W., *goto statements considered harmful*, Communications of ACM, 11, 1968, pp. 147-148
- [44] Dijkstra Edsger, W., *A constructive approach to the problem of program correctness*, BIT 8 (1968)
- [45] Dijkstra Edsger, W., *A Discipline of Programming*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 1976
- [46] DuBois Paul, *MySQL*, Wydanie II rozszerzone, Mikom, Warszawa 2004
- [47] Floyd Richard W., *Assigning meanings to programs*, Appl. Math. Comput. 19, 1967, pp. 19-32
- [48] Forta Ben, *SQL w mgnieniu oka*, Helion 2015
- [49] Ginsburg Seymour, *The mathematical theory of context-free languages*, New York 1966
- [50] Ginsburg Seymour, Rice, H.G., *Two Families of Languages Related to Algol*, Journal of the Association of Computing Machinery, 9 (1962)
- [51] Goguen, J.A., *Abstract errors for abstract data types*, in Formal Descriptions of Programming Concepts (Proc. IFIP Working Conference, 1977, E.Neuhold ed.), North-Holland 1978
- [52] Goguen, J.A., Thatcher J.W., Wagner E.G., Wright J.B., *Initial algebra semantics, and continuous algebras*, Journal of ACM 24 (1977)
- [53] Gordon M.J.C., *The Denotational Description of Programming Languages*, Springer Verlag, Berlin 1979
- [54] Gruber Martin, *SQL*, Helion 1996
- [55] Hoare C.A.R., *An axiomatic basis for computer programming*, Communications of ACM, 12, 1969, pp. 576-583
- [56] Jensen Kathleen, Wirth Niklaus, *Pascal — User Manual and Report*, Springer Verlag 1975
- [57] Kleene Steven Cole, *Introduction to Metamathematics*, North-Holland 1952; later republished in years 1957, 59, 62, 64, 67, 71
- [58] Konikowska Beata, Tarlecki Andrzej, Blikle Andrzej, *A three-valued Logic for Software Specification and Validation*, w tomie VDM'88, VDM: The Way Ahead, Proc. 2nd, VDM-Europe Symposium, Dublin 1988, Lecture Notes of Computer Science, Springer Verlag 1988, pp. 218-242
- [59] Landin, P. *The mechanical evaluation of expressions*, BSC Computer Journal, 6 (1964), 308-320
- [60] Leszczyłowski Jacek, *A theorem of resolving equations in the space of languages*, Bull. Acad. Polonaise de Science, Série de Sci. Math. Astronom. Phys. 19 (1971)
- [61] Leroy Xavier, Doligez Damien, Frisch Alain, Garrigue Jacques, Rémy Didier, Vouillon Jérôme, *The OCaml system release 4.10, Documentation and user's manual*, February 21, 2020, Copyright © 2020 Institut National de Recherche en Informatique et en Automatique
- [62] Madey Jan, *Od wnioskowania gramatycznego do walidacji specyfikacji wymagań*, w tomie „Symulacja w badaniach i rozwoju”, tom 6, Politechnika Białostocka; na Researchgate https://www.researchgate.net/publication/283225534_Od_wnioskowania_gramatycznego_do_walidacji_specyfikacji_wymagan_From_grammatical_inference_to_validation_of_requirements_specification
- [63] Madey J., Matwin S., *Pascal — opis języka*, Sprawozdania IInf UW nr 54 oraz 55, Wydawnictwa Uniwersytetu Warszawskiego, Warszawa 1976

- [64] Mazurkiewicz Antoni, *Proving algorithms by tail functions*, Information and Control, 18, 1971, pp. 220-226
- [65] McCarthy John, *A basis for a mathematical theory of computation*, Western Joint Computer Conference, May 1961 later published in Computer Programming and Formal Systems (P. Brawffort and D. Hirschberg eds), North-Holland 1967
- [66] Microsoft Press (opr. w. polskiej Piotr Stokłosa), *Microsoft Access 2000 — wersja polska*, Wydawnictwo RM, 2000
- [67] Naur Peter (ed.), *Report on the Algorithmic Language ALGOL60*, Communications of the Association for Computing Machinery Vol. 3, No.5, May 1960
- [68] Niemiec Andrzej, *Wielkość współczesnego oprogramowania*, Biuletyn PTI nr 4-5, 2014
- [69] Norton Peter, Samuel Alex, Aitel David, Eriv Foster-Johnson, Richardson Leonard, Diamond Jason, Parker Aleatha, Michael Roberts, *Python od podstaw*, Wydawnictwo Helion 2006
- [70] Parnas D.L., Asmis G.J.K., Madey J., *Assessment of Safety-Critical Software in Nuclear Power Plants*, Nuclear Safety 32, 2, April-June 1991, str. 189-198.
- [71] Paszkowski Stefan, *Język ALGOL 60*, PWN 1965
- [72] Plotkin Gordon D, *An operational semantics for CSP*, in: Formal Description of Programming Concepts II, D. Bjørner, ed., North-Holland, Amsterdam, pp. 199–225.
- [73] Sephens Ryan, Jones D. Arie, Plew Ron, *SQL w 24 godziny*. Helion 2016
- [74] Stoy, J.E., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, MA 1977
- [75] Scott D., Strachey Ch., *Towards a mathematical semantics of computer languages*, Technical Monograph PRG-6, Oxford University 1971.
- [76] Tarski Alfred, *Pojęcie prawdy w językach nauk dedukcyjnych*, Prace Towarzystwa Naukowego Warszawskiego, Nr 34, Wydział III, 1933, str.35
- [77] Tucker J. V., Zucker J. I.. *Program Correctness over Abstract Data Types, with Error-State Semantics*. North-Holland and CWI Monographs, Amsterdam, 1988.
- [78] Turing Alan, *On checking a large routine*, Report of a Conference on High-Speed Calculating Machines, University Mathematical Laboratory, Cambridge 1949, pp. 67-69.
- [79] Vera (del) Pilar Castillo, Curley Martin, Fabry Eva, Gottiz Michael, Hagedorn Peter, Herczog Edit, Higgins John, Joyce Alexa, Korte, Werner, Lanvin Bruno, Parola Andrea, Straub Richard, Tapscott Don, Vassallo John, *Manifest w sprawie e-umiejętności*, European Schoolnet (EUN Partnership AISBL)
- [80] Viescas John, *Podręcznik Microsoft Access 2000*, wydawnictwo RM 2000

Blikle Andrzej in cooperation with Schubert Aleksander, Alenkiewicz Joachim, Dziubiak Marian, Kamas Tomasz, *Lingua-WU Report and a diary of the development of its implementation*, a manuscript in statu nascendi

16 INDICES AND GLOSSARIES

16.1 Index of terms and authors

abstract error.....	46	colloquial syntax.....	66, 73
abstract syntax.....	56	column.....	248
aggregating function.....	233	composite.....	86
aktualne parametry wartosciowe.....	147	compositionality.....	68
algebra of composites.....	86	computable partiality of functions.....	47
algebra of expression denotations.....	102, 294	concatenation of languages.....	38
algebra of types.....	96	concatenation of tuples.....	34
algorithmic condition.....	188	concatenation of words.....	38
ambiguous algebra.....	59	concrete semantics.....	73
ambiguous grammar.....	63	concrete syntax.....	73
applicative layer of a language.....	68	concretization homomorphism.....	73
Apt K.....	164	condition.....	184
arity of a function.....	53	condition (declaration-oriented).....	187
array.....	79, 237	conditions (data-oriented).....	187
Asmis G.J.K.....	165	conservative denotation.....	127
assertion.....	189	constant.....	202
atomic declaration.....	127	constant of an algebra.....	53
atomic instruction.....	130, 190	constructor of an algebra.....	53
Bakker (de) Jaco.....	197	context-free algebra.....	61
binary relation.....	41	context-free grammar.....	38
body.....	82	context-free language.....	38
body of a procedure.....	148	continuation.....	69, 167
body record.....	82	continuous function.....	36
body-creating constructor.....	85	converse relation.....	42
Boolean value.....	97	copy rule.....	69
call of a functional procedure.....	156	correct metaprogram.....	27, 184, 196
call of an imperative procedure.....	151	cursor.....	234
call-time state.....	142	cursor declaration.....	234
carrier of an algebra.....	53	cursor grasp.....	235
Cartesian power.....	30	data.....	79, 237
chain.....	36	data variable.....	128
chain-complete partially ordered set.....	36	database instruction.....	271
child.....	227, 237	database record.....	257
Chomsky's polynomial.....	40	database value.....	257
clan of a body.....	82	declaration of a functional procedure.....	154
clan of a type.....	96	declaration-time state.....	142
clan of yoke.....	90	declared type constant.....	129
clean evaluation.....	196	declared variable.....	128, 129
clean total correctness.....	171	denotation.....	68
CLI.....	222	descriptive layer.....	27, 184
codomain of a relation.....	41	domain.....	44
Collatz hypothesis.....	172	domain of a function.....	31

domain of a relation.....	41
dynamically-compatible parameters.....	145
eager evaluation.....	48
empty data.....	237
empty table.....	237
empty type.....	96
environment.....	101
equational grammar.....	40
equationally definable language.....	40
error-handling mechanism.....	133
error-state transparent denotation.....	127
existential quantifier.....	30
exporting expression.....	154
extension of a signature.....	53
extension of an algebra.....	54
Fermat theorem.....	172
field.....	225
filtering function.....	35
five-step method.....	74
fixed point equation.....	36
flow-diagram.....	166
Floyd Richard.....	164
foreign key.....	227
formal language.....	38
formal reference-parameters.....	140
formal value-parameters.....	140
function.....	42
functional procedure.....	154
general quantifier.....	29
global table-instruction.....	269
Goguen Joe.....	17
goto instruction.....	69
Hoare C.A.R.....	164
homomorphism (many-sorted).....	54
identifier.....	292
identity function.....	32
identity relation.....	41
iff192	
imperative denotation.....	127
imperative layer of a language.....	68
iteration of a function.....	32
iterative program.....	166
joint predicate.....	231
jump instruction.....	166
kernel of a homomorphism.....	55
Kleene's propositional calculus.....	49
lazy evaluation.....	48
least element.....	35
least fixed point of a function.....	36
least upper bound.....	36
left-hand-side linear equation.....	166
limit of a chain.....	36
linking key.....	227
list.....	79, 237
local table instructions.....	270
Madey J.....	165
many-sorted algebra.....	52
many-sorted language.....	39
mapping.....	31
Mazurkiewicz A.....	164
McCarthy's propositional calculus.....	48
metacondition.....	185
metainstruction.....	195
metapredicate.....	192
metaprogram.....	195
metaprograms.....	185
MetaSoft.....	15
monotone function.....	36
multiprocedures.....	153
Olderog H.R.....	164
one-one function.....	42
on-range.....	190
operational semantics.....	68
overwriting of a function.....	33
parent.....	227, 237
parent-child edge.....	228
Parnas D.L.....	165
partial correctness.....	170, 171
partial function.....	30
partial order.....	35
partial precondition.....	171
partially ordered set.....	35
passing actual parametrs.....	146
polynomial.....	40
polynomial equation.....	168
power of a language.....	39
primary constructor.....	78, 82, 110
primary constructors.....	79
primary key.....	227
principle of simplicity.....	72
procedure body.....	142
procedure content.....	142, 148
procedure environment.....	101
procedure name.....	101
programming layer.....	27, 184
property.....	185
proposition.....	184
pseudocomposite.....	101
pseudovalue.....	100
query.....	231, 270
reachable algebra.....	58
reachable subalgebra.....	58
record.....	79, 237
record attribute.....	79
recovery mechanism.....	229
reflexive domain.....	69
reflexivity.....	35
register.....	219

register-expression.....	219
register-identifier.....	219
register-invariant.....	219
relation.....	41
restoring transformation.....	73
restriction of a signature.....	53
roll-back value.....	230
row instruction.....	262
rows.....	225
Scott D.	167
semantics.....	68
semantics of abstract syntax.....	57
sequential composition of relations.....	41
signature of an algebra.....	52
signature of constructor.....	53
<i>silna spełnialność predykatu</i>	193
similar algebras.....	54
similar signatures.....	55
simple recursion.....	169
skeleton function.....	61
skeleton homomorphism.....	63
skeleton of a function.....	61
słaba spełnialność predykatu.....	193
Sokołowski Stefan.....	209
sort of a function.....	53
specified instruction.....	184, 189
specinstruction.....	189
SQL.....	222
state.....	100
statically-compatible parameters.....	144
store.....	101
Strachey Ch.	167
strong invariant.....	193
structural constructor.....	167
structural data.....	79
structural domain.....	79
structured declaration.....	128, 129
structured induction.....	68
structured instruction.....	131
structured programming.....	167
subalgebra.....	54
subordination indicator.....	238
subordination of tables.....	237
subordination relation.....	227
syntactic algebra.....	61
syntax.....	68
table.....	225
table value.....	257
tail function.....	167
total correctness.....	170
total function.....	30
total order.....	35
total postcondition.....	171
total precondition.....	171
transaction.....	229, 265
transfer.....	91
transformational programming.....	184
transitivity.....	35
transparent for errors (constructor).....	97
trivial instruction.....	131
truncation of a function.....	31
trust test.....	81
tuple.....	33
Turing Alan.....	164
type.....	96
type constant.....	129
type environment.....	101
typewise procedure.....	25
typewise procedure call.....	25
typewise procedure declarations.....	25
unambiguous algebra.....	59
unambiguous grammar.....	63
unambiguous key.....	227
update of a function.....	33
upper bound.....	36
valuation.....	101
variable.....	128
view.....	233, 271
view declaration.....	234
violation-control function.....	262
virtual table.....	234
Wagner Eric.....	17
weak antisymmetry.....	35
weak invariant.....	193
weak total correctness.....	171
word.....	38
Wright Jessie.....	17
wrt.....	37
yoke.....	90
yokeless type.....	96
yokeless value.....	97

16.2 Index of notations

ε : empty word \subseteq : a subset of \rightarrow : partial functions \mapsto : total functions \Rightarrow : mappings \bullet : composition of relations \odot : concatenation \exists : there exists \forall : for all	\emptyset : empty set/relation \sqsubseteq : partial order \emptyset : empty element Ω : pseudo data $\{a.i \mid i=1;n\}$: a set $(a.i \mid i=1;n)$: a sequence $[a.i/b.i \mid i=1;n]$: a mapping $\text{Rel.}(A,B)$: set of relations	$[A]$: subset of identity rel. \blacklozenge : overwriting a function $@$: algorithmic formula \blacksquare : end of theorem/proof \Rightarrow : stronger than (Windings 240)
---	--	---

16.3 Glossary of algebras and domains

This glossary serves mainly the authors of the book for keeping consistency in the use of metavariables.

Sec. 2 METASOFT AND ITS MATHEMATICS

Here we only list some special notation that are specific to this book. Subsections, where no new notation has been introduced, have been omitted.

Sec. 2.1 Basic notational conventions of MetaSoft

- $f \blacklozenge g$ — an overwriting of function f by function g
- \mathbb{C} — Cartesian Concatenation of tuples
- CPO — abbr. Chain-complete Partially Ordered Set

Sec. 2.4 A CPO of formal languages

- \odot — concatenation of words and of languages

Sec. 2.6 A CPO of binary relations

- $\text{Rel}(A,B)$ — the set of all binary relations between A and B ,
- $P \bullet R$ — sequential composition of relations,
- R^* — iteration of a relation,

Sec. 3 General remarks about denotational models

No specific notation.

Sec. 4.3.1 Data

- alp : Alphabet
- ide : Identifier
- boo : Boolean
- num : Number
- wor : Word
- lis : List
- arr : Array
- rec : Record
- dat : Data
- dat : SimpleData

Sec. 4.3.2 Bodies

- bod : Body
- bod : BodyE = Body | Error
- CLAN-Bo : BodyE \mapsto Sub.Data
- BOD : Data \rightarrow Body

sort : BodyE \mapsto {'boolean'}, ('number'), ('word'), 'L', 'A', 'R'

Bc : data-algebra-operations \mapsto body-algebra-operations

Sec. 4.3.3 Composites

AlgCom — the algebra of composites

com : Composite

com : BooComposite

com : CompositeE

com : BooCompositeE

oversized : Composite \mapsto Boolean

round : Data \mapsto Data

sort.(dat, bod) = sort.bod

sort.ide = ide

data.(dat, bod) = dat

body.(dat, bod) = bod

data.ide = ide

body.ide = ide

Sec. 4.3.4 Yoke

AlgYok — the algebra of yokes

tra : Transfer

yok : Yoke

Tc[cco] — constructor of transfers where cco is a constructor of composites.

CLAN-Tr : Transfer \mapsto Sub.Composite

TT = Tc[create-bo.tt]

FF = Tc[create-bo.ff]

Sec. 4.3.5 Types

AlgTyp — the algebra of types

typ : Type

typ : TypeE

CLAN-Ty : Type \mapsto Sub.Composite

4.3.6 Values

AlgVal

val : Value, ValueE

Sec. 4.4 Expression denotations

Sec. 4.4.1 Memory states

Ω — a pseudodata
 sta : State
 env : Env
 sto : Store
 vat : Valuation
 tye : TypEnv
 pre : ProEnv
 prc : Procedure

Sec. 4.4.2 The algebra of denotations of Lingua-A

The *algebra of expression denotations* — which we shall denote by AlgExpDen — contains six carriers:

ide	: Identifier	= ...	defined earlier
ded	: DatExpDen	= State \rightarrow ValueE	data-expression denotations
bed	: BodExpDen	= State \mapsto BodyE	type-expression denotations
tra	: TraExpDen	= Transfer	transfer-expression denotations
yok	: YokExpDen	= Yoke	yoke-expression denotations
ted	: TypExpDen	= State \mapsto TypeE	type-expression denotations

The denotations of transfer expressions and yoke expression are not functions on states since we assume that transfers and yokes are not storable. This is, of course, an engineering decision.

Below we define constructors of the denotations of data expressions, body expressions, and type expressions. Constructors of transfers and yokes have been already defined in Sec. 4.3.4.

Denotations of data expression

ded : DatExpDen
 and-ded : DatExpDen \times DatExpDen \mapsto DatExpDen
 Cdd : constructors of values \mapsto constructors of data-expression denotations

Sec. 4.4.4 Denotations of body-, trace-, yoke- and type expression

bed : BodExpDen
 ted : TypExpDen
 Cbd : constructors of bodies \mapsto constructors of body-expression denotations
 Ctd : constructors of types \mapsto constructors of type-expression denotations

Sec. 4.5 Algebras of the syntax of expressions

Sec. 4.5.1 Abstract syntax of Lingua-A

ide : IdentifierA
 dae : DatExpA
 tre : TraExpA
 yoe : YokExpA
 bod : BodExpA
 tex : TypExpA

Sec. 4.5.2 Concrete syntax of Lingua-A

ide : Identifier
 dae : DatExp
 tre : TraExp
 yoe : YokExp
 bod : BodExp
 tex : TypExp

Sec. 4.6 A sketch of the semantics of Lingua-A

Cs : AlgExp \mapsto AlgExpDen

with five components:

Sid : Identifier \mapsto Identifier
 Sde : DatExp \mapsto DatExpDen
 Stre : TraExp \mapsto TraExpDen
 Syoe : YokExp \mapsto YokExpDen
 Sbe : BodExp \mapsto BodExpDen
 Ste : TypExp \mapsto TypExpDen

Sec. 5 LINGUA-1 — AN IMPERATIVE LANGUAGE WITHOUT PROCEDURES

Sec. 5.1 Denotations

ide : Identifier
 ded : DatExpDen
 tra : TraExpDen
 bed : BodExpDen
 yok : YokExpDen
 ted : TypExpDen
 ded : DecDen
 ind : InsDen

prd : ProDen

Sec. 5.2 Syntax

Sec. 5.2.2 Concrete syntax

New domains only.

prg : Program

dec : Declaration

ins : Instruction

Sec. 5.3 Semantics

Sde : Declaration \mapsto DecDen

Sin : Instruction \mapsto InsDen

Spr : Program \mapsto ProDen

Sec. 6 LINGUA-2 — PROCEDURES

Sec. 6.1.3 Imperative procedures in a denotational framework

ipr : ImpPro = AcPaDe x AcPaDe \mapsto Store \rightarrow Store imperative procedures

apd : AcPaDe = Identifier^{c*} list of actual-parameter denotations

fpd : FoPaDe = (Identifier x TypExpDen)^{c*} list of formal-parameter denotations

ipr : ImpPro = AcPaDe x AcPaDe \mapsto Store \rightarrow Store (imperative procedures)

fpr : FunPro = AcPaDe x AcPaDe \mapsto Store \rightarrow CompositeE (functional procedures)

pro : Procedure = ImpPro | FunPro (procedures)

idd : IprDecDen = State \mapsto State (denotations of imp. procedure-declarations)

fdd : FprDecDen = State \mapsto State (denotations of fun. procedure-declarations)

Sec. 6.3.1 Constructor of procedures

ipc : IprConDen = FoPaDe x FoPaDe x ProDen

Sec. 6.4.1 Multiprocedures and their components

mcd : MprComDen = (Identifier x IprConDen)^{c*} multiprocedure-component denotations

Sec. 6.5.3 Constructors of functional-procedure-denotation contents

fcd : FprConDen = FoPaDe x ProDen x DatExpDen x TypExpDen

Sec. 6.8.2 Concrete syntax

New domains only.

```

acp : ActPar = empty-ap | Identifier | ( ActPar , Identifier )
fop : ForPar = empty-fp | Identifier as TypExp sa | ( ForPar , Identifier as TypExp sa )
ico : lprCon = ((va1 ForPar ref ForPar) Program )
mpc : MprCon = (Identifier, lprCon) | (MprCon , MprCon)
fco : FprCon = Identifier (ForPar) Pro return DatExp as TypExp )

dec : Declaration =
  (variable declaration and type declarations as in Lingua-1) |
  proc Identifier lprCon endproc |
  mulproc MprCon endmulproc |
  fun Identifier FprCon endfun

```

Lingua-SQL, Sec. 10 ???

row : Row = Identifier \Rightarrow SimData

tab : Table = Row^{c*}

sbo : SimBody = {'boolean'}, ('number'), ('word'), ('date'), ('time'), (date-time)}

bod : RowBody = {'Rq'} x BodRow

ror : BodRow = Identifier \Rightarrow SimBody

bod : TabBody = {'Tq'} x Row x RowBody

com : SimCom =

{(dat, bod) | (dat, bod) : CompositeE **and** bod : SimBody}

Θ : CLAN-Bo.bod

A sub[ide] B — the subordination of tables

col : ColumnE = SimCom^{c+} | Error

RowVal = {(com, tra) | sort.com = 'Rq' **and** tra.com = (tt, ('boolean'))}

TabVal = {(com, tra) | sort.com = 'Tq' **and** tra.com = (tt, ('boolean'))}

dbr : DatBasRec = Identifier \Rightarrow TabVal

sb-graph — that binds subordination graphs in type environments,

copies — that binds finite sets of tables in valuations,

- monitor — that binds tables in valuations,
- check — that binds words 'yes' and 'no' in valuations.