The Syntax of Lingua-D

(a work in progress)

Andrzej Jacek Blikle

November 15th, 2025

1 Introductory remarks

Lingua-D is a metaprogramming language to be used by programmers developing correct metaprograms in **Lingua-V**. From a logical perspective, **Lingua-D** is a language of a formalized theory of the denotations of **Lingua-V**. Its syntax includes the syntax of **Lingua-V**, "enriched" by metavariables that run over the denotations of **Lingua-V** (first-order variables). Due to algorithmic axioms we do not need second-order variables.

All syntactic elements of **Lingua-V** except metaconditions constitute ground terms in **Lingua-D,** whereas metaconditions constitute ground formulas. Free terms and formulas include metavariables.

Lingua-V is a language of so-called validating programming, and its denotational model includes a model of a programming language **Lingua**. Both are described in [2], but the syntax of **Lingua-V** is only sketched. The denotational model of **Lingua-D** is outlined in [1].

The main task undertaken in the present paper is to give a first draft of a "sufficiently complete" version of **Language-D** to be used in an experimental development of programs written in **Lingua-V.** The process of program development will be supported by an ecosystem for programmers outlined in [1]. One of its main components will be an intelligent text editor with a grammar checker based on **Lingua-D** grammar. Visual Studio Code has been chosen as a platform on which this editor will be implemented.

According to the method of the development of syntaxes of programming languages described in [2], we will develop the syntax of our **Lingua-D** in three steps corresponding to its three versions:

1. abstract syntax,
2. concrete syntax,
3. colloquial syntax.

All three versions of this language will be defined by equational grammars whose theory is outlined in [2]. They correspond closely to the so-called Backhus-Naur forms on one hand and to Chomsky's grammars, on the other.

2 A recollection of building syntaxes in a denotational framework

The ultimate syntax of a programming language with a denotational model, called in [2] *colloquial syntax,* is derived from the signature of the algebra of denotation of this language in three steps:

1. the transformation of the signature into a grammar (algebra) of *abstract syntax*; the elements of this syntax may be regarded as linear representation of parsing trees; A2D is the unique homomorphism of abstract syntax into denotations and is the *semantics of abstract syntax*,
2. a homomorphic transformation of abstract-syntax grammar (algebra) into a *concrete-syntax* grammar (algebra); the corresponding homomorphism A2C is usually not an isomorphism (is glueing), but it must be *adequate* for A2D, i.e., it must not glue more than A2D (see Sec. Sec. 2.16 and 2.17 of [2]),
3. the construction of such a *colloquial-syntax* grammar (algebra) for which there exists a *restoring transformation* of colloquial syntax (back) to a concrete syntax.

This process is visualized in Fig. 2-1.

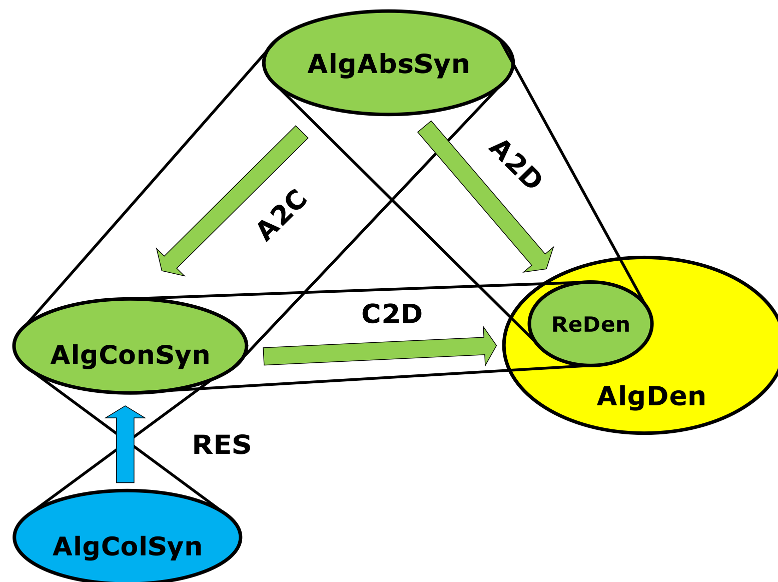

Fig. 2-1 An algebraic model of a language with colloquial syntax

If we denote:

A2D : **AlgAbsSyn** ⟼ **AlgDen** — the homomorphic semantics of abstract syntax,
A2C : **AlgAbsSyn** ⟼ **AlgConSyn** — a homomorphic transformation of abstract to concrete syntax,
C2D : **AlgConSyn** ⟼ **AlgDen** — a homomorphic semantics of concrete syntax,
RES : **AlgColSyn** ⟼ **AlgConSyn** — a non-homomorphic transformation of col. to concrete syntax.

then the semantics SEMC of colloquial syntax may be symbolically defined in the following way:

SEMC = RES ● A2C^{-1} ● A2D.

In this equation A2C^{-1} denotes a (chosen) parsing procedure of concrete syntax, i.e., a concrete-to-abstract transformation. Since A2C is glueing, some words in concrete syntax may have more than one parsing tree. However, due to the adequacy of A2C the choice of any of them is semantically irrelevant.

We recall in this place that although abstract syntax is unambiguously determined by the signature of the algebra of denotations, a corresponding concrete syntax is not. Its choice is dictated by the task of making it possibly user-friendly. This same concerns colloquial syntax.

The described process of syntax derivation applies to programming language such as **Lingua** or **Lingua-V**. In this report, we shall apply it to the latter, but our ultimate goal is to design the syntax of **Lingua-D**, which is derived from **Lingua-V** by adding to it denotational variables. The family of these variables is many-sorted which means that every variable has a sort corresponding to one syntactic category of **Lingua-V**.

Note that denotational variables are “logical” variables in **Lingua-D**, which is a language of a formalized theory of the denotation of **Lingua-V**. They must not be confused with “programming” variable in **Lingua-V**, which are single-identifier value expressions.

In our exercise with the derivation of the syntax of **Lingua-D** we shall add denotational variables at the beginning of syntax derivation, i.e., to the abstract syntax of **Lingua-V**. A technical advantage of this solution is such that in the step from concrete to colloquial syntax we will introduce some auxiliary categories where denotational variables are not needed.

The outlined process of the derivation of a colloquial syntax is, in fact, a process of the derivation of their grammars. In our exercise we shall split this process into four following steps (rather than three):

1. The derivation of an abstract syntax — this step is routine as described in [2], except for adding denotational variables. We only recall that abstract syntax is a prefixed syntax and that all prefixes derived from the constructors of denotations are typeset in green Arial Narrow.
2. An *isomorphic transformation* of abstract syntax to concrete syntax. All modifications applied in this step have isomorphic character, i.e., are unambiguously reversible. For instance, the definitional clause

sid-compose(SpeIns , SpeIns) is transformed into (SpeIns ; SpeIns). In this step we create an *isomorphic concrete grammar*.

3. A non-isomorphic transformation of the isomorphic concrete grammar into a *homomorphic concrete grammar*. All modifications of this step have a non-isomorphic character, but must be adequate. E.g., definitional clause (SpeIns ; SpeIns) is replace by SpeIns ; SpeIns. The omission of parentheses causes non-isomorphicity.
4. The final transformation of the homomorphic concrete grammar into a colloquia grammar supplemented with algorithmic definitions of the corresponding restoring transformation. E.g., we allow for the omission of parentheses in integer-, real- and boolean expressions and assume that the restoring transformation prioritizes multiplication and division over addition and substruction and adds the remaining parentheses from left to right. E.g., an expression a + b + a*b + c will be restored into (a + (b + (a*b)) + c).[1]

The reason of splitting the step from abstract to concrete into two steps (2. and 3.) is to make clearly visible which transformations are not isomorphic. Such transformations should be chosen in a thoughtful way, since they contribute to the complexity of a future parsing process.

In the end let's make it clear that the syntax described in this paper should be regarded as an early proposal of some future final syntax. It will be used in experiments with our language which — we hope — should contribute to building a first practical version of **Lingua-D.**

3 Notational conventions

Notational conventions that we introduce below are thought to make our grammars reader-friendly for programmers. We assume that to make them readable by a parser generator, they will have to be reformulated into a form suitable for the generator to be used.

- Syntactic domains are denoted by strings of letters typeset with Arial and starting with a capital letter, e.g., TexValExp. The names of these domains are referred to as *nonterminals* of our grammars.
- A | B denotes the union of domains A and B.
- A © B, or A B, if it does not lead to confusion, denotes the concatenation of domains A and B.
- A* denotes the domain of finite, possibly empty, concatenations of the elements of A. It is called the *star-iteration* of A.
- A^+ denotes the domain of finite, non-empty concatenations of the elements of A. It is called the *plus-iteration* of A. Consequently $A^* = A^+ | \{()\}$ where () denotes the empty word (an empty tuple of characters).
- We assume that power operators '*' and '+' bind stronger than the union and concatenation and concatenation binds stronger than union, e.g., A | B C* means A | (B © (C)*).
- Strings of characters typeset in Arial Narrow and referred to as *terminals,* denote single-element sets consisting of these string, e.g., and(denotes { and(}. Nonterminals, i.e., the names of syntactic domains are typeset in Arial.
- Scripts combining terminals with nonterminals e.g., and(ValExp , ValExp) should be understood, accordingly to the former rules, as { and(} © ValExp © { , } © ValExp © {) }

The grammar of **Lingua-D** was outlined in [1] as an extension of a grammar of **Lingua** sketched in [2]. Here, we shall use a mixture of notations introduced in both these sources and also allow some exceptions:

- compared to [1], we omit the postfix '-D' in the names of domains, e.g., we write ValExp instead of ValExp-D,
- compared to [2], we omit the prefixed Abs (abstract syntax), Con (concrete syntax) and Col (colloquial syntax) in the names of domains.

---

[1] The omission of parentheses in 2. is adequate, whereas a similar omission in 3. is not, since the composition of functions is associative and in our arithmetics with errors, arithmetic operations are not so. Note that ((a + b) – c) may generate an overflow error, whereas (a + (b – c)) — will not.

We allow the second exception since each of our three syntaxes will be described by an “independent” grammar and, therefore, we may use common nonterminals in these grammars without a risk of confusion.

4 Abstract syntax

4.1 Auxiliary domains

Syntactic domains of our future grammar will be associated with the carriers of the algebra of denotations **AlgDen-V** — one syntactic domain for every such carrier. To define them we will need some auxiliary domains that are not derived from **AlgDen-V** and will not become the carriers of **AlgSyn-D**.

LowLetter = a | b | … | x | y | z
CapLetter = A | B | … | X | Y | Z
Letter = LowLetter | CapLetter

PositiveDig = 1 | 2 | …| 9
Digit = 0 | PositiveDig

VisibleSign = (|) | . | , | ; | : | - | _ | – | ! | ? | @ | # | $ | % | ^ | & | * | + | / | \ | “ | < | > | = | ≠
InvisibleSign = ¶ | ¥ carriage return and white space respectively
SpecialSymbol = ‘ the unique special symbol is an apostrophe
Sign = VisibleSign | InvisibleSign

Label = (Letter | Digit)*
Text = SpecialSymbol (Letter | Digit | Sign)* SpecialSymbol

NonNegInt = 0 | PositiveDig Digit*
NegativeInt = – NonNegInt
IntegerNum = NonNegInt | NegativeInt

NonNegRea = IntegerNum , NonNegInt
NegativeRea = – NonNegRea
RealNum = NonNegRea | NegativeRea

Boolean = true | false

4.2 Variables

4.2.1 Denotational variables

The set of denotational variables is split into categories corresponding to the carriers of **AlgDen-V**. To make the sorts of variables “visible” by the future parser of **Lingua-D**, each of them includes a *sort indicator* followed by a (possibly empty) label. Their domains are written in a colloquial-syntax, since they will be identical for all of our four grammars:

Primitive carrier variables

IdeVar = ide Label identifier variables, i.e. variables running over **Lingua-V** identifiers
PriStaVar = pst Label privacy statuses indicator variables
LisOfIdeVar = loi Label lists of identifier variables
ClaIndVar = cli Label class indicator variables

Applicative carrier variables

ValExpVar = vex Label value-expression variables
TypExpVar = tex Label type-expression variables
RefExpVar = rex Label reference-expression variables
CovExpVar = cex Label covering-relation-expression variables

Imperative carrier variables

InsVar	= ins Label	instruction variables
DecVar	= dec Label	declaration variables
OpeProVar	= pro Label	procedure opening variables
ClaTraVar	= clt Label	class-transformation variables
ProPreVar	= prp Label	program-preamble variables
ProVar	= pro Label	program variables

Declaration-oriented carrier variables

DecSecVar	= dse Label	declaration section variables
ForParVar	= fpa Label	formal-parameter variables
ActParVar	= apa Label	actual-parameter variables

Signature carrier variables

ImpProSigVar	= ips Label	imperative-procedure signature variables
FunProSigVar	= fps Label	functional-procedure signature variables
ObjConSigVar	= ocs Label	object-constructor signature variables

Condition and metacondition variables

ConVar	= con Label	condition variables
MetConVar	= mec Label	metacondition variables

The need of labeling individual variables is a consequence of the fact that in in some terms or formulas of **Lingua-D** we may need to use more than one variable of the same sort, as e.g. in (in concrete syntax):

if vex **then** ins1 **else** ins2 **fi**

4.2.2 Identifiers, class indicators, and privacy statuses

We assume that identifiers in **Lingua-V** are nonempty strings of letters and digits. In **Lingua-D** they play the role of grounded identifiers. In turn, identifiers in **Lingua-D** are either grounded identifiers or identifier variables.

GroIde	= (Letter | Digit | _)$^+$	grounded identifiers
Identifier	= GroIde | IdeVar	identifiers

From grounded identifiers we have to exclude all key-words such as con, if, then, else etc. that appear in colloquial syntax. In short — all green prefixes and infixes.

In a similar style we define the indicators of classes and of privacy statuses:

ClaInd	= empty-class | Identifier | ClaIndVar	class indicators
PriStaInd	= private | public | PriStaVar	privacy-status indicators

And again in all these cases corresponding equations will be included in the future concrete and colloquial grammars.

4.3 Expressions

4.3.1 Type expressions

Below and in all future equations the first grammatical clause consists of a name of a category of corresponding denotational variables. In its place we could have written its definition, e.g., tex Label instead of TypExpVar. However, we have introduced explicit syntactic categories of variables to clearly sort them out in the process of **Lingua-D** derivation.

TypExp =

TypExpVar |

ted-create-bo |
ted-create-in |
ted-create-re |
ted-create-tx |

ted-create-ot(Identifier) | object type is an identifier (the name of a class)

ted-constant(Identifier , Identifier) |

ted-create-li(TypExp) |
ted-create-ar(TypExp , ValExp) |
ted-create-rc(Identifier , TypExp) |
ted-extend-rc(Identifier , TypExp , TypExp)

4.3.2 Value expressions

ValExp =

variables

ValExpVar | free value-expressions in **Lingua-D** consisting of a single-variable
ved-variable(Identifier) | variables in **Lingua**; grounded value-expressions in **Lingua-D**
ved-attribute(ValExp , Identifier) | the value of an object's attribute

integer-value expressions

IntegerNum |
ved-in(IntegerNum) |
ved-plus-in(ValExp , ValExp) |
ved-minus-in(ValExp , ValExp) |
ved-times-in(ValExp , ValExp) |
ved-divide-in(ValExp , ValExp) |

real-value expressions

RealNum |
ved-re(RealNum) |
ved-plus-re(ValExp , ValExp) |
ved-minus-re(ValExp , ValExp) |
ved-times-re(ValExp , ValExp) |
ved-divide-re(ValExp , ValExp) |

boolean-value expressions

integer operators

ved-smaller-in(ValExp , ValExp) |
ved-greater-in(ValExp , ValExp) |
ved-smallerOrEqual-in(ValExp , ValExp) |
ved-greaterOrEqual-in(ValExp , ValExp) |
ved-equal-in(ValExp , ValExp) |
ved-unequal-in(ValExp , ValExp) |

real operators

ved-smaller-re(ValExp , ValExp) |
ved-greater-re(ValExp , ValExp) |
ved-smallerOrEqual-re(ValExp , ValExp) |
ved-greaterOrEqual-re(ValExp , ValExp) |
ved-equal-re(ValExp , ValExp) |
ved-unequal-re(ValExp , ValExp) |

textual operators

ved-equal-tx(ValExp , ValExp) |
ved-unequal-tx(ValExp , ValExp) |

logical operators

ved-bo(Boolean) |
ved-and-m(ValExp , ValExp) | -m stands for McCarthy
ved-or-m(ValExp , ValExp) |
ved-implies-m(ValExp , ValExp) |
ved-not-m(ValExp) |

textual-value expressions

Text |
ved-tx(Text) |
concatenate(ValExp , ValExp) |

At this moment we define a restricted class of textual-value expressions. In the future it may be enriched by other operators.

list-value expressions

ved-create-li(ValExp) |
ved-push-to-li(ValExp , ValExp) |
ved-head-of-li(ValExp) |
ved-tail-of-li(ValExp) |

array-value expressions

ved-create-ar(TypExp , ValExp) | creating an empty array
ved-replace-in-ar(ValExp , ValExp , ValExp) |
ved-get-from-ar(ValExp , ValExp) |

record-value expressions

ved-create-rc(TypExp) | creating an empty record
ved-replace-in-rc(ValExp , Identifier , ValExp) |
ved-get-from-rc(ValExp , Identifier) |

structured value-expressions

ved-conditional(ValExp , ValExp , ValExp) |
ved-call-fun-pro(Identifier , Identifier , ActPar) |

An example of an abstract-syntax expression may be the following:

ved-plus-in(ved-make-var(x), ved-times-in(ved-make-var(y), ved-make-var(z)))

This expression written in concrete and respectively in colloquial style will be the following:

(x + (y * z))
x + y * z

4.3.3 Reference expressions

RefExp =

RefExpVar |
red-variable(Identifier) |
red-attribute(ValExp , Identifier)

4.3.4 Expressions defining covering relations

Covering-relation expressions, called briefly *cov-expressions* (Sec. 9.2.3 of [2]) are necessary to define conditions describing the compatibility of formal parameters with current cov-relations. They do not appear in **Lingua**, but do appear in **Lingua-V**.

CovExp =

CovExpVar |
ced-make(TypExp , TypExp) |
ced-add(TypExp , TypExp , CovExp)

4.4 Instructions

Since in [2] we have not formalized the constructors of the denotations of assertions, specified instructions, specified declarations and specified programs, we ad-hock create such names to use them in our syntax. E.g. asd- stands for "assertion denotation", sid- for "specified instruction denotation", etc.

4.4.1 Assertions

Assertion =

AsrVar) |
asd-make-con(Condition)

4.4.2 Specified instructions

SpeIns =

InsVar |
sid-skip |
sid-asd-to-sin(Assertion) |
sid-on-zone(Condition , SpeIns) |
sid-off-zone(Condition , SpeIns) |
sid-assign(RefExp , ValExp) |
sid-call-imp-pro(Identifier , Identifier , ActPar , ActPar) |
sid-call-obj-con(Identifier , Identifier , ActPar) |
sid-if(ValExp , SpeIns , SpeIns) |
sid-if-error(ValExp , SpeIns) |
sid-while(ValExp , SpeIns) |
sid-compose(SpeIns , SpeIns)

4.5 Specified declarations

SpeDec =

DecVar |
sdd-skip |
sdd-asd-to-sde(Assertion) |
sdd-variable(ListOfIde , TypExp) |
sdd-constant(ListOfIde , TypExp) |
sdd-enrich-cov-rel(TypExp , TypExp) |
sdd-class(Identifier , ClaInd , ClaTra) |
sdd-compose(SpeDec , SpeDec)

4.5.1 Openings of procedures

OpePro =

OpeProVar |
opd-create

4.5.2 Specified class transformers

SpeClaTra =

ClaTraVar |

ctd-skip

ctd-add-abs-att(Identifier , TypExp, PriSta) |
ctd-con-abs-att(Identifier , ValExp) |
ctd-add-con-att(Identifier , ValExp , TypExp, PriSta) |

ctd-add-abs-typ(Identifier) |
ctd-concretize-typ(Identifier , TypExp) |
ctd-add-con-typ(Identifier , TypExp) |

ctd-add-abs-imp-met(Identifier , ImpProSig) |
ctd-concretize-imp-met(Identifier , ImpProSig , SpePro) |
ctd-add-con-imp-met(Identifier , ImpProSig , SpePro) |

ctd-add-abs-fun-met(Identifier , FunProSig) |
ctd-concretize-fun-met(Identifier , FunProSig , SpePro , ValExp) |
ctd-add-con-fun-met(Identifier , FunProSig , SpePro , ValExp) |

ctd-add-abs-obj-met(Identifier , ObjConSig) |
ctd-concretize-obj-met(Identifier , ObjConSig , SpePro) |
ctd-add-con-obj-met(Identifier , ObjConSig , SpePro) |

ctd-compose(SpeClaTra, SpeClaTra)

4.6 Specified preambles of programs

SpeProPre =

ProPreVar |
ppd-dec-to-ppr(SpeDec) |
ppd-ins-to-ppr(SpeIns) |
ppd-compose(SpeProPre , SpeProPre)

4.7 Specified programs

SpePro =

ProVar |
spd-build(ProPre , OpePro , SpeIns)

4.8 Declaration-oriented categories

GroLisOfIde = grounded list of identifiers

gli-build(GroIde) |
gli-loi-add(GroIde , GroLisOfIde)

LisOfIde =

LisOfIdeVar |
loi-var-to-loi(GroLisOfIde)

DecSec=

DecSecVar |
dsd-build(LisOfIde , TypExp)

ForPar =

ForParVar |
DecSec |
fpa-add(DecSec , ForPar)

ActPar =

ActParVar |
apa-build(LisOfIde)

4.9 Signatures

ImpProSig =

ImpProSigVar |
ips-build(ForPar , ForPar)

FunProSig =

FunProSigVar |
fps-build(ForPar , TypExp)

ObjConSig =

ObjConSigVar |
ocs-build(ForPar , Identifier)

4.10 Conditions

The domain of conditions has been only sketched in [2] but in any "practical" **Lingua-D** it may constitute quite a large group. To systematize its description we split it into three categories:

- *common atomic conditions* — boolean value-expressions from **Lingua** without logical operators,
- *special atomic conditions* — **Lingua-V**-specific conditions (Sec. 0),
- *compound conditions* — common and special conditions combined by Kleenee's operators.

Note that in the category of special conditions we introduce a polymorphic relation of mathematical equality cod-equal-math applicable to any categories of values.

Condition =

ConVar |

common atomic conditions

integer operators

cod-smaller-in(ValExp , ValExp) |
cod-greater-in(ValExp , ValExp) |
cod-smallerOrEqual-in(ValExp , ValExp) |
cod-greaterOrEqual-in(ValExp , ValExp) |
cod-equal-in(ValExp , ValExp) |

cod-unequal-in(ValExp , ValExp) |

real operators

cod-smaller-re(ValExp , ValExp) |
cod-greater-re(ValExp , ValExp) |
cod-smallerOrEqual-re(ValExp, ValExp) |
cod-greaterOrEqual-re(ValExp , ValExp) |
cod-equal-re(ValExp , ValExp) |
cod-unequal-re(ValExp , ValExp) |

textual operators

cod-equal-tex(ValExp , ValExp) |
cod-unequal-tex(ValExp , ValExp) |

special atomic conditions

value-, type-, and reference oriented conditions

cod-equal-math(ValExp , ValExp) |

cod-increasingly-ordered-re(Identifier) |
cod-increasingly-ordered-in(Identifier) |
cod-increasingly-ordered-tex(Identifier) |

cod-current(CovExp) |
cod-well-valued(ForPar , CovExp) |
cod-consistent(TypExp , TypExp) |
cod-att-declared(Identifier , TypExp , Identifier , PriStaInd) |
cod-is-type-in(Identifier , Identifier) |
cod-is-var-type(Identifier , TypExp) |
cod-is-value(ValExp) |
cod-is-type(TypExp) |
cod-is-class(Identifier) |
cod-is-child-of(Identifier , ClaInd) |
cod-free(Identifier) |

procedure oriented conditions

cod-is-imperative-pro-dec(Identifier , ForPar , ForPar , Program , Identifier) |
cod-is-functional-pro-dec(Identifier , ForPar , TypExp , Program , ValExp , Identifier) |
cod-is-objectional-dec(Identifier , ForPar , Identifier , Program , Identifier) |
cod-is-opened(Identifier , Identifier) |
cod-pass-parameters(ActPar , ActPar , ForPar , ForPar , Identifier , Condition) |

algorithmic conditions

left-algorithmic-con(SpePro , Condition) |
right-algorithmic-con(Condition , SpePro) |

compound conditions

cod-create-true |
cod-create-false |
cod-and-k(Condition , Condition) -k stands for "Kleene" |
cod-or-k(Condition , Condition) |
cod-implies-k(Condition , Condition) |
cod-not-k(Condition)

4.11 Metaconditions

Similarly as in the case of conditions, also metaconditions may be split into two categories:

- *atomic metaconditions*
- *compound metaconditions*

Their grammar is the following:

```
MetCon =
    MetConVar                                                              |

    mcd-stronger( Condition , Condition )                                  |
    mcd-weak-equivalent( Condition , Condition )                           |
    mcd-less-defined( Condition , Condition )                              |
    mcd-strong-equivalent( Condition , Condition )                         |

    mcd-stronger-whenever( Condition , Condition , Condition )             |
    mcd-weak-equivalent-whenever( Condition , Condition )                  |
    mcd-strong-equivalent-whenever( Condition , Condition , Condition )    |

    mcd-metaprogram( Condition , SpePro , Condition )                      |
behavioral metaconditions
    mcd-insures-LR-of( Condition , SpeIns )                                |
    mcd-resilient-to( Condition , SpePro )                                 |
    mcd-nourishing( Condition , SpePro )                                   |
    mcd-catalyzing-for( Condition , SpePro )                               |
    mcd-essential-for( Condition , SpePro )                                |
    mcd-irrelevant-for( Condition , Condition , SpePro , Condition )       |
temporal metaconditions
    mcd-primary-in( Condition , Condition , SpePro , Condition )           |
    mcd-induced-in( Condition , Condition , SpePro , Condition )           |
    mcd-hereditary-in( Condition , Condition , SpePro , Condition )        |
    mcd-co-hereditary-in( Condition , Condition , SpePro , Condition )     |
    mcd-perpetual-in( Condition , Condition , SpePro , Condition )         |
language-related metaconditions
    mcd-immunizing( Condition )                                            |
    mcd-immanent( Condition )                                              |
    mcd-error-transparent( Condition )                                     |
    mcd-underivable( Condition )                                           |
compound metaconditions
    mcd-and( MetCon , MetCon )                                             |
    mcd-or( MetCon , MetCon )                                              |
    mcd-implies( MetCon , MetCon )                                         |
    mcd-not( MetCon )
```

The logical operators are not postfixed (neither by -m, not by -k) since this time they are classical.

5 Concrete syntax

5.1 Introductory remarks

5.1.1 A special treatment of spaces

In the context of concrete syntax we have to revise our treatment of spaces. In Sec. 3 we have assumed that a metaspace between two names of syntactic domains is regarded as an invisible metasymbol of the concatenation of formal languages. E.g.,

cod-and(Condition, Condition)

stands for

{cod-and(}©Condition©{,}©Condition©{)}

That assumption was possible because the only separators in abstract syntax were parentheses and commas and consequently we did not need spaces in that syntax.

The situation in concrete syntax is different. Now, some separators are strings of letters, such as if or then in which case we have to separate them from neighboring strings of letters, representing e.g., expressions or identifiers. We will separate them by *language-level spaces* which is a usual practice in programming languages. To distinguish these spaces from metaspaces at the level of grammars we shall use symbol ¥ to denote the former spaces. E.g., we shall assume that

while ¥ ValExp ¥ **do** ¥ SpeIns ¥ **od**

will stand for

{**while**}©{¥}©ValExp©{¥}©{**do**}©{¥}©SpeIns©{¥}©{**od**}

An example of a concrete-syntax **while**-instruction may be the following:

while x > 0 **do** x := x - 1 **od**

In this instruction spaces between

while and x,
0 and **do**,
do and x,
1 and **od**,

are *obligatory spaces*. They are separators necessary for a parser to recognize the structure of an instruction. In colloquial syntax we will additionally assume that each such space may be replaced by a carriage return, which means that our instruction may appear on the monitor of a programmer as:

```
while x > 0
   do
      x := x - 1
   od
```

Note that in this example we have not only replaced obligatory spaces by carriage returns, but we have also added some new *tabulating spaces*. These spaces will be regarded as optional and will be neglected by parsers. Technically, a parser will remove all of them before proceeding to tokenization. The same concerns spaces between x and >, between > and 0, etc. We shall not mention this issue anymore assuming that the designer of an editor and a parser will handle the corresponding rules.

5.1.2 The names of domains

In Sec. 5 we define our concrete syntax in such a way that each line in a domain equation of concrete syntax corresponds to one abstract-syntax line in Sec. 3. We shall use the same names of domains as in Sec. 3 and

we assume that the definitions of auxiliary domains of variables (Sec. 4.2), and of indicators (Sec. **Błąd! Nie można odnaleźć źródła odwołania.**), remain unchanged.

5.1.3 On a way from abstract to concrete syntax

In [2] the transformation from abstract to concrete syntax, abbr. A2C, is regarded as a single step. However, a deeper insight into it reveals several stages which may be worth considering individually, since each offers different mathematical challenges. Also from a practical perspective splitting A2C into a sequence of transformations performed one after another, may contribute to a better understanding of the nature of each of them. For the purpose of this paper we shall split A2C into three steps:

1. first *isomorphic transformation* — more user friendly with full parenthesizing:
   a. some prefixes shortened,
   b. some prefixes omitted,
   c. some prefixes replaced by infixes,
2. second isomorphic transformation — the omission of *redundant parentheses*,
3. a homomorphic transformation — the omission of *default parentheses*

Below we show a few typical examples of each of the three categories of transformations.

abstract syntax	first isomorphic concrete syntax
ted-create-bo - boolean type	boolean
ted-create-ot(Identifier) - object type	object(Identifier) object(EmployerClass)
ted-extend-rc(Identifier , TypExp , TypExp)	**extend-rc** TypExp **at** Identifier **with** TypExp **txe** **extend-rc** employee **at** salary **with** real **txe**
ved-create-ar(TypExp , ValExp)	array(TypExp , ValExp) array(integer, 3) array(array(integer, 3), ((2*x)+y)) array(myArrayType, ((2*x)+y))
ved-attribute(ValExp , Identifier) - val. of obj. att.	(ValExp **.** Identifier) (Object1.name1) ((Object1.name1).name2) (((x+y).name1).name2) — syntactically acceptable, but semantically incorrect
ved-get-from-ar(ValExp , ValExp)	(ValExp [ValExp])
ved-get-from-rc(ValExp **,** Identifier)	(ValExp (Identifier))
ved-plus-in(ValExp , ValExp)	(ValExp + ValExp) (x + y) ((x+y)+(2*z))
red-attribute(ValExp , Identifier) - ref. of obj. att.	(ValExp **.** Identifier)
sid-assign(RefExp , ValExp)	(RefExp **:=** ValExp) (x := (x+1))
sid-if(ValExp , SpeIns , SpeIns)	**if** ValExp **then** SpeIns **else** SpeIns **fi** **if** ((x+1)>0) **then** (x := (x+2)) **else** (x := (x – 2)) **fi**
sid-compose(SpeIns , SpeIns)	(SpeIns ; SpeIns) ((x := (x+1)) ; (y := (x+2)))

Tab. 5.1.3-1 First isomorphic transformation

Note that in the concrete-syntax column, the phrase (ValExp **.** Identifier) is a common element of two categories:

- value expressions that return values of objects' attributes,
- reference expressions that return references of objects' attributes.

Still, this fact does not mean that our transformation is gluing (corrupting isomorphism) since the "ambiguous" phrase means two things in two different categories. A parser will unambiguously recognize these two meanings since reference expressions may appear exclusively on the left-hand-side of an assignment operator **:=**.

first isomorphic concrete syntax	second isomorphic concrete syntax
(ValExp . Identifier) - get object att. value	ValExp . Identifier
(ValExp [ValExp]) - get array's element	ValExp [ValExp]
(ValExp (Identifier)) - get record att. value	ValExp (Identifier)
(ValExp . Identifier) - reference expression	ValExp . Identifier
(LisOfIde ¥ as ¥ TypExp) - declaration section	LisOfIde ¥ as ¥ TypExp
(ProPre ; OpePro ; SpeIns)	ProPre ; OpePro ; SpeIns
(pre ¥ Condition : SpePro ¥ post ¥ Condition)	pre ¥ Condition : SpePro ¥ post ¥ Condition
(Identifier . Identifier (ActPar))	Identifier . Identifier (ActPar)
(Identifier . Identifier)	Identifier . Identifier

Tab. 5.1.3-2 Second isomorphic transformation

second isomorphic concrete syntax	homomorphic concrete syntax
(SpeIns ; SpeIns)	SpeIns ; SpeIns
(SpeDec ; SpeDec)	SpeDec ; SpeDec
(SpeClaTra ; SpeClaTra)	SpeClaTra ; SpeClaTra
(ValExp ¥ concatenate ¥ ValExp)	ValExp ¥ concatenate ¥ ValExp

Tab. 5.1.3-3 Homomorphic transformation

5.2 First isomorphic concrete syntax

5.2.1 Type expressions

```
TypExp =
    TypExpVar                                   |
    bool                                        |
    integer                                     |
    real                                        |
    text                                        |
    object( Identifier )                        |   an object type is an identifier (the name of a class)
    ( Identifier . Identifier )                 |   a type assigned to an identifier in a class
    list( TypExp )                              |
    array( TypExp , ValExp )                    |   e.g. in type decl. set MyArray = array(integer, 3) tes
    record( Identifier , TypExp )               |
    extend-rc TypExp at Identifier with TypExp tex
```

5.2.2 Value expressions

ValExp =

ValExpVar	|	free value-expressions in **Lingua-D** consisting of a single-variable
(Identifier)	|	variables in **Lingua**; grounded value-expressions in **Lingua-D**
(ValExp **.** Identifier **)**	|	the value assigned to an attribute of an object

integer-value expressions

IntegerNum	|	
(ValExp **+** ValExp)	|	
(ValExp - ValExp)	|	
(ValExp * ValExp)	|	
(ValExp / ValExp)	|	integer division returns the integer part of division

real-value expressions

RealNum	|	
(ValExp **+.** ValExp)	|	real-value operations are marked with dot "**.**"
(ValExp -. ValExp)	|	
(ValExp *. ValExp)	|	
(ValExp /. ValExp)	|	

boolean-value expressions

integer operators

(ValExp **<** ValExp)	|
(ValExp **>** ValExp)	|
(ValExp =< ValExp)	|
(ValExp >= ValExp)	|
(ValExp **=** ValExp)	|
(ValExp **≠** ValExp)	|

real operators

(ValExp **<.** ValExp)	|
(ValExp **>.** ValExp)	|
(ValExp =<. ValExp)	|
(ValExp >=. ValExp)	|
(ValExp **=.** ValExp)	|
(ValExp **≠.** ValExp)	|

textual operators

(ValExp **=t** ValExp)	|
(ValExp **≠t** ValExp)	|

logical operators

true	|	
false	|	
(ValExp ¥ and-m ¥ ValExp)	|	-m stands for "McCarthy"
(ValExp ¥ or-m ¥ ValExp)	|	
(ValExp ¥ implies-m ¥ ValExp)	|	
not-m(ValExp)		

textual-value expressions

Text	|
(ValExp ¥ concatenate ¥ ValExp)	|

At this moment we define a restricted class of textual-value expressions. In the future it may be enriched by other operators.

list-value expressions

list(ValExp) |
push ¥ ValExp ¥ to ¥ ValExp ¥ sup |
head(ValExp) |
tail(ValExp) |

array-value expressions

array(TypExp , ValExp) |
replace-in-ar ValExp at ValExp with ValExp per |
(ValExp [ValExp]) | a value assigned to an index of an array

record-value expressions

record(TypExp) |
replace-in-rc ValExp at Identifier with ValExp per |
(ValExp (Identifier)) | a value assigned to an attribute of a record

structured value-expressions

if ValExp then ValExp else ValExp fi |
(Identifier . Identifier (ActPar)) a call of a functional procedure

Note that the only difference between the selection expressions for arrays and records in in their parentheses.

5.2.3 Reference expressions

RefExp =

RefExpVar |
(Identifier) |
(ValExp . Identifier)

It is worth noticing in this place that in our isomorphic concrete syntax some expressions may have two or even three different meanings:

(Identifier . Identifier) a type assigned to an identifier in a class (Sec. 5.2.1)
(ValExp . Identifier) the value assigned to an attribute of an object (Sec. 5.2.2)
(ValExp . Identifier) a reference assigned to an attribute of an object (this section)

Note that if in the second and the third case the value expressions are identifiers, then their syntaxes are of the same patterns as in case one. These observations may lead to a conclusion that our isomorphism is glueing, hence, can't be an isomorphism. However, there is no glueing effect here, since the three expressions belong to three different syntactic categories.

5.2.4 Expressions defining covering relations

Covering-relation expressions, called briefly *cov-expressions* (Sec. 9.2.3 of [2]) are necessary to define conditions describing the compatibility of formal parameters with current cov-relations. They do not appear in **Lingua**, but do appear in **Lingua-V**.

CovExp =

CovExpVar |
make-cov(TypExp , TypExp) |
add-to-cov(TypExp , TypExp , CovExp)

5.2.5 Assertions

Assertion =

AsrVar |
asr Condition rsa

5.2.6 Specified instructions

SpeIns =

InsVar |
skip |
(Assertion) |
asd-on ¥ Condition ¥ in ¥ SpeIns ¥ asd-off |
asd-off ¥ Condition ¥ in ¥ SpeIns ¥ asd-on |
(RefExp **:=** ValExp **)** |
call-pro ¥ Identifier **.** Identifier ¥ (val ¥ ActPar ¥ ref ¥ ActPar) |
call-obj ¥ Identifier **.** Identifier (ActPar) |
if ¥ ValExp ¥ then ¥ SpeIns ¥ else ¥ SpeIns fi |
if-error ¥ ValExp ¥ then ¥ SpeIns ¥ fi |
while ¥ ValExp ¥ do ¥ SpeIns ¥ od |
(SpeIns ; SpeIns)

5.2.7 Specified declarations

SpeDec =

DecVar |
skip |
(Assertion) |
variable ¥ ListOfIde ¥ as ¥ TypExp ¥ sa |
constant ¥ Identifier ¥ as ¥ TypExp ¥ val ¥ ValExp ¥ sa |
set ¥ Identifier **=** TypExp ¥ tes |
enrich-cov(TypExp , TypExp) |
class ¥ Identifier ¥ parent ¥ ClaInd ¥ with ¥ ClaTra ¥ ssalc |
(SpeDec ; SpeDec)

5.2.8 Openings of procedures

OpePro =

OpeProVar |
open-procedures

5.2.9 Specified class transformers

SpeClaTra =

ClaTraVar |

skip

abs-att ¥ Identifier ¥ as ¥ TypExp ¥ **:** ¥ PriSta ¥ tta | declare abstract attribute
con-att ¥ Identifier ¥ **=** ¥ ValExp ¥ tta | concretize abstract attribute
att ¥ Identifier ¥ **=** ¥ ValExp ¥ as ¥ TypExp ¥ **:** PriSta tta | declare concrete attribute

abs-typ ¥ Identifier ¥ pyt | declare abstract type
con-typ ¥ Identifier ¥ **=** ¥ TypExp ¥ pyt | concretize abstract type
typ ¥ Identifier ¥ **=** ¥ TypExp ¥ pyt | declare concrete typ

abs-imp ¥ Identifier ¥ (ImpProSig) | declare abstract procedure
con-imp ¥ Identifier (ImpProSig) ¥ SpePro ¥ pmi | concretize abstract procedure
imp ¥ Identifier (ImpProSig) ¥ SpePro ¥ pmi | declare concrete procedure

abs-fun ¥ Identifier (FunProSig) | declare abstract function
con-fun ¥ Identifier (FunProSig) ¥ SpePro ¥ return ¥ ValExp ¥ nuf | concretize abstract function

fun ¥ Identifier (FunProSig) ¥ SpePro ¥ return ¥ ValExp ¥ nuf | declare concrete function

(abs-cns ¥ Identifier (ObjConSig)) | declare abstract obj. const.
con-cns ¥ Identifier (ObjConSig) ¥ SpePro snc | concrete abstract obj. const.
cns ¥ Identifier (ObjConSig) ¥ SpePro snc | declare concrete obj. const.

(SpeClaTra ; SpeClaTra) compose sequentially

5.2.10 Specified preambles of programs

SpeProPre =

ProPreVar |
(SpeDec) |
(SpeIns) |
(SpeProPre ; SpeProPre)

5.2.11 Specified programs

SpePro =

ProVar |
(ProPre ; OpePro ; SpeIns **)**

5.2.12 Declaration-oriented categories

GroLisOfIde = grounded list of identifiers

GroIde |
(GroIde , GroLisOfIde **)**

LisOfIde =

LisOfIdeVar |
(GroLisOfIde)

DecSec=

DecSecVar |
(LisOfIde **as** ¥ TypExp)

ForPar =

ForParVar |
(DecSec) |
(DecSec , ForPar)

ActPar =

ActParVar |
(LisOfIde)

5.2.13 Signatures

ImpProSig =

ImpProSigVar |
(val ¥ ForPar ¥ ref ¥ ForPar)

FunProSig =

FunProSigVar |
(ForPar ¥ return ¥ TypExp)

ObjConSig =

ObjConSigVar |
(ForPar ¥ class ¥ Identifier)

5.2.14 Conditions

Equality and inequality for texts are postfixed with **t**. By == we denote the polymorphic mathematical equality.

Condition =

ConVar |

common atomic conditions

integer operators

(ValExp < ValExp) |
(ValExp > ValExp) |
(ValExp =< ValExp) |
(ValExp >= ValExp) |
(ValExp = ValExp) |
(ValExp ≠ ValExp) |

real operators

(ValExp <. ValExp) |
(ValExp >. ValExp) |
(ValExp =<. ValExp) |
(ValExp >=. ValExp) |
(ValExp =. ValExp) |
(ValExp ≠. ValExp) |

textual operators

(ValExp =t ValExp) |
(ValExp ≠t ValExp) |

special atomic conditions

value-, type-, and reference oriented conditions

(ValExp == ValExp) |

ord-re(Identifier) |
ord-in(Identifier) |
ord-tx(Identifier) |

(CovExp ¥ is-current) |
(ForPar ¥ well-valued-in ¥ CovExp) |
consistent(TypExp , TypExp) |
att ¥ Identifier ¥ is ¥ TypExp ¥ in ¥ Identifier ¥ as ¥ PriStaInd ¥ tta |
(Identifier ¥ is-type-in ¥ Identifier) |
(Identifier ¥ is-var-type ¥ TypExp) |
(ValExp ¥ is-value) |
(TypExp ¥ is-type) |
(Identifier ¥ is-class) |
(Identifier ¥ is-child-of ¥ ClaInd) |
(TypExp ¥ consistent-with ¥ TypExp) |
(Identifier ¥ is-free) |

procedure oriented conditions

```
is-pro ¥ Identifier ( val ¥ ForPar ¥  ref ¥ ForPar )
                begin ¥ Program ¥ end ¥ imperative-in ¥ Identifier ¥ orp |
is-fun ¥ Identifier ( ForPar ¥ return ¥ TypExp )
                begin  ¥ Program ¥ return ¥ ValExp ¥ end ¥
                functional-in Identifier)   nuf                          |
is-obj ¥ Identifier ( ForPar ¥ return ¥ Identifier )
                begin ¥ Program ¥ end ¥ in Identifier ) jbo              |
is-opened( Identifier . Identifier )                                     |
(pass-actual val ¥ ActPar ¥ ref ¥ ActPar ¥
                to-formal ¥ val ForPar ¥ ref ¥ ForPar ¥
                with Identifier ) @ Condition                            |
```

algorithmic conditions

```
( SpePro @ Condition )                                                   |
( Condition @ SpePro )                                                   |
```

compound conditions

```
true                                                                     |
false                                                                    |
( Condition ¥ and-k ¥ Condition )                                        |
( Condition ¥ or-k ¥ Condition )                                         |
( Condition ¥ implies-k ¥ Condition )                                    |
not-k( Condition )
```

5.2.15 Metaconditions

Similarly as in the case of conditions, also metaconditions may be split into two categories:

- *atomic metaconditions*
- *compound metaconditions*

Their grammar is the following:

```
MetCon =
    MetConVar                                                            |
```

relational metaconditions

```
( Condition => Condition )                                 |  stronger
( Condition <=> Condition )                                |  weakly equivalent
( Condition ¥ LD ¥ Condition )                             |  less defined
( Condition ¥ SE ¥ Condition )                             |  strongly equivalent

( Condition => Condition ¥ WNV ¥ Condition )               |  WNV – whenever
( Condition <=> Condition ¥ WNV)                           |
( Condition ¥ SE ¥ Condition ¥ WNV ¥ Condition )           |

( pre ¥ Condition : SpePro ¥ post ¥ Condition )            |
```

behavioral metaconditions

```
( Condition ¥ insures-LR-of ¥ SpeIns )                                   |
( Condition ¥ resilient-to ¥ SpePro )                                    |
( Condition ¥ nourishing ¥ SpePro )                                      |
( Condition ¥ catalyzing-for ¥ SpePro )                                  |
( Condition ¥ essential-for ¥ SpePro )                                   |
( Condition ¥ irrelevant-for ¥ pre ¥ Condition : SpePro ¥ post ¥ Condition )  |
```

temporal metaconditions

(Condition ¥ primary-in ¥ pre ¥ Condition : SpePro ¥ post ¥ Condition) |
(Condition ¥ induced-in ¥ pre ¥ Condition : SpePro ¥ post ¥ Condition) |
(Condition ¥ hereditary-in ¥ pre ¥ Condition : SpePro ¥ post ¥ Condition) |
(Condition ¥ co-hereditary-in ¥ pre ¥ Condition : SpePro ¥ post ¥ Condition) |
(Condition ¥ perpetual-in ¥ pre ¥ Condition : SpePro ¥ post ¥ Condition) |

language-related metaconditions

immunizing(Condition) |
immanent(Condition) |
error-transparent(Condition) |
underivable(Condition) |

compound metaconditions

(MetCon ¥ and ¥ MetCon) |
(MetCon ¥ or ¥ MetCon) |
(MetCon ¥ implies ¥ MetCon) |
not(MetCon)

5.3 Second isomorphic concrete syntax

In this section we show only the changes introduced in the first isomorphic concrete syntax, to make it the second one.

First isomorphic concrete syntax	Second isomorphic concrete syntax
Type expressions	
(Identifier . Identifier)	Identifier . Identifier
Value expressions	
(Identifier)	Identifier
(ValExp . Identifier)	ValExp . Identifier
(ValExp [ValExp])	ValExp [ValExp]
(ValExp (Identifier))	ValExp (Identifier)
(Identifier . Identifier (ActPar))	Identifier . Identifier (ActPar)
Reference expressions	
(Identifier)	Identifier
(ValExp . Identifier)	ValExp . Identifier
Specified instructions	
(Assertion)	Assertion
Specified declarations	
(Assertion)	Assertion
Specified preambles of programs	
(SpeDec)	SpeDec
(SpeIns)	SpeIns
Specified programs	
(ProPre ; OpePro ; SpeIns)	ProPre ; OpePro ; SpeIns
Declaration-oriented categories	
(GroIde , GroLisOfIde)	GroIde , GroLisOfIde
(GroLisOfIde)	GroLisOfIde
(LisOfIde **as** ¥ TypExp)	LisOfIde **as** ¥ TypExp

(DecSec)	DecSec
(DecSec , ForPar)	DecSec , ForPar
(LisOfIde)	LisOfIde
Signatures	
(val ¥ ForPar ¥ ref ¥ ForPar)	val ¥ ForPar ¥ ref ¥ ForPar
(ForPar ¥ return ¥ TypExp)	ForPar ¥ return ¥ TypExp
(ForPar ¥ class ¥ Identifier)	ForPar ¥ class ¥ Identifier

5.4 Homomorphic concrete syntax

Similarly as in Sec. 5.3 we show only modifications introduced into the first isomorphic concrete syntax.

First isomorphic concrete syntax	Homomorphic concrete syntax
Value expressions	
(ValExp ¥ concatenate ¥ ValExp)	ValExp ¥ concatenate ¥ ValExp
Specified instructions	
(SpeIns ; SpeIns)	SpeIns ; SpeIns
Specified declarations	
(SpeDec ; SpeDec)	SpeDec ; SpeDec
Specified class transformers	
(SpeClaTra ; SpeClaTra)	SpeClaTra ; SpeClaTra
Specified preambles of programs	
(SpeProPre ; SpeProPre)	SpeProPre ; SpeProPre

In all these cases the corresponding denotational constructors are associative which implies the adequacy of our transformations. We assume that the corresponding restoring functions will add parentheses from left to right, but, of course, any other strategy may be assumed.

6 Colloquial syntax

6.1 What shall we modify?

On our way from concrete to colloquial syntax we will introduce only two categories of modifications:

1. the omission of parentheses in arithmetic and boolean expressions, in compound conditions and in compound metaconditions,
2. a reconstruction of the declarations of and assignments to arrays and records.

We recall that the introduction of every colloquialism requires the definition of a corresponding restoration function.

6.2 The omission of parentheses

The omission of parentheses takes place in the following syntactic categories:

1. value expressions,
   - 1.1. integer and real value expressions with arithmetic operations,
   - 1.2. integer and real value expressions with comparison operations,
   - 1.3. boolean value expressions with McCarthy's logical operations except not-m,
2. conditions,
   - 2.1. common atomic conditions,
   - 2.2. compound conditions with Kleene's logical operators except not-k,

3. metaconditions,
   3.1. relational metaconditions,
   3.2. behavioral metaconditions,
   3.3. compound metaconditions with classical logical operators except not.

Contrary to the cases of Sec. 5.3 where the omission of parentheses is "forced", in the colloquial syntax the omission is optional. This means that to each clause with omittable parentheses we add a corresponding clause without parentheses, e.g., we write:

(Condition ¥ and-k ¥ Condition) |
Condition ¥ and-k ¥ Condition.

We shall not define the restoration function formally. We assume that in all listed cases we add the "missing" parentheses from left to write and additionally we assume the following priorities of constructors:

1. multiplications and divisions have priorities over additions and subtractions,
2. conjunctions and implications (in all three logical calculi) have priorities over alternative,
3. all binary constructors in relational metaconditions and behavioral metaconditions have priorities over all logical operators except negation.

The described rules of adding parentheses seem to unambiguously define a restoration function. Whether that is really the case, should be checked by trying to write a program that implements this function.

6.3 Colloquialisms with arrays

Let's start from an example. Assume that in a reference expression:

MyObj.HisObj.HerArr

the attribute HerArr of a deep object is of an array type MyType whose declaration is the following:

set MyType = array(integer, 3) **tes** .

Assume further that we want to replace the second element of the indicated array by a new value, say 75. To do that we have to execute the following assignment:

MyObj.HisObj.HerArr := **replace-in** MyObj.HisObj.HerArr **at** 2 **with** 75 **per** (6.3-1)

Here many programmers would probably ask, why can't we simply write

MyObj.HisObj.HerArr[2] := 75? (6.3-2)

The answer is — because in concrete syntax MyObj.HisObj.HerArr[2] is not a reference expression, but a value expression. Consequently, our "assignment" will be rejected by a parser of concrete syntax as not correct.

To allow *array pseudoassignments* like (6.3-2) in **Lingua-D** we have to introduce a colloquialism into the category of instructions by adding to its definition one more clause

SpeInsCol =
… | former clauses
RefExp [ValExp] := ValExp

The corresponding restoration function maps colloquial to concrete specinstructions

restoreArrAsg1 : SpeInsCol ⟼ SpeIns

in such a way that it is an identity function for all cases of specinstructions except array pseudoassignments where it is defined as follows:

restoreArrAsg1.(rex[vex-1] := vex-2) = rex **:= replace-in** rex **at** vex-1 **with** vex-2 **per**

Note that the right-hand-side rex appears in a context where we expect a value expression. Nevertheless, this situation does not lead to a syntactic error, since syntactically reference expressions of the form

Identifier   and

ValExp . Identifier

are also value expressions (cf. (6.3-1)).

Note also that in colloquial syntax MyObj.HisObj.HerArr[2] is not, as before, a reference expression, still (6.3-2) is a (colloquially) correct instruction. Formally, we could have used here an operator different from **:=**. We use **:=** as more intuitive and usual.

It may be worth mentioning in this place that an alternative solution to our (colloquial) transformation might be an extension of the category of concrete reference expressions by new clause

ValExp [ValExp]

in which case our pseudoassignment becomes a particular case of a concrete assignment. That, however, would force a denotational redefinition of arrays as mappings from indices to references rather than to values. Such solution complicates the model of arrays and of array expressions; cf. also de Bakker's paradox described in Sec. 9.4.6.6 of [2]. And, of course, such a transformation would not be an introduction of a colloquialism, but a redefinitions of the denotational model of the language.

Another colloquialism that we associate with arrays will allow writing

MyObj.HisObj.HerArr := [120, 75, 2345]

instead of

MyObj.HisObj.HerArr[1] := 120;
MyObj.HisObj.HerArr[2] := 75;
MyObj.HisObj.HerArr[3] := 2345.

This case requires the introduction of a new carrier in the algebra of colloquial syntax:

ArrayContent =
ValExp | former clauses
ArrayContent , ValExp

and an expansion of the category of specinstructions by a new clause:

SpeInsCol =
…
RefExp **:=** [ArrayContent]

A half formal definition of a corresponding restoration function is the following:

restoreArrAsg2.(rex **:=** [vex-1,…,vex-n]) =
rex := **replace in** rex **at** 1 **with** vex-1 **per**
…
rex := **replace in** rex **at** n **with** vex-n **per**

Note that in this case an assignment like

MyObj.HisObj.HerArr := [120, 2345]

is legal and means

MyObj.HisObj.HerArr[1] := 120;
MyObj.HisObj.HerArr[2] := 2345

whereas the assignment

MyObj.HisObj.HerArr := [120, 75, 2345, 21]

generates an error message 'index beyond range'.

It may be worth noticing in this place that also our second colloquialism could be "replaced" by a modification of the denotational model of our language. We could simply assume that every [vex-1,…,vex-n] is a concrete expression and that its denotation creates an array of dimension n under the condition that the values of all vex-i's are defined and of the same type.

As we see, both our colloquialisms are reducible to concrete-syntax modifications, but in both cases at a "semantical cost". Whether such costs are worth paying is to be decided by language designer. An argument for the idea of colloquialisms may be that a language designer should be free of thinking about syntax when they are building an algebra of denotations and are trying to make it possibly simple.

6.4 Colloquialisms with records

Colloquialisms with records are similar in spirit to these with arrays, although technically they are not quite analogous. In the first place they concern not only assignments, but also type expressions. Let's start from the latter. Instead of building a record type by the following expression:

extend-rc(position, text, **extend-rc**(salary, real **record**(name, text)))

we would like to write:

[name / text, salary / real , position / text]

which would allow us to write the following type declaration:

set employeeType = [name / text, salary / real , position / text] **tes**

To realize this goal we need two new syntactic categories — one of *record type expressions*:

RecTypExp =
  [RecTypContent]

and another one of *record-type contents*:

RecTypContent =
  Identifier / TypExp |
  Identifier / TypExp , RecTypContent.

plus a new clause in the definition of specified declarations:

SpeDecCol =
  … | former clauses
  set ¥ Identifier = RecTypExp ¥ tes

The corresponding restoration function is the following (a half-formal definition):

restoreRecTyp : RecTypExp ⟼ TypExp
restoreRecTyp.[ide-1/tex-1, …,ide-n/tex-n] =
  n = 1 ➔ **record(** ide-1 , tex-1 **)**
  n > 1 ➔ **extend-rc(** ide-n , tex-n , restoreRecTyp.[ide-1/tex-1, …,ide-(n-1)/tex-(n-1)] **)**

We also need a restoration function for type declarations that is an identity everywhere except a colloquial record-type declaration:

restoreRecTypDec : SpeDecCol ⟼ SpeDec
restoreRecTypDec.(**set** ¥ Identifier = RecTypExp ¥ **tes**) =
  set ¥ Identifier = restoreRecTyp.(RecTypExp) ¥ **tes**

Next colloquialism which we may associate with records is analogous to the array pseudoassignment, e.g.,

accountant.name := 'Stanley'

In that case we add a new clause to the definition of specinstructions:

SpeInsCol =
  … | all former clauses (including these for array assignments)
ValExp (Identifier) := ValExp

The corresponding restoration function is then

restoreRecAsg.(rex(ide) := vex) = rex := replace-in-rc rex **at** ide **with** vex **rer**

Here the same comment regarding rex as in Sec. 6.3 is applicable. Also similarly as with arrays we may introduce a colloquialism allowing to write:

accountant := [name / 'Stanley', salary / 10000 , position / 'junior']

We leave the formalization of this colloquialism to the reader.

7 A research problem

Our omission of parentheses in different stages of syntax derivation was based on an intuitive feeling that such transformations will be acceptable for a future parser of our language. We need, therefore, a adequate theory that would allow to take consistent decisions about the omission of parentheses.

8 References

[1] Blikle Andrzej, *Investigations on the logical aspects of ecosystems for programmers in Lingua-V*, a report in progress 2025, https://moznainaczej.com.pl/spotkania-robocze-lingua

[2] Blikle Andrzej, Chrząstowski-Wachtel Piotr, Jabłonowski Janusz, and Tarlecki Andrzej, *A Denotational Engineering of Programming Languages*, a book in progress, 2024, https://moznainaczej.com.pl/what-has-been-done/the-book