

# The Syntax of Lingua-T

(a work in progress)

Andrzej Jacek Blikle  
December 21<sup>th</sup>, 2025

Changes introduced to the version of December 20, 2025 concern colloquial syntax of variable declarations:

`variable $ LisOfIde as $ TypExp sa`

replaced by

`let $ LisOfIde be $ TypExp tel`

plus corresponding changes in preceding syntaxes of variable declarations.

**UWAGA: Osoby czytające ten dokument w edytorze Word powinny ustawić tabulator na 0,5 cm i wszystkie cztery marginesy strony na 1,27 cm. Jeżeli tego nie zrobicie, wzory Wam się rozjadą.**

## Contents

Contents .....	1
1 Introductory remarks.....	3
1.1 How Lingua-T is built .....	3
1.2 Notational conventions .....	3
2 Abstract syntax of Lingua .....	4
2.1 Auxiliary domains.....	4
2.2 Identifiers, class indicators, and privacy statuses.....	5
2.3 Abstract type expressions.....	5
2.4 Abstract value expressions.....	5
2.5 Abstract reference expressions .....	7
2.6 Abstract declaration-oriented categories .....	7
2.7 Abstract signatures .....	7
2.8 Abstract declarations .....	8
2.9 Abstract class transformers .....	8
2.10 Abstract opening of procedures.....	8
2.11 Abstract instructions .....	9
2.12 Abstract preambles of programs.....	9
2.13 Abstract programs .....	9
3 Concrete syntax of Lingua.....	9
3.1 Introductory remarks.....	9
3.1.1 A special treatment of spaces.....	9
3.1.2 General remarks about the way from abstract to concrete syntax .....	10
3.2 First isomorphic concrete syntax .....	12
3.2.1 First concrete type expressions .....	12
3.2.2 First concrete value expressions .....	12
3.2.3 First concrete reference expressions.....	14
3.2.4 First concrete declaration-oriented categories.....	14
3.2.5 First concrete signatures.....	14
3.2.6 First concrete declarations.....	15
3.2.7 First concrete class transformers.....	15
3.2.8 First concrete opening of procedures .....	15
3.2.9 First concrete instructions .....	15
3.2.10 First concrete preambles of programs .....	16
3.2.11 First concrete programs .....	16
3.3 Second isomorphic concrete syntax.....	16
3.4 Homomorphic concrete syntax .....	17

4	The colloquial syntax of Lingua .....	17
4.1	What shall we modify? .....	17
4.2	The omission of optional parentheses .....	17
4.3	Colloquialisms with arrays .....	18
4.4	Colloquialisms with records .....	20
4.5	General remarks about restoring functions .....	21
4.6	Final (full) version of the colloquial grammar of Lingua .....	21
4.6.1	Identifiers, class indicators, and privacy statuses .....	21
4.6.2	Type expressions .....	22
4.6.3	Value expressions .....	22
4.6.4	Reference expressions .....	24
4.6.5	Array contents (new domain) .....	24
4.6.6	Record-type contents (new domain) .....	24
4.6.7	Record-type expressions (new domain) .....	24
4.6.8	Declaration-oriented categories .....	24
4.6.9	Signatures .....	24
4.6.10	Declarations .....	25
4.6.11	Class transformers .....	25
4.6.12	The openings of procedures .....	25
4.6.13	Instructions .....	25
4.6.14	The preambles of programs .....	26
4.6.15	Programs .....	26
5	The syntax of Lingua-V .....	26
5.1	The taxonomy of Lingua-V categories .....	26
5.2	New categories .....	27
5.2.1	Conditions .....	27
5.2.2	Metaconditions .....	28
5.2.3	Anchored class transformers .....	29
5.2.4	Funding class creators .....	29
5.2.5	Covering-type expressions .....	30
5.3	Modified categories .....	30
5.3.1	Specified instructions .....	30
5.3.2	Specified declarations .....	30
5.3.3	Specified class transformers .....	30
5.3.4	Specified preambles of programs .....	31
5.3.5	Specified programs .....	31
5.4	Unchanged categories .....	31
5.4.1	Identifiers, class indicators, and privacy statuses .....	31
5.4.2	Type expressions .....	31
5.4.3	Value expressions .....	32
5.4.4	Reference expressions .....	33
5.4.5	Array contents .....	34
5.4.6	Record-type contents .....	34
5.4.7	Record-type expressions .....	34
5.4.8	Declaration-oriented categories .....	34
5.4.9	Signatures .....	34
5.4.10	The openings of procedures .....	34
6	The syntax of Lingua-T .....	35
6.1	Metavariables .....	35
6.2	The remaining syntactic categories .....	36
7	References .....	36

# 1 Introductory remarks

## 1.1 How **Lingua-T** is built

**Lingua-T** is a metaprogramming language to be used by programmers in the development of correct metaprograms in **Lingua-V**. From a logical perspective, **Lingua-T** is a language of a formalized theory (hence **-T**) 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 to make sure that all models of our theory include standard arithmetics.

Syntactically **Lingua-V** includes the colloquial incarnation of **Lingua**, called **CoLingua** whereas **Lingua-T** includes **Lingua-V**. These relationships may be symbolically written as the following chain of languages:

$$\mathbf{CoLingua} \subseteq \mathbf{Lingua-V} \subseteq \mathbf{Lingua-T} \quad (1.1-1)$$

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

Programmers in **Lingua** will be developing programs in **Lingua-V** using sound program-construction rules (lemmas) formulated in **Lingua-T**. While a correct program in **Lingua-V** is developed, it is “cleared” from its specification, i.e., from conditions and assertions, thus becoming a program in **Lingua**. That program is then executed by an interpreter or compiler. To build each such implementation engine we shall need a restoring function from colloquial to concrete syntax.

According to our method of designing programming languages (described in [2]), the syntax of **CoLingua** is developed from an algebra of denotations in three steps where we develop successively:

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

After these steps **Lingua-V** is developed from **CoLingua** by adding conditions, specified instructions, specified declaration, and metaconditions. The latter include metaprograms. **Lingua-T** is developed from **Lingua-V** by adding metavariables, which we need to formulate sound program-construction rules as axioms or lemmas of our theory. The grammars of **Lingua-V** and **Lingua-T** are created by adding to the grammar of **CoLingua** new syntactic categories with corresponding equations.

Note that whereas **Lingua**, hence also **CoLingua**, are developed from denotations to syntax, **Lingua-V** and **Lingua-T** are developed starting from their syntaxes and assigning them denotations later. This way seems more suitable to make sure that the symbolic inclusions (1.1-1) are satisfied. It is also justified by the fact that the syntaxes of both new languages are “ultimate”, i.e. colloquial.

The main task undertaken in the present paper is to give a first draft of a “sufficiently complete” version of **Lingua-T** to be used in an experimental development of correct programs. The process of programs’ 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-T** grammar. Visual Studio Code has been chosen as a platform where this editor will be implemented.

Of course, 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 that — we hope — should contribute to building a first practical versions of all the three languages.

## 1.2 Notational conventions

Notational conventions that we introduce below are thought to make our grammars reader-friendly. We assume that to make them readable by a parser generator, they may need an appropriate technical reformulation.

- 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.
- $A \mid B$  denotes the union of domains  $A$  and  $B$ .
- $A \odot B$ , or  $A B$ , if that does not lead to confusion, denote 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  $A^* = A^+ \mid \{()\}$  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.,  $A \mid B C^*$  means  $A \mid (B \odot (C)^*)$ .
- Strings of characters typeset in **Arial Narrow** and referred to as *terminals*, denote single-element sets consisting of these string, e.g., **and(** denotes  $\{\text{and}(\}$ . Nonterminals, i.e., the names of syntactic domains are typeset in **Arial**.
- Strings of characters including terminals with nonterminals e.g., **and( ValExp , ValExp )** should be understood, accordingly to the former rules, as  $\{\text{and}(\} \odot \text{ValExp} \odot \{ , \} \odot \text{ValExp} \odot \{ ) \}$

The grammar of **Lingua-T**<sup>1</sup> 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 compared to [1] we omit the postfix ‘-D’ in the names of domains, e.g., we write **ValExp** instead of **ValExp-D**.

## 2 Abstract syntax of Lingua

### 2.1 Auxiliary domains

Syntactic domains of our future grammar will be associated with the carriers of the algebra of denotations **AlgDen** — one syntactic domain for every such carrier. To define them we will need some auxiliary domains that are not derived from **AlgDen** and will not become carriers of none of our syntaxes. We write their definitions already in concrete-syntax style.

LowLetter	= a   b   ...   x   y   z
CapLetter	= A   B   ...   X   Y   Z
Letter	= LowLetter   CapLetter
PositiveDig	= 1   2   ...   9
Digit	= 0   PositiveDig
VisibleSign	= ( )   .   ,   ;   :   -   _   -   !   ?   @   #   \$   %   ^   &   *   +   /   \   “   <   >   =   ≠
InvisibleSign	= ¶   ¥
Sign	= VisibleSign   InvisibleSign
SpecialSymbol	= ‘
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

<sup>1</sup> **Lingua-T** (-T for “theory”) was there called **Lingua-D** (-D for “denotations”). Since **Lingua-T** is a language of a formalized theory, the new name seems more adequate.

## 2.2 Identifiers, class indicators, and privacy statuses

We assume that identifiers in **Lingua** are nonempty strings of letters, digits and the underline symbol:

$$\text{Identifier} = (\text{Letter} \mid \text{Digit} \mid \_ )^+$$

We have to exclude from them all key-words such as **if**, **then**, **else** etc. that appear in colloquial syntax. In short — all **green** prefixes and infixes are such key words.

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

Clalnd = <b>empty-class</b>   Identifier	class indicators
PriStaInd = <b>private</b>   <b>public</b>	privacy-status indicators

We assume that all these equations will be included in the future concrete and colloquial grammars of **Lingua**.

## 2.3 Abstract type expressions

Abstract type expressions evaluate to types and appear in the declarations of type constants, value variables and formal parameters of procedures.

AbsTypExp =

<b>ted-create-bo</b> ()	constant expression with value <b>boolean</b>
<b>ted-create-in</b> ()	constant expression with value <b>integer</b>
<b>ted-create-re</b> ()	constant expression with value <b>real</b>
<b>ted-create-tx</b> ()	constant expression with value <b>text</b>
<b>ted-create-ot</b> ( Identifier )	object type is an identifier (the name of a class)
<b>ted-constant</b> ( Identifier , Identifier )	indicates a type declared in a class
<b>ted-create-li</b> ( AbsTypExp )	evaluates to a list type
<b>ted-create-ar</b> ( AbsTypExp , AbsValExp )	evaluates to an array type
<b>ted-create-rc</b> ( Identifier , AbsTypExp )	evaluates to a record type with one attribute
<b>ted-extend-rc</b> ( Identifier , AbsTypExp , AbsTypExp )	adds an attribute and to a record type

## 2.4 Abstract value expressions

Value expressions evaluate to values, i.e., to typed data and objects. They appear in assignment instructions, abstract attribute concretizations in class declarations and in the declarations of array types.

AbsValExp =

*variables*

<b>ved-variable</b> ( Identifier )	a value variable
<b>ved-attribute</b> ( AbsValExp , Identifier )	the value of an object's attribute

*integer-value expressions*

<b>ved-in</b> ( IntegerNum )	expression with a fixed integer value
<b>ved-plus-in</b> ( AbsValExp , AbsValExp )	integer addition
<b>ved-minus-in</b> ( AbsValExp , AbsValExp )	
<b>ved-times-in</b> ( AbsValExp , AbsValExp )	
<b>ved-divide-in</b> ( AbsValExp , AbsValExp )	

*real-value expressions*

<b>ved-re</b> ( RealNum )	expression with a fixed real value
<b>ved-plus-re</b> ( AbsValExp , AbsValExp )	real addition
<b>ved-minus-re</b> ( AbsValExp , AbsValExp )	
<b>ved-times-re</b> ( AbsValExp , AbsValExp )	

<code>ved-divide-re( AbsValExp , AbsValExp )</code>		
<i>boolean-value expressions</i>		
<i>integer operators</i>		
<code>ved-smaller-in( AbsValExp , AbsValExp )</code>		smaller-than relation for integers
<code>ved-greater-in( AbsValExp , AbsValExp )</code>		
<code>ved-smallerOrEqual-in( AbsValExp , AbsValExp )</code>		
<code>ved-greaterOrEqual-in( AbsValExp , AbsValExp )</code>		
<code>ved-equal-in( AbsValExp , AbsValExp )</code>		
<code>ved-unequal-in( AbsValExp , AbsValExp )</code>		
<i>real operators</i>		
<code>ved-smaller-re( AbsValExp , AbsValExp )</code>		smaller-than relation for reals
<code>ved-greater-re( AbsValExp , AbsValExp )</code>		
<code>ved-smallerOrEqual-re( AbsValExp , AbsValExp )</code>		
<code>ved-greaterOrEqual-re( AbsValExp , AbsValExp )</code>		
<code>ved-equal-re( AbsValExp , AbsValExp )</code>		
<code>ved-unequal-re( AbsValExp , AbsValExp )</code>		
<i>textual operators</i>		
<code>ved-equal-tx( AbsValExp , AbsValExp )</code>		equality relation for texts
<code>ved-unequal-tx( AbsValExp , AbsValExp )</code>		
<i>logical operators</i>		
<code>ved-bo( Boolean )</code>		expression with a fixed boolean value
<code>ved-and-m( AbsValExp , AbsValExp )</code>		McCarthy's (-m) conjunction
<code>ved-or-m( AbsValExp , AbsValExp )</code>		
<code>ved-implies-m( AbsValExp , AbsValExp )</code>		
<code>ved-not-m( AbsValExp )</code>		

*textual-value expressions*

<code>ved-tx( Text )</code>		expression with a fixed textual value
<code>concatenate( AbsValExp , AbsValExp )</code>		the concatenation of two texts

At this moment we define a restricted class of textual-value expressions. In the future it may be enriched by other operators, e.g., texts' inclusion, superfluous space removal, etc.

*list-value expressions*

<code>ved-create-li( AbsValExp )</code>		the creation of a one-element list
<code>ved-push-to-li( AbsValExp , AbsValExp )</code>		pushing an element to the top of list
<code>ved-head-of-li( AbsValExp )</code>		getting an element from the top of list
<code>ved-tail-of-li( AbsValExp )</code>		getting the tail — list with head omitted

*array-value expressions*

<code>ved-create-ar( AbsTypExp , AbsValExp )</code>		the creation of an empty array
<code>ved-replace-in-ar( AbsValExp , AbsValExp , AbsValExp )</code>		the replacement of an element in an array
<code>ved-get-from-ar( AbsValExp , AbsValExp )</code>		getting an element from an array

*record-value expressions*

<code>ved-create-rc( AbsTypExp )</code>		the creation of an empty record
<code>ved-replace-in-rc( AbsValExp , Identifier , AbsValExp )</code>		the replacement of an element in a record
<code>ved-get-from-rc( AbsValExp , Identifier )</code>		getting an element from a record

For the meanings of array- and value expressions see Sec. 4.3 of [2] where operations on typed values are defined.

*structured value-expressions*

<code>ved-conditional( AbsValExp , AbsValExp , AbsValExp )</code>		if-then-else expression
<code>ved-call-fun-pro( Identifier , Identifier , ActPar )</code>		functional-procedure call

An example of an abstract-syntax arithmetic 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 is as follows:

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

## 2.5 Abstract reference expressions

Reference expressions appear in assignment instructions.

`AbsRefExp =`

<code>red-variable( Identifier )</code>		a reference to a value variable
<code>red-attribute( AbsValExp , Identifier )</code>		a reference to an attribute of an object

## 2.6 Abstract declaration-oriented categories

Some auxiliary categories used later in the declarations of value-variables and of procedures.

`AbsLisOfIde =` lists of identifiers

<code>loi-build-empty()</code>	
<code>loi-add-to( Identifier , AbsListOfIde )</code>	

`AbsDecSec =` declaration sections are used in the declarations of many variables of the same type

`dsd-build( AbsLisOfIde , AbsTypExp )`

`AbsForPar =` lists of formal parameters

<code>fpd-build-empty()</code>	
<code>fpa-add-to( AbsDecSec , AbsForPar )</code>	

`AbsActPar =` lists of actual parameters

`apd-build( AbsLisOfIde )`

Intuitively the last equation means that abstract actual parameters are just list of identifiers. Set-theoretically we could have dropped the category `AbsActPar`, and use `AbsLisOfIde` instead, but algebraically we keep it to have the concept of actual parameters in our model.

## 2.7 Abstract signatures

Signatures appear either in stand-alone declarations of signatures in classes or in the declaration of procedures in classes. They indicate the names and the types of formal parameters of procedures (Sec. 6.6 of [2]).

`AbsImpProSig =` the signatures of imperative procedures

`ips-build( AbsForPar , AbsForPar )`

`AbsFunProSig =` the signatures of functional procedures

`fps-build( AbsForPar , AbsTypExp )`

`AbsObjConSig =` the signatures of object constructors

```
ocs-build( AbsForPar , Identifier )
```

## 2.8 Abstract declarations

Declarations listed in this section may be said to be “global” since they create items at the level of states, rather than (locally) in classes.

AbsDec =

<code>ded-skip()</code>		this declaration does nothing
<code>ded-variable( AbsListOfIde , AbsTypExp )</code>		declarations of lists of variables of a common type
<code>ded-enrich-cov-rel( AbsTypExp , AbsTypExp )</code>		the enrichment of (current) covering relations
<code>ded-class( Identifier , ClaInd , AbsClaTra )</code>		declarations of classes
<code>ded-compose( AbsDec , AbsDec )</code>		the sequential composition of declarations

## 2.9 Abstract class transformers

Classes are declared in three steps: first a *parent class* is indicated (by a class indicator), then it is cleared of all items that will not be inherited by the declared class, thus creating a *funding class*, finally the funding class is enriched by new attributes, types and methods. All these enrichments are effectuated by class transformers.

AbsClaTra =

<code>ctd-skip()</code>		does nothing
<code>ctd-add-abs-att( Identifier , AbsTypExp, PriSta )</code>		adds abstract attribute
<code>ctd-con-abs-att( Identifier , AbsValExp )</code>		concretizes abstract att.
<code>ctd-add-con-att( Identifier , AbsValExp , AbsTypExp, PriSta )</code>		adds concrete attribute
<code>ctd-add-abs-typ( Identifier )</code>		adds abs. type constant
<code>ctd-con-typ( Identifier , AbsTypExp )</code>		
<code>ctd-add-con-typ( Identifier , AbsTypExp )</code>		
<code>ctd-add-abs-imp-met( Identifier , AbsImpProSig )</code>		adds a. imp. method
<code>ctd-con-imp-met( Identifier , AbsImpProSig , AbsPro )</code>		
<code>ctd-add-con-imp-met( Identifier , AbsImpProSig , AbsPro )</code>		
<code>ctd-add-abs-fun-met( Identifier , AbsFunProSig )</code>		adds a. fun. method
<code>ctd-con-fun-met( Identifier , AbsFunProSig , AbsPro , AbsValExp )</code>		
<code>ctd-add-con-fun-met( Identifier , AbsFunProSig , AbsPro , AbsValExp )</code>		
<code>ctd-add-abs-obj-met( Identifier , AbsObjConSig )</code>		adds a. object method
<code>ctd-con-obj-met( Identifier , AbsObjConSig , AbsPro )</code>		
<code>ctd-add-con-obj-met( Identifier , AbsObjConSig , AbsPro )</code>		
<code>ctd-compose( AbsClaTra, AbsClaTra )</code>		sequential composition

## 2.10 Abstract opening of procedures

Our programs consist of three segments:

1. a preamble that includes all declarations and possibly some instructions in-between,
2. a single procedure-opening operator,
3. an instruction (possibly compound).

The second component elaborates all pre-procedure of all classes (that may be mutually recursive) thus creating and storing the declared procedures in procedure environment of the current state (Sec. 6.8 of [2]).

AbsProOpe =

```
pod-create-open-pro()
```

## 2.11 Abstract instructions

Instructions modify states by assigning new values to the references of variables.

AbsIns =

<code>ind-skip()</code>		does nothing
<code>ind-assign( AbsRefExp , AbsValExp )</code>		assignment
<code>ind-call-imp-pro( Identifier , Identifier , AbsActPar , AbsActPar )</code>		imperative-procedure call
<code>ind-call-obj-con( Identifier , Identifier , AbsActPar )</code>		object-constructor call
<code>ind-if( AbsValExp , AbsIns , AbsIns )</code>		if-then-else
<code>ind-if-error( AbsValExp , AbsIns )</code>		if-error-then
<code>ind-while( AbsValExp , AbsIns )</code>		while loop
<code>ind-compose( AbsIns , AbsIns )</code>		sequential composition

## 2.12 Abstract preambles of programs

Preambles of programs include declarations and instructions.

AbsProPre =

<code>ppd-make-of-dcd( AbsDec )</code>		make a preamble out of a declaration
<code>ppd-make-of-ind( AbsIns )</code>		make a preamble out of an instruction
<code>ppd-compose( AbsProPre , AbsProPre )</code>		sequential composition

## 2.13 Abstract programs

Programs consist of three segments already describe in Sec. 2.10

AbsPro =

`prd-make-prog( AbsProPre , AbsOpePro , AbsIns )`

# 3 Concrete syntax of Lingua

## 3.1 Introductory remarks

### 3.1.1 A special treatment of spaces

In the context of concrete syntax we have to revise our treatment of spaces. In Sec. 1.2 we have assumed that a metaspace (in grammatical equations) 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 didn't need spaces used as separators.

The situation in concrete syntax is different. With an infix notation we have key-words like *if*, *then*, *else*, etc., which have to be separated from neighboring strings of characters — e.g., representing expressions or identifiers — by *language-level spaces*. To distinguish these spaces from metaspaces at the level of grammars we shall use symbol  $\yen$  to denote language-level spaces. E.g., we shall assume that

`while  $\yen$  ValExp  $\yen$  do  $\yen$  Spelns  $\yen$  od`

stands for

**{while}**⊙{¥}⊙ValExp⊙{¥}⊙**{do}**⊙{¥}⊙Spelns⊙{¥}⊙**{od}**.

An example of a **while**-instruction in concrete-syntax may be, therefore, 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 necessary for a parser (tokenizer) to recognize the structure of this instruction. In colloquial syntax we will additionally assume that each space ¥ may be replaced by a sequence of spaces or by a carriage return. That means that our instruction may appear on the monitor in the following form:

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

All these *superfluous spaces* will be regarded as optional and will be removed 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 introduce the corresponding rules into the grammar.

### 3.1.2 General remarks about the 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 split **A2C** into three steps:

1. first *isomorphic transformation* — more user friendly but with full parenthesizing where:
  - a. some prefixes are shortened or omitted,
  - b. some prefixes are replaced by infixes,
2. second isomorphic transformation — *redundant parentheses* (to be explained) are omitted,
3. a homomorphic transformation — *default parentheses* (to be explained) are allowed to be omitted optionally.

The names of concrete-syntax domains are prefixed with **Con** and postfixed by digits indicating stages from abstract to homomorphic concrete syntax.

In passing from prefix to infix structures we most frequently replace closing parentheses by short strings of characters (up to three) that mirror the initial characters of the corresponding prefixes, like, e.g., in

```
if ConValExp1 then ConIns1 else ConIns1 fi
```

but sometimes we make an exception from this rule like in

```
while ConValExp1 do ConIns1 od
```

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

abstract syntax	isomorphic concrete syntax 1 (new clauses followed by examples)
<b>ted-create-bo</b> — boolean type	<b>boolean</b>
<b>ted-create-ot</b> ( Identifier ) — object type	<b>object</b> ( Identifier ) <b>object</b> (EmployerClass)
<b>ted-extend-rc</b> ( Identifier , AbsTypExp , AbsTypExp )	<b>extend-rc</b> ConTypExp1 <b>at</b> Identifier

	<b>with</b> ConTypExp1 <b>txe</b> <b>extend-rc</b> employee <b>by</b> salary <b>with</b> real <b>txe</b>
ved-plus-in( AbsValExp , AbsValExp )	( ConValExp1 + ConValExp1 ) ( x + (y*z) )
ved-create-ar( AbsTypExp , AbsValExp )	array( ConTypExp1 , ConValExp1 ) array(integer, 3) array(array(integer, 3), ((2*x)+y)) array(myArrayType, ((2*x)+y))
ved-attribute( AbsValExp , Identifier ) — value of object's attribute	( ConValExp1 . Identifier ) (Object1.name1) ((Object1.name1).name2) (((x+y).name1).name2) — syntactically acceptable, but semantically incorrect
ved-get-from-ar( AbsValExp , AbsValExp )	( ConValExp1 [ ConValExp1 ] ) ((Object1.name1).name2)[x+1]
ved-get-from-rc( AbsValExp , Identifier )	( ConValExp1 ( Identifier ) ) ((Object1.name1).name2)(salary)
ved-plus-in( AbsValExp , AbsValExp )	( ConValExp1 + ConValExp1 ) ( x + y ) ((x+y)+(2*z))
red-attribute( AbsValExp , Identifier ) — reference of object's attribute	( ConValExp1 . Identifier ) ((Object1.name1).name2)
ind-assign( AbsRefExp , AbsValExp )	( ConRefExp1 := ConValExp1 ) ( x := (x+1) ) ((Object1.name1) := (x+1))
ind-if( AbsValExp , AbsIns , AbsIns )	<b>if</b> ConValExp1 <b>then</b> ConIns1 <b>else</b> ConIns1 <b>fi</b> <b>if</b> ((x+1)>0) <b>then</b> ( x := (x+2) ) <b>else</b> ( x := (x - 2) ) <b>fi</b>
ind-compose( AbsIns , AbsIns )	( ConIns1 ; ConIns1 ) ((x := (x+1)) ; (y := (x+2)))

**Tab. 3.1.2-1 The first isomorphic transformation**

Note that in the concrete-syntax column, the phrase ( ConValExp1 . Identifier ) is a common element of two different 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 (is not an isomorphism) since these phrases belong to two different categories, and a parser will always “know” which category is currently parsed.

isomorphic concrete syntax 1	isomorphic concrete syntax 2
( ConValExp1 . Identifier ) — get object's attribute value	ConValExp2 . Identifier
( ConValExp1 [ ConValExp1 ] ) — get array's element	ConValExp2[ ValExp ]
( ConValExp1( Identifier ) ) — get record's attribute value	ConValExp2( Identifier )
( ConValExp1. Identifier ) — reference expression	ConValExp2. Identifier

( ConLisOfIde1 $\neq$ as $\neq$ TypExp ) — declaration section	ConLisOfIde2 $\neq$ as $\neq$ ConTypExp2
( ConProPre1 ; ConOpePro1 ; ConIns1 )	ConProPre2 ; ConOpePro2 ; ConIns2
( Identifier . Identifier ( ConActPar1 ) )	Identifier . Identifier ( ConActPar2 )
( Identifier . Identifier )	<b>Identifier . Identifier</b>

**Tab. 3.1.2-2 The second isomorphic transformation**

Intuitively speaking, in the second isomorphic transformation we omit these parentheses (called *redundant*) that are not necessary for a parser to recognize the end of a phrase of a given category. E.g., we do not need to close single-identifier value-expressions in parentheses since they are always separated from their contexts by some adjacent symbols such as **:=**, **if**, **+**, colon, semicolon etc. The concept of *redundant parentheses* requires some theoretical elaboration, that we leave for a future research.

isomorphic concrete syntax 2	homomorphic concrete syntax 3
( ConIns2 ; ConIns2 )	ConIns3 ; ConIns3
( ConDec2 ; ConDec2 )	ConDec3 ; ConDec3
( ConClaTra2 ; ConClaTra2 )	ConClaTra3 ; ConClaTra3
( ConValExp2 $\neq$ concatenate $\neq$ ConValExp2 )	ConValExp3 $\neq$ concatenate $\neq$ ConValExp3

**Tab. 3.1.2-3 The homomorphic transformation**

Homomorphic transformations are “genuinely” glueing, hence make the resulting grammar ambiguous, which means that some phrases may be parsed in more than one way. Still, the corresponding homomorphism is adequate since the denotations of alternative parsing trees are the same. In our case, homomorphic transformation removes parentheses corresponding to two semantically associative operators: the sequential composition of functions represented by semicolon **;** and texts’ concatenation represented by **concatenate**. We call these parentheses *default parentheses* since we assume in this place that a parser will add them in a default way, i.e., from left to right.

## 3.2 First isomorphic concrete syntax

### 3.2.1 First concrete type expressions

In this and in the following subsections domain names are prefixed with **CON** and postfixed with **1**.

ConTypExp1 =

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( ConTypExp1 )		
array( ConTypExp1 , ConValExp1 )		e.g. in type decl. <b>typ MyArray = array(integer, 3) pyt</b>
record( Identifier , ConTypExp1 )		
extend-rc ConTypExp1 by Identifier with ConTypExp1 tex		

### 3.2.2 First concrete value expressions

ConValExp1 =

( Identifier )		variables in <b>Lingua</b> ; grounded value-expressions in <b>Lingua-T</b>
( ConValExp1 . Identifier )		a value assigned to an attribute of an object

*integer-value expressions*

IntegerNum		
( ConValExp1 + ConValExp1 )		
( ConValExp1 - ConValExp1 )		
( ConValExp1 * ConValExp1 )		
( ConValExp1 / ConValExp1 )		integer division returns the integer part of division

*real-value expressions*

RealNum		
( ConValExp1 +. ConValExp1 )		real-value operations are marked with dot “.”
( ConValExp1 -. ConValExp1 )		
( ConValExp1 *. ConValExp1 )		
( ConValExp1 /. ConValExp1 )		

*boolean-value expressions**integer operators*

( ConValExp1 < ConValExp1 )	
( ConValExp1 > ConValExp1 )	
( ConValExp1 <= ConValExp1 )	
( ConValExp1 >= ConValExp1 )	
( ConValExp1 = ConValExp1 )	
( ConValExp1 ≠ ConValExp1 )	

*real operators*

( ConValExp1 <. ConValExp1 )	
( ConValExp1 >. ConValExp1 )	
( ConValExp1 <=. ConValExp1 )	
( ConValExp1 >=. ConValExp1 )	
( ConValExp1 =. ConValExp1 )	
( ConValExp1 ≠. ConValExp1 )	

*textual operators*

( ConValExp1 =t ConValExp1 )	
( ConValExp1 ≠t ConValExp1 )	

*logical operators*

true		
false		
( ConValExp1 ≧ and-m ≧ ConValExp1 )		-m stands for “McCarthy”
( ConValExp1 ≧ or-m ≧ ConValExp1 )		
( ConValExp1 ≧ implies-m ≧ ConValExp1 )		
not-m( ConValExp1 )		

*textual-value expressions*

Text	
( ConValExp1 ≧ concatenate ≧ ConValExp1 )	

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( ConValExp1 )	
push ≧ ConValExp1 ≧ to ≧ ConValExp1 ≧ sup	
head( ConValExp1 )	
tail( ConValExp1 )	

*array-value expressions*

array( ConTypExp1 , ConValExp1 )	
replace-in-ar $\neq$ ConValExp1 $\neq$ at $\neq$ ConValExp1 $\neq$ with $\neq$ ConValExp1 $\neq$ per	
( ConValExp1 [ ConValExp1 ] )	a value of an element of an array

*record-value expressions*

record( ConTypExp1 )	
replace-in-rc $\neq$ ConValExp1 $\neq$ by $\neq$ Identifier $\neq$ with $\neq$ ConValExp1 $\neq$ per	
( ConValExp1 ( Identifier ) )	value of an attribute of a record

*structured value-expressions*

if ConValExp1 $\neq$ then $\neq$ ConValExp1 $\neq$ else $\neq$ ConValExp1 $\neq$ fi	
( Identifier . Identifier ( ConActPar1 ) )	a call of a functional procedure

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

### 3.2.3 First concrete reference expressions

ConRefExp1 =

( Identifier )	
( ConValExp1 . Identifier )	

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

( Identifier . Identifier )	a type assigned to an identifier in a class (Sec. 3.2.1)
( ConValExp1 . Identifier )	the value assigned to an attribute of an object (Sec. 3.2.2)
( ConValExp1 . 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. As we have already explained, these glueing effects are not “destroying” the isomorphicity of our transformation.

### 3.2.4 First concrete declaration-oriented categories

ConLisOfIde1 =

list of identifiers

Identifier	
( Identifier , ConLisOfIde1 )	

ConDecSec1 =

( ConLisOfIde1 as  $\neq$  ConTypExp1 )

ConForPar1 =

( ConDecSec1 )	
( ConDecSec1 , ConForPar1 )	

ConActPar1 =

ActParVar	
( ConLisOfIde1 )	

### 3.2.5 First concrete signatures

ConImpProSig1 =

( val  $\neq$  ConForPar1  $\neq$  ref  $\neq$  ConForPar1 )

```

ConFunProSig1 =
  ( ConForPar1 ¥ return ¥ ConTypExp1 )
ConObjConSig1 =
  ( ConForPar1 ¥ class ¥ Identifier )

```

### 3.2.6 First concrete declarations

```

ConDec1 =
  skip
  let ¥ ConLisOfIde1 ¥ be ¥ ConTypExp1 ¥ tel
  enrich-cov( ConTypExp1 , ConTypExp1 )
  class ¥ Identifier ¥ parent ¥ ClaInd ¥ with ¥ ConClaTra1 ¥ ssalc
  ( ConDec1 ; ConDec1 )

```

### 3.2.7 First concrete class transformers

```

ConClaTra1 =
  skip
  abs-att ¥ Identifier ¥ as ¥ ConTypExp1 : PriSta ¥ sba           | declare abstract attribute
  con-att ¥ Identifier = ConValExp1 ¥ noc                       | concretize abstr. attribute
  att ¥ Identifier = ConValExp1 ¥ as ¥ ConTypExp1 ¥ : PriSta tta | declare concrete attribute
  abs-typ ¥ Identifier ¥ sba                                    | declare abstract type
  con-typ ¥ Identifier = ConTypExp1 ¥ noc                      | concretize abstract type
  typ ¥ Identifier = ConTypExp1 ¥ pyt                          | declare concrete typ
  abs-imp ¥ Identifier ( ConImpProSig1 )                       | declare abstract procedure
  con-imp ¥ Identifier ( ConImpProSig1 ) ConPro1 ¥ noc         | concretize abstract proc.
  imp ¥ Identifier ( ConImpProSig1 ) ConPro1 ¥ pmi             | declare concrete proc.
  abs-fun ¥ Identifier ( ConFunProSig1 )                       | declare abstract function
  con-fun ¥ Identifier ( ConFunProSig1 ) ConPro1 ¥ return ¥ ConValExp1 ¥ noc | concretize abs. f.
  fun ¥ Identifier ( ConFunProSig1 ) ConPro1 ¥ return ¥ ConValExp1 ¥ nuf | declare concr. f.
  abs-cns ¥ Identifier ( ConObjConSig1 )                       | declare abstract obj. const.
  con-cns ¥ Identifier ( ConObjConSig1 ) ConPro1 noc           | concretize abs. obj. const.
  cns ¥ Identifier ( ConObjConSig1 ) ConPro1 snc               | declare concr. obj. const.
  ( ConClaTra1 ; ConClaTra1 )                                  | compose sequentially

```

### 3.2.8 First concrete opening of procedures

```

ConOpePro1 =
  open-procedures

```

### 3.2.9 First concrete instructions

```

ConIns1 =
  skip
  ( ConRefExp1 := ConValExp1 )
  call-pro ¥ Identifier . Identifier ¥ ( val ¥ ConActPar1 ¥ ref ¥ ConActPar1 )
  call-obj ¥ Identifier . Identifier ( ConActPar1 )
  if ¥ ConValExp1 ¥ then ¥ ConIns1 ¥ else ¥ ConIns1 fi
  if-error ¥ ConValExp1 ¥ then ¥ ConIns1 ¥ fi

```

```
while  $\neq$  ConValExp1  $\neq$  do  $\neq$  ConIns1  $\neq$  od
( ConIns1 ; ConIns1 )
```

### 3.2.10 First concrete preambles of programs

```
ConProPre1 =
( ConDec1 ) |
( ConIns1 ) |
( ConProPre1 ; ConProPre1 )
```

### 3.2.11 First concrete programs

```
ConPro1 =
( ConProPre1 ; ConOpePro1 ; ConIns1 )
```

## 3.3 Second isomorphic concrete syntax

In this section we list only the changes introduced in the second isomorphic concrete syntax. Wszystkie przypadki w tabeli trzeba sprawdzić i jakoś ogólnie opisać, bo na razie te nawiasy opuściłem „na nosa” ???.

First isomorphic concrete syntax	Second isomorphic concrete syntax
<b>Type expressions</b>	
( Identifier . Identifier )	Identifier . Identifier
<b>Value expressions</b>	
( Identifier )	Identifier
( ConValExp1 . Identifier )	ConValExp2 . Identifier
( ConValExp1 [ ConValExp2 ] )	ConValExp2 [ ConValExp2 ]
( ConValExp1 ( Identifier ) )	ConValExp2 ( Identifier )
( Identifier . Identifier ( ConActPar1 ) )	Identifier . Identifier ( ConActPar2 )
<b>Reference expressions</b>	
( Identifier )	Identifier
( ConValExp1 . Identifier )	ConValExp2 . Identifier
<b>Instructions</b>	
( ConRefExp1 := ConValExp1 )	ConRefExp1 := ConValExp1
<b>Preambles of programs</b>	
( ConDec1 )	ConDec2
( ConIns1 )	ConIns2
<b>Programs</b>	
( ConProPre1 ; ConOpePro1 ; ConIns1 )	ConProPre2 ; ConOpePro2 ; ConIns2
<b>Declaration-oriented categories</b>	
( Identifier , ConLisOfIde1 )	Identifier , ConLisOfIde2
( ConLisOfIde1 $\neq$ as $\neq$ ConTypExp1 )	ConLisOfIde2 $\neq$ as $\neq$ ConTypExp2
( ConDecSec1 )	ConDecSec2
( ConDecSec1 , ConForPar1 )	ConDecSec2 , ConForPar2
( ConLisOfIde1 )	ConLisOfIde2 (in ConForPar2)
<b>Signatures</b>	
( val $\neq$ ConForPar1 $\neq$ ref $\neq$ ConForPar1 )	val $\neq$ ConForPar2 $\neq$ ref $\neq$ ConForPar2
( ConForPar1 $\neq$ return $\neq$ ConTypExp1 )	ConForPar1 $\neq$ return $\neq$ ConTypExp2

( ConForPar1 $\not\equiv$ class $\not\equiv$ Identifier )	ConForPar2 $\not\equiv$ class $\not\equiv$ Identifier
---	---

### 3.4 Homomorphic concrete syntax

Similarly as in Sec. 3.3 we show only modifications introduced into the second isomorphic concrete syntax.

Second isomorphic concrete syntax	Homomorphic concrete syntax
<b>Value expressions</b>	
( ConValExp2 $\not\equiv$ concatenate $\not\equiv$ ConValExp2 )	ConValExp3 $\not\equiv$ concatenate $\not\equiv$ ConValExp3
<b>Concrete instructions</b>	
( ConIns2 ; ConIns2 )	ConIns3 ; ConIns3
<b>Concrete declarations</b>	
( ConDec2 ; ConDec2 )	ConDec3 ; ConDec3
<b>Concrete class transformers</b>	
( ConClaTra2 ; ConClaTra2 )	ConClaTra3 ; ConClaTra3
<b>Concrete preambles of programs</b>	
( ConProPre2 ; ConProPre2 )	ConProPre3 ; ConProPre3

## 4 The colloquial syntax of **Lingua**

### 4.1 What shall we modify?

On our way from the third, i.e., homomorphic, concrete syntax to colloquial syntax we introduce three groups of modifications:

1. the omission of optional parentheses in arithmetic and boolean expressions,
2. a redefinition of assignments with arrays,
3. a redefinition of the declarations of record types,
4. a redefinition of assignments with records.

Of course, the introduction of every colloquialism requires the definition of a corresponding restoring function.

Since colloquial syntax is an “ultimate” syntax of **Lingua**, we shall not prefix the names of corresponding domains with **Col** and write simply, e.g., **ValExp** instead of **ColValExp**. Note that programmers in **Lingua** do not need to know anything about our way of building colloquial syntax. For them colloquial syntax is just the syntax.

### 4.2 The omission of optional parentheses

The omission of optional parentheses takes place in the following subcategories of value expressions:

1. integer and real value expressions with arithmetic operations,
2. integer and real value expressions with comparison operations,
3. boolean value expressions with logical operations except **not-m**,

Contrary to the cases of Sec. 3.3 where the omission of parentheses is “imposed”, i.e., programmers can’t use them, the “colloquial omissions” are optional. This means that to each clause with omissible parentheses we add a corresponding clause without parentheses, e.g., in the place of

( ValExp  $\not\equiv$  and-m  $\not\equiv$  ValExp )

we write:

( ValExp  $\not\equiv$  and-m  $\not\equiv$  ValExp ) |

$\text{ValExp} \not\asymp \text{and-m} \not\asymp \text{ValExp}$

We shall not define the corresponding restoring function formally assuming that we add the “missing” parentheses from left to right and additionally we respect the following priorities of constructors:

1. multiplication and division have priorities over addition and subtraction,
2. conjunction and implication have priorities over alternative.

The rules described above seem to unambiguously identify a restoring function. Whether that is really the case, should be checked by trying to write a program that implements this function.

### 4.3 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 (within a class declaration) is the following:

`typ MyType = array(integer, 3) pyt .`

Assume further that we want to replace the second element of the indicated array by a new value, say `75`. In concrete syntax such action corresponds to the following assignment:

`MyObj.HisObj.HerArr := replace-in MyObj.HisObj.HerArr at 2 with 75 per` (4.3-1)

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

`MyObj.HisObj.HerArr[2] := 75?` (4.3-2)

The answer is — because in concrete syntax `MyObj.HisObj.HerArr[2]` is not a reference expression, but a value expression. Consequently, our “assignment” is syntactically incorrect.

To allow *array pseudoassignments* like (4.3-2) in colloquial **Lingua** we have to introduce a colloquialism into the category of instructions by adding to its definition one more clause

Instruction =  
 ... | former clauses with appropriately modified domain names  
 RefExp [ ValExp ] := ValExp

The corresponding restoring function maps colloquial to concrete specinstructions

`restoreArrAsg1 : Instruction ↦ ConIns3`

in such a way that it is an identity function for all cases of specinstructions except array pseudoassignments, and in that case 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 expression, which are of the form

Identifier or  
 ValExp . Identifier

are identical with value expressions (cf. (4.3-1)).

Note also that in colloquial syntax `MyObj.HisObj.HerArr[2]` is, as before, not a reference expression, still (4.3-2) is, as a whole, a (colloquially) correct instruction. Formally, we could have used here an operator different from `:=`, but use `:=` as more intuitive.

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 “legal” particular case of a concrete assignment. That, however, would force a (denotational) redefinition of arrays as mappings from indices to references rather than from indices to values. Such solution complicates the model of arrays and of array expressions; cf. also de Bakker’s paradox described in Sec. 9.4.4.6 of [2]. Of course, such 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          |
  ArrayContent , ValExp
```

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

```
Instruction =
  ...
  RefExp := [ ArrayContent ].
```

Here, again, [120, 75, 2345] is not a new sort of expressions and may appear only in the context of an assignment. A half formal definition of a corresponding restoring 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’. In turn, a sequence of assignments:

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

can’t be written as one assignment.

Note that also our second colloquialism could be “replaced” by a modification of the denotational model of our language. We could assume that [ 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 a language designer. An argument for colloquialisms may be that a language designer should be free of thinking about syntax when they are building an algebra of denotations and trying to make it possibly transparent.

## 4.4 Colloquialisms with records

Colloquialisms with records are similar, in spirit, to these with arrays, but technically 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 cumulatively 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:

```
typ employeeType = [name / text, salary / real , position / text] pyt
```

To realize this goal we introduce two new syntactic categories — one of *record-type contents*:

```
RecTypContent =
  Identifier / TypExp          |
  Identifier / TypExp , RecTypContent
```

and another one of *record type expressions*:

```
RecTypExp =
  [ RecTypContent ]
```

Note that record-type expressions are not another case of type expressions, but a new syntactic category! Next we add a new clause to the definition of class transformers:

```
ClaTra =
  ...                               | former clauses
  typ ≠ Identifier = RecTypExp ≠ pyt
```

The restoring function for colloquial type declarations is an identity everywhere except colloquial record-type declaration:

```
restoreRecTypDec : Declaration ↦ ConDec3
restoreRecTypDec.(typ ≠ Identifier = RecTypExp ≠ pyt) =
  typ ≠ Identifier = restoreRecTypExp.(RecTypExp) ≠ pyt
```

where (a half-formal definition):

```
restoreRecTypExp : RecTypExp ↦ ConTypExp3
restoreRecTypExp.[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)] )
```

Note that we do not need a restoring function for `RecTypContent` and `RecTypExp` since there are no corresponding domains in concrete **Lingua**.

Next colloquialism that we introduce for records concerns assignments and is analogous to the array pseudoassignment, e.g.,

```
MyObj.HisObj.HerRec(name) := 'Stanley'
```

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

```
Instruction =
  ...                               | all former clauses (including these for array assignments)
  RefExp ( Identifier ) := ValExp
```

The corresponding restoring function is then

```
restoreRecAsg : Instruction ↦ ConIns3
restoreRecAsg.(rex(ide) := vex) = rex := replace-in-rc rex at ide with vex per
```

Here the same comment regarding **rex** as in Sec. 4.3 is applicable. Also similarly as with arrays we may introduce a colloquialism allowing to write:

```
MyObj.HisObj.HerRec := [name / 'Stanley', salary / 10000.00 , position / 'junior']
```

We leave the formalization of this colloquialism to the reader.

## 4.5 General remarks about restoring functions

Restoring functions are “shortening the distance” between a current syntax and abstract syntax. There are two basic cases to be considered:

1. turning a parenthesized syntax into another form, e.g., turning  
`if  $\forall$  exp  $\forall$  then  $\forall$  ins1  $\forall$  else  $\forall$  ins2 fi` into  
`ind-if( exp , ins1 , ins2 ),`
2. adding removed parentheses, e.g., turning `ins1; ins2` into `(ins1; ins2)`.

In the first case the restoring function may be said to be context-free in the sense that it turns a phrase into another phrase independently of the context where the first phrase appears. In the second case the situation is more complicated since our function must recognize the “boundaries” of the source syntax, e.g., where `ins1` and `ins2` end, to identify the context where parentheses should be introduced.

Similarly to homomorphisms between algebras also restoring functions are many-sorted functions, i.e., families of functions indexed by the sorts of the corresponding algebras. They will be defined as mutually recursive procedures similar to procedures that implement parsers. E.g., a restoring function **PM** assigned to the sort of metaprograms may be defined as follows:

```
MP[pre con1 : spr post con2] = mcd-metaprogram( CN[con1] , SP[spr] , CN[con2] )
```

where **CN** and **SP** restoring functions corresponding to conditions and to specprograms respectively, and where square brackets [ and ] are metabrackets to be distinguished from **Lingua** green parentheses ( and ). The restoring functions should be defined case-by-case following the grammatical clauses of each of syntactical sorts. E.g.:

```
CN[con1  $\forall$  and-k  $\forall$  con2] = ( CN[con1] and-k CN[con2] )
CN[con1  $\forall$  or-k  $\forall$  con2] = ( CN[con1] or-k CN[con2] )
CN[con1  $\forall$  implies-k  $\forall$  con2] = ( CN[con1] implies-k CN[con2] )
CN[not-k( con )] = not-k( CN[con] )
```

where `con1` and `con` are atomic conditions. Note that if `con-1` is an atomic condition, then the parsing of the argument condition under square brackets is unique. Our function **CN** adds parentheses from left to right.

Once **CN** for compound conditions has been defined, we define it for each category of atomic conditions, e.g.:

```
CN[ vex1 < vex2 ] = ( VE[vex1] < VE[vex2] )
```

**A research problem.** Our omission of parentheses at 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, an adequate theory that would allow to take consistent decisions about the omission of parentheses.

## 4.6 Final (full) version of the colloquial grammar of **Lingua**

### 4.6.1 Identifiers, class indicators, and privacy statuses

```
ide : Identifier = ( Letter | Digit | _ )+
cli : ClInd = empty-class | Identifier
psi : PriStalnd = private | public
```

class indicators  
privacy-status indicators

## 4.6.2 Type expressions

tex : TypExp =

```

bool
integer
real
text
object( Identifier )
Identifier . Identifier
list( TypExp )
array( TypExp , ValExp )
record( Identifier , TypExp )
extend-rc TypExp at Identifier with TypExp tex

```

an object type is an identifier (the name of a class)  
a type assigned to an identifier in a class

e.g. in type decl. **set** MyArray = array(integer, 3) **tes**

## 4.6.3 Value expressions

vex : ValExp =

```

Identifier
ValExp . Identifier

```

variables in **Lingua**; grounded value-expressions in **Lingua-T**  
a value assigned to an attribute of an object

*integer-value expressions*

```

IntegerNum
( ValExp + ValExp )
ValExp + ValExp      colloquialism
( ValExp - ValExp )
ValExp - ValExp      colloquialism
( ValExp * ValExp )
ValExp * ValExp      colloquialism
( ValExp / ValExp )
ValExp / ValExp      colloquialism

```

*real-value expressions*

```

RealNum
( ValExp +. ValExp )
ValExp +. ValExp     colloquialism
( ValExp -. ValExp )
ValExp -. ValExp     colloquialism
( ValExp *. ValExp )
ValExp *. ValExp     colloquialism
( ValExp /. ValExp )
ValExp /. ValExp     colloquialism

```

*boolean-value expressions*

*integer operators*

```

( ValExp < ValExp )
ValExp < ValExp      colloquialism
( ValExp > ValExp )
ValExp > ValExp      colloquialism
( ValExp <= ValExp )
ValExp <= ValExp     colloquialism
( ValExp >= ValExp )
ValExp >= ValExp     colloquialism
( ValExp = ValExp )

```

ValExp = ValExp		colloquialism
( ValExp ≠ ValExp )		
ValExp ≠ ValExp		colloquialism

*real operators*

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

*textual operators*

( ValExp =t ValExp )		
ValExp =t ValExp		colloquialism
( ValExp ≠t ValExp )		
ValExp ≠t ValExp		colloquialism

*logical operators*

true		
false		
( ValExp ≠ and-m ≠ ValExp )		-m stands for “McCarthy”
ValExp ≠ and-m ≠ ValExp		colloquialism
( ValExp ≠ or-m ≠ ValExp )		
ValExp ≠ or-m ≠ ValExp		colloquialism
( ValExp ≠ implies-m ≠ ValExp )		
ValExp ≠ implies-m ≠ ValExp		colloquialism
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 of an element of an array

*record-value expressions*

<code>record( TypExp )</code>	
<code>replace-in-rc ValExp at Identifier with ValExp per</code>	
<code>ValExp ( Identifier )</code>	value of an attribute of a record

*structured value-expressions*

<code>if ValExp then ValExp else ValExp fi</code>	
<code>Identifier . Identifier ( ActPar )</code>	a call of a functional procedure

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

#### 4.6.4 Reference expressions

`rex : RefExp =`

<code>Identifier</code>	
<code>ValExp . Identifier</code>	

#### 4.6.5 Array contents (new domain)

`arc : ArrCon =`

<code>ValExp</code>	
<code>ArrCon , ValExp</code>	

#### 4.6.6 Record-type contents (new domain)

`rtc : RecTypCon =`

<code>Identifier / TypExp</code>	
<code>Identifier / TypExp , RecTypCon</code>	

#### 4.6.7 Record-type expressions (new domain)

`rte : RecTypExp =`

`[ RecTypContent ]`

#### 4.6.8 Declaration-oriented categories

`loi : LisOfIde =`

list of identifiers

<code>Identifier</code>	
<code>Identifier , LisOfIde</code>	

`dse : DecSec =`

`LisOfIde  $\forall$  as  $\forall$  TypExp`

`fpa : ForPar =`

<code>DecSec</code>	
<code>DecSec , ForPar</code>	

`apa : ActPar =`

`LisOfIde`

#### 4.6.9 Signatures

`ips : ImpProSig =`

`val  $\forall$  ForPar  $\forall$  ref  $\forall$  ForPar`

```

fps : FunProSig =
  ForPar ¥ return ¥ TypExp
ocs : ObjConSig =
  ForPar ¥ class ¥ Identifier

```

#### 4.6.10 Declarations

```

dec : Declaration =
  skip
  let ¥ LisOfIde ¥ be ¥ TypExp ¥ tel
  enrich-cov( TypExp , TypExp )
  class ¥ Identifier ¥ parent ¥ ClaInd ¥ with ¥ ClaTra ¥ ssalc
  Declaration ; Declaration

```

#### 4.6.11 Class transformers

```

ctr : ClaTra =
  skip
  abs-att ¥ Identifier ¥ as ¥ TypExp : PriSta ¥ sba | declare abstract attribute
  con-att ¥ Identifier = ValExp ¥ noc | concretize abstract attribute
  att ¥ Identifier = ValExp ¥ as ¥ TypExp ¥ : PriSta tta | declare concrete attribute
  abs-typ ¥ Identifier ¥ sba | declare abstract type
  con-typ ¥ Identifier = TypExp ¥ noc | concretize abstract type
  typ ¥ Identifier = TypExp ¥ pyt | declare concrete typ
  typ ¥ Identifier = RecTypExp ¥ pyt | colloquialism
  abs-imp ¥ Identifier ( ImpProSig ) | declare abstract procedure
  con-imp ¥ Identifier ( ImpProSig ) Program ¥ noc | concretize abstract proc.
  imp ¥ Identifier ( ImpProSig ) Program ¥ pmi | declare concrete proc.
  abs-fun ¥ Identifier ( FunProSig ) | declare abstract function
  con-fun ¥ Identifier ( FunProSig ) Program ¥ return ¥ ValExp ¥ noc | concretize abs. f.
  fun ¥ Identifier ( FunProSig ) Program ¥ return ¥ ValExp ¥ nuf | decl. con. f.
  abs-cns ¥ Identifier ( ObjConSig ) | declare abstract obj. const.
  con-cns ¥ Identifier ( ObjConSig ) Program noc | concretize abs. obj. const.
  cns ¥ Identifier ( ObjConSig ) Program snc | declare concr. obj. const.
  ClaTra ; ClaTra | compose sequentially

```

#### 4.6.12 The openings of procedures

```

opr OpePro =
  open-procedures

```

#### 4.6.13 Instructions

```

ins : Instruction =
  skip
  RefExp := ValExp
  RefExp [ ValExp ] := ValExp | array colloquialism
  RefExp := [ ArrayContent ] | array colloquialism

```

RefExp ( ValExp ) := ValExp	record colloquialism
call-pro $\not\equiv$ Identifier . Identifier $\not\equiv$ ( val $\not\equiv$ ActPar $\not\equiv$ ref $\not\equiv$ ActPar )	
call-obj $\not\equiv$ Identifier . Identifier ( ActPar )	
if $\not\equiv$ ValExp $\not\equiv$ then $\not\equiv$ Instruction $\not\equiv$ else $\not\equiv$ Instruction fi	
if-error $\not\equiv$ ValExp $\not\equiv$ then $\not\equiv$ Instruction $\not\equiv$ fi	
while $\not\equiv$ ValExp $\not\equiv$ do $\not\equiv$ Instruction $\not\equiv$ od	
Instruction ; Instruction	

#### 4.6.14 The preambles of programs

pre : ProPre =

Declaration	
Instruction	
ProPre ; ProPre	

#### 4.6.15 Programs

pro : Program =

ProPre ; OpePro ; Instruction

## 5 The syntax of **Lingua-V**

### 5.1 The taxonomy of **Lingua-V** categories

The transformation from colloquial **Lingua** to **Lingua-V** consists in adding some “genuinely” new grammatical categories, in expending some categories of colloquial **Lingua** and in taking some categories of **Lingua** unchanged. This leads to the following taxonomy of the syntactic categories of **Lingua-V**:

1. new categories,
  - 1.1. conditions,
  - 1.2. metaconditions,
  - 1.3. anchored class-transformers,
  - 1.4. funding-class creators,
  - 1.5. covering-type expressions
2. modified (extended) categories
  - 2.1. specified instructions,
  - 2.2. specified declarations,
  - 2.3. specified class transformers,
  - 2.4. specified preambles of programs,
  - 2.5. specified programs,
3. unchanged categories
  - 3.1. identifiers, class indicators and privacy-status indicators,
  - 3.2. all three categories of expressions,
  - 3.3. record-type contents,
  - 3.4. record-type expressions,
  - 3.5. declaration-oriented categories,
  - 3.6. signatures.

The syntax of all these categories is ultimate, hence colloquial, and their semantics — denotational. Conditions and metaconditions are “fully parenthesized”, since otherwise their axiom system would become hardly manageable.

## 5.2 New categories

### 5.2.1 Conditions

Conditions describe properties of states. To better understand their nature we split them into a few subcategories:

1. atomic conditions,
  - 1.1. common atomic conditions — atomic value expressions with boolean values,
  - 1.2. special atomic conditions — they go beyond boolean expressions and may describe any mathematically definable properties of data, e.g., that a list is increasingly ordered, or a database is consistent,
2. compound conditions — the combination of atomic conditions by means of Kleene’s propositional connectives.

Below we show only some basic conditions. The development of their more complete list requires experiments with programming in **Lingua-V**.

con : Condition =

*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* (see Sec. 9.2 of [2])

( ValExp == ValExp )		an abstract (polymorphic) mathematical equality
ord-in( Identifier )		increasingly ordered list of integers
ord-re( Identifier )		increasingly ordered list of reals
ord-tx( Identifier )		increasingly ordered list of texts
( CovExp ≠ is-current )		
( ForPar ≠ well-valued-in ≠ CovExp )		
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 $\neq$ is-child-of $\neq$ ClInd )		
consistent( TypExp , TypExp )		
compatible( RefExp , ValExp )		
( Identifier $\neq$ is-free)		
<b>NT</b>		nearly true

*procedure oriented conditions*

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

*algorithmic conditions*

( SpePro @ Condition )	
( Condition @ SpePro )	

*compound conditions*

true	
false	
( Condition $\neq$ and-k $\neq$ Condition )	
( Condition $\neq$ or-k $\neq$ Condition )	
( Condition $\neq$ implies-k $\neq$ Condition )	
not-k( Condition )	

## 5.2.2 Metaconditions

Intuitively, metaconditions describe the properties of conditions. We split them into two categories:

- *atomic metaconditions*
- *compound metaconditions*

Also in this case their definitions includes some “initial list” of cases. A practical list should be elaborated in the course of an experimental program development.

mco : MetCon =

*relational metaconditions*

( Condition => Condition )		stronger
( Condition <=> Condition )		weakly equivalent
( Condition $\neq$ LD $\neq$ Condition )		less defined
( Condition $\neq$ SE $\neq$ Condition )		strongly equivalent
( Condition => Condition $\neq$ WNV $\neq$ Condition )		WNV – whenever
( Condition <=> Condition $\neq$ WNV )		
( Condition $\neq$ SE $\neq$ Condition $\neq$ WNV $\neq$ Condition )		

*pre-post metaconditions*

( pre $\neq$ Condition : SpePro $\neq$ post $\neq$ Condition )		metaprograms
( pre $\neq$ Condition : SpeIns $\neq$ post $\neq$ Condition )		metainstructions
( pre $\neq$ Condition : SpeDec $\neq$ post $\neq$ Condition )		metadeclarations

( pre $\neq$ Condition : SpeProPre $\neq$ post $\neq$ Condition )		metapreambles
( pre $\neq$ Condition : AncClaTra $\neq$ post $\neq$ Condition )		meta anchored cl. tr.
( pre $\neq$ Condition : FunClaCre $\neq$ post $\neq$ Condition )		meta fun.-cl. creator

*behavioral metaconditions*

( Condition $\neq$ insures-LR-of $\neq$ SpeIns )	
( Condition $\neq$ resilient-to $\neq$ SpePro )	
( Condition $\neq$ nourishing $\neq$ SpePro )	
( Condition $\neq$ catalyzing-for $\neq$ SpePro )	
( Condition $\neq$ essential-for $\neq$ SpePro )	
( Condition $\neq$ irrelevant-for $\neq$ pre $\neq$ Condition : SpePro $\neq$ post $\neq$ Condition )	

*temporal metaconditions*

( Condition $\neq$ primary-in $\neq$ pre $\neq$ Condition : SpePro $\neq$ post $\neq$ Condition )	
( Condition $\neq$ induced-in $\neq$ pre $\neq$ Condition : SpePro $\neq$ post $\neq$ Condition )	
( Condition $\neq$ hereditary-in $\neq$ pre $\neq$ Condition : SpePro $\neq$ post $\neq$ Condition )	
( Condition $\neq$ co-hereditary-in $\neq$ pre $\neq$ Condition : SpePro $\neq$ post $\neq$ Condition )	
( Condition $\neq$ perpetual-in $\neq$ pre $\neq$ Condition : SpePro $\neq$ post $\neq$ Condition )	

*language-related metaconditions*

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

*compound metaconditions*

( MetCon $\neq$ and $\neq$ MetCon )	
( MetCon $\neq$ or $\neq$ MetCon )	
( MetCon $\neq$ implies $\neq$ MetCon )	
not( MetCon )	

Note that in this case logical connectives are classical.

### 5.2.3 Anchored class transformers

Anchored class transformers do not appear in **Lingua** but we need them in **Lingua-V** to formulate a rule for the construction of correct class declarations (Sec. 9.4.5.4 of [2])

```
act : AncClaTra =
    SpeClaTra in Identifier
```

Class transformers are components of class declarations and are used to modify (enrich) successive incarnations of a future class that is being declared. These incarnations are indicated by an identifier, i.e., the name of the future class. The denotations of class transformers are functions that given an identifier return a state-to-state transformation. Anchored class transformers “are anchored” to a concrete identifier. Their denotations are state-to-state transformations.

### 5.2.4 Funding class creators

Funding class creators play a similar role as anchored class transformers. They transform states by assigning funding class to a given identifier of a class that is being declared.

```
fcc : FunClaCre =
    funding Identifier of parent ClaInd
```

## 5.2.5 Covering-type expressions

ConCovExp =

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

## 5.3 Modified categories

### 5.3.1 Specified instructions

Specified instructions (abbr. specinstructions) are like colloquial instructions but may optionally include assertions plus ON and OFF assertion blocks.

sin : Spelns =

```
skip |
asr Condition rsa | assertions
asr ¥ Condition ¥ in ¥ Spelns ¥ rsa | ON assertions bocks
off ¥ Condition ¥ in ¥ Spelns ¥ on | OFF assertions bocks
RefExp := ValExp |
RefExp [ ValExp ] := ValExp | colloquialism
RefExp := [ ArrayContent ] | colloquialism
RefExp ( ValExp ) := ValExp | colloquialism
call-pro ¥ Identifier . Identifier ( val ¥ ActPar ¥ ref ¥ ActPar ) |
call-obj ¥ Identifier . Identifier ( ActPar ) |
if ¥ ValExp ¥ then ¥ Spelns ¥ else ¥ Spelns fi |
if-error ¥ ValExp ¥ then ¥ Spelns ¥ fi |
while ¥ ValExp ¥ do ¥ Spelns ¥ od |
Spelns ; Spelns
```

### 5.3.2 Specified declarations

Specified declarations (abbr. specdeclarations) are declarations with optionally nested assertions, and with specclastransformers in the place of class transformers.

sde : SpeDec =

```
skip |
asr Condition rsa |
let ¥ LisOfIde ¥ be ¥ TypExp ¥ tel |
enrich-cov( TypExp , TypExp ) |
class ¥ Identifier ¥ parent ¥ ClaInd ¥ with ¥ SpeClaTra ¥ ssalc |
Declaration ; Declaration
```

### 5.3.3 Specified class transformers

Specified class-transformers (abbr. spectransformers) differ from class transformers only in that they refer to specprograms rather than to programs. However, unlike specinstructions and specdeclarations they do not include assertions as a subcategory.

sct : SpeClaTra =

```
skip |
abs-att ¥ Identifier ¥ as ¥ TypExp : PriSta ¥ sba | declare abstract attribute
con-att ¥ Identifier = ValExp ¥ noc | concretize abstract attribute
att ¥ Identifier = ValExp ¥ as ¥ TypExp : PriSta tta | declare concrete attribute
```

<code>abs-typ</code> $\neq$ Identifier $\neq$ <code>sba</code>	declare abstract type
<code>con-typ</code> $\neq$ Identifier = TypExp $\neq$ <code>noc</code>	concretize abstract type
<code>typ</code> $\neq$ Identifier = TypExp $\neq$ <code>pyt</code>	declare concrete typ
<code>typ</code> $\neq$ Identifier = RecTypExp $\neq$ <code>pyt</code>	colloquialism
<code>abs-imp</code> $\neq$ Identifier ( ImpProSig )	declare abstract procedure
<code>con-imp</code> $\neq$ Identifier ( ImpProSig ) SpePro $\neq$ <code>noc</code>	concretize abstract procedure
<code>imp</code> $\neq$ Identifier ( ImpProSig ) SpePro $\neq$ <code>pmi</code>	declare concrete procedure
<code>abs-fun</code> $\neq$ Identifier ( FunProSig )	declare abstract function
<code>con-fun</code> $\neq$ Identifier ( FunProSig ) SpePro $\neq$ <code>return</code> $\neq$ ValExp $\neq$ <code>noc</code>	concretize abstract function
<code>fun</code> $\neq$ Identifier ( FunProSig ) SpePro $\neq$ <code>return</code> $\neq$ ValExp $\neq$ <code>nuf</code>	declare concrete function
<code>abs-cns</code> $\neq$ Identifier ( ObjConSig )	declare abstract obj. const.
<code>con-cns</code> $\neq$ Identifier ( ObjConSig ) SpePro <code>noc</code>	concrete abstract obj. const.
<code>cns</code> $\neq$ Identifier ( ObjConSig ) SpePro <code>snc</code>	declare concrete obj. const.
<code>SpeClaTra</code> ; <code>SpeClaTra</code>	compose sequentially

### 5.3.4 Specified preambles of programs

```
spp : SpeProPre =
  Declaration           |
  SpeIns                |
  SpeProPre ; SpeProPre
```

Note that specified preambles may include assertions nested in-between declarations because the latter belong to the category of specinstructions.

### 5.3.5 Specified programs

```
spr : SpePro =
  SpeProPre ; OpePro ; SpeIns
```

## 5.4 Unchanged categories

### 5.4.1 Identifiers, class indicators, and privacy statuses

```
ide : Identifier = ( Letter | Digit | _ )+
cli : Clalnd    = empty-class | Identifier           class indicators
psi : PriStalnd = private | public                 privacy-status indicators
```

### 5.4.2 Type expressions

```
tex : TypExp =
  bool
  integer
  real
  text
  object( Identifier )
  Identifier . Identifier
  list( TypExp )
  array( TypExp , ValExp )
  record( Identifier , TypExp )
  extend-rc TypExp at Identifier with TypExp tex
```

an object type is an identifier (the name of a class)  
a type assigned to an identifier in a class

e.g. in type decl. `set MyArray = array(integer, 3) tes`

### 5.4.3 Value expressions

vex : ValExp =

Identifier

ValExp . Identifier

| variables in **Lingua**; grounded value-expressions in **Lingua-T**  
| a value assigned to an attribute of an object

*integer-value expressions*

IntegerNum

( ValExp + ValExp )

ValExp + ValExp

( ValExp - ValExp )

ValExp - ValExp

( ValExp \* ValExp )

ValExp \* ValExp

( ValExp / ValExp )

ValExp / ValExp

*real-value expressions*

RealNum

( ValExp +. ValExp )

ValExp +. ValExp

( ValExp -. ValExp )

ValExp -. ValExp

( ValExp \*. ValExp )

ValExp \*. ValExp

( ValExp /. ValExp )

ValExp /. ValExp

*boolean-value expressions*

*integer operators*

( ValExp < ValExp )

ValExp < ValExp

( ValExp > ValExp )

ValExp > ValExp

( ValExp <= ValExp )

ValExp <= ValExp

( 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

( ValExp >=. ValExp )

ValExp >=. ValExp

( ValExp =. ValExp )

ValExp =. ValExp

( ValExp ≠. ValExp )

ValExp ≠. ValExp

*textual operators*

( ValExp =t ValExp )

ValExp =t ValExp

( ValExp ≠t ValExp )

ValExp ≠t ValExp

*logical operators*

true

false

( ValExp ≠ and-m ≠ ValExp )

ValExp ≠ and-m ≠ ValExp

( ValExp ≠ or-m ≠ ValExp )

ValExp ≠ or-m ≠ ValExp

( ValExp ≠ implies-m ≠ ValExp )

ValExp ≠ implies-m ≠ ValExp

not-m( ValExp )

-m stands for “McCarthy”

*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 of an element of an array

*record-value expressions*

record( TypExp )

replace-in-rc ≠ ValExp ≠ at ≠ Identifier ≠ with ≠ ValExp ≠ per

ValExp ( Identifier )

value of 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 are in their parentheses.

#### 5.4.4 Reference expressions

rex : RefExp =

Identifier

ValExp . Identifier

### 5.4.5 Array contents

arc : ArrCon =  
 ValExp |  
 ArrCon , ValExp

### 5.4.6 Record-type contents

rtc : RecTypCon =  
 Identifier / TypExp |  
 Identifier / TypExp , RecTypContent

### 5.4.7 Record-type expressions

rte : RecTypExp =  
 [ RecTypContent ]

### 5.4.8 Declaration-oriented categories

loi : LisOfIde = list of identifiers  
 Identifier |  
 Identifier , LisOfIde

dse : DecSec =  
 LisOfIde **as** ¥ TypExp

fpa : ForPar =  
 DecSec |  
 DecSec , ForPar

apa : ActPar =  
 LisOfIde

### 5.4.9 Signatures

ips : ImpProSig =  
**val** ¥ ForPar ¥ **ref** ¥ ForPar

fps : FunProSig =  
 ForPar ¥ **return** ¥ TypExp

ocs : ObjConSig =  
 ForPar ¥ **class** ¥ Identifier

### 5.4.10 The openings of procedures

opr OpePro =  
 open-procedures

## 6 The syntax of **Lingua-T**

### 6.1 Metavariables

The difference between **Lingua-V** and **Lingua-T** is that the latter includes metavariables running over the denotations of the former. With every category of **Lingua-V**, with an exception of procedure opening, we assign a corresponding category of metavariables. Intuitively, a metavariable of a given category may appear in a term or formula of **Lingua-T** wherever a syntactic element of that category may appear. Let's take as an example the following specified instruction in **Lingua-V**:

```
if x < 1 then x := x+1 else asr x > 0 rsa ; x := x < 1 fi ; asr x > 0 rsa
```

In **Lingua-T** it represents a grounded term, because includes no metavariables. Examples of corresponding free terms in **Lingua-T** are the following:

```
if x < 1 then $ins$ else x := x < 1 fi ; asr $con$ rsa
if $vex$ then x := x+1 else x := x < 1 fi ; asr x > 0 rsa
if $ide$ < 1 then x := x+1 else x := x < 1 fi ; asr x > 0 rsa
if $vex$ then $ins$1 else $ins$2 fi ; asr $con$ rsa
...
```

To make metavariables “visible” by a future parser of **Lingua-T**, we assume that each of them includes a *sort indicator* closed in-between two \$ signs, followed by a (possibly empty) label. For instance, **\$ins\$1** is an instruction metavariable with sort indicator **\$ins\$** and label **1**. The domains of metavariables are the following

#### New categories

ConMetVar	= \$con\$ Label	condition metavariables
MetConMetVar	= \$mco\$ Label	metacondition metavariables
AncClaTraMetVar	= \$act\$ Label	anchored class-transformer metavariables
FunClaCreMetVar	= \$fcc\$ Label	funding-class creators metavariables

#### Modified categories

SpeInsMetVar	= \$ins\$ Label	specinstruction metavariables
SpeDecMetVar	= \$sde\$ Label	specdeclaration metavariables
SpeClaTraMetVar	= \$sct\$ Label	specified-class-transformer metavariables
SpeProPreMetVar	= \$spp\$ Label	specified program-preamble metavariables
SpeProMetVar	= \$spr\$ Label	specprogram metavariables

#### Unchanged categories

IdeMetVar	= \$ide\$ Label	identifier metavariables
ClaIndMetVar	= \$cli\$ Label	class indicator metavariables
PriStaIndMetVar	= \$psi\$ Label	privacy statuses indicator metavariables
TypExpMetVar	= \$tex\$ Label	type-expression metavariables
ValExpMetVar	= \$vex\$ Label	value-expression metavariables
RefExpMetVar	= \$rex\$ Label	reference-expression metavariables
ArrConMetVar	= \$arc\$ Label	array-content metavariables
RecTypConMetVar	= \$rtc\$ Label	record-type-content metavariables
RecTypExpMetVar	= \$rte\$ Label	record-type-expression metavariables
LisOfIdeMetVar	= \$loi\$ Label	list-of-identifiers metavariables
DecSecMetVar	= \$dse\$ Label	declaration section metavariables
ForParMetVar	= \$fpa\$ Label	formal-parameter metavariables
ActParMetVar	= \$apa\$ Label	actual-parameter metavariables
ImpProSigMetVar	= \$ips\$ Label	imperative-procedure signature metavariables
FunProSigMetVar	= \$fps\$ Label	functional-procedure signature metavariables
ObjConSigMetVar	= \$ocs\$ Label	object-constructor signature metavariables

## 6.2 The remaining syntactic categories

The remaining syntactic categories of **Lingua-T** one-one correspond to the categories of **Lingua-V** and differ from them only by the appearance of metavariables. Consequently, from each grammatical equation of **Lingua-V** we create a corresponding equation of **Lingua-T** by adding to the former a clause corresponding to metavariables. For simplicity we keep the former names of syntactic domains while giving them new meanings. Note that in this case we have to accordingly modify also the domains of identifiers, class indicators and privacy statuses, which earlier were common for all syntaxes.

Identifier =		
IdeMetVar		
( Letter   Digit   _ ) <sup>+</sup>		
Clalnd =		
ClalndMetVar		
empty-class		
Identifier		
PriStalnd =		
PriStalndMetVar		
private		
public		
TypExp =		
TypExpMetVar		
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. <b>set</b> MyArray = array(integer, 3) <b>tes</b>
record( Identifier , TypExp )		
extend-rc TypExp at Identifier with TypExp tex		
ValExp =		
ValExpMetVar		
Identifier		
ValExp . Identifier		
<i>integer-value expressions</i>		
IntegerNum		
( ValExp + ValExp )		
ValExp + ValExp		
...		

We leave the remaining equations to the reader.

## 7 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, Chrzastowski-Wachtel Piotr, Jablonowski Janusz, and Tarlecki Andrzej, *A Denotational Engineering of Programming Languages*, a book in progress, 2024, <https://moz-nainaczej.com.pl/what-has-been-done/the-book>