Language for Grammatical Rules

Abstract: The *Language for Grammatical Rules* is a formalism that allows linguists to describe (not only) surface syntax rules of a natural language. The main objective of the formalism is to be as simple as possible (so that the linguists could use it without difficulties) and to keep its computational power at the same time.

The formalism is designed in two levels. The lower level covers the basic language functions that keeps a linguist away from the typical problems such as data formats, character sets, data structure administration and so on. The higher level contains the functions which enable the linguists to write linguistic rules directly — these functions use the formalism available on the lower level.

Pavel Květoň

Short Contents

1 Motivation and History

I assume that everybody who reads this handbook understands the importance of natural language corpora — either for the linguistic research, or for their indirect exploitation in many commercial applications (e.g. for training the stochastic models of speech recognition, for developing frequency dictionaries etc.).

Generally, every corpus increases its value if it is annotated (if the annotation is correct and relevant, of course) — in the case of natural language corpora the *annotation* mainly means morphological, syntactic or semantic annotation, i.e. assigning grammatical interpretations to the words (*positions*) in the corpus, creating grammatical structures,...

A language corpus can contain several kinds of errors — either annotation errors, or the errors present already in the source text. It depends on the particular application to what extent these errors are considered fatal for the results. It is natural to assume that the corpora with fewer errors (while the notion of *errors* and the notion of the corpus *quality* are still very hazy terms) are useful both for the linguistic research and for the corpora-based applications[1].

There are many methods for the build-up and annotation of a language corpus. To minimize errors in the corpus it seems best to employ an experienced linguists to annotate the corpus "manually". It is a matter of course, however, that modern stochastic methods need corpora containing hundreds of millions of words or even billion words and that it is not possible to analyze the corpora of such size by hand, unless we have thousands of linguists at our disposal.

The idea of the Language for Grammatical Rules (abbrev. LanGR in the sequel) has originated in the Institute of the Czech National Corpus (ICNC) at the time when Czech National Corpus (more precisely, the corpus of synchronic Czech labeled SYN2000) was annotated by stochastic taggers with the error-rate oscillating around 5 %. Stochastic taggers are able to annotate huge amounts of data in a relatively short time, but they still produce a lot of errors which block the possibilities of further using the corpus for more demanding and sophisticated applications.

Thus, a group of linguists decided to develop a bunch of rules that could detect and remove some of the fatal errors committed by stochastic taggers used till then. This set of rules has been rewritten to the C++ programming language and applied with very interesting results.

As soon as the practical use showed that the hand-made rules led to significant improvement of the tagging accuracy, it was decided that it was not feasible to rewrite the rules manually into a compilable programming language. We wanted the rules to be written by linguists themselves in a very powerful programming language (formalism), but at the same time these rules should not constrain a linguist too much. Our objective was to develop a powerful programming language for linguists which could be automatically compiled and executed. This language is described in the present publication.

Acknowledgement: The work was supported by the financial support granted to the Institute of the Czech National Corpus and to the Institute of Formal and Applied Linguistics (grant no. 405/03/0913 of the Grant Agency of the Czech Republic). LanGR would never work without the assistance of the linguists — Vladimir Petkevič and Karel Oliva.

[1] With the exception of the research focused on error-detection or on a similar objective.

2 Overview

LanGR wants to be a compromise between the convenience and usage-simplicity for linguists on the one hand and the linguists' independence of information technologies on the other hand. It is possible to design a formal language in such a way that a linguist need not study its intricate formal properties, but each simple extension of such a language will be paid by the work of information technology (IT) specialists. Moreover, the specialist in programming will probably directly encode the specific features of the natural language being described, but in this way he risks to commit errors while encoding the rules.

On the other hand, we could teach linguists to use any programming language with the power of the Turing machine and let the IT people rest in peace (which is, with respect to the huge amount of linguists that could write syntactic rules, unrealistic).

So, the priority is to keep LanGR syntactically simple — simple rules should be easily readable for a linguist that has never seen a manual for LanGR, but they should be also readable for a non-linguist. We want to eliminate the noisy element, i.e. the programmer, too.

We have chosen two-level schema of LanGR:

1. **LanGR application** — the set of functions, data types and variables defined in LanGR core. The functions should be defined in such a way that most of the rules could be as simple as if they were written on paper. If a rule cannot be written in LanGR-application, it implies that such a rule is a very special one or new functions should be incorporated into LanGR-application. The new functions can be written by anyone who is acquainted with LanGR core. LanGR-application should also contain all language-specific information (definitions of linguistic notions, the set of morphological categories, various dictionaries, valency frames, ...).
2. **LanGR core** — a procedural language with the power of the Turing machine like C/C++ or Pascal. To work in LanGR core will be apparently difficult for the linguists but they will probably never need to use LanGR core. LanGR core must, however, enrich the existing languages in the features missing in them, e.g. the branching of the computation, minimizing the problems with character sets, predicate expressions. LanGR core will be natural-language independent.

This manual describes only LanGR core. The manual for the particular LanGR-application will be language-dependent and the direct cooperation of the linguists is expected there.

3 The Language Definition

3.1 Lexical symbols

Lexical symbols are the basic building blocks of the programmer, and we shall start our description with them. All characters mentioned in this section are special — they should be used with caution:

`\` backslash, see Section 3.1.3 [Strings], page 5 and Section 3.1.5 [Numbers], page 5,

`"` double-quotes, see Section 3.1.3 [Strings], page 5,

`;` semi-colon and

`{ }` curly brackets, see Section 3.5 [Expressions and commands], page 33, Section 3.2.7 [Structure], page 12 and Section 3.2.6 [Domain], page 11,

`.` dot, see Section 3.1.5 [Numbers], page 5, Section 3.2.7 [Structure], page 12 and Section 3.4.1 [Function identifiers], page 23,

`[ ]` square brackets, see Section 3.2.8 [Set], page 16,

`,` comma, see Section 3.2.7 [Structure], page 12, Section 3.2.8 [Set], page 16, Section 3.4.7 [Using functions], page 28,

`'` quotes, see Section 3.1.4 [Regular expressions], page 5,

`white-characters`
see Section 3.1.1 [White characters], page 3,

`( )` parentheses, see Section 3.5.5 [Expressions], page 36 and Section 3.4.7 [Using functions], page 28.

3.1.1 White characters

A white character is always a string of non-zero length, composed exclusively of ASCII characters `0x20` (space), `0x0a` (line-feed), `0x0d` (carriage-return), `0x09` (tab). In other words, any number of spaces, tabs and newlines is identical with the single space in LanGR.

White characters separate two identifiers in the program. In some cases a white character could be omitted — a white character *could* be omitted (on both sides) near the semi-colon, comma, hash, dot, double-dot and left parenthesis, on the left side immediately before all right brackets and on the right side immediately after the left curly bracket and the left square bracket.

3.1.2 Comments

Comments in LanGR could be written in two ways (in the same ways as in the C++): anything after `//` (double slash) to the end of the line is a comment, anything between `/*` and `*/` is a comment.

`//` has higher priority than `/**/`, so `/*` or `*/` placed after `//` on the line lose their special meaning.

The only place where all comment boundaries lose their meaning are strings and regular expressions.

3.1.2.1 Highlighting

Comments are the best (and the only) place for the description of functions, rules and groups that linguists write. It could sometime happen that there are more comments than the programming code in a file and comments become unreadable. No programming language has a possibility to avoid it, although, LanGR cooperates with web-interface that allows us to use some formatting and highlighting of the comments.

Hence, a linguist can use several typesetting directives in comments — these directives (similarly as comments) have no effect on the interpretation of the code itself, but they could be translated into HTML directives and the comments are thus better visible by means of an HTML browser.

The directives are defined in TeX-like format (`\action`, or `\action{arguments}`). The following actions are recognized:

`\bf{text}`
: `text` will be printed in bold,

`\it{text}`
: `text` will be printed in italics,

`\color{col}{text}`
: `text` will be printed in color `col` (available colors are `red`, `green`, `blue`, `yellow`),

`\par` new paragraph will start here,

`\newline` new line will start here

`\itemize \item text \enditemize`
: an unnumbered enumeration,

`\example text \endexample`
: `text` will be printed in a highlighted font, commonly `typewritter`.

The exhaustive table of the actions supported can be always obtained directly from the compiler, see Section 4.1.4 [Invoking the compiler], page 48.

If an `action` is not recognized by the compiler, it is left untouched, i.e. `\action` will be printed.

3.1.2.2 Examples and counter-examples

In the rules, it is always helpful to give examples of sentences that are related to the rule in question. An example of the sentence where a rule applies will be (in a comment) enclosed in `\appliesto{sentence}`. A counter-example (a rule does not apply in the sentence) will be enclosed in `\doesnotapplyto{sentence}`.

The examples will be (of course) somehow highlighted. In future, they could be also used to test the rule functionality.

In every comment, there could be an unlimited amount of examples and counter-examples.

3.1.3 Strings

String constant is a sequence of characters enclosed in double-quotes (`"`), e.g.:

```
"string constant"
```

All special characters (except for `"` and `\`) lose their special meaning in a string constant.

A backslash (`\`) in a string constant can be followed by `"`, `\` (this results into one character `"` or `\` in the constant) or by an integer followed again by `\`, e.g.:

```
\<integer>\
```

Here `<integer>` must be a non-negative non-decimal number. The whole sequence `\<integer>\` results in one character in the constant (the character that is assigned the code `<integer>` in the input alphabet, see Section 4.1.1 [Character sets], page 46). An example:

```
"const\97\nt"
// if the input is plain ASCII, the result will be "constant"
```

3.1.4 Regular expressions

Not yet implemented.

3.1.5 Numbers

A number constant in LanGR always begins with `0-9` or with `-` and contains only the following characters: `+`, `-`, `.`, `x`, `X`, `e` and `E` (the letter case makes no difference).

The separator of integer and decimal part of a number is the dot (`.`). Also, exponent notation can be used for shortening:

```
<int>.<dec>e<exp>
```

Here `<int>` is the integer part, `<dec>` the decimal part and `<exp>` the exponent — the base (commonly `10`) is raised to the power of `<exp>` and `<int>.<dec>` is multiplied by the result.

The exponent `<exp>` must always start with `+` or `-` and rest of the exponent can contain only numbers `0-9`.

If a number constant contains `x` or `X`, the base of the number is changed (the default base is naturally `10`):

```
<base>x<number>
```

The `<number>` is processed as a number in the base `<base>` (including the exponent, if any). If `<base>` is greater than `10`, alphabetical characters can/must be used in `<number>`, starting from `a` (or `A`). The amount of accepted characters is `<base>-10`. If `<base>` is less than or equal to `10`, then `<number>` can use only characters from `0` to `<base>-1`. From practical reasons, the `<base>` must lie between `2` and `32` (inclusive).

Leading zeroes are ignored both in `<base>`, `<int>` and `<dec>`.

Examples:

```
10.1
25
0.76E-20
16xA1FE-1
8x7052e+3
2x1001.101011
-18
-6x5142
```

Counter-examples:

```
10 . 1
0.76 E-20
0.76E -20
16xGH12
8x0194
45X43652DRF
5AB
5!45623
1axbda
1.5e12
.13
- 15
```

3.1.6 Boolean values

Boolean values are `true` and `false`, see Section 3.2.5 [Predex], page 8.

3.1.7 Identifiers

In LanGR, everything apart from special characters, strings and numbers is an *identifier* — e.g. a variable name, function name, data type name.

An identifier cannot contain a special character and cannot start with `0-9`. The letter case is not significant in any identifier. The identifiers of functions can consist of several *words*, see Section 3.4.1 [Function identifiers], page 23.

3.1.7.1 Validity scope

Every identifier has its *validity scope*, i.e. an area in which the identifier is valid. All identifiers are valid (with non-modifiable meaning) globally in the entire program — except for local *variables*.

The validity of local variables is limited to the command sequence (i.e. the enclosing curly brackets `{}`) where the variable is declared (the variable is valid from the moment of declaration, of course), including optional more deeply nested command sequences.

Since the command sequences are usually nested, we talk about *nesting levels*. The global level is 0, function bodies are 1, and so on. More deeply nested levels are called *higher* levels and the global level is *the lowest* one.

An example:

```
private integer i; // global variable 'i'

integer function secti integer a integer b {
k = a + b; // local variable 'k'
i = k; // global 'i' is assigned the value of local 'k'
return k;
};
```

3.1.7.2 Meaning of identifiers

This section is purely technical and it concerns itself with the algorithm that the compiler uses for identifying the meaning of identifiers. Before reading this part, the reader should already be acquainted with the chapter Section 3.4.1 [Function identifiers], page 23.

Syntactic analysis of the source code is performed in two stages (runs). In the first run (*stage I*) the code is divided into single commands. In the second run (*stage II*), the individual commands are processed and the syntax tree is built.

Every identifier encountered in *stage I* could have already known meaning, or its meaning must be identified. For such an unknown identifier, the compiler tries to find a file with the name identical to the identifier except for the suffix (i.e. for an identifier `id`, the file expected is `id.rlm`) — if such a file is found, the compiler processes it immediately. If an identifier is still unknown after the file has been processed (i.e. the file does not contain the definition of this identifier) — it is a syntactic error.

If the file with the identifier's definition is not found, the compiler assumes that the identifier is used as a variable (free variable in a predicate expression or an additional variable), even though the type of the variable need not to be known at this time.

The only exception are the function headers, where an undefined identifier can be assumed as both an argument name and an additional word. These possibilities are unequivocally separated from each other by the previous identifier (if the previous identifier is a type identifier, the current one is an argument name. Otherwise the identifier is an additional word).

An advice: The files should be as short as possible, i.e. they should contain the definition of one identifier only (with possible references to other identifiers constituting the definition). Just let the compiler find the identifiers alone.

3.2 Data types

There are *basic* data types (`integer`, `string`, `predex` (predicate expressions) and `regex` (regular expressions)) and *user-defined* data types (derived via `set`, `structure` or `domain` from simpler types) in LanGR.

The new data type can be defined by the command `type`:

```
type <new_name> <type_definition>;
```

where `<type_definition>` depends on the type we define. A data type can be defined only on the global level.

3.2.1 Value types

The result of an expression (a variable, function call etc.) is always a value. LanGR distinguishes three classes of values:

`undef` undefined value (no value set);

`N/A`

`NotApplicable`
: this value is used mainly for categories, that are irrelevant for a given word, e.g. the category of person for nouns. `NotApplicable` is equivalent to `N/A`;

a particular value
: has the following meaning: different from `N/A` and `undef` — this is a value of some data type.

The function `defined` specifies whether a value is defined or not. It returns `false` for `undef` and `true` for any other value.

The function `applicable` returns `true` for both `undef` and `N/A`, `false` for all other values (even dynamic ones). See also Section 3.4.8.1 [Comparison operations], page 30.

3.2.2 Integer

It represents any number, either decimal or integer, negative or positive[1]. The value of type `integer` is always identified with the maximum permissible range (limited by the computer's capacity).

Note: Up-to-date implementation of `integer` uses `double` for the storage of the value, hence the range is limited more explicitly.

3.2.3 String

A sequence of characters in the internal encoding.

3.2.4 Regex

Not yet implemented.

3.2.5 Predex

Predicate expressions (PE) represent logical formulae in LanGR. The result of a PE is typically a boolean value `true` or `false`, but it can also be a dynamic value (e.g. in case the PE contains some free variable or links, see Section 3.5.5 [Expressions], page 36). All PEs are values of the data type `predex`.

The function `constant <predex>` returns either `false` for dynamic expressions, or `true` for boolean constants. The function `constant` neither checks whether a dynamic predex

[1] *number* could be a better name for the data type, but *number* is already reserved for the morphological category of Number

could be evaluated in the given program state, nor even checks whether the expression is a tautology or a contradiction. It just distinguishes a static expression from a dynamic one.

We call the expressions without any quantifier *boolean* expressions, see Section 3.2.5.2 [Boolean expressions], page 9, for the others see Section 3.2.5.4 [Quantifiers], page 10. In LanGR, only *predicate expressions* are available, subsuming both abovementioned expressions.

3.2.5.1 The power of PE

First of all we shall look at the *boolean* and *predicate* expressions, their content and differences between them.

Boolean expression contain values, function calls and logical conjunctions. The values could be either variables or constants.

Predicate expressions are boolean expressions enriched by *quantifiers*. A quantification always runs over some value range. As soon as the range is known (and finite) the quantifiers could be rewritten to a boolean expression using the conjunctions `and` and `or`. Infinite domains are not relevant for us (at least since our data are always finite). On the other hand, the possibility of specifying the range of the quantified variable later in the program definition could be very interesting, see Section 3.2.5.4 [Quantifiers], page 10.

The power of the predicate expressions in LanGR consists in how it processes the variables. A user can (of course) point to all pieces of information stored in the data and in the program. Moreover, the value of the variable can be used directly, or just a *reference* to the variable can be used — this makes the whole expression dynamic and allows for its evaluation later in different situations and different values.

The approach described is very strong and it has a lot of weak and hard-to-manage points, mainly in the implementation. But the advantages are evident.

On the other hand, if we allow a quantification over any kinds of values, we need no other language and we can write the rules directly as the predicate expressions whenever we introduce the possibility of specifying macro-definitions and modifying data (the operator `=`). Hence we allow for only "reasonable" quantification over homogeneous sets (which is, however, exactly what the theory of predicate expressions allows).

3.2.5.2 Boolean expressions

A boolean expression is any expression that doesn't use quantifiers and which results in a boolean value. The simplest boolean expression is the boolean value `true` or `false`.

Boolean expression can be extended by either using variables or using functions (comparison, see Section 3.4.8.1 [Comparison operations], page 30, or logical conjunctions `and`, `or`, `not`, `implies`, `xor`). User is, however, allowed to define any other logical conjunction (a function with `predex` arguments that returns again a `predex` value).

3.2.5.3 Free variables

In *boolean expression*, free variables can be used. Free variables are identifiers that do not represent any existing entity in the given program state (e.g. a local variable is not

a free variable). The value of a free variable is set later at the evaluation time — either directly or by a quantifier.

A free variable need not be declared, just use its identifier. This could be a bit dangerous while writing the names of identifiers, since any misspelling identifies the identifier with a free variable. These errors are found mostly at run-time (since such a free variable is not assigned a value).

A free variable is valid only where it has been used. Its data type is undefined until a value is set to the variable.

The value for a free variable is set by the function `where`:

```
<predex_expr> where <var> is <val>
```

The function `where` assigns the identifier `<var>` (a free variable) the value `<val>` which is valid in the whole expression `<predex_expr>`. Assigning two different values `<val>` to `<var>` results in an error.

The value of the free variable `<var>` from the `where` command is propagated to the whole `<predex_expr>`, but also (if `<predex_expr>` is a dynamic expression) to the root node of its result. The value of `<var>` *remains defined* in every node of `<predex_expr>` also if `<predex_expr>` is used as a part of another expression, e.g. if `<predex_expr>` evaluates in such a way that some of its parts are quantified.

3.2.5.4 Quantifiers

The quantifiers are operators that make it possible to analyze properties of a homogeneous set. The quantification over a structure makes no sense since the fields are typically of different types. Three quantifiers are available: `exists`, `forall` and `amount of`, used in the notation:

```
<quantifier> <varname> <predex>
```

Here `<quantifier>` is one of the quantifiers and `<varname>` is an identifier of the quantified variable. The quantified variable has no formal relation to `<predex>`, but it can be expected that it will appear there as a free variable.

The range of the variable `<varname>` is set similarly to the free variable:

```
<predex1> where <varname> in <set_value>
```

Every quantifier in `<predex1>` that quantifies over `<varname>` will use `<set_value>` as a range for `<varname>`.

In practice, the quantifier goes through the whole range `<set_value>` and for each `<item>` from the range it executes `<predex1> where <varname> is <item>` — it depends on the particular quantifier and the result whether it either continues the processing of the set or it stops.

Note: `<predex1> where <varname> is <item>` is being executed until a constant value is obtained, similarly as with `if` and other conditional commands.

The value of `<varname>` is propagated to `<predex1>` in the same way as for the free variables, see above. The identifiers from `where-in` can collide both with identifiers from `where-is` and `where-in` (but in this case the identifiers must share the same value).

Note: As a `<set_value>` in `where-in` neither `undef`, nor `N/A` value can be used and it must be a value of some set type. If `<set_value>` is an empty set, then without any

processing of `<predex1> forall` is always `true`, `exists` is always `false` and `amount of` is always zero.

Note: `where-in` is allowed only for the set types that have been defined somewhere in the program. If for instance the program doesn't use a set of integers, then `where ... in <set_of_integers>` cannot be used.

The quantifiers are interpreted as follows:

`forall` returns `true` if for the whole range of `<varname>` the expression `<predex1> where <varname> is <item>` is `true`,

`exists` returns `true` if there exists a value `<item>` in the range of `<varname>` such that `<predex1> where <varname> is <item>` is `true`,

`amount of` returns the amount of items `<item>` from the range of `<varname>` such that `<predex1> where <varname> is <item>` is `true`.

Note: Both `where` functions take `predex` as its first argument. The quantifier `amount` returns `integer` and hence it cannot be used directly as an argument to `where` (moreover, `amount` cannot be the root node of any dynamic expression). The problem can be solved by the construction

```
(<var> == amount of <varname> in <predex>) where <varname>
in <set_value>,
```

which is a correct dynamic expression with the free variable `<var>`.

3.2.6 Domain

The `domain` construction defines a domain (set of possible values) for a new data type:

```
type <typename> domain {<val1>,<val2>,...,<valn>};
```

The items of a domain are always identifiers, the letter case is not significant. The data type of `<vali>` is incompatible with any other type (even with `string` type) and domain types are also mutually incompatible. Every `<vali>` is a global identifier, see Section 3.1.7 [Identifiers], page 6[2].

An example:

```
type TCase domain { Nom, Gen, Dat };
```

The definition of a domain type `<typename>` implies an automatic definition of the data type `set <typename>` — this is a set type with items from `<typename>`. Within this type also global *shared* variable `domain <typename>` is defined — a set of all items of the given domain type `<typename>`.

For instance, the abovementioned `TCase` will imply the data type `set TCase`, as if

```
type <set TCase> set TCase;
```

were used (although, the user cannot write it explicitly in LanGR, since `<set TCase>` has to be a single identifier without white characters). Similarly, the variable `domain TCase` would be defined as follows:

```
shared <set TCase> <domain TCase>;
<domain TCase> = [ Nom, Gen, Dat ];
```

[2] It is mainly the identifiers in the domain type which must differ from the other ones in all other domains.

3.2.7 Structure

A structure stores heterogeneous items, like structure in C/C++ or `record` in Pascal. Every item of a structure takes a name (in LanGR we call it a *category*), see Section 3.2.7.2 [Categories], page 12. The values of categories can differ in different instances of the same structure data type.

Note: The structure data type is the only possibility of how to return more than one value (of different types) from a function, see Section 3.4.6 [Return value], page 27.

3.2.7.1 Structure type definition

New structure type is defined as follows:

```
type <typename> structure {
<type1> <cat1>;
<type2> <cat2>;
...
<typen> <catn>;
};
```

The above structure type `<typename>` contains categories `<cat1>`...`<catn>` of the corresponding types `<type1>`...`<typen>`. The data types `<typei>` must be predefined. Every structure type must contain at least one category.

Similarly as the domain type, the definition of a new structure type causes an automatic definition — a domain data type that contains the list of all category names is created, see Section 3.2.6 [Domain], page 11 and Section 3.2.7.2 [Categories], page 12.

The structure types can be freely nested, e.g.:

```
type <my_2nd_type> structure {
<typename> <inner_struct>;
<some_type> <item2>;
};
```

The power of structures increases in combination with sets. While defining sets of structures, it is possible to define some categories as *keys*, that play key role in structure comparison, see Section 3.4.8.1 [Comparison operations], page 30, and indirectly also in selecting an item from a set, see Section 3.2.8.3 [Item selection], page 17.

3.2.7.2 Categories

The categories are the names of items of a structure type. Continuing the example from previous section, there are names `<cat1>`,...,`<catn>` of types `<type1>`,...,`<typen>` forming the structure type `<typename>`.

The *category* denotation came into existence from the primary usage of a structure in LanGR — the structure collects linguistic categories relevant for a particular natural language being processed —- e.g. *case*, *gender*, *person*.

With respect to the expected wide use of the category names in the source code files, the categories are defined as global identifiers. Formally, with every structure type `<typename>` there comes domain type `fields <typename>` that contains all the categories from the structure `<typename>` — with this domain type `set fields <typename>` and `domain fields <typename>` are, of course, implicitly defined, see Section 3.2.6 [Domain], page 11.

Every category `<cati>` thus represents two data types — `<typei>` (the data type of the item of the structure) and `fields <typename>`. The particular data type used must be known at least at run-time (mostly at compile-time) — if the data type cannot be determined, `fields <typename>` is used by default! The default behavior can be always overridden by the construction `field(<cati>)`.

Note: If there is a `shortcut` defined in the structure type (see Section 3.2.8.5 [Shortcut], page 18), its name is also a category.

3.2.7.3 Structure instance definition

Both regular and additional variables of structure types can be declared in the same way as the variables of other data types. The categories are defined always as regular (no item of a structure can be defined additionally, see Section 3.3 [Variables], page 20).

Sometimes it is meaningful to access the whole structure as one object — when it is assigned a structure constant or a structure variable or when it is passed as an argument to a function (including I/O operations).

The values of a structure can be set from another structure trivially by an assignment (`=`). The structure on the left side will be a copy of the structure on the right side.

Generally, a *structure constant* of type `<typename>` from previous section is defined as follows:

```
{ <cat1> = <val1>,..., <catk> = <valk>, <valk+1>,...,<valn> },
```

where `<cati>` is a category identifier and `<vali>` is the assigned value (potentially an expression), for `0 <= i <= n`.

It can be seen that the items of a particular structure can be assigned their values either by using the category name or not. Both notations can be mixed in an arbitrary way.

While creating the structure constant, the compiler goes left-to-right through the expressions in the constant. If a *named* definition `<cati> = <vali>` is found, the corresponding category is set. Unnamed items `<vali>` are set from the beginning of the structure (independently of the named definitions).

In the case of conflicts (both named and unnamed definitions set the same category), it is a compile-error, e.g.:

```
type MorphAnalysis structure {
string gender;
string number;
};
private MorphAnalysis i;
i = { gender = "Masc", number = "Sing" }; // ok
i = { number = "Sing", gender = "Masc" }; // ok, the same
i = { gender = "Masc", "Fem" }; // error, both define 'gender'
```

The data type of the structure constant must be known at compile-time. It can be determined either from the named category definition (since categories are globally unique identifiers) or from the context (from the argument type expected by the calling function). If the data type remains unknown, it is an error.

An advice: If the compiler reports an undetermined data type of a structure constant, it can always be solved by defining a local variable of the particular type — the variable is set to the constant and later used.

An unnamed definition of a category being set to the value `undef` can be performed explicitly by specifying `undef` in the constant, or by complete omitting the definition. If there are only `undef`s at the end of the constant, they can also be omitted.

```
private MorphAnalysis i;
i = { undef,undef };
i = { , undef };
i = { undef, };
i = { , };
i = { }; // all these specifications have the same meaning!
i = undef; // NOT the same meaning, different!
```

Note: `undef` and `N/A` values take special meaning in comparison, see Section 3.4.8.1 [Comparison operations], page 30.

3.2.7.4 Accessing categories

The categories of a structure `<struct>` can be accessed separately by a dot-construction `<struct>.<cat>`. It is always required that `<struct>` is defined (i.e. NOT `undef`) for this access. E.g.:

```
private MorphAnalysis i;
i = {};
i.gender = "Masc"; // ok

i = undef;
i.gender = "Masc"; // error

type MA structure {
MorphAnalysis variant1;
MorphAnalysis variant2;
};

private MA j;
j.variant1.gender = i.gender; // error, j is undef
j = {};
j.variant1.gender = i.gender; // error, j.variant1 is undef
j.variant1 = {};
j.variant1.gender = i.gender; // ok
```

There can be also an expression returning a structure type on the left side of the dot operator (`.`). On the right side of the dot operator (i.e. as a category specification) there can also appear the whole expression — just be careful with the parentheses here.

The dot operator cannot be overloaded due to its special meaning.

3.2.7.5 Categories without structures

There are two powerful additional ways of accessing the categories in the dynamic expressions — in both of them it is expected that the particular structure instance will be specified later (see below):

- the function `field <cat>` returns the value of the category `<cat>` in the structure (like the dot operator without the left side);

- the use of a standalone category identifier without any operator — just as a variable of the same data type as the item identified by the category in the structure type.

An example:

```
private predex p;
p = field(gender) == "Masc";
p = gender == "Masc";

declare getCat;
priority getCat 100; // greater than '=='
fields MorphAnalysis function getCat {
// something returning the categories of
// MorphAnalysis
};

p = field getCat == "Masc";
```

The expression stored in `p` cannot be evaluated now — until a structure is specified by the `use` command:

```
private MorphAnalysis stru;
stru = { gender = "Masc", number = "Sing" };
if p use stru then { print "OK"; };
```

The command `use` adds to its first argument the information that all unknown categories should be taken from the structure instance `stru`, and returns the enriched expression (like `where` commands, see Section 3.2.5.4 [Quantifiers], page 10).

The `use` command can be nested — the information about the structures is accumulated and for each category the corresponding structure is used (depending on the data type). However, in the nested `use` command two instances of the same data type cannot appear together (unless they are exactly the same).

The values `undef` and `N/A` are accepted as a values for `use`, but they add no information to the expression. Neither non-structure values, nor dynamic ones are accepted.

3.2.7.6 Keys

Like in the databases, LanGR also uses the *keys*. The keys are critical for the comparison of structures (similarly as primary keys in the databases).

The comparison of two structure instances (of the same type, of course) is performed exactly in the key categories — if two structures share the key values, they are treated as identical.

Note: If there is no key defined in the structure type, it could be expected that all instances of this type are identical. Since sometimes it is not possible to find out the acceptable key category but we need to store a lot of these structures to a set and access them only sequentially, the expected behavior has been changed — if there is no key category, every instance of such a type is *different* from any other instance (even if they are "optically" identical).

The keys are defined in the structure by means of the specifier `key`:

```
type Tposition structure {
integer index;
key string form;
TMorphAnal ma;
};
```

Two structure instances of `Tposition` will now be identical whenever they share the value in `form` (which is, however, probably not a good example from linguistic...).

3.2.8 Set

The sets in LanGR replace the databases. Every set is an *ordered* homogeneous group of *items*. The items are always instances of a data type, e.g. `string` or typically some data type derived by `structure`.

There are several possibilities of accessing the items of sets, see Section 3.2.8.5 [Shortcut], page 18, Section 3.2.8.3 [Item selection], page 17, Section 3.2.8.6 [Sequential access], page 19.

For sets, no implicit operations are defined (such as intersection or union), but they will probably be implemented soon.

3.2.8.1 Set type definition

New set type is defined as follows:

```
type <newtype> set <itemtype>;
```

where `<itemtype>` is the type of the items — either one of the basic types (`integer`, `string`) or derived by `domain` or `structure`, e.g.:

```
type SetOfIntegers set integer;
```

A set type cannot be built over the data types without comparison — `predex` and all set types.

3.2.8.2 General set features

It is always guaranteed that no set instance can contain two identical items (by the implicit comparison over the data type of the items, see Section 3.4.8.1 [Comparison operations], page 30). Trying to insert an existing item in the set is a run-time error.

No set can contain either an `undef` item, or a `N/A` item — inserting such a value into a set is a run-time error. This is (in addition other reasons) because undefined values identify the end of items in the sequential access. `undef` and `N/A` values can be, of course, formally inserted into a structure that will form the items.

All the operations over sets assume that *defined* sets are arguments (neither `undef`, nor `N/A`) and that a set is not specified by a dynamic expression. The same restriction is applied almost always also to all other arguments of the set functions (with the exception of functions working with underspecified items).

The amount of items actually present in the set is returned by the function `cardinality <set_value>`. The function `empty <set_value>` checks whether the set is empty.

Note: The type of a set passed as an argument to the functions need not be necessarily known at compile time (contrary to the structures), since it can happen that the data

type of items can be recognized at run-time and then the set type is well recognized too (unless a user defines two set types based on the identical item type — but this is the user's problem...).

3.2.8.3 Item selection

The existence of a value `<item>` in the set `<set_value>` can be tested by

```
<item> member of <set_value>
```

which returns `true` in case the value is present in the set and `false` otherwise. It uses the implicit value comparison, see Section 3.4.8.1 [Comparison operations], page 30.

The sets based on the structure items can be analyzed also by using the underspecified comparison, see Section 3.4.8.1 [Comparison operations], page 30, by the function

```
get underspecified <item> from <set_value>,
```

which returns a subset (either empty) of `<set_value>` containing all items that match `<item>`.

Note: If `<item>` is an undefined value `undef` or `N/A`, than due to the definition of underspecification the original set value is returned.

3.2.8.4 Inserting/Removing items

Only defined values of the correct data type can be inserted into a set instance (i.e. neither `undef`, `N/A`, nor dynamic value can be inserted). Also, the value must not be already present in the set.

An new value is inserted by

```
insert <value> into <set>;
```

Here, of course, `<value>` must be of the same data type as the items of `<set>`. The set `<set>` must be defined (either empty). A very common problem is then

```
private SetOfIntegers iset;
insert 10 into iset;
```

Here `iset` is not defined; hence in order to make the above code correct, the line `iset = [];` must be inserted in between the two lines.

This looks useless. However, the operation `insert` is now symmetric to `remove` which cannot make an undefined set out of any defined set.

Symmetrically to `insert` there is a `remove` operation:

```
remove <item> from <set>;
```

If the `<item>` is present in the `<set>`, it is removed. The sets over structure items can also use an underspecified removal:

```
remove underspecified <item> from <set>;
```

See `get underspecified` in the previous section. The functions `insert` and `remove` return nothing. The value `<set>` in the construction above must be a left side value, as in the assignment operation.

3.2.8.5 Shortcut

And we are now back at the databases. A *shortcut* is a structure field that functions as a primary key in the set:

```
type MyStrType structure {
// some fields
shortcut MuStrTypeIndex;
// some other fields
};
```

Every structure type can contain at most one shortcut. Formally, a shortcut is not a key, hence it doesn't participate in the implicit comparison. However, once the shortcut is defined, the set instance ensures that the each shortcut value is unique in the set (no two items can share the defined shortcut value).

On the other hand, it is always allowed to insert items with undefined shortcut into the sets (although these items are then inaccessible by simplified notation, see below) — but once defined, the shortcut value must be unique and different from `N/A`.

The base difference between the access via `shortcut` and the sequential access, see Section 3.2.8.6 [Sequential access], page 19, is that `shortcut` is completely in the user's competence. Shortcut value is never automatically changed. It is kept even while writing the structure into the file or whatever, unless the user changes it.

For structure operations, `shortcut` is a normal category of type `integer`.

Finding a free value for `shortcut` could be sometimes difficult. To this end, the system offers the function

```
get unused from <set>;
```

This function is defined for all sets containing a shortcut in the definition. It always returns the smallest integer value that is greater than any other existing shortcut in `<set>`. For empty sets it returns zero.

To access an item by shortcut, use the construction

```
<set>#<shrtct>
```

If used as right side value (an expression), it returns a value of an item determined by the shortcut `<shrtct>` in the set `<set>` (or `undef`, if such an item is not present in the set).

If used as left side value, the item with the given shortcut is modified. If the item with the shortcut `<shrtct>` doesn't exists, it is an error. If the new value contains a shortcut different from `<shrtct>`, it is also an error.

Note: It is appropriate to note that shortcut could be also a decimal value, but this case is usually not very interesting.

An item could be also removed using its shortcut:

```
remove <shrtct> from <set>;
```

Technical note: The function `remove <shrtct> from <set>` is, of course, overloaded only for the set data types derived from structures containing a shortcut. Hence, it cannot happen that the following overloads will collide:

```
remove <item> from <set>
remove <shrtct> from <set>
```

Every set with shortcut can be also processed using the shortcut:

```
get after <int> from <set>
get before <int> from <set>
```

The first function returns a shortcut value that exists in `<set>` such that it is greater than `<int>`, the second one returns a shortcut value that is lower than `<int>`. If there is no such shortcut in the set, `undef` is returned.

It must be said that such an access to items is not very useful, see better Section 3.2.8.6 [Sequential access], page 19.

3.2.8.6 Sequential access

Every set can be processed sequentially independently of the data type of its items. The order of the items returned is never guaranteed, but it is guaranteed that the user will get all items, every item exactly once (followed by `undef` value marking the end of the set). If the set is modified during the access, the result of the operation is unsafe (some items can be made invisible for the user).

The advantage is that the sequential access is always very fast and it is independent of any other features of the set.

The sequential access is realized by

```
<set>:<index>
```

where `<set>` is a set to be accessed and `<index>` is a non-negative integer value. If `<index>` is lower than `cardinality <set>`, the result of `<set>:<index>` is an item of the set (it could be used both on the left and right side). Otherwise, `undef` is returned (or, if used on the left side, it is an error).

An example of typical usage:

```
shared Set_Of_Integers seti;
private integer i;
i = 0;
// fill in 'seti' somehow...
while defined seti:i do {
// do something with seti:i
i = i + 1;
};
```

Note: It has already been said that the set modification during the sequential access can lead to confusing information for the user. The operation `insert` is relatively safe since the items are added to the end of the set. The operation `remove` is, however, dangerous, since it removes the item and (to keep continuity of the set) the last item is moved to its place. Thus, some items could be omitted during the access. In any case, it is safe to keep the set untouched during the access cycle.

3.2.8.7 Set constant

Set constant can be used in the program as a value of a set type:

```
[ <item1>, <item2>,...,<itemn> ]
```

where `<itemi>` are the items (possibly expressions, all of them must return the same data type). The assignment (no matter whether from a set constant or from another existing set) must keep the data type, e.g.:

```
type Tstr structure {
integer i;
string s;
};

type Tset set Tstr;
type Tiset set integer;

shared Tset sample_set;

sample_set = [ {1,"one"}, {2,"two"} ]; // ok

shared Tset another_set;

another_set = sample_set;

shared Tiset sample_set_2;
sample_set_2 = [ 1, 2 ]; // correct

sample_set = sample_set_2; // error
```

Of course, the set constant can be used only on right side, hence it cannot be, for instance, assigned to the set constant.

3.3 Variables

3.3.1 Variable declaration

The life of a variable in LanGR starts by the declaration — every variable must be declared before it is used. LanGR recognizes two declaration types: *regular* and *additional*.

Variables declared regularly and additionally have completely the same features. However, it is recommended to use a regular declaration, since it safely mediates the data type of a variable to the compiler (without parsing the variable's definition) — this leads to the simplification of expression parsing and decreases the possibility of errors at run-time.

Every variable keeps its data type through the whole life of the variable. Any attempt at assigning a value of different data type to the variable is an error. No variable can be "undeclared" (it is performed automatically at the end of the nesting level). The data type of the variable can be tested via the construction `istype`, see Section 3.3.4 [Testing the data type], page 23.

To declare a new *regular* variable, use the commands `private` and/or `shared`:

```
private <type1> <ident1>;
shared <type2> <ident2>;
```

Here `<type1>` and `<type2>` must be existing data types.

The declaration given above introduces the variable `<ident1>` of type `<type1>` and the variable `<ident2>` of type `<type2>`. The validity scope of the variables is generally based on the validity scope of identifiers, see Section 3.1.7.1 [Validity scope], page 6.

The declaration by `private` and `shared` differ only at the `split` command — `private` variables are copied to every thread, `shared` variable is the only one for all the threads, see Section 3.5.3 [Threading], page 34.

A regularly declared variable is assigned `undef` value after the declaration.

Additional declaration (contrary to the regular one) doesn't use any special command — it just starts with every assignment (`=`) whenever the identifier on the left side is unknown. In this case, the identifier on the left side is declared as a new variable of the same data type as the right side of `=`. E.g.:

```
i = 10;
```

declares a `integer` additional variable `i` with the value `10`. Similarly

```
private string j;
j = "blabla";
i = j;
i = 10; // error
```

declares a `string` regular variable `j` and then `string` additional variable `i`. The last assignment is obviously an error, since `i` keeps its data type from the previous code. Of course, there can be a more complex expression on the right side of the assignment.

The additional declaration starts *after* the evaluation of the right side — it implies that the identifier of the newly declared variable (on the left side) is treated as unknown on the right side (and hence said to be a free variable):

```
i = i == 10;
```

Here `i` is a free variable on the right side, while `i` on the left side will be declared as `predex` variable with the value set to the dynamic expression `i == 10`.

If the code given above is modified to

```
i = evaluate(i) == 10;
```

it will be a compile error — an attempt at an evaluation of an undeclared variable.

Caution: Additional variables are always `private`.

The data type of an additional variable need not be necessarily identified at compile time (however, the compiler uses a relatively complicated algorithm for the identification of the data type), since the data type of the additional variable is set always at run-time (it depends on the *value* of the right side).

Unless there is a typed value set to the additional variable (NOT `undef` and `N/A`), the variable remains untyped. Once a typed value is assigned, the variable obtains the type and keeps it[3].

Note: Usually, the situation in which the data type cannot be determined at the first assignment (at the declaration time) is very sophisticated, since the system does not know (for instance) which type of assignment should be used (every data type obtains its assignment operator — if it is unknown even at run-time, it is an error[4].

Hence, it is better to use additional variables if the data type from the right side is immediately known.

---

[3] A variable that is assigned a dynamic expression takes `predex` data type.

[4] Unless `firstoverload` option is used, see Section 4.2 [Run-time], page 49.

Technical note: The data type is more precisely identified at compile-time than at run-time (where the data type is simply determined on the basis of the right side value). This is wrong, since the algorithms should be identical. The question is whether this bug is really critical...

Note: Global variables are defined in the order in which they have been read during the processing of the source code (it also holds for the declaration order, but the declarations run before any execution). Therefore, one must be careful when using global variables in definitions of other global variables. However, the typical use of global variables avoids it.

3.3.2 Variable definition

Once a variable is declared, it can be *defined* — i.e. its value can be set, see Section 3.2.1 [Value types], page 8. If a typed value is assigned to the variable, its type must match the variable's type.

The value is set using the *assignment* operator (`=`). The former value of the left side is forgotten.

On the right side of the assignment there can be any expression (that returns the data type matching the data type of the left side), the left side must be an identifier of the variable (or an expression pointing to an element of a set or to an item of a structure).

Technical note: The operation `=` formally recursively *copies* the right side to the left side. In the implementation, we try to prefer a lazy strategy — we copy the values later at the time when something really changes inside the value itself.

Caution: LanGR supports NO pointers or anything alike. If you want to change a value of something, the only way is to put the whole path leading to it on the left side of the assignment (or to other operations that accept left-side values, e.g. `insert` on sets), e.g.:

```
t = text#10.form;
t = "blabla";
```

The second assignment doesn't change anything in the variable `text`, but it changes just the value of the variable `t`. It works in the same way also while passing the arguments to the functions, see Section 3.4.7 [Using functions], page 28.

Note: In addition to the assignment operator, there are several functions that accept left-side values (i.e. they change the values of variables). These are the functions for adding/removing items from sets (`insert`, `remove`) and the functions working with identifiers of free variables (quantifiers and `where` function).

Every left-side value must be (up to the last component) defined and must not be dynamic — this applies both to structures and sets, e.g.:

```
type mstr structure {
shortcut i;
string s;
};
type mset set mstr;

private mset mytest;

mytest#10 = { 10 }; // wrong, mytest is undef
```

```
mytest = [];
mytest#10 = { 10 }; // wrong, operator '#' can't
// add new items to the set

insert {10,"huj"} into mytest;
mytest#10 = { 10, "teststr" }; // ok

remove 10 from mytest;
mytest#10.s = "hoj"; // wrong, mytest#10 is undef
```

3.3.3 Constants

In some programming languages it can be specified whether a variable will change its value during the program or whether it will remain unchanged (*constant*). In LanGR, there is nothing of the sort. However, there are two possibilities of obtaining this effect:

- either a preprocessor can be used (Section 4.1.3 [Preprocessor], page 47), which is mainly useful for small objects,
- or a `shared` (global) variable is declared — this avoids any copying of the variable's value between different threads of the program.

3.3.4 Testing the data type

It can be sometimes useful to test the data type of the given value — almost always in case the function parses more arguments than the arguments listed in the header.

```
<val> istype <type>
```

returns `true` if the value `<val>` has the data type `<type>`, otherwise returns `false`. If `<val>` is undefined, `N/A` or dynamic variable, it is a run-time error.

3.4 Functions

For simplification of the code, a user has at his disposal a facility to define *functions* — the blocks of code that are invoked repeatedly in the program. Several functions are predefined, see Section 3.4.8 [Predefined functions], page 30, other functions can be defined using this section.

Generally, a function takes some arguments on input (Section 3.4.4 [Arguments], page 26) and returns a value (Section 3.4.6 [Return value], page 27). The body of the function is a piece of normal code in LanGR with some restrictions (Section 3.4.3 [Defining functions], page 25). Once the function is defined, it can be used later in the program (Section 3.4.7 [Using functions], page 28).

Each identifier of a function can be *overloaded*, i.e. used for different data types of the arguments, see Section 3.4.5 [Overloading functions], page 27.

3.4.1 Function identifiers

A function can (unlike other objects in LanGR) use more identifiers which support a better separation of the arguments passed to the function and hence a better understanding of the program. In this respect, we talk about *complex identifiers*.

Such identifiers must contain one *main* word, the remaining words are called *additional* words.

The main word unambiguously identifies the function in the program (like the variable identifiers and so on) and therefore it cannot be used in a different meaning. Additional words just surround the main word and separate the arguments. Their meaning need not be unambiguous in the program — every additional word can be used as an additional word of another function, but it cannot collide with other kind of identifiers (variables, main words, data types, ...) and even with free and additional variable names.

In this manual the words of a complex identifier (if used as a whole) are separated by a space or by a dot. The main word is usually highlighted with italics.

An example of a valid set of identifiers follows:

while`.do`
with`.do`
`==`
`+=`
array`.of.first.last`

3.4.2 Declaring functions

Every main word of a complex identifier must first be declared by a command `declare`:

```
declare <head>;
```

While calling a function by a space-construction, function's *priority* and *associativity* are taken into consideration. The main word of the function associates its additional words and arguments using these attributes.

The priority of the main word `<head>` can be set as follows:

```
priority <head> <integer>;
```

Here `<integer>` is the target priority level. The `priority` command can be used once at most for each main word — the main word must be already declared and there could be no overload defined for this main word yet. The allowed priority levels range from 0 (the lowest) to 100 (the highest).

Whenever two main words appear in an expression such that both of them share the same priority, the associativity is taken into consideration. The associativity can be set to "left-to-right" (by default, the arguments of the leftmost main words are set first) or to "right-to-left". The associativity can be changed as follows:

```
reverse <integer>;
```

The command `reverse` changes the associativity for the priority level `<integer>` to right-to-left and for every priority the associativity can be changed once at most (and only in case no main word on this level has been overloaded yet). Neither the priority, nor the associativity can be changed for the predefined functions. The actual values of priority and associativity are stored in a special file (processed at compile-time) which is also part of LanGR specification. The priority can be modified by parenthesizing the expression, see Section 3.4.7 [Using functions], page 28.

The commands `declare`, `priority` and `reverse` must be used only at the global level. Additional words are specified at the overload definition, see Section 3.4.3 [Defining functions], page 25.

Note: The existing implementation of the compiler is explicitly controlled by the priority and associativity, hence a main word with the priority `k` can never (unless parenthesized) be used as an argument for a main word with a priority higher than `k` (similarly with the associativity in case the priorities are equal). This holds (unexpectedly) even in case the expression has the only one "valid" reading, e.g.:

```
declare a;
priority a 50;
function a integer x {
// something
};

declare b;
priority b 20;
integer function b integer x {
// something
};

a b 10; // here only a(b(10)) is possible,
 // but forbidden by the priority setting
```

3.4.3 Defining functions

Once a main word is declared, the function can be *defined*, i.e. the arguments' data types, additional words and the function body can be written. To define the arguments and additional words, use a space construction, e.g.:

```
optional <ret_type> function <t1> <a1> <id1> <head>
<t2> <a2> <id2> <t3> <a3> {
// function body
};
```

The arguments are always tuples `<ti> <ai>` where `<ti>` is the argument's data type and `<ai>` is argument's identifier that can be used in the function body. Any number of additional words can be interspersed in between the arguments — additional words need no special declaration, they are just used.

If `<ret_type>` is specified, it must be a data type identifier — this type must be returned by the function (by `return` command, see Section 3.4.6 [Return value], page 27). If `optional` stands in front of `<ret_type>`, the function can be used both as a command and as an expression (Section 3.4.7 [Using functions], page 28), i.e. the function body must return the value of the type `<ret_type>`, but this value can be ignored by the calling function. If `optional` is not specified and `<ret_type>` is present, the returned value must be always used by the caller.

The special word `generic` can stand instead of `<ret_type>`. This indicates that different data types can be returned from the function (although something must always be returned).

Note: The functions returning `generic` are dangerous since they could lead to complex errors at run-time.

The function body is an independent program — all global identifiers, argument identifiers and (of course) locally declared identifiers (variables) are valid and accessible there. In the function body, no local identifiers of the caller are accessible, though.

```
declare secti;

integer function SECTI integer a S integer b {
private integer c; // local variable, valid only in this function
c = a + b;
return c;
};

declare pouzij;
function pouzij {
private integer x;
x = 10;
print secti x s 15;
// 'x' is not accessible inside SECTI
};
```

Note: The above declaration is not very practical, since one-letter identifier '`s`' blocks this name for any other use (e.g. as an additional word or a local variable).

A thread which enters the function when the function is used leaves the function by the command `return` or by executing the last command of the body. If the thread is split in the body, each of the generated threads leaves the function (unless it is finished with `fail`) and returns to the caller, see Section 3.5.3 [Threading], page 34.

Note: A function body must always contain at least one command.

3.4.4 Arguments

The arguments of a function, their names and data types are specified in the function header, see Section 3.4.3 [Defining functions], page 25. When a function is called, it must always be given at least as many arguments as specified in the header. These arguments are then available in the function body like local variables via their names specified in the header. A function can also take no argument.

If a function is called via a parenthesis construction or the function is on the top of the syntactic tree of the expression (i.e. it has the lowest priority of all the elements in the expression), it is possible to pass additional arguments to the function. These arguments are available through the `arguments` command, see below, but the data types of these arguments are unknown in advance.

All arguments the function was given on call are available by using the command `arguments(<index>)`, where `<index>` is the target argument number (`1`, `2`, ...). The call of `arguments(x)`, where `x` is greater that the number of arguments really passed to the function, results in a run-time error.

Note: The number of arguments is accessible by calling `arguments(0)` — it returns a non-negative integer.

Technical note: Formally, the arguments themselves form an independent nesting level (level 1). Thus, the nesting level in the function body is 2 and more.

The arguments of every function are always passed as *copies* of the real objects used in the function invocation. Thus, if the arguments are modified in the function, the original objects in the calling function remain unchanged, see Section 3.4.7 [Using functions], page 28.

3.4.5 Overloading functions

Once a function is declared, it can be defined, see above. Once a function is defined, it can be even overloaded, i.e. the same main word with different couple of arguments and additional words can be used (and, of course, for another function body).

Note: Almost all main words can be overloaded except for very special ones whose overloading could cause confusion and less perspicuity in the program, especially `return`, `fail`, `die`, `arguments`, `defined`, `constant`, `applicable`, `output`, `input`, `print`, `rulename`, `variantname` and quantifiers.

For the specification of a new overload it is necessary for data types of the arguments and for additional words in the new overload to differ at least in one place from all old existing overloads. Otherwise, it is not possible to distinguish the new overload from the old one and an error is produced.

Caution: The overload of two functions does *not* differ especially in the following cases:

- if they differ only in the return value — it is not always possible to determine the expected return value from the function call (e.g. in the additional variable declaration);
- if they differ only in the distribution of the main/additional words in the arguments — this has the consequence that the overloads are the same as in the parenthesis call, e.g.:

```
declare f;
integer function f integer a string s {
return a -- 1;
};

integer function integer a f string s { // error --- 2nd def.
return a + 1;
};
```

If, for instance, an order of arguments changes, the abovementioned overloads will be different.

3.4.6 Return value

A function can, but need not return a value. If the value is returned, its data type is previously known from the function definition and the function must always use the command `return` (in all threads) to return the value. The argument of `return` (if specified) must be always a value of the correct data type (with respect to the function header).

If the return value is not used in the header, the function returns nothing and `return` need not be used in the body (however, it can be used without any argument to finish the function body before reaching the end of function).

The return value can be modified with `optional` — the function body must then return the value as if `optional` were not specified, but the function can be used as a *command* in the program (the returned value is not used by the caller and it is forgotten).

The function cannot return more than one value — but it can always return a structure or a set.

3.4.7 Using functions

In the program, the functions can be used as commands (if they return no value) or in an expression (if a value is returned). The value returned (if any) is used at the place where a function call has been made. If the data type of the returned value is incompatible with the excepted data type at the given place, it is an error and the compiler (or interpreter) detects it (e.g. if a function returns a value without `optional` and it is used as a command).

If a function is to be called, two constructions can be used: *space* and *parenthesis*, see the subsections.

The values passed to the function are always copies of the original values in the calling function, Section 3.4.4 [Arguments], page 26. The invoked function never changes variables in the calling function, with the exception of a few special functions that accept left-side values (e.g. assignment, inserting/deleting to/from sets). The only effect of the called function in the calling function is the returned value and optionally also the change of the global variables.

3.4.7.1 Parenthesis construction

The *parenthesis* construction is preferred because it guarantees the non-ambiguity of the coding, but it is less readable. Moreover, using the parenthesis construction is always a safe way of passing additional arguments to functions.

General call of the function `<head>` with additional words `<add1>` to `<addn>` is

```
<head>.<add1>.<add2>...<addn>(<arg1>,<arg2>,...,<argn>)
```

The additional words (if specified) must follow the main word in the left-to-right order in which they are specified in the function definition. Also, the order of arguments must correspond to their specification in the function header — behind the last "regular" argument additional arguments can follow, they are accessible by the `arguments` command in the function body.

The additional words can be completely or partially omitted at the end of the list (i.e. `<add1>` must be present whenever `<add2>` is present). In any case, the main word, additional words and the arguments must unambiguously determine the particular function overload. The ambiguities are detected by the compiler or in run-time and an error is reported.

Note: Every construction where `.` or `(` follows immediately (without a space) the main word is treated as a parenthesis construction by the compiler. This is important while using individual characters that are function identifiers and that need not be separated by white spaces from the surrounding environment — once `(` is used immediately behind such character, it is a parenthesis call:

```
// let A be a structure with a field B
A.B // is ok, no parenthesis
(A).B // ok, parenthesis before a dot is safe

A.(B) // this is a parenthesis call!!!
// this is probably not what the user intended,
```

```
// although syntactically it can be O.K. in
// dependence on the context

.(A,B) // correct, the same as A.B
A. (B) // correct, same as A.B
A . B // same as A.B
```

The parenthesis construction can also be used in the input files, see Section 3.7 [Input/Output], page 42. Every other thing is space construction, see below.

3.4.7.2 Space construction

A space construction is usually more readable for non-programmers, so it is expected that the space construction will be preferred. The function identifiers (main/additional words) and the arguments are separated by white characters[5]. The positions of the arguments and additional words around the main word must correspond with the function header.

The processing of an expression in the space construction is driven by priority and associativity. The main words with higher priority (or preferred by the associativity in the case of equal priorities) bind their arguments and additional words former than the main words with lower priority. *The arguments and additional words that have been already bound by a main word can no longer be used by any other main word in the expression.*

This allows for a very quick analysis of the expression, but on the other hand, it sometimes leads to the refusal of a "correct" expression (i.e. in case the expression has an unambiguous reading, that, however, doesn't correspond to the priority demands, see Section 3.4.2 [Declaring functions], page 24).

Discussion: It is quite clear that this feature could be difficult to understand. However, the parsing algorithm can be rewritten in future and, moreover, it seems that most of the problems can be solved by the careful assignment of the priority levels and associativity.

In any case, if the expression is refused by the compiler/interpreter, the parenthesis construction can always be used, or the priority/associativity can be changed by enclosing an expression part in parentheses, e.g.:

```
declare g;
declare f;
priority g 20;
priority f 10;
// some definitions of 'f' and 'g'
a = g (f 10); // correct
```

Sometimes the other extreme can appear — an expression has more interpretations, e.g. if one or several arguments are of an unknown data type (values `undef`, `N/A`, free variables, a function returning `generic`, ...).

Every ambiguity at run-time is always an error. However, at compile-time it is not so clear — we distinguish two types of ambiguity:

- an ambiguity in the topology of the expression tree — this can happen only in case a function is overloaded for two argument sets that differ in the number of arguments.

[5] With some exceptions, see Section 3.1.1 [White characters], page 3.

This situation should normally never happen, but when it happens and the compiler cannot decide which overload to use, it is a compile error.

- an ambiguity in the particular overload — e.g. let's have a binary comparison `eq` used in the context of another function whose return value is unknown to us, e.g.:

```
// let A, B, be structure variables of the same type StruType,
// let addit1, addit2 be variables of the "fields StruType" type
// assume 'eq' is overloaded for strings and integers
A.addit1 eq B.addit2
```

It is clear that `eq` takes two arguments `A.addit1` and `B.addit2`, and therefore the expression topology is the same for both overloads. This is not an error at compile-time and the particular overload can be selected at run-time in dependence on the arguments really specified.

It is possible, of course, that the data types of the values returned from `A.addit1` and `B.addit2` will differ at run-time and then it is an error.

Other ambiguities should not occur because the processing of the expression is based on priority levels.

Note: In the space construction the additional arguments can be specified behind the last "regular" argument expected by the function definition. Moreover, all additional arguments will be passed to the top of the expression tree.

3.4.8 Predefined functions

In this section, we present the functions built-in directly into LanGR — this is not the complete list of built-in functions, the rest is mentioned in the sections which better correspond to the functions' use (e.g. working with sets and so on).

3.4.8.1 Comparison operations

For every data type (with the exception of `predex` and all set data types) the comparison operators `eq` (equality) and `ne` (inequality) are predefined.

The equality for basic data types (`string`, `integer`, `domain`) should be clear. The instances of a structure type are compared category by category in the key categories — the equality of all key categories implies the equality of the structures. If there is no key defined in the structure, the comparison `eq` returns always `false`!

`undef` and `N/A` are (for `eq`) different values and they are both different from any defined value.

The inequality `ne` is the negation of the equality `eq`.

Note: The data types without `eq` defined cannot be used either as items in the sets, or as key categories in structures.

For all data types (except for `predex` and all set types) the operator `*===*` (extended equality, *underspecification*) is implicitly defined. Here, `undef` and `N/A` are equal to anything. Constant values of the basic data types are compared according to `eq`, the difference consists mainly in the structures' comparison then.

If the structure contains no key category, `*===*` returns `true`. If key categories are present in the compared structures (`shortcut` is *not* a key!), the structures are equal if and only if all the key categories are equal (in `*===*`, of course).

For `string` and `integer` there exist also the operations `le` (less or equal), `lt` (less than), `ge` (greater or equal), `gt` (greater than). Passing `undef` or `N/A` values to these operators is an error — they operate only on "real" values. For `integer` the meaning is quite clear, for `string` the lexicographic ordering of strings is *not* yet implemented, the strings are compared character by character using the Unicode index of characters. If a user is doubtful about how a comparison works, he should try it.

The above functions have a preprocessor equivalents:

`eq`	`==`
`ne`	`<>`, `!=`
`gt`	`>`
`ge`	`>=`, `=>`
`le`	`<=`, `=<`
`lt`	`<`

3.4.8.2 Arithmetic operations

For `integer` data type there are binary arithmetic operators `+`, `-`, `*` (multiplication), `/` (division) predefined in LanGR. All the operators take two `integer`s and return also `integer`.

To cut the decimal part of a number you can use the function `cut`, for the rounding of real numbers use `round`. The result of `round` applied to negative numbers is `-round(|<number>|)`, where `|<number>|` is the absolute value of `<number>`.

```
private integer i;
i = 10;
j = 6;
k = round (i / j); // k == 2
k = cut (i / j); // k == 1
```

3.4.8.3 String operations

The string concatenation is represented by the overload of the operator `+` — it takes two `string`s and returns their concatenation. The operator `length <string>` returns the length of the string `<string>`, which is always a non-negative integer. For undefined or `N/A` argument the function returns the `undef`.

The important function is

```
substring <str> <beg> <len>
```

This function takes three arguments — one string `<str>` and two integers `<beg>` and `<len>`. The result is again a string — a substring of `<str>` depending on `<beg>` (the starting index of the result in `<str>`) and `<len>` (the length of the substring). Both `<len>` and `<beg>` must be integer values. The positions in `<str>` are indexed from 1 in the left-to-right order:

- If `<beg>` is greater than the last index in `<str>`, the error occurs;
- If `<beg>` is lower than 1, the first substring character is treated from the end of `<str>`, see examples below. If `<beg>` is lower than `(1 - length <str>)`, it is an error;
- If the index `<len>` counted from `<beg>` is outside `<str>`, the result is a suffix of `<str>` starting from `<beg>`;
- The length `<len>` can be negative — then the resulting substring is to the "left" of `<beg>` instead of to the "right". The position `<beg>` is, however, always included in the result, unless `<len>` is zero.

The examples:

```
private string s;
s = "Dobry den";

s1 = substring s 2 4; // = "obry";
s1 = substring s 0 -3; // = "den";
s1 = substring s 6 1; // = " ";
s1 = substring s 6 0; // = "";
s1 = substring s 10 -6; // error: (10) outside 's'
s1 = substring s -4 6; // = "y den";
s1 = substring s -9 6; // error: (-9) outside 's'
s1 = substring s -6 -6; // = "Dob";
```

The following functions take a string argument and return always a string (mostly a different one):

`lower` converts the argument to the lower case;

`upper` converts the input to the upper case;

`capital` converts the first letter to the upper case and the rest to the lower case;

`ascii` removes the diacritics (not yet implemented).

See also Section 3.7 [Input/Output], page 42. If the conversion is impossible, it is generally a run-time error.

There are comparison operators `eq`, `ne`, `lt`, `le`, `gt`, `ge` defined for strings, see Section 3.4.8.1 [Comparison operations], page 30 — all these operators compare in the case-sensitive fashion. Case non-sensitive comparison can be performed by a couple `ieq`, `ine`, `ilt`, `ile`, `igt`, `ige` — formally these functions use `lower` before the comparison.

The string capabilities are extended with regular expressions, see Section 3.1.4 [Regular expressions], page 5.

The example using string operators:

```
a = "k";
b = "retize";

b = b + a;

c = "retizek";

if c eq b
```

```
then {
print "strings are the same"; // this will be printed
} else {
print "string are different";
};
```

A function `stringprint` makes it possible to convert any value to `string`, see Section 3.7.1 [I/O commands], page 43.

3.5 Expressions and commands

The program in LanGR consists of *commands* that are grouped into `command sequences`. A typical command sequence is a function body, the commands on the global level (outside the functions) or a rule/group body.

Every command is always ended with the semicolon (`;`). We distinguish three basic types of commands: *controlling the command flow*, *declarative* and *executive*. The commands driving the command flow are directly built into the language, their identifiers are reserved and they have always an unambiguous meaning.

The group of declarative commands contains the commands for a variable declaration and the data type definitions, see Section 3.3.1 [Variable declaration], page 20 and Section 3.2 [Data types], page 7.

The executive commands are the function calls. Usually, there are some arguments passed to the function — these arguments can be either constants or more complex expressions — again the function calls.

Executive commands do not return any value — this is their only difference from expressions. Hence in the sequel, executive commands will be considered expressions returning no value.

In this section, we will first investigate *the command flow controlling commands* and then *the expressions*.

The commands controlling the command flow directly change the flow of processed commands. Most of the commands belonging to this group take a command sequence (or more command sequences) as an argument — these sequences are always enclosed into curly brackets. Every command controlling the command flow is formally a command and therefore it must be ended with a semicolon. A command sequence must never be empty — at least one command must be present (with the only exception of the `split` command, see Section 3.5.3 [Threading], page 34).

3.5.1 Conditional commands

LanGR recognizes the following conditional constructions:

```
if <predex> then { <command_sequence1> };

if <predex> then { <command_sequence1> }
else { <command_sequence_else> };
```

Here `<predex>` is an arbitrary predicate expression. If `<predex>` is evaluated as `true`, the program continues by executing the `<command_sequence1>`, otherwise it continues with

`<command_sequence_else>` — if `else` branch is not specified, the control continues with the command that immediately follows the `if` command. If `<predex>` cannot be evaluated, it is an error, see Section 3.5.5 [Expressions], page 36.

3.5.2 Cycles

In LanGR, two cycle commands are available:

```
while <predex> do { <command_sequence> };

do { <command_sequence> } while <predex>;
```

The `<command_sequence>` is executed while `<predex>` is `true`. In the second case, the sequence `<command_sequence>` is executed at least once.

Caution: if `<predex>` contains a variable, it must be declared at the time `<predex>` is evaluated.

The most nested cycle can be interrupted by the command `break`. The program then continues by executing the first command after the interrupted cycle. The command `continue` in `<command_sequence>` skips the rest of `<command_sequence>` and continues by evaluating `<predex>`.

```
i = 10;

while i > 0 do {
i = i - 1;
if i == 8 then { continue; };
if i == 5 then { break; };
print i;
};
// "976" will be printed
// i is now equal to 5
```

3.5.3 Threading

Threading means that at some points the program execution splits into several threads and all threads will be executed separately. We do not simulate real parallelism, it is sufficient that all threads will be sequentially executed in the given order:

```
split {
{ <command_sequence1> }
{ <command_sequence2> }
{ <command_sequence3> }
.
.
.
{ <command_sequencen> }
};
```

The command `split` splits the current thread (which proceeded up to the command `split`) into `n` new threads — every thread then starts with the corresponding `<command_sequencei>` and continues after the `split` command, unless it reaches the end of program or a command `fail`, see Section 3.5.4 [Finishing the program], page 35.

Unlike all other control flow commands, `split` can contain a block with empty `<command_sequencei>` — the thread then continues after `split`.

The threads are executed sequentially in the order given by the list of command sequences in the `split` command.

Very special attention should be paid to using the variables in the threads. It is a well-known problem with the variables' sharing and collisions while changing them. In LanGR, every variable is either `private` or `shared`, see Section 3.3.1 [Variable declaration], page 20:

- `shared` variable is shared by all threads — the threads executed later can access this variable already modified by the former threads. The advantage of `shared` variables consists also in that they are almost never really copied and hence they do not decrease the program speed.
- `private` variable is private for each thread. In `split` command, for each thread its own copy of the set of private variables is created, hence the previously executed threads don't affect the variables for later threads. All additional variables are `private` by default, see Section 3.3.1 [Variable declaration], page 20.

Although there is no limit of the use of `split`, it is expected that it will be used in the rule-context (i.e. in some functions called from a rule variant). If it is used in this way, the program that has been split can be never reconnected (since all threads generated in a rule variant are by default finished by `fail`).

In the situation above a problem of simultaneous modification of the `private` variables from different threads can never appear. This problem can appear theoretically only in global `private` variables that survive the end of rule variants, but there should be the minimum number of global `private` variables and each thread should initialize it (e.g. the variable `TextIndex`).

For these reasons, the problem of collisions in modifying a private variable is neither expected nor solved.

An example of `split` usage:

```
private integer i;
i = 5;
split {
{ i = 1; }
{}
{ i = i + 3; }
};

if i == 5 then { fail; };
print i;
// prints "18"
// the second thread will finish with fail
```

3.5.4 Finishing the program

The execution of a program finishes automatically by executing its last command of the global level, or after executing all the commands required by the switches from the command line, see Section 4.2 [Run-time], page 49.

The whole program can be immediately stopped by the command `die` (no parameters). Individual threads can be finished by the command `fail`.

Every function (for the current thread) can be finished in any point by `return`, see Section 3.4.6 [Return value], page 27.

3.5.5 Expressions

Every expression consists of the identifier of an *action* that is to be performed and of the *arguments* that participate in this action.

The passing of arguments has been exhaustively documented in the Section 3.4 [Functions], page 23. This section will concern itself with the detailed description of how to use *dynamic* and *static* expressions.

One of the most important differences between LanGR and any other programming language (as far as I know) is a possibility of passing the whole expression as an argument to a function — it may be that such an expression is not evaluated at the time of expression passing, it can contain free variables and so on and the receiving function can either define these variables and evaluate the expression or define just a part of the variables and pass the expression as a return value or whatever.

First of all, we will show the power of this approach on an example. Let's suppose that a linguist would like to find a position in the text containing (possibly) the lemma *do*:

```
ITEM Possible lemma == "do";
```

Simply said — the expression `lemma == "do"` is a condition that could be applied to one morphological interpretation (a *tag*). The function `Possible` extends the condition to the whole position and specifies that for the given word form there must exist at least one tag in the repertory of its tags such that it meets the condition `lemma == "do"`. But we still do not know which position should be tested in the text (the variable `lemma` remains a free variable).

This is specified by the function `ITEM` — it passes an "actual" position to `Possible` and verifies the condition.

Note: Of course, `Possible` and similar modifiers can be defined as additional words for `ITEM` and the whole idea of dynamic expressions could be destroyed. However, a lot of confusing overloads of `ITEM` will appear. Moreover, `Possible` and similar modifiers will be probably used also in other functions in the same manner.

It is worth seeing that neither `ITEM`, nor `Possible` can directly set the value of `lemma` for the evaluation of `lemma == "do"`, since the arguments must always be specified (in a non-dynamic approach) before the function call, even in interpreted programming languages.

Dynamic expressions are, in the end, absolutely necessary if we insist on storing them in output files (conditional tag deletion, creating dynamic bindings between tags).

From the above it follows that the expression `lemma == "do"` must be compiled into a construction that is passed to `Possible` as an expression — just to avoid direct execution of `lemma == "do"`.

The call of `Possible` could be in some cases compiled statically, it depends on a particular implementation — generally `Possible` would just somehow extend its argument with some quantifier and return this extended expression — hence there would be no problem with a static call.

We distinguish (at compile time) two kinds of expressions according to when the function on top of the expression will be executed:

- static — the function will be executed before being passed to the caller (i.e. the *result* of the function will be passed to the caller);
- dynamic — the function with its arguments will form a dynamic value (a tree with the reference to the function in the top node and the arguments in daughter nodes) and this dynamic expression will be passed to the caller.

Please note that the argument of a dynamically compiled function can be also a dynamic expression.

Now let's look at the problem in general. We have an expression tree that has been built by means of the priority and associativity or by means of a parenthesis construction. There are constant values in the terminals and functions to be executed residing in non-terminals.

For every node we have to decide whether to compile it dynamically or statically. In the following paragraphs we will investigate the compiler's decision strategy for solving this problem.

Note: Due to the expected usage of dynamic expressions — forming conditions imposed on positions, tags and so on — the dynamic expressions live at run-time in the values of the data type `predex`. It implies that there must be `predex`-returning function on top of every dynamic expression (however, in the subtrees there could appear also non-predex functions).

For readability of the following lines, we will introduce (similarly as the compiler does) the following modifiers for every node in the syntactic analysis of an expression. Once the compiler finishes the parsing of the whole expression, each node is assigned exactly one modifier (which indicates the method of compilation of this node):

`EVALUATE` compile statically (prepare a function call, process the arguments, call the function);

`VALUE` in non-terminals, compile dynamically (prepare dynamical non-terminal, process the arguments and hang them to the non-terminal), in terminals, compile statically (see below);

`LINK` only in terminals — compile dynamically. The actual value of the terminal is (at the execution time) taken from the global variables and `where-is` and `use` information (i.e. *not* from local variables);

`NAME` only in terminals, dynamic compilation for left-side arguments.

A user can override the implicit algorithm by explicit use of `evaluate`, `value` or `link` — the node immediately below this modifier will be compiled in this way (with exception, see Section 3.5.5.2 [Processing functions], page 38.

3.5.5.1 Processing terminals

Constant values directly receive `EVALUATE` (for decision algorithm all structure and set constants are treated as non-terminals, see below).

The variables in the leftmost branch of the left-side value and the names of quantified variables take `NAME` (a user cannot override it).

Free variables cannot be evaluated at the place where the expression is created, hence they are `LINK`. A category identifier (used as a reference to a value of the category) is also a free variable.

Global variables could always be executed, but it is expected that the user mostly wants to use their actual value in the evaluation time — hence they are `LINK`. Global variables will probably be the most frequent candidates of explicit use of the `evaluate` operator.

Unlike global variables, local variables are usually evaluated only in the moment of the creation of an expression, hence we compile them statically — they are either `EVALUATE` (if they are not of the predex type) or `VALUE` (those with the `predex` data type).

As for the compilation, there is no difference between `EVALUATE` and `VALUE` for terminals (just as a mark of "dynamic" and "static" node for the non-terminal processing, see below). Both take (at run-time) the actual value of the variable and return it as an argument to the calling function. The value of `predex` variable is often a dynamic expression, hence it is passed to the caller as the argument without any change.

Note: Terminals modified with `LINK` could be (in some conditions) re-marked to `EVALUATE`, see below.

3.5.5.2 Processing functions

The modifiers in the non-terminals (nodes of functions) are dependent also on the information outside the node itself, mainly:

- if a subtree contains a node that is compiled dynamically;
- if there is a node that returns `predex` on the path from the node to the root node;
- if the node is marked as `value` or `evaluate` by the user.

The node that does not return a `predex` value and no `predex` value is returned from any of its predecessors (on the path up to the root node) is compiled immediately statically — this is also the case of terminals where `LINK` is changed to `EVALUATE`.

If the node itself or some of its predecessors returns `predex` (this explicitly means `predex`, i.e. the function must be built to create dynamic expressions), then the decision depends on the existence of a dynamic node in the subtree — if there exists a dynamic node in the subtree, the node is compiled dynamically (otherwise the compilation is static).

The terminals marked with `VALUE` are considered dynamic nodes for this purpose, since we don't know whether an expression stored in the given variable is static or dynamic.

Terminals marked with `EVALUATE` are always static.

This algorithm (however simple) has also some exceptions, viz. the left-side values that are always compiled dynamically.

An example:

```
// global level
a = 3;
private predex p;
p = a == 5; // (1)

a = 5;
if p then {
```

```
print "EQUAL";
print;
private integer b;
p = b == 5; // (2)
b = 5;
};
```

In the line (1) is a global variable a compiled as LINK. The parent node == returns predex and contains a dynamic node (a) in the subtree, hence it will be compiled dynamically with VALUE. This means that the p will be assigned the dynamic expression a == 5 — it is evaluated to true/false according to the actual value of a.

Then a is assigned the value 5 and on the next line the evaluation of p is performed. Here the expression in if is executed (at run-time) until a constant is reached, hence p is replaced with a == 5, which is immediately evaluated with a being set to 5, and therefore EQUAL will be printed.

Then the local variable b is declared (it is valid only up to the closing }, i.e. to the end of then branch). In the assignment on line (2) the variable b is local, hence it compiles with EVALUATE (statically). There is no reason for the == to be compiled dynamically (since everything in its subtree is static) and thus == compiles statically as well. The variable p will be assigned the immediate result of comparison of b (undef) and 5, i.e. false.

Any later change of b cannot change the value of p.

A more complex example:

```
// global level
a = 3;
private predex q;
q = false;
while a > 0 do {
private predex p;
p = a == evaluate(a); // (1)
q = p or value(q); // (2)
a--;
};
// 'q' contains dynamic "(a == 3) or (a == 2 or (a == 1 or (false)))"
```

On line (1) the first occurrence of the global variable a is compiled as LINK, but the second occurrence of a is overridden by an explicit use of the evaluate operator. Hence == will be compiled with EVALUATE. Then p is assigned (in the cycle) by expressions a == 3, a == 2 and a == 1.

On line (2), the local variable p is compiled with VALUE, i.e. with its immediate value. The same situation appears with q (its implicit LINK has been explicitly overridden by value). The result of the right side in (2) is then dynamic (since under p and/or q a dynamic expression could be hidden — and it really is). Final result is given in the comment.

A user can (similarly as with the terminals) apply the operators value and evaluate also to non-terminals (including set or structure constants). However, value has no effect when applied to non-terminals, since a non-terminal can be implicitly assigned EVALUATE and if this happens it cannot be modified any more.

Hence only `evaluate` makes some sense in connection with non-terminals. The compiler is said that the node must be compiled statically in any case — this can be useful for functions that make a predicate expressions out of another predicate expression (and no more), e.g. `IsSafe`. These functions can always be called with `evaluate`.

Note: Neither at compile-time nor at run-time it is verified whether a cycle has appeared between `predex` variables, e.g.:

```
// global level
private predex p;
private predex q;
p = p or q; // links to both 'p' and 'q', 'p' cycled
```

The user must avoid these constructions, although there is an upper limit of the maximum number of nested expressions that are evaluated to achieve the `predex` constant.

3.6 Modules

As in many object-oriented programming languages, a program in LanGR consists of a lot of *modules* — every module is an individual file defining one of the identifiers in the program (a function, data type, global variable, ...).

Since the compiler looks for unknown identifiers in files identified by their names, see Section 3.1.7.2 [Meaning of identifiers], page 7, the user is recommended to create separate files for every identifier (like in C++ the header file contains mostly one class declaration).

Every module is a plain ASCII text, or a plain text in some other charset, see Section 4.1.1 [Character sets], page 46.

The general content of a module is described by surrounding sections. It remains to say how to define rules and their groups.

3.6.1 Rules

New rule is defined by the keyword `rule`:

```
rule <ruleid> {
rulevariant <var1> {
};
rulevariant <var2> {
};
// ...
};
```

The command defines a new rule `<ruleid>` (global identifier). Every rule is formally a function (with the main word `<ruleid>`) and it is possible to use it anywhere where any other function is applicable — although it is reasonable only in the groups of rules, see Section 3.6.2 [Groups], page 41. A rule takes no argument and returns no value.

The rule body (contrary to function bodies) contains exclusively rule-variants and comments. The rule variants are initiated by `rulevariant`. Once a rule is executed, particular variants are processed sequentially in the order in which they have been specified in the definition.

The identifiers `<vari>` are not global identifiers and they can collide with anything, even (unfortunately) with each other in one rule.

Inside any rule variant (and also functions recursively called from this variant) the rule and the variant identifiers are available:

```
variantname
rulename
variantidentifier
```

All of the abovementioned commands take no argument and return string. `variantidentifier` returns exactly `(rulename + "." + variantname)`.

Note: The system doesn't forbid nesting rules, although, it is probably something the user does not want. In such a case, the abovementioned functions refer to the deepest rule/variant in the stack.

A variant body already contains the executive part of the program, it directly implements the intentions of a linguist. The content of a rule variant is formally syntactically and semantically the same as the content of any other function. The compiler, however, slightly modifies the beginning and the end of each rule-variant:

```
split {
{
RuleStart;
// original variant body content
RuleFinish;
fail;
};
{ return; }; // always survives
};
```

The idea behind this construction is as follows: the original meaning of `split` is to solve non-determinism in the rule variants — although we expect that the particular threads that have originated in the variant will also finish within the variant, unless we really want a non-deterministic machine (we do not need real non-determinism, we just want to apply the variant in all configurations wherever it is applicable).

The final `fail` ensures it. However, it kills the entire computation, since after the end of variant no thread survives and the program will finish. Therefore it is necessary to create one "surviving" variant — and this is ensured by the starting `split`.

All variants require some specific actions (that are not, however, necessarily the part of the compiler, but more or less the part of the application level of LanGR), the compiler adds to the beginning and the end of each variant the functions `RuleStart` and `RuleFinish` — these are simple functions defined in the global identifiers. They take no argument and return no value.

Note: A rule need not to contain any variant.

3.6.2 Groups

To built some hierarchy in the set of rules, it is possible to create groups of rules:

```
group <groupid> {
// group body
};
```

The identifier `<groupid>` is a global identifier of the group. Similarly as a rule, a group is formally a function taking no argument and returning nothing.

Unlike a rule, the body of a group is really a function body, i.e. there is no restriction applied to its body. However, it is expected that a group contains calls of rules or calls other groups, e.g.:

```
group mygrp {
rule1;
rule2;
groupa;
groupb;
rule3;
};
```

Instead of functions, the group body can be empty:

```
group mygrp {
};
```

3.6.3 Rule/Group cycles

So far we have spoken about rules and groups and about their application to the potential input text. It is important that rules and groups are executed in the order defined by their bodies, i.e. in a sequential order.

However, while applying rules and groups it can happen that a rule makes it possible for other rules to be applied. These rules, however, have already been executed in the sequential order (and hence they will not be executed again).

LanGR solves this problem by a cyclic application of rules, or whole groups, respectively. This means that if there is any change in the text after all rules/groups have been executed in the sequential order, all the rules/groups are executed again (in the same order). This is performed in a cycle until some change is found in the text.

The recommended way of using the cyclic application of rules/groups is to keep the groups and rules sequential in their bodies, but to use the switch `-r` in run-time, see Section 4.2 [Run-time], page 49.

The second possible way is not to use `-r`, but rather to define (on the global level) the user's own sequence of rule/group execution and simulate the cycle "by hand". This, however, requires special skills in working with the application functions and with the text implementation.

Let's just remark that `-r` switch is equivalent to the following sequence on the global level:

```
IncreaseTextChangeNestingLevel;
while TextChange {
    TextChange = false;
    // rules and groups to be cycled
    UpdateTextChange;
};
DecreaseTextChangeNestingLevel;
```

3.7 Input/Output

At run-time, a program in LanGR communicates with the environment through three channels:

- input
- output
- errors (notes)

LanGR offers no other possibilities to open some files and so on.

Usually the compiled rules are executed in such a way that the channels are connected to the *standard* input/output/error channels of the operating system.

On the command line it is possible to specify the names of alternative files that will be used as the channels by means of the switches `-I <inp>`, `-O <out>`, `-E <err>`, where `<inp>`, `<out>` and `<err>` are the file names. It is not necessary to use all these switches together (unspecified channels remain standard).

By default, LanGR uses ASCII encoding of the files — this is, of course, highly restrictive. Through the switches `-i <charset>`, `-o <charset>`, `-e <charset>` the encoding of particular channels can be changed. Instead of `<charset>` it is allowed to use everything that is admissible in the command `charset`, see Section 4.1.1 [Character sets], page 46.

3.7.1 I/O commands

The command `input <lsval>` reads from the input channel. It returns `predex` — `true` if the read was successful, or `false` if the end of input was reached or when an error occurred.

The result of the reading is stored into the only argument `<lsval>` that must be a left-side value. Moreover, from the argument's type it must exactly follow which data type is expected on the input, and moreover, only a structure type, set type, `string` type or `integer` type are allowed to be expected. Hence `predex` and `domain` values cannot be directly read from the input.

Inside a structure constant or a set constant it is, however, possible to read all types that are expected on the basis of the definition of the structure/set data type.

The commands `output`, resp. `print` are used for writing to the output, resp. error channel. Both work almost in the same way and contrary to `input`, although, all data types can be written to the output/error channel.

Both functions take an arbitrary number of arguments and they write them to the channel in the sequential order without any separators. The only *difference* is that `print` writes string constants without enclosing double-quotes, while `output` writes also these double-quotes.

The commands `input`, `output` and `print` can be used both on local and global level without restrictions.

The function `stringprint` works completely in the same way as the `print` function with the only difference — its result is not written onto the error stream, but it is returned in a string.

3.7.2 Formatting I/O

Apart from our desire to simplify the work with external files as much as possible, we also want to have the input and the output of LanGR very general, independent of any particular format of the linguistic data.

Both input and output of LanGR are the sequences of constants, see Section 3.1 [Lexical symbols], page 3, separated by white-spaces. Generally, it is possible to store also non-constant expressions (e.g. `predex` instances) by enclosing them into a structure.

Hence it can be logical that the input/output will have the same format as the source code in LanGR — it is almost true with the following restrictions that are due to the fact that the input/output is not separated via semicolons and due to the demand of the fast parsing of the input:

- undefined value must always be specified as `undef`, unless it is present inside a structure constant (here it can be omitted in the same way as in the source code);
- all expression must be enclosed in parentheses construction (NOT in a space construction);
- neither in the input, nor in the output preprocessor directives can appear;
- no identifier that is unknown from the compile-time can appear in the input;
- the command `istype` can be used (on the input) in connection only with data types that have been seen at compile-time;
- all expressions in the input are treated as dynamic expressions including individual variables.

It can be seen that if the input contains only the objects that have been seen at compile-time, everything is OK. We will finish now with a more detailed description of output of particular data formats. This also applies to the input and error channels:

`undef` writes `undef`, except for the structure constant;

`N/A` writes `N/A`;

`integer` so far implemented as `double` in C++, cf. the C++ handbook. Shortly — it writes also in decimal base;

`string` a sequence of characters enclosed in double-quotes (`"`) (the function `print` omits these double-quotes);

`regex` not yet implemented

`domain` the same as during compilation. The only difference concerns categories — in the input the category is never seen as a free variable, it is always a constant of a domain type.

If we want to use a category in the input as a free variable, we have to use `field` operator there, e.g. `field(person)`. The output functions work in this way.

`predex` constant values are printed and expected as identifiers `true` or `false`. Dynamic expressions are written in parenthetical constructions with all additional words being specified;

`structure`
the same as any structure constant in the source code, just the use of category names is forbidden;

`set` no restrictions.

3.7.3 Converting corpora

So far, only conversion from/to the format of the Prague Dependency Treebank (PDT) is supported (with a lot of restrictions), written in `Perl` — `pdt2internal.pl`, `internal2pdt.pl`, `PDT.pm`.

4 Compilation and run-time

4.1 Compilation

4.1.1 Character sets

This is a common problem of encoding non plain-ASCII texts on computers. We will discuss the problem for Czech, however, for the other languages the situation is almost the same.

Several *charsets* are available to encode Czech on computers, i.e. MS Windows use their code-page `1250`, Unix systems mostly use `ISO-8859-2`, and so on. The encoding always assign particular characters their *indexes*. One-byte encodings (i.e. of the type given the above) use indexes (0 to 255) and they differ just in the assigning function. To unify these encodings a two-byte encoding was developed, known as *Unicode* (with several variants, e.g. UTF-8, UTF-16, ...).

Internally, we will use `UTF-8` encoding to encode the characters.

A more difficult problem emerges with the processing of the input and output. The *Unicode* encoding seems to be rarely used (current versions of MS Windows seem to use the Unicode internally but it is not clear whether it is possible to export the data from MS Windows into the plain Unicode), hence we must expect an arbitrary encoding on input. There is probably no safe method of how to decide automatically which encoding is on input — however, the authors mostly know it.

It is recommended to specify the encoding of every particular module by means of the `charset` command:

```
charset <chs>;
```

The `charset` command is used on the global level and it must stand alone on the first line of the file[1]. Until a `charset` command is used, it is expected that the file uses the encoding specified in the switch on the compiler's command line, see Section 4.1.4 [Invoking the compiler], page 48.

The argument `<chs>` is a string containing the name of the supported encoding, see Section 4.1.4 [Invoking the compiler], page 48.

4.1.2 Precompiling files

The compiler tries to compile only the source modules that are really necessary, however, sometimes it is necessary to specify where to look for some identifiers, see Section 3.1.7.2 [Meaning of identifiers], page 7.

This concerns mainly the case when a programmer uses a category name but he doesn't previously use a structure type where the category is defined — here the system has no possibility to find out which module should contain the definition of the category.

[1] It is not required by the compiler that it should be in the first line, however, it is strongly recommended since already initial comments can use non-ASCII characters; on the other hand, the `charset` command in the first line is expected by the web presenter!

In these cases, it is necessary to advice the compiler where to find these identifiers, since otherwise they are considered to be *free variables*, which is *not* a compile error!

Use a simple command

```
parse <ident>;
```

where `<ident>` is (in the sample given above) the identifier of a structure data type that contains the particular category. Generally, the `parse` command interrupts the processing of the actual file and the system primarily processes the file for the identifier `<ident>`.

4.1.3 Preprocessor

The preprocessor performs *only string* replacement. No macro support is available in LanGR. The replacement is defined by the command

```
preprocess <ident> <string>;
```

The `preprocess` command can be used exclusively on the global level. It defines the global identifier `<ident>` — each occurrence of this identifier will be replaced by the string constant `<string>` (it must be always non-empty). The identifier `<ident>` is replaced only in those places where a valid identifier is expected, i.e. not in a string constant, in a comment and so on.

Wherever the identifier `<ident>` is found, it is replaced by the content of `<string>` and then the processing of the source continues (by analysing the first token of `<string>`).

An example:

```
preprocess prep_ident "10 + 20";

i = prep_ident; // prep_ident will be replaced
s = "prep_ident is not replaced here";

preprocess pi "i = 3.14;";

preprocess count_pi "
declare circle;
integer function circle integer a {
pi // sets 'pi' to the Pi value
api = 2 * i * a; // computes the length of
// the circle
return api;
};
";

count_pi
```

The code really processed by the compiler (after all replacements made) is then:

```
i = 10 + 20;
s = "prep_ident is not replaced here";

declare circle;
integer function circle integer a {
i = 3.14;
```

```
api = 2 * i * a;
return api;
};
```

Multiple definition of the preprocessor identifier is an error. A cycle in the preprocessor identifiers is also an error.

Note: Inside a string which replaces a preprocessor identifier it is necessary to put a backslash in front of every occurrence of the double-quotes.

Note: Due to the implementation of the reading of the source modules, the following situation is also erroneous:

```
// let this example be enclosed in the file prep.rlm

preprocess prep "something";

if prep then { //anything
};
```

Once this file is opened (when the meaning of `prep` identifier is searched), the information about processing `prep.rlm` file is written to the stack of opened files. Then `prep` is correctly defined and used. The use of `prep`, however, implies that new "source" is opened (the source is the string `"something"`), and this source is opened also for the identifier `prep`. Now there are two files opened on the stack for the same identifier, which produces an error. Hence a preprocessor identifier cannot be used in its definition file.

4.1.4 Invoking the compiler

This section concerns invoking the compiler from the command line to produce an executable file which implements the rules/groups — this doesn't concern invoking the compiler automatically, e.g. from the web-presenter.

The complete list of the compiler switches is always available from the on-line help of the compiler. Generally, every switch on the command line must be followed by a value, e.g. `-o unix`.

Let's look at the main switches on the command line:

`-c` just produces the highlighting of the source file — the complete analysis of the source is made (up to the compilation part); it is used mainly for the web-presenter;

`-e` compilation of the source modules — the output is the C++ code implementing the rules (it must be later compiled by the general C++ compiler);

`-p <path>` `<path>` is added to the list of directories where modules and other files needed for the compilation are searched;

`-o <charset>`
output encoding will be `<charset>`, see Section 4.1.1 [Character sets], page 46;

`-i <token>`
the compilation will start with processing the identifier `<token>`;

`-f <filename>`
the output will be redirected to `<filename>`;

`-0` the input will be read from standard input, even if `-i` has been used.

Any corruption in the command line leads to displaying the on-line help.

4.2 Run-time

A C++ source file is produced by the compiler out of the input modules. This file must be compiled by an available C++ compiler (which is not a part of LanGR) to obtain executable rules.

The executable file accepts several switches on the command line. The actual list of switches is available by the switch `-h`. The list of the main switches follows:

`-r <rg>` rule or group `<rg>` will be applied (in a cycle until nothing changes in the input) to every input sentence, see Section 3.6.2 [Groups], page 41;

`-i <charset>`
input alphabet;

`-o <charset>`
output alphabet (the function `output`);

`-e <charset>`
error stream alphabet (the functions `print`, `debug`);

`-I <filename>`
the input will be read from `<filename>` (default is stdin);

`-O <filename>`
the output goes to `<filename>` (default is stdout);

`-E <filename>`
the error stream will be written to `<filename>` (default stderr);

`-p <path>` see Section 4.1.4 [Invoking the compiler], page 48;

`-l` list of all rules and group names compiled and hence usable in `-r`;

`-d <integer>`
sets the debug level, see Section 4.3 [Debugging], page 50 (default is 0);

`-R<option>`
sets a special switch `<option>`. The list of available `<option>`s follows:

`firstoverload`
during run-time, untyped values (`undef`, `N/A`) can make the selection of the appropriate overload impossible. By default, an error is produced in these cases. With `-Rfirstoverload` the first acceptable overload is selected and the program continues. Use with caution, although sometimes it is necessary.

4.3 Debugging

LanGR supports no enhanced tools for debugging the program like debuggers for C/C++ or other. The main reason is that the development of such a tool is almost as difficult as to develop the LanGR compiler.

LanGR, however, makes it possible to trace the run-time using the outputs to the error stream (like `#ifdef DEBUG`). The special `debug` command has been made available for this purpose:

```
debug <level> <args>;
```

The program always runs in some *debug level* — higher level implies more detailed debugging. This level can be changed on the command line by using the `-d`, see Section 4.2 [Run-time], page 49.

All `debug` commands where `<level>` is higher that the actual debug level are ignored. The others behave exactly as the command sequence `print <args>; print;` does.

It is not determined which `<level>` values are recommended to use. In any case, in the levels higher that `10` the internal program messages begin to appear.

On the other hand, the levels `0,1,2` are used to change the level of detail when an error appears. Hence the user should use the levels `2-9`.

References

1. Karlsson, F., Voutilainen, A., Heikkila, J., and Antilla, A. (eds.): *Constraint Grammar — A Language-Independent System for Parsing Unrestricted Text*, Mouton de Gruyter, Berlin & New York, 1995
2. Stroustrup, B.: *The C++ Programming Language*, Addison-Wesley, 1997
3. Hajič, J., Krbec, P., Oliva, K., Květoň, P., Petkevič, V.: *Serial Combination of Rules and Statistics: A Case Study in Czech Tagging.* In Proceedings of the ACL-2001, Toulouse, France

Index

#

\

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

U

V

W

Table of Contents