THE REFAL PLUS PROGRAMMING LANGUAGE

Ruten Gurin, Sergei Romanenko

Intertekh
Moscow 1991

CONTENTS

INTRODUCTION

Chapter I. PROGRAMMING IN REFAL PLUS
1.Your first Refal Plus program
2.Data structures
2.1.Ground expressions
2.2.Representation of data by ground expressions
2.3.Objects and values
2.4.Garbage collection
3.Evaluation and analysis of ground expressions
3.1.Result expressions
3.2.Variables
3.3.Formats of functions
3.4.Patterns
3.5.Paths, rests, and sources
3.6.Delimited paths
3.7.Result expressions as sources
3.8.Right hand sides
4.Functions defined in the program
4.1.Function definitions
4.2.Local variables
4.3.Recursion
5.Failures and errors
5.1.Failures produced by evaluating result expressions and paths
5.2.Matches
5.3.Failure trapping
5.4.Control over failure trapping
5.5.Meaning of right hand sides
5.6.Failing and unfailing functions
6.Logical conditions
6.1.Conditions and predicates
6.2.Conditionals
6.3.Logical connectives
6.4.Example: formal differentiation
6.5.Example: comparison of sets
7.Direct access selectors
8.Functions returning several results
8.1.Ground expression traversal
8.2.Quicksort
9.Iteration
10.Search and backtracking
10.1.The queens problem
10.2.The sequence problem
11.Example: a compiler for a small imperative language
11.1.The source language
11.2.The target language
11.3.The general structure of the compiler
11.4.The modules of the compiler and their interfaces
11.5.The main module

11.6.The scanner
11.7.The parser
11.8.The code generator
11.9.The dictionary module

Chapter II. SYNTAX AND SEMANTICS OF REFAL PLUS
1.Notation for syntax description
2.Natural semantics description
3.Lexical structure of program
3.1.Comments
3.2.Tokens
3.3.Key words
3.4.Character symbols
3.5.Word symbols
3.6.Numeric symbols
3.7.Variables
3.8.Normalization of the token stream
4.Objects and values
5.Ground expressions
5.1.Ground expression syntax
5.2.Static and dynamic symbols
5.3.Symbolic expression names
5.4.Elimination of symbolic expression names
6.Variable values and environments
7.Result expressions
7.1.Syntax
7.2.Evaluation of result expressions
7.3.Examples
8. Patterns
8.1.Syntax
8.2.Pattern matching
8.3.Examples
9.Hard expressions
9.1.Syntax
9.2.Matching against hard expressions
9.3.Examples
10.Paths
10.1.Syntax
10.2.Evaluation of paths
10.3.Conditions
10.4.Bindings
10.5.Searches
10.6.Matches
10.7.Delimited paths
10.8.Negative conditions
10.9.Fences
10.10.Cuts
10.11.Failures
10.12.Right hand sides
10.13.Error generators
10.14.Error traps
10.15.Alternatives
10.16.Alternative matches

10.17.Result expressions as sources
11.Function definitions
12.Declarations
12.1.Constant declarations
12.2.Object declarations
12.3.Function declarations
12.4.Trace directives
13.Context dependent restrictions
13.1.Elimination of redundant constructs
13.2.Restrictions imposed by function declarations
13.3.Restrictions on the use of references to functions
13.4.Restrictions on the use of variables
13.5.Restrictions on the use of cuts
14.Modules
15.Execution of program

Chapter III. LIBRARY OF FUNCTIONS
1.How to use library functions
2.ACCESS: direct access to ground expressions
3.APPLY: application of functions passed as arguments
4.ARITHM: arithmetic operations on integers
5.BOX: box operations
6.CLASS: predicates for determining classes of symbols
7.COMPARE: comparison operations
8.CONVERT: data convertions
9.DOS: calls to the operating system
10.STDIO: standard input/output
11.STRING: string operations
12.TABLE: table operations
13.VECTOR: vector operations

REFERENCES
INDEX OF LIBRARY FUNCTIONS

INTRODUCTION

Refal Plus is a dialect of the programming language Refal.

Refal (Recursive Function Algorithmic Language) was designed by V.F.Turchin as a tool for describing the semantics of other algorithmic languages [Tur 86]. Later, when reasonably efficient Refal implementations had been created [BsR 77], [Rom 87a], Refal was used as a symbol manipulation language in such fields as computer algebra, compiler and interpreter writing, artificial intelligence, etc.

The principal data type in Refal are arbitrary trees, referred to as ground expressions. In programs and text files ground expressions are represented by linear sequences of symbols and parentheses, with parentheses being properly paired. Symbols represent such elementary data objects as characters, words, numbers and references to objects).

The principal means of analyzing and accessing ground expressions is pattern matching. Refal patterns may contain symbols, parentheses, and variables. If matching a ground expression against a pattern succeeds, the pattern's variables are bound to the corresponding components of the ground expression, which can be used later for building new ground expressions.

A Refal program may contain function definitions. Each function takes as argument a ground expression and returns as its result a ground expression. Functions can call each other. In particular, a function can call itself (directly as well as indirectly, through other functions), in which case the function is said to be recursive. And it is recursion that is the principal way of structuring the control in Refal programs.

Refal Plus has been developed to take into account the experience gained from the design, implementation and use of such languages as Basic Refal [BsR 77], Refal-2 [Rom 87a], Refal-4 [Rom 87b], Refal-5 [Tur 89], and RL [Rom 88].

As compared to the other Refal dialects, Refal Plus provides the following features.

*** More advanced modules

Each module is divided into two components: the interface of the module and the implementation of the module. The interface contains the parts of the module that may be visible in other modules, whereas the implementation contains the parts of the module that are invisible in other modules.

The interface of a module may contain any declarations, which means that not only function declarations may be exported, but also declarations of constants and objects (such as boxes and i/o channels). When a function declaration is exported, not only the function name becomes visible, but also the formats of the function's arguments and results, which enables the calls to the function to be checked for correctness at compile time, rather than at run time.

For a module to be compiled, there must be known the interfaces of other modules, rather than other module's implementations. Thus, a module can be compiled even if the implementations of other modules have not been created. Besides, the fact that a module's implementation has been modified does not necessitate recompiling the modules importing that module. Thus arbitrary intermodule dependencies are allowed (including the cyclic ones).

*** Static declarations of dynamic objects

All objects that can be created dynamically at run time (such as i/o channels, boxes, vectors, and tables) can also be declared statically, in which case they are given symbolic names to be used in the program text for referencing the objects.

*** Function declarations

Each Refal Plus function is declared as either failing or unfailing one. The evaluation of a call to a failing function can result in returning a special "failure" value. For example, all predicate functions return either an empty ground expression or a "failure". On the other hand, an unfailing function never returns a "failure".

It should be noted that earlier Refal dialects enabled the programmer to define only unfailing functions.

One more feature of Refal Plus is the possibility of defining functions accepting several arguments and returning several results. The number and type of a function's arguments is said to be the function's arity, whereas the number and type of the function's results is said to be the function's co-arity.

The arity and co-arity of a function are specified by declaring the function's input and output formats, which are patterns containing symbols, parentheses, and variables. The input format imposes syntax restrictions on the form of the calls to the function, whereas the output format imposes restrictions on the contexts in which the calls to the function may appear. By stripping the input format of all the symbols and parentheses, we get the variable sequence describing the function's arity. By stripping the output format, we get the description of the function's co-arity.

The explicit function format declarations allow many errors to be detected at compile time, and also reduce the costs of evaluating the function calls.

It should be noted that earlier Refal dialects assumed each function to accept a single argument and to return a single result.

*** Failure and error trapping

If evaluating a Refal Plus construct terminates, it either succeeds, fails, or produces an error.

If the evaluation succeeds, the result returned by the construct is a ground expressions. (The format of the returned expression is always known in advance, which permits Refal Plus implementations to represent the result by a tuple of ground expressions.)

If the evaluation fails, the result returned is a "failure".

If the evaluation produces an error, the result returned is an "error" value containing a ground expressions (which, usually, is an error message).

Refal Plus provides several constructs enabling failures and errors to be caught and analyzed.

*** Input/output of ground expressions

Refal Plus provides functions that enable programs to input and output character strings as well as character representations of ground expressions, the conversion of ground expressions into character sequences and vice versa being done automatically.

*** Operations on boxes, vectors, and tables

Refal Plus provides a way to deal with dynamically created objects such as boxes, vectors, strings, and tables. Boxes are treated in the same way as in Refal-2, whereas vectors, strings, and tables are a feature of Refal Plus.

A box is an object containing a ground expression.

A vector is an object containing a finite sequence of ground expressions.

A string is an object containing a finite sequence of characters.

Boxes, vectors, and strings can be accessed via reference symbols pointing to these objects. Refal Plus provides functions for creating, accessing and updating boxes, vectors, and strings, including accessing and updating individual components of vectors and strings.

A table is an object containing a finite set of keys, each key associated with its value. The keys as well as values are ground expressions. A table can be accessed via reference symbols pointing to the table. Refal Plus provides functions for creating and copying tables, for getting the value associated with a key in a table, and getting all the keys contained in a table. Essentially, a table is a representation of a function with the finite domain.

*** "Vector" representation of ground expressions

The present implementations of Refal Plus are based on the "vector" representations of ground expressions [AbR 88], which allows the copying of ground expressions to be reduced to copying a pair of pointers to the expression's representation.

The cheapness of the copying operation permits Refal programs to be written in functional style, whereas the earlier Refal implementations forced the programmer to be careful with copying, thereby inducing him/her to stick to the imperative style.

The objects that have become inaccessible to the program are automatically destroyed by the garbage collector provided by the Refal Plus implementations.

Chapter I. PROGRAMMING IN REFAL PLUS

This chapter gives a step-by-step tutorial introduction to the language Refal Plus and provides a diverse group of program examples demonstrating some of the ways in which Refal Plus can be used to solve problems. A complete description of Refal Plus is given by Chapter II, "Syntax and Semantics of Refal Plus", where you can find information about certain subtle points and technical details. Some of the program examples may contain calls to unknown functions, in which case you may consult Chapter III, "Library of Functions", as well as "Alphabetical Index of Functions".

1.YOUR FIRST REFAL PLUS PROGRAM

To maintain the historically established tradition, we begin by considering a simple program in Refal Plus:

```
$use STDIO;                 /* Import i/o functions */
                            /* from the module STDIO */
Main                        /* Define of the main function */
  = <Println "Hello!">;     /* Print a line */
```

This program consists of two directives. The first directive

```
$use STDIO;
```

states that the program is going to use library input/output functions, which are to be imported from the module STDIO. The second directive is the definition of the function Main, and, by convention, the execution of a Refal Plus program always begins by evaluating the call to the function Main.

The argument of the function Main must be empty. In the above program, the function Main calls the library function Println with the argument "Hello!", thereby causing the character string

```
Hello!
```

followed by the character "new line", to be sent to the standard output device. Then the execution of the program terminates.

2.DATA STRUCTURES

2.1.GROUND EXPRESSIONS

All data processed by Refal Plus programs are so-called ground expressions.

Here are three examples of ground expressions

```
"John" "Smith" 33 "years"
("Dave" 17) ("Mary" 24) ("Elizabeth" 6)
("my" "house") "has" ("large" ("light" "windows"))
```

The salient feature of the above examples is the use of parentheses. If we modify the expressions by rearranging the parentheses, the structure of the expressions will be modified, changing the implied meaning of the expressions.

In addition to parentheses, the above expressions contain symbols. Here are a few examples of symbols:

```
"John" "johN" "bye-bye" 1988 -99999999999999
```

In general, ground expressions consist of symbols and parentheses. A ground expression is a sequence of zero or more ground terms. A ground term is either a symbol or a ground expression enclosed in parentheses "(" and ")". Thus, a ground expression is a sequence of symbols and parentheses, in which the parentheses are "properly paired".

When in computer memory, ground expressions are usually stored as tree-structured objects. Nevertheless, in order to be input or output (printed, written to a file, read from a file, etc.), a ground expressions has to be represented as a linear sequence of characters.

Refal Plus implementations enable the ground expressions to be input or output, with all necessary conversions performed automatically.

A ground expression represented by a character stream is a sequence of tokens, each token representing either a parenthesis or a symbol. Tokens may be separated by spaces, which are ignored unless they are essential to separate two consecutive tokens. (New line characters are considered to be equivalent to spaces.)

The following symbols can appear in source Refal Plus programs as constants: character symbols, word symbols, and numeric symbols.

A character symbol corresponds to a printable character. A sequence of several character symbols is written as a single string consisting of the corresponding characters and enclosed in acute accents.

A word symbol corresponds to a character string and is written as the corresponding string enclosed in double quotes.

If a word symbol begins with either a capital letter, a question mark (?), or an exclamation mark (!), and contains only letters, digits, minus signs (-), question marks (?), and exclamation marks (!), the double quotes enclosing the symbol may be omitted.

Here are examples of words:

```
"John"
"A-Word"
"a-very-very-long-Word"
X-25m3s--
```

```
"equal?"
?-?
?
```

A numeric symbol corresponds to a signed integer, and is written as a non-empty sequence of decimal digits, which may be preceded by one of the characters "+" or "-". For example:

```
237
-99999999999999999999999999999999999999999999999
+13
```

Numeric symbols may be arbitrary large.

2.2.REPRESENTATION OF DATA BY GROUND EXPRESSIONS

Ground expressions are especially convenient for representing symbolic (i.e. not purely numeric) data.

For example, suppose we want to deal with algebraic formulae represented by ground expressions. In this case, we have to devise a way of representing constants, variables, and formulae formed by applying a binary operator to two smaller formulae. We may choose, for example, the following representation.

Let [p] denote the ground expression that represents the formula p. Then, numbers may be represented by the corresponding numeric symbols, variables by the corresponding word symbols, and formulae formed by applying binary operators according to the following rules:

```
[p+q]      =     ("plus" [p] [q])
[p-q]      =     ("minus" [p] [q])
[pÕq]      =     ("mult" [p] [q])
[p/q]      =     ("div" [p] [q])
  q
[p ]       =     ("power" [p] [q])
```

Thus the formula

```
    2
(X+Y )-512
```

is to be represented by the ground expression

```
("plus" ("minus" X ("power" Y 2)) 512)
```

The next example is the problem of representing chess positions by ground expressions.

First of all we have to denote the name and color of each piece. For example, ("white" King"), ("black" "Pawn"). Then we have to specify the square occupied by each piece. For example, ("e" 2), ("h" 7). Now a position may be represented as a sequence of ground terms, each term specifying the name, color,

and square of a piece. For example

```
        (("white" "King")        ("g" 5))
        (("black" "King")        ("a" 7))
        (("white" "Pawn")        ("c" 6))
        (("white" "Knight")      ("g" 1))
        (("black" "Knight")      ("a" 8))
```

2.3.OBJECTS AND VALUES

In the broad sense, "object" is usually understood to mean an entity that exists in time and may vary, but, nevertheless, does not lose its identity.

A good example of objects is a human, who gets born, grows up, develops, and dies, but, nevertheless, remains, in a sense, the same person.

Another classic example is due to Heraclitus (the prime of whose creative forces falls approximately on the years 504-501 BC). Heraclitus taught that one cannot enter twice the same river, since, "even if you enter the same river, the water running against you is always new". Thus, the river may also serve as a good example of objects.

In the broad sense, "value" is usually understood to mean an entity that is unable to vary, does not develop, and, in a sense, exists out of time.

It is unknown whether values exist in real life, but they are the favorite subject of the mathematicians. For example, the number 25 is a typical value of that kind.

A value may, certainly, be regarded as a special, degenerate, case of object (i.e. as a rigid object unable to develop). Nevertheless, the term "object" will be usually applied only to "proper" objects, which are not values.

Since objects may vary, they are more difficult to deal with than values are. Thus objects are often provided with names. The basic property of names is that a name is unambiguously associated with an object (i.e. a name unambiguously identifies the object). In contrast to objects, their names are typical values, there being no changes in the names in spite of there being changes in the objects. For example, the state of the River Thames is continuously changing, but, nevertheless, it has no effect on the word "Thames". One more example is given by the particulars of a person: the family name, the first name, the date and place of birth, etc.

Within the scope of Refal Plus, the terms "object" and "value" have a more narrow sense.

A Refal Plus value is a ground expression.

A Refal Plus object is a "container", in which there can be kept ground expressions and other information.

Refal Plus objects may be created at compile time as well as at run time. Each object is created simultaneously with a reference symbol, which is said to reference to, and to be the name of, the object. The basic property of the name of an object

is that it must be different from all other reference symbols existing at the moment the object is being created. Owing to this property, each reference symbol corresponds to a unique object, and equal reference symbols correspond to one and the same object.

The interrelation between the name of an object, the object, and the object's contents can be represented by the following picture:

```
R --> [ ... ]
```

Refal Plus programs deal with object of the following types.

Function objects contain compiled function definitions, and are created at compile time.

All other objects may be created statically (i.e. at compile time) as well as dynamically (i.e. at run time).

Box objects are designed for storing ground expressions, each box containing one ground expression

Table objects are designed for storing unordered sets of ordered pairs, each pair consisting of two ground expressions. The first component of a pair is said to be a key, whereas the second component is said to be the value associated with the key. All keys appearing in a table must be different from each other. Thus, each key in a table unambiguously corresponds to its value. Thus, a key uniquely determines its value.

Channel objects are designed for performing input/output operations.

Vector objects are designed for storing finite sequences of ground expressions.

String objects are designed for storing finite character sequences.

2.4.GARBAGE COLLECTION

In spite of the fact that, at run time, Refal Plus programs can create objects, there is no explicit way in which the objects can be destroyed. Thus, the computer memory may well be filled with new and new objects, although many of them may not be needed any more. Theoretically, this is no problem, but, in practice, Refal programs are to be run by real computers with limited memory capacity. For that reason, all Refal Plus implementations include a garbage collector.

Garbage collection is automatically started each time the free memory is exhausted, in order to find and destroy all objects that, being inaccessible via the references contained in variable values, are thus unable to influence the program's behavior.

Figure 2.1 schematically shows the variable values as well as several objects along with their contents. The stars denote the parts of expressions that are not reference symbols. To facilitate the discussion, all objects are labeled with numbers.

The corresponding numbers denote reference symbols appearing in the ground expressions.

It can be easily seen that reference 1 appearing in the variable values enables the access to object 1 and, indirectly (via object 1), to object 4, whereas reference 2 enables the access to object 2 and, indirectly (via object 2), to objects 4, 5, 6, 3. Thus, there is no way of getting information from objects 7 and 8. Therefore, if the garbage collection started at this moment, objects 7 and 8 would be destroyed. Now, if reference 1 were removed from the variable values, object 1 would become inaccessible. But, if reference 1 were retained, and reference 2 removed, then all the objects would become inaccessible, except objects 1 and 4.

```
VARIABLE VALUES:
[* * 1 * * * * * * 2 * * * * *]

1:[* * * 4]
2:[4 * * 5]
3:[* * 5]
4:[* * *]
5:[* 6 * 3]
6:[* * 4 *]
7:[3 * 8]
8:[* 7]
```

Fig.2.1. Objects and references.

3.EVALUATION AND ANALYSIS OF GROUND EXPRESSIONS

3.1.RESULT EXPRESSIONS

Refal Plus result expressions are, in a sense, an analog to the well-known arithmetic expressions. For example, the arithmetic expression X*Y+3 corresponds to the Refal Plus result expression

```
<"+" <"*" sX sY> 3>
```

Each pair of angular brackets designates a function call of the form <Fname Re>, where Fname is the name of the function to be called, and Re is the argument to be passed to the function. Thus, the arguments of function calls are always enclosed in angular "functional" brackets, which eliminates the necessity to use parentheses for indicating the order in which the subexpressions are to be evaluated. For example, the expression X*(A+B) rewritten in Refal becomes

```
<"*" sX <"+" sA sB>>
```

whereas the expression X*A+B is written in Refal as

```
<"+" <"*" sX sA> sB>
```

Result expressions, similarly to arithmetic expressions in other languages, are used for producing new values from other ones. Thus, a result expression is evaluated by replacing all its variables with their values and evaluating all function calls. If there are nested function calls, the inner calls are evaluated before the surrounding ones.

It is obvious that, for a result expression to be evaluated, it is necessary to know the values of the variables appearing in the expression. The information about the variable values will be referred to as an environment. The notation

```
{V1 = Ge1, ..., Vn = Gen}
```

will be used for denoting the environment in which the variables V1, ..., Vn have the respective values Ge1, ..., Gen.

As can be seen from the above, the representation of arithmetic expressions by result expressions is rather clumsy. Nevertheless, it does have certain advantages.

The point is that the choice of one or another notation is determined by the nature of the objects to be dealt with, as well as by the set of operations to be applied to the objects.

It is reasonable to choose the notation in such a way that the most frequently used operations be denoted as concisely as possible. But the most succinct notation is, certainly, no notation at all, i.e. an empty place!

As far as arithmetic expressions are concerned, we have two basic operations: addition and multiplication. One of the operations may be denoted by empty place, and the common practice is to omit the operator of multiplication.

On the other hand, the principal data dealt with by Refal Plus are ground expressions, rather than numbers. Since the basic operations on ground expression are the concatenation of two expressions and the enclosing of an expression in parentheses, it is for these operations that the syntax of Refal Plus provides a very concise notation.

Namely, if Re' and Re" are result expressions, so is the construct

```
Re' Re"
```

which means that Re' and Re" are to be evaluated and the values returned are to be concatenated to produce the result of the whole expression. Thus, if the evaluation of Re' and Re" results in returning ground expressions Ge' and Ge" respectively, the ground expression Ge' Ge" is returned as the result of evaluating Re' Re".

If Re is a result expression, so is the construct

```
( Re )
```

which means that Re is to be evaluated and the value returned is to be enclosed in parentheses to produce the result of the whole expression. Thus, if the evaluation of Re results in returning a ground expression Ge, the ground expression (Ge) is returned as the result of evaluating (Re) .

For example, the result of evaluating the result expression

```
sX '+' sY (eZ)
```

in the environment {sX = 25, sY = 36, eZ = A (B C) D} is the ground expression

```
25 '+' 36 (A (B C) D)
```

3.2.VARIABLES

Each variable in Refal Plus begins with a variable type designator. The type designator specifies the set of values the variable can be bound to, and must be one of the four letters: s, t, v, or e. The variables are, accordingly, distinguished into four classes: s-variables, t-variables, v-variables, and e-variables.

A variable's value should be consistent with the type of the variable: an s-variable's value must be a symbol, a t-variable's value must be a ground term, a v-variable's value must be a non-empty ground expression, and, finally, an e-variable's value may be any ground expression,

In the following, the term "ve-variable" will be understood to mean "a variable that is either a v-variable or an e-variable".

3.3.FORMATS OF FUNCTIONS

From the purely formal point of view, all Refal Plus functions are assumed to take a single argument and to return a single result. In many cases, however, the structure of a function's argument and result is known in advance. For example, the function "+" is known to accept a ground expression consisting of two symbols and to return a ground expression consisting of a single symbol.

The restrictions imposed on the argument and result of a function are specified by the declaration of the function. For example, the declaration of the function "+" has the form:

```
$func "+" sX sY = sZ;
```

In general, the declaration of a function Fname has the form

```
$func Fname Fin = Fout;
```

where Fin is the input format of the function, and Fout is its output format. The formats of functions may contain symbols, parentheses, and variables. The variable indices in formats are insignificant, serve as comments, and may be omitted.

All input and output formats must be "hard", which means that any subexpression of a format may contain no more that one ve-variable at the top level of parentheses. For example, the format (e)(e) is hard, whereas the format e A e is not hard, containing as it does two e-variables at the same level of parentheses.

All inputs to, and results of, a function must have the structure specified by the function's declaration. The function's declaration must precede all references to the function made in the result expressions appearing in the program. If the function is defined in the program, its declaration must explicitly appear in the program prior to the definition. Otherwise, if the function is defined in other module, its declaration must be imported into the program by a directive $use.

When the program is being compiled, the compiler verifies that the argument expressions in the calls to the function are consistent with the input format of the function. For example, consider the result expression

```
<"+" 2 <"+" sX sY>>
```

The inner call is obviously correct. But, to check the outer call, we have to make use of the information about the structure of the results returned by the function "+". Thus, on replacing <"+" sX sY> with the output format of the function "+" we get <"+" 2 s>. Now we see that the argument of the outer call conforms to the input format of the function "+". On the other hand, the result expression

```
<"+" 2 <"+" sX sY> 3>
```

is regarded as illegal, because the argument of the outer call consists of three symbols, despite the input format of the function "+" requiring the argument to consist of two symbols.

Thus, specifying the input and output formats enables many errors to be found at compile time, rather than at run time.

3.4.PATTERNS

Patterns provide the principal way of analyzing ground expressions.

Patterns may contain symbols, parentheses, and variables. For example:

```
A B C
tX (eY B)
```

A pattern may be regarded as representing the set of all

ground expressions that can be produced from the pattern by replacing the pattern's variables by some values consistent with the types of the variables. For example, the pattern A eX represents the set of ground expressions beginning with the symbol A, and the pattern sX sY the set of ground expressions consisting of exactly two symbols.

If there are several occurrences of the same variable in a pattern, all the occurrences must be bound to the same value. For example, the pattern tX tX represents the set of ground expressions consisting of two equal terms.

Let Ge be a ground expression, and P a pattern. Then Ge can be matched against P to determine whether Ge has the structure specified by P. If so, the matching of Ge against P is said to succeed, otherwise to fail.

If the matching of Ge against P succeeds, the variables appearing in P are bound to the corresponding components of Ge. Thus, the result of matching Ge against P is an environment Env. For example, the result of matching the ground expression AAA BBB CCC against the pattern eX sY is the environment {eX = AAA BBB, sY = CCC}.

Now let us try to match the ground expression A B C against the pattern e1 sX e2. It can be easily seen that the matching can succeed in three different ways, resulting in three different environments:

```
{e1 = ,    sX = A, e2 = B C}
{e1 = A,   sX = B, e2 = C}
{e1 = A B, sX = C, e2 = }
```

What is to be considered the result of matching in such situations? Refal Plus solves the problem in the following way. All variants of matching are considered to be acceptable, but some of variants "take precedence" over others.

More specifically, let Env1 and Env2 be different variants of matching Ge against P. Consider all variables appearing in P. Since Env1 and Env2 are different, P must contain some variables whose values in Env1 and Env2 are different. Let V be the left-most of such variables, and compare the length of the values assigned to V by Env1 and Env2. If the value assigned by Env1 is shorter than the value assigned by Env2, then Env1 is assumed to "precede" Env2 (i.e. to take precedence over Env2), otherwise Env2 is assumed to "precede" Env1.

For example, matching the ground expression (A1 A2 A3) (B1 B2) against the pattern e1 (eX sA eY) e2 results in the following set of environments

```
{e1 = , eX = ,      sA = A1, eY = A2 A3, e2 = (B1 B2)}
{e1 = , eX = A1,    sA = A2, eY = A3,    e2 = (B1 B2)}
{e1 = , eX = A1 A2, sA = A3, eY = ,      e2 = (B1 B2)}
{e1 = (A1 A2 A3), eX = ,   sA = B1, eY = B2, e2 = }
{e1 = (A1 A2 A3), eX = B1, sA = B2, eY = ,   e2 = }
```

where the variants of matching are listed in accordance with

their precedence, i.e. the first variant comes first, etc.

If the variants of matching are ordered as described above, the matching is said to be done from left to right. Refal Plus, however, enables the matching to be also done from right to left, which means that, instead of comparing the values of the leftmost variable, we have to compare the values of the rightmost variable. The direction of matching can be changed by prefixing the key word $r to the pattern. For example, if the ground expression (A1 A2 A3) (B1 B2) is matched against the pattern $r e1 (eX sA eY) e2, the set of variants of matching will be ordered as follows:

```
{e1 = (A1 A2 A3), eX = B1, sA = B2, eY = ,    e2 = }
{e1 = (A1 A2 A3), eX = ,   sA = B1, eY = B2, e2 = }
{e1 = , eX = A1 A2, sA = A3, eY = ,       e2 = (B1 B2)}
{e1 = , eX = A1,    sA = A2, eY = A3,     e2 = (B1 B2)}
{e1 = , eX = ,      sA = A1, eY = A2 A3, e2 = (B1 B2)}
```

3.5.PATHS, RESTS, AND SOURCES

Result expressions and patterns may be used to construct larger syntax units, paths. Whereas result expressions may be considered as an analog to arithmetic expressions, paths are an analog to statements (such as assignment statements, loop statements, etc.).

A path is evaluated with respect to an environment. If the evaluation terminates, it returns a value.

Refal Plus is rather sparing in using keywords and delimiters, which may cause some syntactical problems. For example, some paths begin with result expressions. Let Q be such a path and Re a result expression. Consider the construct Re Q obtained by juxtaposing Re and Q. This construct is obviously ambiguous, because it is impossible to determine the boundary between Re and Q.

This ambiguity can be removed by inserting a delimiter (for example, a comma) between the result expression and the path: Re , Q. Radical though this solution may seem, in many cases the delimiter would be superfluous, Q being easy to separate from the preceding construct.

Such "good-looking" paths, easy to separate from the preceding constructs, are referred to as rests.

Another important class of paths is formed by sources, whose principal syntactical feature is that a source cannot contain comma at the top level of curly braces.

Henceforth, we shall denote paths by Q, rests by R, and sources by S.

Now let us consider several simple kinds of paths, rests, and sources.

3.6.DELIMITED PATHS

Any path Q can be turned into a rest by prefixing a comma

to Q. Thus we get the delimited path

, Q

which in all respects is equivalent to the original path Q (except that, syntactically, it is a special kind of paths, a rest).

3.7.RESULT EXPRESSIONS AS SOURCES

Any result expression Re is a source, and, therefore, a path. Thus it can be turned into the rest

, Re

by prefixing a comma to Re.

Evaluating a source of the form Re amounts to evaluating the result expression Re. If this evaluation results in returning a ground expression Ge, then Ge is taken to be the result of the source Re.

3.8.RIGHT HAND SIDES

A construct of the form

= Q

where Q is a path, is referred to as a right hand side. Syntactically, a right hand side is a rest, and, therefore, a path.

Evaluating a right hand side = Q amounts to evaluating the path Q. If the evaluation of Q results in returning a ground expression Ge, then Ge is taken to be the result of the whole right hand side.

The difference between the path Re and the path = Re seems to be purely syntactic, but this is not so. Actually, the subtle semantic difference between the two constructs does exist and manifest itself in cases where the evaluation of Re results in returning a failure. Later this question will be given due consideration, but now we won't dwell on it any more.

4.FUNCTIONS DEFINED IN THE PROGRAM

4.1.FUNCTION DEFINITIONS

A Refal program consists of function definitions, each definition having either of the two forms:

```
Fname \{ Snt1; Snt2; ... Sntn; };
Fname  { Snt1; Snt2; ... Sntn; };
```

where Fname is the name of the function being defined, and Snt1, Snt2, ..., Sntn are sentences. (Being, at present, of no importance, the subtle difference between "\{" and "{" will be explained later.)

Each sentence Sntj is of the form Pj Rj, with Pj being the input pattern of the sentence, and Rj the rest of the sentence.

A function definition specifies the way in which the calls to the function are to be evaluated. Suppose a call

```
<Fname Re>
```

to the function Fname is to be evaluated. Then the result expression Re is evaluated. If a ground expression Ge is returned, an attempt is made to match Ge against the input patterns P1, P2, ..., Pn, in order to find the first pattern Pj such that matching Ge against P succeeds. Let Env be the "first" variant of matching Ge against P. Then the rest Rj is evaluated in the environment Env. If a ground expression Ge' is returned, this expression is taken to be the result of evaluating the function call.

For example, let us consider a function Sumsq computing the sum of the squares of two numbers. Here is the definition of this function written in traditional notation

```
Sumsq(X,Y) = X*X + Y*Y
```

which may be rewritten in Refal in the following way:

```
$func Sumsq sX sY = sZ;

Sumsq
  {
  sX sY = <"+" <"*" sX sX> <"*" sY sY>>;
  };
```

It should be noted that the declaration of a function must precede the function's definition as well as the calls to the function, since the information provided by the declaration is necessary for compiling the function's definition as well as the calls to the function.

If the function declaration has the form

```
$func Fname Fin = Fout;
```

the compiler verifies that the input patterns P1, P2, ..., Pn are instances of the input format Fin, whereas all the rests R1, R2, ..., Rn are certain to return ground expressions satisfying the output format Fout.

Some constructs appearing in function definitions may be abbreviated in the following way.

If a sentence Sntj has the rest Rj consisting of a single comma, the rest can be omitted, so that the sentence takes the form Pj.

If a function definition contains a single sentence Snt, i.e. has the form

```
Fname \{ Snt; };
```

it can be abbreviated to

```
Fname Snt;
```

For example, the above definition of the function Sumsq can be written as

```
Sumsq  sX sY = <"+" <"*" sX sX> <"*" sY sY>>;
```

4.2.LOCAL VARIABLES

Consider a function Sq-Sub1 that decreases the argument by one and squares the number obtained:

```
Sq-Sub1(X) = (X-1)*(X-1)
```

This function can be defined in Refal in the following way:

```
$func Sq-Sub1 sX = sZ;

Sq-Sub1  sX = <"*" <"-" sX 1> <"-" sX 1>>;
```

An obvious deficiency of this definition is that it involves duplicate calculations: the expression <"-" sX 1> is to be evaluated twice. But this can be avoided by introducing an auxiliary function Sq:

```
$func Sq-Sub1 sX = sZ;
$func Sq      sY = sZ;

Sq-Sub1  sX = <Sq <"-" sX 1>>;
Sq       sY = <"*" sY sY>;
```

The function Sq serves the only purpose: it waits for the argument to be decremented by one, catches the result obtained, and continues the computation. It is obvious that superfluous auxiliary functions can make the program obscure and difficult to understand, for which reason Refal Plus enables us to introduce local variables for denoting intermediate values. This can be achieved by means of bindings, which are paths of the form

```
S :: He R
```

where S is a source, R is a rest, and He is a so called "hard expression". The hard expression He, which consists of symbols, brackets, and variables, must satisfy the following restrictions. First, He must not contain two occurrences of the same

variable. Second, each subexpression of He can contain no more than one ve-variable at the top level.

It can be easily seen that, being a hard expression, He can be regarded as a format expression, and the Refal Plus compiler verifies that S is certain to return ground expressions satisfying the format He.

The path S :: He R is evaluated as follows. First, the source S is evaluated. If the result returned is a ground expression Ge, the variables in He are bound to the corresponding subexpressions of Ge. Then the rest R is evaluated, and the result returned is taken to be the result of the whole construct.

Now the definition of Sq-Sub1 can be rewritten in the following way:

```
$func Sq-Sub1 sX = sZ;

Sq-Sub1  sX =
  <"-" sX 1> :: sY,
    <"*" sY sY>;
```

It should be noted that the evaluation of the path S :: He R begins by evaluating the source S in the environment in which the whole construct is evaluated. Then the variables in He are bound, and the environment is extended with the new bindings, so that the rest R is evaluated in the extended environment. Thus the evaluation of the path

```
100 :: sX, <"+" sX 1> :: sX = sX
```

returns 101.

The hard expression He in a path S :: He R may be empty, in which case the path takes the form S :: R and can be abbreviated to S R. This construct (called condition) is usually used in cases where we are interested in the side effects produced by evaluating S, rather than in the result returned by S. For example, evaluating the path

```
<Println "A">, <Println "B">, <Println "C"> =
```

causes three lines to be printed, the first line consisting of the character A, the second of the character B, and the third of the character C.

The rest R in a path S :: He R may consist of a single comma, in which case the path takes the form S :: He , and can be abbreviated to S :: He.

4.3.RECURSION

A function definition may contain calls to library functions as well as calls to functions defined in the program. In particular, a function may call itself (either directly or

through other functions), in which case the function definition is said to be recursive.

A function may have to be defined recursively if the set of arguments for which the function is defined is infinite, and there is no limitation on the size of the arguments.

Let us consider, for example, the following problem. Suppose we have to define a function Reverse that "reverses" a ground expression by rearranging its top-level terms in reverse order. Thus, if the argument has the form

```
Gt1 Gt2 ... Gtn
```

where Gt1, Gt2, ..., Gtn are ground terms, then the function is to return the ground expression

```
Gtn ... Gt2 Gt1
```

If the length of the argument expression were limited, for example, if we knew that n<=3, we could consider four separate cases to produce the following function definition

```
$func Reverse e.Exp = e.Exp;

Reverse
  {
  = ;
  t1 = t1;
  t1 t2 = t2 t1;
  t1 t2 t3 = t3 t2 t1;
  };
```

There is no limit on the length of the input expressions, however. Thus, the function definition has to consider an infinite number of cases, which seems to imply that the program has to be infinite in size.

This difficulty, however, can be circumvented by means of recursion. We can reason in the following way. Let us consider an argument expression

```
Gt1 Gt2 ... Gtn
```

If n=0, then the result to be returned is the empty expression. Otherwise, if n>=1, the problem can be reduced to a less difficult one. Namely, by discarding the first term in the argument expression we get the expression

```
Gt2 ... Gtn
```

which is n-1 terms in length. By reversing this expression we get

```
Gtn ... Gt2
```

Now, by adding Gt1 to the end of the expression, we get the desired result

```
Gtn ... Gt2 Gt1
```

Reasoning in this way, we come to the following recursive definition of the function Reverse:

```
Reverse
  {
  = ;
  t.X e.Rest = <Reverse e.Rest> t.X;
  };
```

It is interesting that there exists another solution to the problem of the expression reversion, which is in no way worse than the above. Namely, the problem can be reduced to a less difficult one by discarding the last term, rather than the first one, in which case we get the following solution:

```
Reverse
  {
  = ;
  e.Rest t.X = t.X <Reverse e.Rest>;
  };
```

It can be easily seen that the essence of the solution consists in dividing the original expression Ge into two smaller non-empty expressions Ge1 and Ge2 such that

```
Ge = Ge1 Ge2
```

Now, each of the expressions Ge1 and Ge2 can be reversed separately. Let the corresponding expressions obtained be Ge1' and Ge2'. Then the expression

```
Ge2' Ge1'
```

is obviously the result of reversing the original expression Ge.

If Refal Plus is implemented for a multi-processor computer in such a way that the reversion of Ge1 and Ge2 can be performed simultaneously, it is advantageous to make Ge1 and Ge2 approximately equal in length. In this way we get the following modification of the above function definition, in which there are calls to library functions from the modules ACCESS and ARITHM.

```
$func Reverse e.Exp = e.Exp;

Reverse
  {
  = ;
  t1 = t1;
  eX,
```

```
    <Length e.X> :: sLen,
    <Div sLen 2> :: sDiv,
    = <Reverse <Middle sDiv 0 eX>>
      <Reverse <Left   0 sDiv eX>>;
  };
```

5.FAILURES AND ERRORS

5.1.FAILURES PRODUCED BY EVALUATING RESULT EXPRESSIONS AND PATHS

The evaluation of a path Q has hitherto been assumed to return a ground expression Ge. It can, however, also result in failure or error.

In case of failure, the result returned is a special value "failure", rather that a ground expression.

The simplest way of producing a failure is to evaluate the rest of the form

```
$fail
```

In case of error, the result returned is a special error value, rather than a ground expression. This value has the form $error(Ge), where the ground expression Ge is an error message. The error message usually begins with a word symbol, the symbol being the name of the function of which the evaluation has caused the error. For example, an attempt at evaluating the function call

```
<DIV 10 0>
```

causes the error "divide by zero", which results in returning the value

```
$error(DIV "Divide by zero")
```

The values of the form $error(Ge) possess the following property. Suppose that a construct is to be evaluated, and the evaluation of a constituent part of the construct results in returning $error(Ge). Then, the evaluation of the construct terminates, the result returned being $error(Ge). The only exception to this rule is the construct $trap specifically designed for "trapping" errors.

In order for the informal language description to be concise, the detailed consideration of the subtle points involved in dealing with errors will be postponed until Chapters II and III.

5.2.MATCHES

Consider the following problem. Let Ge be a ground expression known to contain no less than two character symbols '+' at

the top level, and we want this expression Ge to be divided into three parts GeX, GeA, and GeY, such that Ge = GeX '+' GeA '+' GeY, with GeX and GeY not containing '+' at the top level. Let us give the function performing this task the name of "++". Then, for example,

```
<"++" 'AAA+BBB+CCC+DDD+EEE'> =>
     ('AAA')('BBB+CCC+DDD')('EEE')
```

Thus, it is necessary to find the leftmost '+' in Ge as well as the rightmost one. The leftmost '+' can be easily found by matching Ge against the pattern $l eX '+' eP, whereas the rightmost '+' can be found by matching Ge against the pattern $r eQ '+' eY. Any pattern enables the matching to be done either from left to right or from right to left, there being no way of combining the two directions of matching in a single pattern, for which reason we have to perform the analysis of the expression in two steps. This can be expressed in the following way:

```
$func "++"    eZ      = (eX)(eA)(eY);
$func "++Aux" (eX)(eP) = (eX)(eA)(eY);

"++"     $l eX '+' eP = <"++Aux" (eX)(eP)>;
"++Aux"  $r (eX)(eA '+' eY) = (eX)(eA)(eY);
```

Thus, to break up the analysis of the expression into two stages, we have had to introduce an auxiliary function. This could have been avoided, however, by making use of the construct "match".

A match is a path of the form

```
S : Snt
```

where S is a source, and Snt a sentence of the form P R, the sentence Snt consisting of the pattern P and the rest R.

If the rest R consists of a single comma, it may be omitted, in which case the match S : P , takes the form S : P .

The evaluation of a match S : P R proceeds as follows. First, the source S is evaluated. If the value returned is a failure, the result of evaluating the match is a failure. Otherwise, if the value returned is a ground expression Ge, Ge is matched against the pattern P, and consideration is given to the variants of matching satisfying the following additional restriction: if in the environment in which the whole match is evaluated some variables are bound to values, these variables must be given the same values in the environment produced by matching.

For example, suppose the variable sX has been given the value 1. Then matching the ground expression 1 2 1 2 against the pattern eA sX eB results in producing only two variants of matching

```
{eA = , sX = 1, eB = 2 1 2}
```

```
{eA = 1 2, sX = 1, eB = 2}
```

whereas, if there were no restriction of the value of sX, there would be four variants.

Now, let Env1, Env2, ..., Envn be all the variants of matching thus obtained listed according to the order relation introduced above on the set of the variants of matching. Then an attempt is made to evaluate R in the environment Env1. If the value returned is a ground expression Ge, this expression is taken to be the result of the whole match. Otherwise, if the value returned is a failure, this failure is "caught", i.e. the first variant of matching is discarded, and the same attempt is made to evaluate R for all remaining variants of matching.

If, for all the variants of matching, evaluating the rest R results in a failure, the result of the whole match is a failure.

For example, evaluating the path

```
'ABC' : $r e1 sX e2, <Print sX> $fail
```

results in the character sequence 'CBA' being printed, and a failure being returned as the result.

Now we can give a definition of the function "++" without introducing an auxiliary function "++Aux":

```
$func "++"  eZ = (eX)(eA)(eY);

"++"
  $l eX '+' eP,
    eP : $r eA '+' eY
      = (eX)(eA)(eY);
```

5.3.FAILURE TRAPPING

A programs written in Refal Plus can determine whether the evaluation of a path has resulted in returning a failure, the result of the test being used to control the execution of the program.

A negative condition is a path of the form

```
# S R
```

where S is a source, and R a rest. If the rest R consists of a single comma, it may be omitted, in which case the negative condition takes the form # S .

Syntactically, a negative condition is a rest.

The evaluation of a negative condition proceeds as follows. The source S is evaluated. If the value returned is an empty ground expression, the result of evaluating the whole construct is a failure. Otherwise, if the value returned is a failure, the rest R is evaluated to produce the result of the whole construct.

An alternative is a path of the form

```
\{ Q1; Q2; ... Qn; }
```

where Q1, Q2, ..., Qn are paths. Syntactically, an alternative is a source.

The evaluation of an alternative proceeds as follows. The path Q1 is evaluated. If the value returned is a ground expression Ge, this expression is taken to be the result of the whole alternative. Otherwise, if the result of evaluating Q1 is a failure, an attempt is made to evaluate \{Q2; ..., Qn;}, and the value returned is taken to be the result of the whole construct. Thus, alternatives can be used for "catching" failures.

If the evaluation of all the paths Q1, Q2, ..., Qn results in returning failures, the result of evaluating the whole construct is taken to be a failure.

For example, consider the path

```
1 :: sX,
\{ sX : 0 = 1; sX : 1 = 0; }
```

which is evaluated in the following way. First, the evaluation of the binding 1 :: sX results in the variable sX being given the value 1. Then an attempt is made to evaluate the first path of the alternative, i.e. sX : 0 = 1 . Matching 0 against sX fails, and, consequently, so does the evaluation of the first path. Thus, an attempt is made to evaluate the second path sX : 1 = 0 , which results in returning 0, and this value is taken to be the result of the whole alternative.

In some cases, however, there may prove to be useful another variety of alternatives, which has the form

```
{ Q1; Q2; ... Qn; }
```

The difference between the two kinds of alternatives emerges in cases where the evaluation of all the paths results in returning failures. On such occasions, instead of returning a failure, the latter form of alternatives returns an error $error(Fname "Unexpected fail"), where Fname is the name of the function in which the alternative appears.

Thus, the pair of brackets \{ ... } can be regarded as "transparent" for failures, whereas the pair { ... } can be regarded as "opaque", the reason being that a failure is incapable of "jumping" out of an alternative { Q1; Q2; ... Qn; }.

Programs written in Refal Plus often contain paths of the form

```
S : Ve, \{Ve : Snt1; Ve : Snt2; ..., Ve : Sntn;}
S : Ve,  {Ve : Snt1; Ve : Snt2; ..., Ve : Sntn;}
```

where Ve is an e-variable that does not appear in other places of the function definition, and each sentence Sntj has the form Pj Rj. Such paths can, correspondingly, be abbreviated to alter-

native matches

```
S : \{Snt1; Snt2; ... Sntn;}
S :  {Snt1; Snt2; ... Sntn;}
```

where the constructs \{Snt1; Snt2; ... Sntn;} and {Snt1; Snt2; ... Sntn;} are called pattern alternatives.

If a rest Rj consists of a single comma, it can be omitted, in which case the corresponding sentence Pj Rj takes the form Pj.

For example, the alternative { sX : 0 = 1; sX : 1 = 0; } can be abbreviated to the alternative match sX : { 0 = 1; 1 = 0; }, and the alternative \{ sX : A,; sX : B,; } to sX : \{ A; B; }.

Syntactically, all alternative matches are sources, which, for example, enables us to write the paths of the form

```
sX : { 0 = 1; 1 = 0; } :: sY = <"+" sX sY>
```

The evaluation of this paths begins by evaluating the source sX : {0 = 1; 1 = 0;}. Then the variable sY is bound to the value returned, and the path = <"+" sX sY> is evaluated in the extended environment.

5.4. CONTROL OVER FAILURE TRAPPING

As we have seen, Refal Plus enables failures to be caught, providing as it does a fairly rich collection of failure trapping constructs. Sometimes, however, we want to produce so powerful a failure as to overcome all the traps waiting for it (or, at least, some of them). This can be achieved by means of fences and cuts.

A fence is a rest of the form \? Q , whereas a cut is a rest of the form \! Q .

Fences and cuts appearing in a program serve as marks controlling the propagation of failures. Each cut \! Q is required to be enclosed in a fence of the form \? ... \! Q ... . The evaluation of a cut \! Q proceeds as follows. An attempt is made to evaluate the path Q. If this evaluation terminates and results in returning a value X, this X is taken to be the result of the whole construct \? ... \! Q ... .

In particular, if X is a failure, so is the result of evaluating the whole construct \? ... \! Q ... .

The following example illustrates the use of fences and cuts. Consider the evaluation of the path

```
eA : e1 '+' e2 '-' e3
  = (e1)(e2)(e3)
```

if the value of the variable eA contains '+' followed by '-' at the top level, the matching of this value against the pattern results in finding the leftmost '+' followed by the nearest '-'.

Now consider the case where eA contains '+' at the top level, but there is no '-' at the top level. Then the leftmost '+' is found, and the rest of the expression examined in order to find a '-'. Since the search for a '-' fails, the value of the variable e1 is extended, and the search for a '-' repeated. This needn't be done, however, because, after the failure of the first search for a '-', the second search is bound to fail as well.

The constructs \? and \! enable us to avoid the above unnecessary search. To achieve this, we begin by rewriting the original match in the following way:

```
eA : e1 '+' eX,
  eX : e2 '-' e3
    = (e1)(e2)(e3)
```

Now, if matching the value of eX against the pattern e2 '-' e3 fails, an attempt is made to find the next variant of matching the value of eA against e1 '+' eX. This, however, can be avoided by inserting \? and \! in the following way:

```
\? eA : e1 '+' eX
  \! eX : e2 '-' e3
    = (e1)(e2)(e3)
```

Now, if the inner match returns a failure, this failure is returned as the result of the whole path.

5.5.MEANING OF RIGHT HAND SIDES

A right hand side, which has the form = Q , where Q is a path, is an even more powerful means of restricting the search than fences and cuts.

To explain the meaning of the right hand sides, we have to introduce a few additional concepts.

Suppose that a construct appears as a component in a larger construct, and, according to the semantics of Refal Plus, the result of evaluating the inner construct is taken to be the result of evaluating the surrounding construct. Then the inner construct is said to be a vassal of the surrounding construct. For example, the rest R in a path S :: He R is a vassal, since the result of evaluating R is taken to be the result of the whole path.

A construct that is not a vassal of the surrounding constructs is said to be a sovereign. For example, the source S in a path S :: He R is a sovereign, since its result, in general, is not the result of the whole construct (despite the fact that this result may be used in evaluating R).

More specifically, if a function definition has the form Fname Palt , then the pattern alternative Palt is a sovereign.

If a path has either of the forms: S R , S :: He R , S : P R or # S R , then the source S is a sovereign.

If a source has the form S : Palt , then the source S is a sovereign.

Now, let us consider a construct along with all the sovereigns surrounding this construct. The smallest of the sovereigns, included in all the others, is said to be the patron of the construct in question. In particular, if a construct is a sovereign, the construct's patron is the construct itself.

Now we are able to describe the semantics of right hand sides.

Suppose that the patron of a right hand side = Q is a surrounding construct ... = Q ... . Then, if the evaluation of the path Q results in returning a value X, this value X is taken to be the result of the whole patron ... = Q ... .

In particular, if X is a failure, the result of evaluating the patron ... = Q ... is a failure, in spite of the fact that there may be failure traps in the patron.

For example, the evaluation of the path

```
\{ A B C : $l e sX e, sX : B, sX} :: sY, sY
```

proceeds as follows. First, sX is bound to the value A, and an attempt to match this value against the symbol B fails. Then sX is bound to the new value B, and the evaluation succeeds, the result returned being the symbol B. On the other hand, if we replace the comma with the equality sign, we get the path

```
\{ A B C : $l e sX e = sX : B, sX} :: sY, sY
```

the evaluation of which fails.

Some restrictions are imposed on the use of fences, cuts, and right hand sides.

If a cut \! Q is enclosed in a fence \? ... \! Q ... , both constructs must have the same patron.

If a cut \! Q is enclosed in a fence \? ... \! Q ... , there must be no right hand side = ...\! Q ... surrounding the cut \! Q , but enclosed in the fence \? ... \! Q ... .

5.6.FAILING AND UNFAILING FUNCTIONS

All functions defined and called in Refal Plus programs are classified as either failing or unfailing.

If a function Fname is an unfailing one, then the evaluation of a call <Fname Re> cannot result in returning a failure. On the other hand, if a function Fname is a failing one, then the evaluation of a call <Fname Re> can, in general, result in returning a failure.

Function declarations have hitherto been assumed to have the form

```
$func  Fname Fin = Fout;
```

which is correct only in cases where Fname is an unfailing func-

tion. Otherwise, if Fname is a failing function, its declaration must have the form

```
$func? Fname Fin = Fout;
```

Now the semantics of function definitions can be given a more accurate description. Let the definition of a function Fname have the form

```
Fname Palt
```

where Palt is a pattern alternative whose form is either \{P1 R1; P2 R2; ... Pn Rn;} or {P1 R1; P2 R2; ... Pn Rn;}. Then the evaluation of a call <Fname Re> proceeds as follows. The result expression Re is evaluated. If the value returned is a failure, the result of evaluating <Fname Re> is taken to be a failure, without actually calling the function Fname. Otherwise, if the value returned is a ground expression Ge, the function Fname is called, i.e. the source

```
Ge : Palt
```

is evaluated in the empty environment (in which no variable is bound to a value). Suppose the evaluation of the above source results in returning a value X. Then there are a few cases to be considered.

If X is a ground expression, X is taken to be the result of evaluating the call <Fname Re>. Otherwise, if X is a failure, the following depends on the Function Fname being a failing one.

If Fname is a failing function, and X is a failure, the result of evaluating the call <Fname Re> is a failure.

If Fname is an unfailing function, and X is a failure, this failure is "caught" and transformed into the error $error(Fname "Unexpected fail"), which is taken to be the result of evaluating the call <Fname Re>.

6.LOGICAL CONDITIONS

6.1.CONDITIONS AND PREDICATES

In some cases, the program has to test some conditions in order to select one of the alternative courses of action.

The exact way in which conditions can be written and tested depends on the programming language. As far as Refal Plus is concerned, we use the following terminology.

A path Q is said to be a condition, if the value returned by the path is always either an empty ground expression or a failure. If the result is an empty expression, the condition is considered to be satisfied, otherwise, if the result is a failure, the condition is considered not to be satisfied.

Thus empty expressions and failures may be considered as corresponding to the well-known truth values "true" and "false".

It should be kept in mind, however, that the evaluation of a condition Q may non-terminate or produce an error, in which case we consider either the program or the input data to be incorrect.

Some of the library functions are specifically designed for testing conditions. Such functions are referred to as predicates. In Refal Plus a predicate returns either an empty expression (if its arguments satisfy the condition) or a failure (if the condition is not satisfied). For example, the function "<" tests whether the first argument is less than the second one. In other words, let Ge1 and Ge2 be ground expressions. Then if Ge1 is "less" than Ge2, the result of evaluating <"<" (Ge1)(Ge2)> is an empty expressions, otherwise the result is a failure.

If a program defines a predicate function, the declaration of the function must have the form

```
$func? Fname Fin = ;
```

Now we consider several ways of using and combining conditions.

6.2.CONDITIONALS

Suppose we have a condition represented by a source S and two paths Q' and Q". Consider the path

```
\? {S \! Q'; \! Q";}
```

If the result of evaluating S is an empty expression, the path Q' is evaluated and the value returned is taken to be the result of the whole construct. Otherwise, if the result of evaluating S is a failure, the path Q" is evaluated and the value returned is taken to be the result of the whole construct.

Notice should be taken of the use of cuts \! . They prove to be essential in cases where the evaluation of Q' or Q" fails. Let us try removing the cuts, and consider the path thus obtained:

```
{ S, Q'; Q";}
```

Now, if the condition S is satisfied, the path Q' is evaluated. Suppose the evaluation of Q' fails. Then, instead of being returned as the result of the whole construct, the failure is caught, which causes the evaluation of the path Q". But this, certainly, was not our intention! Thus the first cut is necessary to prevent the control from "jumping" to the next path in the alternative.

Now, let us consider the case where the condition is not satisfied, i.e. the evaluation of S fails. Then the failure is caught, which causes the evaluation of the path Q". Suppose that the evaluation of Q" fails. Then the failure is caught and an attempt is made to evaluate the next path in the alternative.

But there is no such path! Hence, an error is generated, which, again, was not our intention!

Nevertheless, in some cases, the cuts can be omitted. Thus an alternative of the form

```
\? {S \! = Q'; \! = Q";}
```

can always be, and usually is, rewritten as

```
{ S = Q'; = Q";}
```

As an example let us consider the function Min-Ge, which takes two ground expressions Ge1 and Ge2 as arguments, and returns either Ge1 or Ge2. Namely, if Ge1 precedes Ge2, the result is Ge1, otherwise the result is Ge2.

```
$func Min-Ge (eX)(eY) = e.Min-X-Y;

Min-Ge  (eX)(eY) =
  {
  <"<" (eX)(eY)>
    = eX;
    = eY;
  };
```

Now consider the case where a condition is represented by a path Q, and a path Q' must be evaluated if the condition is not satisfied, whereas a path Q" must be evaluated if the condition is not satisfied. This case can be reduced to the above by enclosing the condition Q in curly braces thereby making the path Q into the source \{ Q; } . Now the conditional can be written as follows:

```
\? { \{Q;} \! Q'; \! Q";}
```

6.3.LOGICAL CONNECTIVES

Sometimes we have to test complicated logical conditions. Complex conditions can often be expressed in terms of more elementary conditions by means of the logical connectives "AND", "OR", and "NOT". Although Refal Plus does not provide logical connectives explicitly, they can be easily represented by other constructs.

*** Logical "AND"

Suppose we have two conditions and must determine whether both of them are satisfied.

If both conditions are represented by paths Q' and Q", the compound condition can be tested by evaluating the path

\{ Q';}, Q"

If the first condition is represented by a source S, and the second by a path Q, the compound condition can be tested by evaluating the path S,░Q.

And, finally, if both conditions are represented by result expressions Re' and Re", the compound condition can be tested by evaluating the result expression Re' Re".

*** Logical "OR"

Suppose we have two conditions and must determine whether one (or both) of them are satisfied.

If both conditions are represented by paths Q' and Q", the compound condition can be tested by evaluating the path

\{ Q'; Q"; }

*** Logical "NOT"

Suppose we have a condition represented by a path Q, and must determine whether the condition is not satisfied. This can be done by evaluating the path

\{Q;}

which is an abbreviation to the path # \{Q;}, .

In cases where the condition is represented by a source S, the negated condition can be tested by evaluating the path

S

which is an abbreviation to the path # S , .

In both cases we take the opportunity of omitting the rests consisting of a single comma.

6.4.EXAMPLE: FORMAL DIFFERENTIATION

Suppose we want to define a function that, given an algebraic expression and a variable, will produce the derivative of the expression with respect to the variable [Hen 80]. To keep the presentation concise, we deal only with simple formulae consisting of integers, variables, and binary operators + and *. The generalization to more complicated formulae is straightforward, and is left for the reader as an exercise.

Let x and y stand for arbitrary variables, i for an integer, and e for a formula. Let Dx(e) denote the result of differentiating e with respect to x. Then the rules of differentiation can be written as follows:

```
Dx(x)     =   1
Dx(y)     =   0         (where y is different from x)
```

```
Dx(i)              =     0
Dx(e1 + e2)        =     Dx(e1) + Dx(e2)
Dx(e1 * e2)        =     e1 * Dx(e2) + e2 * Dx(e1)
```

Before writing the program of differentiating, we have to represent formulae by ground expressions. Let [e] stand for the formula e represented by a ground expression. Then we may choose the representation defined by the following rules:

```
[x]                =     x
[i]                =     i
[e1 + e2]          =     (Sum  [e1] [e2])
[e1 * e2]          =     (Prod [e1] [e2])
```

Now a function Diff can be easily defined whose first argument is a variable, and the second argument a formula. The function returns the result of differentiating the formula with respect to the variable.

```
$func Diff sX tE = tE;

Diff  sX tE =
  tE :
  {
  sX = 1;
  sY = 0;
  (s.Oper t.E1 t.E2) =
    <Diff sX tE1> :: t.DxE1,
    <Diff sX tE2> :: t.DxE2,
    s.Oper :
    {
    Sum   = (Sum t.DxE1 t.DxE2);
    Prod  = (Sum (Prod t.E1 t.DxE2) (Prod t.E2 t.DxE1));
    };
  };
```

An obvious deficiency of the above definition of the function Diff is that the formulae produced by the function contain a lot of unnecessary parts. For example, according to the above rules of differentiation we have

```
DX(3*(X*X)+5)  =   (3*((X*1)+(X*))+(X*X)*0)+0
```

which could have been reduced to

```
3*(X+X)
```

by means of evident simplifications. Thus we can enhance the definition of the function Diff by making the function perform the following reductions:

```
0 + e2      ==>    e2
e1+ 0       ==>    e1
```

```
0 * e2      ==>   0
e1* 0       ==>   0
1 * e2      ==>   e2
e1* 1       ==>   e1
```

(We won't consider more complicated reductions, to keep the presentation concise.)

There are two ways of implementing the above simplifications. The first way is to perform the simplifications only after the result of the differentiation has been completely built. The second way is to try the simplifications "on the fly", during the differentiation. And it is the second way that we are going to implement.

As the first step, we define two functions Sum and Prod, each function taking two formulae and returning respectively the sum and the product of the formulae. It is in these functions that the simplifications are performed.

```
$func Sum  t1 t2 = t;
$func Prod t1 t2 = t;

Sum
  {
  0  t2 = t2;
  t1 0  = t1;
  t1 t2 = (Sum t1 t2);
  };

Prod
  {
  0  t2 = 0;
  1  t2 = t2;
  t1 0  = 0;
  t1 1  = t1;
  t1 t2 = (Prod t1 t2);
  };
```

Now we can rewrite the above definition of the function Diff, inserting at appropriate places the calls to the functions Sum and Prod:

```
Diff  sX tE =
  tE :
  {
  sX = 1;
  sY = 0;
  (s.Oper t.E1 t.E2) =
    <Diff sX tE1> :: t.DxE1,
    <Diff sX tE2> :: t.DxE2,
    s.Oper :
    {
    Sum   = <Sum t.DxE1 t.DxE2>;
    Prod  = <Sum <Prod t.E1 t.DxE2> <Prod t.E2 t.DxE1>>;
```

```
        };
      };
```

6.5.EXAMPLE: COMPARISON OF SETS

The following example illustrates the use of recursion along with logical connectives.

According to the set theory, two sets are considered to be equal, if they contain the same elements. Suppose we want to define a Refal Plus function testing two sets for equality. The first thing we have to invent is the representation of sets by ground expressions. First, let us consider the sets whose elements may be Refal symbols only. A set of symbols {Gs1, Gs2, ..., Gsn} can, obviously, be represented by the ground expression

```
Gs1 Gs2 ... Gsn
```

A feature of this representation is that any non-empty set of symbols has lots of different representations. For example, the set {John, Mary} may be represented as John Mary or Mary John , or even Mary John John Mary . Thus, different representations may correspond to equal sets.

It is well known that an element of a set can be a set itself. So, we must be able to represent sets containing symbols as well as sets, which may contain sets, etc. How shell we represent set elements that are sets?

A simple solution is the following. If an element of a set is a symbol Gs, the element is represented by the symbol Gs. Otherwise, if an element of a set is a set X, the element is represented by the ground term (X'), where X' is a representation of X. For example, the set {A, {A,B}, {A}} may be represented by the ground expression A (A B) (A) .

Now we define the predicate function Eqset? determining whether its two arguments represent the same set. This function performs the test for equality by reducing it to several simpler tests.

Namely, two sets A and B are equal iff A is a subset of B and B is a subset of A. Further, a set A is a subset of a set B iff each element X of A belongs to B.

Thus, instead of defining a single function, we have to define four mutually recursive predicate functions. Eqset? determines whether its two arguments are representations of the same set. Subset? determines whether the set represented by the first argument is a subset of the set represented by the second argument. El? determines whether the first argument represents a set belonging to the set represented by the second argument. And, finally, Eqel? determines whether its two arguments represent the same element of a set.

Note that, to test for equality two set elements that are sets themselves, we have to test for equality the corresponding sets, for which reason the function Eqel? has to call the func-

tion Eqset?. Thus, finally, Eqset? turns out to be defined in terms of itself.

```
$func? Eqset?  (eA)(eB) = ;
$func? Subset? (eA)(eB) = ;
$func? El?     tX  (eA) = ;
$func? Eqel?   tX  tY   =;

Eqset?  (eA)(eB) =
  <Subset? (eA)(eB)><Subset? (eB)(eA)>;

Subset?  (eA)(eB) =
  eA :
  {
  = ;
  tX eR = <El? tX (eB)><Subset? (eR)(eB)>;
  };

El?  tX (eA) =
  eA : tY eR,
  \{ <Eqel? tX tY>; <El? tX (eR)>; };

Eqel?  tX tY =
  \{
  tX tY : s s
    = tX : tY;
  tX tY : (eA)(eB)
    = <Eqset? (eA)(eB)>;
  };
```

7.DIRECT ACCESS SELECTORS

A typical case where the direct access to ground expressions turns out to be useful is the implementation of the algorithms based on the technique known as "divide and conquer". The general idea is to solve a problem by dividing it into subproblems - each an instance of the original problem but on inputs of smaller size - in such a way that the solution of the original problem can be assembled from the solutions to the subproblems. The principle "divide and conquer" is usually applied together with the principle of "balancing" requiring that the original problem should be divided into subproblems of roughly equal size [AHU 74].

A classic application of the principle "divide and conquer" is the problem of sorting (i.e. arranging in ascending order).

One of the sorting methods is the merge sort [AHU 74]. The idea is to divide the original set S into two disjoint sets S1 and S2 of roughly equal size, sort S1 and S2 to produce two ordered sequences Q1 and Q2, and then merge Q1 and Q2 into one ordered sequence Q, thereby obtaining the solution to the original problem.

Now let us define the function MSort, which takes an inte-

ger sequence as argument, divides it into two parts of approximately equal size, and calls itself recursively in order to sort both parts. Then the sequences thus obtained are merged by the function Merge to produce the final result.

```
$func MSort eS = eS;
$func Merge (eX)(eY) = eZ;

MSort  eS =
  <Length eS> :: sLen,
  {
  <"<=" (sLen) (1)>
    = eS;
    = <Div sLen 2> :: sK,
      <Left 0 sK eS> :: eS1,
      <Middle sK 0 eS> :: eS2,
        <Merge ( <MSort eS1> )( <MSort eS2> )>;
  };
```

How we have to define the function Merge, which takes two ordered integer sequences as arguments and merges them into one ordered sequence.

```
Merge  (eX)(eY) =
  {
  eX :
    = eY;
  eY :
    = eX;
  (eX)(eY)  : (sA eXRest)(sB eYRest)
    = {
      <"<=" (sA)(sB)>
        = sA <Merge (eXRest)(eY)>;
        = sB <Merge (eX)(eYRest)>;
     };
  };
```

8.FUNCTIONS RETURNING SEVERAL RESULTS

8.1.GROUND EXPRESSION TRAVERSAL

The following examples illustrate the usefulness of functions returning several results.

Suppose we want to define a function NMB replacing all symbols appearing in a ground expression with their ordinal numbers. For example,

```
<NMB A (B A) C A>   =>   1 (2 3) 4 5
```

The main difficulty is that, having encountered a pair of parentheses, the function cannot know in advance the number of symbols enclosed in the parentheses. But this information will

be necessary for the function to resume the processing of the top level of the expression after the contents of the parentheses will be done away with. Therefore, the symbol numbering function must have two arguments: the expression to be processed and the number to be assigned to the first symbol in the expression (if any). This function must return two results: the expression processed and the first "unused" number. Thus we come to the following definition of the function NMB (making use of two auxiliary functions NMB-Exp and NMB-Term).

```
$func NMB      e.Exp    = e.Exp;
$func NMB-Exp  e.Exp sN = e.Exp sN;
$func NMB-Term t.Exp sN = t.Exp sN;

NMB  e.Exp =
  <NMB-Exp e.Exp 1> :: e.Exp s,
    e.Exp;

NMB-Exp  e.Exp sN =
  e.Exp :
  {
  = sN;
  tX e.Rest =
    <NMB-Term tX sN> :: tX sN,
    <NMB-Exp e.Rest sN> :: e.Rest sN,
      tX e.Rest sN;
  };

NMB-Term  tX sN =
  tX :
  {
  s =
    sN <"+" sN 1>;
  (eE) =
    <NMB-Exp eE sN> :: eE sN,
      (eE) sN;
  };
```

8.2.QUICKSORT

There is a second way we can apply the idea of divide and conquer to the problem of sorting, the so-called quicksort algorithm [AHU 74].

Suppose we have to sort a set of integers S. The idea is to choose X, an arbitrary element of S, and to divide S into three disjoint sets S1, S2, and S3, such that S1 contains integers that are less than X, S2 contains integers equal to X, and S3 contains integers that are greater that X. Then, by sorting S1, S2, and S3, we get three ordered sequences Q1, Q2, and Q3 (the sorting of Q2 is trivial, because all elements of Q2 are equal to X). Then we can concatenate Q1, Q2, and Q3 into the new sequence Q1 Q2 Q3, which gives us the solution to the original

problem.

Now we can define the function QSort, which sorts an integer sequence according to the above method. The auxiliary function Split is used for partitioning the input sequence into three subsequences.

```
$func QSort  eS = eQ;
$func Split  sX eS = (eS1)(eS2)(eS3);
$func Split-Aux  sX (eS1)(eS2)(eS3) eS = (eS1)(eS2)(eS3);

QSort  eS =
  {
  eS :
    = ;
  eS : t
    = eS;
  eS : sX e
    = <Split sX eS> :: (eS1)(eS2)(eS3),
        <QSort eS1> eS2 <QSort eS3>;
  };

Split  sX eS =
    <Split-Aux sX ()()() eS>;

Split-Aux  sX (eS1)(eS2)(eS3) eS =
  eS :
  {
  =
    (eS1)(eS2)(eS3);
  sY eRest =
    {
    <"<" (sY)(sX)>
      = <Split-Aux sX (eS1 sY)(eS2)(eS3) eRest>;
    <">" (sY)(sX)>
      = <Split-Aux sX (eS1)(eS2)(eS3 sY) eRest>;
      = <Split-Aux sX (eS1)(eS2 sY)(eS3) eRest>;
    };
  };
```

9.ITERATION

In Refal Plus, recursion is the principal means of representing loops. In many cases, however, this means is too universal, for which reason Refal Plus provides a special search construct, which, syntactically, is a path of the form

```
S" $iter S' :: He R
```

where the sources S" and S' are sovereigns, and the rest R a vassal (which is essential in cases where S", S', or R contain right hand sides of the form = Q).

If the hard expression He is empty, it may be omitted along

with the keyword "::". If the rest R consists of a single comma, it may also be omitted.

A search construct introduces new local variables (in the same way as a binding S :: He R does). The initial values of these variables are obtained by evaluating the source S". Then an attempt is made to evaluate the rest R. If the evaluation of R succeeds, the value returned is taken to be the result of the whole construct. Otherwise, if the evaluation of R fails, the local variables are bound to new values (obtained by evaluating the source S' in the old environment associating the local variables with their old values). Then, again, an attempt is made to evaluate the rest R, etc.

Thus, in a sense, the search construct tries to find for the variables in He such values that the evaluation of the rest R succeeds.

The easiest way to explain the exact meaning of the search construct consists in defining it in terms of more elementary constructs, such as bindings and alternatives. Namely, a search S" $iter S' :: He R is equivalent to the path

```
S" :: He, \{ R; S' $iter S' :: He R; }
```

This path, again, contains a search construct, which, again, may be "unfolded". Thus we get

```
S" :: He, \{ R;
     S' :: He, \{ R;
          S' $iter S' :: He R;
};}
```

By repeating the unfolding infinitely many times, we can transform the original construct into the infinite path

```
S" :: He, \{ R;
     S' :: He, \{ R;
          S' :: He, \{ R;
 ...
 ... };};}
```

The following example illustrates the use of the search construct.

Let us consider the well-known factorial function, which is usually given the following recursive definition:

```
$func Fact sN = sFact;

Fact
  {
  0 = 1;
  sN = <"*" sN <Fact <"-" sN 1>>>;
  };
```

The drawback of the above definition is that the call to

the function "*" cannot be evaluated until the evaluation of the internal call to the function Fact has terminated. Thus, the calls to "*" accumulate. However, the function Fact can be given a more "iterative" definition (making use of the auxiliary function Fact-Aux).

```
$func Fact sN = sFact;
$func Fact-Aux sR sK = sFact;

Fact  sN =
  <Fact-Aux 1 sN>;

Fact-Aux  sR sK =
  {
  sK : 0
    = sR;
    = <Fact-Aux <"*" sR sK> <"-" sK 1>>;
  };
```

The same can be expressed with the search construct in the following way:

```
$func Fact sN = sFact;

Fact  sN =
  1 sN
    $iter <"*" sR sK> <"-" sK 1>
      :: sR sK,
  sK : 0,
    = sR;
```

10. SEARCH AND BACKTRACKING

10.1. THE QUEENS PROBLEM

Our next example is the classic Eight Queens Problem [Hen 80]. Given a chessboard and eight queens, one must place the queens on the board so that no two queens hold each other in check; that is, no two queens may lie in the same row, column, or diagonal.

We shall consider a slightly more general problem of placing n queens on the board of the size nxn.

Let the rows and columns of the board be numbered from 1 to n. A chessboard square is said to have the coordinates (i,j), or, in other words, to be the square (i,j), if it lies in column i and row j.

Note that all squares lying in the same diagonal running upwards from left to right have the same sum of the column and row numbers, whereas all squares lying in the same diagonal running downwards from left to right have the same difference of the column and row numbers.

Thus two squares (i,j) and (i1,j1) lie in the same diago-

nal, if either i+j = i1+j1 or i-j = i1-j1. This condition is easy to check. Namely, if the evaluation of the path

```
\{
<"+" sI sJ> :: sN1, <"+" sI1 sJ1> :: sN2, sN1 : sN2;
<"-" sI sJ> :: sN1, <"-" sI1 sJ1> :: sN2, sN1 : sN2;
}
```

succeeds, the squares (i,j) and (i1,j1) lie in the same diagonal.

Now we need a way to represent a board containing queens in the first m columns.

It is obvious that we may confine our attention to the positions in which each column contains no more than one queen, because two queens lying in the same column would hold each other in check, thereby preventing the position from being a solution. On the other hand, the number of the queens to be placed is equal to the number of columns, implying that each column must contain exactly one queen. Hence, a position can be represented by a sequence of integers

```
I1 J2 ... In
```

where the number Ik represents the queen lying in column k and row Ik.

The solution will be constructed incrementally, by filling the columns one by one. Each time, a queen is placed in a column, it must be checked that no queen puts the new queen in check. Suppose the board contains k queens lying in the columns 1, 2, ..., k. This partially constructed position can be represented by the sequence of integers

```
I1 I2 ... Ik
```

where the number Im represents the queen lying in column m and row Im.

Now we can define the predicate Attack?, which returns an empty expression if the square (i,j) is attacked by the queens placed on the board, or a failure, if the square is not attacked.

```
$func? Attack?     sI sJ ePos = ;

Attack?  sI sJ ePos =
  ePos : $r eRest e, eRest : e sJ1,
  <Length eRest> :: sI1,
  \{
  sI1 : sI;
  sJ1 : sJ;
  <"+" sI sJ> :: sN1, <"+" sI1 sJ1> :: sN2, sN1 : sN2;
  <"-" sI sJ> :: sN1, <"-" sI1 sJ1> :: sN2, sN1 : sN2;
  };
```

It should be noted that the test i1=i could have been removed, since our program calls the function Attack? in such a way that the parameter i is guaranteed to be greater than the column numbers of the queens placed on the board.

Now we can define the function Next-Queen? making an attempt to add a new queen to a partially constructed position. Next-Queen? tries to place the new queen in different rows. If the queen can be placed, but this queen is not the last, an attempt is made to place the next queen, etc. If the current queen cannot be placed, the program "backtracks": i.e. tries to change the position of the previous queen.

```
$func? Next-Queen? sI sN ePos = ePos;

Next-Queen?  sI sN ePos =
  1 $iter \{ <"<" (sJ) (sN)> = <"+" sJ 1>; }
    :: sJ,
  # <Attack? sI sJ ePos>,
  ePos sJ :: ePos,
  \? {
  sI : sN
    \!  ePos;
    \!  <Next-Queen? <"+" sI 1> sN ePos>;
  };
```

There are some subtle points in the definition of the function Next-Queen? deserving special attention.

First, the search construct tries to evaluate its rest, sequentially binding the variable j to the values 1, 2, ..., n, and incrementing j by 1 after each failure to evaluate the rest of the construct.

Second, the evaluation of the rest of the search construct may fail for two reasons: either the square (i,j) is attacked by the queens already placed on the board, in which case the evaluation of the call to the function Attack? succeeds, and, therefore, the negation of this call fails, or, despite the fact that the current queen can be placed on the square (i,j), the following queens cannot be placed on the board, and, therefore, the recursive call to the function Next-Queen? fails.

Finally, we can define the function Solution?, which takes the size of the board as argument and returns either a solution to the problem, or, if there is no solution, a failure:

```
$func? Solution?   sN = ePos;

Solution?  sN =
  <Next-Queen? 1 sN >;
```

10.2.THE SEQUENCE PROBLEM

Now we consider the problem of finding a ground expression Ge having the following property [Wir 73]:

(1) Ge contains no parentheses, and any symbol appearing in Ge is either 1, 2, or 3.

(2) The length of Ge is equal to a given number Len.

(3) There is no such ground expressions Gea, Geb, and Gec that Gec is non-empty, and there holds

```
Ge = Gea Gec Gec Geb
```

i.e. Ge does not contain two adjacent non-empty equal subexpressions.

The desired expression can be found in the following way. We may start with an empty expression, and then try to extend it, adding digits to it one by one. Upon adding a digit, we have to check the expression thus obtained, to make sure that the expression does not have the form Gea Gec Gec Geb, where Gec is non-empty. A moment's thought reveals that, actually, it is sufficient to check that the expression obtained by adding a digit does not have the form

```
Gea Gec Gec
```

Here is the definition of the predicate Unacceptable?, which determines whether the argument has the above form:

```
$func? Unacceptable? e.String = ;

Unacceptable?  e.String =
  <Div <Length e.String> 2> :: s.Max,
  {
  s.Max : 0
    = $fail;
    = 1
      $iter \{ <"<" (sK) (s.Max)> = <"+" sK 1>; }
      :: sK,
      <Right 0 sK <Middle 0 sK e.String>> :: eU,
      <Right 0 sK e.String> :: eV,
      eU : eV;
  };
```

Now we can define the function Extend? trying to add a digit to the expression, until the sequence has the desired length. If the expression cannot be extended, the function "backtracks", and tries to change previous digits.

```
$func? Extend?       s.Len e.String = e.String;

Extend?  s.Len e.String =
  {
  <Length e.String> : s.Len
    = e.String;
    = 1 $iter \{ <"<" (s.Digit) (3)> = <"+" s.Digit 1>; }
        :: s.Digit,
      e.String s.Digit :: e.String,
```

```
           # <Unacceptable? e.String>,
             <Extend? s.Len e.String>;
       };
```

And, finally, we define the function Find-String?, taking as argument the length of the desired sequence, and returning either the desired sequence (if found), or a failure (if the desired sequence does not exist).

```
$func? Find-String?  s.Len = e.String;

Find-String?  s.Len =
  <Extend? s.Len >;
```

11.EXAMPLE: A COMPILER FOR A SMALL IMPERATIVE LANGUAGE

The primary objective of this section is to consider the traditional compiler writing techniques in the framework of Refal Plus. These techniques are applied to a compiler for a small imperative language, the language and the compiler being similar to those described in [War 80].

Illustrative though this compiler may be, it exceeds in size all other example programs dealt with in the book, and consists of several modules.

11.1.THE SOURCE LANGUAGE

A source language program is a finite sequence of tokens. A token is represented by a finite character sequence, whose syntax is described by the following grammar (see Chapter II, section 1):

```
$     Token =
$          KeyWord | Identifier | Numeral.
$     KeyWord =
$          ";" | "(" | ")" | "+" | "-" | "*" | "-" |
$          ":=" | "<=" | '<>' | "<" | ">=" | ">" | "=".
$          "DO" | "ELSE" | "IF" | "READ" | "THEN" |
$          "WHILE" | "WRITE".
$     Identifier = Letter { Letter | Digit }.
$     Numeral = Digit { Digit }.
```

The keywords are words reserved for special purposes and must not be used as normal identifier names.

Keywords are case insensitive, i.e. the small and capital letters appearing in the keywords are considered as completely equivalent.

Tokens may be separated by spaces, horizontal tabs, and newline characters, which cannot occur within tokens and are ignored unless they are essential to separate two consecutive tokens.

Some token sequences are not syntactically correct programs. Hence, the token sequence produced by scanning the input character stream must be parsed to see whether it has the following syntax:

```
$     Program = StatementSequence.
$     StatementSequence = Statement { ";" Statement }.
$     Statement =
$          "IF" Test "THEN" Statement "ELSE" Statement |
$          "WHILE" Test "DO" Statement |
$          "READ" VariableName |
$          "WRITE" Expression |
$          "(" StatementSequence ")".
$          VariableName ":=" Expression |
$          Empty.
$     Empty = .
$     Test = Expression CompOperator Expression.
$     CompOperator = "=" | "<=" | "<>" | "<" | ">=" | ">".
$     Expression = Term { AddOperator Term }.
$     Term = Factor { MultOperator Factor }.
$     Factor = VariableName | Value | "(" Expression ")".
$     AddOperator = "+" | "-".
$     MultOperator = "*" | "/".
$     VariableName = Identifier.
$     Value = Integer.
```

A program is a statement sequence. The statements are executed sequentially, from left to right. Each statement may access, and change, the values of variables.

An if statement

```
IF Cond THEN St1 ELSE St2
```

tests the condition Cond. If the condition is satisfied, the statement St1 is executed, otherwise, the statement St2 is executed.

A while statement

```
WHILE Cond DO St
```

tests the condition Cond. If the condition is satisfied, the statement St is executed, and the execution of the whole construct is repeated. Otherwise, if the condition is not satisfied, the execution of the construct terminates.

A read statement

```
READ Var
```

reads an integer from the input device, and assigns the integer as value to the variable Var.

A write statement

```
WRITE Expr
```

evaluates the arithmetic expression Expr to produce an integer, which is written to the output device.

A compound statement

```
( St1; St2; ... StN )
```

specifies the sequential execution of the statements St1, St2, ..., StN.

An assignment statement

```
Var := Expr
```

evaluates the expression Expr to produce an integer, which is assigned as value to the variable Var.

An empty statements specifies no action.

Conditions and arithmetic expressions have their conventional meaning. The multiplication and division operators have precedence over the addition and subtraction operators.

The variables appearing in the program don't have to be declared. The initial variable values are undefined.

Here is an example program, which inputs an integer, and then computes and outputs the factorial of the integer.

```
read value;
count:=1;
result:=1;
while count<value do
  (
  count:=count+1;
  result:=result*count
  );
write result
```

11.2.THE TARGET LANGUAGE

The target program produced by the compiler is written in "machine code", and has the following syntax:

```
$    Program = { Directive }.
$    Directive =
$         Instruction | "BLOCK" "," Value ";".
$    Instruction =
$         InstructionCode "," Value ";" |
$    InstructionCode =
$         ADD  | SUB  |  MUL  | DIV  | LOAD  | STORE |
$         ADDC | SUBC |  MULC | DIVC | LOADC |
$         JUMPEQ | JUMPNE | JUMPLT | JUMPGT | JUMPLE | JUMPGE
$         JUMP | READ | WRITE | HALT.
$    Value = Integer.
```

A program is a directive sequence, each directive being

either an "instruction", i.e. machine command, or a memory allocation directive.

We assume the main store of the machine to consist of cells, each cell associated with its address, a unique non-negative integer (thus, the cells are numbered from 1). A cell may hold either an instruction or an integer.

The execution of the program always starts from the first sell.

In addition to the main store, the machine has an accumulator, which is capable of containing an integer.

A directive

```
BLOCK,Int;
```

specifies that at this place in the program there must be allocated Int store cells containing no instructions. This directive usually is put at the end of the program, and used for allocating cells that are to hold the values of the program's variables.

A machine instruction has the form

```
Op,Value;
```

where Op is the instruction's name, and Value the instruction's operand. The meaning of the operand Value depends on the instruction's name. Some instructions assume Value to be the address of the cell. Others assume Value to be an integer. There are instructions, however, which needn't any operand, in which cases Value must be equal to zero.

An instruction LOAD,Addr; loads the contents of the cell having the address Addr into the accumulator.

An instruction STORE,Addr; puts the contents of the accumulator into the cell having the address Addr.

An instruction LOADC,Int; loads the integer Int into the accumulator.

Instructions ADD, SUB, MUL and DIV have the form Op,Addr; and compute respectively the sum, difference, product, and the the truncated quotient of two integers. The first integer is the one contained by the accumulator, and the second the one contained in the cell having the address Addr. The result of the operation is put into the accumulator.

Instructions ADDC, SUBC, MULC, and DIVC have the form Op,Int; and compute respectively the sum, difference, product, and the truncated quotient of two integers. The first integer is the one contained in the accumulator, and the second integer is Int, i.e.the one contained in the operand of the instruction. The result of the operation is put into the accumulator.

An instruction READ,Addr; reads an integer from the input device and puts it into the cell having the address Addr.

An instruction WRITE,0; writes the integer contained by the accumulator to the output device.

An instruction HALT,0; halts the execution of the program.

An instruction JUMP,Addr; causes the control to jump to the instruction contained in the cell having the address Addr.

And, finally, the last group of instructions comprises the conditional jumps JUMPEQ, JUMPNE, JUMPLT, JUMPGT, JUMPLE, and JUMPGE, all having the form Op,Addr;. They are executed in the following way. First, the contents of the accumulator is compared with zero. If the condition implied by the instruction's name is satisfied, the control jumps to the instructions contained in the cell having the address Addr, otherwise, to the next instruction.

Which condition is tested, is determined by the last two letters in the instruction's name. EQ means testing the accumulator's contents for being equal to 0, NE for not being equal to 0, LT for being less than 0, GT for being greater than 0, LE for being less than or equal to 0, GE for being greater than or equal to 0.

The above program computing the factorial will be translated by the compiler into the following target program in machine code.

```
001  READ,21;       008  JUMPGE,16;     015  JUMP,6;
002  LOADC,1;       009  LOAD,19;       016  LOAD,20;
003  STORE,19;      010  ADDC,1;        017  WRITE,0;
004  LOADC,1;       011  STORE,19;      018  HALT,0;
005  STORE,20;      012  LOAD,20;       019  BLOCK,3;
006  LOAD,19;       013  MUL,19;
007  SUB,21;        014  STORE,20;
```

The address of each directive is shown on the left of the directive.

11.3. THE GENERAL STRUCTURE OF THE COMPILER

Our compiler has the "classic" structure, and comprises the following parts.

The source character stream (which is often called the concrete program) is read and broken up into tokens by the scanner.

Then the token sequence is analyzed by the parser to produce an abstract syntax tree (which is often called the abstract program).

The abstract program is further translated by the code generator into a program in assembly language. A program in assembly language is very close to the target program, except that, instead of concrete cell addresses, it contains labels, each label representing some (yet) unknown address.

The program in assembly language is then processed by the assembler, which replaces all the label with concrete addresses, thereby producing the target machine code program.

The information about the correspondence between the variable names and labels is kept in the dictionary of variables. Thus the compiler contains a module dealing with the dictionary,

which is used by the code generator as well as by the assembler.

In comparison with the simplicity of the source language, the structure of our compiler may well seem to be rather complicated. And, actually, the compiler could have been simplified by merging many compiler's components together. For example, this could have been done with the scanner, parser, and code generator.

It should be kept in mind, however, that, should the source language be more complicated, such "unionism" would make the compiler messy, unreliable and difficult to understand. But, the purpose of our compiler is just to illustrate, in the framework of Refal Plus, the traditional compiler writing techniques applicable to "real-size" compilers.

Taking our example compiler as the starting point, the reader may try to improve it in two respects. First, the source language can be made more complex and more realistic. Second, the compiler can be simplified at the expense of making it less "scientific" and less general.

11.4.THE MODULES OF THE COMPILER AND THEIR INTERFACES

The compiler consists of the following modules:

```
CMP        - the main module
CMPSCN     - the scanner
CMPPRS     - the parser
CMPGEN     - the code generator and assembler
CMPDIC     - the dictionary module
```

The main module does not have the interface part and contains the definition of the goal function Main. All other modules consist of two parts: the interface and the implementation.

The module CMPSCN has the following interface:

```
**
** File CMPSCN.RFI
**

$func  Init-Scanner  s.Channel = ;
$func  Read-Token     = s.TokenClass s.TokenInfo;
$func  Term-Scanner  = ;
```

The module exports three functions.

The function Init-Scanner initializes the scanner. The parameter s.Channel is a reference to the channel that provides characters read by the scanner. This channel must have been opened for reading before calling Init-Scanner.

The function Term-Scanner must be called after the reading of the source program has been finished. This enables the scanner to terminate its activities and to get ready for reading another source program.

The function Read-Token returns the source programs's current token represented by two symbols: the first symbol indicates the class the token belongs to, while the second symbol provides additional information about the token.

The module CMPPRS has the following interface:

```
**
** File: CMPPRS.RFI
**

$func Parse  s.Channel = t.Program;
```

The interface exports the function Parse, which reads the source program from the channel s.Channel (via the scanner) and produces the abstract program t.Program. The channel s.Channel must have been opened for reading before calling Parse.

If the source program contains syntax errors, the function Parse returns $error(Ge), where Ge is an error message describing the first error encountered by Parse.

The module CMPGEN has the following interface:

```
**
** File: CMPGEN.RFI
**

$func Gen-Code    t.Program = t.Code;
$func Write-Code  t.Code = ;
```

The interface exports two functions.

The function Gen-Code takes as argument t.Program, an abstract program, and returns t.Code, the result of compiling t.Program into the machine code. The program t.Code is represented by an abstract syntax tree.

The function Write-Code takes as argument a machine code program represented by an abstract syntax tree, and, upon converting it into the character stream representation, writes it to the standard output device.

The module CMPDIC has the following interface:

```
**
** File: CMPDIC.RFI
**

$func Make-Dic      = s.Dic;
$func Lookup-Dic    s.Key s.Dic = s.Ref;
$func Allocate-Dic  s.Dic s.StartAddr = s.FreeAddr;
```

The interface exports four functions.

The function Make-Dic returns a reference to a new empty dictionary.

The function Lookup-Dic returns the label associated with the key s.Key in the dictionary referred to by s.Dic. If the key s.Key has not been registered in the dictionary, a new unique label is created, associated with the key s.Key, and returned as the function's result.

The function Allocate-Dic looks through the dictionary referred to by s.Dic and binds all labels registered in the dictionary to different addresses. If the dictionary contains N keys, the labels get bound to consecutive addresses starting with s.StartAddr. The result returned by the function is the first free address.

11.5. THE MAIN MODULE

The main module of the compiler links all parts of the compiler together. The name of the source program's file is assumed to be passed to the compiler as the first argument in the command line. Thus the compiler should be called by the command

```
          CMP FileName
```

where FileName is a file name. This name is accessed by the compiler by means of the library function Arg.

```
**
** File CMP.RF
**

$use DOS;
$use STDIO;

$use CMPPRS;
$use CMPGEN;

$func Compile          e.FileName = ;

Main  =
  <Arg 1> :: e.FileName,
  <Compile e.FileName>;

Compile  e.FileName =
  <Channel> :: s.Chl,
  <Open-File s.Chl e.FileName "r">,
  <Parse s.Chl> :: t.AProgram,
  <Close-Channel s.Chl>,
  <Gen-Code t.AProgram> :: t.Code,
  <Write-Code t.Code>;
```

11.6. THE SCANNER

The result produced by the scanner is a token sequence, each token being represented by two symbols. The first of the symbols indicates the class of the token.

In the following we describe the syntax of ground expressions by means of an extended Backus-Naur form (EBNF), with non-terminals written as Refal Plus variables. The ground expressions denoted by the non-terminals are assumed to correspond to the types of the non-terminals.

Thus the syntax of the token sequence produced by the scanner can be described as follows:

```
$    e.Tokens = { e.Token }.
$    e.Token =
$        Key s.Key | Name s.Name | Value s.Value |
$        Char s.Char.
$    s.Key  = s.Word.
$    s.Name = s.Word.
$    s.Value = s.Int.
```

A token of the form Key s.Key represents a keyword, s.Key being the word symbol whose character representation corresponds to the key word. A token of the form Name s.Name represents a variable name, s.Name being the word symbol whose character representation corresponds to the variable name (which, syntactically, is an identifier). A token of the form Value s.Value represents a numeric constant, s.Value being the corresponding numeric symbol. A token of the form Char s.Char represents an unidentified character s.Char.

When the reading of the source program has been finished, the scanner generates the token Key Eof.

The module CMPSCN has the following implementation:

```
**
** File: CMPSCN.RF
**

$use STDIO;
$use CLASS;
$use CONVERT;
$use BOX;

$func  Scan-Token
          s.Chl e.Line = s.TokenKey s.TokenInfo (e.Line1);
$func  Scan-Id-Rest
          (e.Id-Chars)  e.Chars = s.TokenKey s.Word (e.Rest);
$func  Scan-Int-Rest
          (e.Int-Chars) e.Chars = s.TokenKey s.Int  (e.Rest);
$func? Blank?            s.Char = ;
$func? One-Char-Token?   s.Char = ;
$func? Compound-Token?   s.Char e.Line = s.Word e.Rest;
$func? KeyWord?          s.Word = ;

** Boxes for storing the channel to be read,
```

```
** and the rest of the current line.


$box Scan-Chl Scan-Line;

Init-Scanner  s.Chl =                 ** Scanner initialization.
  <Store &Scan-Chl s.Chl>,            ** The channel into box.
  <Store &Scan-Line >;                ** The current line is empty.

Term-Scanner  =                       ** Scanner termination.
  <Store &Scan-Chl   >,               ** Forgetting the channel
  <Store &Scan-Line  >;               ** and the current line.

Read-Token  =                         ** The reading of a token.
  <? &Scan-Chl> : s.Chl,
  <? &Scan-Line> :: e.Line,
  <Scan-Token s.Chl e.Line>
        :: s.TokenKey s.TokenInfo (e.Line),
  <Store &Scan-Line e.Line>,
    = s.TokenKey s.TokenInfo;

Scan-Token  s.Chl e.Line =
  e.Line :
  {
  =                                         ** The line rest is
    {                                       ** empty. Reading the
    <Read-Line! s.Chl> :: e.Line            ** next line.
      = <Scan-Token s.Chl e.Line>;
      = Key Eof ();                         ** End of file.
    };
  s.Char e.Rest =                           ** Examining the
    {                                       ** current character.
    <Blank? s.Char>
      = <Scan-Token s.Chl e.Rest>;
    <Letter? s.Char>
      = <Scan-Id-Rest (s.Char) e.Rest>;
    <Digit? s.Char>
      = <Scan-Int-Rest (s.Char) e.Rest>;
    <One-Char-Token? s.Char>
      = Key <To-Word s.Char> (e.Rest);
    <Compound-Token? s.Char e.Rest> :: s.Word e.Rest
      = Key s.Word (e.Rest);
      = Char s.Char (e.Rest);         ** Unidentified character.
    };
  };

** Getting the rest of an identifier.

Scan-Id-Rest  (e.Id-Chars) e.Rest =
  {
  e.Rest : s.Char e.Rest1,
              \{<Letter? s.Char>; <Digit?  s.Char>;}
    = <Scan-Id-Rest (e.Id-Chars s.Char) e.Rest1>;
```

```
    = <To-Word <To-Upper e.Id-Chars>> : s.Word,
      {<KeyWord? s.Word> = Key; = Name;} :: s.TokenKey,
      = s.TokenKey s.Word (e.Rest);
  };

** Getting the rest of an integer.

Scan-Int-Rest  (e.Int-Chars) e.Rest =
  {
  e.Rest : s.Char e.Rest1, <Digit?  s.Char>
    = <Scan-Int-Rest (e.Int-Chars s.Char) e.Rest1>;
    = Value <To-Int e.Int-Chars> (e.Rest);
  };

Blank?  s.Char =                    ** A whitespace?
  ' \n\t' : e s.Char e;

One-Char-Token?  s.Char =           ** A one-character token?
  ';()+-*/' : e s.Char e;

Compound-Token?                     ** Trying to get a multi-
  \{                                ** character token.
  ':=' e.Rest = ":=" e.Rest;
  '<=' e.Rest = "<=" e.Rest;
  '<>' e.Rest = "<>" e.Rest;
  '<'  e.Rest = "<"  e.Rest;
  '>=' e.Rest = ">=" e.Rest;
  '>'  e.Rest = ">"  e.Rest;
  '='  e.Rest = "="  e.Rest;
  };

KeyWord?              ** Is the identifier a key word?
  \{
  DO ; ELSE ; IF ; READ ; THEN ; WHILE ; WRITE ;
  };
```

11.7. THE PARSER

The parser, residing in the module CMPPRS, transforms a token sequence into an abstract program, i.e. a parse tree.

Our parser will use the technique referred to as a recursive-descent analysis.

Consider, for example, the following grammar:

```
$     Sentence = Subject Predicate.
$     Subject = "cats" | "dogs".
$     Predicate = "sleep" | "eat".
```

Suppose we are given the token sequence

```
          "dogs" "eat"
```

and want to determine whether this sequence is a well-formed sentence. This amounts to determining whether this sequence can be derived from the non-terminal Sentence. But, the grammar specifies that the set of token sequences generated by the non-terminal Sentence is equal to the set of sequences generated by the non-terminal sequence Subject Predicate. Thus, the original problem can be reduced to determining whether the input sequence can be divided into two subsequences such that the first one can be derived from the non-terminal Subject, and the second one from the non-terminal Predicate.

How can a sequence be divided into two parts, of which the first is generated by the non-terminal Subject? It, can, obviously, be done by testing whether the sequence begins with one of the tokens "cats" or "dogs".

Thus we come to the following method of analyzing token sequences.

Each non-terminal A appearing in the grammar is associated with a function A having the following declaration:

```
$func? A e.Token = e.Rest;
```

This function A tests whether the input token sequence e.Token begins with a sequence derivable from the non-terminal A, and, if so, deletes this beginning and returns the rest of the input sequence thus obtained. Otherwise, if the input sequence does not begin with a sequence derivable from the non-terminal A, the function A returns a failure.

It goes without saying that the above method is applicable only in cases where, for each non-terminal A and each input sequence Z there exists no more than one way of dividing Z into two subsequences, of which the first is derivable from A. In many cases, however, the grammar can be rewritten in such a way that this restriction will be satisfied. An interested reader may find further details in [Wir 76].

Proceeding from the above consideration, we can now define the function "Sentence" either deleting from the input sequence the beginning derivable from the non-terminal Sentence, or failing, if this is unfeasible.

```
$func? "Sentence" e.Token = e.Rest;
$func? "Subject"  e.Token = e.Rest;
$func? "Predicate"   e.Token = e.Rest;
$func? Token?  s? e.Token = e.Rest;

"Sentence"  eZ =
  <"Subject" eZ> :: eZ,
  <"Predicate"  eZ> :: eZ,
    = eZ;

"Subject"  eZ =
  \{
  <Token? "dogs" eZ> :: eZ = eZ;
  <Token? "cats"  eZ> :: eZ = eZ;
```

```
  };

"Predicate"  eZ =
  \{
  <Token? "sleep" eZ> :: eZ = eZ;
  <Token? "eat" eZ> :: eZ = eZ;
  };

Token?  s? eZ =
  eZ : s? eZ0
    = eZ0;
```

The function Token? is used for deleting a terminal symbol, which is passed as the first argument.

Now we can return to considering the module CMPPRS, in which we have to deal with two additional problems.

First, instead of returning the input token sequence as a whole, the scanner produces tokens one by one. Thus, each of the parsing functions, instead of taking as argument the whole token sequence, takes as argument a single token, the one that has been read last. This token is the one to be analyzed next. Similarly, each of the parsing functions, instead of returning the whole rest of the token sequence, returns only the first unparsed token. (It should be kept in mind, however, that each token is represented by two Refal Plus symbols.)

Second, in addition to checking the syntax correctness of the source program, the parser has to transform the token sequence into the corresponding abstract program, i.e. into an abstract syntax tree. Thus, the parsing function associated with a non-terminal A is usually declared as follows:

```
$func A sC sI = sC sI tX;
```

where sC sI represent the current token, and tX is the result of translating the token sequence consumed by the function into an abstract syntax tree.

Third, if a syntax error is detected, the parser, instead of returning a failure, must produce an error $error(Ge), where Ge is an error message describing the error. For this reason, the parsing functions are declared as unfailing ones.

Here is the syntax of the abstract programs produced by the parser:

```
$    t.Program = (Program t.Statement).
$    t.Statement =
$         (Assign s.Name t.Expr) |
$         (If t.Test t.Statement t.Statement) |
$         (While t.Test t.Statement) |
$         (Read s.Name) |
$         (Write t.Expr) |
$         (Seq t.Statement t.Statement)
$         (Skip).
$    t.Test = (Test s.Comp-Oper t.Expr t.Expr).
```

```
$    t.Expr =
$         (Const s.Value) |
$         (Name s.Name) |
$         (Op t.Oper t.Expr t.Expr).
$    s.Comp-Oper = Eq | Ne | Gt | Ge | Lt | Le.
$    s.Oper = Add | Sub | Div | Mul.
$    s.Name = s.Word.
$    s.Value = s.Int.
```

Thus, a construction written in abstract syntax usually has the form

```
          (KeyWord Gt1 Gt2 ... GtN)
```

where the key word KeyWord is a word symbol representing the construct's name, and the ground terms Gt1, Gt2, ..., GtN represent the component constructs also written in abstract syntax. Since the correspondence between the constructs written in concrete and abstract syntax is evident, we won't dwell on this point.

Here is the implementation of the module CMPPRS:

```
**
** File: CMPPRS.RF
**

$use CMPSCN;

$func  Program         sC sI          = sC sI tX;
$func  Statement-Seq   sC sI          = sC sI tX;
$func  Rest-St-Seq     sC sI tX0      = sC sI tX;
$func  Statement       sC sI          = sC sI tX;
$func  Test            sC sI          = sC sI tX;
$func  Expr            sC sI          = sC sI tX;
$func  Rest-Expr       sC sI tX1      = sC sI tX;
$func  Term            sC sI          = sC sI tX;
$func  Rest-Term       sC sI tX1      = sC sI tX;
$func  Factor          sC sI          = sC sI tX;
$func  Comp-Op         sC sI          = sC sI s.Comp-Oper;
$func? Add-Op?         sC sI          = sC sI s.Oper;
$func? Mul-Op?         sC sI          = sC sI s.Oper;
$func? Token?          sI? sC sI      = sC sI ;
$func  Accept          sI? sC sI      = sC sI ;
$func? Name?           sC sI          = sC sI s.Name;
$func? Value?          sC sI          = sC sI s.Value;


Parse  s.Chl =
  <Init-Scanner s.Chl>,
  <Program <Read-Token>> :: sC sI t.Program,
  <Term-Scanner>,
  {                         ** Is the rest of the program
  sC sI : Key Eof           ** empty?
```

```
    = t.Program;
    =  $error sC sI " instead of Eof after the program";
  };

Program  sC sI =                          ** Program.
  <Statement-Seq sC sI> :: sC sI tX,
    = sC sI (Program tX);

Statement-Seq  sC sI =                    ** Statement
  <Statement sC sI> :: sC sI tX0,         ** sequence.
    = <Rest-St-Seq sC sI tX0>;

Rest-St-Seq  sC sI tX0 =
  \? {
  <Token? ";" sC sI> :: sC sI \!
    <Statement-Seq sC sI> :: sC sI tX,
    = sC sI (Seq tX0 tX);
  \!
    = sC sI tX0;
  };

Statement  sC sI =                        ** Statement.
  \? {
  <Name? sC sI> :: sC sI s.Name \!
    <Accept ":=" sC sI> :: sC sI,
    <Expr sC sI> :: sC sI t.Expr,
      = sC sI (Assign s.Name t.Expr);
  <Token? "IF" sC sI> :: sC sI \!
    <Test sC sI> :: sC sI t.Test,
    <Accept "THEN" sC sI> :: sC sI,
    <Statement sC sI> :: sC sI t.Then,
    <Accept "ELSE" sC sI> :: sC sI,
    <Statement sC sI> :: sC sI t.Else,
      = sC sI (If t.Test t.Then t.Else);
  <Token? "WHILE" sC sI> :: sC sI \!
    <Test sC sI> :: sC sI t.Test,
    <Accept "DO" sC sI> :: sC sI,
    <Statement sC sI> :: sC sI t.Do,
      = sC sI (While t.Test t.Do);
  <Token? "READ" sC sI> :: sC sI \!
    <Name? sC sI> :: sC sI s.Name,
      = sC sI (Read s.Name);
  <Token? "WRITE" sC sI> :: sC sI \!
    <Expr sC sI> :: sC sI t.Expr,
      = sC sI (Write t.Expr);
  <Token? "(" sC sI> :: sC sI \!
    <Statement-Seq sC sI> :: sC sI t.Stmt,
    <Accept ")" sC sI> :: sC sI,
      = sC sI t.Stmt;
  \!
      = sC sI (Skip);
  };
```

```
Test  sC sI =                                ** Test.
  <Expr sC sI> :: sC sI t.Expr1,
  <Comp-Op sC sI> :: sC sI t.Op,
  <Expr sC sI> :: sC sI t.Expr2,
    = sC sI (Test t.Op t.Expr1 t.Expr2);

Expr  sC sI =                                ** Expression.
  <Term sC sI> :: sC sI t.X0,
    = <Rest-Expr sC sI t.X0>;

Rest-Expr  sC sI t.X1 =
  \? {
  <Add-Op? sC sI> :: sC sI s.Op \!
    <Term sC sI> :: sC sI t.X2,
      = <Rest-Expr sC sI (Op s.Op t.X1 t.X2)>;
  \!
      = sC sI t.X1;
  };

Term  sC sI =                                ** Term.
  <Factor sC sI> :: sC sI t.X0,
    = <Rest-Term sC sI t.X0>;

Rest-Term  sC sI t.X1 =
  \? {
  <Mul-Op? sC sI> :: sC sI s.Op \!
    <Factor sC sI> :: sC sI t.X2,
      = <Rest-Term sC sI (Op s.Op t.X1 t.X2)>;
  \!
      = sC sI t.X1;
  };

Factor  sC sI =                          ** Factor.
  \? {
  <Name? sC sI> :: sC sI s.Name \!
    = sC sI (Name s.Name);
  <Value? sC sI> :: sC sI s.Value \!
    = sC sI (Const s.Value);
  <Token? "(" sC sI> :: sC sI \!
    <Expr sC sI> :: sC sI t.Expr,
    <Accept ")" sC sI> :: sC sI,
      = sC sI t.Expr;
  \!
      $error "Invalid factor start: " sC sI;
  };

Comp-Op  sC sI =                     ** Comparison operator.
  {
  sC : Key,
  ("=" Eq) ("<>" Ne) ("<=" Le) ("<" Lt) (">=" Ge) (">" Gt)
  : e (sI s.Op) e
    = <Read-Token> s.Op;
    = $error "Invalid comparison operator: " sC sI;
```

```
  };

Add-Op?  Key sI =              ** Additive operator.
  ("+" Add) ("-" Sub) : e (sI s.Op) e
    = <Read-Token> s.Op;

Mul-Op?  Key sI =              ** Multiplicative operator.
  ("*" Mul) ("/" Div) : e (sI s.Op) e
    = <Read-Token> s.Op;

** Tries to consume a key word sI?, and
** returns a failure, if this is unfeasible.

Token?  sI? Key sI? = <Read-Token>;

** Tries to consume a key word sI?, and
** generates an error, if this is unfeasible.

Accept
  {
  sI? Key sI? = <Read-Token>;
  sI? sC  sI  = $error sC sI " instead of " Key sI?;
  };

** Variable name.

Name?  Name sI = <Read-Token> sI;

** Value.

Value?  Value sI = <Read-Token> sI;
```

11.8. THE CODE GENERATOR

Assembler language programs produced by the code generator are represented by ground terms having the following syntax:

```
$     t.Code =
$          (Seq { t.Code } ) |
$          (Instr s.Instr s.Operand) |
$          (Label s.Label) |
$          (Block s.Value).
$     s.Operand = s.Label | s.Value.
$     s.Label = s.Box.
$     s.Value = s.Int.
$
$     s.Instr =
$          Add  | Sub  | Div  | Mul  | Load  | Store |
$          Addc | Subc | Divc | Mulc | Loadc |
$          Jumpeq | Jumpne | Jumplt | Jumpgt | Jumple | Jumpge
$          Jump | Read | Write | Halt |
```

Assembler language programs may contain labels to be replaced with absolute addresses by the assembler. Assembling a program proceeds in two steps. First, the assembler determines the addresses associated with instructions and variables, and puts each address associated with a label into the box referred to by the label. Second, all labels are replaced with the addresses associated with them, i.e. each reference to a box is replaced with the contents of the box.

The module CMPGEN has the following implementation:

```
**
** File: CMPGEN.RF
**

$use STDIO;
$use CLASS;
$use ARITHM;
$use BOX;

$use CMPDIC;

$func Enc-Program      t.Program s.Dic = t.Code;
$func Enc-St           t.St s.Dic = t.Code;
$func Enc-Test         t.Test s.Label s.Dic = t.TestC;
$func Unless-Op        s.Op = s.Jump-If;
$func Enc-Expr         t.Expr s.Dic = t.ExprC;
$func Enc-Sub-Expr     t.Expr sN s.Dic = t.ExprC;
$func Literal-Op       s.Op = s.OpCode;
$func Memory-Op        s.Op = s.OpCode;
$func Assemble         t.Code s.StartAddr = s.FreeAddr;
$func Assemble-Seq     e.CodeSeq s.Addr = s.FreeAddr;
$func Dereference      t.Code = t.Target;
$func Dereference-Seq e.CodeSeq = e.CodeSeqD;
$func Write-Code-Seq   e.CodeSeq = ;

** Generates an assembler language program
** from an abstract program.

Gen-Code  t.Program =
      ** Creating an empty dictionary.
  <Make-Dic> :: s.Dic,
      ** Generating the abstract program.
  <Enc-Program t.Program s.Dic> :: t.Code,
      ** Allocating memory for the program's instructions.
  <Assemble t.Code 1> :: s.FreeAddr,
      ** Allocating memory for the program's variables.
  <Allocate-Dic s.Dic s.FreeAddr> :: s.EndAddr,
      ** Replacing the labels with their addresses.
  <Dereference t.Code> :: t.CodeD,
      ** Generating the directive BLOCK.
  <"-" s.EndAddr s.FreeAddr> :: s.BlockLength,
  (Seq t.CodeD (Block s.BlockLength)) :: t.Target,
    = t.Target;
```

```
** Encodes a program.

Enc-Program  (Program t.St) s.Dic =
  <Enc-St t.St s.Dic> :: t.StC,
  <Box> :: s.L,
    = (Seq t.StC (Instr Halt 0) (Label s.L));

** Encodes a statement.

Enc-St  (s.KeyWord e.Info) s.Dic =
  (s.KeyWord e.Info) :
  {
  (Assign sX t.Expr) =
    <Lookup-Dic sX s.Dic> :: s.Addr,
    <Enc-Expr t.Expr s.Dic> :: t.ExprC,
      = (Seq t.ExprC (Instr Store s.Addr));
  (If t.Test t.Then t.Else) =
    <Box> :: s.L1, <Box> :: s.L2,
    <Enc-Test t.Test s.L1 s.Dic> :: t.TestC,
    <Enc-St t.Then s.Dic> :: t.ThenC,
    <Enc-St t.Else s.Dic> :: t.ElseC,
      = (Seq
            t.TestC
            t.ThenC
            (Instr Jump s.L2)
          (Label s.L1)
            t.ElseC
          (Label s.L2)
        );
  (While t.Test t.Do) =
    <Box> :: s.L1, <Box> :: s.L2,
    <Enc-Test t.Test s.L2 s.Dic> :: t.TestC,
    <Enc-St t.Do s.Dic> :: t.DoC,
      = (Seq
          (Label s.L1)
            t.TestC
            t.DoC
            (Instr Jump s.L1)
          (Label s.L2)
        );
  (Read s.X) =
    <Lookup-Dic s.X s.Dic> :: s.Addr,
      = (Instr Read s.Addr);
  (Write t.Expr) =
    <Enc-Expr t.Expr s.Dic> :: t.ExprC,
      = (Seq t.ExprC (Instr Write 0));
  (Seq t.St1 t.St2) =
    <Enc-St t.St1 s.Dic> :: t.StC1,
    <Enc-St t.St2 s.Dic> :: t.StC2,
      = (Seq t.StC1 t.StC2);
  (Skip) =
      = (Seq );
```

```
  };

** Encodes a test.

Enc-Test  (Test s.Op t.Arg1 t.Arg2) s.Label s.Dic =
  <Enc-Expr (Op Sub t.Arg1 t.Arg2) s.Dic> :: t.ExprC,
  <Unless-Op s.Op> :: s.Jump-If,
    = (Seq t.ExprC (Instr s.Jump-If s.Label));

Unless-Op                         ** Generates a jump.
  {
  Eq = Jumpne; Ne = Jumpeq;
  Lt = Jumpge; Gt = Jumple;
  Le = Jumpgt; Ge = Jumplt;
  };

** This function compiles an arithmetic expression.
** Auxiliary variables are created to keep
** the values obtained by evaluating subexpressions.
** The evaluation order of the subexpressions is chosen in
** such a way as to reduce the number of auxiliary variables.

Enc-Expr  t.Expr s.Dic
  = <Enc-Sub-Expr t.Expr 0 s.Dic>;

Enc-Sub-Expr  (s.KeyWord e.Info) sN s.Dic =
  (s.KeyWord e.Info) :
  {
  (Const sC) =
      = (Instr Loadc sC);
  (Name sX) =
    <Lookup-Dic sX s.Dic> :: s.Addr,
      = (Instr Load s.Addr);
  (Op s.Op t.Expr1 t.Expr2) =
    t.Expr2 :
    {
    (Const sC2) =
      <Enc-Sub-Expr t.Expr1 sN s.Dic> :: t.Expr1C,
      <Literal-Op s.Op> :: s.OpCode,
        = (Seq t.Expr1C (Instr s.OpCode sC2));
    (Name sX2) =
      <Enc-Sub-Expr t.Expr1 sN s.Dic> :: t.Expr1C,
      <Memory-Op s.Op> :: s.OpCode,
      <Lookup-Dic sX2 s.Dic> :: s.Addr,
        = (Seq t.Expr1C (Instr s.OpCode s.Addr));
    (Op e) =
      <Lookup-Dic sN s.Dic> :: s.Addr,
      <Enc-Sub-Expr t.Expr2 sN s.Dic> :: t.Expr2C,
      <"+" sN 1> :: sN1,
      <Enc-Sub-Expr t.Expr1 sN1 s.Dic> :: t.Expr1C,
      <Memory-Op s.Op> :: s.OpCode,
        = (Seq
            t.Expr2C
```

```
            (Instr Store s.Addr)
            t.Expr1C
            (Instr s.OpCode s.Addr)
          );
    };
  };

Literal-Op                      ** Generates the names of
  {                             ** the instructions with
  Add = Addc; Sub = Subc;       ** literal operands.
  Mul = Mulc; Div = Divc;
  };

Memory-Op                       ** Generates the names of
  {                             ** the instructions with
  Add = Add; Sub = Sub;         ** address operands.
  Mul = Mul; Div = Div;
  };

** Allocates memory for the instructions.

Assemble  t.Code s.A0 =
  t.Code :
  {
  (Seq e.CodeSeq) =
      = <Assemble-Seq e.CodeSeq s.A0>;
  (Instr s s) =
      = <"+" s.A0 1>;
  (Label s.Label) =
    <Store s.Label s.A0>
      = s.A0;
  };

Assemble-Seq  e.CodeSeq s.A0 =
  e.CodeSeq :
  {
  t.Code e.Rest =
    <Assemble t.Code s.A0> :: s.A1,
    = <Assemble-Seq e.Rest s.A1>;
  =
    = s.A0;
  };

** Replaces the labels with their addresses.

Dereference  t.Code =
  t.Code :
  {
  (Seq e.CodeSeq) =
    (Seq <Dereference-Seq e.CodeSeq>);
  (Instr s.Instr s.Value) =
    {
    <Int? s.Value>
```

```
      = t.Code;
    <Box? s.Value>
      = (Instr s.Instr <? s.Value>);
    };
  (Label s.Label) =
    (Label <? s.Label>);
  };

Dereference-Seq
  {
  t.Code e.CodeSeq =
      <Dereference t.Code><Dereference-Seq e.CodeSeq>;
  = ;
  };

** Converts the assembler language program to
** the character sequence, and outputs it to
** the standard output device.

Write-Code
  {
  (Seq e.CodeSeq) =
      <Write-Code-Seq e.CodeSeq>;
  (Instr s.Instr s.Value) =
      <Print "    "><Print s.Instr><Print ",">
      <Print s.Value><Print ";\n">;
  (Label s.Label) =
      <Print s.Label><Print ":\n">;
  (Block s.Value) =
      <Print "    BLOCK,"><Print s.Value><Print ";\n">;
  };

Write-Code-Seq
  {
  t.Code e.CodeSeq =
      <Write-Code t.Code><Write-Code-Seq e.CodeSeq>;
  = ;
  };
```

11.9. THE DICTIONARY MODULE

Dictionaries are represented by binary trees [AHU 74]. Each tree node is represented by a box containing three symbols: a key, a value associated with the key, a reference to the left subtree, and a reference to the right subtree. An empty tree is represented by a reference to an empty box.

The module CMPDIC has the following implementation:

```
**
** File: CMPDIC.RF
**
```

```
$use BOX;
$use COMPARE;
$use ARITHM;

** Creates an empty dictionary.

Make-Dic
  = <Box>;

** Looks up the dictionary s.Dic for the label associated
** with the key s.Key. If the key s.Key is not registered
** in the dictionary, the dictionary is updated:
** the key s.Key is associated with a new unique label.

Lookup-Dic  s.Key s.Dic =
  <? s.Dic> :
  {
  =
    <Box> :: s.Ref,
    <Store s.Dic s.Key s.Ref <Box> <Box>>,
      = s.Ref;
  s.Key1 s.Ref1 s.DicL s.DicR =
    <Compare (s.Key)(s.Key1)> :
    {
    '<' = <Lookup-Dic s.Key s.DicL>;
    '>' = <Lookup-Dic s.Key s.DicR>;
    '=' = s.Ref1;
    };
  };

** Allocates memory for the labels registered in
** the dictionary. s.A is the start address.
** The address corresponding to a label is put
** into the box referred to by the label.

Allocate-Dic  s.Dic s.A =
  <? s.Dic> :
  {
  = s.A;
  s.Key s.Ref s.DicL s.DicR =
    <Allocate-Dic s.DicL s.A> :: s.A,
    <Store s.Ref s.A>,
    <"+" s.A 1> :: s.A,
      = <Allocate-Dic s.DicR s.A>;
  };

Write-Dic  s.Dic =
  <? s.Dic> :
  {
  = <Print "_">;
  s.Key s.Ref s.DicL s.DicR =
    <Print "(">, <Write-Dic s.DicL>, <Print " ">,
    <Write s.Key>,
```

```
  <? s.Ref> :
    {
    = ;
    e.Value = <Print "->"> <Write e.Value>;
    },
  <Print " ">,
  <Write-Dic s.DicR>, <Print ")">;
};
```

Chapter II. SYNTAX AND SEMANTICS OF REFAL PLUS

1.NOTATION FOR SYNTAX DESCRIPTION

The syntax is described by means of an extended Backus-Naur form (EBNF).

Syntactic entities (non-terminals) are denoted by English words expressing their intuitive meaning. Terminal symbols of the language are written between acute accents (') or double quotes in order to be distinguished from non-terminals.

A syntax definition is a collection of productions. Each production has the form

```
S = E.
```

where S is a non-terminal and E a syntax expression denoting the set of constructs for which S stands. An expression E has the form

```
T1 | T2 | ... | Tn        (n>0)
```

where the Ti's are the terms of E. Each Ti stands for a set of constructs, and E denotes their union. Each term T has the form

```
F1 F2 ... Fn              (n>0)
```

where the Fi's are the factors of T. Each Fi stands for a set of constructs, and T denotes their product, i.e. the set of constructs of the form X1 X2 ... Xn, where each Xi belongs to the set denoted by Fi.

Each factor F has either the form

```
"x"
```

(x is a terminal symbol, and "x" denotes the singleton set consisting of this single symbol), or

```
( E )
```

(denoting the expression E), or

```
[ E ]
```

(denoting the union of the set denoted by E and the empty construct), or

```
{ E }
```

(denoting the set consisting of the union of the empty construct and the sets E, E E, E E E, etc.).

Here are a few examples of syntactic EBNF-expressions along with the sets of constructs described by the expressions.

```
(A|B)(C|D)     A C, A D, B C, B D
A[B]C          A B C, A C
A {B A}        A, A B A, A B A B A, A B A B A B A, ...
{A|B} C        C, A C, B C, A A C, A B C, B B C, B A C, ...
```

In order for the EBNF syntax description to be distinguishable from the surrounding English text, all the lines containing EBNF productions will be marked by the character $ in the first column.

Since an EBNF-description may be regarded as a text in a language, the syntax of EBNF-descriptions may also be defined in terms of EBNF in the following way:

```
$     Syntax          = { SyntFormula }.
$     SyntFormula     = Identifier "=" SyntExpression ".".
$     SyntExpression = SyntTerm { "|" SyntTerm }.
$     SyntTerm        = SyntFactor { SyntFactor }.
$     SyntFactor      = Identifier | '"' TerminalSymbol '"' |
$          "(" SyntExpression ")" | "[" SyntExpression "]" |
$          "{" SyntExpression "}".
```

2.NATURAL SEMANTICS DESCRIPTION

The method that will be used to describe the execution of Refal Plus programs is known as Natural Semantics or Structural Operational Semantics [Plotkin 1983], [Apt 1983].

The name Natural Semantics is due to the similarity of this description technique to Gentzen's Natural Deduction in mathematical logic. When this technique is applied, the semantics of a language is considered to be an unordered set of judgments about programs and their fragments.

For example, suppose the language to be described deals with expressions containing variables, and the evaluation of the expressions may cause side effects (which may be due to the input/output operations). Then, the language description may involve the judgments of the form

```
Env,St' |- E => X,St"
```

where E is a language expression, Env is an environment, which binds variables to their values in the context of E, St' and St" are global states before the evaluation of E and after the evaluation of E, and X is the result of evaluating E. A global state may contain the state of the store, the files etc.

Informally, such a judgment may be interpreted in the following way: if the evaluation of E starts in the environment E and global state St', it may result in producing the value X and the global state St".

The symbol "|-" (which may be pronounced "implies" or "entails") indicates the dependency of E's evaluation on the current environment Env and the global state St'.

Thus, to define a language semantics, we have to describe a set of (true) judgments about programs and their fragments.

A Natural Semantics definition is an unordered collection of inference rules, which enables true judgments to be derived from other true judgments.

A rule has basically two parts, a numerator and a denominator. The numerator of a rule is an unordered collection of formulae, the premises of the rule, whereas the denominator is always a single formula, the conclusion. A rule that contains no premise on the numerator is called an axiom, in which case the horizontal line may be omitted.

Besides, a rule may contain additional conditions, which impose certain restrictions on the applicability of the rule. The restrictions are placed slightly to the right of the rule or under the rule.

For example, suppose that the language to be described has the construct if E then E' else E", whose meaning may be informally defined as follows.

Evaluate E. If the value of E is true, evaluate E' and assume the value obtained to be the result of the whole construct. Otherwise, if the value of E is false, evaluate E" and assume the value obtained to be the result of the whole construct.

A drawback of the above description is that there is no explicit information about the environment in which the evaluation of E, E', and E" takes place. Thus, the description may be reformulated as follows.

If the result of evaluating E in the environment Env is true, and the result of evaluating E' in the environment Env is X, then X is the result of evaluating if E then E' else E" in the environment Env.

If the result of evaluating E in the environment Env is false, and the result of evaluating E" in the environment Env is X, then X is the result of evaluating if E then E' else E" in the environment Env.

This verbose definition may be given a more concise and comprehensible formulation by means of two inference rules:

```
Env,St |- E => true,St'
Env,St'|- E' => X,St"
---------------------------------------
Env,St |- if E then E' else E" => X,St"


Env,St |- E => false,St'
Env,St'|- E" => X,St"
---------------------------------------
Env,St |- if E then E' else E" => X,St"
```

Take notice of the fact that, in contrast to the informal semantics definition, the formal one provides a precise description of the way in which the global state is modified when the

program is executed.

3.LEXICAL STRUCTURE OF PROGRAM

A program in Refal Plus is a finite character sequence. The syntax analysis of programs is done in two steps. First, the program is scanned, in order to break up the character stream into tokens. Then the token sequence is parsed to produce an abstract syntax tree. Thus, the definition of the Refal Plus syntax comprises two parts. The first part describes the lexical structure of programs, i.e. how tokens are represented by character sequences, whereas the second part describes how to construct programs by combining tokens.

```
$     Program = { Token | WhiteSpace }.

$     WhiteSpace = WhiteStuff { WhiteStuff }.

$     WhiteStuff = Space | HorizontalTab | NewLine | Comment.
```

A program is a finite sequence of tokens. Tokens may be separated by spaces, horizontal tabs, new line characters, and comments, which cannot occur within tokens and are ignored unless they are essential to separate two consecutive tokens.

3.1.COMMENTS

```
$     Comment = "*" CommentTail NewLine
$           | "/*" CommentBody "*/".
$     CommentTail =
$           any character string not containing NewLine.
$     CommentBody =
$           any character string not containing "*/".
```

A comment may begin with an asterisk, in which case it extends to the following new line. Otherwise, it must be enclosed in "comment brackets" /* and */.

```
* This is a comment.
          * And this is a comment.
     /* As well as this one! */
```

3.2.TOKENS

```
$     Token =
$           Bracket | KeyWord |
$           CharacterStringLiteral |
$           WordLiteral | NumericLiteral |
$           Variable.
```

```
$     Bracket = "(" | ")" | "{" | "\{" | "}" | "<" | ">".
```

A token is either a bracket, a key word, a character string literal, a word literal, a numeric literal, or a variable.

3.3.KEY WORDS

```
$     KeyWord =
$          "$box" | "$channel | "$const" | "$error" | "$fail" |
$          "$func" | "$func?" | "$iter" | "$l" | "$r" |
$          "$string" | "$table" | "$trace | | "$traceall" |
$           "$trap" | "$use" | "$vector" | "$with" |
$          "#" | "&" | "," | ":" | "::" | ";" | "=" |
$          "\?" | "\!".
```

The key words that begin with the character $ are case insensitive. For example, here are three different representations of the same key word:

```
$func $Func $FUNC
```

3.4.CHARACTER SYMBOLS

```
$     CharacterStringLiteral =
$          "'" { CharacterLiteral | ContinuationToNewLine } "'".

$     CharacterLiteral =
$          NonSpecialCharacterLiteral | SpecialCharacterLiteral.
$     NonSpecialCharacterLiteral =
$          any ASCII character except acute accent ('),
$          double quote ("), back slash (\), and new line.
$     SpecialCharacterLiteral =
$          "\n" | "\t" | "\v" | "\b" | "\r" | "\f" |
$          "\\" | "\'" | '\"' .
$     ContinuationToNewLine =
$          "\" NewLine.
```

Each character symbol corresponds to an ASCII character, and is represented by a character literal enclosed in acute accents. For instance:

```
'A' 'a' '7' '$'
```

Ordinarily, an ASCII character is represented by itself, except the following characters:

```
New line (line feed)          HL (LF)    '\n'
Horizontal tabulation         HT         '\t'
Vertical tabulation           VT         '\v'
Backspace                     BS         '\b'
Carriage return               CR         '\r'
```

Form feed	FF	'\f'
Back slash	\	'\\'
Acute accent	'	'\''
Double quote	"	'\"'

A sequence of several character symbols may be written as a single string consisting of character literals and enclosed in acute accents. For instance:

```
'ABC'
'123'
'\"I don\'t like swimming!\" - said a little girl.'
```

Thus, the sequence of three character symbols 'A', 'B', and 'C' may be written in any of the following ways:

```
'A' 'B' 'C'
'A''B''C'
'ABC'
```

If a back slash \ followed by a new line (LF) appears in a character string literal, this back slash and the new line character are ignored, which enables long strings to be written on more than one line. For example:

```
'A\
BC'
```

3.5.WORD SYMBOLS

```
$    WordLiteral =
$         Identifier |
$         '"' { CharacterLiteral | ContinuationToNewLine } '"'.

$    Identifier = IdentifierHead IdentifierTail.
$    IdentifierHead = CapitalLetter | "!" | "?".
$    IdentifierTail = { Letter | Digit | "!" | "?" | "-" }.

$    Letter = CapitalLetter | SmallLetter.

$    CapitalLetter =
$         "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" |
$         "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" |
$         "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z".

$    SmallLetter =
$         "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |
$         "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" |
$         "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z".

$    Digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7"
$         | "8" | "9".
```

Each word symbol corresponds to an ASCII character string and is written as a sequence of character literals enclosed in double quotes. The character literals appearing in word symbols are the same as those appearing in character symbols. For example:

```
"ABC"
"123"
"\"I don\'t like swimming!\" - said a little girl."
```

Notice should be taken of the fact that "ABC" represents a single word symbol, whereas 'ABC' represents the sequence of three character symbols. Besides, a word symbols consisting of a single character is regarded as different from the character symbol consisting of the same character. For example, the character symbol 'A' is different from the word symbol "A".

The double quotes enclosing a word symbol may be omitted, provided that the symbol satisfies the two following restrictions.

First, the word symbol may contain only the following ASCII characters: capital letters, small letters, digits, exclamation marks, question marks, and minus signs.

Second, the first character of the word symbol must be either a capital letter, an exclamation mark, or a question mark.

If a word symbol is written without enclosing double quotes, it is case insensitive, i.e. all small letters are considered to be representations of the corresponding capital letters.

For example, here are three representations of the same word symbol:

```
I-do-not-like-swimming!
I-DO-NOT-LIKE-SWIMMING!
"I-DO-NOT-LIKE-SWIMMING!"
```

3.6.NUMERIC SYMBOLS

```
$     NumericLiteral = [ "+" | "-" ] Digit { Digit }.
```

Numeric symbols represent signed integers, which may be arbitrarily large. For example:

```
123   +121   -123   -123456789012345678901234567890
```

3.7.VARIABLES

```
$     Variable =
$          s-variable | t-variable | e-variable |
$          v-variable.
```

```
$     s-variable = "s" [ "." ] VariableIndex.
$     t-variable = "t" [ "." ] VariableIndex.
$     v-variable = "v" [ "." ] VariableIndex.
$     e-variable = "e" [ "." ] VariableIndex.

$     VariableTypeDesignator = "s" | "t" | "v" | "e".
$     VariableIndex = IdentifierTail.
```

A variable consists of a variable type designator followed by a variable index. The type designator and the index may be separated by an optional dot. For example:

```
tHead  eTail e.1 e1  tX s t e
```

Variable indices are case insensitive. For example, eI, e.I, ei, and e.i represent the same variable.

Adjacent variables must be separated. For example, sAeB is a single variable, whereas sA eB is a sequence of two variables.

The index of a variable may be omitted, which means that the index is unique and different from the indices of all other variables appearing in the program. Thus, for example, if the variables e1000 and e2000 do not appear in the program, the sequence e e may be replaced with e1000 e2000.

Variables are distinguished into four classes: s-variables, t-variables, v-variables, and e-variables, the class of a variable being determined by the type designator.

3.8. NORMALIZATION OF THE TOKEN STREAM

A program is scanned to break up the source character stream into tokens. Despite being different in form, many tokens have the same meaning. For example, all the three tokens

```
125  000125  +125
```

denotes the same number 125.

It is for this reason that the description of the lexical structure of programs deals with such terms as "numeric literal" and "word literal" rather that "number" and "word".

Besides, a token like "character string literal" represents a sequence of characters rather that a single syntax entity.

Thus, when describing the syntax, we assume the token stream to have been "normalized", each token having been reduced to its normal form, so that different tokens always represent different entities.

In addition we assume each character string literal to have been broken up into the string of separate tokens, a token representing a single character.

The above enables us to describe the syntax in terms of syntax "entities" rather than "representations of syntax entities".

Here is the correspondence between the source tokens and the normalized tokens:

```
CharacterStringLiteral   ==>
     Character1 Character2 ... CharacterN
WordLiteral              ==>  Word
NumericLiteral           ==>  Number
```

The character symbols obtained by scanning a program should not be confused with the characters appearing in the source text of the program. For example, parsing the three characters

```
'A'
```

results in producing a single character symbol.

4.OBJECTS AND VALUES

"Object" is usually understood to mean an entity that exists in time and may vary, but, nevertheless, does not lose its identity.

"Value" is usually understood to mean an entity that is unable to vary and, in a sense, exists out of time.

A value may, certainly, be regarded as a special, degenerate, case of object (i.e. as a rigid object unable to develop). Nevertheless, the term "object" will be usually applied only to "proper" objects, which are not values.

Since objects may vary, they are more difficult to deal with than values are. Thus objects are often provided with names. The basic property of names is that a name is unambiguously associated with an object (i.e. a name unambiguously identifies the object). In contrast to objects, their names are typical values, there being no changes in the names in spite of there being drastic changes in the objects.

Programs in Refal Plus deal with objects as well as values.

All values manipulated by Refal programs are ground expressions, which are finite sequences of symbols and parentheses, the parentheses being properly paired. Parentheses are used for giving a tree structure to ground expressions, whereas symbols represent basic data, such as characters, numbers, words, and references to objects.

Objects dealt with by a Refal program may contain ground expressions, which, in turn, may contain references to objects. The contents of objects may be modified by the Refal program, in which case the objects are accessed through their names, reference symbols.

Objects may be created at compile time as well as at run time. Theoretically, having been created, an object exists eternally. In practice, however, Refal programs are to be run by computers with limited memory capacity, thus all Refal implementations must include a garbage collector, whose purpose is to automatically destroy objects inaccessible to the program, and,

thus, unable to influence the program's behavior.

5.GROUND EXPRESSIONS

5.1.GROUND EXPRESSION SYNTAX

```
$     GroundExpression = { GroundTerm }.
$     GroundTerm = Symbol | "(" GroundExpression ")".
```

Henceforth, ground expressions will be denoted by Ge, ground terms by Gt, and symbols by Gs.

5.2.STATIC AND DYNAMIC SYMBOLS

```
$     Symbol = StaticSymbol | DynamicSymbol.
$     StaticSymbol = Character | Word | Number.
$     DynamicSymbol = ReferenceToFunction | ReferenceToTable |
$          ReferenceToBox | ReferenceToVector |
$          ReferenceToString | ReferenceToChannel.
```

The symbols are divided into two classes: static symbols and dynamic symbols.

A static symbol is either a character symbol, a word symbol, or a numeric symbol.

The static symbols exist "objectively": a static symbol may be written to an input/output channel, and then read back, the symbol read being the same as the symbol written. Thus, the static symbols, in a sense, exist before the program is run, and continue to exist after the program has been run.

A dynamic symbol is a reference to an object. This symbol contains a pointer to the memory location where the object resides at run time. The object may be either a function definition, a box, a vector, a string, a table, or a channel.

The dynamic symbols, in contrast to the static ones, exist "subjectively". A dynamic symbol is created either at the moment the program is loaded, or when the program is being executed. A dynamic symbol may be written into an input/output channel, but it cannot be read back. The execution of a program having been brought to completion, all dynamic symbols created during the execution lose any meaning.

A dynamic symbol is either a function reference, a box reference, a vector reference, a string reference, a table reference, or a channel reference.

5.3.SYMBOLIC EXPRESSION NAMES

```
$     NamedExpression = "&" Word.
```

Ground expressions appearing in a Refal Plus program may be given symbolic names (which are word symbols). If a word symbol

Gs denotes a ground expression Ge, the construct & Gs may be used instead of the expression Ge.

Since all references to objects as well as the objects are created when the program is compiled, loaded or executed, references cannot appear in the source program text as literals. Nevertheless, when an object is declared in a program, the references to the object are given a symbolic name, which may be used in the program for denoting the references.

5.4.ELIMINATION OF SYMBOLIC EXPRESSION NAMES

If a word Gs is a symbolic name for a ground expression Ge, all occurrences of the name in a program may be eliminated by replacing each construct &Gs with Ge.

Henceforth, when describing context dependent restrictions and the syntax of the language, we assume the above transformation to have been done and, thus, symbolic expression names not to appear in the program. On the other hand, the transformed program text may well contain dynamic symbols.

6.VARIABLE VALUES AND ENVIRONMENTS

To evaluate a Refal Plus construct, it is necessary to know the values of the variables appearing in the construct. The information about the variable values may be represented in a natural way by an environment, which is a function with finite domain that associates each variable from the domain with the variable's value.

We shall use the following notation.

Let Env be an environment with the domain {V1, ..., Vn}, Env(Vj) = Gej being the value the variable Vj is bound to. Then this environment is denoted by {V1 = Ge1, ..., Vn = Gen}. In particular, the empty environment is denoted by {}.

The domain of the environment Env is denoted by dom[Env]. Thus, dom[{V1 = Ge1, ..., Vn = Gen}] = {V1, ..., Vn}.

All environments are assumed to satisfy the requirement that a variable's value should be consistent with the type of the variable. Thus, an s-variable's value must be a symbol, a t-variable's value must be a ground term, an e-variable's value must be a ground expression, and a v-variable's value must be a non-empty ground expression.

Env+Env' denotes the environment Env extended with the bindings from the environment Env' in the following way.

dom[Env+Env'] contains the variables from dom[Env'], as well as the variables from dom[Env], whose indices are different from the indices of the variables from dom[Env'].

For all V in dom[Env+Env'], if Env'(V) is defined, then (Env+Env')(V) = Env'(V), otherwise, if Env'(V) is undefined, then (Env+Env')(V) = Env(V).

For example,

```
{sX = 1, sY = 2} + {sY = 200, sZ = 300}
     = {sX = 1, sY = 200, sZ = 300}

{sX = 1, sY = 2} + {eY = 200, sZ = 300}
     = {sX = 1, eY = 200, sZ = 300}
```

7.RESULT EXPRESSIONS

7.1.SYNTAX

```
$     ResultExpression =
$          { ResultTerm | NamedExpression }.
$     ResultTerm =
$          StaticSymbol | Variable |
$          "(" ResultExpression ")" |
$          FunctionCall.
$     FunctionCall =
$          "<" FunctionName CallArgument ">".
$     CallArgument =
$          ResultExpression.
```

If two different variables appear in the same result expression, they must have different indices.

Henceforth, result expressions will be denoted by Re, result terms by Rt, variables by V, e-variables by Ve, and function names by Fname.

7.2.EVALUATION OF RESULT EXPRESSIONS

A result expression Re may be evaluated in an environment Env, on condition that Env provides values for all variables appearing in Re.

If the evaluation of Re terminates, it results in producing either a ground expression Ge, a failure $fail(0), or an error $error(Ge), where Ge is an error message.

Evaluating a function call may result in the global program state being changed (for example, if it involves input/output or some manipulations with objects). Hence, if a result expression contains function calls, evaluating the expression may also result in the global state being changed.

A judgment Env,St |- Re => X,St' means that the result of evaluating the result expression Re in the environment Env is X, and if the evaluation starts in the global state St, it terminates in the global state St'.

A result expression Re is evaluated from left to right, the variables being replaced with their values, and the function calls being executed.

The evaluation of a function call <Fname Re> begins by

evaluating the result expression Re. If a ground expression Ge is returned, the function Fname is applied to Ge.

A judgment St |- <Fname Ge> => X,St' means that the result of applying the function Fname to the ground expression Ge is X, and if the evaluation starts in the global state St, it terminates in the global state St'.

```
Env,St |-  =>  ,St


Env,St |- Re => Ge',St'
Env,St'|- Rt => Ge",St"
-------------------------------
Env,St |- Re Rt => Ge' Ge",St"


Env,St |- Re => Ge',St'
Env,St'|- Rt => $fail(0),St"
--------------------------------
Env,St |- Re Rt => $fail(0),St"


Env,St |- Re => Ge',St'
Env,St'|- Rt => $error(Ge"),St"
-----------------------------------
Env,St |- Re Rt => $error(Ge"),St"


Env,St |- Re => $fail(0),St'
--------------------------------
Env,St |- Re Rt => $fail(0),St'


Env,St |- Re => $error(Ge'),St'
-----------------------------------
Env,St |- Re Rt => $error(Ge'),St'


Env,St |- Gs => Gs,St

Env,St |- V => Ge,St
     gde Ge=Env(V).


Env,St |- Re => Ge,St'
--------------------------
Env,St |- (Re) => (Ge),St'


Env,St |- Re => $fail(0),St'
-------------------------------
Env,St |- (Re) => $fail(0),St'
```

```
Env,St |- Re => $error(Ge),St'
-------------------------------
Env,St |- (Re) => $error(Ge),St'


Env,St |- Re => Ge,St'
St' |- <Fname Ge> => X,St"
----------------------------
Env,St |- <Fname Re> => X,St"


Env,St |- Re => $fail(0),St'
-----------------------------------
Env,St |- <Fname Re> => $fail(0),St'


Env,St |- Re => $error(Ge),St'
-------------------------------------
Env,St |- <Fname Re> => $error(Ge),St'
```

7.3. EXAMPLES

Here are examples of result expressions:

```
(A B) C D
t.Head e.Tail
While t.Condition Do t.Statement
<"*" sN <Factorial <"-" sN 1>>
```

The following result expressions are result terms:

```
(A B)
t.Head
<"*" sN <Factorial <"-" sN 1>>
```

Let Env1 = {sM = 2, sN = 3, eA = A B C, tB = (D E F)}, "+" be the name of the function that adds integers, and "*" be the name of the function that multiplies integers. Thus, the judgments

```
St |- <"+" 3 100> => 103, St
St |- <"*" 2 103> => 206, St
```

hold for any global state St, because the functions "+" and "*" do not change the global state.

Then we have

```
Env1,St |- eA (eA tB) tB =>
                        A B C (A B C (D E F)) (D E F), St
Env1,St |- <"*" sM <"+" sN 100>> => 206, St
```

8.PATTERNS

8.1.SYNTAX

```
$    Pattern = DirectionDesignator PatternExpression.
$    DirectionDesignator = [ "$l" | "$r" ].

$    PatternExpression =
$         { PatternTerm | NamedExpression }.
$    PatternTerm =
$         StaticSymbol | Variable |
$         "(" PatternExpression ")".
```

A pattern is a pattern expression, which may be preceded by a direction designator "$l" or "$r". The designator "$l" indicates that the pattern matching must be done from left to right. The designator "$r" indicates that it must be done from right to left. If the direction designator is omitted, the designator "$l" is implied.

If two different variables appear in the same pattern expression, they must have different indices.

Henceforth, patterns are denoted by P, pattern expressions by Pe, pattern terms by Pt, and direction designators by D.

8.2.PATTERN MATCHING

An environment Env is said to be a result of matching a ground expression Ge against a pattern P in an initial environment Env0, if the following conditions holds.

(1) The environment Env is an extension of Env0, i.e. dom[Env] includes dom[Env0], and for all V in dom[Env0] holds Env(V)=Env0(V).

(2) If each variable V appearing in P is replaced with Env(V), and the direction designator is removed, the ground expression thus obtained is Ge.

This environment Env is said to be a variant of matching Ge against P in the environment Env0, and the set of such variants of matching is denoted by Match(Env0,Ge,P).

The set Match(Env0,Ge,P) is assumed to be equipped with the order relation defined by the following rules.

Suppose Match(Env0,Ge,P) contains two different variants of matching Env1 and Env2. Consider all occurrences of variables in P.

If P has the direction designator $l, find the leftmost occurrence that is given different values by the matching variants Env1 and Env2.

If P has the direction designator $r, find the rightmost occurrence that is given different values by the matching vari-

ants Env1 and Env2.

Suppose the occurrence found is an occurrence of a variable V. Then compare Env1(V) to Env2(V). If Env1(V) is shorter than Env2(V), then Env1 is taken to precede Env2. Otherwise Env2 is taken to precede Env1.

A finite sequence of environments Env1, Env2, ..., Envn is written as [Env1, Env2, ... , Envn], and the empty sequence as [].

[Env0]^[Env1,...,Envn] denotes [Env0, Env1,..., Envn].

A judgment Env0 |- Ge : P => [Env1,...Envn] means that Match(Env0,Ge,P) = {Env1,...,Envn}, where Envi precedes Envj for all i<j.

8.3.EXAMPLES

Here are examples of patterns:

```
t.Head e.Tail
eX (eY)
eA '+' eB
$l eA '+' eB
$r eA '+' eB
```

Here are examples of pattern matching:

```
{} |- A () C D E : $l sX tY tZ e1
     => [ {sX = A, tY = (), tZ = C, e1 = D E} ]

{} |- 1 2 3 : $l eA eB =>  [
                    {eA = ,          eB = 1 2 3},
                    {eA = 1,         eB = 2 3},
                    {eA = 1 2,       eB = 3},
                    {eA = 1 2 3,     eB = }   ]

{} |- 1 2 3 : $r eA eB => [
                    {eA = 1 2 3,     eB =        },
                    {eA = 1 2,       eB = 3      },
                    {eA = 1,         eB = 2 3    },
                    {eA = ,          eB = 1 2 3 }  ]

{eA = 1 2} |- $l 1 2 3 4 5 : eA eB
     => [ {eA = 1 2, eB = 3 4 5} ]
```

9.HARD EXPRESSIONS

9.1.SYNTAX

```
$    HardExpression =
$         { HardCorner } |
$         { HardCorner } e-variable { HardCorner } |
$         { HardCorner } v-variable { HardCorner }.
$    HardCorner = { HardTerm | NamedExpression }.
```

```
$    HardTerm =
$         StaticSymbol | s-variable | t-variable |
$         "(" HardExpression ")".
```

Thus, any subexpression of a hard expression may contain no more that one occurrence of e-variable or v-variable at the top level of parentheses.

A variable may appear in a hard expression no more that once. If two different variables appear in the same hard expression, they must have different indices.

Henceforth, hard expressions are denoted by He, and hard terms by Ht.

9.2.MATCHING AGAINST HARD EXPRESSIONS

Hard expressions may be regarded as a particular case of pattern expressions. A feature of hard expressions is that there can exist no more that one way of matching a ground expression Ge against a hard expression He. Thus there holds either {} |- Ge : He => [ ] or {} |- Ge : He => [Env].

A judgment Env |- Ge :: He => Env' means that {} |- Ge : He => Env" and Env' = Env+Env". Consequently, Env' is produced from Env in the following way. First, Ge is matched against He in the empty environment. Thus, the variable values provided by the current environment Env are not taken into account. The environment Env" thus obtained contains bindings for all variables appearing in He. Then the original environment Env is extended with the bindings from Env" to produce the final environment Env'.

9.3.EXAMPLES

Here are example hard expressions:

```
t.Head e.Tail
sX (eY) eZ (A eA)
```

Here are examples of matching hard expressions

```
{sX = XXX, eA = A B C}  |-  X Y Z :: sY eA
     =>  {sX = XXX, eA = Y Z, sY = X}

{sX = XXX, eA = A B C}  |-  X Y Z :: eA sY
     =>  {sX = XXX, eA = X Y, sY = Z}
```

10.PATHS

10.1.SYNTAX

```
$    Path =
```

```
$           Condition | Binding | Search | Match |
$           Rest | Source.

$      Condition =
$           Source Rest.
$      Binding =
$           Source "::" HardExpression [ Rest ].
$      Search =
$           Source "$iter" Source
$                 [ "::" HardExpression ] [ Rest ].
$      Match =
$           Source ":" Pattern [ Rest ].

$      Rest =
$           DelimitedPath | NegativeCondition |
$           Fence | Cut |
$           Failure | RightHandSide | ErrorGenerator |
$           ErrorTrap.

$      DelimitedPath =
$           "," Path.
$      NegativeCondition =
$           "#" Source [ Rest ].
$      Fence =
$           "\?" Path.
$      Cut =
$           "\!" Path.
$      Failure =
$           "$fail".
$      RightHandSide =
$           "=" Path.
$      ErrorGenerator =
$           "$error" Path.
$      ErrorTrap =
$           "$trap" Path "$with" PatternAlternative.

$      Source =
$           Alternative | AlternativeMatch | ResultExpression.

$      Alternative =
$           "\{" PathList "}" |
$            "{" PathList "}".

$      AlternativeMatch =
$           Source ":" PatternAlternative.

$      PatternAlternative =
$           "\{" SentenceList "}" |
$           " {" SentenceList "}" |

$      PathList = { Path ";" }.

$      SentenceList =
```

```
$          { Sentence ";" }.

$    Sentence = Pattern [ Rest ].
```

Henceforth, paths are denoted by Q, rests by R, sources by S, pattern alternatives by Palt, and sentences by Snt.

The syntax of paths seems to be rather complicated. This is due to the desire to save the user the trouble of writing redundant delimiters without real necessity.

This is achieved by distinguishing two particular cases of paths: "rests" and "sources", which possess some useful syntax properties. Rests begin with key words, which are easy to recognize. Thus, if a result expression or a pattern is followed by a rest, there is no danger that they could "stick" together. Sources cannot contain commas at the top level of curly brackets, for which reason they can be unambiguously separated from the constructs they are followed by.

10.2.EVALUATION OF PATHS

A path Q is evaluated with respect to an environment Env and a non-negative integer m. The environment Env associates variables with their values, which may be necessary to evaluate the path. The integer m, which is referred to as the "level" of the path, specifies the number of fences "\?" that surrounds Q without being closed by cuts "\!".

If the evaluation of Q terminates, it returns either a ground expression Ge, a failure $fail(k), the non-negative integer k being the "level" of the failure, or an error $error(Ge), Ge being an error message.

If evaluating a path at the level m returns a failure $fail(k), the failure level is certain to satisfy the restriction 0 <= k <= m+1. In particular, if a path is evaluated at the level 0, there holds either k=0 or k=1.

A judgment Env,m,St |- Q => X,St' means that evaluating the path Q in the environment at the level m returns X, and if the evaluation of Q starts in the global state St, it terminates in the global state St'.

Rests and sources are particular cases of paths, for which reason the above notation is also used for describing the evaluation of rests and sources.

The meaning of a path Q can often be reduced to the meaning of other path Q'. To put it more exactly, the evaluation of Q is done by evaluating Q', and the result thus obtained is taken to be the result of evaluating Q. This may be formulated by means of the following inference rule:

```
Env,m,St |- Q' => X,St'
-----------------------
Env,m,St |- Q => X,St'
```

Such rules are rather frequent, for which reason they will

be abbreviated in the following way:

```
Q =>=> Q'
```

10.3.CONDITIONS

The evaluation of a path S R proceeds as follows. The source S is evaluated, and, if the evaluation succeeds, the rest R is evaluated.

The source S is considered to be at the zero level.

```
Env,0,St |- S => ,St'
Env,m,St'|- R => X,St"
------------------------
Env,m,St |- S R => X,St"
```

```
Env,0,St |- S => $fail(k),St'
-------------------------------
Env,m,St |- S R => $fail(0),St'
```

```
Env,0,St |- S => $error(Ge),St'
---------------------------------
Env,m,St |- S R => $error(Ge),St'
```

10.4.BINDINGS

The evaluation of a path S :: He R proceeds as follows. The source S is evaluated, and, if the evaluation succeeds, the ground expression obtained is matched against He. The variables from He are bound to new values, and the environments is extended with the new bindings. Then the tail R is evaluated in the new environment.

The source S is considered to be at the zero level.

If the rest R is an empty delimited path (which always returns an empty ground expression), it may be omitted.

```
S :: He  =>=>  S :: He ,
```

```
Env,0,St |- S => Ge,St'
Env |- Ge :: He => Env'
Env',m,St' |- R => X,St"
-----------------------------
Env,m,St |- S :: He R => X,St"
```

```
Env,0,St |- S => $fail(k),St'
-------------------------------------
Env,m,St |- S :: He R => $fail(0),St'


Env,0,St |- S => $error(Ge),St'
---------------------------------------
Env,m,St |- S :: He R => $error(Ge),St'
```

For example, the evaluation of the path

```
100 :: sN, <"+" sN 1> :: sN = sN
```

produces the number 101.

10.5. SEARCHES

The goal of evaluating the path

```
S" $iter S' :: He R
```

is to find such values for the variables appearing in He that the evaluation of R succeeds, in which case the result obtained is taken to be the result of evaluating the whole construct.

An empty hard expression He may be omitted along with the key word "::". If the rest R is an empty delimited path (which always returns an empty ground expression), it may be omitted.

```
S" $iter S'  =>=>  S" $iter S' :: ,

S" $iter S' R  =>=>  S" $iter S' :: R

S" $iter S' :: He  =>=>  S" $iter S' :: He ,
```

The initial values for the variables appearing in He are obtained by evaluating the source S", whereas the evaluation of S' enables the new variable values to be obtained from the previous ones.

The sources S" and S' are considered to be at the zero level.

The search for the variable values proceeds as follows. First, the initial variable values are found by evaluating the source S" and matching the ground expression Ge obtained against the pattern He. Then an attempt is made to evaluate the rest R in the new environment. If the value returned is a failure $fail(0), then S' is evaluated and a ground expression obtained is matched against He, and then a new attempt is made to evaluate R, etc.

```
S" $iter S' :: He R  =>=>
          S" :: He, \{ R; S' $iter S' :: He R; }
```

For example, if the values of the variables eA and eB are not defined in the current environment, the match

```
 eX : $l eA eB,
  <Writeln eA>, <Writeln eB> $fail
```

is equivalent to the search

```
()(eX)
  $iter \{ eB : t1 e2 = (eA t1)(e2); }
    :: (eA)(eB),
  <Writeln eA>, <Writeln eB> $fail
```

10.6.MATCHES

The evaluation of a path S : P R proceeds as follows. The source S is evaluated and a ground expression Ge obtained is matched against the pattern P to produce a sequence of the variants of matching. Then an attempt is made to find the first variant of matching appearing in this sequence such that the evaluation of the rest R succeeds.

If the rest R is an empty delimited path (which always returns an empty ground expression), it may be omitted.

To describe the semantics of matches, we need the following notation. A judgment EnvList,m,St ||- Q => X,St' means that the evaluation of the path Q at the level m with the list of environments EnvList returns X.

```
Env,m,St |- Q => Ge,St'
----------------------------------
[Env]^EnvList,m,St ||- Q => Ge,St'


Env,m,St |- Q => $fail(0),St'
EnvList,m,St' ||- Q => X,St"
---------------------------------
[Env]^EnvList,m,St ||- Q => X,St"


Env,m,St |- Q => $fail(k+1),St'
-----------------------------------------
[Env]^EnvList,m,St ||- Q => $fail(k+1),St'


Env,m,St |- Q => $error(Ge),St'
-----------------------------------------
[Env]^EnvList,m,St ||- Q => $error(Ge),St'
```

```
[],m,St ||- Q => $fail(0),St.
```

Now we describe the semantics of matches.

```
S : P  =>=>  S : P ,


Env,0,St |- S => Ge,St'
Env |- Ge : P => EnvList
EnvList,m,St' ||- R => X,St"
----------------------------
Env,m,St |- S : P R => X,St"


Env,0,St |- S => $fail(k),St'
-----------------------------------
Env,m,St |- S : P R => $fail(0),St'


Env,0,St |- S => $error(Ge),St'
-------------------------------------
Env,m,St |- S : P R => $error(Ge),St'
```

For example, the evaluation of the following path fails, which results in the character string 'CBA' being printed.

```
'ABC' : $r e sX e, <Print sX> $fail
```

10.7.DELIMITED PATHS

Evaluating the rest

```
     , Q
```

always produces the same result as the evaluation of the path Q.

```
, Q  =>=>  Q
```

10.8.NEGATIVE CONDITIONS

The evaluation of a rest `# S R` proceeds as follows. The source S is evaluated. If the result obtained is an empty ground expression, the evaluation of the whole construct fails, but, if the result is a failure, the rest R is evaluated, and the result obtained is taken to be the result of the whole construct. Thus, this construct enables us to test the "logical negation" of the condition S.

If the rest R is an empty delimited path (which always

returns an empty ground expression), it may be omitted.

```
# S  =>=>  # S ,


Env,0,St |- S => ,St'
---------------------------------
Env,m,St |- # S R => $fail(0),St'


Env,0,St |- S => $fail(k),St'
Env,m,St' |- R => X,St"
-----------------------------
Env,m,St |- # S R => X,St"


Env,0,St |- S => $error(Ge),St'
-----------------------------------
Env,m,St |- # S R => $error(Ge),St'
```

10.9.FENCES

The evaluation of a rest \? Q proceeds as follows. The path Q is evaluated. If the result obtained is a failure $fail(k), where k>0, then the "weakened" failure $fail(k-1) is taken to be the result of the whole construct. Otherwise, the result of evaluating Q is taken to be the result of the whole construct.

The path Q is evaluated at the level m+1, where m is the level at which the whole construct is evaluated.

```
Env,m+1,St |- Q => Ge,St'
--------------------------
Env,m,St |- \? Q => Ge,St'


Env,m+1,St |- Q => $fail(0),St'
--------------------------------
Env,m,St |- \? Q => $fail(0),St'


Env,m+1,St |- Q => $fail(k+1),St'
---------------------------------
Env,m,St |- \? Q => $fail(k),St'


Env,m+1,St |- Q => $error(Ge),St'
----------------------------------
Env,m,St |- \? Q => $error(Ge),St'
```

10.10.CUTS

The evaluation of the rest \! Q proceeds as follows. The path Q is evaluated. If the result obtained is a failure $fail(k), then the "strengthened" failure $fail(k+1) is taken to be the result of the whole construct. Otherwise, the result of evaluating Q is taken to be the result of the whole construct.

The path Q is evaluated at the level m-1, where m is the level at which the whole construct is evaluated.

```
Env,m,St |- Q => Ge,St'
----------------------------
Env,m+1,St |- \! Q => Ge,St'


Env,m,St |- Q => $fail(k),St'
------------------------------------
Env,m+1,St |- \! Q => $fail(k+1),St'


Env,m,St |- Q => $error(Ge),St'
------------------------------------
Env,m+1,St |- \! Q => $error(Ge),St'
```

For example, the evaluation of the following path results in the character string 'ABD' being printed, and the result '2' being returned.

```
{
  \? {
    <Print 'A'> $fail;
    <Print 'B'> \! $fail;
    <Print 'C'> = '1';
    };
  <Print 'D'> = '2';
  }
```

10.11.FAILURES

The evaluation of the rest $fail always returns the failure $fail(0).

```
Env,m,St |- $fail => $fail(0),St
```

10.12.RIGHT HAND SIDES

The evaluation of a rest = Q at a level m proceeds as follows. The path Q is evaluated at the level 0. If the result obtained is a failure $fail(k), then the whole construct returns the failure $fail(m+1), which is so strong as to overcome all surrounding fences that are not neutralized by cuts.

```
Env,0,St |- Q => Ge,St'
-------------------------
Env,m,St |- = Q => Ge,St'
```

```
Env,0,St |- Q => $fail(k),St'
---------------------------------
Env,m,St |- = Q => $fail(m+1),St'
```

```
Env,0,St |- Q => $error(Ge),St'
---------------------------------
Env,m,St |- = Q => $error(Ge),St'
```

10.13.ERROR GENERATORS

The evaluation of a rest $error Q returns an error $error(Ge), where Ge is the result of evaluating the path Q.

```
Env,0,St |- Q => Ge,St'
--------------------------------------
Env,m,St |- $error Q => $error(Ge),St'
```

```
Env,0,St |- Q => $fail(0),St'
--------------------------------------------------------------
Env,m,St |- $error Q => $error(Fname "Unexpected fail"),St'
     Fname is the name of the function in which the con-
     struct appears.
```

```
Env,0,St |- Q => $error(Ge),St'
--------------------------------------
Env,m,St |- $error Q => $error(Ge),St'
```

10.14.ERROR TRAPS

The evaluation of a rest

```
     $trap Q $with Palt
```

proceeds as follows. An attempt is made to evaluate the path Q. If the result obtained is an error $error(Ge), then the alternative match

```
          Ge : Palt
```

is evaluated, and the result obtained is taken to be the result of the whole construct.

The path Q is evaluated at the level 0.

```
Env,0,St |- Q => Ge,St'
----------------------------------------
Env,m,St |- $trap Q $with Palt => Ge,St'
```

```
Env,0,St |- Q => $fail(k),St'
Env,m,St' |- Fname "Unexpected fail" : Palt => X,St"
----------------------------------------------------
Env,m,St |- $trap Q $with Palt => X,St"
```

Fname is the name of the function in which the construct appears.

```
Env,0,St |- Q => $error(Ge),St'
Env,m,St' |- Ge : Palt => X,St"
---------------------------------------
Env,m,St |- $trap Q $with Palt => X,St"
```

10.15.ALTERNATIVES

The evaluation of a source \{Q1; Q2; ... Qn;} proceeds as follows. The paths Q1, Q2, ..., Qn are evaluated from left to right until the evaluation of a path succeeds.

More specifically, consider the result of evaluating the path Qj.

If the result is a ground expression Ge, then Ge is taken to be the result of the whole construct. If the result is $error(Ge), then $error(Ge) is the result of the whole construct. If the result is $fail(k+1), then $fail(k+1) is the result of the whole construct. And, finally, if the result is $fail(0), this failure is "caught", i.e. an attempt is made to evaluate the next path. If there exists no next path (i.e. j=n), the failure $fail(0) is the result of the whole construct.

An alternative {Q1; Q2; ... Qn;} is equivalent to the alternative \{ Q1; Q2; ... Qn; $error(Fname "Unexpected fail"); }, where Fname is the name of the function in which the construct appears.

```
{Q1; Q2; ... Qn;}  =>=>
```

```
          \{Q1; Q2; ... Qn;
               $error(Fname "Unexpected fail");}
     Fname is the name of the function in which the con-
     struct appears.
```

```
Env,m,St |- \{} => $fail(0),St


Env,m,St |- Q1 => Ge,St'
----------------------------------------
Env,m,St |- \{Q1; Q2; ... Qn;} => Ge,St'


Env,m,St |- Q1 => $fail(0),St'
Env,m,St'|- \{Q2; ... Qn;} => X,St"
---------------------------------------
Env,m,St |- \{Q1; Q2; ... Qn;} => X,St"


Env,m,St |- Q1 => $fail(k+1),St'
-----------------------------------------------
Env,m,St |- \{Q1; Q2; ... Qn;} => $fail(k+1),St'


Env,m,St |- Q1 => $error(Ge),St'
-----------------------------------------------
Env,m,St |- \{Q1; Q2; ... Qn;} => $error(Ge),St'
```

10.16.ALTERNATIVE MATCHES

The evaluation of a source S : \{Snt1; ... Sntn;} always produces the same result as the evaluation of the path S : Ve, \{Ve : Snt1; ... Ve : Sntn;}, provided that Ve is an e-variable that does not appear in the program in other places.

A source S : {Snt1; ... Sntn;} is equivalent to the source S : \{Snt1; ... Sntn; e $error(Fname "Unexpected fail"); }, where Fname is the name of the function in which the construct appears.

```
S : {Snt1; ... Sntn;}  =>=>
          S : \{Snt1; ... Sntn;
                    e $error(Fname "Unexpected fail");}
     Fname is the name of the function in which the con-
     struct appears.
```

```
Env,0,St |- S => Ge,St'
Env,m,St'|- \{Ge : Snt1; ... Ge : Sntn;} => X,St"
-------------------------------------------------
```

```
Env,m,St |- S : \{Snt1; ... Sntn;} => X,St"
```

```
Env,0,St |- S => $fail(k),St'
--------------------------------------------------
Env,m,St |- S : \{Snt1; ... Sntn;} => $fail(0),St'
```

```
Env,0,St |- S => $error(Ge),St'
----------------------------------------------------
Env,m,St |- S : \{Snt1; ... Sntn;} => $error(Ge),St'
```

10.17.RESULT EXPRESSIONS AS SOURCES

A source of the form Re, where Re is a result expression, is evaluated by evaluating the result expression Re. The result thus obtained is taken to be the result of the source.

```
Env,St |- Re => X,St'
-----------------------
Env,m,St |- Re => X,St'
```

11.FUNCTION DEFINITIONS

```
$     FunctionDefinition =
$          FunctionName FunctionBody ";".
$     FunctionBody =
$          PatternAlternative | Sentence.
```

A function's definition binds the function's name to the function's body, which is a construct that describes the way in which the function is to be evaluated.

A function definition of the form Fname Snt; is equivalent to the definition Fname \{ Snt; }; .

Let the definition of a function Fname be of the form

Fname Palt

Then evaluating a function call <Fname Ge> amounts to evaluating the source Ge : Palt. If the result obtained is a ground expression Ge' or an error $error(Ge'), it is taken to be the result of evaluating the call. Otherwise, if the result is a failure $fail(k), the following actions depend on the function Fname having been declared either failing or unfailing. If the function is a failing one, the result returned is $fail(0), otherwise, if the function is an unfailing one, the result returned is $error(Fname "Unexpected fail").

```
{},0,St |- Ge : Palt => Ge',St'
```

```
     -------------------------------
     St |- <Fname Ge> => Ge',St'


     {},0,St |- Ge : Palt => $fail(k),St'
     ------------------------------------
     St |- <Fname Ge> => $fail(0),St'
          where the function Fname is a failing one,
          i.e. it has been declared as
          $func? Fname Farg = Fres;.


     {},0,St |- Ge : Palt => $fail(k),St'
     -----------------------------------------------------------
     St |- <Fname Ge> => $error(Fname "Unexpected fail"),St'
          where the function Fname is an unfailing one,
          i.e. it has been declared as
          $func Fname Farg = Fres;.


     {},0,St |- Ge : Palt => $error(Ge'),St'
     ---------------------------------------
     St |- <Fname Ge> => $error(Ge'),St'
```

The above inference rules assume the function Fname to have the definition Fname Palt.

12.DECLARATIONS

12.1.CONSTANT DECLARATIONS

```
$     ConstantDeclaration =
$          "$const" [ ConstDecl { "," ConstDecl } ] ";".
$     ConstDecl = ExpressionName "=" ConstantExpression.
$     ConstantExpression =
$          { ConstantTerm | NamedExpression }.
$     ConstantTerm =
$          StaticSymbol | "(" ConstantExpression ")".
$     ExpressionName = Word.
```

Constant declarations enable ground expressions to be denoted by symbolic names to be used instead of the expressions. A symbolic name is a word symbol. If a ground expression has been given a name N, the construct &N is a representation of the expression Ge. For example, the declaration

```
$const LF = 10, CR = 13, "***" = A B C;
```

gives names to three ground expressions, so that &LF denotes 10, &CR denotes 13, and &"***" denotes A B C.

A constant definition may contain references to previous declarations of constants, boxes, tables, channels, and functions. For example, the declaration

```
$const "CR-LF-***" = &CR &LF &"***";
```

gives a name to an expression, so that &"CR-LF-***" stands for 13 10 A B C.

12.2.OBJECT DECLARATIONS

```
$     BoxDeclaration       = "$box"        { ReferenceName } ";".
$     VectorDeclaration    = "$vector"     { ReferenceName } ";".
$     StringDeclaration    = "$string"     { ReferenceName } ";".
$     TableDeclaration     = "$table"      { ReferenceName } ";".
$     ChannelDeclaration   = "$channel"    { ReferenceName } ";".

$     ReferenceName = Word.
```

An object declaration associates symbolic names with references to boxes, vectors, strings, tables, and channels. These objects are to be created at the moment the program is loaded.

The symbolic names introduced by an object declaration may be used for getting references to the objects declared.

For example, the declaration $box X; makes the construct &X denote a reference to a box. Here are examples of object declarations:

```
$box B1 B2 B3;
$vector V1 V2;
$table T1 T2;
$channel Input Output;
```

12.3.FUNCTION DECLARATIONS

```
$     FunctionDeclaration =
$          "$func"   FunctionName
$               InputFormat "=" OutputFormat ";" |
$          "$func?"  FunctionName
$               InputFormat "=" OutputFormat ";".
$     FunctionName = Word.
$     InputFormat = FormatExpression.
$     OutputFormat = FormatExpression.

$     FormatExpression = HardExpression.
```

A function declaration introduces a function name. The declaration of a function must precede all references to the function as well as the definition of the function.

The declaration of a function imposes restrictions on the forms that can take the calls to the function, the input pat-

terns in the function's definition and the result expressions producing the values returned by the function. These restrictions will be described in detail below.

The input and output formats must be hard, i.e. any subexpression of a format expression may contain no more than one e-variable or v-variable.

The variable indices appearing in formats serve as comments, thus they have no effect on the meaning of the program and may be omitted.

It should be noted that the format expressions and the hard expressions are considered to be different constructs, despite their having the same context-free syntax. This is due to the differences in the interpretation of variable indices.

If the declaration of a function begins with the key word $func, the function is an unfailing one, i.e. evaluating a call to the function may result in returning either a ground expression or an error.

If the declaration of a function begins with the key word $func?, the function is a failing one, i.e. evaluating a call to the function may result in returning either a ground expression, a failure, or an error.

Here are function declarations

```
$func  Interpreter (e.Program) (e.Input) = e.Result;
$func? Attempt t.Arg = s.Result1 t.Result2 (e.Result3);
```

12.4.TRACE DIRECTIVES

```
$    TraceDirective =
$         "$trace" { FunctionName } ";" |
$         "$traceall" ";".
```

A directive "$trace" specifies that some debugging information is to be printed at the run time about the functions listed in the directive. When a function is called, its name is printed as well as the arguments passed. Then, when the call has been evaluated, the function name is printed as well as the results returned by the function.

A directive "$traceall" specifies that the debugging information is to be printed about all the functions whose definitions appear below the directive.

13.CONTEXT DEPENDENT RESTRICTIONS

13.1.ELIMINATION OF REDUNDANT CONSTRUCTS

This section describes different context dependent restrictions that must be satisfied by any program written in Refal Plus.

In order for the description to be concise, the program is

supposed to have been normalized, which means that all constructs considered to be abbreviations for other constructs have been replaced with their expansions.

The normalization is performed as follows.

The empty hard expressions and empty delimited paths omitted in bindings, searches, and matches are restored.

```
S :: He                =>        S :: He ,
S' $iter S"            =>        S' $iter S" ::  ,
S' $iter S" :: He      =>        S' $iter S" :: He ,
S' $iter S" R          =>        S' $iter S" ::  R
S : P                  =>        S : P ,
# S                    =>        # S ,
```

The empty delimited paths omitted in sentences are restored.

```
P                      =>        P ,
```

The "opaque" curly brackets "{" appearing in alternatives and alternative matches are replaced with the "transparent" curly brackets "\{".

```
{Sntl; ... Sntn;}    =>
     \{Sntl; ... Sntn;
               $error(Fname "Unexpected fail");}

S : {Sntl; ... Sntn;}    =>
     S : \{Sntl; ... Sntn;
               e $error(Fname "Unexpected fail");}
```

where Fname is the name of the function in which the construct appears.

The "opaque" curly brackets "{" appearing in function definitions are replaced with the "transparent" curly brackets "\{".

```
Fname {Sntl; ... Sntn;}  =>
     Fname \{Sntl; ... Sntn;
               Farg $error(Fname "Unexpected fail");}
```

where Farg is the input format provided by the declaration of the function Fname (the variable indices in function declarations are supposed to be omitted).

The function bodies consisting of a sentence are replaced with the corresponding pattern alternatives.

```
Fname Snt;      =>   Fname \{ Snt; };
```

13.2.RESTRICTIONS IMPOSED BY FUNCTION DECLARATIONS

A function declaration may have either of the two forms

```
$func  Fname Farg = Fres;
$func? Fname Farg = Fres;
```

where Farg is the input format of the function, i.e. a format expression which specifies the structure of the function's argument, and Fres is the output format of the function, i.e. a format expression which specifies the structure of the function's result. As mentioned previously the variable indices appearing in formats are of no significance, for which reason, without any loss of generality, they will be supposed to have been omitted.

The definition of a function must satisfy the restrictions imposed by the input and output formats of the function. To formulate these restrictions, we assume the set of format expressions to be equipped with the following partial ordering.

Let F1 and F2 be two formats. Then F2>>F1 means that the format F1 is an instance of the format F2. The relation >> is defined by the following rules.

(0) F >> F.
(1) If F1' >> F1 and F2' >> F2, then F1' F2' >> F1 F2.
(2) If F1 >> F2, then (F1) >> (F2).
(3) e >> F.
(4) If F is not of the form e e ... e, then v >> F.
(5) t >> Gs, for all symbols Gs.
(6) t >> s.
(7) t >> (F).
(8) s >> Gs, for all symbols Gs.

Now consider a program written in Refal Plus and the constructs appearing in the program.

Let Re be a result expression appearing in the program. A format expression F is said to be the format of Re, if F can be produced from Re by the following transformations.

(1) The indices of all variables appearing in Re are discarded.

(2) All function calls appearing in Re are replaced with the output formats of the corresponding functions. In other words, suppose that a function Fname has been declared as either $func Fname Farg = Fres; or $func? Fname Farg = Fres; . Then each call <Fname Re'> is replaced with Fres.

Let P be a pattern appearing in the program. A format expression F is said to be the format of P, if F can be produced from P by the following transformations.

(1) The indices of all variables appearing in P are discarded.

(2) If P has a direction designator, the designator is

discarded.

Let He be a hard expression appearing in the program. A format expression F is said to be the format of He, if F can be produced from P by discarding the indices of all variables appearing in He.

Henceforth, the format of a result expression will be denoted by form[Re], the format of a pattern P by form[P], and the format of a hard expression He by form[He].

It should be emphasized that not only does the format of a result expression Re depend on the appearance of Re, but it also depends on the output formats of the functions called in Re. Nevertheless, given a particular program, the meaning of form[Re] is unambiguously defined.

Now we can formulate the restrictions that must be met by the function definitions. These restrictions are imposed on the function calls, the input patterns in the function definitions, and the results returned by the paths.

Suppose the declaration of a function Fname contains the input format Farg, the output format Fres, whereas the pattern alternative Palt appearing in the function definition

```
          Fname Palt
```

has the form \{P1 R1; ... Pn Rn;}.

Then the following conditions must be satisfied.

The function's input patterns P1, ..., Pn must satisfy the restriction Farg >> form[Pj].

The calls to the function Fname in all function definitions must satisfy the following condition.

Let a call to the function Fname have the form <Fname Re> . Then there must be satisfied the restriction Farg >> form[Re].

To describe the restrictions imposed on the results returned by paths, we use the following notation.

The fact that the results returned by a path Q satisfy a format F will be written as F |- Q.

Rests, sources, and result expressions may be regarded as particular cases of paths, for which reason the above notation is applicable to them as well as to paths.

Similarly, the fact that the results returned by a pattern alternative Palt satisfy a format F will be written as F |- Palt.

Now we can formulate the restrictions imposed on the function definitions by the output formats of the functions.

Let the definition of a function Fname be Fname Palt, and the output format Fres. Then there must be satisfied Fres |- Palt.

The relations F |- Q and F |- Palt are defined by the following inference rules.

```
form[] |- S                 form[He] |- S
F |- R                      F |- R
------------                ---------------
F |- S R                    F |- S :: He R


form[He] |- S"
form[He] |- S'
F |- R
-------------------------
F |- S" $iter S' :: He R


                                      form[] |- S
F |- R               F |- Q           F |- R
-------------        ---------        ------------
F |- S : P R         F |- , Q         F |- # S R


F |- Q               F |- Q
----------           ----------       F |- $fail
F |- \? Q            F |- \! Q


F |- Q
---------            F |- $error Q
F |- = Q


F |- Q
F |- Palt
------------------------
F |- $trap Q $with Palt


F |- Qj  for all j=1,...,n
---------------------------
F |- \{Q1; ... Qn;}


F |- Palt            F >> form[Re]
--------------       --------------
F |- S : Palt        F |- Re
```

```
F |- Rj  for all j=1,...,n
--------------------------
F |- \{P1 R1; ... Pn Rn;}
```

13.3.RESTRICTIONS ON THE USE OF REFERENCES TO FUNCTIONS

If a construct &Fname, which is a reference to the function Fname, appears in a pattern expression or in a result expression, the function Fname must be declared as either $func Fname e = e or $func? Fname e = e.

13.4.RESTRICTIONS ON THE USE OF VARIABLES

The variables appearing in a function definition must satisfy certain restrictions.

Namely, a variable appearing in a result expression must have been already defined. A variable gets defined, when it appears in a pattern or in a hard expression.

If several different variables have been defined at the same place, their indices must be different.

Now, to give these restrictions a more exact formulation, we introduce the following notation.

vars[X] denotes the set of variables appearing in the construct X.

{} denotes the empty set.

v1+v2 denotes the union of the sets v1 and v2.

v1++v2 denotes the variable set v1 extended with the variable set v2. To put it more exactly, v1++v2 contains all the variables from v2, as well as all variables from v1 whose indices are different from the indices of the variables contained in v2. For example, {sX, sY} ++ {eY, eZ} = {sX, eY, eZ}.

A judgment v |- Q means that all variables in v have different indices, and all variables whose values are needed for the evaluation of the path Q belong to v.

Rests, sources, and result expressions may be regarded as particular cases of paths, for which reason the above notation is applicable to them as well as to paths.

Similarly, a judgment v |- Palt means that all variables in v have different indices, and all variables whose values are needed for the evaluation of the pattern alternative Palt belong to v.

Now we can formulate the restrictions imposed on the use of variables in the function definitions.

Let the definition of a function Fname be Fname Palt. Then there must be satisfied {} |- Palt.

The relations v |- Q and v |- Palt are defined by the following inference rules.

```
v |- S                v |- S
v |- R                v++vars[He] |- R
----------            -----------------
v |- S R              v |- S :: He R


v |- S"
v++vars[He] |- S'               v |- S
v++vars[He] |- R                v+vars[P] |- R
-------------------------       ---------------
v |- S" $iter S' :: He R        v |- S : P R


                      v |- S
v |- Q                v |- R                v |- Q
---------             -----------           ----------
v |- , Q              v |- # S R            v |- \? Q


v |- Q                                      v |- Q
----------            v |- $fail            ---------
v |- \! Q                                   v |- = Q


                      v |- Q
v |- Q                v |- Palt
--------------        ------------------------
v |- $error Q         v |- $trap Q $with Palt


v |- Qj  for all j=1,...,n
--------------------------
v |- \{Q1; ... Qn;}


v |- S
v |- Palt
--------------
v |- S : Palt


all variables in v have different indices
all variables in  vars[Re]  belong to v
-----------------------------------------
v |- Re


v+vars[Pj] |- Rj  for all j=1,...,n
-----------------------------------
v |- \{P1 R1; ... Pn Rn;}
```

13.5. RESTRICTIONS ON THE USE OF CUTS

Each path appearing in a function definition can be assigned a non-negative integer k, the level of the path. If we move forward along a path, the level increases by 1 each time we pass over "\?", and decreases by 1 each time we pass over "\!". Thus, each cut "\!" unambiguously corresponds to its "pair" fence "\?".

Now, to give this requirement a more exact formulation, we introduce the following notation.

Let k be a non-negative integer, and Q a path. The judgment k |- Q means that the path Q can be assigned the level k.

Rests, sources, and result expressions may be regarded as particular cases of paths, for which reason the above notation is applicable to them as well as to paths.

Similarly, let Palt be a pattern alternative. Then the judgment k |- Palt means that the pattern alternative Palt can be assigned the level k.

Now we can formulate the restrictions imposed on the use of cuts in the function definitions.

Let the definition of a function Fname be Fname Palt. Then there must be satisfied 0 |- Palt.

The relations k |- Q and k |- Palt are defined by the following inference rules.

```
0 |- S                 0 |- S
k |- R                 k |- R
---------              ---------------
k |- S R               k |- S :: He R


0 |- S"
0 |- S'                          0 |- S
k |- R                           k |- R
-------------------------        -------------
k |- S" $iter S' :: He R         k |- S : P R


                       0 |- S
k |- Q                 k |- R                  k+1 |- Q
---------              -----------             ----------
k |- , Q               k |- # S R              k |- \? Q


k |- Q                                         0 |- Q
------------           k |- $fail              ---------
k+1 |- \! Q                                    k |- = Q
```

```
                          0 |- Q
    0 |- Q                k |- Palt
    --------------        ------------------------
    k |- $error Q         k |- $trap Q $with Palt


    k |- Qj  for all j=1,...,n
    --------------------------
    k |- \{Q1; ... Qn;}


    0 |- S
    k |- Palt
    --------------        k |- Re
    k |- S : Palt


    k |- Rj  for all j=1,...,n
    --------------------------
    k |- \{P1 R1; ... Pn Rn;}
```

14.MODULES

A program written in Refal Plus consists of one or more modules. Each module comprises two components: the interface of the module and the implementation of the module.

The interface of a module contains the parts of the module that may be visible in other modules, whereas the implementation of the module contains the parts of the module that are invisible in other modules.

In the operating system MSDOS each module MMMM occupies two files. Namely, the interface of the module is kept in the file MMMM.RFI, and the implementation in the file MMMM.RF.

```
$    ModuleInterface =
$         { Declaration }.
$    Declaration =
$         ConstantDeclaration | BoxDeclaration |
$         VectorDeclaration | StringDeclaration |
$         TableDeclaration | ChannelDeclaration |
$         FunctionDeclaration.

$    ModuleImplementation =
$         { Import } { ImplementationDirective }.
$    ImplementationDirective =
$         Declaration |
$         TraceDirective |
$         FunctionDefinition.
```

```
$    Import = "$use" { ModuleName } ";".
$    ModuleName = Word.
```

The names declared in the interface of a module YYYY can be made visible in the implementation of a module XXXX by putting the directive $use YYYY into the implementation of the module XXXX in the following way:

```
------------------------------------------------------------
/* File XXXX.RFI */
/* The interface of the module XXXX. */
......
------------------------------------------------------------
/* File XXXX.RF */
$use ... YYYY ... ;
/* Henceforth, the names declared in YYYY.RFI */
/* will be visible. */
......
------------------------------------------------------------
/* File YYYY.RFI */
/* The interface of the module YYYY. */
......
------------------------------------------------------------
/* File YYYY.RF */
/* The implementation of the module YYYY. */
......
------------------------------------------------------------
```

15.EXECUTION OF PROGRAM

A program in Refal Plus may consist of several modules, one of which must export the function Main. This function is said to be the main function of the program, and must have the following declaration:

```
$func Main = e;
```

If a function with the name Main is declared in some other way, but, nevertheless, is exported by a module, this situation is considered to be an error.

The execution of the Refal program amounts to evaluating the call to the function Main, the argument of the call being empty.

```
        <Main >
```

The module that contains the definition of the main function is permitted to have no interface part, in which case the Refal Plus compiler assumes the module's interface to consist of the single function declaration:

```
$func Main = e;
```

Chapter III. LIBRARY OF FUNCTIONS

1.HOW TO USE LIBRARY FUNCTIONS

An essential part of the Refal Plus system is the library of functions, consisting of several modules.

If a user-written module contains references to library functions defined in a library module MMMM, then, at the beginning of the user-written module, there must appear the directive

```
$use MMMM;
```

which imports into the user-written module the declarations of all functions defined in the library module MMMM.

At present, the library of functions comprises the following modules:

```
ACCESS    - direct access to ground expressions.
APPLY     - application of functions passed as arguments.
ARITHM    - arithmetic operations on integers.
BOX       - box operations.
CLASS     - predicates for determining classes of symbols.
COMPARE   - comparison operations.
CONVERT   - data conversions.
DOS       - calls to the operating system.
STDIO     - standard input/output.
STRING    - string operations.
TABLE     - table operations.
VECTOR    - vector operations.
```

In future, the library of function may be extended with other modules.

2.ACCESS: DIRECT ACCESS TO GROUND EXPRESSIONS

```
$func   LENGTH  e.Exp = s.ExpLen;
$func?  LEFT    s.Left  s.Len e.Exp = e.SubExp;
$func?  RIGHT   s.Right s.Len e.Exp = e.SubExp;
$func?  MIDDLE  s.Left  s.Right e.Exp = e.SubExp;
$func?  L       s.Left  e.Exp = t.SubExp;
$func?  R       s.Right e.Exp = t.SubExp;
```

These functions provide direct access to the components of ground expressions. The arguments s.Left, s.Right, and s.Len must be non-negative integers. e.Exp may be any ground expression.

If s.Left, s.Right, and s.Len are not non-negative integers, the functions return the error $error(Fname "Invalid argument"), where Fname is the function's name.

LENGTH returns the length of the expression e.Exp measured in terms. In other words, a ground expression Ge of the form

Gt1 Gt2 ... GtN is assumed to have the length N.

For example:

```
<LENGTH >                  =>   0
<LENGTH A B C>             =>   3
<LENGTH (A B) C (D E)>     =>   3
```

LEFT removes the first s.Left terms from e.Exp, and then returns the first s.Len terms of the remaining expression.

RIGHT removes the last s.Right terms from e.Exp, and then returns the last s.Len terms of the remaining expression.

MIDDLE removes the first s.Left and the last s.Right terms from e.Exp, and returns the remaining expression.

L removes the first s.Left terms from e.Exp, and returns the first term of the remaining expression.

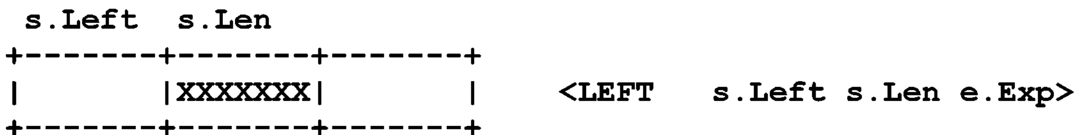
R removes the last s.Right terms from e.Exp, and returns the last term of the remaining expression.

If the length of e.Exp is not sufficient for the operation to be performed, all the above functions return $fail(0).

For example:

```
<MIDDLE 2 3    A B C D E F>    =>   C
<MIDDLE 2 3    A B C D>        =>   $fail(0)
<MIDDLE 0 0    A B C>          =>   A B C
<LEFT   2 3    A B C D E F>    =>   C D E
<LEFT   2 3    A B C D>        =>   $fail(0)
<LEFT   0 0    A B C>          =>
<RIGHT  2 3    A B C D E F>    =>   B C D
<RIGHT  2 3    A B C D>        =>   $fail(0)
<RIGHT  0 0    A B C>          =>
<L      2      A B C D E F>    =>   C
<L      2      A B>            =>   $fail(0)
<R      2      A B C D E F>    =>   D
<R      2      A B>            =>   $fail(0)
```

The operations MIDDLE, LEFT, and RIGHT may be depicted in the following way:



3.APPLY: APPLICATION OF FUNCTIONS PASSED AS ARGUMENTS

```
$func? APPLY s.Name e.Exp = e.Exp;
```

APPLY returns the result of applying the function referred to by the reference s.Name to the expression e.Exp.

4.ARITHM: ARITHMETIC OPERATIONS ON INTEGERS

```
$func "+"     s.Int1 s.Int2 = s.Int;
$func "-"     s.Int1 s.Int2 = s.Int;
$func "*"     s.Int1 s.Int2 = s.Int;
$func DIV-REM s.Int1 s.Int2 = s.Quo s.Rem;
$func DIV     s.Int1 s.Int2 = s.Quo;
$func REM     s.Int1 s.Int2 = s.Rem;
$func GCD     s.Int1 s.Int2 = s.Gcd;
```

These functions provide operations on signed integers of arbitrary size. Each of the arguments of the arithmetic functions must be a single numeric symbol.

If one of the arguments of an arithmetic function is not a numeric symbols, the function returns the error $error(Fname "Invalid argument"), where Fname is the function's name.

If both arguments of an arithmetic function are numeric symbols, the function produces the result, depending on the function.

"+" returns the sum of its arguments, "-" the difference of its arguments, "*" the product of its arguments, DIV and REM respectively the quotient AND the remainder of its arguments, DIV-REM both the quotient and the remainder of its arguments, GCD the greatest common divisor of its arguments.

If the result produced by one of the operations "+", "-", or "*" exceeds the size limit imposed by the Refal Plus implementation, the value returned is the error $error(Fname "Size limit exceeded"), where Fname is the function's name.

For example:

```
<"+" 3 5>        =>   8
<"+" 3 -5>       =>   -2
<"-" 3 -5>       =>   8
<"*" -2 3>       =>   -6
<DIV 5 2>        =>   2
<REM 5 2>        =>   1
<DIV-REM 5 2>    =>   2 1
<DIV 6 2>        =>   3
<REM 6 2>        =>   0
<DIV-REM 6 2>    =>   3 0
```

The signs of the quotient and the remainder are determined according to the following rule. If the sign of the dividend is the same as that of the divisor, the quotient must be positive,

otherwise the quotient must be negative. The sign of the remainder must be the same as that of the dividend. Thus, there must always hold the equation

```
dividend = (quotient * divisor) + remainder
```

For example:

```
<DIV 5 3>       =>   1
<REM 5 3>       =>   2
<DIV 5 -3>      =>   -1
<REM 5 -3>      =>   2
<DIV -5 3>      =>   -1
<REM -5 3>      =>   -2
<DIV -5 -3>     =>   1
<REM -5 -3>     =>   -2
```

An attempt at dividing a number by zero results in returning the error $error(Fname "Divide by zero"), where Fname is the function's name. For example:

```
<DIV 5 0>       =>   $error(DIV "Divide by zero")
<REM 5 0>       =>   $error(REM "Divide by zero")
<DIV-REM 5 0>   =>   $error(DIV-REM "Divide by zero")
```

The function GCD, unless both its arguments are equal to zero, returns a positive integer, the greatest common divisor of its arguments. Otherwise, if both arguments are equal to zero, the result is the error $error(GCD "Zero arguments"). For example:

```
<GCD 6 15>      =>   3
<GCD -6 15>     =>   3
<GCD 15 1>      =>   1
<GCD 15 0>      =>   15
<GCD 0 0>       =>   $error(GCD "Zero arguments")
```

5.BOX: BOX OPERATIONS

```
$func BOX     e.Exp = s.Box;
$func ?       s.Box = e.Exp;
$func STORE   s.Box e.Exp = ;
```

BOX creates a new box, puts the expression e.Exp into the box, and returns a reference to the box.

"?" returns the contents of the box referred to by s.Box.

STORE puts the expression e.Box into the box referred to by s.Box.

6.CLASS: PREDICATES FOR DETERMINING CLASSES OF SYMBOLS

```
$func? BOX?     e.Exp = ;
$func? CHANNEL? e.Exp = ;
$func? CHAR?    e.Exp = ;
$func? DIGIT?   e.Exp = ;
$func? FUNC?    e.Exp = ;
$func? INT?     e.Exp = ;
$func? LETTER?  e.Exp = ;
$func? STRING?  e.Exp = ;
$func? TABLE?   e.Exp = ;
$func? VECTOR?  e.Exp = ;
$func? WORD?    e.Exp = ;
```

These functions provides a way to determine whether e.Exp is a symbol belonging to a certain class of symbol.

If e.Exp is not a single symbol, the functions return $fail(0).

If e.Exp is a symbol, the test is performed whether the symbol belongs to the corresponding class of symbols. If so, the value returned is an empty ground expression. Otherwise, the value returned is $fail(0).

The correspondence between the predicate functions and the sets of symbols is as follows.

- BOX? - references to boxes.
- CHANNEL? - references to channels.
- CHAR? - character symbols.
- DIGIT? - character symbols corresponding to decimal digits.
- FUNC? - references to functions.
- INT? - references to integers.
- LETTER? - character symbols corresponding to small and capital letters.
- STRING? - references to strings.
- TABLE? - references to tables.
- VECTOR? - references to vectors.
- WORD? - word symbols.

7.COMPARE: COMPARISON OPERATIONS

```
$func? "="  (e.Exp1)(e.Exp2) = ;
$func? "/=" (e.Exp1)(e.Exp2) = ;
$func? ">=" (e.Exp1)(e.Exp2) = ;
$func? ">"  (e.Exp1)(e.Exp2) = ;
$func? "<=" (e.Exp1)(e.Exp2) = ;
$func? "<"  (e.Exp1)(e.Exp2) = ;
```

These functions compare two expressions e.Exp1 and e.Exp2 to determine whether the corresponding relation between the arguments holds. The correspondence between the functions and the relations is as follows. "=" corresponds to "equal to", "/=" to "not equal to", ">=" to "greater than or equal to", ">" to "greater than", "<=" to "less than or equal to", "<" to "less

than".

If the condition is satisfied, the value returned by the functions is an empty ground expression, otherwise $fail(0).

```
$func COMPARE (e.Exp1)(e.Exp2)  = s.Res;    /* '<', '>', '=' */
```

COMPARE compares two expressions e.Exp1 and e.Exp2, and returns either '<', if e.Exp1 is less than e.Exp2, '>', if e.Exp1 is greater than e.Exp2, or '=', if e.Exp1 is equal to e.Exp2.

Ground expressions are compared according to the following linear ordering relation <.

For all two ground expressions Ge' and Ge", there holds either Ge'<Ge", Ge'=Ge", or Ge"<Ge'.

Two expressions Ge' = Gt1' ... Gtm' and Ge" = Gt1" ... Gtn" are compared lexicographically, which means that their top level terms are compared pairwise from left to right, until a pair is found of two unequal terms Gtk' and Gtk". Then, if Gtk' < Gtk", it is assumed that Ge' < Ge".

If Ge' turns out to be shorter than Ge", and all top level terms in Ge' are equal to the corresponding terms in Ge", it is assumed that Ge' < Ge".

Formally speaking, for all ground expressions Ge, Ge', Ge" and for all ground terms Gt, Gt', Gt" the following holds:

```
If Ge' < Ge",  then  Gt Ge' < Gt Ge".
If Gt' < Gt",  then  Gt' Ge' < Gt" Ge".
[] < Gt Ge.
```

where [] denotes an empty ground expression.

The ordering of the ground terms is defined as follows.

Symbols are assumed to be less than the terms of the form (Ge). In other words, for all symbols Gs and ground expressions Ge,

```
Gs < (Ge)
```

Comparing the terms of the form (Ge) is reduced to comparing their contents according to the rule:

```
If Ge' < Ge",  then  (Ge') < (Ge").
```

Each symbol belongs to one and only one of the following sets of symbols:

```
character symbols
word symbols
numeric symbols
reference symbols
```

These sets will be referred to as symbol classes. We consider the set of symbol classes as equipped with a linear ordering, the ordering being given by the above list of symbol classes.

Thus the set of character symbols precedes the set of word symbols, etc.

If two symbols Gs' and Gs" belong to two different classes Class' and Class", and Class' < Class", then it is assumed that Gs' < Gs".

If two symbols belong to the same class, they are compared according the following rules.

Character symbols are ordered according their ASCII codes.

Word symbols are converted to corresponding sequences of character symbols, which are compared as described above.

Numeric symbols are compared as corresponding numbers.

The ordering on the set of reference symbols depends on the Refal Plus implementation.

8.CONVERT: DATA CONVERSIONS

```
$func  TO-LOWER  e.Char = e.Char;
$func  TO-UPPER  e.Char = e.Char;
$func  CHARS-TO-BYTES  e.Char = e.Int;
$func  BYTES-TO-CHARS  e.Int = e.Char;
$func  TO-CHARS  e.Exp  = e.Char;
$func  TO-WORD   e.Exp  = s.Word;
$func? TO-INT    e.Exp  = s.Int;
```

TO-LOWER converts a sequence of character symbols to a character sequence in which all capital letters are replaced with the correspondent small letters.

TO-UPPER converts a sequence of character symbols to a character sequence in which all small letters are replaced with the corresponding capital letters.

CHARS-TO-BYTES converts a sequence of character symbols to a sequence of numbers, each number being the ASCII code of the corresponding character.

If one of the above functions is given an argument that is not a sequence of character symbols, the value returned is $error(Fname "Invalid argument"), where Fname is the function's name.

For example:

```
<TO-LOWER 'AbCd+'>  =>   'abcd+'
<TO-LOWER 25>       =>
                  $error(TO-LOWER "Invalid argument")
<TO-UPPER 'AbCd+'>  =>   'ABCD+'
<TO-UPPER 25>       =>
                  $error(TO-UPPER "Invalid argument")

<CHARS-TO-BYTES 'ABC'>   =>   65 66 67
```

BYTES-TO-CHARS takes as argument a sequence of numbers, each number ranging between 0 and 255, and converts it to a sequence of character symbols, each character having the ASCII code equal to the corresponding number.

For example:

```
<BYTES-TO-CHARS 65 66 67>     =>   'ABC'
```

TO-CHARS, TO-WORD, and TO-INT take an arbitrary ground expression as argument, and, first of all, convert it to a character sequence. The conversion is performed as follows. Character symbols are replaced with the corresponding characters, the parentheses are replaced with the characters '(' and ')', word symbols are replaced with the corresponding character sequences, numeric symbols are replaced with their character representations, references to strings are replaced with the contents of the strings, all other references are replaced with their character representations, which depend on the Refal Plus implementation.

If the character sequence thus obtained exceeds the size limit imposed by the Refal Plus implementation, the value returned by the functions is $error(Fname "Argument too large for conversion"), where Fname is the function's name.

Then the functions TO-CHARS, TO-WORD, and TO-INT proceed in the following way.

TO-CHARS just returns the character sequence thus obtained as its result.

```
<TO-CHARS "John">    =>   'John'
<TO-CHARS 'John'>    =>   'John'
<TO-CHARS 326>       =>   '326'
<TO-CHARS -326>      =>   '-326'
<TO-CHARS (-326) "John">    =>   '(-326)John'
```

TO-WORD converts the character sequence thus obtained to the corresponding word.

```
<TO-WORD "John">     =>   "John"
<TO-WORD 'John'>     =>   "John"
<TO-WORD 326>        =>   "326"
<TO-WORD -326>       =>   "-326"
<TO-WORD (-326) "John">    =>   "(-326)John"
```

TO-INT considers the character sequence thus obtained as the character representation of an integer, and converts it to the corresponding numeric symbol. If the character string is not a correct representation of an integer, the value returned is $fail(0).

For example:

```
<TO-INT '326'>       =>   326
<TO-INT '+326'>      =>   326
<TO-INT "-3" '26'>   =>   -326
<TO-INT -32 006>     =>   -326
<TO-INT 'John'>      =>   $fail(0)
```

9.DOS: CALLS TO THE OPERATING SYSTEM

```
$func ARG    s.Int = e.Arg;
$func GETENV e.VarName = e.Value;
$func TIME   = e.String;
$func EXIT   s.ReturnCode = ;
```

These functions provide some ways of calling the operating system.

The arguments of the functions must satisfy the following restrictions. s.Int must be a non-negative integer, e.VarName a sequence of character and word symbols, s.ReturnCode an integer ranging from 0 to 255. If one or more of the above restrictions are violated, the result returned by the functions is $error(Fname "Invalid argument"), where Fname is the function's name.

ARG returns the command line argument having the number s.Int. If there is no such argument, an empty ground expression is returned.

GETENV returns the value associated in the MSDOS environment with the variable having the name e.VarName.

TIME returns the current date and time represented by a ground expression of the form

```
DD MMM YYYY HH:MM:SS.SS
```

where DD is the month's day, MMM the abbreviated month name ("Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"), YYYY the year number, HH:MM:SS.SS the ours, minutes, seconds, and hundredth of a second. DD, YYYY, HH, MM, SS are represented by integers, MMM by a word. The separators are character symbols ' ', ':' i '.'.

EXIT terminates the execution of the program, with the completion code being equal to s.ReturnCode. If the program terminates in the normal way, i.e. the evaluation of the call to the main function Main terminates, the completion code depends on the result returned by the function Main. If the result is a ground expression, the completion code is equal to 0. Otherwise, if the result has the form $error(Ge), the completion code is equal to 100.

10.STDIO: STANDARD INPUT/OUTPUT

```
$channel STDIN STDOUT STDERR;
```

STDIN, STDOUT, and STDERR are the standard input/output channels, which are automatically opened before the program execution starts, and automatically closed after the program execution has terminated.

```
$func CHANNEL = s.Channel;
```

CHANNEL creates a new channel s.Channel.

```
$func? OPEN-FILE      s.Channel e.FileName s.Mode = ;
$func  CLOSE-CHANNEL  s.Channel = ;
```

OPEN-FILE opens the channel s.Channel and associates it with the file having the name e.FileName. s.Mode is a word symbol specifying the mode in which the file is to be dealt with: "r" or "R" indicates that data are to be read from the file, "w" or "W" that data are to be written to the file, "a" or "A" that data are to be appended to the existing file. If the file cannot be opened, OPEN-FILE returns $fail(0).

CLOSE-CHANNEL closes the channel s.Channel.

```
$func? EOF?         s.Channel = ;
```

EOF? tests whether the current position in the file associated with the channel s.Channel is at the end of the file.

```
$func? READ        = t.Term;
$func? READ-CHAR   = s.Char;
$func? READ-LINE   = e.Char;
$func  WRITE       e.Exp = ;
$func  WRITELN     e.Exp = ;
$func  PRINT       e.Exp = ;
$func  PRINTLN     e.Exp = ;
```

READ reads the current character representation of a ground term from the channel &STDIN. If there is no term to be read, the function returns $fail(0).

READ-CHAR reads the current character from the channel &STDIN. If there is no character to be read, the function returns $fail(0).

READ-LINE reads the characters from the channel &STDIN up to the nearest newline character (inclusive), and returns the characters as the result (not including the newline character). If there is no character to be read, the function returns $fail(0).

WRITE writes the character representation of the expression e.Exp to the channel &STDOUT (if e.Exp does not contain dynamic symbols, the terms comprising the expression can later be read back by the function READ).

WRITELN works in the same way as WRITE does, except that it adds a newline character to the end of the expression's representation.

PRINT converts the expression e.Exp to a character sequence in the way the function TO-CHARS does, and writes this sequence to the channel &STDOUT.

PRINTLN works in the same way as PRINT does, except that it adds a newline character to the end of the character sequence.

```
$func? READ!        s.Channel = t.Term;
$func? READ-CHAR!   s.Channel = s.Char;
```

```
$func? READ-LINE!  s.Channel = e.Char;
$func  WRITE!      s.Channel e.Exp = ;
$func  WRITELN!    s.Channel e.Exp = ;
$func  PRINT!      s.Channel e.Exp = ;
$func  PRINTLN!    s.Channel e.Exp = ;
```

These functions work in the same way as the corresponding functions without the exclamation marks do, except that the operations are performed on the channel s.Channel.

11.STRING: STRING OPERATIONS

```
$func  STRING         e.Source = s.String;
$func  STRING-INIT    s.String s.Len s.Fill = ;
$func  STRING-FILL    s.String s.Fill = ;
$func  STRING-LENGTH  s.String = s.Len;
$func  STRING-REF     s.String s.Index = s.Char;
$func  STRING-SET     s.String s.Index s.Char = ;
$func  STRING-REPLACE s.String e.Source = ;
$func  SUBSTRING      s.String s.Index s.Len = s.NewString;
$func  SUBSTRING-FILL s.String s.Index s.Len s.Fill = ;
```

These functions provide a way to create, modify, and access strings. The arguments of the functions must satisfy the following restrictions. s.String must be a reference to a string, s.Index and s.Len non-negative integers, s.Fill a character symbol, e.Source a sequence of references to strings, word symbols, and character symbols.

If one or more of the above restrictions are violated, the result returned by the functions is $error(Fname "Invalid argument"), where Fname is the function's name.

At any moment, a string contains a finite sequence (which may be empty) of character symbols, which is said to be the contents of the string. A string containing a sequence of N+1 character symbols Gc0, Gc1, ..., GcN is said to have the length N+1. The contents of the string will be written as

    Gc0 Gc1 ... GcN

Thus the string components Gc0, Gc2, ..., GcN are numbered starting from zero.

STRING creates a new string and returns a reference to the new string. The contents of the new string is formed from e.Source in the following way.

Suppose the parameter e.Source has the form Gs1 Gs2 ... GsM, where each symbol Gsj is either a character symbol, a word symbol, or a reference to a string. Then each symbol Gsj is transformed as follows.

If Gsj is a character symbol Gc, Gsj is left unchanged.

If Gsj is a word symbol, Gsj is replaced with the character sequence that is the contents of the word.

If Gsj is a reference to a string, Gsj is replaced with the

contents of the string (without changing the state of the string).

The value of the parameter e.Source thus transformed becomes the contents of the new string.

STRING-INIT replaces the contents of the string referred to by s.String with a new contents of length s.Len where all the characters are s.Fill.

STRING-FILL replaces each character in the string referred to by s.String with s.Fill. The length of the string remains unchanged.

STRING-LENGTH returns the length of the string referred to by s.String.

STRING-REF returns the character contained in the position s.Index in the string referred to by s.String.

STRING-SET replaces the character contained in the position s.Index in the string referred to by s.String with s.Char. The length of the string remains unchanged.

STRING-REPLACE replaces the contents of the string referred to by s.String with the new contents formed from s.Source in the same way as it is done by the function STRING.

SUBSTRING creates a new string, and returns a reference to the new string, the contents of which is formed in the following way. Let the contents of the string referred to by s.String be Gc0 Gc1 ... GcN. Then the contents of the new string is obtained by removing the first s.Index characters from this sequence, and selecting the first s.Len characters of the remaining sequence.

The contents of the source string remains unchanged.

SUBSTRING-FILL replaces s.Len consecutive characters in the string referred to by s.String with s.Char, starting from the character in the position s.Index. The length of the string remains unchanged.

If the length of the string is not sufficient for one of the above operations to be performed, the string remains unchanged, and the value returned by the functions is $error(Fname "Index out of range"), where Fname is the function's name.

If one of the above operations has to create a string contents whose length exceeds the size limit imposed by the Refal Plus implementation, the string remains unchanged, and the value returned by the functions is $error(Fname "Size limit exceeded"), where Fname is the function's name.

12.TABLE: TABLE OPERATIONS

```
$func   TABLE         = s.Tab;
$func   BIND          s.Tab (e.Key) (e.Val) = ;
$func   UNBIND        s.Tab e.Key = ;
$func?  LOOKUP        s.Tab e.Key = e.Val;
$func?  IN-TABLE?     s.Tab e.Key = ;
$func   DOMAIN        s.Tab = e.Domain ;
$func   TABLE-COPY    s.Tab = s.TabCopy;
$func   REPLACE-TABLE s.TargetTable s.SourceTable = ;
```

TABLE creates a new empty table, and returns a reference to this table.

BIND binds the key e.Key to the value e.Val in the table referred to by s.Tab.

UNBIND removes the key e.Key as well as the value associated with the key in the table referred to by s.Tab. If the table does not contain the key e.Key, the state of the table remains unchanged.

LOOKUP returns the value associated with the key e.Key in the table referred s.Tab. If the table does not contain the key e.Key, the function returns $fail(0).

IN-TABLE? tests whether the table referred to by s.Tab contains the key e.Key.

DOMAIN returns the list of the keys registered in the table referred to by s.Tab. Let the set of the keys registered in the table be {Ge1, Ge2, ..., Gen}, then e.Domain has the form

```
(Ge1)(Ge2) ... (Gen)
```

where the order of the keys depends on the Refal Plus implementation.

TABLE-COPY creates a new table, copies into the new table the contents of the table referred to by s.Tab, and returns a reference to the new table.

REPLACE-TABLE replaces the contents of the table referred to by s.TargetTable with a copy of the contents of the table referred to by s.SourceTable.

13.VECTOR: VECTOR OPERATIONS

```
$func  VECTOR          e.Source = s.Vector;
$func  VECTOR-TO-EXP   s.Vector = e.Exp;
$func  VECTOR-INIT     s.Vector s.Len e.Fill = ;
$func  VECTOR-FILL     s.Vector e.Fill = ;
$func  VECTOR-LENGTH   s.Vector = s.Len;
$func  VECTOR-REF      s.Vector s.Index = e.Exp;
$func  VECTOR-SET      s.Vector s.Index e.Exp = ;
$func  VECTOR-REPLACE  s.Vector e.Source = ;
$func  SUBVECTOR       s.Vector s.Index s.Len = s.NewVector;
$func  SUBVECTOR-FILL  s.Vector s.Index s.Len e.Fill = ;
```

These functions provide a way to create, modify, and access vectors. The arguments of the functions must satisfy the following restrictions. s.Vector must be a reference to a vector, s.Index and s.Len non-negative integers, e.Fill an arbitrary ground expression, e.Source a sequence of references to vectors and terms of the form (Ge).

If one or more of the above restrictions are violated, the result returned by the functions is $error(Fname "Invalid argument"), where Fname is the function's name.

At any moment, a vector contains a finite sequence (which may be empty) of ground expressions, which is said to be the

contents of the vector. A vector containing a sequence of N+1 ground expressions Ge0, Ge1, ..., GeN is said to have the length N+1. The contents of the vector will be written as

(Ge0)(Ge1) ... (GeN)

Thus the vector components Ge0, Ge2, ..., GeN are numbered starting from zero.

VECTOR creates a new vector and returns a reference to the new vector. The contents of the new vector is formed from e.Source in the following way.

Suppose the parameter e.Source has the form Gt1 Gt2 ... GtM, where each ground term Gtj either is a reference to a vector, or has the form (Ge). Then each term Gtj is transformed as follows.

If Gtj has the form (Ge), Gtj is left unchanged.

If Gtj is a reference to a vector, Gtj is replaced with the contents of the vector (without changing the state of the vector).

The value of the parameter e.Source thus transformed becomes the contents of the new vector.

VECTOR-TO-EXP returns the ground expression representing the contents of the vector referred to by s.Vector.

VECTOR-INIT replaces the contents of the vector referred to by s.Vector with a new contents of length s.Len where all the components are e.Fill.

VECTOR-FILL replaces each component in the vector referred to by s.Vector with e.Fill. The length of the vector remains unchanged.

VECTOR-LENGTH returns the length of the vector referred to by s.Vector.

VECTOR-REF returns the ground expression contained in the position s.Index in the vector referred to by s.Vector.

VECTOR-SET replaces the ground expression contained in the position s.Index in the vector referred to by s.Vector with e.Exp. The length of the vector remains unchanged.

VECTOR-REPLACE replaces the contents of the vector referred to by s.Vector with the new contents formed from e.Source in the same way as it is done by the function VECTOR.

SUBVECTOR creates a new vector, and returns a reference to the new vector, the contents of which is formed in the following way. Let the contents of the vector referred to by s.Vector be (Ge0)(Ge1) ... (GeN). Then the contents of the new vector is obtained by removing the first s.Index terms from this sequence, and selecting the first s.Len terms of the remaining sequence.

The contents of the source vector remains unchanged.

SUBVECTOR-FILL replaces s.Len consecutive components in the vector referred to by s.Vector with e.Exp, starting from the component in the position s.Index. The length of the vector remains unchanged.

If the length of the vector is not sufficient for one of the above operations to be performed, the vector remains unchanged, and the value returned by the functions is $error(Fname

"Index out of range"), where Fname is the function's name.

If one of the above operations has to create a vector contents whose length exceeds the size limit imposed by the Refal Plus implementation, the vector remains unchanged, and the value returned by the functions is $error(Fname "Size limit exceeded"), where Fname is the function's name.

REFERENCES

[AbR 88]
S.M.Abramov, S.A.Romanenko. How to Represent Ground Expressions by Vectors in Implementations of the Language Refal. Preprint, Inst. Appl. Mathem., the USSR Academy of Sciences, 1988, N 186. (In Russian)

[AHU 74]
A.V.Aho, J.E.Hopcroft, J.D.Ullman. The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading, Mass., 1974.

[Apt 1983]
K.R.Apt. Formal Justification of a Proof System for Communicating Sequential Processes, Journal Assoc. Comput. Machin., 30(1), pp.197-216, 1983.

[BjJ 82]
D.Bjorner, C.B.Jones. Formal Specification and Software Development. Prentice-Hall International, London, 1982.

[BsR 77]
Basic Refal and its Implementation on Computers. GOSSTROJ SSSR, TsNIPIASS, Moscow, 1977. The authors are not indicated in the book. They are: V.F.Khoroshevski, And.V.Klimov, Ark.V.Klimov, A.G.Krasovski, S.A.Romanenko, I.B.Shchenkov, and V.F.Turchin. (In Russian)

[Hen 80]
P.Henderson. Functional Programming: Application and Implementation. Prentice-Hall, 1980.

[Plotkin 1983]
G.D.Plotkin. An Operational Semantics for CSP, in: D.Bjorner (ed.), Formal Description of Programming Concepts II, North-Holland, Amsterdam, pp.199-223.

[Rom 87a]
S.A.Romanenko. Refal-2 Implementation. Inst. Appl. Mathem., the USSR Academy of Sciences, 1987. (In Russian)

[Rom 87b]
S.A.Romanenko. Refal-4, an Extension of Refal-2 enabling the results of Driving to be represented. Preprint, Inst. Appl. Mathem., the USSR Academy of Sciences, 1987, N 147. (In Russian)

[Rom 88]
S.A.Romanenko. A Compiler Generator Produced by a Self-Applicable Specializer Can Have a Surprisingly Natural and Understandable Structure. In D.Bjorner, A.P.Ershov and N.D.Jones, editors, Partial Evaluation and Mixed Computation, pages 445-463, North-Holland, 1988.

[Sch 86]
D.A.Schmidt. Denotational Semantics. Allyn and Bacon, Boston, 1986.

[Tur 86]
V.F.Turchin. The concept of a supercompiler. ACM Transactions on Programming Languages and Systems, Vol.8, No.3, July 1986, pp.292-325.

[Tur 89]
V.F.Turchin. Refal-5, Programming Guide and Reference Manual. New England Publishing Co., Holyoke, 1989.

[War 80]
D.H.D.Warren. Logic Programming and Compiler Writing. Software - Practice and Experience, Vol.10, 97-125 (1980).

[Wir 73]
N.Wirth. Systematic Programming. An Introduction. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1973.

[Wir 76]
N.Wirth. Algorithms + Data Structures = Programs. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1976.

INDEX OF LIBRARY FUNCTIONS

```
$func  "*"          s.Int1 s.Int2 = s.Int;                      ARITHM
$func  "+"          s.Int1 s.Int2 = s.Int;                      ARITHM
$func  "-"          s.Int1 s.Int2 = s.Int;                      ARITHM
$func? "/="         (e.Exp1)(e.Exp2)  = ;                       COMPARE
$func? "<"          (e.Exp1)(e.Exp2)  = ;                       COMPARE
$func? "<="         (e.Exp1)(e.Exp2)  = ;                       COMPARE
$func? "="          (e.Exp1)(e.Exp2)  = ;                       COMPARE
$func? ">"          (e.Exp1)(e.Exp2)  = ;                       COMPARE
$func? ">="         (e.Exp1)(e.Exp2)  = ;                       COMPARE
$func  ?            s.Box        = e.Exp;                       BOX
$func? APPLY        s.Name e.Exp = e.Exp;                       APPLY
$func  ARG          s.Int = e.Arg;                              DOS
$func  BIND         s.Tab (e.Key) (e.Val) = ;                   TABLE
$func  BOX          e.Exp        = s.Box;                       BOX
$func? BOX?         e.Exp = ;                                   CLASS
$func  BYTES-TO-CHARS e.Char = e.Int;                           CONVERT
$func  CHANNEL      = s.Channel;                                STDIO
$func? CHANNEL?     e.Exp = ;                                   CLASS
$func? CHAR?        e.Exp = ;                                   CLASS
$func  CHARS-TO-BYTES e.Int  = e.Char;                          CONVERT
$func  CLOSE-CHANNEL  s.Channel = ;                             STDIO
$func  COMPARE      (e.Exp1)(e.Exp2)  = s.Res;                  COMPARE
$func? DIGIT?       e.Exp = ;                                   CLASS
$func  DIV          s.Int1 s.Int2 = s.Quo;                      ARITHM
$func  DIV-REM      s.Int1 s.Int2 = s.Quo s.Rem;                ARITHM
$func? DOMAIN       s.Tab = e.KeyList ;                         TABLE
$func? EOF?         s.Channel = ;                               STDIO
$func  EXIT         s.ReturnCode = ;                            DOS
$func? FUNC?        e.Exp = ;                                   CLASS
$func  GCD          s.Int1 s.Int2 = s.Gcd;                      ARITHM
$func  GETENV       e.VarName = e.Value;                        DOS
$func? IN-TABLE?    s.Tab e.Key = ;                             TABLE
$func? INT?         e.Exp = ;                                   CLASS
$func? L            s.Left  e.Exp = t.SubTerm;                  ACCESS
$func? LEFT         s.Left  s.Len e.Exp = e.SubExp;             ACCESS
$func  LENGTH       e.Exp = s.ExpLen;                           ACCESS
$func? LETTER?      e.Exp = ;                                   CLASS
$func? LOOKUP       s.Tab e.Key = e.Val;                        TABLE
$func? MIDDLE       s.Left  s.Right e.Exp = e.SubExp;           ACCESS
$func? OPEN-FILE    s.Channel e.FileName s.Mode = ;             STDIO
$func  PRINT        e.Expr = ;                                  STDIO
$func  PRINT!       s.Channel e.Expr = ;                        STDIO
$func  PRINTLN      e.Expr = ;                                  STDIO
$func  PRINTLN!     s.Channel e.Expr = ;                        STDIO
$func? R            s.Right e.Exp = t.SubTerm;                  ACCESS
$func? READ         = t.Term;                                   STDIO
$func? READ!        s.Channel = t.Term;                         STDIO
$func? READ-CHAR    = s.Char;                                   STDIO
$func? READ-CHAR!   s.Channel = s.Char;                         STDIO
$func? READ-LINE    = e.Char;                                   STDIO
$func? READ-LINE!   s.Channel = e.Char;                         STDIO
```

```
$func   REM            s.Int1 s.Int2 = s.Rem;                     ARITHM
$func   REPLACE-TABLE  s.TargetTab s.SourceTab = ;                TABLE
$func?  RIGHT          s.Right s.Len e.Exp = e.SubExp;            ACCESS
$func   STORE          s.Box e.Exp = ;                            BOX
$func   STRING             e.Source = s.String;                   STRING
$func   STRING-FILL        s.String s.Fill = ;                    STRING
$func   STRING-INIT        s.String s.Len s.Fill = ;              STRING
$func   STRING-LENGTH      s.String = s.Len;                      STRING
$func   STRING-REF         s.String s.Index = s.Char;             STRING
$func   STRING-REPLACE     s.String e.Source = ;                  STRING
$func   STRING-SET         s.String s.Index s.Char = ;            STRING
$func?  STRING?        e.Exp = ;                                  CLASS
$func   SUBSTRING      s.String s.Index s.Len = s.NewString;      STRING
$func   SUBSTRING-FILL s.String s.Index s.Len s.Fill =;           STRING
$func   SUBVECTOR          s.Vector s.Ind s.Len = s.Vector;       VECTOR
$func   SUBVECTOR-FILL s.Vector s.Index s.Len e.Fill =;           VECTOR
$func   TABLE          = s.Tab;                                   TABLE
$func   TABLE-COPY s.Tab = s.TabCopy ;                            TABLE
$func?  TABLE?         e.Exp = ;                                  CLASS
$func   TIME           = e.String;                                DOS
$func   TO-CHARS       e.Exp  = e.Char;                           CONVERT
$func?  TO-INT         e.Char = s.Int;                            CONVERT
$func   TO-LOWER       e.Char = e.Char;                           CONVERT
$func   TO-UPPER       e.Char = e.Char;                           CONVERT
$func   TO-WORD        e.Char = s.Word;                           CONVERT
$func   UNBIND         s.Tab e.Key = ;                            TABLE
$func   VECTOR             e.Source = s.Vector;                   VECTOR
$func   VECTOR-FILL        s.Vector e.Fill = ;                    VECTOR
$func   VECTOR-INIT        s.Vector s.Len e.Fill = ;              VECTOR
$func   VECTOR-LENGTH      s.Vector = s.Len;                      VECTOR
$func   VECTOR-REF         s.Vector s.Index = e.Exp;              VECTOR
$func   VECTOR-REPLACE     s.Vector e.Source = ;                  VECTOR
$func   VECTOR-SET         s.Vector s.Index e.Exp = ;             VECTOR
$func   VECTOR-TO-EXP      s.Vector = e.Exp;                      VECTOR
$func?  VECTOR?        e.Exp = ;                                  CLASS
$func?  WORD?          e.Exp = ;                                  CLASS
$func   WRITE          e.Expr = ;                                 STDIO
$func   WRITE!         s.Channel e.Expr = ;                       STDIO
$func   WRITELN        e.Expr = ;                                 STDIO
$func   WRITELN!       s.Channel e.Expr = ;                       STDIO
```

ADDITIONAL FEATURES OF REFAL PLUS

HEXADECIMAL NUMERIC AND CHARACTER CONSTANTS

Non-negative integers can be written as follows

```
0xZZZ...ZZ
```

where ZZZ...ZZ stands for a non-empty sequence of hexadecimal digits.

For example, 0xFF and 0xff are both equivalent to 255.

The representations of characters appearing in character string literals and word literals may be written as follows

```
\xZZ
```

where ZZ stands for two hexadecimal digits, specifying the ASCII code of the character. A word literal containing such character representations must be enclosed in double quotes.

For example, "\x2A" and "\x2a" are both equivalent to "*".

NEW LIBRARY FUNCTIONS

*** BIT: BITWISE OPERATIONS

The functions providing bitwise operations are defined in the module BIT.

These functions deal with sequences of binary digits represented by signed integers.

Each integer represents a sequence of binary digits, which is infinite to the left, and can be obtained by writing the integer as a two's complement binary number of infinite size. If the integer is non-negative, the sequence thus obtained contains a finite number of ones. Otherwise, if the integer is negative, the sequence contains a finite number of zeros. For example:

```
+3   ...000011
+2   ...000010
+1   ...000001
+0   ...000000
-1   ...111111
-2   ...111110
-3   ...111101
```

The positions in the binary sequence are numbered from right to left, starting from zero.

```
$func  BIT-OR   s.Int1 s.Int2 = s.Int;
$func  BIT-AND  s.Int1 s.Int2 = s.Int;
```

```
$func  BIT-XOR  s.Int1 s.Int2 = s.Int;
```

BIT-OR returns the bitwise logical "or" of the arguments.
BIT-AND returns the bitwise logical "and" of the arguments.
BIT-XOR returns the bitwise logical "exclusive or" of the arguments.

```
$func  BIT-NOT  s.Int = s.Int;
```

BIT-NOT returns the bitwise logical "not" of the argument.

```
$func  BIT-LEFT   s.Int s.Shift = s.Int;
$func  BIT-RIGHT  s.Int s.Shift = s.Int;
```

BIT-LEFT returns the result of logically shifting s.Int by the number of positions specified by s.Shift. If s.Shift is non-negative, s.Int is shifted left, the new bits being zero-filled. Otherwise, if s.Shift is negative, s.Int is shifted right.

BIT-RIGHT returns the result of logically shifting s.Int by the number of positions specified by s.Shift. If s.Shift is non-negative, s.Int is shifted right. Otherwise, if s.Shift is negative, s.Int is shifted left, the new bits being zero-filled.

```
$func? BIT-TEST   s.Int s.Pos = ;
```

BIT-TEST returns a failure, if the position s.Pos in s.Int is equal to zero, otherwise, it returns an empty ground expression.

```
$func  BIT-SET    s.Int s.Pos = s.Int;
$func  BIT-CLEAR  s.Int s.Pos = s.Int;
```

BIT-SET sets the position s.Pos in s.Int to 1, and returns the integer thus obtained.

BIT-CLEAR sets the position s.Pos in s.Int to 0, and returns the integer thus obtained.

```
$func  BIT-LENGTH  s.Int = s.Len;
```

BIT-LENGTH returns the "length" of s.Int. Namely, if s.Int is non-negative, the function returns the position of the rightmost 0 such that there is no 1 to the left of this 0. Otherwise, if s.Int is negative, the function returns the position of the rightmost 1 such that there is no 0 to the left of this 1.

For example:

```
<BIT-LENGTH  3>     ==>  2
<BIT-LENGTH  2>     ==>  2
<BIT-LENGTH  1>     ==>  1
<BIT-LENGTH  0>     ==>  0
<BIT-LENGTH -1>     ==>  0
<BIT-LENGTH -2>     ==>  1
```

```
<BIT-LENGTH -3>     ==>  2
```

*** DOS: CALLS TO THE OPERATING SYSTEM

The module DOS is extended with the following functions.

```
$func  DELAY s.MSeconds = ;
$func  SLEEP s.Seconds = ;
$func  RANDOM s.Max = s.Rand;  /* 0 <= s.Rand < s.Max */
$func  RANDOMIZE = ;
```

DELAY suspends the current program from execution for the number of milliseconds specified by s.MSeconds. The interval is accurate only to the nearest hundredth of a second, or the accuracy or the MSDOS clock, whichever is less accurate.

SLEEP suspends the current program from execution for the number of seconds specified by s.Seconds. The interval is accurate only to the nearest hundredth of a second, or the accuracy or the MSDOS clock, whichever is less accurate.

RANDOM returns a pseudorandom integer in the range 0 to s.Max minus 1.

RANDOMIZE initializes the random number generator with a random value.

SCREEN INPUT/OUTPUT

*** SCREEN POSITIONS

Each screen position is specified by two non-negative integers s.Pos s.Col, where s.Pos is the row, and s.Col the column of the position. The rows and columns are numbered starting from 0, the top left corner of the screen being at row 0, column 0.

*** WINDOWS

A window is an area on the screen, possibly surrounded by a border. Each window has an attached number ranging from 1 to 255. The screen is considered to be a special, fictitious window having the number 0.

When you create the window, you give the coordinates for the upper left corner, and the number of rows and columns the window should occupy. At any moment, one of the windows is considered to be the current one (which may be fictitious window number 0, if there is no "true" window on the screen).

When a window is created, it becomes the current window, and all output will automatically be sent to it. However, you may make any other window the current one, thereby redirecting the input and output.

Unless otherwise stated, all input/output functions described later operate relative to the current window, the screen positions being specified with respect to the upper left corner of the current window. Each window has an attached cursor position, which the program remembers as you shift between windows.

When you remove a window, the contents of the screen behind the window is automatically reestablished.

You can use the same number more than once for creating windows, but only the last window created with a given number can be accessed by the functions dealing with windows.

*** ERRORS

The screen coordinates, as well as the window and field sizes, must be integers. The color attribute values must be integers ranging from 0 to 255. Window numbers must be integers ranging from 0 to 255.

If a function is given arguments violating the above conditions, the function returns $error(Fname "Invalid argument").

If a function is given screen coordinates lying outside the screen, the function return $error(Fname "Invalid cursor values").

If a function is required to perform an operation on a window that does not exist, the function returns $error(Fname "Unknown window")

If a function is required to perform an operation on the current window, and there exists no window (except window number 0), the function returns $error(Fname "No window").

If a function is required to perform an operation on the frame of a window, and the window has no frame, the function returns $error(Fname "No frame").

If a function is required to created a window such that some parts of the window lie outside the screen, the function returns $error(Fname "Invalid argument").

*** CONIO: CONSOLE INPUT/OUTPUT

```
$func? KEY-PRESSED? = ;
$func  READ-KEY = s.Char;
```

KEY-PRESSED? returns an empty ground expression if a key on the keyboard has been pressed, otherwise, it returns a failure.

READ-KEY returns a single character from the keyboard, if a key has been pressed. Otherwise it waits for a key to be pressed. A number of keys, including the function and cursor keys, will return two characters, where the first is ASCII 0.

```
$func  GET-SCR-CHAR-ATTR  s.Row s.Col = s.Ch s.Attr;
$func  PUT-SCR-CHAR  s.Row s.Col s.Ch = ;
$func  PUT-SCR-ATTR  s.Row s.Col s.Attr = ;
```

GET-SCR-CHAR-ATTR returns the character s.Ch along with its attribute s.Attr at position s.Row s.Col.

PUT-SCR-CHAR writes the character s.Ch on the screen at position s.Row s.Col. The attribute at the position remains unchanged.

PUT-SCR-ATTR sets the attribute of the character at position s.Row s.Col to the value s.Attr. The character at the position remains unchanged.

```
$func  GET-FIELD-STR  s.Row s.Col s.Length = s.Chars;
$func  PUT-FIELD-STR  s.Row s.Col s.Length s.Chars = ;
$func  PUT-FIELD-ATTR  s.Row s.Col s.Length s.Attr = ;
```

These functions deal with fields. A field is specified by its starting position s.Pos s.Col, and its length s.Length, and must fit inside the current window.

GET-FIELD-STR returns the text occupying the field represented by a word symbol.

PUT-FIELD-STR writes the text s.Chars represented by a word symbol into the field. If s.Chars contains more characters than s.Length indicates, only the first s.Length characters are written. If s.Chars is shorter than s.Length, the rest of the field will be filled with blank spaces. The attributes of all the positions in the field remain unchanged.

PUT-FIELD-ATTR gives the attribute s.Attr to all the positions in the field.

```
$func  GET-CURSOR  = s.Row s.Col;
$func  SET-CURSOR  s.Row s.Col = ;
```

GET-CURSOR returns the current cursor position in the current window.

SET-CURSOR moves the cursor to the indicated position s.Row s.Col relative to (0,0) in the current window.

```
$func  GET-CURSOR-FORM  = s.StartLine s.EndLine;
$func  SET-CURSOR-FORM  s.StartLine s.EndLine = ;
```

The height and vertical position of the cursor within a single-character display area (cell) is determined by the start scan line number and the end scan line number, which are small non-negative integers s.StartLine and s.Endline.

GET-CURSOR-FORM returns the current cursor form.

SET-CURSOR-FORM sets the current cursor form.

```
$func  GET-ATTRIBUTE  = s.Attr;
$func  SET-ATTRIBUTE  s.Attr = ;
```

Each window has its own write attribute, which is given to the characters written to this window. When you create a window, the write attribute automatically receives the value of the window attribute.

GET-ATTRIBUTE returns the current write attribute of the

current window.

SET-ATTRIBUTE sets the write attribute of the current window to the new value s.Attr.

```
$func  GET-TEXT-MODE  = s.Rows s.Cols;
```

GET-TEXT-MODE returns the current screen size.

```
$func  CLEAR-SCREEN = ;
```

CLEAR-SCREEN clears the screen within the limits of the current window. All the positions in the window are filled with blank spaces with the attributes set to the write attribute of the window.

```
$func  CWRITE    e.Exp = ;
$func  CWRITELN  e.Exp = ;
$func  CPRINT    e.Exp = ;
$func  CPRINTLN  e.Exp = ;
```

CWRITE writes the character representation of the ground expression e.Exp to the current window.

CWRITELN works in the same way as CWRITE does, except that, after e.Exp has been written, it causes a carriage return/linefeed sequence to be sent to the current window.

CPRINT converts the ground expression e.Exp to a character sequence in the same way as the function TO-CHARS does, and writes the sequence to the current window.

CPRINTLN works in the same way as CPRINT does, except that, after e.Exp has been written, it causes a carriage-return/linefeed sequence to be sent to the current window.

Writing a carriage-return character causes the cursor to move to the start of the current line. Writing a linefeed character causes the cursor to move to the next line without changing its horizontal position. Thus, to move the cursor to the start of the next line, we have to write two characters: a carriage-return and a linefeed.

*** WINDOW: WINDOW HANDLING

```
$const  NO-FRAME = -1;

$func  MAKE-WINDOW
         s.WindowNo s.WindowAtt s.FrameAtt s.FrameStr
         s.Row s.Col s.Height s.Width = ;
$func  MAKE-WINDOW!
         s.WindowNo s.WindowAtt s.FrameAtt s.FrameStr
         s.Row s.Col s.Height s.Width
         s.ClearWindow s.FrameStrPos s.FrameTypeStr = ;
```

MAKE-WINDOW and MAKE-WINDOW! create a new window on the screen, which becomes the current one. There must be specified

the following arguments.

s.WindowNo is the number of the window. Each window is identified by a number, which you use when selecting the active window.

s.WindowAtt is the window write attribute.

s.FrameAtt is the attribute of the frame and title of the window. If this argument is equal to -1, the window will have neither a frame nor a title, in which case the arguments s.FrameStr, s.FramePos and s.FrameTypeStr are ignored.

s.FrameStr is the title of the window represented by a word symbol. The title will appear in the top border line. If the title is empty, no text will appear in the top border. If the title is longer than the border, it will be truncated.

s.Row and s.Col are the row and column positions of the top left corner of the window, relative to the whole screen.

s.Height is the height of the window, in terms of rows (including the frame, if any).

s.Width is the width of the window, in terms of columns (including the frame, if any).

s.ClearWindow specifies whether the program will clear the text area of the window after creating it. If the argument is equal to 0, the text area of the newly created window is cleared. If the argument is equal 1, the text area is filled with blank spaces.

s.FrameStrPos specifies where the window title will be located (within the top border of the frame). If the argument is equal to 255, the title will be centered. If the argument is an integer ranging from 0 to 254, the title will be placed at the specified position (column), relative to the left border of the window.

s.FrameTypeStr specifies how to draw the window frame. This argument must be a word symbol containing exactly six characters, which will be used for drawing the following elements of the frame:

1st char	Upper left corner
2nd char	Upper right corner
3rd char	Lower left corner
4th char	Lower right corner
5th char	Horizontal line
6th char	Vertical line

MAKE-WINDOW enables only the first eight of the above arguments to be specified, the remaining arguments being given the following default values:

s.ClearWindow	1	(the window is cleared)
s.FrameStrPos	255	(the title is centered)
s.FrameTypeStr	"\xDA\xBF\xC0\xD9\xC4\xB3"	(a single-line border)

```
$func  CURRENT-WINDOW-NO  = s.WindowNo;
$func  CURRENT-WINDOW =
```

```
          s.WindowNo s.WindowAtt s.FrameAtt s.FrameStr
          s.Row s.Col s.Height s.Width;
$func  CURRENT-WINDOW! =
          s.WindowNo s.WindowAtt s.FrameAtt s.FrameStr
          s.Row s.Col s.Height s.Width
          s.ClearWindow s.FrameStrPos s.BorderChars ;
```

These functions enable the program to get the parameters of the current window. If there is no "true" window on the screen, the parameters of fictitious window number 0 (corresponding to the whole screen) are returned.

```
$func? EXIST-WINDOW?  s.WindowNo = ;
$func  SHIFT-WINDOW  s.WindowNo = ;
$func  REMOVE-WINDOW  = ;
$func  REMOVE-WINDOW!  s.WindowNo s.Refresh = ;
```

EXIST-WINDOW returns a failure if there is no window number s.WindowNo. Otherwise, it returns an empty ground expression.

SHIFT-WINDOW changes the current window to the one referred to by s.WindowNo. (The contents of the previously active window and the cursor position in it are stored.) The new current window is then refreshed, in case it has been overwritten since its last activation. (Fictitious window number 0 can't be shifted.)

REMOVE-WINDOW removes the current window from the screen, and refreshes any windows behind this window. (Fictitious window number 0 can't be removed.)

REMOVE-WINDOW removes the window specified by s.WindowNo, which doesn't have to be the current one. (Fictitious window number 0 can't be removed.) The value of s.Refresh determines whether windows behind the removed one will be refreshed. If s.Refresh is equal to 0, the windows won't be refreshed. If s.Refresh is equal to 1, the windows will be refreshed.

```
$func  RESIZE-WINDOW!  s.Row s.Col s.Height s.Width = ;
```

RESIZE-WINDOW! changes position and size of the current window. Its arguments specify the new position (starting row and column) and dimensions (number of rows and columns) for the window.

```
$func  SET-WINDOW-ATTR  s.Attr = ;
$func  SET-FRAME-ATTR  s.Attr = ;
$func  SET-WINDOW-FRAME  s.FrameAtt s.FrameStr s.FrameStrPos
                        s.BorderChars = ;
```

SET-WINDOW-ATTR sets the write attribute of the current window to s.Attr, and sets all the attribute values in the text area of the window to s.Attr.

SET-FRAME-ATTR changes the attribute for the frame of the current window.

SET-WINDOW-FRAME changes attribute and characters for the frame of the current window. The window must have a frame, i.e.

the previous value of the frame attribute must be different from -1.

```
$func  SCROLL s.NoOfRows  s.NoOfCols = ;
```

SCROLL scrolls the contents of the current window up (or down) and left (or right). s.NoOfRows indicates the number of lines to be scrolled up or down. A positive number scrolls up; a negative number scrolls down. s.NoOfCols indicates the number of columns to be scrolled left or right. A positive number scrolls left; a negative number scrolls right.

```
$func  GET-WINDOW-STR  = s.ScreenString;
$func  PUT-WINDOW-STR  s.ScreenString = ;
```

GET-WINDOW-STR returns the contents of the text area of the current window represented by a word symbol s.ScreenString. The contents of the symbol is formed in the following way. s.ScreenString contains the same number of lines as there are lines in the current window. The length of each line is determined by the last non-blank character in that line. Each line in s.ScreenString is terminated by a newline character.

PUT-WINDOW-STR puts the text contained by the word symbol s.ScreenString to the current window according to the following criteria:

* If there are more lines in the text than lines in the window, PUT-WINDOW-STR writes lines until the window space is exhausted.
* If there are fewer lines in the string than in the window, PUT-WINDOW-STR fills out the remaining lines in the window with blank spaces.
* If there are more characters on a text line than are available on a window line, PUT-WINDOW-STR truncates the text line to fit.
* If there are fewer characters in a line than columns in the window, PUT-WINDOW-STR fills out the line with blank spaces.