Nepal 1.0

Reference Manual

Version 1

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

Table of contents

Version history....3
1 Installation....4
2 Starting the interpreter....4
3 Programs....5
4 Program execution....5
5 Comments....6
5.1 Word comment....6
5.2 Line comment....6
5.3 General comment....6
6 Scopes....7
7 Data model....8
7.1 Life cycle of data objects....8
7.2 State of data objects....8
7.3 Allocation of data objects....8
7.4 Initialisation of data objects....9
7.5 Access of data objects....10
7.6 Data contents....11
7.7 Data assignments....13
7.8 Data comparison....15
7.9 Data Input and Output....16
7.9.1 Data output....16
7.9.2 Data input....18
8 Types....19
8.1 Inheritance....19
8.2 Small and big types....19
8.3 Access of type procedures and functions....19
8.4 Built-in types....20
8.4.1 Logical data....20
8.4.2 Numerical data (integer)....20
8.4.3 Numerical data (fix point)....21
8.4.4 Characters....22
8.4.5 Strings....23
8.4.6 Polymorphic data....28
8.4.7 Files....29
8.4.8 Directories....32
8.4.9 Lists....33
8.4.10 Hash arrays....35
8.4.11 One-dimensional arrays....37
8.4.12 Two-dimensional arrays....39
8.4.13 Sets....40
8.4.14 Errors....42
8.4.15 Operators....42
8.4.16 Programming codes....42
8.4.17 Arguments....43
8.4.18 System data....43
8.5 User-defined types....46
8.5.1 Modular concept for user-defined types....47
8.5.2 Application-specific extension of user-defined types....47
9 Variables....47
9.1 User-defined variables....47
9.2 Built-in variables....48
10 Procedures and functions....48
10.1 User-defined procedures and functions....48
10.2 Built-in procedures and functions....48
11 Statements....49
11.1 Definition....49
11.2 Inclusion of program files....49
11.2.1 Standard inclusion....49

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

11.2.2 Module inclusion....49
11.2.3 Application-specific inclusion....50
11.3 Procedural operators....50
11.3.1 Assignments....51
11.3.2 Mathematical operators (for types int, real, str, set and list)....52
11.3.3 Structural operators....52
11.3.4 Data access operator....53
11.3.5 Type access operator....53
11.3.6 Module access operator....53
11.3.7 Definitions of variables and initialisations....53
11.4 Control structures....53
11.4.1 Branches....53
11.4.2 Loops....54
11.4.3 Termination of procedures or functions....54
11.4.4 Termination of Nepal programs....54
11.4.5 Exceptions....55
11.5 Procedure calls....55
11.5.1 General procedures....55
11.5.2 Object procedures....55
11.5.3 Type procedures....55
12 Blocks....55
12.1 Definition....55
12.2 Syntactical sugar....56
13 Expressions....56
13.1 Definition....56
13.2 Access of variables....56
13.2.1 General variables....56
13.2.2 Object variables....57
13.3 Function calls....57
13.3.1 General functions....57
13.3.2 Object functions....57
13.3.3 Type functions....57
13.4 Conditional expressions....58
13.5 Functional operators....58
13.5.1 Boolean operators....58
13.5.2 Operators for comparison....58
13.5.3 Mathematical operators (for types int, real, str, set, and list)....58
13.5.4 Range operators 58
13.5.5 Data access operator....58
13.5.6 Type access operator....59
13.5.7 Module access operator....59
13.6 Constants....59
14 Glossary....59

Version history

1	Initial version

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

1 Installation

The easiest way to work with the programming language Nepal is to copy the interpreter *nepal.exe* directly into the working directory. Then starting the interpreter from the console will execute the addressed Nepal program residing in the working directory. Section 2 describes how to start the interpreter.

Another possibility is to establish an installation directory, e.g. “C:\nepal”, copy the interpreter *nepal.exe* into a subdirectory “bin”, and extend the environment variable “path“ by the absolute path where the executable resides, e.g. “C:\nepal\bin”. Then the start of the interpreter in any working directory will use the executable from the installation path.

An initialisation file *nepal.sys* can be established to define options for the Nepal programs. The location of this file always corresponds to the directory where the executable *nepal.exe* resides. The file format is line-oriented. Each line belongs to a certain program option. Every option has the format “<key> <value>” and may occur multiple times. Empty lines are ignored and can be used to structure the option list. The following options are available:

`i` Specify an input file for the Nepal program, e.g. “nepal.in”. The functions for reading data from the console are redirected to the given input file. The affected system functions are *in()*, *sin()*, *pin()* and *inl()*. For a definition of these functions, see section 7.9.2.

`ib` Like option “i”, but for accessing binary files.

`o` Specify an output file for the Nepal program, e.g. “nepal.out”. The functions for writing data to the console are redirected to the given output file. The affected system functions are *out()*, *sout()*, *pout(), fout(), fsout(), outl(), soutl(), foutl()* and *fsoutl()*. For a definition of these functions, see section 7.9.1. If the output file already exists, it will be overwritten.

`ob` Like option “o”, but for accessing binary files.

`O` Specify an output file for the Nepal program, e.g. “nepal.out”. In contrast to the option ‘o’, the written data are appended to the file.

`Ob` Like option “O”, but for accessing binary files.

`a` Specify a relevant application for the Nepal program, e.g. “calculator”. The meaning of this option is described in sections 8.5.2 and 11.2.3.

`v` The value of this option is empty. It forces the interpreter to print the current program version onto the console – immediately after starting the interpreter. The options 'o' and 'O' also redirect this information to the specified output file.

`n` Specify an include directory for the Nepal program, e.g. “incl” or “../incl”. A program file to be included will be searched within the given include directories (in the order they occur in file *nepal.sys*). The first search always tries to find the file directly within the current working directory. The second search is done relative to the working directory for all specified include directories. If the file cannot be found in any include directory, a third search is performed relative to the directory where the interpreter nepal.exe resides. Section 11.2 describes how to specify an include file within a Nepal program.

2 Starting the interpreter

The command line

```
nepal [-i <input file>]
      [-ib <binary input file>]
      [-o <output file>]
      [-ob <binary output file>]
      [-O <output file>]
      [-Ob <binary output file>]
      [-a <application>]
      [-v]
      [-n <include directory>]
      <code file>
      [<argument 1> <argument 2> ...]
```

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

starts the interpreter for the programming language Nepal from the console. The program code is contained in file <code file>.

Optionally, an arbitrary number of program arguments can be entered after the code file. The arguments are separated by space or tab characters. The Nepal program can access these arguments by using the Nepal system functions *argc()* and *argv()* (see section 8.4.18).

Moreover the same program options as provided by the initialisation file *nepal.sys* can be entered before the code file. For a description of these options, see section 1. If one of the options "i", "ib", "o", "ob", "O", "Ob", or "a" is already specified in file *nepal.sys*, the definition on the command line will overwrite the previous value. If the option "n" is already specified in file *nepal.sys*, the include directories from the command line will be merged with the previous list of directories.

3 Programs

A Nepal program consists of definitions, statements and comments. It is specified by a list of ASCII characters (i.e. by a string).

Comments are used to document the Nepal programs. They are ignored by the interpreter. The different types of comments are described in section 5. After elimination of all comments, the remaining Nepal code is a list of statements and definitions separated by semicolons.

A definition can introduce either new types, variables, procedures or functions. These different types of definitions are described in sections 8.5, 9.1and 10.1. The definitions are analysed initially by the interpreter. During program execution they are skipped since they cannot be executed like statements. All definitions are organised in a hierarchy of scopes as described in section 6.

The statements are executed sequentially according to their occurrence within the Nepal program. The different types of statements are described in section 11.

4 Program execution

The Nepal interpreter reads the program string from the code file, applies some preprocessing and transforms the resulting string into a syntax tree. The following preprocessing steps are performed:

- Eliminate all comments
- Eliminate white-space characters (blanks, tabs and end-of-lines) except within character or string constants (e.g. ' ' or "how are you") as well as definitions (e.g. "int n" or "func f")
- Introduce initialisation procedures (cf. section 8.5). E.g. "int n(10)" is replaced by "int n.""(10)".
- Introduce initialisation functions (cf. section 8.5). E.g. "list:(1,2,3)" is replaced by "list:""(1,2,3)".
- Replace squared brackets by corresponding procedure/function calls. For example "a[10]" is replaced by "a."[]"(10)".
- Replace sub-strings of the form ")<statement>...;" by "){<statement>...}" and sub-strings of the form "}<statement>...;" by "}{<statement>...}" (cf. section 12.2)
- Append semicolons after curly brackets if necessary (cf. section 12.2)

Especially end-of-line characters are significant for definitions and strings. For example, the following two statements are equivalent (*str* is the built-in type for strings; the sub-string "\n" represents an end-of-line):

```
str s = "how are\nyou"
```

and

```
str
s = "how are
you"
```

If the resulting string after preprocessing cannot be transformed into a feasible syntax tree, a syntax error is raised and the program execution is stopped. After a successful generation of the tree, the Nepal interpreter analyses the tree, e.g. in order to expand included program files (cf. section 11.2) and to generate symbol tables for all scopes (cf. section 6).

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

If the analysis is not successful, an analysis error is raised and the program execution is stopped. After a successful analysis the statements of the program are interpreted and executed sequentially. System errors during this phase always generate exceptions which can be caught by suitable catch-statements (cf. Section 11.4.5). Errors specified by the user are transformed into exceptions by suitable throw-statements. Errors raised during preprocessing, tree generation and analysis cannot be caught because of their severity.

5 Comments

A Nepal comment always begins with the character ‘#’. Depending on the next character, three different types of comments are distinguished.

5.1 Word comment

Syntax: #<text>

Conditions: The string <text> contains an arbitrary number of characters of any type except blank or tab. Moreover the string <text> must not begin with the character ‘(‘. If the character '#' is the very first character of the (main) program file, the string <text> must not begin with the character '!' (due to the shebang mechanism, cf. Section 5.2).

Meaning: The string <text> is a comment ending at the next white-space character (blank, tab or end of line).

5.2 Line comment

Syntax: # <text> or
<text> or
#!<text>

Conditions: The next character after ‘#’ is either a blank or tab. If the character '#' is the very first character of the (main) program file, the next character '!' also introduces a line comment. This supports the shebang mechanism. The string <text> contains an arbitrary number of characters of any type.

Meaning: The string <text> is a comment ending at the end of the line.

5.3 General comment

Syntax: #(<text 1>#(<text 2>...)#<text n-1>)#<text n>)#

Conditions: The next character after ‘#’ is the character ‘(‘. The comment ends at the next sub-string “)#”. The string <text i>, for 1 = i...n, contains an arbitrary number of characters of any type except the sub-strings “#(“ and “)#”. The comments can be nested with unlimited depth.

Meaning: (Nested) comment of arbitrary length.

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

6 Scopes

A scope is a connected part of a Nepal program containing definitions of user-defined types, variables, procedures or functions (= scope elements). Sections 8.5, 9.1, and 10.1 describe how to specify user-defined types, variables, and procedures/functions. Each type and variable has got a name. Each procedure and function has got a signature, built from the name and input arguments. For a certain scope the defined names and signatures have to be unique per type of definitions.

The following types of scopes exist:
- The outermost scope of a Nepal program (= global scope).
- A block of statements embraced with curly brackets.
- The block of a type definition.
- A block of a procedure/function definition including the input and/or output arguments.
- An included program file.

There is an additional global scope surrounding the outermost scope which contains the built-in types, variables and procedures/functions.

If a scope contains an *need*-statement (cf. section 11.2), the definitions of the corresponding scope are added to the definitions of the current scope. This means that these definitions are accessible from both the current scope and the scope of the included program file. The addition of definitions to the current scope is applied recursively if the included files contain further program inclusions.

Scopes can be nested with unlimited depth. There is even no restriction with respect to the type of nested scopes.

The access to types occurs at
- variable definitions, e.g.
  - `<type name> <variable name>`
- input or output arguments of procedures/functions, e.g.
  - `proc <procedure name> (<type name> <argument name>) { ... }`
- inheritance of types, e.g.
  - `type <type name> (<type name 1>, <type name 2>) { ... }`

The access to variables, procedures or functions occurs at
- assignments, e.g.
  - `<variable name> = <expression>`
  - `<variable name 1> = <variable name 2>`
- procedure or function calls, e.g.
  - `<procedure name>(<variable name>)`
  - `<procedure name>(<function name>())`
- expressions, e.g.
  - `10*<variable name>`
  - `<function name 1>() + <function name 2>()`

Before accessing a scope element, a bottom-up search from the current scope through the surrounding scopes is performed. The search stops if an element is found with matching name or signature. If the search is not successful – even after inspecting the global scope, an error is raised. For a certain scope the search is performed always in both directions – towards the beginning and end of the program. For example, the following statements represent a correct Nepal program although the definitions of the variable and procedure lie beyond the accessing statement.

```
n = 10;
p(n);

int n;
proc p (int n) { outl("n=",n) }
```

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

If the bottom-up search reaches the scope of an included program file and the scope element cannot yet be found in this scope, then the search proceeds directly with the global scope. Otherwise the inclusion of program files would violate the principal of encapsulation.

If the bottom-up search reaches the scope of a type definition and the scope element cannot yet be found in this scope, then the scopes of all existing base types are traversed in breath first – depth second order. If the element is not defined in any base type, the search checks a matching with any built-in procedures/functions available for all types (see section 8.4.6). If this search is not successful, a matching with any procedure/function of the built-in type *any* is investigated since this type is the generic base type of all built-in types (cf. section 8.1). If this search is not successful, it proceeds with the surrounding scope of the current type definition.

To restrict the search to the scope of certain types, the object access operator "." or type access operator ":" can be used. A description of these operators is given in sections 7.5 and 8.3. Applying the operator ":" to the built-in type *sys* (cf. section 8.4.18) provides access to the global scope of the Nepal program.

7 Data model

7.1 Life cycle of data objects

When a scope (cf. section 6) is entered during execution of a Nepal program, a (named) data object is created for each variable of the scope. The type of the object equals to the type of the variable. When a scope is exited during program execution, the corresponding data objects are destroyed.

Anonymous data objects are created when expressions are evaluated. These objects are destroyed not later than the time when the statement containing the expression has finished execution.

7.2 State of data objects

After creation of a data object its state is always *empty*. The built-in object function *empty()* is used to inquire this state. After modification of the data object, the initial state can be reached again by using the object procedure *clear()*.

The use of the built-in object procedure *del()* destroys the data object. It is still accessible, but undefined. The object function *def()* is used to check whether an object is defined or not. For the initial state of a data object this function returns the value *true*.

The built-in object function *type()* returns the current type of the addressed object as a string, e.g. "int" for built-in types or "test" for user-defined types.

7.3 Allocation of data objects

All data objects are stored on the program stack. Nepal does not use heap storage or garbage collection. Therefore, memory leaks are not possible. To support dynamic (recursive) data structures on the stack, the assignment operator "~" is introduced to allow for the movement of objects (An example for using the movement operator is given in section 7.6). If a scope is entered more than once before exiting it (e.g. within recursive procedures), additional data objects are created and pushed onto the stack. Therefore, the objects from the first traversal are still stored, but not accessible before exiting the scope for the current traversal.

The allocation of attributes of a data object for a user-defined type depends on the question if the corresponding types are *recursive* or not. This property can be either direct or indirect and is determined by the Nepal interpreter automatically, i.e. no special syntax is necessary. All built-in types are non-recursive. Some examples are given below.

```
type A { int n }  # non-recursive type
type B { int n; B b }   # recursive type (direct)
type C { int n; D d }   # recursive type (indirect)
type D { int n; C c }   # recursive type (indirect)
```

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

Attributes of data objects corresponding to non-recursive and recursive types are initially allocated as empty and undefined objects, respectively.

If a data object corresponds to a user-defined type containing base types, their corresponding attributes are allocated in the same way as for the main type. If a certain base type is inherited multiple times, the allocation is done only once for this type.

7.4 Initialisation of data objects

Usually the attributes of a data object are initialised as either empty or undefined objects. This depends on the question if the attributes correspond to non-recursive or recursive types, respectively (cf. section 7.3). It is possible to apply a *basic* initialisation for variables of user-defined types if these variables correspond to certain built-in types (*bool*, *char*, *int*, *real*, *str, oper, file, dir* and *any*). This is achieved by assigning constant values inside the definition of a user-defined type:

```
<type> <var>(<const>)
```

The basic initialisation is applied to every object of this type just after its creation. The initial state of the object is also empty (i.e. the built-in procedure *empty()* yields true). After clearing a non-empty object (using the built-in procedure *clear()*), its state becomes empty and the attributes with basic initialisations will contain the initial values again. An example for this behaviour is given below.

```
type my_list {  # data type for a list

  int n(0);  # size of the list; basic initialisation since an empty list
             # always contains zero elements
  ...
}

my_list l; # empty list; attribute n is zero
l.clear(); # now attribute n is still zero
```

Special object procedures to initialise a data object can be defined inside the definition of a user-defined type:

```
proc "" { ... } # initialisation procedure without arguments
proc "" (...){ ... } # initialisation procedure with arguments
```

These procedures are executed with the statement of the data definition:

```
<type> <var1>(), <var2>(any a21, ...);
```

The Nepal interpreter replaces this statement by the following equivalent statement:

```
<type> <var1>.""(), <var2>.""(any a21, ...);
```

An example is given below.

```
type test {      # user-defined type

  int n;
  proc "" { n = 0 }
  proc "" (int xn) { n = xn }
}

test n1; # define variable with no special initialisation
test n2(); # define var. with special initialisation; the attribute is 0
test n3(7); # define var. with special initialisation; the attribute is 7
```

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

7.5 Access of data objects

The access to the data objects of a certain scope is done via the corresponding variables. An example for accessing a data object corresponding to a built-in type is given below.

```
int n = 10;      # define variable and assign value
n += 3;          # modify variable using arithmetic operator
```

The variables contained in user-defined types correspond to attributes of the data objects. The operator "." is used to access these attributes. Also procedures and functions defined in the corresponding (built-in or user-defined) type are accessed via this operator. An example is given below.

```
type test {    # user-defined type

  int n;
  proc set_n (int xn) { n = xn }
}

test t;          # define variable of user-defined type
t.n = 10         # modification of attribute by assignment (copy)
t.set_n(20);     # modification of attribute by procedure
```

The addressed variables, procedures and functions of a data object are searched firstly in the corresponding type and secondly in the base types due to breath first – depth second order. To restrict the search to the scope of a certain base type, the type access operator ":" can be used as follows:

```
<name of object>.<base type>:<name of element>
```

Here the type itself can also be used as base type.

Inside the procedures and functions of a data object, all types, variables, procedures and functions of the corresponding type as well as base types are accessible without restriction. The object itself can be accessed via the built-in variable *this* as follows:

```
this
```
or
```
this.<name of element>
```
or
```
this.<base type>:<name of element>
```

Note that the usage of variable *this* will raise an error, if it is used outside of a type definition. An example is given below.

```
proc p { this.n = 10 }     # error is raised since
                           # data object is not accessible

type test {       # user-defined type

  int n;
  proc "" { p() } # call external procedure during initialisation
}

test t(); #  define variable with special initialisation
```

Multiple data objects with multiple elements can be accessed via a single "."-operator as follows:

```
(<object_1>,..,<object_m>).(<element_1>,...,<element_n>)
```

The order of evaluation or execution is <object_1>.<element_1>, <object_1>.<element_2>, …, object_1>.<element_n>, …, <object_m>.<element_1>, <object_m>.<element_2>, …, object_m>.<element_n>.

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

7.6 Data contents

The content of a (defined) data object is either a *value* or a *reference*. Initially the content of a data object is always a value. A reference can only be obtained by using the assignment operator “@”. A reference is a kind of a pointer to a data object. Concatenated references are not possible. However, multiple references pointing to the same object are allowed. Initially the defined attributes of a data object are values. However, a certain object can be contained only once as an attribute representing a value. This leads to tree structures with respect to values. To model more complex data structures like circular lists or general graphs, a mixture of values and references is applied. In the following example, a graph structure consisting of nodes is established according to Figure 1. Values and references are represented by solid and dashed arrows, respectively. Both the movement operator “~” and the reference operator “@” are used.

```
# A graph demo

type node { # node of a graph
  int id;     # identifier
  node l, r;  # left and right sub-graph

  proc “” (int xid) { id = xid } # initialiser
}

node n0(0),n1(1),n2(2),n3(3),n4(4);
n0.l ~ n1;         # move n1 to n0.l
n0.r ~ n2;         # move n2 to n0.r
n0.l.r ~ n3;       # move n3 to n0.l.r
n0.r.l @ n0.l.r;   # point n0.r.l to n0.l.r
n0.r.r @ n4;       # point n0.r.r to n4

node n5 @ n0.l;    # point n5 to n0.l
node n6 @ n0;      # point n6 to n0
```

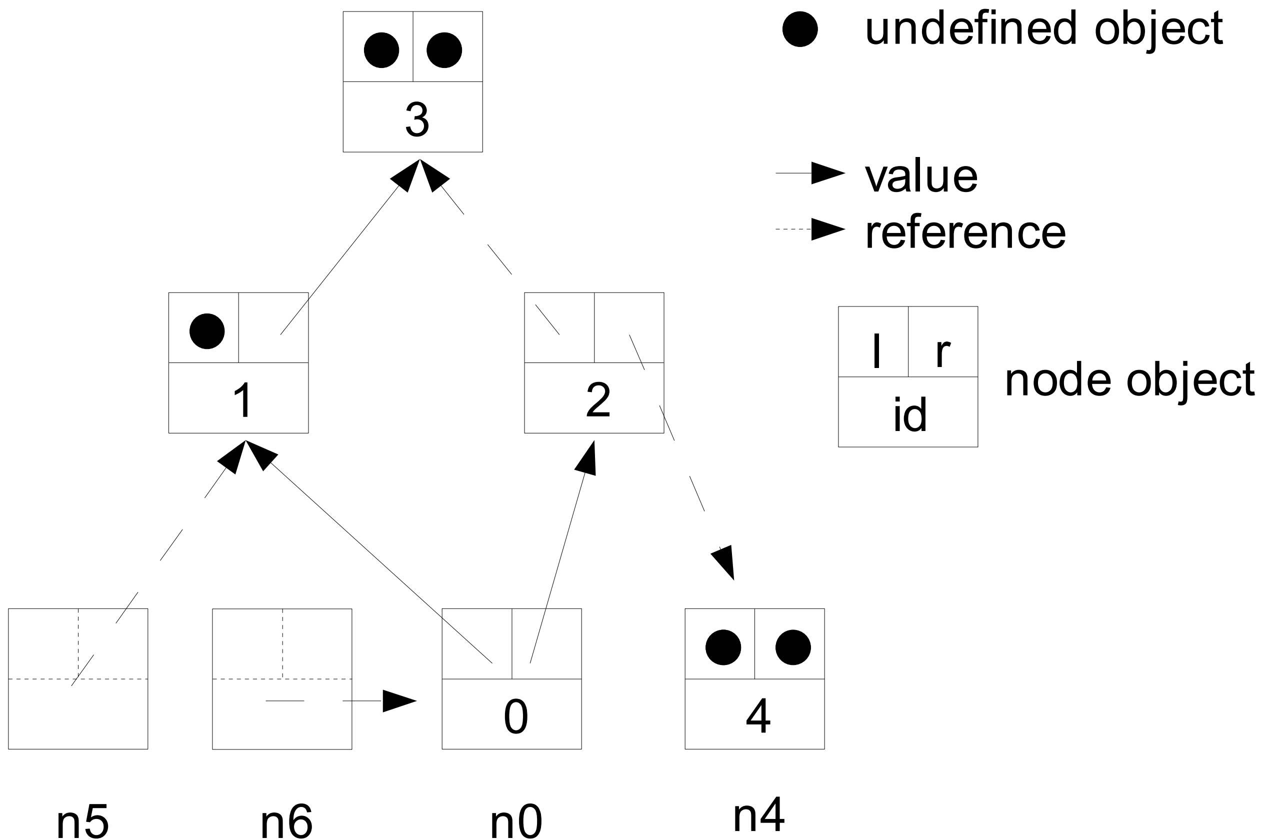

Figure 1: Graph model

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

Clearing the graph n0 by applying the statement “n0.clear()“ makes the data object empty. All attributes corresponding to non-recursive and recursive types are cleared and destroyed, respectively. The resulting data model is shown in Figure 2. References pointing to objects to be destroyed are transformed into empty objects (see node n5). Objects referenced from objects to be destroyed remain unchanged (see node n4).

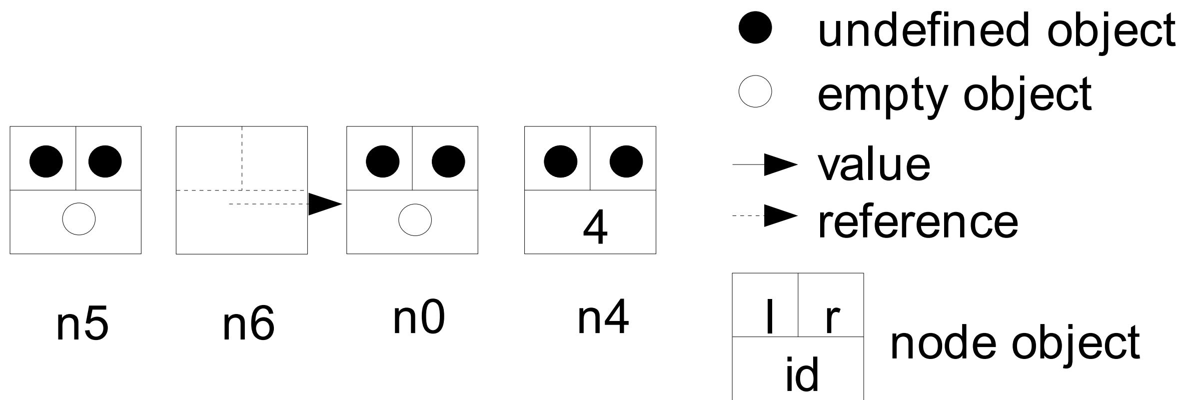

Figure 2: Graph model after clearing n0

Applying the object procedure *del()* instead of *clear()* produces an undefined object n0. The resulting data model is shown in Figure 3. Also node n6 is transformed into an empty object, since it was pointing to an object to be destroyed.

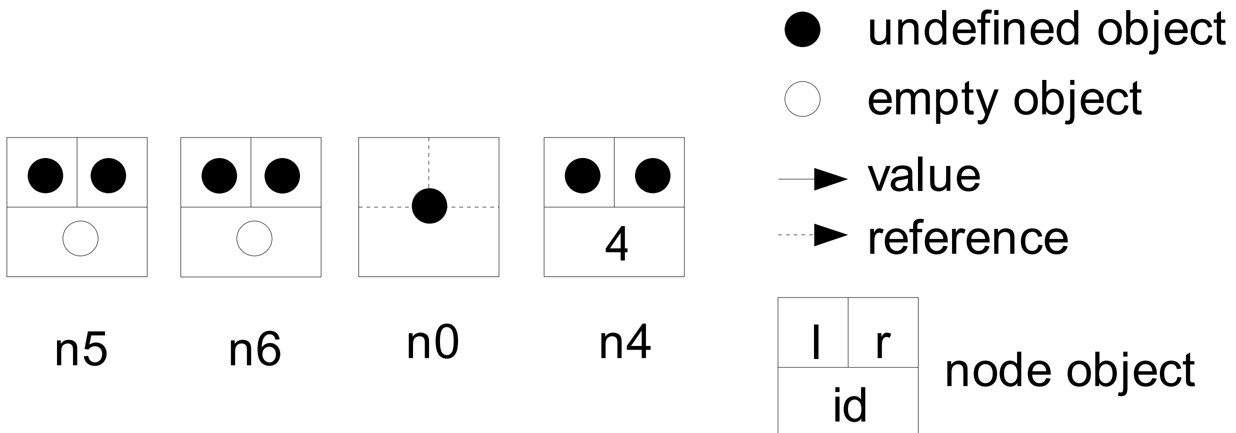

Figure 3: Graph model after destroying n0

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

7.7 Data assignments

Nepal supports assignments of the form

```
<variable> <assignment_operator> <expression>.
```

The following assignment operators are available for all built-in and user-defined types:

- "=" A (deep) copy of the data object on the right-hand side is assigned to the variable on the left-hand side.
- "@" A reference of the data object on the right-hand side is assigned to the variable on the left-hand side.
- "$" An alias of the data object on the right-hand side is assigned to the variable on the left-hand side.
- "?" Either a copy or reference is assigned to the variable on the left-hand side, depending on the question if the data object on the right-hand side contains a value or reference.
- "~" The data object on the right-hand side is moved to the variable.

In section 11.3.1 these operators will be described in more detail.

Even multiple assignments using only one statement are possible as described in section 11.3.1.6.

To demonstrate the assignment operators, the example from section 7.6 is reused. Figure 4 shows the result after applying the statement "`node n7 = n0`". As it can be seen, all contained values are copied recursively. The internal references (here from node 2 to node 3) are also copied completely. Outgoing references (here from node 2 to node 4) are copied in such a way that they point to the same objects as the original references. Ingoing references (here from node 5 to node 1 and from node 6 to node 0) are ignored.

Applying the operator "@" was already demonstrated in section 7.6. Applying the operator "?" to node n0 would have the same effect as the operator "=", since node n0 contains a value, not a reference.

Finally the statement "`node n7 ~ n0`" would result in a data model as shown in Figure 5. As it can be seen, all references point to the same data objects as before the movement. The object n0 is undefined now.

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

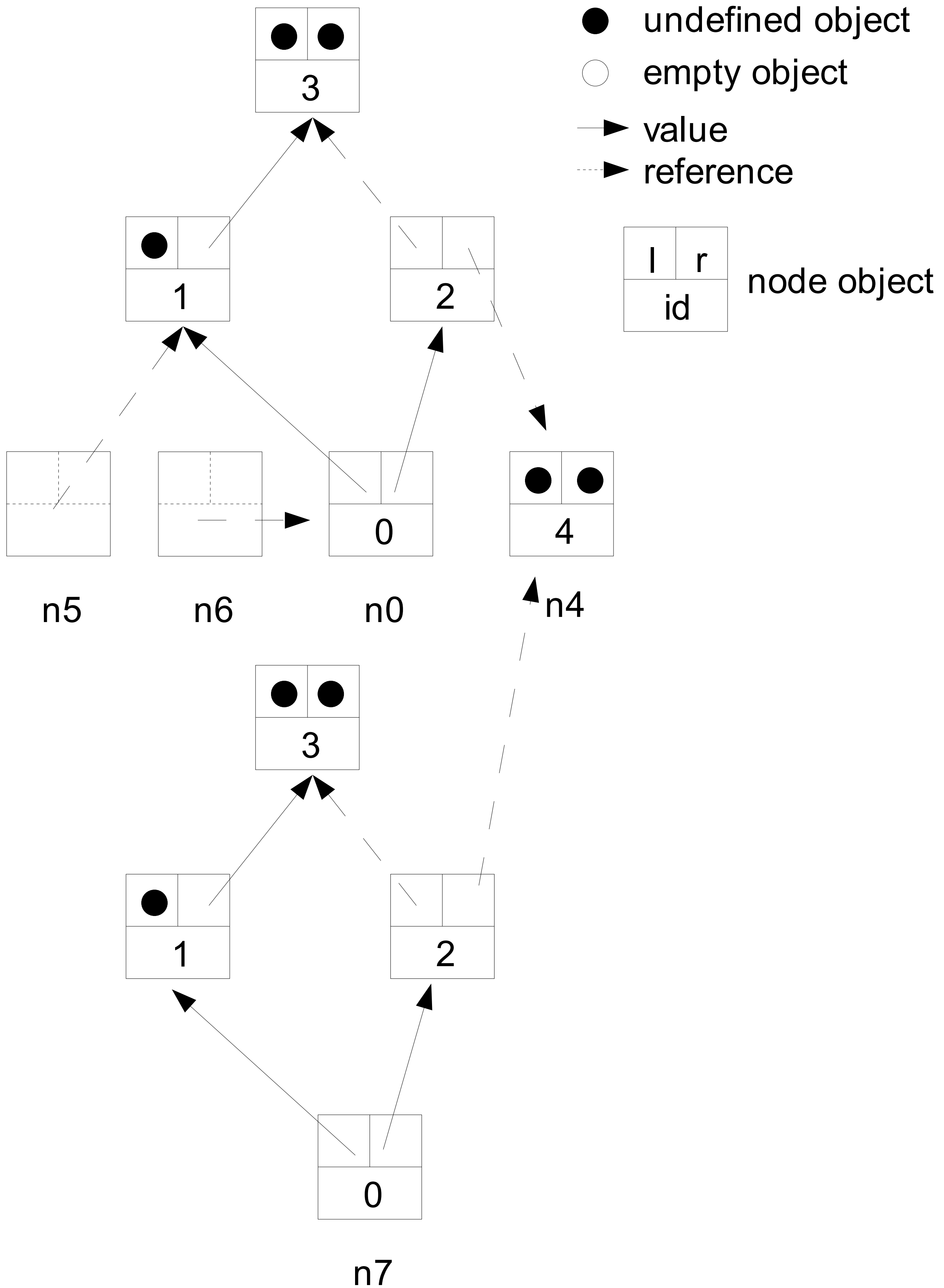

Figure 4: Graph model after copying n0

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

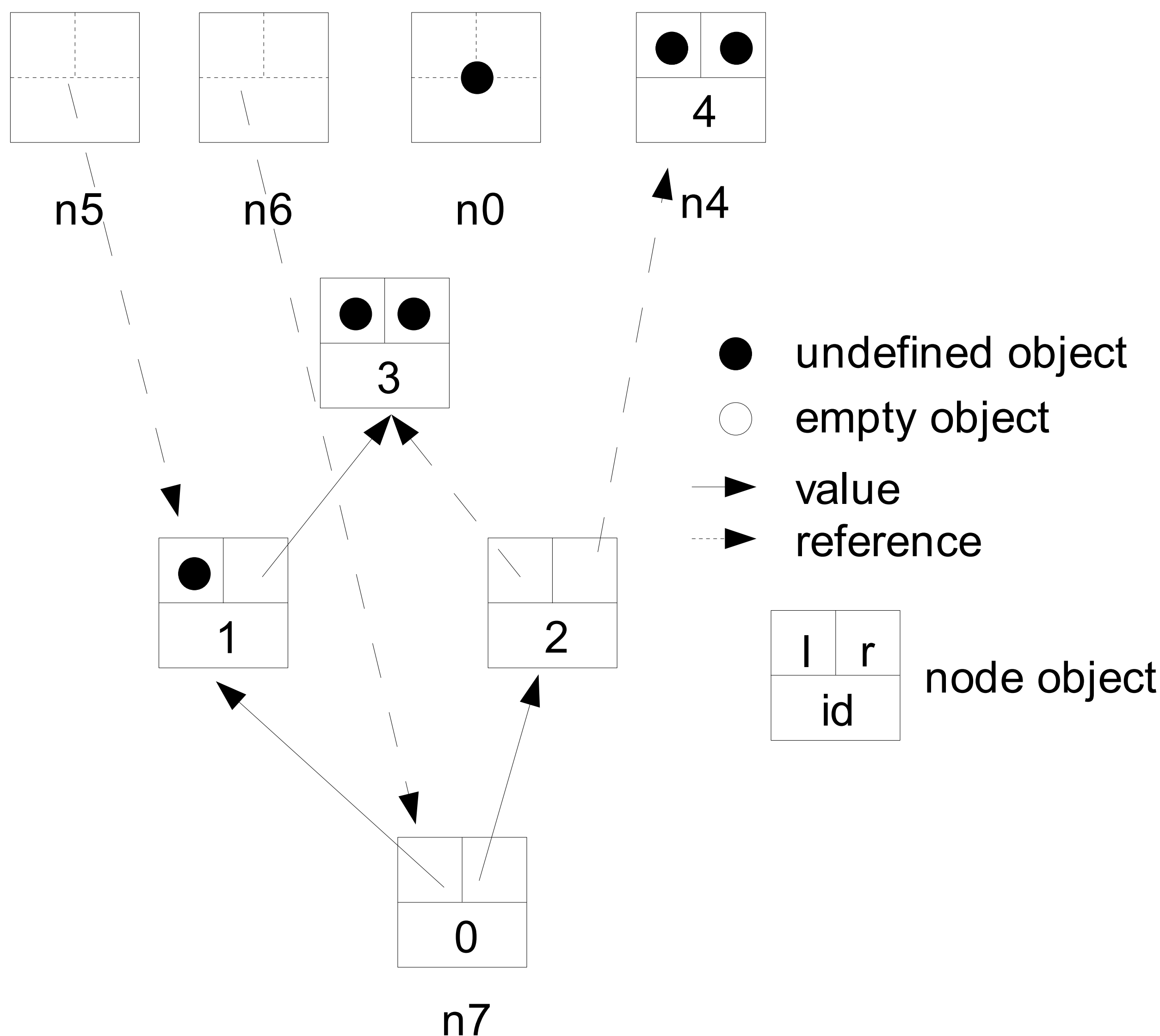

Figure 5: Graph model after moving n0

7.8 Data comparison

Nepal supports comparisons of the form

```
<expression> <comparison_operator> <expression>.
```

Primarily the following comparison operators are available for all built-in and user-defined types:

"==" Returns true iff the values of the two expressions are identical. The operator corresponds to the assignment operator "=": The statement "a=b" always implies "a==b". This means that the following conditions must hold for the comparison of user-defined types (cf. section 7.7):
- Attributes containing values must be identical.
- Attributes containing internal references must point to the corresponding objects.
- Attributes containing external references must point to the same objects.

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

"@@" Returns true iff the address of the two expressions are identical. The operator corresponds to the assignment operator "@": The statement "a@b" always implies "a@@b".

"??" Returns true iff the two expressions contain either identical values or identical references. The operator corresponds to the assignment operator "?": The statement "a?b" always implies "a??b".

Additionally the operators „!=", „!@", and „!?" are available. They return „true" if the corresponding operators return „false", and vice versa.

Finally an object function

```
func "<" (any a)(bool) { ... }
```

can be defined for any user-defined type. This supports the Nepal system functions *min()* and *max()* as well as the procedures for sorting objects of the built-in types *list*, *args*, *set*, *hash*, and *array*. The definition of additional functions "<=", ">", and ">=" is not necessary since they are derived from the function "<" and the built-in operator "==". An example is given below.

```
type test {
  int n;

  proc "" (int xn) { n = xn }
  func "<" (test t)(bool r) { r = n < t.n }
}

test t1(1), t2(2);
outl(t1 < t2);          # the output is "true"
outl(min(t1,t2));       # the output is "<1>".
```

7.9 Data Input and Output

For the input and output of data, Nepal follows a uniform concept applicable for the three built-in types *sys* (the runtime system with access to the console), *str* (sequence of characters) and *file* (handle for read and write access to a file). For a complete list of procedures and functions available for these types, see sections 8.4.18, 8.4.5 and 8.4.7.

7.9.1 Data output

The standard output procedure is

```
proc out (any v1, v2, ..., vn)
```

which writes the data v1 through vn to the output medium sequentially. An extra separator between the individual data can be defined by the type procedure

```
proc set_out_spc (str s)
```

Normally no separator is set. To reset the separator the procedure set_out_spc() is used with an empty string. The current separator can be accessed via the type function

```
func get_out_spc (str s)
```

The output of data with a temporary set separator can be achieved by the procedure

```
proc sout (str s; any v1, ..., vn)        # spaced output
```

For a formatted output the following type procedure can be used:

```
proc set_out_fmt (str f)
```

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

The format string has the form “(+|-<width>)+” where a positive and negative width stands for a right and left alignment, respectively. For example, the statements

```
set_out_fmt("-5+4");
out("how","are","you")
```

would yield the output

```
"how   areyou  "
```

As it can be seen, the format is repeated periodically if the number of arguments exceeds the format length. If the output data does not fit into the format specification, the data will be nevertheless written completely. Using the extended type procedure

```
proc set_out_fmt (char c, str f)
```

an additional filling character c can be set. Normally the filling character is blank. For example, the statements

```
set_out_fmt('*',"-5+4");
out("how","are","you")
```

would yield the output

```
"how***areyou**"
```

The current output format can be accessed by the type function

```
func get_out_fmt (char c, str f)
```

The output of data with temporary set separator and/or format can be done by the procedures

```
proc sout (str s; any v1, v2, ..., vn)          # spaced output
proc fout (str f; any v1, v2, ..., vn)          # formatted output
proc fout (char c; str f; any v1, v2, ..., vn)  # formatted output
proc fsout (str f,s; any v1, v2, ..., vn) # formatted, spaced output
proc fsout (char c; str f,s; any v1, v2, ..., vn) # formatted, spaced
                                                     output
```

The usage of these procedures does not change the current setting of the type-specific separator or format.

The output of data with a final end-of-line character ‘\n’ is performed by the procedure

```
proc outl (any v1, v2, ..., vn)           # output with end-of-line
```

Corresponding procedures with temporary set separator or format are also available (soutl(), foutl(), and fsoutl()).

Writing data objects to a packed representation on the output medium is done by the procedure

```
proc pout (any v1, v2, ..., vn)           # packed output
```

After writing these data they can be read from the medium by the procedure pin(), see section 7.9.2. The packed input and output is available for all user-defined types as well as built-in types except *file*, *dir* and *code*.

Nepal supports the writing to an output medium for all built-in types except *code*. For writing a *real* argument the output precision can be set by the type procedure

```
proc set_out_prec (int prec)
```

Normally no output precision is set, i.e. the current precision of the internal representation is used. To reset the output precision the procedure set_out_prec() is used with no argument. For user-defined types a standard output procedure is used. For example, the following statements

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

```
type test {
            int n;
            str s;
            proc "" (int nn; str ss) { n=nn; s=ss }
          }

test t(17,"how are you");
outl(t)
```

yield the output

```
"<17,how are you>"
```

If a special output is required, a function

```
func out (str s)
```

must be defined for this type. This function transforms the internal data representation into the output string s.

7.9.2 Data input

The standard input procedure is

```
proc in (any :$ v1, v2, ..., vn)
```

which reads the data v1 through vn from the input medium sequentially. The operator ":$" specifies a call-by-alias of the arguments, cf. section 10.1. The reading of the first argument v1 is done until either one of the preset delimiters or the end of the input medium is reached. For the console, the end of the input medium is represented by the key ESC. The second argument x2 is read from the input medium after the first delimiter, and so on. If the end of the medium is reached and there are still arguments to be read, these arguments remain unchanged. The delimiters for reading are set by the type procedure

```
proc set_in_spc (str s1, s2, ..., sn)
```

Normally no delimiter is set. To reset the delimiters the procedure set_in_spc() is used with no arguments. The current delimiters can be accessed via the type function

```
func get_in_spc (str s1, s2, ..., sn)
```

The input of data with temporary set delimiters can be achieved by the procedure

```
proc sin (str s1, ..., sm; any :$ v1, ..., vn)       # spaced input
```

Here the distinction between arguments si, i = 1...m, and vj, j = 1...n, is made according to the variability of the transferred data. All data embodied by variables (from right to left) are assigned to the arguments v1 to vn. The remaining data (left of the left-most variable data) are assigned to the arguments s1 to sm. The use of procedure sin() does not change the current setting of the type-specific delimiters.

Reading a data object until the next end-of-line (character '\n') or the end of the medium can be done by the procedure

```
proc inl (any :$ v)     # read until end-of-line
```

Reading data objects from a packed representation on the input medium is done by the procedure

```
proc pin (any :$ v1, v2, ..., vn)   # packed input
```

Before reading these data they have to be written to the medium by the procedure pout(), see section 7.9.1. The packed input and output is available for all user-defined types as well as built-in types except *file*, *dir* and *code*.

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

The procedures in(), sin(), inl(), and pin() also have corresponding functions with a boolean return value. These functions yield true if all input arguments have been read successfully.

Nepal supports the reading from an input medium for the built-in types *bool*, *int*, *real*, *char*, *str*, and *any*. Reading a *any* argument always results in a string representation (of type *str*). To read an object of a user-defined type from an input medium, a procedure

```
proc in (str s)
```

must be defined for this type. This procedure transforms the input string s into the internal data representation.

The reading position can be reset to the beginning of the medium by the procedure

```
proc in_reset
```

This procedure is available for the types *file* and *str*, but not for the console. It is executed implicitly for all procedures and functions with write access (e.g. out()).

8 Types

8.1 Inheritance

The type system of Nepal supports multiple inheritance, i.e. a user-defined type can inherit all elements from one or more other user-defined types (the so-called "base types"). Multiple inheritance of a certain base type is eliminated automatically. Cycles within inherited types are not allowed. The search order for addressed elements of a type is "breath first – depth second".

Built-in types cannot be inherited by user-defined types, with one exception: The built-in type *any* represents the generic base type which is inherited by all other built-in types and all user-defined types. Therefore, the procedures and functions of type *any* can be used by objects of any type. Section 8.4.6 describes the elements of type *any* in more detail.

8.2 Small and big types

Nepal makes a distinction between small and big types.

When using data of small types as input arguments of procedures or functions, these data are always copied. When using these data as keys of hash arrays, they are compared with respect to their value. The following built-in types are small: *bool*, *int*, *real*, *char*, *str*, *oper* and *error*. Small user-defined types are specified with the keyword *smalltype*.
When using constant / variable data of big types as input arguments of procedures or functions, the value / reference of these data is transferred. When using these data as keys of hash arrays or elements of sets, they are compared with respect to their value / (memory) address. The following built-in types are big: *list*, *set*, *hash*, *array*, *array2*, *code*, *file*, *dir*, *any* and *args*. Big user-defined types are specified with the keyword *bigtype*.

A more detailed description of the calling conventions for procedures and functions is given in section 10.1.

8.3 Access of type procedures and functions

Procedures and functions of a type can be called directly using the type access operator ":":

```
<type name>:<name of procedure or function>
```

Note that a corresponding data object does not exist for this call when it is used outside of a type definition. Therefore, accessing a variable of the type will raise an error. An example is given below.

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

```
bigtype test {      # user-defined type

  int n;
  proc "" { n = 0 } # initialisation procedure
}

test:""(); # error is raised since object does not exist
```

8.4 Built-in types

In the following sections the elements of all built-in types are described in more detail. The used groups have the following meaning:

- Object procedures/functions — Procedures/functions of a type which can be applied only by a certain data object. They use other procedures or functions of the object or do access attributes of the object.
- Type procedures/functions — Procedures/functions of a type which can be applied by any data object of this type as well as without a concrete object. They do not use object procedures or functions and do not access object attributes.
- Procedural object operators — Object procedures with zero or one input argument which can be applied only by a certain data object and are represented by a special symbol, e.g. "+=" for addition/concatenation. The first argument of the operator is the object itself.
- Functional operators — Functions with one or two input arguments, represented by a special symbol, e.g. "==" for comparison of two data objects.

8.4.1 Logical data

Syntax: `bool`
Meaning: Logical value.
Constants: `true, false`
Object procedures:

`"" (bool a)`	Initialisation procedure. Object is set to a copy of a.

Functional operators:

`"!" (bool a)(bool)`	Returns the logical negation of value a.
`"||" (bool a,b)(bool)`	Returns true iff a or b is true. b is not evaluated if a is true.
`"&&" (bool a,b)(bool)`	Returns true iff a and b is true. b is not evaluated if a is false.

8.4.2 Numerical data (integer)

Syntax: `int`
Meaning: Numerical integer value of arbitrary length.
Constants: `[+|-](0-9)+`
Type functions:

`random (int a,b)`	Returns a random number between a and b.

Object procedures:

`"" (int a)`	Initialisation procedure. Object is set to a copy of a.

Object functions:

`fac (int)`	Returns factorial of object.
`binom (int a)(int)`	Returns "object choose a" (binomial coefficient).
`gcd (int a)(int)`	Returns greatest common divisor of object and a.

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

lcm (int a)(int)	Returns least common multiple of object and a.
prime (bool)	Returns true iff object is a prime number
abs (int)	Returns absolute value of object.
sgn (int)	Returns sign of object (either +1, 0, or -1).
real (real)	Transforms object into a real value (all digits right of decimal point are all zero).

Procedural object operators:

“-“	Negates the object.
“+=” (int a)	Adds a to object.
“-=” (int a)	Subtracts a from object.
“*=” (int a)	Multiplies object with a.
“/=” (int a)	Divides object by a.
“%=” (int a)	Sets object to remainder when dividing object by a.
“^=” (int a)	Sets object to the power of a.

Functional operators:

“+” (int a,b)(int)	Returns sum of a and b.
“-“ (int a,b)(int)	Returns difference of a and b.
“*” (int a,b)(int)	Returns product of a and b.
“/” (int a,b)(int)	Returns quotient of a and b.
“%” (int a,b)(int)	Returns remainder when dividing a by b (modulo operator).
“^” (int a,b)(int)	Returns a to the power of b.
“<” (int a,b)(bool)	Returns true iff a is smaller than b.
“>” (int a,b)(bool)	Returns true iff a is greater than b.
“<=” (int a,b)(bool)	Returns true iff a is smaller than or equal to b.
“>=” (int a,b)(bool)	Returns true iff a is greater than or equal to b.
“..” (int a,b)(list)	Returns the sequence of integers from a to b.
“.:” (int a,b)(list)	Returns the increasing sequence of integers from a to b. If a > b, the sequence is empty.
“:.” (int a,b)(list)	Returns the decreasing sequence of integers from a to b. If a < b, the sequence is empty.

8.4.3 Numerical data (fix point)

Syntax: real

Meaning: Numerical real value with arbitrary number of digits left of decimal point and fixed number of digits right of decimal point (= precision). Normally the precision is 8. It can be modified with the type procedure *set_prec()*.

Constants: <int>.<int >= 0>, <int>[.<int >= 0>]e|E<int>,
[+|-].<int >= 0>e|E<int>

Type procedures:

set_prec (int)	Sets the precision of real numbers. This precision is valid for all subsequent initialisations and calculations involving real numbers.
set_out_prec (int)	Sets the precision of real numbers for all subsequent output procedures onto the output medium.
set_out_prec	Resets the precision of real numbers for all subsequent output procedures onto the output medium. This means that the output of a real number corresponds to the current representation of the number. Normally the output precision is reset.

Type functions:

get_prec (int)	Returns the current precision for real numbers.
get_out_prec (int)	Returns the current output precision for real numbers. If the precision is reset, an empty int value is returned.

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

`pi (real)` Returns the circle number.

Object procedures:

`“” (real a)` Initialisation procedure. Object is set to a copy of a.

Object functions:

`sqrt (real)` Returns square root of object.
`sin (real)` Returns sine of object.
`cos (real)` Returns cosine of object.
`tan (real)` Returns tangent of object.
`cot (real)` Returns cotangent of object.
`exp (real)` Returns exponential function of object.
`ln (real)` Returns natural logarithm of object.
`ld (real)` Returns logarithm of object for base 2.
`lg (real)` Returns logarithm of object for base 10.
`log (real a)(real)` Returns logarithm of object for base a.
`pot (real a)(real)` Returns object to the power of a.
`crt (real)` Returns cubic root of object.
`root (real a)(real)` Returns a-th root of object.
`sinh (real)` Returns hyperbolic sine of object.
`cosh (real)` Returns hyperbolic cosine of object.
`tanh (real)` Returns hyperbolic tangent of object.
`coth (real)` Returns hyperbolic cotangent of object.
`arcsin (real)` Returns inverse sine of object.
`arccos (real)` Returns inverse cosine of object.
`arctan (real)` Returns inverse tangent of object.
`arccot (real)` Returns inverse cotangent of object.
`arsinh (real)` Returns inverse hyperbolic sine of object.
`arcosh (real)` Returns inverse hyperbolic cosine of object.
`artanh (real)` Returns inverse hyperbolic tangent of object.
`arcoth (real)` Returns inverse hyperbolic cotangent of object.
`abs (real)` Returns absolute value of object.
`sgn (int)` Returns sign of object (either +1, 0, or -1).
`int (int)` Transforms object into an integer value (truncate digits right of decimal point).

Procedural object operators:

`“-“` Negates the object.
`“+=” (int|real a)` Adds a to object.
`“-=” (int|real a)` Subtracts a from object.
`“*=” (int|real a)` Multiplies object with a.
`“/=” (int|real a)` Divides object by a.

Functional operators:

`“+” (real a,b)(real)` Returns sum of a and b.
`“-“ (real a,b)(real)` Returns difference of a and b.
`“*” (real a,b)(real)` Returns product of a and b.
`“/” (real a,b)(real)` Returns quotient of a and b.
`“<” (real a,b)(bool)` Returns true iff a is smaller than b.
`“>” (real a,b)(bool)` Returns true iff a is greater than b.
`“<=” (real a,b)(bool)` Returns true iff a is smaller than or equal to b.
`“>=” (real a,b)(bool)` Returns true iff a is greater than or equal to b.

8.4.4 Characters

Syntax: `char`
Meaning: Character (Ascii).
Constants: `'(a-z)', '(A-Z)', '(0-9)', '-', '+', ..., '\n' (end-of-line), '\t' (horizontal tab), '\f' (form feed), '\v' (vertical tab), '\\'`

Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

(backslash), '\'' (apostrophe), '\"' (quotation mark), '\0' (null character)

Object procedures:

`“” (char a)`	Initialisation procedure. Object is set to a copy of a.
`toupper`	Transforms the object into an upper-case character.
`tolower`	Transforms the object into a lower-case character.

Object functions:

`isupper (bool)`	Returns true iff object is an upper-case character.
`islower (bool)`	Returns true iff object is a lower-case character.
`isalpha (bool)`	Returns true iff object is an alpha-numeric character.
`isdigit (bool)`	Returns true iff object is a character between ‘0’ and ‘9’.

Functional operators:

`“<” (char a,b)(bool)`	Returns true iff a is smaller than b.
`“>” (char a,b)(bool)`	Returns true iff a is greater than b.
`“<=” (char a,b)(bool)`	Returns true iff a is smaller than or equal to b.
`“>=” (char a,b)(bool)`	Returns true iff a is greater than or equal to b.
`“..” (char a,b)(list)`	Returns the sequence of characters from a to b.
`“.:” (char a,b)(list)`	Returns the increasing sequence of characters from a to b. If a > b, the sequence is empty.
`“:.” (char a,b)(list)`	Returns the decreasing sequence of characters from a to b. If a < b, the sequence is empty.

8.4.5 Strings

Syntax: `str`

Meaning: Sequence of (Ascii) characters with arbitrary length.

Constants: `"(<char>)+"`

Type procedures:

`set_in_spc (str s1, ...)`	Sets the delimiter(s) for the reading functions, e.g. in() and inl(). Normally no delimiter is defined.
`set_in_spc`	Resets the delimiters for the reading functions, e.g. in() and inl().
`set_out_spc (str s)`	Sets the separator for the writing functions, e.g. out() and outl(). Normally the separator is empty.
`set_out_fmt (str f)`	Sets the format for the writing functions, e.g. out() and outl().The filling character is set to blank (‘ ‘). Normally the format is empty.
`set_out_fmt (char c; str f)`	Sets the format f and filling character c for the writing functions, e.g. out() and outl(). Normally the filling character is blank and the format is empty.

Type functions:

`“” (any, ...)(str)`	Initialisation function. Writes the arguments sequentially to the string to be returned. The current format and separator (set by set_out_fmt() and set_out_spc() for type *str*) are considered accordingly.
`get_in_spc (str s1, ...)`	Returns the current delimiter(s) for the reading functions, e.g. in() and inl().
`get_out_spc (str s)`	Returns the current separator for the writing functions, e.g. out() and outl().
`get_out_fmt (char c; str f)`	Returns the current format f and filling character c for the writing functions, e.g. out() and outl().

Object procedures:

`“” (any, ...)`	Initialisation procedure. Clears the string and writes the arguments sequentially to the string. The current format and separator (set by set_out_fmt() and set_out_spc()) are considered accordingly.
`set (int i1,char c1, ...)`	Replaces character at index i1 by value c1 (and so on). The index (>= 0) is counted from the beginning of the string.

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

`Set (int i1,char c1, ...)`	Replaces character at index i1 by value c1 (and so on). The index (>= 0) is counted backwards from the end of the string.
`setp (int i,j; str s)`	Replaces the sub-string from index i to index j by string s. The index (>= 0) is counted from the beginning of the string.
`Setp (int i,j; str s)`	Replaces the sub-string from index i to index j by string s. The index (>= 0) is counted backwards from the end of the string.
`repl (str a,b)`	Replaces the sub-string a by sub-string b (only once). The search is done from the beginning of the string.
`Repl (str a,b)`	Replaces the sub-string a by sub-string b (only once). The search is done backwards from the end of the string.
`repl_all (str a,b)`	Replaces the sub-string a by sub-string b (multiple times). The search is done from the beginning of the string.
`Repl_all (str a,b)`	Replaces the sub-string a by sub-string b (multiple times). The search is done backwards from the end of the string.
`in (any :$ v1, ...)`	Reads the arguments sequentially from the string. The current delimiters (set by set_in_spc()) are considered accordingly. If the string is empty or the end of the string is reached, the procedure has no effect.
`sin (str s1,...; any :$ v1,...)`	Reads the arguments v1,... sequentially from the string using the delimiters s1,... . The delimiters used for procedure in() remain unchanged. If the string is empty or the end of the string is reached, the procedure has no effect.
`inl (any :$ v)`	Reads argument v until character '\n' or the end of the string. If the string is empty or the end of the string is reached, the procedure has no effect.
`pin (any :$ v1, ...)`	Reads the packed arguments sequentially from the string. Corresponds to procedure pout(). If the string is empty or the end of the string is reached, the procedure has no effect.
`out (any v1, ...)`	Writes the arguments sequentially to the string. The current format and separator (set by set_out_fmt() and set_out_spc()) are considered accordingly.
`outl (any v1, ...)`	Writes the arguments sequentially to the string and finally appends the character '\n'. The current format and separator (set by set_out_fmt() and set_out_spc()) are considered accordingly.
`sout (str s; any v1, ...)`	Writes the arguments sequentially to the string using the separator s. The current format (set by set_out_fmt()) is considered accordingly. The separator used for procedures out() and outl() remains unchanged.
`soutl (str s; any v1...)`	Writes the arguments sequentially to the string using the separator s, and finally appends the character '\n'. The current format (set by set_out_fmt()) is considered accordingly. The separator used for procedures out() and outl() remains unchanged.
`fout (str f; any v1, ...)`	Writes the arguments sequentially to the string using the format f. The current separator (set by set_out_spc()) is considered accordingly. The format used for procedures out() and outl() remains unchanged.
`foutl (str f; any v1...)`	Writes the arguments sequentially to the string using the format f, and finally appends the character '\n'. The current space (set by set_out_spc()) is considered accordingly. The format used for procedures out() and outl() remains unchanged.

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

`fout (char c; str f; any v1, ...)`	Writes the arguments sequentially to the string using the format f and filling character c. The current separator (set by set_out_spc()) is considered accordingly. The filling character and format used for procedures out() and outl() remain unchanged.
`foutl (char c; str f; any v1...)`	Writes the arguments sequentially to the string using the format f and filling character c, and finally appends the character '\n'. The current separator (set by set_out_spc()) is considered accordingly. The filling character and format used for procedures out() and outl() remain unchanged.
`fsout (str f; str s; any v1, ...)`	Writes the arguments sequentially to the string using the format f and separator s. The format and separator used for procedures out() and outl() remain unchanged.
`fsoutl (str f; str s; any v1...)`	Writes the arguments sequentially to the string using the format f and separator s, and finally appends the character '\n'. The format and separator used for procedures out() and outl() remain unchanged.
`fsout (char c; str f; str s; any v1, ...)`	Writes the arguments sequentially to the string using the format f, filling character c and separator s. The filling character, format and separator used for procedures out() and outl() remain unchanged.
`fsoutl (char c; str f; str s; any v1...)`	Writes the arguments sequentially to the string using the format f, filling character c, and separator s, and finally appends the character '\n'. The filling character, format and separator used for procedures out() and outl() remain unchanged.
`pout (any v1, ...)`	Writes the arguments sequentially to the string using a packed format. Corresponds to procedure pin().
`in_reset`	Reset the reading position to the beginning of the string (for all reading procedures and functions). This procedure is executed implicitly for all procedures / functions with write access.
`toupper`	Transforms all characters of the string into upper-case characters.
`tolower`	Transforms all characters of the string into lower-case characters.
`ins0 (str s1,...)`	Appends the arguments sequentially at the beginning of the string.
`Ins0 (str s1,...)`	Appends the arguments sequentially at the end of the string.
`ins (int i1; int s1; ...)`	Inserts the argument s1 left of the i1-th character (and so on). The index (>= 0) is counted from the beginning of the string.
`Ins (int i1; int s1; ...)`	Inserts the argument s1 right of the i1-th character (and so on). The index (>= 0) is counted backwards from the end of the string.
`del0`	Removes the first character from the string.
`Del0`	Removes the last character from the string.
`delp (int i,j)`	Removes the sub-string from index i to index j. The index (>= 0) is counted from the beginning of the string.
`Detp (int i,j)`	Removes the sub-string from index i to index j. The index (>= 0) is counted backwards from the end of the string.
`del (int i1, ...)`	Removes the character at index i1 (and so on). The index (>= 0) is counted from the beginning of the string.
`Del (int i1, ...)`	Removes the character at index i1 (and so on). The index (>= 0) is counted backwards from the end of the string.

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

`flip`	Inverts the string.
`chop`	Removes all white-space characters (‘ ‘, ‘\t’, ‘\f’, ‘\v’, ‘\n’, '\0') at the beginning of the string.
`Chop`	Removes all white-space characters (‘ ‘, ‘\t’, ‘\f’, ‘\v’, ‘\n’, '\0') at the end of the string.
`chop (str s1, ...)`	Removes all arguments at the beginning of the string.
`Chop (str s1, ...)`	Removes all arguments at the end of the string.
`for (char :$ c; code C)`	Iterates over all characters c of the string (using copy assignment) and executes code C for each character.
`For (char :$ c; code C)`	Iterates over all characters c of the string (traversing it backwards and using copy assignment) and executes code C for each character.

Object functions:

`size (int)`	Returns current length of string.
`get0 (char)`	Returns first character of string.
`Get0 (char)`	Returns last character of string.
`get (int i1, ...)(char c1, ...)`	Returns character with index i1 (and so on).
`Get (int i1, ...)(char c1, ...)`	Returns character with index i1 (and so on). The index (>= 0) is counted backwards from the end of the string.
`str (int i1, ...)(str)`	Returns a string containing character with index i1 (and so on).
`getp (int i, j)(str)`	Returns sub-string from index i to index j.
`Getp (int i, j)(str)`	Returns sub-string from index i to index j. The index (>= 0) is counted backwards from the end of the string.
`find_str (str s1, ...)(args)`	Searches sub-string s1 (and so on) from the beginning of the string and returns first position (one output argument). The index (>= 0) is counted from the beginning of the string. Returns zero arguments, if no sub-string is found.
`Find_str (str s1, ...)(args)`	Searches sub-string s1 (and so on) from the end of the string and returns first position (one output argument). The index (>= 0) is counted from the beginning of the string. Returns zero arguments, if no sub-string is found.
`find_str_all (str s1, ...)(args)`	Searches sub-string s1 (and so on) from the beginning of the string and returns all found positions. The indices (>= 0) are counted from the beginning of the string.
`Find_str_all (str s1, ...)(args)`	Searches sub-string s1 (and so on) from the end of the string and returns all found positions. The indices (>= 0) are counted from the beginning of the string.
`find (any :$ v; code C)(args)`	Searches for the first character where code C evaluates to the boolean value “true” and returns its position. The index (>= 0) is counted from the beginning of the string. Returns an empty list of arguments if no character was found. For each character of the string a copy is assigned to argument v. During this iteration the code evaluation is done for the currently assigned character.
`Find (any :$ v; code C)(args)`	Searches for the first character where code C evaluates to the boolean value “true” and returns its position. The index (>= 0) is counted from the beginning of the string. The search is done backwards from the end of the string. Returns an empty list of arguments if no character was found. For each character of the string a copy is assigned to argument v. During this iteration the code evaluation is done for the currently assigned character.

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

`find_all (any :$ v; code C)(args)` Searches for all characters where code C evaluates to the boolean value "true" and returns their positions. The indices (>= 0) are counted from the beginning of the string. Returns an empty list of arguments if no character was found. For each character of the string an alias is assigned to argument v. During this iteration the code evaluation is done for the currently assigned character.

`Find_all (any :$ v; code C)(args)` Searches for all characters where code C evaluates to the boolean value "true" and returns their positions. The indices (>= 0) are counted from the beginning of the string. The search is done backwards from the end of the string. Returns an empty list of arguments if no character was found. For each character of the string a copy is assigned to argument v. During this iteration the code evaluation is done for the currently assigned character.

`in (any :$ v1, ...)(bool)` Reads the arguments sequentially from the string. The current delimiters (set by set_in_spc()) are considered accordingly. If the string is empty or the end of the string is reached, the procedure has no effect. Returns true iff all arguments are read successfully.

`sin (str s1,...; any :$ v1,...)(bool)` Reads the arguments v1,... sequentially from the string using the delimiters s1,... . The delimiters used for procedure in() remain unchanged. If the string is empty or the end of the string is reached, the procedure has no effect. Returns true iff all arguments are read successfully.

`inl (any :$ v)(bool)` Reads argument v until character '\n' or the end of the string. If the string is empty or the end of the string is reached, the procedure has no effect. Returns true iff the arguments is read successfully.

`pin (any :$ v1, ...)(bool)` Reads the packed arguments sequentially from the string. Corresponds to procedure pout(). If the string is empty or the end of the string is reached, the procedure has no effect. Returns true iff all arguments are read successfully.

`split (str v1, ...)` Splits the string into sub-strings using all white-space characters (' ', '\t', '\f', '\v', '\n', '\0') as delimiters.

`split (str s1, ...)(str v1, ...)` Splits the string into sub-strings using the string s1 (and so on) as delimiters.

`anysplit (int|real|char|str v1, ...)` Splits the string into sub-strings using all white-space characters (' ', '\t', '\f', '\v', '\n', '\0') as delimiters. Then transforms the sub-strings into integer, real or character values if possible. Otherwise strings are returned.

`anysplit (str s1, ...)(int|real|char|str v1, ...)` Splits the string into sub-strings using the string s1 (and so on) as delimiters. Then transforms the sub-strings into integer, real or character values if possible. Otherwise strings are returned.

`args (char c1, ...)` Splits the string into individual characters.

`int (int)` Transforms the string into integer value if possible. Otherwise a system error is raised.

`real (real)` Transforms the string into real value if possible. Otherwise a system error is raised.

`isupper (bool)` Returns true iff every character of the string is an upper-case character.

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

`islower (bool)`	Returns true iff every character of the string is a lower-case character.
`isalpha (bool)`	Returns true iff every character of the string is an alphanumeric character.
`isdigit (bool)`	Returns true iff every character of the string is a character between ‘0’ and ‘9’.
`isint (bool)`	Returns true iff the string can be transformed into an integer value.
`isreal (bool)`	Returns true iff the string can be transformed into a real value.
`for (char :$ c; code C)(args)`	Iterates over all characters c of the string (using copy assignment) and returns the evaluated code C for each character.
`For (char :$ c; code C)(args)`	Iterates over all characters c of the string (traversing it backwards and using copy assignment) and returns the evaluated code C for each character.

Procedural object operators:

`“+=” (str a)`	Appends a to object.

Functional operators:

`“+” (str a,b)(str)`	Returns concatenation of a and b.
`“<” (str a,b)(bool)`	Returns true iff a is smaller than b.
`“>” (str a,b)(bool)`	Returns true iff a is greater than b.
`“<=” (str a,b)(bool)`	Returns true iff a is smaller than or equal to b.
`“>=” (str a,b)(bool)`	Returns true iff a is greater than or equal to b.

8.4.6 Polymorphic data

Syntax: `any`

Meaning: Objects of this type can store data of arbitrary type. Base type for all other built-in types and all user-defined types.

Type functions:

`“” (any, ...)(any, ...)`	Initialisation function. Returns list of (constant) values from list of (constant or variable) arguments.

Object procedures:

`del`	Destroys the current object. The object is no longer defined, i.e. function def() returns true now.
`clear`	Clears the current object. The function empty() returns true now.
`proc (str type, name; any v1, ...)`	Calls the object procedure “name” of base type “type” with the arguments v1 (and so on). If the type is empty, every base type (including the current type) is considered.
`func (str type, name; any v1, …)`	Calls the object function “name” of base type “type” with the arguments v1 (and so on) and moves the result into the current object. If the type is empty, every base type (including the current type) is considered.

Object functions:

`def (bool)`	Returns true iff current object is defined.
`empty (bool)`	Returns true iff current object is empty. For user-defined types, a comparison is made to a dummy object for which a standard allocation and the default initialisation (cf. section 8.5) have been applied.
`type (str)`	Returns the type name of the current object.
`func (str type, name; any v1, ...)(any, ...)`	Calls the object function “name” of base type “type” with the arguments v1 (and so on). If the type is empty, every base type (including the current type) is considered.

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

`proc (str type, name; any v1, ...)(any)`	Makes a copy of the current object, calls the object procedure “name” of base type “type” with the arguments v1 (and so on) for the copied object, and returns this object. If the type is empty, every base type (including the current type) is considered.

Functional operators:

`"==" (any a,b)(bool)`	Compares the value of a and b. Returns true iff a and b are identical.
`"!=" (any a,b)(bool)`	Compares the value of a and b. Returns true iff a and b are not identical.
`"@@" (any a,b)(bool)`	Compares the address of a and b. Returns true iff the address of a equals to the address of b.
`"!@" (any a,b)(bool)`	Compares the address of a and b. Returns true iff the address of a differs from the address of b.
`"??" (any a,b)(bool)`	Compares the value or address of a and b. Returns true iff a and b contain identical values or references.
`"!?" (any a,b)(bool)`	Compares the value or address of a and b. Returns true iff a and b do not contain identical values or references.

8.4.7 Files

Syntax: `file`

Meaning: File handle for read and write access.

Type procedures:

`set_in_spc (str v1, ...)`	Sets the delimiter(s) for the reading functions, e.g. in() and inl(). Normally no delimiter is defined.
`set_in_spc`	Resets the delimiters for the reading functions, e.g. in() and inl().
`set_out_spc (str s)`	Sets the separator for the writing functions, e.g. out() and outl(). Normally the separator is empty.
`set_out_fmt (str f)`	Sets the format for the writing functions, e.g. out() and outl().The filling character is set to blank (‘ ‘). Normally the format is empty.
`set_out_fmt (char c; str f)`	Sets the format f and filling character c for the writing functions, e.g. out() and outl(). Normally the filling character is blank and the format is empty.

Type functions:

`"" (str)(file)`	Initialisation function. Returns a handle for a file with the specified name.
`get_in_spc (str s1, ...)`	Returns the current delimiter(s) for the reading functions, e.g. in() and inl().
`get_out_spc (str s)`	Returns the current separator for the writing functions, e.g. out() and outl().
`get_out_fmt (char c; str f)`	Returns the current format f and filling character c for the writing, e.g. functions out() and outl().

Object procedures:

`"" (str f)`	Specifies the name of the file with f.
`"" (str f, m)`	Specifies the name of the file with f and opens the file with mode m (“i” for reading access, “o” for writing access, “O” for appending access, “ib” for reading access to binary files, “ob” for writing access to binary files, “Ob” for appending access to binary files).
`open (str m)`	Opens the file with mode m (“i” for reading access, “o” for writing access, “O” for appending access, “ib” for reading access to binary files, “ob” for writing access to binary files, “Ob” for appending access to binary files).
`close`	Close any open stream.
`remove`	Removes file.

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

`move (file f)`	Renames file to file f.
`make`	Creates the file.
`copy (file f)`	Copies file to target file f.
`in (any :$ v1, ...)`	Reads the arguments sequentially from the file. The current delimiters (set by set_in_spc()) are considered accordingly. If the file is empty or the end of the file is reached, the procedure has no effect.
`sin (str s1,...; any :$ v1,...)`	Reads the arguments v1,... sequentially from the file using the delimiters s1,... . The delimiters used for procedure in() remain unchanged. If the file is empty or the end of the file is reached, the procedure has no effect.
`inl (any :$ v)`	Reads argument v until character '\n' or the end of the file. If the file is empty or the end of the file is reached, the procedure has no effect.
`pin (any :$ v1, ...)`	Reads the packed arguments sequentially from the file. Corresponds to procedure pout(). If the file is empty or the end of the file is reached, the procedure has no effect.
`out (any, ...)`	Writes the arguments sequentially to the file. The current format and separator (set by set_out_fmt() and set_out_spc()) are considered accordingly.
`outl (any, ...)`	Writes the arguments sequentially to the file and finally appends the character '\n'. The current format and separator (set by set_out_fmt() and set_out_spc()) are considered accordingly.
`sout (str s; any v1, ...)`	Writes the arguments sequentially to the file using the separator s. The current format (set by set_out_fmt()) is considered accordingly. The separator used for procedures out() and outl() remains unchanged.
`soutl (str s; any v1...)`	Writes the arguments sequentially to the file using the separator s, and finally appends the character '\n'. The current format (set by set_out_fmt()) is considered accordingly. The separator used for procedures out() and outl() remains unchanged.
`fout (str f; any v1, ...)`	Writes the arguments sequentially to the file using the format f. The current separator (set by set_out_spc()) is considered accordingly. The format used for procedures out() and outl() remains unchanged.
`foutl (str f; any v1...)`	Writes the arguments sequentially to the file using the format f, and finally appends the character '\n'. The current separator (set by set_out_spc()) is considered accordingly. The format used for procedures out() and outl() remains unchanged.
`fout (char c; str f; any v1, ...)`	Writes the arguments sequentially to the file using the format f and filling character c. The current separator (set by set_out_spc()) is considered accordingly. The filling character and format used for procedures out() and outl() remain unchanged.
`foutl (char c; str f; any v1...)`	Writes the arguments sequentially to the file using the format f and filling character c, and finally appends the character '\n'. The current separator (set by set_out_spc()) is considered accordingly. The filling character and format used for procedures out() and outl() remain unchanged.
`fsout (str f; str s; any v1, ...)`	Writes the arguments sequentially to the file using the format f and separator s. The format and separator used for procedures out() and outl() remain unchanged.

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

`fsoutl (str f; str s; any v1...)` Writes the arguments sequentially to the file using the format f and separator s, and finally appends the character ‘\n’. The format and separator used for procedures out() and outl() remain unchanged.

`fsout (char c; str f; str s; any v1, ...)` Writes the arguments sequentially to the file using the format f, filling character c and separator s. The filling character, format and separator used for procedures out() and outl() remain unchanged.

`fsoutl (char c; str f; str s; any v1...)` Writes the arguments sequentially to the file using the format f, filling character c, and separator s, and finally appends the character ‘\n’. The filling character, format and separator used for procedures out() and outl() remain unchanged.

`pout (any v1, ...)` Writes the arguments sequentially to the file using a packed format. Corresponds to procedure pin().

`in_reset` Reset the reading position to the beginning of the file (for all reading procedures and functions). This procedure is executed implicitly for all procedures / functions with write access.

`for (char :$ c; code C)` Iterates over all characters c of the file (using copy assignment) and executes code C for each line.

`For (char :$ c; code C)` Iterates over all characters c of the file (traversing it backwards and using copy assignment) and executes code C for each line.

`for (str :$ s; code C)` Iterates over all lines s of the file (using copy assignment) and executes code C for each line.

`For (str :$ s; code C)` Iterates over all lines s of the file (traversing it backwards and using copy assignment) and executes code C for each line.

Object functions:

`exists (bool)` Returns true iff file exists.

`size (int)` Returns size of file (in bytes).

`cmp (file f)(bool)` Compares content of file with content of file f. Returns true iff contents are identical.

`in (any :$ v1, ...)(bool)` Reads the arguments sequentially from the file. The current delimiters (set by set_in_spc()) are considered accordingly. If the file is empty or the end of the file is reached, the procedure has no effect. Returns true iff all arguments are read successfully.

`sin (str s1,...; any :$ v1,...)(bool)` Reads the arguments v1,... sequentially from the file using the delimiters s1,... . The delimiters used for procedure in() remain unchanged. If the file is empty or the end of the file is reached, the procedure has no effect. Returns true iff all arguments are read successfully.

`inl (any :$ v)(bool)` Reads argument v until character ‘\n’ or the end of the file. If the file is empty or the end of the file is reached, the procedure has no effect. Returns true iff the argument is read successfully.

`pin (any :$ v1, ...)(bool)` Reads the packed arguments sequentially from the file. Corresponds to procedure pout(). If the file is empty or the end of the file is reached, the procedure has no effect. Returns true iff all arguments are read successfully.

`split (str v1, ...)` Splits the file into sub-strings using all white-space characters (‘ ‘, ‘\t’, ‘\f’, ‘\v’, ‘\n’, '\0') as delimiters.

`split (str s1, ...)(str v1, ...)` Splits the file into sub-strings using the string s1 (and so on) as delimiters.

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

`anysplit (int|real|char|str v1, ...)`	Splits the file into sub-strings using all white-space characters (' ', '\t', '\f', '\v', '\n', '\0') as delimiters. Then transforms the sub-strings into integer, real or character values if possible. Otherwise strings are returned.
`anysplit (str s1, ...)(int|real|char|str v1, ...)`	Splits the file into sub-strings using the string s1 (and so on) as delimiters. Then transforms the sub-strings into integer, real or character values if possible. Otherwise strings are returned.
`for (char :$ c; code C)(args)`	Iterates over all characters c of the file (using copy assignment) and returns the evaluated code C for each line.
`For (char :$ c; code C)(args)`	Iterates over all characters c of the file (traversing it backwards and using copy assignment) and returns the evaluated code C for each line.
`for (str :$ s; code C)(args)`	Iterates over all lines s of the file (using copy assignment) and returns the evaluated code C for each line.
`For (str :$ s; code C)(args)`	Iterates over all lines s of the file (traversing it backwards and using copy assignment) and returns the evaluated code C for each line.

8.4.8 Directories

Syntax: `dir`

Meaning: Handle for directory (folder).

Type functions:

`"" (str)(dir)`	Initialisation function. Returns a handle for a directory with the specified name.

Object procedures:

`"" (str d)`	Specifies the name of the directory with d.
`remove`	Removes directory.
`move (dir d)`	Renames directory to name d.
`make`	Creates the directory.
`copy (dir d)`	Copies directory to target directory d.
`for (file :$ f; bool rec; str patt1,...; code C)`	Iterates over all files of the directory and executes code C for each file. Only files are considered which match any of the patterns patt1 (and so on). The search is recursive iff the argument rec is true.
`For (file :$ f; bool rec; str patt1,...; code C)`	Iterates over all files of the directory (traversing it backwards) and executes code C for each file. Only files are considered which match any of the patterns patt1 (and so on). The search is recursive iff the argument rec is true.
`for (dir :$ d; bool rec; str patt1,...; code C)`	Iterates over all directories of the directory and executes code C for each directory. Only directories are considered which match any of the patterns patt1 (and so on). The search is recursive iff the argument rec is true.
`For (dir :$ d; bool rec; str patt1,...; code C)`	Iterates over all directories of the directory (traversing it backwards) and executes code C for each directory. Only directories are considered which

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

match any of the patterns patt1 (and so on). The search is recursive iff the argument rec is true.

Object functions:

`exists (bool)`	Returns true iff directory exists.
`size (int)`	Returns size of directory (in bytes).
`for (file :$ f; bool rec; str patt1,...; code C)(args)`	Iterates over all files of the directory and returns the evaluated code C for each file. Only files are considered which match any of the patterns patt1 (and so on). The search is recursive iff the argument rec is true.
`For (file :$ f; bool rec; str patt1,...; code C)(args)`	Iterates over all files of the directory (traversing it backwards) and returns the evaluated code C for each file. Only files are considered which match any of the patterns patt1 (and so on). The search is recursive iff the argument rec is true.
`for (dir :$ d; bool rec; str patt1,...; code C)(args)`	Iterates over all directories of the directory and returns the evaluated code C for each directory. Only directories are considered which match any of the patterns patt1 (and so on). The search is recursive iff the argument rec is true.
`For (dir :$ d; bool rec; str patt1,...; code C)(args)`	Iterates over all directories of the directory (traversing it backwards) and returns the evaluated code C for each directory. Only directories are considered which match any of the patterns patt1 (and so on). The search is recursive iff the argument rec is true.

8.4.9 Lists

Syntax: `list`

Meaning: Ordered, polymorphic list of arbitrary length.

Type functions:

`"" (any, ...)(list)`	Initialisation function. Inserts the arguments - either as value or reference – at the end of the list to be returned.

Object procedures:

`"" (any v1; ...)`	Initialisation procedure. Inserts argument v1 - either as value or reference depending on the question if the type of the transferred data is small or big (cf. section 8.2) – at the end of the list (and so on).
`set (int i1; any v1; ...)`	Sets element at position i1 to argument v1 – either value or reference (and so on). The index (>= 0) is counted from the beginning of the list.
`Set (int i1; any v1; ...)`	Sets element at position i1 to argument v1 – either value or reference (and so on). The index (>= 0) is counted backwards from the end of the list.
`ins (int i1; any v1; ...)`	Inserts argument v1 - either as value or reference - before element at position i1 (and so on). The index (>= 0) is counted from the beginning of the list.
`Ins (int i1; any v1; ...)`	Inserts argument v1 - either as value or reference - after element at position i1 (and so on). The index (>= 0) is counted backwards from the end of the list.
`ins0 (any v1; ...)`	Inserts argument v1 - either as value or reference – at the beginning of the list (and so on).
`Ins0 (any v1; ...)`	Inserts argument v1 - either as value or reference – at the end of the list (and so on).
`del0`	Removes the first element.
`Del0`	Removes the last element.

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

`del (int i1; ...)`	Removes element at position i1 (and so on). The index (>= 0) is counted from the beginning of the list.
`Del (int i1; ...)`	Removes element at position i1 (and so on). The index (>= 0) is counted backwards from the end of the list.
`flip`	Inverts the list.
`sort`	Sorts the list in ascending order.
`Sort`	Sorts the list in descending order.
`sort (any :$ v; code C)`	Sorts the list in ascending order according to the evaluated code C. For each element of the list an alias is assigned to argument v. During this iteration the code evaluation is done for the currently assigned element.
`Sort (any :$ v; code C)`	Sorts the list in descending order according to the evaluated code C. For each element of the list an alias is assigned to argument v. During this iteration the code evaluation is done for the currently assigned element.
`for (any :$ v; code C)`	Iterates over all elements v of the list (using alias assignment) and executes code C for each element.
`For (any :$ v; code C)`	Iterates over all elements v of the list (traversing it backwards and using alias assignment) and executes code C for each element.

Object functions:

`size (int)`	Returns current length of list.
`get (int i1,...)(any v1,...)`	Returns content of element at position i1 – either value or reference (and so on). The index (>= 0) is counted from the beginning of the list.
`Get (int i1,...)(any v1,...)`	Returns content of element at position i1 – either value or reference (and so on). The index (>= 0) is counted backwards from the end of the list.
`get0 (any v)`	Returns content of first element – either value or reference.
`Get0 (any v)`	Returns content of last element – either value or reference.
`"[]" (int i1,...)(any :$ v1,...)`	Returns alias on element at position i1 (and so on).
`list (int i1,...)(list)`	Returns list of contents at position i1 – either valuess or references (and so on).
`find (any :$ v; code C)(args)`	Searches for the first element where code C evaluates to the boolean value "true" and returns its position. The index (>= 0) is counted from the beginning of the list. Returns an empty list of arguments if no element was found. For each element of the list an alias is assigned to argument v. During this iteration the code evaluation is done for the currently assigned element.
`Find (any :$ v; code C)(args)`	Searches for the first element where code C evaluates to the boolean value "true" and returns its position. The index (>= 0) is counted from the beginning of the list. The search is done backwards from the end of the list. Returns an empty list of arguments if no element was found. For each element of the list an alias is assigned to argument v. During this iteration the code evaluation is done for the currently assigned element.
`find_all (any :$ v; code C)(args)`	Searches for all elements where code C evaluates to the boolean value "true" and returns their positions. The indices (>= 0) are counted from the beginning of the list. Returns an empty list of arguments if no element was found. For each element of the list an alias is assigned to argument v. During this

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

	iteration the code evaluation is done for the currently assigned element.
`Find_all (any :$ v; code C)(args)`	Searches for all elements where code C evaluates to the boolean value “true” and returns their positions. The indices (>= 0) are counted from the beginning of the list. The search is done backwards from the end of the list. Returns an empty list of arguments if no element was found. For each element of the list an alias is assigned to argument v. During this iteration the code evaluation is done for the currently assigned element.
`args (any v1, ...)`	Returns the list of individual contents – either values or references.
`for (any :$ v; code C)(args)`	Iterates over all elements v of the list (using alias assignment) and returns the evaluated code C for each element.
`For (any :$ v; code C)(args)`	Iterates over all elements v of the list (traversing it backwards and using alias assignment) and returns the evaluated code C for each element.

Procedural object operators:

`“+=” (list a)`	Appends a to object (concatenation).

Functional operators:

`“+” (list a,b)(list)`	Returns concatenation of a and b.
`“<” (list a,b)(bool)`	Returns true iff a is smaller than b (i.e. the first n-1 elements are identical and the n-th element of a is smaller than the n-th element of b, n >= 1).
`“>” (list a,b)(bool)`	Returns true iff a is greater than b (i.e. the first n-1 elements are identical and the n-th element of a is greater than the n-th element of b, n >= 1).
`“<=” (list a,b)(bool)`	Returns true iff a is smaller than or equal to b.
`“>=” (list a,b)(bool)`	Returns true iff a is greater than or equal to b.

8.4.10 Hash arrays

Syntax: `hash`

Meaning: Associative, polymorphic array (set of key-content pairs).

Type functions:

`“” (any k1,c1, ...)(hash)`	Initialisation function. Inserts the key-content pair (k1,c1) – either as values or references - (and so on) into the hash array to be returned.

Object procedures:

`“” (any k1,c1, ...)`	Initialisation procedure. Inserts the key-content pair (k1,c1) – either as values or references depending on the question if the type of the transferred data is small or big (cf. Section 8.2) - (and so on).
`ins (any k1,c1, ...)`	Inserts the key-content pair (k1,c1) – either as values or references (and so on).
`set (any k1,c1, ...)`	Sets the content of key k1 to c1 – either as value or reference (and so on).
`del (any k1,...)`	Deletes pair for key k1 (and so on).
`for (any :$ k; code C)`	Iterates over all keys (using copy / reference assignment) and executes code C for each key.
`for (any :$ k,c; code C)`	Iterates over all pairs (key,content) – using copy / reference assignment for keys and alias assignment for contents) and executes code C for each pair.

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

`For (any :$ k; code C)`	Iterates over all keys (using copy / reference assignment and traversing the array backwards) and executes code C for each key.
`For (any :$ k,c; code C)`	Iterates over all pairs (key,content) – using copy / reference assignment for keys and alias assignment for contents and traversing the array backwards - and executes code C for each pair.
`sort`	Sorts the array with respect to the contents in ascending order.
`Sort`	Sorts the array with respect to the contents in descending order.
`sort (any :$ v; code C)`	Sorts the array in ascending order according to the evaluated code C. For each content of the array an alias is assigned to argument v. During this iteration the code evaluation is done for the currently assigned content.
`Sort (any :$ v; code C)`	Sorts the array in descending order according to the evaluated code C. For each content of the array an alias is assigned to argument v. During this iteration the code evaluation is done for the currently assigned content.
`sort_keys`	Sorts the array with respect to the keys in ascending order.
`Sort _keys`	Sorts the array with respect to the contents in descending order.
`sort_keys (any :$ v; code C)`	Sorts the array in ascending order according to the evaluated code C. For each key of the array an alias is assigned to argument v. During this iteration the code evaluation is done for the currently assigned key.
`Sort_keys (any :$ v; code C)`	Sorts the array in descending order according to the evaluated code C. For each key of the array an alias is assigned to argument v. During this iteration the code evaluation is done for the currently assigned key.

Object functions:

`size (int)`	Returns current number of pairs.
`def (any k1, ...)(bool b1, ...)`	Returns true iff pair for key k1 exists (and so on).
`get (any k1, ...)(any c1, ...)`	Returns content for key k1 - either as value or reference (and so on).
`"[]" (any k1,...)(any :$ c1,...)`	Returns alias on content of key k1 (and so on). If the key does not yet exist, a pair with empty content of type any is created.
`keys (any k1, ...)`	Returns keys for all defined pairs - either as values or references.
`find (any :$ v; code C)(args)`	Searches for the first content where code C evaluates to the boolean value "true" and returns its key. Returns an empty list of arguments if no content was found. For each content of the array an alias is assigned to argument v. During this iteration the code evaluation is done for the currently assigned content.
`Find (any :$ v; code C)(args)`	Searches for the first content where code C evaluates to the boolean value "true" and returns its key. The search is done backwards from the end of the array. Returns an empty list of arguments if no content was found. For each content of the array an alias is assigned to argument v. During this iteration the code evaluation is done for the currently assigned content.
`find_all (any :$ v; code C)(args)`	Searches for all contents where code C evaluates to the boolean value "true" and returns their keys. Returns an empty list of

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

arguments if no element was found. For each content of the array an alias is assigned to argument v. During this iteration the code evaluation is done for the currently assigned content.

`Find_all (any :$ v; code C)(args)` Searches for all contents where code C evaluates to the boolean value "true" and returns their keys. The search is done backwards from the end of the array. Returns an empty list of arguments if no content was found. For each content of the array an alias is assigned to argument v. During this iteration the code evaluation is done for the currently assigned content.

`for (any :$ k; code C)(args)` Iterates over all keys (using copy / reference assignment) and returns the evaluated code C for each key.

`for (any :$ k,c; code C)(args)` Iterates over all pairs (key,content) – using copy / reference assignment for keys and alias assignment for contents) and returns the evaluated code C for each pair.

`For (any :$ k; code C)(args)` Iterates over all keys (using copy / reference assignment and traversing the array backwards) and returns the evaluated code C for each key.

`For (any :$ k,c; code C)(args)` Iterates over all pairs (key,content) – using copy / reference assignment for keys and alias assignment for contents and traversing the array backwards - and returns the evaluated code C for each pair.

8.4.11 One-dimensional arrays

Syntax: `array`

Meaning: One-dimensional, polymorphic array of fixed length.

Type functions:

`"" (int n; any, ...)(array)` Initialisation function. Returns an array of length n built from the optional arguments – either as values or references.

Object procedures:

`"" (int n; any v1,...,vn)` Initialisation procedure. Allocates an array of length n with undefined elements. The optional arguments v1,...,vn are used to initialise the n elements.

`set (int i1; any v1, ...)` Sets the element with index i1 to argument v1 – either as value or reference depending on the question if the type of the transferred data is small or big (cf. Section 8.2) - (and so on).

`Set (int i1; any v1, ...)` Sets the element with index i1 to argument v1 (and so on). The index (>= 0) is counted backwards from the end of the array.

`del (int i1, ...)` Deletes the element with index i1 (and so on).

`Del (int i1, ...)` Deletes the element with index i1 (and so on). The index (>= 0) is counted backwards from the end of the array.

`for (int :$ i; code C)` Iterates over index of all defined elements and executes code C for each index.

`for (int :$ i; any :$ v; code C)` Iterates over pairs (index, content) of all defined elements – using alias assignment for the contents - and executes code C for each pair.

`For (int :$ i; code C)` Iterates over index of all defined elements – in reverse order - and executes code C for each index.

`For (int :$ i; any :$ v; code C)` Iterates over pairs (index, content) of all defined elements - in reverse order and using alias assignment for the contents - and executes code C for each pair.

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

`sort`	Sorts the array with respect to the contents in ascending order.
`Sort`	Sorts the array with respect to the contents in descending order.
`sort (any :$ v; code C)`	Sorts the array in ascending order according to the evaluated code C. For each content of the array an alias is assigned to argument v. During this iteration the code evaluation is done for the currently assigned content.
`Sort (any :$ v; code C)`	Sorts the array in descending order according to the evaluated code C. For each content of the array an alias is assigned to argument v. During this iteration the code evaluation is done for the currently assigned content.

Object functions:

`size (int)`	Returns length of array.
`def (int i1, ...)(bool b1, ...)`	Returns true iff elements with index i1 exists (and so on).
`get (int i1, ...)(any v1, ...)`	Returns element with index i1 – either as value or reference (and so on).
`Get (int i1, ...)(any v1, ...)`	Returns element with index i1 – either as value or reference (and so on). The index (>= 0) is counted backwards from the end of the array.
`"[]" (int i1,...)(any :$ v1,...)`	Returns alias on element with index i1 (and so on). If an element does not yet exist, the element is created with empty content of type any.
`keys (int i1, ...)`	Returns indices for all defined elements.
`find (any :$ v; code C)(args)`	Searches for the first content where code C evaluates to the boolean value "true" and returns its position. The index (>= 0) is counted from the beginning of the array. Returns an empty list of arguments if no content was found. For each content of the array an alias is assigned to argument v. During this iteration the code evaluation is done for the currently assigned content.
`Find (any :$ v; code C)(args)`	Searches for the first content where code C evaluates to the boolean value "true" and returns its position. The index (>= 0) is counted from the beginning of the array. The search is done backwards from the end of the array. Returns an empty list of arguments if no content was found. For each content of the array an alias is assigned to argument v. During this iteration the code evaluation is done for the currently assigned content.
`find_all (any :$ v; code C)(args)`	Searches for all contents where code C evaluates to the boolean value "true" and returns their positions. The indices (>= 0) are counted from the beginning of the array. Returns an empty list of arguments if no content was found. For each content of the array an alias is assigned to argument v. During this iteration the code evaluation is done for the currently assigned content.
`Find_all (any :$ v; code C)(args)`	Searches for all contents where code C evaluates to the boolean value "true" and returns their positions. The indices (>= 0) are counted from the beginning of the array. The search is done backwards from the end of the array. Returns an empty list of arguments if no content was found. For each content of the array an alias is assigned to argument v. During this iteration the code evaluation is done for the currently assigned content.

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

`for (int :$ i; code C)(args)` Iterates over index of all defined elements and returns the evaluated code C for each index.

`for (int :$ i; any :$ v; code C)(args)` Iterates over pairs (index, content) of all defined elements – using alias assignment for the contents – and returns the evaluated code C for each pair.

`For (int :$ i; code C)(args)` Iterates over index of all defined elements – in reverse order - and returns the evaluated code C for each index.

`For (int :$ i; any :$ v; code C)(args)` Iterates over pairs (index, content) of all defined elements - in reverse order and using alias assignment for the contents - and returns the evaluated code C for each pair.

8.4.12 Two-dimensional arrays

Syntax: `array2`

Meaning: Two-dimensional, polymorphic array of fixed length.

Type functions:

`"" (int m,n; any, ...)(array2)` Initialisation function. Returns an array of m rows and n columns built from the optional arguments – either as values or references.

Object procedures:

`"" (int m,n; any v1,...,vmn)` Initialisation procedure. Allocates an array of m rows and n columns with undefined elements. The optional arguments v1,..,vmn are used to initialise the m*n elements.

`set (int i1,i2; any v1, ...)` Set the element with index (i1,i2) to argument v1 – either as value or reference depending on the question if the type of the transferred data is small or big (cf. Section 8.2) - (and so on).

`Set (int i1,i2; any v1, ...)` Set the element with index (i1,i2) to argument v1 (and so on). The indices (>= 0) are counted backwards from the end of the array.

`del (int i1,i2, ...)` Deletes the element with index (i1,i2) (and so on).

`Del (int i1,i2, ...)` Deletes the element with index (i1,i2) (and so on). The indices (>= 0) are counted backwards from the end of the array.

`for (int :$ i1,i2; code C)` Iterates over index (i1,i2) of all defined elements and executes code C for each index.

`for (int :$ i1,i2; any :$ v; code C)` Iterates over pairs (index, content) of all defined elements – using alias assignment for the contents - and executes code C for each pair.

`For (int :$ i1,i2; code C)` Iterates over index of all defined elements – in reverse order - and executes code C for each index.

`For (int<>i1,i2; any<>v; code C)` Iterates over pairs (index, content) of all defined elements - in reverse order and using alias assignment for the contents - and executes code C for each pair.

Object functions:

`size (int,int)` Returns number of rows and columns of array.

`def (int i1,i2, ...)(bool b1, ...)` Returns true iff elements with index (i1,i2) exists (and so on).

`get (int i1,i2, ...)(any v1, ...)` Returns element with index (i1,i2) – either as value or reference (and so on).

`Get (int i1,i2, ...)(any v1, ...)` Returns element with index (i1,i2) – either as value or reference (and so on). The indices (>= 0) are counted backwards from the end of the array.

`"[]" (int i1,i2,...)(any :$ v1,...)` Returns alias on element with index (i1,i2) (and so on). If an element does not yet

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

exist, the element is created with empty content of type any.

`keys (int i1,i2, ...)`	Returns indices for all defined elements.
`find (any :$ v; code C)(args)`	Searches for the first content where code C evaluates to the boolean value “true” and returns its position. The indices (>= 0) are counted from the beginning of the array. Returns an empty list of arguments if no content was found. For each content of the array an alias is assigned to argument v. During this iteration the code evaluation is done for the currently assigned content.
`Find (any :$ v; code C)(args)`	Searches for the first content where code C evaluates to the boolean value “true” and returns its position. The indices (>= 0) are counted from the beginning of the array. The search is done backwards from the end of the array. Returns an empty list of arguments if no content was found. For each content of the array an alias is assigned to argument v. During this iteration the code evaluation is done for the currently assigned content.
`find_all (any :$ v; code C)(args)`	Searches for all contents where code C evaluates to the boolean value “true” and returns their positions. The indices (>= 0) are counted from the beginning of the array. Returns an empty list of arguments if no content was found. For each content of the array an alias is assigned to argument v. During this iteration the code evaluation is done for the currently assigned content.
`Find_all (any :$ v; code C)(args)`	Searches for all contents where code C evaluates to the boolean value “true” and returns their positions. The indices (>= 0) are counted from the beginning of the array. The search is done backwards from the end of the array. Returns an empty list of arguments if no content was found. For each content of the array an alias is assigned to argument v. During this iteration the code evaluation is done for the currently assigned content.
`for (int :$ i1,i2; code C)(args)`	Iterates over index (i1,i2) of all defined elements and returns the evaluated code C for each index.
`for (int :$ i1,i2; any :$ v; code C)(args)`	Iterates over pairs (index, content) of all defined elements – using alias assignment for the contents - and returns the evaluated code C for each pair.
`For (int :$ i1,i2; code C)(args)`	Iterates over index of all defined elements – in reverse order - and returns the evaluated code C for each index.
`For (int<>i1,i2; any<>v; code C)(args)`	Iterates over pairs (index, content) of all defined elements - in reverse order and using alias assignment for the contents - and returns the evaluated code C for each pair.

8.4.13 Sets

Syntax: `set`

Meaning: Polymorphic set.

Type functions:

`“” (any, ...)(set)`	Initialisation function. Inserts the arguments - either as value or reference – into the set to be returned.

Object procedures:

`“” (any v1, ...)`	Initialisation procedure. Inserts argument v1 – either as value or reference depending on the question if the type of the transferred data is small or big (cf. Section 8.2) - (and so on).
`ins (any v1, ...)`	Inserts argument v1 – either as value or reference (and so on).
`del (any v1, ...)`	Deletes argument v1 (and so on).
`for (any :$ v; code C)`	Iterates over all elements v (using alias assignment) and executes code C for each element.
`For (any :$ v; code C)`	Iterates over all elements v (using alias assignment and traversing the set backwards) and executes code C for each element.
`sort`	Sorts the set in ascending order.
`Sort`	Sorts the set in descending order.
`sort (any :$ v; code C)`	Sorts the set in ascending order according to the evaluated code C. For each element of the set an alias is assigned to argument v. During this iteration the code evaluation is done for the currently assigned element.
`Sort (any :$ v; code C)`	Sorts the set in descending order according to the evaluated code C. For each element of the set an alias is assigned to argument v. During this iteration the code evaluation is done for the currently assigned element.

Object functions:

`size (int)`	Returns number of elements.
`def (any v1, ...)(bool b1, ...)`	Returns true iff argument v1 exists (and so on).
`args (any v1, ...)`	Returns the list of individual contents – either as values or references.
`find (any :$ v; code C)(args)`	Searches for the first element where code C evaluates to the boolean value “true”. Returns an empty list of arguments if no element was found. For each element of the list an alias is assigned to argument v. During this iteration the code evaluation is done for the currently assigned element.
`Find (any :$ v; code C)(args)`	Searches for the first element where code C evaluates to the boolean value “true”. The search is done backwards from the end of the set. Returns an empty list of arguments if no element was found. For each element of the set an alias is assigned to argument v. During this iteration the code evaluation is done for the currently assigned element.
`find_all (any :$ v; code C)(args)`	Searches for all elements where code C evaluates to the boolean value “true”. Returns an empty list of arguments if no element was found. For each element of the set an alias is assigned to argument v. During this iteration the code evaluation is done for the currently assigned element.
`Find_all (any :$ v; code C)(args)`	Searches for all elements where code C evaluates to the boolean value “true”. The search is done backwards from the end of the set. Returns an empty list of arguments if no element was found. For each element of the set an alias is assigned to argument v. During this iteration the code evaluation is done for the currently assigned element.
`for (any :$ v; code C)(args)`	Iterates over all elements v (using alias assignment) and returns the evaluated code C for each element.
`For (any :$ v; code C)(args)`	Iterates over all elements v (using alias assignment and traversing the set backwards) and returns the evaluated code C for each element.

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

Procedural object operators:

- `"+=" (set m)` Unions the object with set m.
- `"*=" (set m)` Sets the object to the intersection with set m.
- `"-=" (set m)` Sets the object to the delta set with set m (“object without m”).

Functional operators:

- `"+" (set a,b)(set)` Returns the union of set a and b.
- `"*" (set a,b)(set)` Returns the intersection of sets a and b.
- `"-" (set a,b)(set)` Returns the delta set of sets a and b (“a without b”)
- `"<" (set a,b)(bool)` Returns true iff a is subset of b.
- `"<=" (set a,b)(bool)` Returns true iff a is subset of b or equal to b.
- `">" (set a,b)(bool)` Returns true iff b is subset of a.
- `">=" (set a,b)(bool)` Returns true iff b is subset of a or equal to a.

8.4.14 Errors

Syntax: `error`
Meaning: Object for error messages (also used for Nepal-internal errors).
Type functions:

- `"" (str)(error)` Initialisation function. Returns an error with the specified error text.

Object procedures:

- `"" (str)` Initialisation procedure. Sets the error text.

Object functions:

- `file (str)` Returns name of code file where the error was raised.
- `line (int)` Returns line number in code file where error was raised.
- `error_type (str)` Returns type of error („system“ or „user“, i.e. generated by Nepal interpreter or by user).

8.4.15 Operators

Syntax: `oper`
Meaning: Unary or binary operator.
Type functions:

- `"" (str)(oper)` Initialisation function. Returns an operator initialised by a string.

Object procedures:

- `"" (str)` Initialisation procedure using a string, e.g. “+” or “+=”.
- `exec (any v1, v2)` Executes the binary operator. Possible operators are “=”, “@”, “$”, “?”, “~”, “+=”, “-=”, “*=”, “/=”, “%=”, and “^=”.

Object functions:

- `eval (any v)(any r)` Evaluates the unary operator. Possible operators are “!”, “+”, and “-“.
- `eval (any v1,v2)(any r)` Evaluates the binary operator. Possible operators are “+”, “-“, “*”, “/”, “%”, “^”, “**”, “..”, “.:”, “:.”, “||”, “&&”, “<”, “>”, “<=”, “>=”, “==”, “!=”, “@@”, “!@”, “??” and “!?”.

8.4.16 Programming codes

Syntax: `code`
Meaning: Nepal programming code.
Constants: Any block, see section 12. E.g. “{ outl(“hello world”) }”.
Object procedures:

- `"" (code)` Initialisation procedure.
- `exec` Executes the programming code.

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

Object functions:

`eval (any r)`	Evaluates the programming code.

8.4.17 Arguments

Syntax: `args`
Meaning: List of arguments for procedures and functions.
Type functions:
The same as for type `list`.
Object procedures:
The same as for type `list`.
Object functions:
The same as for type `list`.
Procedural object operators:
The same as for type `list`.
Functional operators:
The same as for type `list`.
Functional object operators:
The same as for type `list`.

8.4.18 System data

Syntax: `sys`
Meaning: Represents the runtime system with access to the console.
Type procedures:

`in (any :$ v1, ...)`	Reads the arguments sequentially from the console. The current delimiters (set by set_in_spc()) are considered accordingly. Pressing the ESC key stops the input (stands for “end-of-file”).
`sin (str s1,...; any :$ v1,...)`	Reads the arguments v1,... sequentially from the console using the delimiters s1,... . The delimiters used for procedure in() remain unchanged. Pressing the ESC key stops the input (stands for “end-of-file”).
`inl (any :$ v)`	Reads argument v until character ‘\n’ or the key ESC.
`pin (any :$ v1, ...)`	Reads the packed arguments sequentially from the console. Corresponds to procedure pout().Pressing the ESC key stops the input (stands for “end-of-file”).
`out (any v1, ...)`	Writes the arguments sequentially to the console. The current format and separator (set by set_out_fmt() and set_out_spc()) are considered accordingly.
`outl (any v1, ...)`	Writes the arguments sequentially to the console and finally appends the character ‘\n’. The current format and separator (set by set_out_fmt() and set_out_spc()) are considered accordingly.
`sout (str s; any v1, ...)`	Writes the arguments sequentially to the console using the separator s. The current format (set by set_out_fmt()) is considered accordingly. The separator used for procedures out() and outl() remains unchanged.
`soutl (str s; any v1...)`	Writes the arguments sequentially to the console using the separator s, and finally appends the character ‘\n’. The current format (set by set_out_fmt()) is considered accordingly. The separator used for procedures out() and outl() remains unchanged.
`fout (str f; any v1, ...)`	Writes the arguments sequentially to the console using the format f. The current separator (set by set_out_spc()) is considered accordingly. The format used for procedures out() and outl() remains unchanged.

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

`foutl (str f; any v1...)`	Writes the arguments sequentially to the console using the format f, and finally appends the character '\n'. The current separator (set by set_out_spc()) is considered accordingly. The format used for procedures out() and outl() remains unchanged.
`fout (char c; str f; any v1, ...)`	Writes the arguments sequentially to the console using the format f and filling character c. The current separator (set by set_out_spc()) is considered accordingly. The filling character and format used for procedures out() and outl() remain unchanged.
`foutl (char c; str f; any v1...)`	Writes the arguments sequentially to the console using the format f and filling character c, and finally appends the character '\n'. The current separator (set by set_out_spc()) is considered accordingly. The filling character and format used for procedures out() and outl() remain unchanged.
`fsout (str f; str s; any v1, ...)`	Writes the arguments sequentially to the console using the format f and separator s. The format and separator used for procedures out() and outl() remain unchanged.
`fsoutl (str f; str s; any v1...)`	Writes the arguments sequentially to the console using the format f and separator s, and finally appends the character '\n'. The format and separator used for procedures out() and outl() remain unchanged.
`fsout (char c; str f; str s; any v1, ...)`	Writes the arguments sequentially to the console using the format f, filling character c and separator s. The filling character, format and separator used for procedures out() and outl() remain unchanged.
`fsoutl (char c; str f; str s; any v1...)`	Writes the arguments sequentially to the console using the format f, filling character c, and separator s, and finally appends the character '\n'. The filling character, format and separator used for procedures out() and outl() remain unchanged.
`pout (any v1, ...)`	Writes the arguments sequentially to the console using a packed format. Corresponds to procedure pin().
`set_in_spc (str s1, ...)`	Sets the delimiter(s) for the reading functions, e.g. in() and inl(). Normally no delimiter is defined.
`set_in_spc`	Resets the delimiters for the reading functions, e.g. in() and inl().
`set_out_spc (str s)`	Sets the separator for the writing functions, e.g. out() and outl(). Normally the separator is empty.
`set_out_fmt (str f)`	Sets the format for the writing functions, e.g. out() and outl().The filling character is set to blank (' '). Normally the format is empty.
`set_out_fmt (char c; str f)`	Sets the format f and filling character c for the writing functions, e.g. out() and outl(). Normally the filling character is blank and the format is empty.
`system (str)`	Calls the specified system procedure from the operating system.
`proc (str name; any v1, ...)`	Calls the procedure name with the arguments v1 (and so on).
`del (any v1, ...)`	Destroys the objects v1 (and so on). The objects are no longer defined, i.e. function def() returns true now. If the objects contain references, the objects themselves are destroyed, i.e. not the referenced objects. For destroying the referenced objects, the object function del() of type *any* is used.

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

`clear (any v1, ...)`	Clears the objects v1 (and so on). The function empty() returns true now. If the objects contain references, the objects themselves are cleared, i.e. not the referenced objects. For clearing the referenced objects, the object function clear() of type *any* is used.
`set_calc_time`	Resets the time for function get_calc_time() to 0.
`if (bool; code a)`	Control structure, see section 11.4.1
`if (bool; code a,b)`	Control structure, see section 11.4.1
`switch (any; code)`	Control structure, see section 11.4.1
`switch (code)`	Control structure, see section 11.4.1
`for (code a,b,c,d)`	Control structure, see section 11.4.2
`for (any :$ x; any y1,...;code)`	Control structure, see section 11.4.2
`for (code)`	Control structure, see section 11.4.2
`while (code a,b)`	Control structure, see section 11.4.2
`continue (int)`	Control structure, see section 11.4.2
`break (int)`	Control structure, see section 11.4.2
`return (int)`	Control structure, see section 11.4.3
`exit (int)`	Control structure, see section 11.4.4
`throw (any)`	Control structure, see section 11.4.5
`catch (any; code)`	Control structure, see section 11.4.5
`set_cwd (str)`	Sets the current working directory (absolute path).

Type functions:

`argc (int)`	Returns the number of arguments of the Nepal program (including the code file).
`argv (int i1, ...)(str s1, ...)`	Returns the arguments of the Nepal program (including the code file) for index i1 >= 0 (and so on). “argv(0)” yields the code file.
`system (str)(int)`	Calls the specified system function from the operating system and returns the result of this function.
`func (str name; any v1, ...)(any, ...)`	Calls the function name with the arguments v1 (and so on).
`ref (any v1, ...)(bool b1, ...)`	Returns true iff the argument v1 is a reference (and so on).
`def  (any v1, ...)(bool b1, ...)`	Returns true iff object v1 is defined (and so on). If the objects contain references, the objects themselves are analysed, i.e. the return values are always true. For analysing the referenced objects, the object function def() of type *any* is used.
`empty (any v1, ...)(bool b1, ...)`	Returns true iff object v1 is empty (and so on). For user-defined types, a comparison is made to a dummy object for which a standard allocation and the default initialisation (cf. section 8.5) have been applied. If the objects contain references, the objects themselves are analysed, i.e. the return values are always true. For analysing the referenced objects, the object function empty() of type *any* is used.
`count (any, ...)(int)`	Returns the number of arguments.
`min (any, ...)(any)`	Returns the minimum over all arguments. An operator “<” must be defined for the type of every argument.
`max (any, ...)(any)`	Returns the maximum over all arguments. An operator “<” must be defined for the type of every argument.
`sum (any, ...)(any)`	Returns the sum over all arguments.
`prod (any, ...)(any)`	Returns the product over all arguments.
`and (any, ...)(bool)`	Returns true iff all arguments are true.
`or (any, ...)(bool)`	Returns true iff any argument is true.
`get_cwd (str)`	Returns the current working directory (absolute path).
`get_env (str)(str)`	Returns the value of the specified environment variable.

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

`get_calc_time (real)` — Returns the calculation time in seconds since the start of the Nepal program or the last call of set_calc_time().

8.5 User-defined types

Syntax:

```
smalltype|bigtype <name>  (<base_1>, ... ,<base_n>)
                        {
                             [[<module>::]<type> <var_1>,...,<var_n>
                                  [(<const_1>,...,<const_n>)]; |
                              proc <proc_name> ...; |
                              func <func_name> ...; |
                              smalltype|bigtype <type_name> ...;
                             ]+
                        }
```

with

```
<name>,<base_1>,..<base_n>,<module>,<var_1>,..<var_n>,<type_name> =
                              ([a-z],[A-Z],_)([a-z],[A-Z],_,[0-9])+
<proc_name>,<func_name> =     ([a-z],[A-Z],_)([a-z],[A-Z],_,[0-9])+  or
                              "(<char>)+"
<const_1>,...,<const_n>
```

`<const_1>,...,<const_n>`: See specification of constants for types bool, char, int, real and str.

Condition: Name of built-in types cannot be used for a user-defined type.

Meaning: Collection of types, variables, procedures, and functions with optional, inherited user-defined base types. For restrictions with respect to inheritance, see section 8.1. The optional constants are used as a basic initialisation for any object of the specified type.

Type functions (built-in):

`bases (args)`	Returns the names of all base types including the current type.
`vars (str b)(args)`	Returns the names of all type variables of the base type b (can also be the current type). If the string b is empty, the names of all type variables over all base types including the current type are returned.

Type functions (optional, user-definable):

`"" (any,...)(any)`	Initialisation function to generate an anonymous object.

Object procedures (optional, user-definable):

`"" (any,...)`	Initialisation procedure executed with the statement of the data definition.
`in (str s)`	Procedure to read the data from string s, supporting the reading functions for strings, files and the console.

Object functions (built-in):

`get (str b,v)(any r)`	Returns the content of the object variable corresponding to the type variable v of base type b – either a value or a reference.
`"[]" (str b,v)(any :$ r)`	Returns an alias on the object variable corresponding to the type variable v of base type b.

Object functions (optional, user-definable):

`out (str s)`	Function to write the data to string s, supporting the writing functions for strings, files and the console.

Functional object operators (built-in):

Functional object operators (optional, user-definable):

`"<" (any a)(bool)`	Comparison operator. This supports the Nepal system functions *min()* and *max()* as well as the procedures *sort()* and *Sort()* for types list, args, set, hash and array.

Procedural object operators (optional, user-definable):

`"+=" (any a)`	Operator for addition; this defines the binary operator "+" implicitly.

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

`"-=" (any a)`	Operator for subtraction; this defines the binary operator "-" implicitly.
`"*=" (any a)`	Operator for multiplication; this defines the binary operator "*" implicitly.
`"/=" (any a)`	Operator for division; this defines the binary operator "/" implicitly.
`"%=" (any a)`	Operator for modulo; this defines the binary operator "%" implicitly.
`"^=" (any a)`	Operator for power; this defines the binary operator "^" implicitly.

8.5.1 Modular concept for user-defined types

A procedure or function of a user-defined type can also be specified outside of the type definition. This can be done either in the same program file or in another program file to be included. The syntax for "external" procedures and functions is as follows:

```
proc <type_name>:<proc_name> [(<input>)] { <block> }
func <type_name>:<func_name> [(<input>)](<output>) { <block> }
```

8.5.2 Application-specific extension of user-defined types

For a Nepal program is it possible to specify exactly one relevant application (a user-defined type <app>) by using the program option "-a <app>" for the interpreter. Within this type application-specific extensions for other user-defined types can be specified in the following two ways:

1. A block consisting of additional types, variables, procedures or functions is added to another type.

   Syntax:
   ```
   <type_name> += {
                 [[<module>::]<type> <var_1>,...,<var_n>
                        [ = <const_1>,...,<const_n> ]; |
                  proc <proc_name> ...; |
                  func <func_name> ...; |
                  smalltype|bigtype <type_name> = ...;
                 ]+
   }
   ```

   Meaning: At the beginning of the program execution, the block right of the operator "+=" is copied into the type left of the operator "+=". Also extensions defined for base types of the relevant application are handled in the same manner.

2. A type is added to another type.

   Syntax: `<type_name> += <type_name2>`
   Meaning: At the beginning of the program execution the type right of the operator "+=" is added to the list of base types of the type left of the operator "+=". The new base type is added at the beginning of the list in order to prioritize this type over the existing base types. Also extensions defined for base types of the relevant application are handled in the same manner.

9 Variables

For the definition of variables, see section 11.3.7.

9.1 User-defined variables

Syntax:	`([a-z],[A-Z],_)([a-z],[A-Z],_,[0-9])+`
Condition:	Name of built-in variables cannot be used for a user-defined variable.
Meaning:	Name of an accessible memory field.

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

9.2 Built-in variables

`this`	Provides access to the current object. Can only be used inside an object procedure or function.
`true`	Returns a constant value for the built-in type *bool.*
`false`	Returns a constant value for the built-in type *bool.*

10 Procedures and functions

10.1 User-defined procedures and functions

Syntax:

```
proc <proc_name> [ (<input>) ] { <block> }
func <func_name> [(<input>)] (<output>) { <block> }
with
<input>,<output> = [<module_1>::]<type_1> [:$] <var_11>,...,<var_1n>;...;
                   [<module_n>::]<type_n> [:$] <var_n1>,...,<var_nn>
<module_1>,...,<module_n>,<type_1>,...,<type_n>,<var_11>,...,<var_nn> =
                            ([a-z],[A-Z],_)([a-z],[A-Z],_,[0-9])+
<proc_name>,<func_name> = ([a-z],[A-Z],_)([a-z],[A-Z],_,[0-9])+ or
                            "(<char>)+"
```

Conditions: The input arguments may contain 0, 1 or 2 variables of the built-in type *args* (for the sake of uniqueness). For two variables of type *args*, only the two combinations "args a1, args :$ a2" and "args :$ a1, args a2" are allowed - with no extra argument in between. Here the distinction between arguments a1 and a2 is made according to the variability of the transferred data. All data embodied by variables (from right to left and left to right) are assigned to the arguments a2 and a1 in the first and second combination, respectively. The remaining data (left of the left-most variable data and right of the right-most variable data) are linked to the arguments a1 and a2, respectively. For the output arguments, there is no restriction with respect to the built-in type *args*.

Meaning: A procedure consists of an optional list of input arguments and a block to be executed. A function consists of an optional list of input arguments, a mandatory list of output arguments and a block to be executed. For the data transfer across the input arguments, three cases are distinguished:
1. Call-by-value: Here the input data are copied to the variables of the input arguments. This copy is read-only, i.e. the transferred data must not be changed inside the procedure / function or recursively called procedures / functions. Call-by-value is applied to all data of small types and *constant* data of big types. These data are returned by functions not using the operator ":$" for the output arguments (see below).
2. Call-by-reference: Here a reference on the input data is established on the variables of the input arguments. This reference is read-only, i.e. the transferred data must not be changed inside the procedure / function or recursively called procedures / functions. Call-by-reference is applied to all data of *variable* big types. These data either reside on variables or are returned by functions using the operator ":$" for the output arguments (see below).
3. Call-by-alias: Here the input arguments are considered as synonymous variables on the transferred data. Therefore, a read-and-write access is supported inside the procedure / function. A writing access always influences the original data. No restriction exists with respect to small or big types. Call-by-alias is specified by using the operator ":$".
For the output arguments, empty data are initially allocated on the program stack. Then the body (block) of the procedure / function is executed which usually modifies the output arguments. The question whether the resulting output arguments can be changed within the calling statement depends on the usage of the operator ":$". Without using this operator, the returned data are constant and must not be changed. Otherwise a modification is possible.

10.2 Built-in procedures and functions

The built-in procedures and functions in the global scope are provided by the built-in type *sys*. Section 8.4.18 contains a complete list of these elements.

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

11 Statements

11.1 Definition

A statement is an elementary program construct which can be executed. Possible occurrences of statements are:

- as part of a Nepal program (cf. section 3)
- as part of a block (cf. section 12)

11.2 Inclusion of program files

11.2.1 Standard inclusion

A Nepal program can be split into several files. The following statement is used to include a program file:

```
need "<code file to be included>"
```

When this statement is reached during program execution, all statements contained in this code file are executed sequentially according to their location within the file. Moreover all types, variables, procedures and functions at the outermost scope of the included file are accessible from the current program file.

Multiple files can be included by using a comma-separated list:

```
need "<code file 1>", "<code file 2>", ...
```

Multiple inclusion of a certain file is eliminated automatically, i.e. only the first occurrence is relevant for execution. For recursive inclusions the relevant occurrence is determined by a breath first – depth second search. Circular inclusions are not allowed and will raise an error. Within a program file the need-statement must be located at the outermost scope.

11.2.2 Module inclusion

To avoid name clashes, files can be included using a module specifier:

```
need <module>("<code file to be included>")
with
<module> = ([a-z],[A-Z],_)([a-z],[A-Z],_,[0-9])+
```

This Nepal statement attaches the specified module together with the module access operator "::" to every variable, type, procedure and function at the outermost scope of the included file. Multiple inclusions of a certain file with different modules are possible. This means that the included variables, types, procedures and functions are accessible using any of these modules. However, only the "first occurrence" (due to the meaning described in section 11.2.1) is relevant for program execution.

Multiple files can be included with a single module by using a comma-separated list:

```
need <module>("<code file 1>", "<code file 2>", ...)
```

In the following example, a program file "b.npl" includes a program file "a.npl" with the module "mybasiclib" and calls a procedure p() from the included program file.

a.npl:

```
proc p { outl("execute p") }
```

b.npl:

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

```
need mybasiclib(”a.npl”);

mybasiclib::p(); # call p from program file a.npl
```

A recursive inclusion involving several modules leads to a concatenation of modules. The following example demonstrates a recursive inclusion of two program files.

a.npl:

```
proc p { outl(“execute p”) }
```

b.npl:

```
need mybasiclib(”a.npl”)
```

c.npl:

```
need mylib(”b.npl”);

mylib::mybasiclib::p(); # call p from program file a.npl
```

11.2.3 Application-specific inclusion

For a Nepal program is it possible to specify exactly one relevant application (a user-defined type <app>) by using the program option “-a <app>” for the interpreter. For this type an application-specific inclusion of program files can be defined as follows:

```
need <app>:"<code file to be included>"
```

This means that the inclusion of the program file is only performed if the specified type matches either with the relevant application or a base type of the type corresponding to the relevant application. To determine the necessary inclusions, the Nepal interpreter performs several passes: Firstly all unconditional inclusions are executed. Then the base types of the type corresponding to the relevant application are calculated. Finally all conditional inclusions with matching types are executed.

Multiple files can be included by using a comma-separated list:

```
need <app>:"<code file 1>", <app>:“<code file 2>”, ...
```

A mixture of standard, module and application-specific inclusions is possible for every need-statement, for example:

```
need “<code file 1>”, <module1>(<app>:"<code file 2>")
```

11.3 Procedural operators

The following table shows the priority of the procedural and functional operators. The higher the priority, the stronger is the binding between the corresponding operands. For operators with the same priority the grouping of operands is from left to right. To overwrite the built-in priority of operators, the grouping of operands can be forced by use of round brackets. For example, the operands “1” and “2” are linked together within the expression “(1+2)*3” although the operator “*” has higher priority than the operator “+”.

Priority	Operators	Type of operator
1	`)(   ){   }(   }{`	procedural
2	`;`	procedural
3	`:$ `` ` `` (e.g. the operator in between	procedural

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

<table>
<tr><td></td><td>“int n”)</td><td></td></tr>
<tr><td>4</td><td>= @ $? ~ += -= *=
/= %= ^=</td><td>procedural</td></tr>
<tr><td>5</td><td>,</td><td>procedural</td></tr>
<tr><td>6</td><td>||</td><td>functional</td></tr>
<tr><td>7</td><td>&&</td><td>functional</td></tr>
<tr><td>8</td><td>< > == != <= >=
@@ !@ ?? !?</td><td>functional</td></tr>
<tr><td>9</td><td>|</td><td>procedural</td></tr>
<tr><td>10</td><td>&</td><td>procedural</td></tr>
<tr><td>11</td><td>.. .: :. **</td><td>functional</td></tr>
<tr><td>12</td><td>+(binary) -(binary)</td><td>functional</td></tr>
<tr><td>13</td><td>* / %</td><td>functional</td></tr>
<tr><td>14</td><td>^</td><td>functional</td></tr>
<tr><td>15</td><td>! +(unary) -(unary)</td><td>functional</td></tr>
<tr><td>16</td><td>.</td><td>procedural or functional</td></tr>
<tr><td>17</td><td>:</td><td>procedural or functional</td></tr>
<tr><td>18</td><td>::</td><td>procedural or functional</td></tr>
</table>

The following sections describe the procedural operators. The functional operators are described in section 13.5.

11.3.1 Assignments

In the following assignments the data object on the left-hand side is either a variable, an object variable, the built-in variable *this* or an alias function. On the right-hand side there is always an expression.

11.3.1.1 Copy

Syntax: <obj> = <expr>
Meaning: After clearing the data object, the value of the expression is (deeply) copied to the object. Normally the types must match exactly. The exception is that the (initial) object is of type *any*. In this case, the assignment is always possible. If the object contains a reference, the object itself is overwritten (not the object where the reference is pointing to). If the object contains an alias, the object is overwritten where the alias is pointing to. If the expression contains a reference or alias, the value of the object is copied where the chain of aliases/references is pointing to.

11.3.1.2 Reference

Syntax: <obj> @ <expr>
Meaning: After clearing the data object, a reference on the expression is created at the object. Normally the types must match exactly. The exception is that the (initial) object is of type *any*. In this case, the assignment is always possible. If the object contains a reference, the object itself is overwritten (not the object where the reference is pointing to). If the object contains an alias, the object is overwritten where the alias is pointing to. If the expression contains a reference or alias, a reference on the object is established where the chain of aliases/references is pointing to. Concatenated references are not allowed. References must not point to objects containing aliases. But aliases may point to objects containing references.

11.3.1.3 Alias

Syntax: <obj> $ <expr>
Meaning: After clearing the data object, an alias on the expression is created at the object. Normally the types must match exactly. The exception is that the (initial) object is of type *any*. In this case, the assignment is always possible. If the object contains a reference, the object itself is overwritten (not the object where the reference is pointing to). If the object contains an alias, the object is overwritten where the alias is pointing to. If the expression contains a reference, an alias on the object itself is established (not the object where the reference is pointing to). If the expression contains an alias, an alias on the object is established where the chain of aliases is pointing to. The only possible concatenation of aliases occurs when the (implicitly generated) alias contained in the input argument of a procedure / function using the operator “:$” is pointing to an object containing a “normal” alias. Aliases may point to objects containing references. But references must not point to objects containing aliases.

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

11.3.1.4 Copy or Reference

Syntax: <obj> ? <expr>
Meaning: After clearing the data object, either a copy or a reference on the expression is created on the object depending on the question if the expression contains a value or a reference. Normally the types must match exactly. The exception is that the (initial) object is of type *any*. In this case, the assignment is always possible. If the object contains a reference, the object itself is overwritten (not the object where the reference is pointing to). If the object contains an alias, the object is overwritten where the alias is pointing to. If the expression contains a reference or alias, a reference on the object is established where the chain of aliases/references is pointing to.

11.3.1.5 Move

Syntax: <obj> ~ <expr>
Meaning: After clearing the data object, the value of the expression is moved to the object. The expression is then undefined. Normally the types must match exactly. The exception is that the (initial) object is of type *any*. In this case, the assignment is always possible. If the object contains a reference, the object itself is overwritten (not the object where the reference is pointing to). If the object contains an alias, the object is overwritten where the alias is pointing to. If the expression is a reference or alias, the object is moved where the chain of aliases/references is pointing to.

11.3.1.6 Multiple Assignment

Instead of a single assignment between exactly two variables, also multiple assignments in the following form can be used:

```
<obj_1>,...,<obj_n> <assignment_operator> <expr_1>,...,<expr_m>
```

Here MIN(m,n) evaluations and assignments are performed sequentially, i.e. firstly the evaluation of <expr_1> and the assignment from <expr_1> to <obj_1>, then the evaluation of <expr_2> and the assignment from <expr_2> to <obj_2>, and so on.

11.3.2 Mathematical operators (for types int, real, str, set and list)

“+=”	(any a1,..,an; any b1,..,bm)	Addition (for types list and str: concatenation; for type set: union), optionally with multiple arguments.
“-=”	(any a1,..,an; any b1,..,bm)	Subtraction (for type set: delta set), optionally with multiple arguments.
“*=”	(any a1,..,an; any b1,..,bm)	Multiplication (for type set: intersection), optionally with multiple arguments.
“/=”	(any a1,..,an; any b1,..,bm)	Division, optionally with multiple arguments.
“%=”	(any a1,..,an; any b1,..,bm)	Modulo, optionally with multiple arguments.
“^=”	(any a1,..,an; any b1,..,bm)	Power, optionally with multiple arguments.

11.3.3 Structural operators

“)(“ (any a,b)	Pair of input or output arguments (of any type except *code*)
“){“ (any a; code b)	Pair of input or output arguments
“}(“ (code a; any b)	Pair of input or output arguments
“}{“ (code a,b)	Pair of input or output arguments
“;”	Sequence of statements or input/output arguments
“,”	Sequence of variables or expressions
“` `”	Definition of variable, e.g. operator in between “int n”; see also section 11.3.7.
“:$”	Input arguments using call-by-alias or output arguments of an alias function
“|”	Sequence of variables or expressions acting as logical alternatives on left/right-hand side of comparison operator. For example, “a	b == c” stands for “a == c || b == c”.

"&" Sequence of variables or expressions acting as logical conditions on left/right-hand side of comparison operator. For example, "a & b == c" stands for "a == c && b == c".

11.3.4 Data access operator

"." (any a1, …, am; proc b1, …, bn) Executes object procedures in the order a1.b1, a1.b2, …, a1.bn, a2.b1, …, a2.bn, …, am.b1, …, am.bn. For more than one element round brackets have to be used, e.g. "(a1,a2).(b1,b2)".

11.3.5 Type access operator

":" (type a; proc b) Executes procedure b of type a

11.3.6 Module access operator

"::" (module a; proc b) Executes procedure b of module a

11.3.7 Definitions of variables and initialisations

Syntax (First form): [<module>::]<type> <var1>, <var2>, ... ;
Meaning: The variables var1, var2 etc. are defined for the given type in the current scope.
Syntax (Second form with assignment):

```
[<module>::]<type> <var1>, <var2>, ... <assignment_operator> <expr1>,
<expr2>, ...;
```

Meaning: The variables var1, var2 etc. are defined for the given type in the current scope. The (multiple) assignment is then performed while evaluating the expressions on the right-hand side (for the order of assignments and evaluations, see section 11.3.1.6).
Syntax (Third form with initialisation procedure):

```
[<module>::]<type> <var1>(any a11,...), <var2>(any a21,...), ...;
```

Meaning: The variables var1, var2 etc. are defined for the given type in the current scope. The object procedures ""(any a,...) are executed in the order of the given sequence of variables.
Syntax (Forth form with object procedure):

```
[<module>::]<type> <var1>.p1(any a11,...), <var2>.p2(any a21,...), ...;
```

Meaning: The variables var1, var2 etc. are defined for the given type in the current scope. The object procedures p1(any a11,...), p2(any a21,...) etc. are executed in the order of the given sequence of variables.

The first, third and forth form can be mixed arbitrarily.

11.4 Control structures

The following statements are in fact type procedures of the built-in type *sys* (cf. section *8.4.18).* This has the advantage that e.g. the built-in for-loop can be addressed within a user-defined for-loop of a user-defined type. The example below demonstrates this feature.

```
bigtype test {

      proc for (code C) { sys:for(C) }
}
```

11.4.1 Branches

- if-else

Syntax: if(<bool_expr>) { <block_1> } [{ <block_2> }]
Meaning: The boolean expression is evaluated. If the value is true, block 1 is executed, otherwise block 2.

- switch with variable

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

```
Syntax: switch(<obj>) { case (<expr_11>,...,<expr_1m>) { <block_1> }; ...
                        case (<expr_n1>,...,<expr_nm>) { <block_n> };
                      [ case { <block_0> } ] }
```

Meaning: The expressions are evaluated sequentially. If a value matches with the value of the data object, the corresponding block is executed, and the switch-statement is quitted. The optional case-block without an expression is executed if no match with any expression exists.

- switch without variable

```
Syntax: switch { case (<bool_expr_11>,...,<bool_expr_1m>) { <block_1> }; ...
                 case (<bool_expr_n1>,...,<bool_expr_nm>) { <block_n> };
                 [ case { <block_0> } ] }
```

Meaning: The boolean expressions are evaluated sequentially. If a value is true, the corresponding block is executed, and the switch-statement is quitted. The optional case-block without an expression is executed, if no expression has the value true.

11.4.2 Loops

- for

Syntax (First form): for {<block_0>}{<bool_expr>}{<block_1>} { <block_2> }
Meaning: Firstly block 0 is executed. As long as the boolean expression has the value true, block 2 is executed. If block 2 is executed completely (i.e. no break-, exit- or return- statement is contained), block 1 is executed before the repeated evaluation of the boolean expression.

Syntax (Second form): for (any :$ x, any y1, ...) { <block> }
Meaning: The arguments y1 (and so on) are evaluated and assigned to argument x sequentially. After every assignment the block is executed.

Syntax (Third form): for { <block> }
Meaning: Endless loop.

- while

Syntax: while {<bool_expr>} { <block> }
Meaning: As long as the boolean expression has the value true, the block is executed.

- continue

Syntax: continue(int n)
Meaning: (n-1) nested loops are quitted, and for the n-th loop the program jumps to the end of the block.

- break

Syntax: break(int n)
Meaning: n nested loops are quitted.

11.4.3 Termination of procedures or functions

Syntax: return(int n)
Meaning: n nested procedures or functions are quitted.

11.4.4 Termination of Nepal programs

Syntax: exit(int n)
Meaning: The current program is quitted, and the value "n" is transferred to the operating system (calling the system function "exit(int)"). This procedure is always executed, even if an exception has been thrown (cf. section 11.4.5).

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

11.4.5 Exceptions

Syntax:

```
throw (<expr>)
catch (<obj>) { <block> }
```

Meaning: A user-defined exception is raised with the *throw*-statement. All subsequent statements are skipped until a suitable catch-statement or a procedure *exit()* is reached. When raising the exception inside a loop, the loop is quitted at the end of the block. If the type of the catch-object is identical to the type of the thrown value, the catch-block is executed, and no further statements are skipped. A catch-object of (initial) type *any* catches thrown values of any type. The called exit-procedure can be either the built-in procedure (cf. section 11.4.4) or a user-defined procedure (with arbitrary arguments). If a Nepal-internal error occurs, e.g. an infeasible index is used for objects of type *array*, then an exception of type *error* is thrown.

11.5 Procedure calls

11.5.1 General procedures

Syntax: `[<module>::]p(<expr1>,...,<exprn>)`

Meaning: After evaluating the expressions expr1 to exprn, these values are passed to the procedure as input arguments, and the procedure is executed. Optionally, a module can be specified to address procedures of the corresponding program file.

11.5.2 Object procedures

Syntax (First form):

`o.p(<expr1>,...,<exprn>)` or `o.<base type>:p(<expr1>,...,<exprn>)`

Meaning: After evaluating the expressions expr1 to exprn, these values are passed as input arguments to the procedure of the data object o, and then the procedure is executed. The above-mentioned syntax is used outside of object procedures. When the built-in variable `this` (cf. section 9.2) is used for data object o, this syntax can also be used inside of object procedures. Optionally, a base type can be specified to address procedures of this type.

Syntax (Second form):

`p(<expr1>,...,<exprn>)` or `<base type>:p(<expr1>,...,<exprn>)`

Meaning: After evaluating the expressions expr1 to exprn, these values are passed as input arguments to the procedure of the current data object, and then the procedure is executed. The above-mentioned syntax is used inside of object procedures. Optionally, a base type can be specified to address procedures of this type.

11.5.3 Type procedures

Syntax: `[<module>::]<type>:p(<expr1>,...,<exprn>)`

Meaning: After evaluating the expressions expr1 to exprn, these values are passed to the type procedure as input arguments, and then the procedure is executed. The above-mentioned syntax is used inside and outside of object procedures. Optionally, a module can be specified to address procedures of the corresponding program file.

12 Blocks

12.1 Definition

Syntax: `{ [ <statement> ] ( ; <statement> )+ }`

Meaning: The statements are executed sequentially. Executing an empty block has no effect. Evaluating an empty block raises an error. Every block builds an own scope (cf. Section 6) and is a constant object of the built-in type *code* (cf. section 8.4.16). Possible occurrences of blocks are:

- As input arguments of procedures or functions, e.g. "while { a < b } { c += d }"
- As an expression, e.g. the right-hand side of code "c = { 2 < 3 }"
- As body of a type definition
- A body of a procedure or function definition
- As body of a type extension, e.g. "<type_name> += { <type extension> }" (cf. section 8.5.2).

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

12.2 Syntactical sugar

If a block consists of a single statement and follows a closing (round) bracket, it can be written with a final semicolon instead of brackets. This helps to spare curly brackets. For example, the branch statement

```
if(a < b) { c = d }
```

can also be written as

```
if(a < b) c = d;
```

In this way not only if-statements but also case-, for- and catch-statements may be written in a simplified manner.

This simplification can be applied even multiple times for nested blocks. For example, the following double loop

```
for(i,1..10)
{
  for(j,1..10)
  {
      n += i*j
  }
}
```

can also be written as

```
for(i,1..10)
  for(j,1..10)
      n += i*j;
```

To spare semicolons the interpreter appends a semicolon implicitly after the end of each block. For example, the following code

```
bigtype test { int n};
test t
```

can be also written as

```
bigtype test { int n}
test t
```

13 Expressions

13.1 Definition

Expressions are constructs of a Nepal program which can be evaluated. The following occurrences of expressions are possible:

- In combination with operators to build another expression
- The right-hand side of an assignment
- As data transferred to input arguments for procedures and functions

13.2 Access of variables

13.2.1 General variables

Syntax: `[<module>::]<var>`
with
`<module>,<var> = ([a-z],[A-Z],_)([a-z],[A-Z],_,[0-9])+`

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

Meaning: The result of the evaluation is a copy of the value of the addressed variable. If the variable contains a reference, the value of that variable is copied where the reference is pointing to. Optionally, a module can be specified to address variables of the corresponding program file.

13.2.2 Object variables

Syntax (First form): `d.<var>` or `o.<base type>:<var>`
with
`<var>,<base_type> = ([a-z],[A-Z],_)([a-z],[A-Z],_,[0-9])+`

Meaning: The result of the evaluation is a copy of the value of the addressed variable as part of the given data object o. If the variable contains a reference, the value of that variable is copied where the reference is pointing to. The above-mentioned syntax is used outside of object procedures or functions. When the built-in variable `this` (cf. section 9.2) is used for data object o, this syntax can also be used inside of object procedures or functions. Optionally, a base type can be specified to address variables of this type.

Syntax (Second form): `<var>` or `<base type>:<var>`
with
`<var>,<base_type> = ([a-z],[A-Z],_)([a-z],[A-Z],_,[0-9])+`

Meaning: The result of the evaluation is a copy of the value of the addressed variable as part of the current data object. If the variable contains a reference, the value of that variable is copied where the reference is pointing to. The above-mentioned syntax is used inside of object procedures or functions. Optionally, a base type can be specified to address variables of this type.

13.3 Function calls

13.3.1 General functions

Syntax: `[<module>::]f(<expr1>,...,<exprn>)`

Meaning: After evaluating the expressions expr1 to exprn, these values are passed to the function as input arguments, and the function is executed. The evaluated output arguments of the function are the result of the expression. Optionally, a module can be specified to address functions of the corresponding program file.

13.3.2 Object functions

Syntax (First form):
`o.f(<expr1>,...,<exprn>)` or `o.<base_type>:f(<expr1>,...,<exprn>)`

Meaning: After evaluating the expressions expr1 to exprn, these values are passed as input arguments to the function of the data object o, and then the function is executed. The evaluated output arguments of the function are the result of the expression. The above-mentioned syntax is used outside of object functions. When the built-in variable `this` (cf. section 9.2) is used for data object o, this syntax can also be used inside of object functions. Optionally, a base type can be specified to address functions of this type.

Syntax (Second form):
`f(<expr1>,...,<exprn>)` or `<base type>:f(<expr1>,...,<exprn>)`

Meaning: After evaluating the expressions expr1 to exprn, these values are passed as input arguments to the function of the current data object, and then the function is executed. The evaluated output arguments of the function are the result of the expression. The above-mentioned syntax is used inside of object functions. Optionally, a base type can be specified to address functions of this type.

13.3.3 Type functions

Syntax: `[<module>::]<type>:f(<expr1>,...,<exprn>)`

Meaning: After evaluating the expressions expr1 to exprn, these values are passed as input arguments to the type function, and then the function is executed. The evaluated output arguments of the function are the result of the expression. The above-mentioned syntax is used inside and outside of object functions. Optionally, a module can be specified to address functions of the corresponding program file.

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

13.4 Conditional expressions

The following expression is in fact a type function of the built-in type *sys* (cf. section *8.4.18).*

- if

Syntax: if (<bool_expr>) {<expr_1_1>,...,<expr_1_n>} {<expr_2_1>,...,<expr_2_n>}
Meaning: The boolean expression is evaluated. If the value is true, the list of arguments expr_1_i, i = 1...n, is evaluated, otherwise the list of arguments expr_2_i, i = 1...n.

13.5 Functional operators

The following sections describe the functional operators. The procedural operators as well as the priorities of all operators are described in section 11.3.

13.5.1 Boolean operators

"!" (bool a)(bool) — Logical negation
"&&" (bool a,b)(bool) — Logical and
"||" (bool a,b)(bool) — Logical or

13.5.2 Operators for comparison

"==" (any a,b)(bool) — Equality with respect to value
"!=" (any a,b)(bool) — Inequality with respect to value
"@@" (any a,b)(bool) — Equality with respect to reference
"!@" (any a,b)(bool) — Inequality with respect to reference
"??" (any a,b)(bool) — Equality with respect to value or reference
"!?" (any a,b)(bool) — Inequality with respect to value or reference
"<" (any a,b)(bool) — Smaller (for type set: subset)
">" (any a,b)(bool) — Greater (for type set: subset)
"<=" (any a,b)(bool) — Smaller or equal (for type set: subset or equality)
">=" (any a,b)(bool) — Greater or equal (for type set: subset or equality)

13.5.3 Mathematical operators (for types int, real, str, set, and list)

"+" (any a,b)(any) — Addition (for types list, str: concatenation; for type set: union)
"-" (any a,b)(any) — Subtraction (for type set: delta set)
"*" (any a,b)(any) — Multiplication (for type set: intersection)
"/" (any a,b)(any) — Division
"%" (any a,b)(any) — Modulo
"^" (any a,b)(any) — Power

13.5.4 Range operators

".." (int|char a,b)(list) — Range of integer or character values (ascending or descending)
".:" (int|char a,b)(list) — Range of integer or character values (only ascending)
":." (int|char a,b)(list) — Range of integer or character values (only descending)
"**" (any a; int n)(any b1, ..., bn) — Repetition of values (value a is repeated n-times).

13.5.5 Data access operator

"." (any a1,...,am; func|any b1,...,bn) — Executes object functions or evaluates object variables in the order a1.b1, a1.b2, …, a1.bn, a2.b1, …, a2.bn, …, am.b1, …, am.bn..
For more than one element, round brackets have to be used, e.g. "(a1,a2).(b1,b2)".

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.

13.5.6 Type access operator

`":" (type a; func b)`	Executes function b of type a

13.5.7 Module access operator

`"::" (module a; func b)`	Executes function b of module a

13.6 Constants

Constants exist for the built-in types *bool*, *int*, *real*, *char*, *str* and *code* (see section 8.4).

14 Glossary

Alias	A pointer to a data object. Assignments to aliases apply to the objects where they are pointing to (in contrast to references). Aliases are generated with the assignment operator "$" or when data objects are transferred to input arguments of procedures or functions using call-by-alias convention. Moreover aliases are returned from alias functions.
Alias function	Function returning an alias on a data object (by using the operator ":$" for the output arguments).
Argument	A parameter of a procedure, function or program.
Base type	A type whose elements are inherited completely by types derived from this type. All elements can be accessed and used by the derived types.
Basic initialisation	An initialisation applied to a data object just after its creation.
Big type	Constant / variable data of big types as input arguments of procedures or functions are copied by value / reference. When using these data as keys of hash arrays, they are compared with respect to their value / (memory) address.
Block	A list of statements to be executed sequentially.
Call-by-alias	Interpret the input arguments of procedures or functions as synonymous variables for the transferred data (i.e. supporting read-and-write access).
Call-by-reference	Apply a constant copy to the address of the transferred data for input arguments of procedures or functions.
Call-by-value	Apply a constant copy to the values of the transferred data for input arguments of procedures or functions.
Comment	Text within a program for the purpose of documentation. Comments are ignored by the interpreter.
Constant	A memory field whose value cannot be changed during program execution.
Definition	A specification of a type, variable, procedure or function.
Expression	A program construct which can be evaluated.
Function	A program construct consisting of an executable block, optional input arguments and mandatory output arguments.
Functional operator	Function with one or two input arguments, represented by a special symbol, e.g. "==" for comparison of two objects.
Inclusion	Embedding program files into another program file.
Interpreter	A software for executing a program directly from the source code, without compilation into machine-readable code.
Module	A specifier for inclusion of program files.
Object	A concrete instance (data collection) of a certain type.
Object function	A function of a type which can be called only by a certain object. It calls other functions or procedures of the object or does access variables of the object. Otherwise it would be a type function of the corresponding type.
Object procedure	A procedure of a type which can be called only by a certain object. It calls other functions or procedures of the object or does access variables of the object. Otherwise it would be a type procedure of the corresponding type.

Object variable	A variable of a type which can be accessed only by a certain object.
Operand	Argument of an operator.
Operator	Function or procedure with one or two input arguments, represented by a special symbol, e.g. “+” for addition of two objects.
Polymorphism	The possibility to keep data objects of different types within another data object, e.g. within a set or list.
Procedural object operator	Object procedure with zero or one input argument which can be applied only by a certain object and is represented by a special symbol, e.g. “+=” for addition. The first argument of the operator is the object itself.
Procedure	A program construct consisting of an executable block and optional input arguments.
Program	A list of definitions, statements, and comments.
Program argument	Parameter of a program.
Program option	A setting to control the behaviour of a program, e.g. the specification of directories for the inclusion of program files.
Reference	A pointer to a data object. Assignments to references do not apply to the objects where they are pointing to, but to the objects containing the references (in contrast to aliases). References are generated with the assignment operator “@” or when data objects are transferred to input arguments of procedures or functions using call-by-reference convention.
Scope	A connected part of a program with definitions of types, variables, procedures or functions. For a certain scope, the defined names and signatures have to be unique per type of construct.
Signature	A string built from the name and the input arguments of a procedure or function.
Small type	Data of small types as input arguments are copied by value. When using these data as keys of hash arrays, they are compared with respect to their value.
Statement	An elementary program construct which can be executed.
Symbol table	A table to access all types, variables, procedures and functions of a certain scope.
Syntax	The specification of a feasible program.
Type	A collection of variables, procedures, functions, nested types and base types (= the elements of the type).
Type function	A function of a type which can be called by objects of this type as well as without a concrete object. The function does not call object procedures or functions and does not access object variables.
Type procedure	A procedure of a type which can be applied by objects of this type as well as without a concrete object. The procedure does not use object functions or procedures and does not access object variables.
Type variable	A variable of a type.
Value	The result of the evaluation of a data object.
Variable	A memory field accessible for reading and writing via a corresponding name. The size of the memory field is determined by the corresponding type.

 Copyright © 2010 Karl-Heinz Erhard. All rights reserved.