Nepal 1.1

Tutorial

Version 2

Table of contents

Version history

1	Initial version for Nepal 1.0.
2	Initial version for Nepal 1.1. Chapter 13 for new built-in type *graph*.

1 Introduction

Nepal is a new object-oriented programming language. The following examples have been tested successfully on Windows XP / Vista, Linux and Mac OS using the interpreter *nepal.exe*. Section 1 of the reference manual describes how to install the interpreter. The program is started from the Windows console using the command

```
nepal [<program_options>] <code_file> [<program_arguments>].
```

After starting the interpreter the Nepal statements contained in the code file are executed sequentially. In the following tutorial, also a few program options for the Nepal interpreter will be demonstrated. For a complete list of the available program options, see section 2 of the reference manual. Moreover the interpreter may be launched with optional program arguments. Sections 4 and 5 will give examples for how to use these arguments.

2 Data output

The well-known program “hello world” shall demonstrate the output of data onto the console. For this purpose please edit a code file “hello.npl” with the following content:

```
# the famous hello world program

outl("hello world !")
```

After entering the command "nepal hello.npl" on the console you should get the output "hello world !". The global procedure *outl()* prints all arguments sequentially to the console and finishes the output with an end-of-line (character ‘\n’). An output without the closing end-of-line would be obtained with the global procedure *out()*. The first line of the code file is a comment since it starts with the character ‘#’ followed by the character blank or tab. General (multi-line) comments start and end with the string “#(“ and “)#”, respectively. They can be nested with unlimited depth. If the next character after ‘#’ is neither a blank, tab, or ‘(‘, a so-called *word comment* is starting. This kind of comment ends at the next character blank, tab or end-of-line.

If the data output shall be redirected onto a file, the program option “-o” can be used as follows:

```
nepal -o hello.out hello.npl
```

This command writes the text "hello world !" to file "hello.out". Another possibility for writing data into a file is using the built-in type *file*. The following Nepal program “hello2.npl” yields the same output as above – without using the program option “-o”.

```
file f("hello2.out");
f.outl("hello world !");
```

3 Data input

The global procedure *inl()* serves for the line-oriented input of data from the console. Please test the following program "how_are_you.npl". Here the built-in type *str* for representing a sequence of characters is used to store the input string.

```
out("What's your name ? ");
str s;
inl(s);
outl("Hello '",s,"' ! How are you ?");
```

If the data input shall be redirected towards a file, the program option “-i” can be used as follows:

```
nepal -i hello.out how_are_you.npl
```

This command reads the text line "hello world !" from the file "hello.out", and the following output appears on the console:

```
What's your name ? Hello 'hello world !' ! How are you ?.
```


Another possibility for reading data from a file is using the built-in type *file*. The following Nepal program "how_are_you2.npl" yields the same output as above – without using the program option "-i".

```
out("What's your name ? ");
str s;
file f("hello2.out");
f.inl(s);
outl("Hello '",s,"' ! How are you ?");
```

4 Procedures

The following Nepal program "hello3.npl" introduces a user-defined procedure hello() in order to output a name represented by a variable of the built-in type *str*.

```
proc hello (str name) {
  outl("hello ", name, " !")
}

hello(argv(1))
```

The command "nepal hello3.npl world" yields the result

```
hello world !
```

The variable "name" is the (single) argument of the procedure.

The built-in function *argv (int n)(str s)* returns the n-th program argument of the Nepal interpreter as a string. Here the input argument n is greater or equal than 0. The 0-th argument corresponds to the name of the code file. The variable n is of the built-in type *int* which represents integer values of arbitrary length.

Also procedures without arguments are possible. This is demonstrated by the following example "hello4.npl".

```
proc hello_world {
  outl("hello world !")
}

hello_world()
```

After entering the command "nepal hello4.npl" on the console, you should get the output "hello world !" as before.

5 Functions

The following program "fac.npl" defines a recursive function to calculate the factorial of a natural number.

```
func fac (int n)(int r) {
  r = if(n == 0) { 1 } { n*fac(n-1) }
}

outl(fac(int:(argv(1))))
```

The command "nepal fac.npl 50" yields the result

```
30414093201713378043612608166064768844377641568960512000000000000
```

The variable n is the (single) argument or the function. The variable r stores the return value.

The conditional expression *if(a){b}{c}* firstly evaluates the boolean argument a. If the value of a is true, the if-expression yields the value of expression b, otherwise the value of expression c.

The built-in type function *int:"" (str s)(int n*) transforms string s into an integer value. Note that "int:()" is a short-cut for "int:""()".

Special characters can also be used for names of user-defined procedures or functions. In this case the name is specified with quotation marks. The following example "fac2.npl" rewrites the program "fac.npl" using the common symbol "!" for factorials.

```
func "!" (int n)(int r) {
  r = if(n == 0) { 1 } { n*"!"(n-1) }
}

outl("!"(int:(argv(1))))
```

Also functions without input arguments are possible. This is demonstrated by the following example "fac3.npl".

```
func fac3 (int r) {
  r = 3*2*1
}

outl(fac3())
```

After entering the command "nepal fac3.npl" on the console you should get the output "6".

The following program "div.npl" contains a function div() calculating the integer quotient q with remainder r for a dividend a and divisor b. Here a multiple assignment "a_1,a_2,..,a_m = b_1,b_2,..,b_n" is used where the assignment "a_1 = b_1" is executed firstly, then the assignment "a_2 = b_2", and so on. The number of executed assignments is MIN(m,n). The built-in procedure *set_out_spc (str s)* sets the separator for data outputs with the procedures *out()* and *outl()* (the default separator is ""). Therefore, the program prints the result "5,2" to the console.

```
func div (int a,b)(int q,r) {
  q, r = a/b, a-q*b
}

set_out_spc(",");
outl(div(17,3))
```

6 Control structures

The programming language Nepal supports the following control structures.

- The simple branch

```
if(a){b}{c}
```

If the boolean expression a is true, then code b is executed, otherwise code c. Block c is optional.

- The multiple branch with variable

```
switch(a) { case(b) { c } case(d) { e } ... case { z } }
```

The expression a is evaluated and compared sequentially with the values b, d, and so on. If a equals to b, then code c is executed. If a equals to d, then code e is executed, and so on. The optional case-block without an expression serves as default, i.e. code z is executed if no matching with any expression exists. As argument of the case-statement also a list of values can be used, e.g. "case(b_1,b_2,b_3) { c }". Here code c is executed if expression a equals to any of the three values.

- The multiple branch without a variable

```
switch { case(b) { c } case(d) { e } ... case { z } }
```

Firstly, the boolean expression b is evaluated. If the value is true, code c is executed. Otherwise expression d is evaluated, and so on. Code z is finally executed if no expression is true. As argument of the case-statement also a list of expressions can be used, e.g. “case(b_1,b_2,b_3) { c }”. Here code c is executed if any of the three values is true.

- The (endless) loop

```
for{a}
```

Here code a is executed forever.

- The loop

```
for(i,i_1,i_2,...,i_n){a}
```

Here the expressions from i_1 to i_n are evaluated and assigned to variable i sequentially. After each assignment code a is executed.

- The loop

```
for{a}{b}{c}{d}
```

Firstly, code a is executed. As long as the boolean expression b has the value true, code d is executed. If block d is executed completely (i.e. no break- or return- statement is contained), code c is executed before the repeated evaluation of expression b.

- The loop

```
while{a}{b}
```

As long as the boolean expression a has the value true, code b is executed.

For all loops the program constructs *continue* and *break* exist. The procedure *break (int n)* quits exactly n (nested) loops. The procedure *continue (int n)* quits (n-1) nested loops, and causes a program jump to the end of the block for the n-th loop.

A program "cosec.npl" is listed below which demonstrates the usage of the major control structures. The program serves for the **co**nversion of **sec**onds into the format “<hours>:<minutes>:<seconds>” or vice verca. The program reads input lines sequentially from the console until the user enters the string “q” or “quit”. Each line is split into separate strings using the separator “ “. Then each string is transformed in a suitable manner – depending on the question if the string format is “<seconds>”, “<hours>:<minutes>” or “<hours>:<minutes>:<seconds>”.

In the example two built-in types are used the first time: The type *bool* represents a logical value - either *true* or *false*. The type *char* represents a single character. For the known type *str*, several object functions are used: The function *args (char,...)* splits a string into individual characters. The function *find_str (str s)(args a)* searches sub-string s and returns the first position (as either one output argument for success or zero arguments for failure). The built-in type *args* represents a variable list of input or output arguments for procedures or functions. A more detailed description of this type is given in section 10. The function *find_str_all (str s)(args a)* returns all positions of sub-string s. The built-in function *count (args a)(int n)* returns the number of passed arguments. The *str*-function *sin (str sep, any :$ v1,v2,...) (bool)* read the variables v1, v2, ... sequentially from the string using the separator sep and returns true if all variables are read successfully. The operator “:$” indicates a call-by-alias, i.e. the transferred arguments are changeable within the function. The separator for reading is set with the type procedure *str:set_in_spc (str s)*. Moreover the operator “..” is used the first time. It returns a range of integer or character values. Finally the global procedure *return (int n)* is used the first time. It quits the surrounding function n-times immediately. An argument greater than 1 may be used for recursive functions.


```
str feasible_chars(0..9,':');

func check (str s)(bool err)
{
  err = false;
  char c;
  for(c,s.args()) {
    if(count(feasible_chars.find_str(str:(c))) == 0) {
      outl(s, ": infeasible character '", c, "'");
      err = true;
      return(1)
    }
  }
}

str zeile;
str:set_in_spc(" ");

while{ inl(zeile) }
{
  switch(zeile) {
    case("q","quit") { break(1) }
    case {
      str t;
      while{ zeile.in(t) } {

      if(check(t)) continue(1);

        int h,m,s = 0,0,0;

        switch(count(t.find_str_all(":"))) {
          case(0) {
            t.sin(":",s);
            h, s, m, s = s/3600, s - h*3600, s/60, s - m*60;
            outl(t, " = ", h,":",m,":",s)
          }
          case(1,2) {
            t.sin(":",h,m,s);
            outl(t ," = ", h*3600+m*60+s)
          }
          case {
            outl(t,": too many characters ':'");
            continue(1)
          }
        }
      }
    }
  }
}
```


7 Lists

The built-in type *list* represents polymorphic lists, i.e. objects of any type can be inserted into a list. For example, the following Nepal program "list.npl" generates a polymorphic list of integers, characters, and strings, and writes the list to the console.

```
list l(3..1,'a'..'f',"hello");
outl(l)
```

Executing the program results in:

```
(3,2,1,a,b,c,d,e,f,hello)
```

An iteration over all list elements is possible using the object procedure *for()* as demonstrated by the following Nepal program "list2.npl". Here the built-in type *any* represents a fully polymorphic type. Objects of any type can be assigned to this type. Its object function *type()* returns the type name of the object currently assigned to the corresponding variable.

```
list l(1,'a',"hello");
any x;
l.for(x) outl(x, " ", x.type());
```

Executing the above-mentioned Nepal code generates the output:

```
1 int
a char
hello str
```

Another example for using lists is the following Nepal program "subset.npl". Here all sub-lists containing exactly m elements are calculated for a given list l. The result is returned as a list r of sub-lists. Several object procedures and functions are used for type *list*: The procedure *Ins0 (any v1, v2, ...)* appends elements at the end of the list. The function *list (int i1,i2,...)(list)* extracts the elements with the indices i1, i2, ... and generates a new sub-list. Two lists can be concatenated using the procedural operator *"+" (list a,b)*. Moreover the built-in type function *list:"" (any v1, v2,...) (list)* is used which generates an anonymous list containing the elements v1, v2, ... The ".:"-operator returns an ascending range of integers or characters (for example the range "3.:2" is empty).

Concerning the transfer of input arguments for procedures or functions, the following convention is used. Small types like integers are always copied. For big types like lists, either a reference on the list or the value of the list is copied depending on the question if the argument is a variable or not. In the following example the arguments `l.list(i+1 .: n)` and `list:(1..4)` do not represent a variable. Therefore the values of these lists are copied for the function call.

```
func subset (list l; int m)(list r) {
  int n,i = l.size()-1;

  if(m == 0) {
    r.Ins0(list:())
  } #else {
    for(i, 0 .: n) {
      list x;
      subset(l.list(i+1 .: n),m-1).for(x)
        r.Ins0(l.list(i) + x);
    }
  }
}
outl(subset(list:(1..4),3));
```

Executing the above-mentioned Nepal code generates the output:

```
((1,2,3),(1,2,4),(1,3,4),(2,3,4))
```


The last example for using lists is the following Nepal program “list3.npl”. It shall demonstrate polymorphic lists with respect to values and references. Moreover the program shows that operators for assignment and comparison of data objects are generic for built-in types. The example inserts a copy and a reference of a list l0 into a list l3. The initialisation procedure *any:””()* is used to create a constant copy of the addressed list l0. The system function *ref()* yields true (or false) if the addressed object contains a reference (or not). The assignment of list l3 to list l4 using the operator “=” performs a deep copy of the original list. The operator “==” yields true (or false) if the two addressed objects are identical (or not).

```
list l0("0");
list l3(any:(l0),l0);

any x;
l3.for(x) outl(x, " ", x.type(), " ", ref(x));

list l4 = l3;
outl(l3, " ", l4, " ", l3==l4);
```

Executing the above-mentioned Nepal code generates the output:

```
(0) list false
(0) list true
((0),(0)) ((0),(0)) true
```

8 Hash arrays

The built-in type hash represents an associative array consisting of key-value pairs. The keys and values are fully polymorphic.

The first example is the following program “hash.npl” which counts all words in a file.

```
file f("hash.npl");
f.set_in_spc(" ", "\t", "\n");
hash h;
str s;
while { f.in(s) }
{
  h[s] += 1
}
outl(h)
```

Saving the above-mentioned code in a file “hash.npl” and executing the command "nepal hash.npl" on the console should result in the following output:

```
[hash->1,h[s]->1,"\n");->1,str->1,f.set_in_spc("->1,f("hash.npl");->1,{->2,}->2,+=->1,",->1,outl(h)->1,1-
>1,f.in(s)->1,file->1,h;->1,"\t",->1,while->1,s;->1]
```

The “[]”-operator of the hash array returns an alias on the value of the given key. In this way it provides a reading and writing access to the contents of the array. The semantics of the file-procedure *set_in_spc()* and file-function *in()* is the same as for inputs from the console or a string (cf. sections 3 and 6).

The second example “table.npl” shall demonstrate how to realise manipulations of ascii files by using hash arrays. The task is to filter columns from 2 to 4 out of a table consisting of five columns, and then to output these columns in inverse order. The program reads the lines sequentially from file “table.in” using the built-in procedure *file:for(str :$ s)*. Every line is split into five parts using the str-function *split (any,...)*. These parts are assigned to hash array h. Finally using the range "3..1" as key of the “[]”-operator yields the intended transformation.

```
file f1("table.in"), f2("table.out");
f2.set_out_spc(" ");
str s;
f1.for(s) {
  hash h;
  h[0..4] = s.split();
  f2.outl(h[3..1])
}
```

9 The type code

In Nepal user-defined control structures can be realised by using the built-in type code. For example, the following procedure repeat() represents an iterator which executes code c exactly n times. The procedure exec() of type code is applied to execute code c.

```
proc repeat (int n; code c) {
  int i;
  for(i,1..n) c.exec();
}

repeat(5) outl("hello world !");
```

Such “macro” procedures and functions can be even used within user-defined types (see section 14) in order to realize iterators similar to the procedure *for()* of the built-in type list.

10 The type args

The built-in type *args* exists to realize variable lists of arguments for procedures and functions. This type supports the same procedures and functions as type list, but the behaviour at the interface to procedures and functions is different. In the following example "my_for.npl", the built-in loop of the form "for(i,i_1,i_2,..,i_n){a}" is implemented once more using Nepal code. The argument x of type *any* is passed by alias (indicated by the operator “:$”). This means that the access of variable x inside the procedure “my_for” is equivalent to a direct, reading or writing access of the corresponding variable i from the calling scope. The arguments a and c are passed by reference since they correspond to big types. This only allows for a reading access to the transferred data.

```
proc my_for (any :$ x; args a; code c) {
  a.for(x) c.exec();
}

int i;
my_for(i,1..10) outl(i);
```

Executing the above-mentioned example yields the line-oriented output of all integers from 1 to 10.

In Nepal also the return values of functions can be handled as variable lists. To demonstrate this feature, the built-in functions “range” from the programming language Python shall be implemented once more.

```
# range similar to Python

func range (int a,b,d)(args r) # version 1
{
  if(d == 0) { outl("infeasible range value d =", d); exit(1) }

  oper vgl = if(d > 0){ <= } { >= };
  int i;
  for{ i = a } { vgl.eval(i,b) } { i += d } {
    r.Ins0(i)
  }
}

func range (int a,b)(args r) # version 2
{
  r = args:(range(a,b, if(a <= b){ 1 } { -1 }))
}

func range (int b)(args r) # version 3
{
  r = args:(range(0,b))
}

set_out_spc(" ");
outl(range(3,11,2), range(11,0), range(11))
```

The first version of function range() returns all integers from a to b with step size d as list r. The built-in type *oper* is similar to type code and can be used to merge code fragments which only differ with respect to the usage of operators. The function *exec()* of type oper with two arguments is called for binary operators and returns the according value of the current operator. For the second version of function range(), the absolute value of the step size is set to 1. Finally the third version sets the starting value a to 0. To minimise code redundancy, version i is calling version i-1 (for i = 2,3). The usage of the built-in type function *args:”” (any v1,v2, ...)(args)* is necessary to “pack” the result of the function range() into an object of type *args*. Executing the above-mentioned code results in the following output:

```
3 5 7 9 11 11 10 9 8 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9 10 11
```

11 The function func

Symbolic function calls can be realised with the built-in function *func (str,any,...)(any,..)*. For example, the following function map() calls the function with name f and exactly one argument for all arguments in the variable list a. The results are saved in the return list l. The concrete application of function cube() within the following example “map.npl” yields all cubic numbers between 1 and 1000.

```
func cube (int x)(int c) { c = x^3 }

func map (str f; args a)(list l)
{
  any x;
  a.for(x) l.Ins0(func(f,x));
}

outl(map("cube",1..10))
```


12 The procedure system

The built-in procedure *system (str s)* is used to call functions directly from the operating system. For example, the following Nepal program “system.npl” starts the two Nepal programs "hello.npl" and "hash.npl" sequentially.

```
list l("hello.npl", "hash.npl");
str s;
l.for(s) system("nepal " + s);
```

13 Graphs

Mathematical graphs consist of nodes and edges and are useful data structures to model objects with interdependencies, e.g. computer networks. Computers may be represented as nodes of a graph. Connections between the computers may be modelled by edges of a graph. Nepal 1.1 includes the built-in type *graph* with fully polymorphic nodes and edges. This data type supports both directed and undirected graphs as well as loops and parallel edges. The following example “graph.npl” calculates the minimum spanning tree of a graph by applying the well-known algorithm of Kruskal.

```
# Kruskal's algorithm to determine the list of edges (pairs)
# of a minimum spanning tree

func kruskal(graph :$ g)(list l; int sum)
{
  int n = g.size(true); # number of nodes
  array a(n,1..n); # indices of components

  g.sort(false); # sort edges in ascending order

  any e; any2 p;
  g.for(e,p) {
    if(a[p[0]] != a[p[1]])
    {
      # merge two components:
      any v;
      g.for(v)
        if(a[v] == a[p[1]])
         a[v] = a[p[0]];

      l.Ins0(any:(p)); # insert copy of pair
      sum += e; # add length of edge
    }
  }
}


graph G(0..6);
G.ins(0->1,3,1->2,4,0->3,1,1->3,5,1->4,3,2->4,1,
      3->4,11,3->5,2,4->5,4,4->6,5,5->6,7);

outl(G);
outl("#nodes=", G.size(true), " #edges=", G.size(false));
soutl(" ", kruskal(G));
```

Firstly the algorithm defines every node of the graph to be an individual component. Then the edges are sorted in ascending order with respect to their length. Finally the sorted edges are traversed sequentially, and the considered edge either merges the incident components (trees) to form a bigger tree or the edge is skipped since the edge is connecting the same component.

In the above-mentioned example, a graph G is initialised with 7 nodes containing the indices between 0 and 6. Then 11 edges are inserted using the built-in function *graph:ins(any2 i1, any c1, ...)*. Here the built-in type *any2* represents a polymorphic pair, modelling the index of an edge in the form “<index of source node> → <index of target node>”.

Then the graph is passed by alias (indicated by the operator ":$") to the function kruskal() because the graph is modified using the built-in function *graph:sort(bool)*. The graph components are modelled with the built-in type *array*. The size of the object of this type is initialised using the function *graph:size(bool)* to determine the number of nodes (or edges). For iteration over all nodes or edges of the graph, the object procedure *graph:for()* is used. The return values of the function kruskal() are the list of edges of the minimum spanning tree (l) and the total length of the tree (sum). Executing the above-mentioned code results in the following output:

```
<(0,1,2,3,4,5,6),{<0->1,3>,<1->2,4>,<0->3,1>,<1->3,5>,<1->4,3>,
<2->4,1>,<3->4,11>,<3->5,2>,<4->5,4>,<4->6,5>,<5->6,7>}>
#nodes=7 #edges=11
(0->3,2->4,3->5,0->1,1->4,4->6) 15
```

14 User-defined types

14.1 Definition

The programming language Nepal supports the definition of new types in the form

```
smalltype|bigtype <type_name> (<base_1>,...,<base_n>)
  {
     <type> <var_1>,...,<var_n>; ...
     proc <proc_1> (...) { ... }; ...
     func <func_1> (...)(...) { ... }; ...
     smalltype|bigtype <inner_type_1> ...; ...
  }
```

Here a new type with name type_name is introduced. It inherits all properties from the base types base_1 through base_n, i.e. multiple inheritance is available. The first line of the definition block describes object variables (any object of the given type contains data attributes accessible via these variables). The second line defines an object procedure (any object of the given type can call this procedure). The third line defines an object function (any object of the given type can call this function). The forth line defines a local type which is usable only within type type_name. Types, procedures, and functions can be nested with unlimited depth. Therefore, local procedures / functions within a procedure / function can be defined similar to the programming language Pascal. The order of variables, procedures, functions, and types within a type definition block is arbitrary.

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. 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. Big user-defined types are specified with the keyword *bigtype*.

Accessing object variables, procedures and functions is done via the "."-operator (e.g. "O.v" or "O.f()"). The access to the current object within an object procedure or function is supported by the predefined variable "this".

The following conventions are valid for inheritance: All object procedures and functions are virtual. The search order for accessing object variables, procedures, functions, and types is "breath first - depth second". Cycles within inherited types are not allowed.

Standardly, the objects for all variables of a certain scope are allocated on the program stack when the program is entering the scope. Before exiting the scope, all objects are deallocated. Objects contained in a certain object (e.g. partial trees of a tree) are deleted recursively during the destruction.
An optional default initialisation with a constant value can be applied to any object variable of a user-defined type. For example, the following implementation of a type "my_list" initialises the size of the list to zero.

```
bigtype my_list {

  int n(0); # size of list
  ...
}
```


Moreover an optional initialisation procedure “” (with arbitrary signatures) can be defined for a user-defined type. It is executed with the statement of the data definition. For example, the Nepal interpreter transforms the statement

```
str s("hello","how","are","you"), t("i'm","fine");
```

at the beginning of the program execution into the statement

```
str s.""("hello","how","are","you"), t.""("i'm","fine");
```

Therefore, the built-in procedure *"" (args)* of type str is executed twice with this statement.

14.2 Modularisation concepts

For a better modularisation of large types their object procedures and functions can be defined also outside of the definition block as follows:

```
<type_name>:<proc_or_func_1> (...)(...) { ... }
```

An additional declaration of this procedure or function within the definition block of the given type is not necessary.

The second modularisation concept is the inclusion of files in the form:

```
need "<file_name>"
```

When the program execution reaches this statement, the content of this file is executed. Definitions of types, variables, functions and procedures in this file can be addressed in the current file. Recursive inclusions are also possible. However, a circular inclusion is not allowed and raises an error. Multiple inclusion of a certain file is eliminated automatically. The use of inclusions allows for the definition of external object procedures and functions (as introduced above) in a separate code file.

A third modularisation concept are type extensions. The idea is to split a program into application types (implementing the main procedures and functions) as well as data types (implementing the models). Typically special extensions of the data types are necessary for special applications. To model this, Nepal allows for the definition of additional source code within the application types. When starting the program the interpreter moves the code into the addressed data type automatically. It is also possible to introduce an additional base type. In the following example, the application A extends data type D1 with special source code and data type D2 with a special base type D2A.

```
bigtype D1 { ... }  bigtype D2 { ... }
bigtype D2A { ... }

bigtype A {
   D += { ... }
   D2 += D2A
   ...
}
```

The application-specific extensions are also performed for all base types of the considered application. To make use of type extensions, the interpreter has to know the relevant application for which the special extensions shall be applied. This is achieved by using the program option “-a”. For the above-mentioned example the Nepal interpreter is called as follows:

```
nepal -a A <code_file>
```

14.3 Example 1

The following program “my_list.npl” implements a data type for a generic list similar to the built-in type *list*. The provided functionality serves for a second realisation of the above-mentioned examples “list.npl”, “list2.npl”, “subset.npl”, and “list3.npl”.

The data type “elem” represents an element of the list. It contains the object variable x for storing the content. The variable next contains the succeeding element of the list. The initialisation procedure “” uses the operator “?” to initialise variable x, i.e. this variable contains either a value or reference depending on the question if the transferred object is a value or reference. As stated earlier, this depends on the question if the arguments of the procedure correspond to small or big types (e.g. *int* or *list*).

The data type “my_list” contains an object variable first storing the first element of the list. The variable last holds a reference to the last element. The current size of the list is stored in variable n. Since the initial state of a list is empty, variable n is initialised by 0 per default.

To realise example “list.npl” the initialisation procedure “” is implemented using the procedure “Ins0” for appending data objects at the end of the list. Here the operator “~” is used to move an element e created on the program stack to the appropriate variable of the current list object. This statement demonstrates the basic approach of Nepal to support dynamic data structures without using new operators, delete operators, and garbage collectors. The operator “@” is used to establish the reference to the last element. Moreover a function out() is introduced to transform the list into a string. In this way the list can be printed to the console using a special format. If the function out() is not provided, the list object (like any object of a user-defined type) is printed due to a standard format.

To realise example “list2.npl” the iterator “for” is implemented using the operator “$” to establish an alias to the content of each element. In this way a direct, reading and writing access to object variable x is supported from a variable on the calling scope.

To realise example “subset.npl” the initialisation function “” is implemented to create anonymous list objects as arguments of procedures / functions. Moreover a function list() is provided to extract certain elements of the list, using the function get() to access the i-th element. Finally the operator “+=” is introduced for appending other lists to the considered list. Note that this operator defines the operator “+” for concatenating two lists (as used in example “subset.npl”) implicitly.

To realise example “list3.npl” no additional functionality is needed for type “my_list”. This shows that operators for assignment and comparison of data objects are generic not only for built-in types, but even for user-defined types.

```
# example for definition of user-defined types

bigtype elem { # data type for an element of a list

  any x;    # content
  elem next; # next element

  proc "" (any xx) { x ? xx } # initialiser
}

bigtype my_list { # data type for a list

  elem first, last; # first and last element
  int n(0);         # number of elements

  proc "" (args a) { # initialiser
    any x;
    a.for(x) Ins0(x);
  }

  func "" (args a)(my_list l) { # initialiser
    any x;
    a.for(x) l.Ins0(x);
  }

  func get (int i)(any x) { # return content of i-th element

      if(i < 0 || i >= n)
        throw(str:("my_list:get: index ", i, " out of range"));
```



```
      int c = 0;
      elem it;
      for{ it @ first } { it.def() } { it @ it.next } {

        if(c == i) { x ? it.x; break(1) }
        c += 1
      }
  }

  func list (args a)(my_list l) { # get contents of certain elements as a list
    any x;
    a.for(x) l.Ins0(get(x));
  }

  func size (int s) { s = n } # return number of elements

  proc Ins0 (any x) { # insert x at the end of the list

    elem e(x);

    if(n == 0)
    {
      first ~ e;
      last @ first;
    }
    # else
    {
      last.next ~ e;
      last @ last.next
    }

    n += 1;
  }

  proc for (any :$ x; code c) { # loop over all elements

    elem it;
    for{ it @ first } { it.def() } { it @ it.next } {
      x $ it.x;
      c.exec()
    }
  }

  func out (str s) { # transform list into string

    s = "(";

    any x;
    for(x) s += str:(x) + ",";

    if(n > 0) { s.Set(0,')') } #else { s += ")" }
  }

  proc "+=" (my_list l) { # append list l to list

      any x;
      l.for(x) Ins0(x);
  }
}
```



```
# test examples "list", "list2", "subset", and "list3"

my_list l1(3..1,'a'..'f',"hello");
outl(l1);

my_list l2(1,'a',"hello");
any x;
l2.for(x) outl(x, " ", x.type());

func subset (my_list l; int m)(my_list r) {
  int n,i = l.size()-1;

  if(m == 0) {
    r.Ins0(my_list:())
  } #else {
    for(i, 0 .: n) {
      my_list x;
      subset(l.list(i+1 .: n),m-1).for(x)
        r.Ins0(l.list(i) + x);
    }
  }
}

outl(subset(my_list:(1..4),3));

my_list l0("0");
my_list l3(any:(l0),l0);

l3.for(x) outl(x, " ", x.type(), " ", ref(x));

my_list l4 = l3;
outl(l3, " ", l4, " ", l3==l4);
```

Executing the above-mentioned Nepal code generates the output:

```
(3,2,1,a,b,c,d,e,f,hello)
1 int
a char
hello str
((1,2,3),(1,2,4),(1,3,4),(2,3,4))
(0) my_list false
(0) my_list true
((0),(0)) ((0),(0)) true
```

14.4 Example 2

14.4.1 Macro structure

The following program „calculator.npl“ implements a small calculator for integer values. Its functionality is split into the four parts “processor“, „data interface“, „user interface“ and „memory“ with the following tasks:

- Processor: This component calculates the value of a given expression.
- Data interface: Here the user-defined expressions are transformed into an internal data format. Moreover internally stored expressions can be transformed back to a user-readable format.
- User interface: This component serves for the interaction with the user.
- Memory: Here the user-defined expressions are stored.

According to the above-mentioned functional split we define the four types „processor“, „interface“, „calculator“, and „memory“. The application „calculator“ inherits the procedures and functions from the three other components. Each component is realised in a code file with the same name. The macro structure of file „calculator“ is as follows. The user interface is implemented by the object procedure run() of type „calculator“.

```
# a demo program for type extensions

need "memory.npl", "processor.npl", "interface.npl";


# the application

bigtype calculator (memory,interface,processor) {

  proc run { ... }

} # end calculator


# main

calculator calc.run();
```

The execution of the example is started with the following command:

```
      nepal –A calculator calculator.npl
```

14.4.2 The data type syntax_node

The internal model for expressions is realised by the type “syntax_node” representing a node of a syntax tree. The tree represents arithmetic expressions containing either non-negative integers or the operators '+', '-', '*' or '/'. So a feasible expression would be "234*34-14546+3/10". The type “syntax_node” contains the object variable “value” to store the number or operator. The variables “l” and “r” contain the left and right sub-tree within the binary syntax tree. The type contains the object procedure „“ to initialise the node via a string and the iterator „for“ to traverse all nodes of the tree. The global hash table “prio” defines the priority of the operators. The global string “feasible_chars” contains all valid characters as part of an expression.

```
# a type demo

hash prio("+",1,"-",1,"*",2,"/",2);
str feasible_chars(prio.keys(),0..9);


# data type to model a syntax tree
# for representing simple formulas

bigtype syntax_node {

  str value; # an integer value or an operator +,-,*,/
  syntax_node l,r;  # the left and right sub-tree

  proc "" (str s) { value = s } # initializer

  # iterator over all tree nodes

  proc for (syntax_node :$ x; code c) {
    if(l.def()) l.for(x,c);
    x $ this;
    c.exec();
```



```
    if(r.def()) r.for(x,c);
  }
} # end syntax_node
```

14.4.3 The component processor

The type “processor” extends type “syntax_node” by a recursive function eval() which evaluates the syntax tree numerically. Here the built-in function “” of type *oper* is used which generates a local object of this type. Its object function eval() is used to evaluate the considered partial expressions. The used built-in function int() transforms a string value into an integer value.

```
need "syntax_node.npl";


# extend the data for evaluation

bigtype processor {

  syntax_node += {

    # bottom-up evaluation of the tree

    func eval (int res) {
      res = if(prio.def(value)) {
        oper:(value).eval(l.eval(),r.eval())
      } #else {
        int:(value)
      }
    }
  }
} # end processor
```

14.4.4 The component interface

The type “interface” extends type “syntax_node” by functions to convert the expressions from an external string format into the internal tree format and vice versa. The generation of the syntax tree is split into two parts. The scanner (local function scan()) generates a token list tl from the input string s. Then the parser (object function parse()) generates the syntax tree from the token list, considering the priorities of the operators according to hash table “prio”. If an error occurs within scanning or parsing, an exception is thrown by using the built-in procedure *throw()*. A successfully generated syntax tree can be transformed back into a string by using the object procedure print().

```
need "syntax_node.npl";


# extend the data for user input and output

bigtype interface {

  syntax_node += {

    # recursively printing the tree

    func print (str s) {

      syntax_node y;
        for(y) s += y.value;
    }
```



```
    # parse the string s and generate a syntax tree in this

    proc parse (str s) {

      parse(scan(s));

      # scan the formula s and generate a token list tl

      func scan (str s)(list tl) {

        char c;
        for(c,s.args())
          if(count(feasible_chars.find_str(str:(c))) == 0)
             throw(str:("infeasible char ", c));

          str ss(s);
        str o;
        for(o,prio.keys()) ss.repl_all(o," "+o+" ");
        tl = list:(ss.split());
      }
    }


    # parse the token list tl and generate a syntax tree in this

    proc parse (list tl) {

      int pos_op,pos = -1;
      for(pos,0..tl.size()-1)
        if(prio.def(tl.get(pos)) && (pos_op == -1 || prio[tl.get(pos)] <=
           prio[tl.get(pos_op)])) pos_op = pos;

      switch(pos_op)
      {
        case(0,tl.size()-1) throw("syntax error");

        case(-1) {
          syntax_node tt(tl.get(0));
          this ~ tt
        }

        case
        {
          syntax_node tt(tl.get(pos_op)), ll(""), rr("");
          this,l,r ~ tt,ll,rr;
          l.parse(tl.list(0 .. pos_op-1));
          r.parse(tl.list(pos_op+1 .. tl.size()-1))
        }
      }
    }
  }
} # end interface
```

14.4.5 The component memory

The type “memory” stores all user-defined expressions within an internal hash table “tbl” in the form of syntax trees. Moreover the trees are saved on an external file whose name is stored in attribute “filename”. The object procedure ins() is provided to insert a new expression. The initialisation procedure any:””() is used to create a constant copy of the transferred tree f. Note that the statement “tbl.ins(id,f)” stores only a reference on tree f in table tbl. A tree is associated

with an identifier “id”. Using this name and the procedure del(), the tree can be removed from the table again. The procedures load() and save() serve for reading and writing the complete hash table from and to a file, respectively. Here the built-in procedures pin() and pout() are used. They allow for the “packing” and “unpacking” of arbitrary types (both built-in and user-defined types) to/from an internal string format.

```
need "syntax_node.npl";

# data base for formulas

bigtype memory {

  str filename; # file where data are saved
  hash tbl; # function "id of data" -> data

  proc "" (str s) { filename = s }
  proc ins (str id; any f) { tbl.ins(id,any:(f)) }
  proc del (str id) { tbl.del(id) }

  proc load
  {
    file f(filename);
    f.pin(tbl)
  }
  proc save { file:(filename).pout(tbl) }
}
```

14.4.6 The application calculator

In the following section the source code for type calculator is stated completely. At the beginning of procedure run() the data base is initialised by calling the inherited object procedure memory:“”. Then the data base is read from the file “db.txt” via the procedure memory:load(). Finally the main loop is reached where the user can enter the following commands.

- i<id>: Here a new expression is entered. Syntax errors are caught with the built-in procedure *catch*() and are printed to the console.
- d<id>: This command deletes an existing expression.
- c<id>: Here an existing expression is evaluated numerically.
- C<id>: Here the Nepal interpreter itself can be used for calculating the expressions in order to compare the result with the output of the previous command. To do this, the expression to be calculated is saved on a temporary file, and the interpreter is called by use of the built-in procedure system().
- l: This command lists all expressions from the data base.
- q: Here the user can quit the application.

```
# a demo program for type extensions

need "memory.npl", "processor.npl", "interface.npl";


# the application

bigtype calculator (memory,interface,processor) {

  proc run {

        memory:("db.txt");
        load(); # load the database

        for {
```



```
      out(">>> ");
      str comm;
      inl(comm);

      if(comm.size() == 0) continue(1);

      str id(comm.str(1 .: comm.size()-1));

      switch(comm.get(0)) {

        case('q') { return(1) }

          case('l') # list
          {
            tbl.for(id)
              outl(id, " '", tbl[id].print(), "'");
          }

          case('i') # insert
          if(check_id(id,true))
          {
            str f,err;
            out("Formula=");
            inl(f);
            syntax_node xx.parse(f);
            catch(err) { outl(err); continue(1) }
            ins(id,xx);
            save();
          }

          case('d') # delete
          if(check_id(id,false))
          {
            del(id);
            save();
          }

          case('c') # calculate
            if(check_id(id,false))
              outl(tbl[id].print(), " = ", tbl[id].eval());

          case('C') # calculate (version 2)
          if(check_id(id,false))
          {
            file f1("tmp.txt"), f2("tmp.txt.out");
            f1.outl("out(", tbl[id].print(), ")");
            f1.close();
            system("nepal -o tmp.txt.out tmp.txt");

            int res;
            f2.in(res);
            outl(tbl[id].print(), " = ", res);
          }

          case { outl("Unknown command '", comm, "'") }

        } # end switch
      } # end for
}
```



```
  func check_id (str id; bool exist)(bool ok) {
      ok = true;
      if(! tbl.def(id))
      {
        if(! exist)
        {
          outl("Formula '", id, "' does not exist");
          ok = false;
        }
      }
      #else
      {
        if(exist)
        {
          outl("Formula '", id, "' already exists");
          ok = false;
        }
      }
  }
} # end calculator


# main

calculator calc.run();
```

