Basic Libraries − Reference Manual

Mjølner Informatics Report
MIA 90−08
March 2004

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

List of Programs

1 The Basic Libraries

This document contains documentation of the basic libraries of the Mjølner System. It is a programmer's guide. The purpose is to provide the necessary information to the users of the libraries. The facilities are documented by means of the BETA interfaces and examples of how to use the facilities.

This document includes the documentation on the following libraries of the Mjølner System:

- betaenv.bet contains the most basic library of the Mjølner System. It contains patterns describing character, integer, streams, exceptions, etc. as well as control patterns and input/output patterns. Most (if not all) BETA programs will use betaenv, either directly or indirectly.
- math.bet contains an interface to standard real functions in BETA. The library contains patterns for mathematical functions: trigonometric, hyperbolic, exponential and logarithmic, floating point manipulation, power, miscellaneous constants.
- numberio.bet contains patterns to be used for reading numbers from any input stream, and for writing any number to any output stream. The patterns are able to read and write all numeric types of the BETA language.
- formatio.bet contains two patterns for making formatted input and output on any stream.
- random.bet contains an elaborate random generator system, containing random generators with many different statistical properties.
- regexp.bet contains facilities for working with regular expressions in text strings.
- file.bet contains the general interface into files, residing on some file system.
- directory.bet contains the general interface into directories on hierarchical file system.
- systemEnv defines the experimental concurrency system for BETA. SystemEnv contain three closely related libraries: basicsystemenv.bet, systemenv.bet, and timehandler.bet.
- repStream.bet contains the definition of a special type of repetitions that has stream−like operations.
- external.bet contains the various facilities for enabling BETA programs to interface to external languages, like C and Pascal.

along with a few other, minor libraries.

1.1 The betaenv Library

When programming in BETA, the basic BETA environment betaenv is utilized. This chapter describes how to use the facilities in betaenv.

Betaenv contains several attributes that are used in any BETA program. That is, each BETA program must have betaenv in its origin path.

The patterns in betaenv are divided into several different categories such as character patterns, integer patterns, boolean patterns, control patterns, input/output patterns, stream and exception patterns.

The first section of this chapter describes how betaenv is used in general, while the subsequent sections concern the individual patterns.

1.1.1 Mjølner System Version

The mbs_version.bet library implements a few operations which can be used in BETA programs to test the actual release number of the Mjølner System. See the interface files for details.

1.1.2 Backward compatability

This version of betaenv contains a few important changes, which are not backward compatible. The most important are:

- The text attribute findCh has been renamed to findAll.
- The text attributes copyAppend and copyPrepend have been removed. To gain the same effect, you can write:

```
'...'->(t.copy).append
```

- or

```
'...'->(t.copy).prepend
```

To ease the process of porting your code, we have included a small fragment called betaenvold.bet containing these obsolite facilities.

1.2 Using the betaenv Library

The basic structure of betaenv is realized by means of the Mjølner fragment system. It is as follows:

```
BODY 'betaenvbody'
--- betaenv: descriptor ---
(# ...
(* A lot of useful patterns *)
...
theProgram: @<<SLOT program: descriptor>>;
<<SLOT lib: attributes>>
...
do theProgram
#)
```

The program slot must be filled by the user and can have the following form:

```
ORIGIN '~beta/basiclib/betaenv'
--- program: descriptor ---
(#
do 'This is a small BETA program' -> puttext
#)
```

lib slot

The lib slot makes it possible to add attributes to betaenv. An example of this is the following which might be in a file called stack.bet:

```
ORIGIN '~beta/basiclib/betaenv'
--- lib: attributes ---
stack:
(# push: (# e: @integer enter e do ... #);
   pop: (# e: @integer do ... exit e #);
   empty: ...
   ...
#)
```

The stack can then be used as follows:

```
ORIGIN '~beta/basiclib/betaenv';
INCLUDE 'stack'
--- program: descriptor ---
(# s: @stack
do 7 -> s.push;
   s.pop -> putint;
#)
```

1.3 Basic Patterns

1.3.1 Simple types

Betaenv contains definitions of the basic patterns char, integer, real, boolean, true and false that are predefined in the BETA language (i.e. built−in data types). These patterns are self−assignable. This means an integer object can be assigned to another integer object, a char object can be assigned to another char object etc.

For efficiency reasons, the usage of the basic patterns char, integer, real, boolean, true and false is somewhat restricted, compared with all other patterns in the system. These restrictions are:

- They cannot be used as superpatterns to other patterns. E.g. subInteger: integer(# ... #) is illegal.
- Dynamic references to instances of these basic types cannot be obtained. E.g. var[] −> ... is illegal if var is an instance of one of these basic patterns.
- Dynamic references to basic patterns cannot be declared. E.g. var: ^integer is illegal.

True object oriented patterns for integers, characters, reals and booleans are also part of the system (see later). However, using those patterns impose an execution overhead compared with the basic patterns.

1.3.2 The Integer Pattern

1.3.3 Arithmetical operators

Besides the arithmetical operations: +, −, *, div, and mod and the relational operations: =, <>, >, >=, < and <=, the Min, Max and Abs patterns are defined for integers. The patterns MaxInt and MinInt returns the largest (respectively smallest) integer on the machine.

```
(# i,j,k: @integer
do 3 -> i;
   '? '->puttext; getint->j; (* read an integer from keyboard *)
   ((i,j) -> Max, j+5)  -> Min -> k;
   'k is '-> puttext; k -> putint; newline;
   (if k-1
    // 5 then ...
    // 17 then ...
   if)
#)
```

1.3.4 The Boolean Pattern

1.3.4.1 Logical operators

In the current implementation, true and false return respectively the values 1 and 0. The unary operator not and the binary operators and, xor, and or can be applied to booleans.

Booleans are used in the traditional way:

```
(# aBoolean, anotherBoolean: @boolean;
   a: @integer
do ...
   (if aBoolean then
       ...
   else
     (a>7) and (a<17) -> anotherBoolean
   if)
#)
```

1.3.5 The Real Pattern

1.3.5.1 Arithmetical operations

The arithmetical operations: +, −, *, div (or /), and the relational operations: =, <>, >, >=, < and <= are defined for reals. The patterns MaxReal and MinReal (defined in math.bet) returns the largest (respectively smallest) real on the machine.

The following example shows how to use reals. The pattern putreal is described later.

```
(# x,y: @real;
do ... -> x;
   'Print the number 1.23:' -> putline;
   y -> putreal;
   newline;
   x -> y;
   (if x=y then 'x and y are equal!' -> putline if);
   '3.0*7.0 = ' -> puttext;
   3.0 * 7.0 -> putreal;
   newline;
   '(−4.0)*(−3.0) = ' -> puttext;
   −4.0 * (−3.0) -> putreal;
   newline;
   '(−4.0)/8.0 = ' -> puttext;
   −4.0 div 8.0 -> putreal;
   newline;
#)
```

1.3.6 The Char Pattern

1.3.6.1 ASCII

The char pattern enables the manipulation of characters. Characters can be expressed as literals or as the corresponding ASCII values. The pattern Ascii defines all non−printable characters as constants (such as null, nl, cr, esc, del). Newline however, is a variable containing either nl or cr depending on the computer.

Ascii also contains local patterns for various conversions and testings of characters. E.g. the

patterns IsDigit, IsLetter, IsUpper and IsLower are provided for determining the kind of a character. IsSpace testes whether the character is sp, cr, nl, np, ht or vt. Conversion is available through the upCase and lowCase patterns.

The following example shows how to use char and upCase:

```
(# a,b: @char
do 'a' −> a;
   98 −> b;
   (if a −> Ascii.upCase
    // 'A' then ...
    // 'B' then ...
    ...
    if)
#)
```

1.3.7 The Repetition Pattern

The BETA compiler also implements repetitions. Betaenv contains the repetition pattern, defining the available operations on repetitions (apart from the lookup operation: []). These operations are range, new and extend. Range returns the number of repetition positions, new makes it possible to allocate an entire new repetition, and extend is used for dynamically extension of the repetition. Note that the repetition pattern cannot be used as a superpattern:. Also note, that it is only allowed to make repetitions of integer, boolean, char, real, and object references.

1.4 Basic Object Patterns

The basic object oriented patterns are the object pattern and the object oriented variants of the basic patterns.

1.4.1 The Object Pattern

The object pattern functions as the implicit superpattern: for all patterns which do not have any explicit superpattern:. The pattern object is defined as follows:

```
object: (* general superpattern: *)
   (# _struc:
        (#
        exit this(object)##
        #);
     _new:
        (# O: ^object
        do ...
        exit O[]
        #)
   do inner
   #);
```

The attribute _struc returns a reference to the structure of the current object (i.e. a pattern reference to the pattern from which this object was created). The attribute _struc is maintained in v1.6 for backward−compatability reasons. It will be removed in next release, since it will become obsolite, since O## is allowed in the case where O is the name of an object. However, previous releases of the compiler disallows this construct in some cases due to an error. Only in these cases, it is recommended to use the O._struc construct.

The attribute _new returns a new object, that is qualified exactly as THIS(object). This new object is default initialized.

1.4.2 Objects of simple types

All patterns except char, integer, real, boolean, true and false are subpatterns of Object. To enable handling integers, reals, characters and booleans like any other objects in the system, the patterns charObject, integerObject, realObject, booleanObject, trueObject, and falseObject have been introduced. They are genuine patterns, corresponding to the basic patterns, described above. They are specializations of Object that can be used instead of the basic patterns. They have all properties ordinary patterns have (in contrast to the basic patterns).

1.4.3 The charValue, integerValue, booleanValue and realValue Patterns

The charValue, integerValue, booleanValue, and realValue patterns are object oriented variants of the basic patterns (i.e. built−in patterns). They are used in all cases where a value of the given type is needed. E.g. a booleanValue may be used as superpattern for patterns returning boolean results.

1.4.4 The charObject, integerObject, booleanObject, trueObject, falseObject and realObject Patterns

These patterns are subpatterns: of the charValue, integerValue, etc. patterns above, and gives the final functionality to allow these variants of the basic patterns to be used a genuine patterns.

These patterns can for instance be utilized when programming general data structures. Consider a data structure list which defines its element type as a virtual Object.

```
list:
(# element :< object;
   insert: (# e: ^element enter e[] do ... #)
   remove: (# e: ^element do ... exit e[] #)
#)
```

When list is applied in a specific application, the element type is bound. The following is an example of how a list of integers and a list of editors can be declared and manipulated:

```
(# integerList: @list(# element::< integerObject #);
   editor:
     (# window: ...
        menus: ...
        cut: ...
        copy: ...
        paste: ...
     #);
   ed: ^editor;
   editorList: @list(# element::< editor #);
   io: ^integerObject;
do &integerObject[] −> io[]; 7 −> io;
   io[] −> integerList.insert;
   ...
   editorList.remove −> ed[];
   ...
#)
```

1.5 The Streams Patterns

A stream is a generalization of internal and external text objects. An internal text object (text) is a sequence (repetition) of chars. An external text object (file) corresponds to a traditional text file. Stream, text and file are organized in the following hierarchy:

Stream, text, and file

```
stream: (# ... #);
text: stream(# ... #);
file: stream(# ... #);    (* described in a later chapter *)
unixFile: file(# ... #);(* described in a later chapter *)
macFile: file(# ... #); (* described in the Macintosh
                           * Library manual *)
```

1.5.1 The Stream Pattern

The stream pattern is an abstract superpattern: which provides general stream manipulating procedure patterns: getPos, setPos, eos, length, reset, newline, put, get, peek, putint, getint, puttext, gettext, putline, getNonBlank, getline, getAtom, scan, scanBlanks, scanToNL and scanAtom. The stream pattern also defines exception patterns (e.g. EOSerror). See the interface of stream for more details.

1.5.2 Substreams

There is an additional library called substreams.bet, which implements a substrems concept (and a subtext concept). A substream refers to a (consequtive) portion of another stream. Manipulations on the substream will thereby actually change that portion of this other stream. All usual stream operations applies to a substream. See the interface of substream for more details.

1.5.3 The Text Pattern

The text concept is intended for 'small' texts, but there is no size limit. Some of the operations might however be inefficient on large text objects.

A text is a sequence of characters. The range of a text object T is [1,T.length]. A text can be initialized by executing T.clear or by assigning it with another (initialized) text. Like the predefined patterns integer, real, char and boolean, Text objects are self−assignable. A text constant has the form 'foo' or 'a'. The ' character may by specified as part of a text constant by repeating it, e.g. 'is''s like this' is the text constant: is's like this.

Besides defining the implementations of the abstract stream operations (e.g. put and get), the text pattern defines the following patterns: empty, clear, inxGet, inxPut, append, prepend, scanAll, sub, insert, delete, equal, equalNCS, less, greater, makeLC, makeUC, find, findAll, findText, findTextAll, copy, and asInt.

All error messages from exceptions originating from text objects are followed by the text lines:

```
Error in text which begins as follows:
<THIS(text)>.....
```

where <THIS(text)> is the text where the error occurred.

1.5.4 Text utilities

The library textUtils.bet contains a number of additional text attributes, such as getBoolean, putBoolean, set, setText, setInt, setBased, setReal, and setBoolean. See the interface of textUtils for more details.

The library texthash.bet contains a single operation: honeyman, which is an efficient and nearly optimal hash−function for text hash keys. See the interface of texthash for more details.

1.5.5 Using Text

This example gives examples of how to use the text pattern. The text object Records consists of a sequence of records. Each record has the form:

```
name Job:aJob Salary:aSalary /
```

The program shows various patterns for manipulating the Records text:

Program 1: textRecords.bet

```
ORIGIN '~beta/basiclib/betaenv';

--- program: descriptor --
(* Demo example shwoing examples of how to use the text concept from betaenv.
 * The text object Records, consists of a sequence of records. Each record
 * has the form:
 *    name Job:aJob Salary:aSalary /
 * The program shows various patterns for manipulating the Records text.
 *)
(# GetName: (* read next name from T *)
     (# T: ^text; T1: @text
     enter T[]
     do (* scan and skip until a letter is met *)
        T.scan(# while::<(#do NOT (ch->Ascii.isLetter)->value #)#);
        (* scan and read while letters in T *)
        T.scan
        (# while::<(#do ch->Ascii.isLetter->value #)
        do ch->T1.put
        #)
     exit T1
     #);
   GetRecord:
     (* Get the record with the name N  and return name and dat part *)
     (# N: ^Text; name,data: @Text
     enter N[]
     do Records.reset;
        FindName:
          (if not Records.eos then
              Records[]->getName->name;
              data.clear;
              (* scan and read until '/' is met *)
              Records.scan(# while::<(#do (ch<>'/')->value#) do ch->data.put #);
            (if not (N[]->name.equal) then restart FindName
        if)if)
     exit(name,data)
     #);
   GetJobAndSalary:
     (* get the  job and salary from data part, which is the part after name *)
```

```
     (# Data: ^Text; Job: @Text; Salary: @integer
     enter Data[]
     do Data.reset;
        Data[]->GetName; Data.get (* skip ':' *); Data[]->GetName->Job;
        Data[]->GetName; Data.get; Data.GetInt->Salary
     exit(Job,Salary)
     #);

   Records: @Text;
do '----------1:'->putLine;
   (*  initialize Records *)
   'John Job:Programmer salary:120000 / '->Records.append;
   'Joan Job:Doctor salary:130000 / '->Records.append;
   'Mary Job:Boss salary:140000 / '->Records.append;
   Records[]->putLine;

   '----------2:'->putLine;
   (* split Records into atoms *)
   Records.reset;
   scan:
   cycle
     (#
     do Records.getAtom->putline;
        (if Records.eos then leave scan
     if)#);
   '----------3:'->putLine;
   (* Find record with name Joan and decode data part *)
   (# Name,Data,Job: @ text; Salary: @Integer
   do 'Joan'->GetRecord->(Name,Data);
      'Ms. '->Name.prePend;
      ' II'->name.append;
      Name[]->putText; ' has the data: '->putText;
      Data[]->GetJobAndSalary->(Job,Salary);
      'Job='->putText; Job.makeUC;  Job[]->putText;
      ' Salary='->putText; Salary->putInt; newline;
   #)

#)
```

1.5.6 UniCode Text

This release contains an experimental implementation of an UniCode stream and UniCode text. The fragment is called wtext.bet, and a UniCode stream is realized by the wstream pattern and a UniCode text is realized by the wtext pattern.

Besides these two patterns, this fragment defines a conversion pattern ascii2wtext, which takes a regular BETA text and converts it into a UniCode text (a wtext instance). Furthermere, this fragment extends the text interface with one additional operation aswtext, which converts the text instance to a wtext instance.

See the interface file for wtext for further details.

1.6 Exceptions and Program Termination

There are two kinds of exceptions in the BETA programming language: Static exceptions and Dynamic exceptions. Static exceptions are used when you have an exception pattern and know on compile time where it is when you need it. Dynamic exceptions are exception patterns that can handle a specific type of error. They are located by traversing the call stack when an error occurs. This means, that the exception handler does not need to be in scope to be called.

1.6.1 Static Exceptions

The default action of a static exception is to stop the program execution and print an informative error message on the stream screen. In addition the file <programname>.dump contains a dump of the call stack. Exceptions use the pattern Stop for termination. Specific error messages can be defined by specializing the exception pattern. The attribute msg of exception is a text object that is used to accumulate error messages in the classification hierarchy of exceptions. If the programmer wishes to prevent the program execution from being stopped in order to handle the exception himself, the boolean attribute continue of exception must be set to true.

The static exceptions are often defined as virtual procedure patterns of other patterns (such as the file pattern, discussed below). At the appropriate levels in the pattern hierarchy, the virtual patterns are bound so that the error messages are tailored to the specific context. The user can augment these error messages by means of the msg text object or choose to ignore the exception and continue execution.

In order to differentiate between potential fatal static exceptions and to be backward compatible with earlier versions of exceptions in the BETA programming language, the notification pattern is defined as:

```
notification: exception(# do true−>continue; INNER #);
```

1.6.1.1 Examples Using Static Exceptions

In order to illustrate the use of static exceptions, let us look at a file exception example. Without using the exception handling facilities an attempt to open a non−existing file will produce the following error messages:

```
**** Exception processing
  Error in file 'in.bet' No such file
```

Now let us see what can be done by using exceptions.

The binding of noSpaceError shows that a message can be added to msg. Msg could also have been overwritten, by first clearing msg (msg.clear). The binding of noSuchFileError shows how to prevent the system from stopping the execution when the program attempts to open a non−existing file. Instead the user is prompted for another file name. In fact there exists a procedure pattern (exists) that tests for the existence of a file, but this has not been used in this example.

```
(#
   inFile: @file
     (# noSuchFileError:: (# do true−>continue; false−>OK #)#);
   OK: @ boolean;

do 'in.bet' −> inFile.name;
   true −> OK;
   openFile:
     (#
     do inFile.openRead;
        (if not OK then
            'File does not exist!' −>  screen.putline;
            'Type input file name: ' −>  screen.puttext;
            getLine−>inFile.name;
            true −> OK;
            restart openFile
```

```
        if)
     #);

   inFile.close;
#)
```

An attempt to open a non−existing file will produce the following error messages:

```
File does not exist!
Type input file name:
```

It gives the programmer the possibility to proceed with another file name.

1.6.2 Dynamic Exceptions

The exception pattern can be used for dynamic exceptions as well as for static exceptions. The way to raise a dynamic exception is to define an exception:

```
myExcetion: exception (# ... #)
```

and then throw it:

```
&myException[]−>throw
```

This will initiate a search through the call stack to find a handler for the exception.

To catch a dynamic exception, the known try−catch−finally triplet from fx java can be used. In BETA you use a try pattern with a handler inside it:

```
(#
do try
   (#
      handler::
        (#
        do when
           (#
              type:: myException;
              predicate:: myPredicate
           do ...
           #);
        #);
      finally::
        (#
        do ...
        #);
   do ...
   #);
#)
```

Of course there can be more than one when clause in a handler.

In the when part of a handler, there are four ways to return to the calling code: propagate, abort, retry and continue.

Propagate is the default action for a handler. It tells throw that it does not handle the exception and

throw sends the exception to the next handler on the call stack. If the programmer explicitly calls propagate, the same thing happens, except the code in the handler is executed too.

Abort aborts the current try blocks and executes all the finally part. If the exception was propagated through other try blocks, their finally parts are executed too.

Retry executes the finally part of the try block and starts it over.

Continue returns the control to the statement in the try do part which caused the exception.

You can use the predicate of the when part to further specify which exceptions are caugt in this when part.

1.6.2.1 Examples Using Dynamic Exceptios

Suppose we wish to cath a reference is none error, if assigning something to a NONE object:

```
(#
myPattern: object
  (#
     a: @integer
  enter a
  exit a
  #);

myObj: ^myPattern
do 10->myObj
#)
```

This program will give the following error message when run:

```
# Beta execution aborted: Reference is none.
# Look at 'testrefnone.dump'
# Generating dump file.  This may take a little while − please be patient
```

But if we use the framework presented above, we can catch the exception:

```
ORIGIN '~beta/basiclib/betaenv';
INCLUDE '~beta/basiclib/private/systemExceptionHandler';
INCLUDE '~beta/basiclib/systemExceptions';

-- program: descriptor --

(#
   myPattern: object
     (#
        a: @integer
     enter a
     do 'entering '->puttext; a->putint; putline;
     exit a
     #);

   myObj: ^myPattern

do runtimeExceptionHandler##->InstallSystemExceptionHandler;
   try
   (#
      name:: (# do 'mytry'->n[] #);
```

```
      handler::
        (#
        do when
           (# type:: refNoneException
           do &myPattern[]->myObj[];
              retry;
           #)
        #);
   do 10->myObj
   #)
#)
```

This will yield the following result:

```
***Throwing refNoneException#***
                         ***Retrying mytry***
entering 10
```

We can see, that the refNoneException is caught in the when part, myObj is initialized and the try block is retried and this time, there is no refNoneException.

1.7 Various Other Patterns

1.7.1 Control Patterns

Betaenv contains three predefined control patterns: forTo, cycle and loop. They are respectively defined as:

```
forTo: (* for inx in [min,max] do inner *)
  (# min,max,inx: @integer
  enter (min,max)
  do ... inner;
     ...
  #);

cycle: (* executes inner forever *)
  (# do l:(#... inner; ...#) #);

loop: (* control pattern for while- and repeat-loop *)
  (# while:< booleanValue(# do ... inner; ... #);
     until:< booleanValue(# do ... inner; ... #);
  do ... inner; ...
  #)
```

Recall that control patterns are procedure patterns that are to be used as superpattern:s. The first example illustrates the use of forTo in an inserted item:

```
do ...
   (3,17) -> forTo(# do inx*inx -> putint; newline #);
   ...
```

It will cause printing of the values 32, 42, ..., 172. The next example illustrates the use of cycle in the definition of another control pattern.

```
countCycle: cycle
```

```
  (* increments inx and executes inner forever *)
  (# inx: @integer
  do inx + 1 -> inx;
  inner
  #);
```

Finally, the following loop example reads a sequence of integers from standard input until either a non−positive integer is read or the sum of integers exceeds 1000:

```
loop(# while::< (# do getint->i; i>0->value #);
       until::< (# do sum>1000->value #);
       sum, i: @integer
    do i+sum->sum
    #)
```

1.7.2 Input/Output Patterns

1.7.2.1 Screen and keyboard

Standard input/output is available through dynamic references to objects that are instances of the pattern stream (see later). These streams are automatically opened. Abbreviations for the most often used input/output operations are defined (e.g. put for screen.put and get for keyboard.get).

```
(# j: @integer
do (for i: 5 repeat
        '? '-> puttext; getint -> j;
        j*j -> putint; newline
   for)
#)
```

1.7.2.2 Single character input − iget.bet

The iget.bet library implements immediate character input (i.e. non−buffered input). This is through the iget operation, which returns a single character. iget returns as soon as a character is typed.

1.7.3 Command Line Arguments

It is possible to let BETA programs access the command line arguments through the noOfArguments and arguments patterns. NoOfArguments returns the number of text atoms on the command line, including the program name. The text atoms are numbered from 1 to noOfArguments. The program name is obtained by 1 −> arguments, the first argument is obtained by 2 −> arguments, etc.

The following example displays the number of arguments of a command line followed by the arguments.

```
(#
do 'This program was called with ' -> puttext;
   noOfArguments -> putint; ' argument(s):' -> putline;
   (for i: noOfArguments repeat
        i -> arguments -> puttext; ' ' -> put;
   for);
   newline;
#)
```

1.7.4 Object Pool

The objectPool is for keeping track of unique instances of patterns. A call of the form

```
objectPool.get(# type::< T #) −> obj[]
```

will return an instance of T. The first call will create an instance of T. Subsequent calls will return this instance again. The objectPool is useful in systems where many fragments must refer to the same unique instance of a pattern T. ObjectPool also defines a scan operation which may be used for scanning the objects in the pool. For more operations, please consult the interface descriptions later.

1.7.5 Concurrency − SystemEnv

The current version of the Mjølner System includes an experimental implementation of concurrency. The complete environment for concurrency is defined in the systemEnv library, described in a later chapter. However, some concurrency facilities are necessarily defined in the betaenv library. Please refer to the later chapter on the systemEnv library for details.

1.7.6 External Language Interface

External and cStruct

Betaenv contains the shortInt, external and cStruct patterns for interfacing into facilities written in other languages, such as C and Pascal. Please refer to the later chapter on the external library for details.

1.8 Time and Date Library

The basic libraries also contains a library: timedate.bet, which defines a series of patterns for manipulating time and date informations. See the interface of timedate for more details.

2 The math Library

This library provides mathematical patterns: trigonometric, hyperbolic, exponential and logarithmic, floating point manipulation, and miscellaneous constants.

2.1 Using the math Fragment

A program using the math fragment will have the following structure:

```
INCLUDE '~beta/basiclib/math
--- program: descriptor ---
(# ...
   r: @real;
do ...
   1.234 -> sin -> r;
   ...
#)
```

3 The numberio Library

3.1 Based integers and reals

Numberio is a library for reading and writing numerals (integers, based integers, radix integers, and reals). The format of these numerals corresponds directly to the format of these numerals as defined by the BETA language (except for radix integers that are not supported by the BETA language). Numberio also contains a general getNumber operation, that is able to read any numeral, and return the proper value read.

Grammar for getNumber operation

The following grammar defines the exact syntax of the numerals:

```
N ::= D+                        int             314
   |  D+ '.' D+                 real            3.14
   |  D+ '.' D+ 'E' E           real            3.14E8
                                 real            3.14E+8
                                 real            3.14E−8
   |  D+  'E' E                 real            3E8
                                 real            3E+8
                                 real            3E−8
   |  D+ 'X' (D|L)+             based           2X0101
                                 based           8x0845
                                 based           16xAF12
   |  (D|L)+                    radix           AF12
D ::= '0' | ... | '9'
L ::= 'A' | ... | 'Z'
E ::= D+
   |  '+' D+
   |  '−' D+
```

Integer examples:

```
10   0   123
```

A based integer has the form <base>X<number>. Examples are:

```
2X101   base=2,   number= 4*1 + 2*0 + 1*1 = 5
8X12    base=8,    number= 8*1 + 1*2 = 10
16x2A1  base=16, number= 256*2 + 16*10 + 1*1 = 673
0x2A1   base=16, i.e. base=0 is interpreted as base=16
```

Examples of reals are:

```
3.14   3.14E−8   3E+8
```

All letters may be in lower or upper case.

The various read and write operations contain several facilities for controlling the read and write. Please consult the interface description later for details.

3.2 Using the numberio Fragment

A program using the numberio fragment will have the following structure:

```
INCLUDE '~beta/basiclib/numberio
--- program: descriptor ---
(# ...
   r: @real;
do ...
   getReal -> r;
   ...
#)
```

3.2.1 Reading numbers from standard input

The following example illustrates using numberio to read general numerics, as well as based, integer and reals from keyboard.

Program 2: NumberioDemo.bet

```
ORIGIN '~beta/basiclib/numberio';
--- program: descriptor ---
(#
do L: (#
      do 'Enter a general number: '->puttext;
         getNumber
         (# valueError::< (# do msg[]->screen.putline; true->continue #);
            syntaxError::< (# do msg[]->screen.putline; get; true->continue #);
            baseError::< (# do msg[]->screen.putline; true->continue #);
            EOSerror:: (# do screen.newline; leave L #);
            integerValue::<
              (# do '\tInteger='->puttext; value->putInt #);
            basedValue::<
              (#
              do '\tBase='->puttext; base->putint;
                 ' Value='->puttext; value->putInt;
                 ' BasedValue='->puttext; (base,value)->putBased;
              #);
            realValue::<
              (#
              do '\tReal='->puttext;
                 value->screen.putReal(# format::< (# do exp->style; true->upcase #)
              #) #);
         do newline;
         #);
         'Enter a based number: '->puttext;
         getBased
         (# valueError::< (# do msg[]->screen.putline; true->continue #);
            syntaxError::< (# do msg[]->screen.putline; true->continue #);
            baseError::< (# do msg[]->screen.putline; true->continue #);
            EOSerror:: (# do screen.newline; leave L #);
         do '\tBase='->puttext; b->putint;
            ' Value='->puttext; i->putInt;
            ' BasedValue='->puttext; (b,i)->putBased;
         #); newline;
         'Enter a real number: '->puttext;
         getReal
         (# valueError::< (# do msg[]->screen.putline; true->continue #);
            syntaxError::< (# do msg[]->screen.putline; true->continue #);
```

```
            baseError::< (# do msg[]−>screen.putline; true−>continue #);
            EOSerror:: (# do screen.newline; leave L #);
         do '\tReal='−>puttext
         #)−>screen.putreal(# format::< (# do exp−>style; true−>upcase #) #); newline;
         'Enter an integer number: '−>puttext;
         getInteger
         (# valueError::< (# do msg[]−>screen.putline; true−>continue #);
            syntaxError::< (# do msg[]−>screen.putline; true−>continue #);
            baseError::< (# do msg[]−>screen.putline; true−>continue #);
            EOSerror:: (# do screen.newline; leave L #);
         do '\tInteger='−>puttext
         #)−>putInt; newline;
         restart L
      #)
#)
```

4 The formatio Library

4.1 Formatted input/output

This library provides facilities for formatted input and output (similar to the scanf and printf functions in C). These facilities are implemented in the form of the getFormat and putFormat operations that are added as stream attributes, making formatted input and output available for any stream.

4.2 Format string

Both getFormat and putFormat take a text string as argument. This text string must contain a format specification of the input to be read from (respectively output to) the stream. The format string may be any string, possibly with one or more embedded markers. The markers specify the variable parts of the expected input (respectively output), such as integer values. The markers are indicated in the string by a leading '%'. Following the '%' is the specification of the marker type.

getFormat accepts the following marker syntax:

```
%[width][.[precision]]dioxXrRbBfeEgGcsn%
```

putFormat accepts the following marker syntax:

```
%-+ [[0]width][.[[0]precision]]dioxXrRbBfeEgGcsn%
```

As it can be seen, the marker syntax is very similar.

4.3 Marker type

Corresponding to every marker type (given by the dioxXrRbBfeEgGcsn part of the marker syntax), getFormat and putFormat defines an attribute (with the same name) as the marker symbol an attribute with the name d, corresponding to the d marker type. Due to BETA not being case−sensitive in identifiers, the attributes corresponding to the upper−case marker types are called e.g. uG for upper g.

The functionality of formatted input and output can now easiest be described by showing the following example:

```
(# t, s: @text;
   theName: ^text; theValue: integer;
do 'name: temperature value: 72 name: speed value: 35' -> t;
   0->t.pos;
   'name: %s value: %d'
   t.getFormat(# do s->theName[];  d->theValue #);
   'The name is %s and the value is %d\n'
   s.putFormat(# do theName[]->s; theValue->d #);
   'name: %s value: %d'
   t.getFormat(# do s->theName[];  d->theValue #);
   'The name is: %s and the value is: %d\n'
   s.putFormat(# do theName[]->s; theValue->d #)
#)
```

At the end of this program, s will contain the following text:

```
The name is: temperature and the value is: 72
The name is: speed and the value is: 35
```

The getFormat and putFormat operations raise exceptions if the format specifications are not satisfied.

4.4 Using the formatio Fragment

A program using the formatio fragment will have the following structure:

```
INCLUDE '~beta/basiclib/formatio
--- program: descriptor ---
(# ...
   r: @real;
do ...
   'real value: %e' -> putformat(# do r->e #);
   ..
#)
```

4.4.1 Illustrating putFormat

Program 3: putformatDemo.bet

```
ORIGIN '~beta/basiclib/formatio';
--- program: descriptor ---
(#
do 'string=%s, integer=%i, real=%f\n\n'
     -> putFormat(# do 'abc'->s; -30->i; 3.14->f #);

   '(1234567){12345678}[123456789]\n' -> putFormat;
   '(%7i){%8i}[%9i]\n\n' -> putFormat(# do 27->i; 30->i; -45->i #);

   '"%%s,%%.5s,%%s,%%2.3s#" = %s,%.5s,%s,%2.3s#\n\n'
     -> putFormat(# do '27'->s; 'Hello world'->s; '-45'->s; none->s #);

   '  123456789   1234567890123456       12345678\n' -> putFormat;
   'x=%9.3f,y=%16.e,z=%.8f\n\n'
     -> putFormat(# do 27.5->f; 30.5->e; -45.5->f #);

   '  1234567890   1234567890   12345678\n' -> putFormat;
   'x=%*.*f,y=%*.2e,z=%8.*f\n\n'
     -> putFormat(# do 10->width; 3->precision; 27.5->f; 30.5->e; -45.5->f #);

#)
```

4.4.2 Illustrating getFormat

Program 4: getformatDemo.bet

```
ORIGIN '~beta/basiclib/formatio';
--- program: descriptor ---
(# i1, i2, i3, i4, i5: @integer;
   s1, s2: ^text;
   r1, r2, r3: @real;
```

```
   chr: @char;
   t, fmt: ^text;
do '123,%456, 789 JorgenLKnudsen 1.2,2+2=43.45, 67.89 abc, 101, 0x101, 0101, 101xxx'->t|
   0->t.pos;

   'input:\t"%s"\n\n' -> putFormat(# do t[]->s #); newline;

   '%i,%%%i, %i' -> fmt[] -> t.getFormat(# do i->i1; i->i2; i->i3 #);
   'format:\t%s\nread:\t%i, %i, %i\n\n'
     -> putFormat(# do fmt[]->s; i1->i; i2->i; i3->i #);

   '%6s%c%s' -> fmt[] -> t.getFormat(# do s->s1[]; c->chr; s->s2[] #);
   'format:\t"%s"\nread:\t"%s", "%c", "%s"\n\n'
     -> putFormat(# do fmt[]->s; s1[]->s; chr->c; s2[]->s #);

   '%f,2+2=4%e, %f' -> fmt[] -> t.getFormat(# do f->r1; e->r2; f->r3 #);
   'format:\t"%s"\nread:\t%f, %f, %f\n\n'
     -> putFormat(# do fmt[]->s; r1->f; r2->f; r3->f #);

   '%x, %o, %i, %i, %i%s' -> fmt[]
     -> t.getFormat(# do x->i1; o->i2; i->i3; i->i4; i->i5; s->s1[] #);
   'format:\t"%s"\nread:\t%i, %i, %i, %i, %i, "%s"\n'
     -> putFormat(# do fmt[]->s; i1->i; i2->i; i3->i; i4->i; i5->i; s1[]->s #);

   '123,%456, 124 %JorgenLKnudsen 1.2,2+2=43.45, 67.89 abc, 111, 0x111, 0111, 111xxx'->t
   0->t.pos;
   '\n\ninput:\t"%s"\n\n' -> putFormat(# do t[]->s #); newline;

   '%i,%%%i, %i %6s%c%s%f' -> fmt[] -> t.getFormat(# do i->i1; i; i->i2; s; c; s; f->r1;
   'format:\t%s\nread:\t%i, %i, %f\n\n'
     -> putFormat(# do fmt[]->s; i1->i; i2->i; r1->f; #);

#)
```

5 The random Library

This library provides elaborate random number generation facilities. The random library contains random generators with different statistical distributions, such as uniform, normal, binomial and poison distributions.

The library offers facilities for specifying the seeds of the generator by the setsd operation.

The library offers facilities for maintaining up to 32 parallel random generators.

For more details, see the interface descriptions.

5.1 simple random generators: ignlgi and ranf

The simple random generator most often used (namely the uniform integer random generator) is in this library the ignlgi generator which returns a uniformly distributed integer in the range [1, 2147483562]. It is also available in the ignuin variant which returns a uniformly distributed integer in a range specified by the user.

Another simple and often used random generator is the uniform real random generator which in this library is available as the ranf generator which returns a uniformly distributed real in the range]0, 1[(i.e. 0 and 1 are never returned). The genunf variant returns a uniformly distributed real in a range specified by the user.

5.2 Using the random Fragment

A program using the random fragment will have the following structure:

```
INCLUDE '~beta/basiclib/random
--- program: descriptor ---
(# ...
   i: @integer;
do ...
   (123, 456) -> setsd;
   ...
   (1,100) -> inguin -> i;
   ...
#)
```

5.2.1 Illustrating Integer Random Generator

Program 5: tstign.bet

```
ORIGIN '~beta/basiclib/random';
--- program: descriptor ---
(# testIgn:
     (# iarray: [mxint]@integer;
        itmp: @integer;
        lo, hi, i, up: @integer
     do (for i:nrep repeat
              (1,mxint)->ignuin->itmp;
              itmp->screen.putint(# do 7->width #);
              (if (i mod 10) = 0 then newline if);
```

```
            iarray[itmp]+1->iarray[itmp];
        for);
        newline;
        '         Counts of Integers Generated:'->putline;
        (* Print 10 to a line using 7 characters for each field.         *)
        mxint->up; 1->lo;
        l: (if lo<=up then
               (lo+9,up)->min->hi;
               lo->i;
               ll: (if i<=hi then
                       iarray[i]->screen.putint(# do 7->width #);
                       i+1->i;
                       restart ll
                   if);
               screen.newline;
               lo+10->lo;
               restart l;
           if)
     #);

   mxint,nrep: @integer;

do ' Tests uniform random integer generator.'->putline; newline;

   ' Enter two seeds to initialize rn generator: '->puttext;
   (getint,getint)->setall;

   ' Enter maximum uniform integer: '->puttext;
   getint->mxint;

   ' Enter number of randoms to generate: '->puttext;
   getint->nrep;

   testIgn;
#)
```

6 The regexp Library

The regexp library augments the text pattern in betaenv with four new attributes:

```
regexp_match
regexp_search
regexp_replace
regexp_replace_literally
```

All four operations give facilities for working with regular expressions in text strings.

6.1 Regular expression

A regular expression (regexp, for short) is a pattern that denotes a set of strings, possibly an infinite set. Searching for matches for a regexp is a very powerful operation that editors on Unix systems have traditionally offered.

Regular expressions are used to locate occurrences of substrings in text, where the substring is expected to be of a certain structure. E.g. the regexp '[Ww]ord' will match the substring 'word' or 'Word'. The four operations offer slightly different possibilities, described below:

6.2 regexp_match

Takes a regexp as enter parameter (in the form of a reference to a text, containing the regexp. Matches THIS(text) against the regexp. INNER is executed if THIS(text) matches the regexp, and the virtual notification noMatch is invoked otherwise. Returns true if a match is found, false otherwise. The regexp must be found starting at the current position of THIS(text).

6.3 regexp_search

Like regexp_match, except that the match is allowed to be found anywhere between the current position and the end of THIS(text).

6.4 regexp_replace

Like regexp_search, except that it takes a second enter parameter, replacement_string. Regexp_replace searches for the regexp, and replaces the matched substring of THIS(text) with the replacement string. The replacement string may contain \0, \1, ..., \9, representing the substring matched by the i'th parenthesis in the regexp. \0 represents the entire substring matched. INNER is executed after the replace has taken place.

6.5 regexp_replace_literally

Like regexp_replace, except that the replacement string is taken literally (i.e. \0, \1, etc. are not representing any matched substrings)

All four regexp operations defines the same local attributes:

- start: start position for search in THIS(text). Default: pos
- limit: end position for search in THIS(text). Default: length
- posToMatchEnd: if true, move THIS(text).pos to the end of the matched substring. Default: false
- regs: structure for getting access to the matched substring. noMatch: invoked if no matches are found.
- regexpError: is invoked if error occurs in the regexp implementation.
- value: true, if any match is found.

Regexp_string are compiled implicitly by the operations to ensure efficient matching in all operations. If the same regexp_string is to be used several times, the repetition of this compilation can be avoided by generating an instance of the operation (say regexp_search), and then use that instance repeatedly (as is illustrated in the following example):

```
(# regexp_s: @mytext.regexp_search(# ... #);
do ...
regexp_string[]−>regexp_s;(* first search, implicit regexp
compilation *)
...
regexp_s; (* repeating the search with the same regexp.
No regexp compilation *)
...
regexp_s; (* repeating the search with the same regexp.
No regexp compilation *)
...
#)
```

6.6 Syntax of Regular Expressions

The syntax of regular expressions in this library follows the syntax of Emacs regular expressions. Some of the documentation below is from the interactive Emacs Info system.

Regular expressions have a syntax in which a few characters are special constructs and the rest are ordinary. An ordinary character is a simple regular expression which matches that character and nothing else. The special characters are '$', '^', '.', '*', '+', '?', '[', ']' and '\'; no new special characters will be defined. Any other character appearing in a regular expression is ordinary, unless a '\' precedes it.

For example, 'f' is not a special character, so it is ordinary, and therefore 'f' is a regular expression that matches the string 'f' and no other string. (It does not match the string 'ff'.) Likewise, 'o' is a regular expression that matches only 'o'.

Any two regular expressions A and B can be concatenated. The result is a regular expression which matches a string if A matches some amount of the beginning of that string and B matches the rest of the string.

As a simple example, we can concatenate the regular expressions 'f' and 'o' to get the regular expression 'fo', which matches only the string 'fo'. Still trivial. To do something nontrivial, you need to use one of the special characters. Here is a list of them.

6.7 List of special characters

- '.' is a special character that matches any single character except a newline. Using concatenation, we can make regular expressions like 'a.b' which matches any three−character string which begins with 'a' and ends with 'b'.

- '*' is not a construct by itself; it is a suffix, which means the preceding regular expression is to be repeated as many times as possible. In 'fo*', the '*' applies to the 'o', so 'fo*' matches one 'f' followed by any number of 'o's. The case of zero 'o's is allowed: 'fo*' does match 'f'.
- '*' always applies to the smallest possible preceding expression. Thus, 'fo*' has a repeating 'o', not a repeating 'fo'.
- The matcher processes a '*' construct by matching, immediately, as many repetitions as can be found. Then it continues with the rest of the pattern. If that fails, backtracking occurs, discarding some of the matches of the '*'−modified construct in case that makes it possible to match the rest of the pattern. For example, matching 'ca*ar' against the string 'caaar', the 'a*' first tries to match all three 'a's; but the rest of the pattern is 'ar' and there is only 'r' left to match, so this try fails. The next alternative is for 'a*' to match only two 'a's. With this choice, the rest of the regexp matches successfully.
- '+' Is a suffix character similar to '*' except that it requires that the preceding expression be matched at least once. So, for example, 'ca+r' will match the strings 'car' and 'caaaar' but not the string 'cr', whereas 'ca*r' would match all three strings.
- '?' Is a suffix character similar to '*' except that it can match the preceding expression either once or not at all. For example, 'ca?r' will match 'car' or 'cr'; nothing else.
- '[...]' '[' begins a character set, which is terminated by a ']'. In the simplest case, the characters between the two form the set. Thus, '[ad]' matches either one 'a' or one 'd', and '[ad]*' matches any string composed of just 'a's and 'd's (including the empty string), from which it follows that 'c[ad]*r' matches 'cr', 'car', 'cdr', 'caddaar', etc.
- Character ranges can also be included in a character set, by writing two characters with a '−' between them. Thus, '[a−z]' matches any lower−case letter. Ranges may be intermixed freely with individual characters, as in '[a−z$%.]', which matches any lower case letter or '$', '%' or period.
- Note that the usual special characters are not special any more inside a character set. A completely different set of special characters exists inside character sets: ']', '−' and '^'.
- To include a ']' in a character set, you must make it the first character. For example, '[]a]' matches ']' or 'a'. To include a '−', write '−−', which is a range containing only '−'. To include '^', make it other than the first character in the set.
- '[^...]' '[^' begins a complement character set, which matches any character except the ones specified. Thus, '[^a−z0−9A−Z]' matches all characters except letters and digits.
- '^' is not special in a character set unless it is the first character. The character following the '^' is treated as if it were first ('−' and ']' are not special there).
- Note that a complement character set can match a newline, unless newline is mentioned as one of the characters not to match.
- '^' is a special character that matches the empty string, but only if at the beginning of a line in the text being matched. Otherwise it fails to match anything. Thus, '^foo' matches a 'foo' which occurs at the beginning of a line.
- '$' is similar to '^' but matches only at the end of a line. Thus, 'xx*$' matches a string of one 'x' or more at the end of a line.
- '\' has two functions: it quotes the special characters (including '\'), and it introduces additional special constructs.
- Because '\' quotes special characters, '\$' is a regular expression which matches only '$', and '\[' is a regular expression which matches only '[', and so on.
- Note, that '\' is also a special character in BETA literals. This implies, that in order to specify a '\' regexp special character in a BETA string literal, you have to type it twice, e.g. 'US\\$'.

Note: for historical compatibility, special characters are treated as ordinary ones if they are in contexts where their special meanings make no sense. For example, '*foo' treats '*' as ordinary since there is no preceding expression on which the '*' can act. It is poor practice to depend on this behavior; better to quote the special character anyway, regardless of where is appears.

For the most part, '\' followed by any character matches only that character. However, there are several exceptions: characters which, when preceded by '\', are special constructs. Such

characters are always ordinary when encountered on their own. Here is a table of '\' constructs.

6.8 Table of '\' constructs

- '\|' specifies an alternative. Two regular expressions A and B with '\|' in between form an expression that matches anything that either A or B will match.
- Thus, 'foo\|bar' matches either 'foo' or 'bar' but no other string.
- '\|' applies to the largest possible surrounding expressions. Only a surrounding '\(... \)' grouping can limit the grouping power of '\|'.
- Full backtracking capability exists to handle multiple uses of '\|'.
- '\(...\)' is a grouping construct that serves three purposes:
    - ♦ To enclose a set of '\|' alternatives for other operations. Thus, '\(foo\|bar\)x' matches either 'foox' or 'barx'.
    - ♦ To enclose a complicated expression for the postfix '*' to operate on. Thus, 'ba\(na\)*' matches 'bananana', etc., with any (zero or more) number of 'na' strings.
    - ♦ To mark a matched substring for future reference.
- This last application is not a consequence of the idea of a parenthetical grouping; it is a separate feature which happens to be assigned as a second meaning to the same '\(... \)' construct because there is no conflict in practice between the two meanings. Here is an explanation of this feature:
- '\DIGIT' after the end of a '\(... \)' construct, the matcher remembers the beginning and end of the text matched by that construct. Then, later on in the regular expression, you can use '\' followed by DIGIT to mean 'match the same text matched the DIGIT'th time by the '\(... \)' construct.'
- The strings matching the first nine '\(... \)' constructs appearing in a regular expression are assigned numbers 1 through 9 in order that the open−parentheses appear in the regular expression. '\1' through '\9' may be used to refer to the text matched by the corresponding '\(... \)' construct.
- For example, '\(.*\)\1' matches any newline−free string that is composed of two identical halves. The '\(.*\)' matches the first half, which may be anything, but the '\1' that follows must match the same exact text.
- '\`' matches the empty string, provided it is at the beginning of the buffer.
- '\'' matches the empty string, provided it is at the end of the buffer.
- '\b' matches the empty string, provided it is at the beginning or end of a word. Thus, '\bfoo\b' matches any occurrence of 'foo' as a separate word. '\bballs?\b' matches 'ball' or 'balls' as a separate word.
- '\B' matches the empty string, provided it is not at the beginning or end of a word.
- '\<' matches the empty string, provided it is at the beginning of a word.
- '\>' matches the empty string, provided it is at the end of a word.
- '\w' matches any word−constituent character. The editor syntax table determines which characters these are.
- '\W' matches any character that is not a word−constituent.
- '\sCODE' matches any character whose syntax is CODE. CODE is a character which represents a syntax code: thus, 'w' for word constituent, '−' for whitespace, '(' for open−parenthesis, etc.
- '\SCODE' matches any character whose syntax is not CODE.

6.9 A complicated regexp

Here is a complicated regexp, used to recognize the end of a sentence together with any whitespace that follows. It is given in BETA text string syntax to enable you to distinguish the spaces from the tab characters. In BETA text string syntax, the string constant begins and ends with a double−quote. '"' stands for a double−quote as part of the regexp, '\\' for a backslash as part of the regexp, '\t' for a tab and '\n' for a newline.

```
"[.?!][]""')]*\\($\\|\t\\| \\)[ \t\n]*"
```

This contains four parts in succession: a character set matching period, '?' or '!'; a character set matching close−brackets, quotes or parentheses, repeated any number of times; an alternative in backslash−parentheses that matches end−of−line, a tab or two spaces; and a character set matching whitespace characters, repeated any number of times.

6.10 Regular Expression Registers

The special regexp parenthesis (i.e. \(... \)) are used for three different purposes:

- to make the regexp operators work on entire sub−regexp: e.g.
- 123*4 matches 124, 1234, 12334, 123334, etc., whereas 1\(23\)*4 matches 124, 1234, 123234, 12323234, etc. (and not 12334, 123334, etc.).
- to delimit alternatives: e.g. 1\(2\|3\)4 matches 124 and 134.
- to mark a matched substring for future reference. In regexg_replace, \0, \1, \2, ..., \9 substrings will be replaced with the matched substring. We will here give a short introduction to accessing these registers from a BETA program.

We would like to give a little more details on the (3) purpose. Let us assume, that we invoke:

```
'sum: \([0−9]+\)\.\([0−9]\) US$'
  −> t.regexp_search
        (#
        do 'The amount is: '−>puttext;
           (1−>regs.start+1,1−>regs.end)−>t.sub−>puttext;
           ' dollars and '−>puttext;
           (2−>regs.start+1,2−>regs.end)−>t.sub−>puttext;
           ' cents'−>putline
        #)
```

The intent is to find some amount i the t text variable. The amount should be found in a context of the form: 'sum: XXX.YYY US$', where XXX and YYY may be any number. If found, the amount will be written on standard screen in the form: 'The amount is: XXX dollars and YYY cents'.

Note the use of i−>regs.start and i−>regs.end, which returns the character positions of the i'th matched substring (in this case the 1'st and 2'nd substring, containing the amount of dollars and the amount of cents, respectively). As it can be seen, these constructs can be used within the action−parts of all regexp operations to gain access to the matched substrings (note that the 0'th substring is the entire string matched).

6.11 Known bugs or Inconveniences

- \i (i= 0, 1, ...,9) matches the empty string if it is positioned before the i'th parenthesis. According to the specification, \i should match 'i' if positioned before the i'th parenthesis. E.g. \1 matches the empty string in \1\(2\)3.
- There is a slight error in the current implementation of registers, when the parenthesis bounds a repetitive regexp, i.e. a '*' or '+' regexp. In this case, the register will not contain the correct character positions, but instead refer to the empty string, immediately after the matched substring. Use an extra pair of parenthesis to get the correct substring (e.g. use 1\(\(2\)*\)3 instead of 1\(2\)*3 to refer to the matched '2's − the matched '2's can be referred to through register 1.

6.12 Using the regexp Fragment

A program using the regexp fragment will have the following structure:

```
INCLUDE '~beta/basiclib/regexp
--- program: descriptor ---
(# ...
t: ^text
do ...
   'Hello world' -> t[]; t.reset;
   ('\\<w.*\\>', '\\0 champion') -> t.regexp_replace;
   t[] -> putline; (* prints 'Hello world champion' *)
   ...
#)
```

6.13 Regexp Demos

6.13.1 Search for regexp

The following example is a simple program that enters a regexp from the keyboard, followed by another test string, trying to find the regexp in the test string. If found, the registers are printed on the screen, and the corresponding matched substrings:

Program 6: searchDemo.bet

```
ORIGIN '~beta/basiclib/regexp';
--- program: descriptor ---
(#
do loop:
     (# regex: ^text;
     do 'Search for: ' -> puttext; getLine -> regex[];
        (if regex.length = 0 then leave loop if);
        loop:
          (# string: ^text;
          do 'Search for: '->puttext; regex[]->puttext;
             ' in: ' -> puttext; getline -> string[];
             (if string.length = 0 then leave loop if);
             0 -> string.pos;
             regex[] -> string.regexp_search
             (# noMatch::< (# do '\tNo match' -> putline #)
             do (for i: regexp_numberOfRegisters repeat
                     (if (i-1->regs.start)>=0 then
                         '\tregister '->puttext; i-1 -> putint; ': '->puttext;
                         i-1->regs.start->putint;
                         ' to '->puttext;
                         i-1->regs.end->putint;
                         ': '->puttext;
                         ((i-1->regs.start)+1, i-1->regs.end)->string.sub->putline
                     if)
                for);
             #);
             restart loop
          #);
        restart loop
     #)
#)
```

Output from the demo might be:

```
Search for: 1\(234\)5
Search for: 1\(234\)5 in: dfjdfhdf1235jfdkj12345ghdf
register 0: 17 to 22: 12345
register 1: 18 to 21: 234
Search for: 1\(234\)5 in: 12345
register 0: 0 to 5: 12345
register 1: 1 to 4: 234
Search for: 1\(234\)5 in: 123
No match
Search for: 1\(234\)5 in:

Search for: \(22\).*\1
Search for: \(22\).*\1 in: xxx22yyy2zzz
No match
Search for: \(22\).*\1 in: xxx22yyy22zzz
register 0: 3 to 10: 22yyy22
register 1: 3 to 5: 22
```

6.13.2 Replace with a regexp

The following example is nearly identical to the previous, except that it allows for the specification of a replacement string:

Program 7: replaceDemo.bet

```
ORIGIN '~beta/basiclib/regexp';
--- program: descriptor ---
(#
do loop:
     (# regex: ^text;
     do 'Replace: ' -> puttext; getLine -> regex[];
        (if regex.length = 0 then leave loop if);
        loop:
          (# string: ^text;
          do 'Replace: '->puttext; regex[]->puttext;
             ' in: ' -> puttext; getline -> string[];
             (if string.length = 0 then leave loop if);
             loop:
               (# t, replacement: ^text
               do string.copy -> t[]; 0 -> t.pos;
                  'Replace: '->puttext; regex[]->puttext;
                  ' in: ' -> puttext; string[]->puttext;
                  ' with: '->puttext; getline -> replacement[];
                  (if replacement.length = 0 then leave loop if);
                  (regex[], replacement[]) -> t.regexp_replace
                  (# noMatch::< (# do '\tNo match'->screen.putline #) #);
                  'Replace: '->puttext; regex[]->puttext;
                  ' in: ' -> puttext; string[]->puttext;
                  ' with: '->puttext; replacement[]->puttext;
                  ' gives: '->puttext; t[] -> putline;
                  restart loop
               #);
             restart loop
          #);
        restart loop
     #)
#)
```

Output from the demo might be:

```
Replace: [Ww]ord
Replace: [Ww]ord in: This Word is in capital
Replace: [Ww]ord in: This Word is in capital with: sentence
Replace: [Ww]ord in: This Word is in capital with: sentence gives: This sentence is in c
```

7 The file Library

The file library implements the diskEntry, file, fileRep, patterns, all used to model external storage media such as disk files.

7.1 File and diskEntry

The file library implements the file pattern that is used to represent external storage media such as disk files. File is an abstract subpattern of stream, specifying the machine independent attributes of such media. Specific subpatterns: exists for the different machine types such as UNIX and Macintosh (unixFile, respectively macFile, see later).

The attributes of file are divided into two parts: the disk entry attributes and the contents related attributes. The disk entry attributes are located in the entry attribute of file, whereas the contents related attributes are ordinary attributes of file.

7.2 Disk entry attributes

The disk entry attributes are defined in the diskEntry pattern: path, size, readable, writable, isFile, isDirectory, exists, modtime, touch, and rename. Also related to the disk entry are the following exceptions: diskEntryExistsException, diskEntryModtimeException, diskEntryTouchException, and diskEntryRenameException. These attributes are accessed through the entry attribute of a file, e.g. if aFile is a file object, then aFile.entry.modtime will return the last modification time of the associated disk file.

7.3 File attributes

The contents related attributes are: entry, name, touch, delete, openRead, openWrite, openAppend, openReadWrite, openReadAppend, flush, and close. Note that file also inherits all the stream attributes (further binding several of them). File also defines the following exceptions: openException, accessError, writeError, readError, noSuchFileError, fileExistsError, noSpaceError, and otherError.

7.4 The FileRep Library

The fileRep pattern is consisting of a repetition of integers and operations which makes it possible to save and restore this repetition in one chunk from a file. When saving, the repetition elements [1:top−1] are saved (top is an attribute of fileRep) and when restoring, top will become equal to range of the stored repetition. The repetition is in the R attribute, and save and restore is used for saving and restoring the repetition onto some file.

7.5 The BinFile Library

This fragment declare attributes for direct reading/writing of various data sizes to a file. The data is written out exactly as is:

```
64 −> aBinFile.putLong
```

will write the number 0x00000020 to aBinFile, whereas

```
64 -> aBinFile.putInt
```

will write the two characters '6' (ascii 54) and '4' (ascii 52) to the file.

The operations putBytes and getBytes allow an arbitrary sequence of bytes to be written/read to/from a file. E.g.

```
buffer: [1000]@char;
   putB: @aFile.putBytes;
do ...
   (@@buffer[1],500) -> putB;
```

This will write the first 500 characters from the buffer repetition to the file. NOTICE, that you must have a static instance of putBytes/getBytes when using them, since they require an address argument. If dynamic instances are used, a garbage− collection may be triggered, and the address argument would be illegal.

These operations are declared in FileLib, e.g. they become usable for any file, by just including this fragment file. However, on some platforms, the "binary" virtual of File MUST be further bound to TrueObject for these operations to work. The binfile pattern below adds this further binding.

You should remember this further binding if you are using these operations on a file, that is not a binfile.

Exceptions: If any of the put−operations fail, they raise the WriteError file−exception. If any of the get−operations fail, they raise the ReadError file−exception.

7.5.1 Using the file Fragment

A program using the file fragment will have the following structure:

```
INCLUDE '~beta/basiclib/file
--- program: descriptor ---
(# ...
   f: @file;
do ...
   'Hello world' -> f.putline;
...
#)
```

7.5.2 Using diskEntry

An example showing the use of the path attribute of DiskEntry: The path specified on the command line is decomposed, and the various parts of it are printed:

Program 8: decompose.bet

```
ORIGIN '~beta/basiclib/file';

--- program: descriptor ---
(* An example showing the use of the 'path' attribute of DiskEntry: The path
 * specified on the command line is decomposed, and the various parts of it are
```

```
 * printed.
 *)

(# e: @diskentry;
do (if noOfArguments <> 2 then
       'Usage: ' −> puttext; 1−>arguments−>puttext; ' path' −> putline;
       stop;
   if);
   2 −> arguments −> e.path;
   'The path \''−>puttext;
   e.path −> puttext;
   '\' is composed like this:'−>putline;
   'Head of path:      '−>puttext; e.path.head −> putline;
   'Name:              '−>puttext; e.path.name −> putline;
   'Prefix of name:    '−>puttext; e.path.name.prefix −> putline;
   'Suffix of name:    '−>puttext; e.path.name.suffix −> putline;
   'Extension of name: '−>puttext; e.path.name.extension −> putline;
   (if e.exists then '(and the entry exists on disk)' −> putline;
    else '(there is no such entry on disk)' −> putline;
   if);
#)
```

7.6 Using File

Program showing the use of exceptions in files: if the input file specified on the command line cannot be opened, the exception noSuchFileError is raised. In this example, it is further bound specifying that the program should continue after the exception is raised. Then a new file name is prompted for. Instead of this approach, of course, the attributes exists of diskEntry could have been used.

If the output file could not be opened, the exception noSpaceError is raised, and this exception is further bound to print a message:

Program 9: fileerror.bet

```
ORIGIN '~beta/basiclib/file';
--- program: descriptor ---

(* Program showing the use of exceptions in files: if the input file
 * specified on the command line cannot be opened, the exception
 * NoSuchFileError is raised. In this example, it is further bound
 * specifying that the program should continue after the exception is raised.
 * Then a new file name is prompted for.
 * Instead of this approach, of course, the attributes 'exists' of DiskEntry
 * could have been used.
 * If the output file could not be opened, the exception NoSpaceError is raised,
 * and this exception is further bound to print a message.
 *)

(# outFile: @file
     (# NoSpaceError ::<
          (# do 'It is time to delete garbage!' −> msg.putLine #);
     #);
   inFile: @ file
     (# NoSuchFileError ::< (# do true −> continue; false −> OK #);
     #);
   OK: @ boolean;

do (if noOfArguments <> 2 then
```

```
       'Usage: ' -> puttext; 1->arguments->puttext; ' out-file' -> putline;
       stop;
   if);
   2 -> arguments -> outFile.name;
   outFile.OpenWrite;

   'nofile.bet' -> inFile.name;
   true -> OK;
   openFile:
     (#
     do inFile.OpenRead;
        (if not OK then
            'File does not exist: ' ->  screen.puttext;
            infile.name -> putline;
            'Type input file name: ' ->  screen.putText;
            inFile.name.read;
            true -> OK;
            restart openFile
        if);
     #);

   readFile:
     (#
     do (if not inFile.eos then
            inFile.getatom -> outFile.putText;
            outFile.newLine;
            restart readFile
            else leave readFile
        if)
     #);
   inFile.close;
   outFile.close;
   'Tokens from \''->puttext; infile.name -> puttext;
   '\' copied to \''->puttext; outfile.name -> puttext;
   '\''->putline;
#)
```

8 The directory Library

The directory library defines the directory pattern that is the interface into file directories in a hierarchical file system. Directory is an abstract pattern, specifying the machine independent attributes of such directories. Specific subpatterns: exists for the different machine types such as UNIX and Macintosh (unixDirectory, respectively macDirectory, see later).

Similar to file, the attributes of directory are divided into two parts: the disk entry attributes and the directory related attributes. The disk entry attributes are located in the entry attribute of directory, whereas the directory related attributes are ordinary attributes of directory .

The disk entry attributes of directory are the same as for file, and will therefore not be discussed here (see previous chapter).

8.1 Directory attributes

The directory related attributes are: entry, name, touch, delete, createFile, deleteFile, createDir, deleteDir, noOfEntries, empty, findEntry, and scanEntries. Directory also defines the following exceptions: entryExistException, dirScanException, dirSearchException, noSuchException, and notFoundException.

8.2 Using the directory Fragment

A program using the directory fragment will have the following structure:

```
INCLUDE '~beta/basiclib/directory'
--- program: descriptor ---
(# ...
   dir: @directory;
do ...
   '/user/local/lib/beta/basiclib/' -> dir.name;
   dir.scanEntries(# do found.name -> putline #);
...
#)
```

8.2.1 Listing a directory

Program showing a simple use of directory: The directory with the path given as argument is scanned, and the names of all the entries are printed with an indication of what type of entry it is. This is done using the select pattern of scanentries. This is a more efficient strategy than using found.entry.isFile etc., possibly correcting for the case that 'd' is not current working directory.

If the path given is not a directory, an exception will be raised:

Program 10: listdir.bet

```
ORIGIN '~beta/basiclib/directory';

--- program: descriptor ---

(* Program showing a simple use of directory: The directory with the path
 * given as argument is scanned, and the names of all the entries are printed
```

```
 * with an indication of what type of entry it is.
 * This is done using the 'select' pattern of 'scanentries'. This is a more
 * efficient startegy than using 'found.entry.isFile' etc., possibly
 * correcting for the case that 'd' is not current working directory.
 * If the path given is not a directory, an exception will be raised.
 *)

(# arg: ^text;
   d: @directory;
   nl: @boolean;
   full: @boolean;
   usage:
     (# do 'Usage: ' -> puttext;
        1->arguments->puttext;
        ' [-f] path' -> putline;
        stop;
     #);
do (* Parse command line *)
   (if noOfArguments
    // 1 then '.' -> d.name
    // 2 then
       2 -> arguments -> arg[];
       (if '-f' -> arg.equal then
           true -> full;
           '.' -> d.name;
        else
           arg[] -> d.name;
       if)
    // 3 then
       2 -> arguments -> arg[];
       (if '-f' -> arg.equal then
           true -> full;
        else usage;
       if);
       3 -> arguments -> d.name;
    else
       usage;
   if);

   (* Scan directory *)
   newline;
   'The content of \''-> puttext;
   d.name -> puttext;
   '\' is: ' -> putline;
   d.scanEntries
   (#
   do select
      (# whenFile::<
           (# do 'File:      ' -> puttext; #);
         whenDir::<
           (# do 'Directory: ' -> puttext; #);
         whenOther::<
           (# do '(Unknown): ' -> puttext; #);
      #);
      (if full then foundFullPath -> putline;
       else found.path -> putline;
      if);
   #);
   newline;
#)
```

9 The systemEnv Libraries

The systemEnv libraries define the experimental concurrency system for BETA. The systemEnv libraries contain five closely related libraries: basicsystemenv.bet, systemenv.bet, and timehandler.bet.

9.1 basicsystemenv

The basicsyst emenv.bet fragment contains the core of the concurrency system, and is the prime fragment for information on the concurrency facilities.

9.2 systemenv

The systemenv.bet fragment is to be used if the concurrency is to be used in a program that is not using any graphical user interface system, such as the X Window System. Systemenv.bet does not define any new attributes at all.

9.3 timehandler

The timehandler.bet fragment contains facilities for setting timers in the form of objects to be executed when a given time period have elapsed.

The rest of the chapter will only describe the basicsystemenv.bet fragments. For details on the use of the other fragments, please see the documentation in the interface descriptions.

9.4 The basicSystemEnv Library

The systemEnv fragment contains abstract superpattern:s for describing the BETA concepts of concurrent systems. The basic ideas are:

9.4.1 Basic concepts

- Components (coroutines) can be executed concurrently.
- A primitive semaphore pattern is available for synchronization. The operations on a semaphore must be executed as an indivisible unit.
- An abstract pattern Monitor similar to the monitor proposed by Hoare and Brinch−Hansen.
- An abstract pattern System is defined. System defines communication between systems by means of synchronized rendezvous. A concurrency imperative conc and an alternation imperative alt are defined for system.

Currently an exploratory style is used to experiment with different variants of the abstract patterns. The current version is thus far from any final form of definition and may contain errors. The separation into interface and implementation has not been completely carried out.

The abstractions defined here are based on the ones described in chapter 12 of the BETA book.

9.4.2 Changes from original design

The implementation is identical to the design in the BETA book, except for the following changes:

- The syntax of fork is

```
S[]->fork
```

- and not S.fork.
- The syntax of conc is

```
conc(# do S1[]->start; S2[]->start; S3[]->start #)
```

- and not conc(# do S1.start; S2.start; S3.start #).
- The syntax of alt is

```
alt(#do S1[]->start; S2[]->start; S3[]->start #)
```

- and not alt(# do S1.start; S2.start; S3.start #).

9.4.3 New facilities

This implementation of systemenv included a few new facilities, not described in the BETA book:

- semaphore have an additional attribute: tryP, which is a non−blocking call of P.
- In addition to s[]−>fork, s[]−>kill is now possible, and in addition to pause, 100−>sleep is possible.
- system have a new virtual attribute, onKilled, that is invoked before the system terminates
- systemenv has a new virtual attribute, deadlocked, that is invoked if all processes are deadlocked.
- Finally, systemenv defines three new attributes to cope with event driven user interfaces: windowEnvType, theWindowEnv, and setWindowEnv. See further details on cooperation with user interface environments below.

9.4.4 The Concurrency is Simulated

In order to implement real concurrency, an interrupt mechanism must be implemented. This is currently not done. A component/system will thus keep the control until it makes an explicit or implicit SUSPEND. An implicit SUSPEND is made when a component must wait for a semaphore, executes the pause pattern, executes the sleep pattern, or performs a blocking communication using the shellEnv distribution libraries (not described in this manual).

9.4.5 Concurrency and User Interface Environments

User interface environments (such as X Window System) are usually event−driven in the sense that actions in the program are executed as a response to user input events. To handle this, a number of separate implementations of SystemEnv exists for the different user interface libraries, such as xtEnv, awEnv, motifEnv, and guiEnv:

- Use systemenv.bet as origin for programs not using event−driven user−interface libraries.
- Use ~beta/Xt/xsystemenv.bet as origin for programs using xtEnv, awEnv or motifEnv.
- Use ~beta/guienv/guienvsystemenv.bet as origin for programs using GUIenv.

See xsystemenv and guienvsystemenv for a description of using systemenv in conjunction with X and GUIenv programs, respectively.

For examples of using SystemEnv, see the demo directory.

Please note, that programs should only use one of the systemenv, xsystemenv, and guienvsystemenv fragments. It is a fairly common mistake in systemEnv programs to find more

than one of these fragments.

9.5 Using the SystemEnv Fragment

A program using the systemEnv fragment will have the following structure:

```
INCLUDE '~beta/basiclib/systemenv'
--- program: descriptor ---
systemenv
(# process: @ |system(# ... #);
do ...
   process[] -> fork;
   ...
#)
```

9.5.1 The Monitor Example

The following is an example of a producer/consumer system with a shared buffer (implemented as a 20 element character buffer, protected as a monitor. The producer and consumer are concurrent objects:

Program 11: buffer.bet

```
ORIGIN '~beta/basiclib/systemenv';

---program: descriptor---
SystemEnv
(# buffer: @Monitor
     (# R: [20] @char; in,out: @integer;
        full,empty: @Condition;

        put: Entry
          (# ch: @char
          enter ch
          do (if in = out then full.wait if);
             ch->R[in]; (in mod R.range)+1 ->in;
             empty.signal;
          #);
        get: Entry
          (# ch: @char
          do (if in = (out mod R.range)+1 then empty.wait if);
             R[(out mod R.range)+1->out]->ch;
             full.signal;
             (if ch=ascii.eot then stop if);
          exit ch
          #);
        init::< (# do 1->in; R.range->out #)
     #);

   prod: @| System(# do cycle(# do (if keyboard.EOS then ascii.eot->buffer.put else keyb
   cons: @| System(# do cycle(# do buffer.get->screen.put #) #);

do buffer.init;
   conc(# do prod[]->start; cons[]->start #)
#)
```

9.5.2 The Monitor with Wait Example

This example is similar to the previous, except that wait is used instead of condition to control the medium−term scheduling of access to the buffer.

Program 12: OCbuf.bet

```
ORIGIN '~beta/basiclib/systemenv';

−−−program: descriptor−−
systemenv
(# buffer: @Monitor
     (# R: [4] @char; in,out: @integer;
        full: (# exit in=out #);
        empty: (#exit (in = (out mod R.range)+1) #);
        Put: Entry
          (# ch: @char
          enter ch
          do wait(# do (not full)−>cond #);
             ch−>R[in]; (in mod R.range)+1 −>in;
          #);
        get: Entry
          (# ch: @char
          do wait(# do (not empty)−>cond #);
             R[(out mod R.range)+1−>out]−>ch;
          exit ch
          #);
        init::< (# do 1−>in; R.range−>out #)
     #);

   prod: @| System(# do cycle(# do (if keyboard.EOS then stop else keyboard.get−>buffer.
   cons: @| System(# do cycle(# do buffer.get−>screen.put #) #);

do buffer.init;
   conc(# do prod[]−>start; cons[]−>start #)
#)
```

9.5.3 The Ports Example

The following is an example of three communicating objects: S, R1 and R2. R1 and R2 are similar (instances of the same Rtype pattern).

Rtype defines two ports: p1 and p2, with a get entry in p1 and a put entry in p2. The get entry prints the value of the x attribute on standard output, and put prints the y attribute. Rtype objects repeatedly accepts p1 communications followed by p2 communications (i.e. since in this example, only get is define in p1, and only put in p2, this implies get followed by put). Note that the use of ports, allows specializations of Rtype to define new entries in p1 and/or p2, such that these communications follow the same communication structure.

The S object defines two internal objects, C1 and C2, which are executed alternating. C1 is responsible for the communication with R1 and C2 is responsible with the communication with R2. It is in this way ensured that the communication pattern between S and R1 follows the get followed by put pattern (the same for S and R2), but these two communication patterns may be non−deterministically interleaved.

Program 13: altex1.bet

```
ORIGIN '~beta/basiclib/systemenv';

---program: descriptor--
SystemEnv
(# S: @| System
     (# C1: @| System
          (#
          do cycle(# do '1'->put; R1.get; '2'->put; R1.put #);
          #);
        C2: @| System
          (#
          do cycle(# do 'a'->put; R2.get; 'b'->put; R2.put #)
          #)
     do alt(# do C1[]->start; C2[]->start #)
     #);

   Rtype: System
     (# get: p1.entry(# do x->screen.put; #); p1: @port;
        put: p2.entry(# do y->screen.put; #); p2: @port;
        x,y: @char
     do cycle(# do p1.accept; p2.accept #)
     #);
   R1: @| Rtype;
   R2: @| Rtype;

do '('->R1.x; ')'->R1.y; '['->R2.x; ']'->R2.y;
   conc(# do S[]->start; R1[]->start; R2[]->start #)
#)
```

9.5.4 The ObjectPort Example

This example illustrates the use of the ObjectPort facility. The S object defines f1, f2, and f3 communication entries, where f1 is controlled by an objectPort. This enables S to control exactly which object which is allowed to communicate f3's in an accept. Note, that initially, R.R1 is allowed, and later R.R3 is allowed:

Program 14: sys1.bet

```
ORIGIN '~beta/basiclib/systemenv';

---program: descriptor--
SystemEnv
(# S: @| System
     (# P1: @ObjectPort;
        f1: P1.entry(# do 'f1 called' ->putline #);
        P2: @Port;
        f2: P2.entry(# do 'f2 called' ->putline #);
        P3: @Port;
        f3: P3.entry(# do 'f3 called' ->putline #);
     do R.R1[]->P1.accept;
        P2.accept;
        P3.accept;
        R.R3[]->P1.accept
     #);

   R: @| System
     (# R1: @| System(# do S.f1; S.f3 #);
        R2: @| System(# do S.f2; #);
```

```
        R3: @| System(# do S.f1 #);
     do 'Start R'−>putLine;
        conc(# do R1[]−>start;  R2[]−>start; R3[]−>start #);
        'End R'−>putLine
     #);

do 'Start'−>putLine;
   conc(# do R[]−>start; S[]−>start  #);
   'End'−>putLine
#)
```

10 The external Library

10.1 Interfacing to C and Pascal

The external fragment contains a general interface to other programming languages. This interface is provided by the patterns external and cStruct. The external pattern is used to interface to procedures and functions written in other languages e.g. C or Pascal. The cStruct pattern is used to be able to create BETA objects with a structure similar to C structs or Pascal records. This interface is used heavily in the system, e.g. in the interface to UNIX and Macintosh, and in the user interface toolkit. The basic external and cStruct patterns are defined in betaenv and the external fragment defines the specific attributes of these patterns. Furthermore, the external fragment defines the externalRecord pattern, which is used for defining the BETA interface to data structures, allocated by the external language (e.g. C).

For passing strings to and from C programs you can use the CString pattern.

10.2 Using the external Fragment

A program using the external fragment will have the following structure:

```
INCLUDE '~beta/basiclib/external
--- program: descriptor ---
(# foo: external(# ... #)
do ...
   ... foo ...
#)
```

10.2.1 Interfacing to External C Functions

When interfacing to C, the pattern callC must be called in the do−part of the External specialization. The BETA compiler will then generate a call to an external routine with the same name as the BETA pattern, using C's style of passing parameters.

A pattern of the form

```
foo: external
  (# enter ... do callC exit ... #)
```

describes the interface to an external C function with entry−point foo(_foo). (The do−part can be left out.) As a convenience, the call to C above need not be specified in which case the compiler will insert it automatically.

If you prefer to give the external a different name from the entry−point name, you can state the entry−point name explicitly. If the entry−point name contains special characters, you are forced to do this:

```
foo: external
  (# ... enter... do 'X$bar' -> callC exit... #);
```

This pattern describes the interface to an external C function, whose entry−point name is X$bar (and not foo).

It is important that there exist a C function with the same name and exactly the number of enter parameters corresponds to the number of parameters of the C function. If the C function returns any results, it is important that an exit parameters is specified in the BETA external pattern, and that this exit value is evaluated in all usage's of this external (due to an error in the current compiler) If the C function does not return any result, no exit parameters may be specified.

10.2.2 Using call back from C

The following example shows how to install call backs from C to BETA. Readers not familiar with call backs should skip this section.

We use declarations like:

```
callBackProc: external
  (* This pattern describes the interface to the procedure
   * that is called on the call back. It may have the
   * following type definition in C:
   *
   *   typedef void (*callBackProcPtr)(int i)
   *)
  (# i: @integer;
  enter i
  (* only the types shortInt, integer, char and boolean
   * can be used in the enter and exit parts
   *)
  do cExternalEntry;
     inner;
     (* Had the return type not been void,
      * the exit part should have appeared here.
      *)
  #);

installCallBack: external
  (* This pattern describes the interface for the C
   * function that installs the call back.
   * It has the following C description:
   *
   *   void installCallBack(callBackProcPtr theProcPtr,int j)
   *)
  (# theProcPtr: ##callBackProc;
     j: @integer;
  enter (theProcPtr##,j)
  do callC;
  #);
```

When writing the actual procedure to be called on the call back, it is easiest to specialise the above callBackProc pattern, as in:

```
(# ...
  myCallBack: callBackProc
    (# ...
    (* do not specialize the enter part *)
    do 'There is a call back.' -> putline;
       'Value received in parameter i is ' -> puttext;
        i -> putint; '.' -> put; newline;
   (* do not specialize the exit part *)
   #);
   j: @integer;
do 46 -> j;
   (* install the call back: *)
```

```
   (myCallBack##, j) -> installCallBack;
   ...
#)
```

10.2.3 Interfacing to External Pascal Procedures

When interfacing to Pascal or another programming language with a similar activation record organization the pattern Pascal must be called in the do−part of the External specialization. The BETA compiler will then generate a call to an external routine with the same name as the BETA pattern, using the Pascal style of passing parameters.

A pattern of the form

```
foo: external
  (# enter ... do Pascal exit ... #)
```

describes the interface to an external Pascal function with entry−point foo (_foo). (Note the exit parameters that must be present in Pascal function interfaces.

A Pascal procedure can be interfaced to through a pattern of the form

```
foo: external
  (# enter ... do Pascal #)
```

If the entry−point of the Pascal function or procedure needs to be explicitly specified, a pattern of the form

```
foo: external
  (# enter ... do 'X$bar' -> Pascal ... #)
```

can be used to describe the interface to an external Pascal procedure, where the entry−point for the Pascal procedure is X$bar instead of foo.

A pattern of the form

```
foo: external
  (# enter ... do '$...' -> PascalTrap exit ... #)
```

or

```
foo: external
  (# enter ... do '{$...,$...,...}' -> PascalTrap exit ... #)
```

describes the interface to an external Pascal procedure that is called using Motorola traps. The string in the first example is in the form of a single hexadecimal number, preceded with a $ (e.g. '$A9FF'). The string in the second example is in the form of a comma separated list of hexadecimal numbers, each preceded with a $ and enclosed with braces (e.g. '{$A9FF,$02F4}'). Decimal numbers may be used for specifying the traps. This is done by leaving out the $.

10.2.4 Example Interfacing to Pascal

As an example, an interface routine to a Pascal function (NewHandle) may be implemented in the following way:

```
NEWHANDLE: external
  (# theHandle: @integer
  do pascal
  exit theHandle
  #);
```

10.2.5 Using call backs from Pascal

Call backs from Pascal is handled similar to call backs from C, except that you should use pascalExternalEntry instead of cExternalEntry.

10.2.6 Interfacing to External Data Structures

Transferring data to and from the external languages is dealt with through two special purpose patterns: cStruct and externalRecord.

cStruct is the means for specifying a BETA object with a specific storage layout, and with the purpose of transferring this object to the external language for processing. That is, a cStruct object is allocated by BETA and made available for processing by the external language.

ExternalRecord is the means for specifying a BETA interface into some data structures, allocated by the external language.

10.2.7 cStruct

cStruct defines byteSize [1] that is used for specifying the number of bytes that should be allocated for the BETA object. For specifying the fields, the local patterns byte, short, signedShort and long are available. These patterns contain a local virtual attribute, pos, that is used to specify the byte position of this field in the cStruct object. Note that there is no check for overlapping fields. cStruct also defines put/getByte, put/getShort, put/getSignedShort and put/getLong operations for accessing the bytes, longs, etc. of the cStruct object directly.

10.2.8 ExternalRecord

ExternalRecord defines the ptr attribute, that is used to contain the memory address of the externally allocated data structure. For specifying the interface into the fields of this data structure, externalRecord defines byte, short, signedShort and long with local virtual attribute, pos, to describe the byte position of each field (just as cStruct). ExternalRecord also defines the put/getByte, etc. to make direct access to the bytes, shorts, etc. of the external data structure.

The connection between the externalRecord object and the external data structure is established by letting some external routine return the address of the external data structure, and then transfer this integer into the ptr attribute of the externalRecord object. If it is necessary to transfer this address back to the external language, it can be done by transferring the ptr attribute back through some external language routine.

10.2.9 Example Using cStruct

The following example shows how to interface to the C language using cStruct and external. The BETA pattern myStruct describes a BETA object to be transferred to the foo C function. The BETA pattern foo describes the interface to a C function called foo. It is important that there exist a C function with the name foo and exactly the same parameters and result.

```
(# myStruct: cStruct
   (* myStruct describes a cStruct consisting of 8 bytes
      'a' denotes byte[0]
      'b' denotes byte[1]
      'c' denotes byte[2−5]
      'd' denotes byte[6−7]
   *)
   (# byteSize ::< (# do 8 −> value #);
      a: byte (# pos ::< (# do 0 −> value #) #);
      b: byte (# pos ::< (# do 1 −> value #) #);
      c: long (# pos ::< (# do 2 −> value #) #);
      d: short (# pos ::< (# do 6 −> value #) #)
   #);
   foo: external
     (* This pattern describes the interface to the following
      * C function, called 'foo':
      *
      *  int foo(int i, short si, char a, char *t, myStruct *r)
      *)
     (# i: @integer; si: @shortint;
        a: @char; t: [1] @char;
        r: ^myStruct;
        status: @ integer;
     enter (i, si, a, t, r[])
     exit status
     #);
   theStruct: @myStruct;
   m, n, status: @integer;
   c: @char;
do ...
   m −> theStruct.a; (* overflow is not detected *)
   17 −> theStruct.b; ...
   (n, 117, c, 'smith', theStruct[]) −> foo −> status;
   (if status
    // 117 then
       theStruct.d −> m;
    ...
   if);
   (11, m, 'x', 'smith', NONE) −> foo −> status;
#)
```

10.2.10 Example Using externalRecord

Here TCSbuffer is the interface into some data structure in some image processing software. TCAlloc is the interface into the C routine in that software, allocating this data structure, and allocate is an example of using this routine for getting access to the externally allocated data structure. Finally, update illustrates how to transfer the memory address back into the external language.

```
TCSbuffer: externalRecord
(# display:         @long(# pos ::< (# do 0 −> value #)#);
   window:           @long(# pos ::< (# do 4 −> value #)#);
   visual:           @long(# pos ::< (# do 8 −> value #)#);
```

```
   colormap:       @long(# pos ::< (# do 12 -> value #)#);
   depth:                  @long(# pos ::< (# do 16 -> value #)#);
   gc:           @long(# pos ::< (# do 20 -> value #)#);
   colorLookup: @long(# pos ::< (# do 24 -> value #)#);
   width:                  @long(# pos ::< (# do 56 -> value #)#);
   height:           @long(# pos ::< (# do 60 -> value #)#);
   data:                     @long(# pos ::< (# do 64 -> value #)#);
   xOffset:         @long(# pos ::< (# do 68 -> value #)#);
   yOffset:         @long(# pos ::< (# do 72 -> value #)#);
   zoom:                     @long(# pos ::< (# do 76 -> value #)#);
   updateTile:   @long(# pos ::< (# do 80 -> value #)#);
#);

TCAlloc: External
  (# width, height: @integer;
     buffer: @integer
  enter (width, height)
  exit buffer
#);

allocate:
  (* allocates a true color buffer of resolution
   * width x height
   *)
  (# width, height: @integer; buffer: ^TCSbuffer
     noMemoryError :< TCSnoMemoryError;
  enter(width, height, buffer[])
  do (width, height) -> TCALLOC -> buffer.ptr;
     (if ptr //-1 then noMemoryError if);
     INNER
  #);

update:
  (* Draws a region of a true color buffer on the window it's
   * associated with.  The x, y, width and height arguments
   * give the location and size of the region in BUFFER
   * coordinates, NOT window coordinates.
   *)
  (# x, y, width, height: @integer; buffer: ^TCSbuffer
     noBufferError :< TCSnoBufferError;
     internError :< TCSinternError;
  enter (buffer, x, y, width, height)
  do (if (buffer.ptr, x, y, width, height) -> TCUPDATE
      // 0 then 'update' -> NoBufferError
      //-4 then 'update' -> internError
     if);
   INNER
#);
```

[1] Note, that cStruct is defined in betaenv, and that the external library defines additional attributes to this cStruct pattern. ByteSize is defined in betaenv, whereas the rest of the attributes mentioned here, are described in external

11 The Perl Compatible Regular Expression Library

The Perl Compatible Regular Expression Library (PCRE) implements the pcre pattern with the method attributes

```
init
options
match
matchAll
replace
replaceAll
```

the value attribute

```
subPatterns
```

the exception attribute

```
compilationError
```

11.1 Regular Expressions

A regular expression (regexp, for short) is a pattern that denotes a set of strings, possibly an infinite set. Searching for matches for a regexp is a very powerful operation that editors and scripting languages on Unix systems have traditionally offered.

This implementation of regular expressions implements a regular expression syntax that is compatible with that in the language Perl. It is based on Philip Hazel's PCRE library. Most of Philip Hazel's documentation also applies to the BETA version.

11.2 pcre

The pcre pattern encapsulates a regular expression. It takes a Text reference as an enter parameter. The enter parameter is given to the Init method.

The pcre pattern has an empty do−part

The pcre pattern exits a reference to itself

You can use the pcre pattern as in the following example

```
   re: ^Pcre;
do 'trigger' −> pcre −> re[];
   (filename[], re[]) −> myGrep;
```

11.3 init

The init method takes a Text reference as an enter parameter. This string describes the regular expression according to the syntax described in the pcre documentation. Init compiles the regular expression into an internal format suitable for matching against strings. This operation takes some CPU time, so the result (stored in the pcre object) should be kept if the same pattern is to be used many times.

When compiling the regular expression the options defined by the options method are used.

You can call init several times if you want to change the regular expression matched by the pcre object.

11.4 options

The options method is a virtual pattern, which you can specialise. Put the options you need into the do part. For example:

```
   re: @Pcre (# options:: (# do CASELESS; DO_STUDY; #); #)
do 'tRiGgEr' −> re;
   (filename[], re[]) −> myGrep;
```

There is an alternative way to specify certain options, which involves placing them in the textual representation of the regular expression. For example the option CASELESS can be specified by prepending the string '<?i>' to the regular expression.

The following options are supported

Option	Text version	Used in	Description
CASELESS	(?i)	init	Ignore case when matching
MULTILINE	(?m)	init	^ and $ match after/before newlines
DOTALL	(?s)	init	. matches newlines
EXTENDED	(?x)	init	Extended regexp syntax
ANCHORED	^	init	Match only at start of string
DOLLAR_ENDONLY		init	$ doesn't match before terminal newline
EXTRA	(?X)	init	Support PCRE extensions to Perl regexps
NOTBOL		init or match	Do not match ^ at start of string
NOTEOL		init or match	Do not match $ at end of string
UNGREEDY	(?U)	init	Quantifiers not greedy by default
NOTEMPTY		init or match	Empty string cannot match entire expression
C_LOCALE		init	Use C locale instead of default localei
DO_STUDY		init	Study regexp after compiling it
RETURN_NONE		init or match	Return NONE for subpatterns that didn't match

Notes:

- ***More information***
  More details are available in the documentation of the pcre library.
- ***When to set options***
  Some options are only used in the init method, while others are set to a default in the init method, but may be overridden in the options method of match (inherited by matchAll, replace and replaceAll).
- ***C_LOCALE***
  Normally pcre will use the locale defined by your C library to determine whether a given character is a letter, etc. If you set this option, then pcre will use the C locale, ie only the characters a−z are letters. Most of the time this option will make no difference.
- ***DO_STUDY***
  If you are going to be using a pattern many times, then this option may improve performance. See more about this in the documentation of the pcre library.
- ***RETURN_NONE***
  Normally the subMatchText methods and similar will exit an empty string in the case where

the subpattern didn't match at all. This helps prevent unexpected ref−NONE exceptions in your code. Unfortunately it also makes it difficult to tell the difference between a subpattern that matched an empty string and a subpattern that didn't match at all (because it was in an alternation that wasn't used). If you set this option then you risk getting NONE back from an invocation of subMatchText and your program must be able to cope with that.

- ***Clearing options***
  In some cases you may want to clear an option that has been set by a superpattern. You do this by prefixing the option name with `clear`. For example:

```
(* p is a perl regexp with my favourite options, including case
 * insensitivity, but just this once I want a case sensitive
 * regexp.
 *)
p: @PcreWithMyFavouriteOptions (# options:: (# do clearCASELESS #) #);
```

11.5 match

The match method takes a Text reference and exits true or false, depending on whether the text matched the expression. It also contains a set of methods that can be overridden to provide much more information about the match.

The INNER part of the match method is only called in the case of a match.

```
options
pre
matchPos
matchText
preMatchText
postMatchText
subMatchPos
subMatchText
sub1, sub2, sub3...
noMatch
position
```

11.6 match.options

This method can be overridden in much the same way as the options method in the pcre pattern in order to pass options to the matching stage of the regular expression engine.

11.7 match.pre

This method is called before any matching takes place. It does nothing, but you can specialise it in your own subclasses.

11.8 match.matchPos

This method can be called from the inner part of the match method. It exits an integer pair, indicating the start and end positions of the matched text in the original text. See the example below.

11.9 match.matchText

This method can be called from the inner part of the match method. It exits a Text reference indicating the text that matched the regular expression. See the example below.

11.10 match.preMatchText and match.postMatchText

These methods can be called from the inner part of the match method. They exit a Text reference indicating the text that preceeded (or followed) the text that matched the regular expression.

For example:

```
(# t1: ^Text;
   t2: ^Text;
   r3: ^Text;
   s: @Integer;
   e: @Integer;
do 'abc123def' -> ('\\d+' -> pcre).match
   (#
   do preMatchText -> t1[];
      matchText -> t2[];
      postMatchText -> t3[];
      matchPos -> (s, e);
   #);
   ...
#);
```

Will put 'abc' in t1, '123' in t2 and 'def' in t3. It also puts 4 in s and 6 in e.

11.11 match.subMatchPos

This method can be called from the inner part of the match method. It enters an integer and exits an integer pair, indicating the start and end positions of the nth subpattern in the original text. See the example below.

11.12 match.subMatchText

This method can be called from the inner part of the match method. It enters an integer and exits a text, indicating the text matched by the nth subpattern. See the example below.

11.13 match.sub1, match.sub2, match.sub3...

These methods can be called from the inner part of the match method. They exit a text, indicating the text matched by the nth subpattern. They are simply a shorthand method of invoking subMatchText. See the example below:

```
(# t1: ^Text;
   t2: ^Text;
   r3: ^Text;
   s: @Integer;
   e: @Integer;
do 'abc123def' -> ('([a-z])(\\d+)([a-z]+)' -> pcre).match
   (#
   do sub1 -> t1[];
      sub2 -> t2[];
      3 -> subMatchText -> t3[];
      3 -> subMatchPos -> (s, e);
   #);
   ...
#);
```

Will put 'c' in t1, '123' in t2 and 'def' in t3. It also puts 7 in s and 9 in e.

11.14 match.noMatch

This method is called by match when no match is found. You can specialise it to specify an action if no match is found.

11.15 match.position

This method controls where in the input string the search for a regular expression match starts. You can specialise it, putting a different number into the variable 'value'.

11.16 replace

The replace method inherits from the match method. It takes two inputs, firstly a Text reference to a search string, and secondly a text reference to a default replacement string. It exits two values, firstly a boolean (true or false), depending on whether the text matched the expression. Secondly the a text reference to the new text with the replacement carried out. If no replacement is carried out then the text exited is a copy of the search string entered. Replace also contains a set of methods that can be overridden to provide much more information about the match and to control the replacement text more accurately. See the example below.

The INNER part of the replace method is only called in the case of a match.

```
options
pre
matchPos
matchText
preMatchText
postMatchText
subMatchPos
subMatchText
sub1, sub2, sub3...
noMatch
position
rep
```

11.17 replace.rep

This method controls the replacement string. The 'value' variable is a reference to the default replacement text. By assigning a new reference to 'value' you can dynamically choose another replacement string, based on information gleaned from the other methods available in replace.

```
(# t1: ^Text;
do ('The y2k problem', 'year 2000' ->
      ('\\by2k\\b' -> pcre).replace ->
      (p, t1[]);
   ...
#);
```

Will put 'The year 2000 problem' in t1. (The escape sequence '\b' in a regular expression matches a word boundary. In a BETA string you have to double the backslash.)

```
(# t1: ^Text;
do ('The y3k problem', '' ->
      ('\\by([0-9]+)k\\b' -> pcre).replace
      (# rep::
         (#
         do 'year %s000' -> putFormat (# do sub1 -> s #) -> value[];
```

```
        #);
      #) -> (p, t1[]);
   ...
#);
```

11.18 matchAll

This method is similar to match, but calls INNER several times, once for each match. It is not yet fully documented. Please see pcre.bet comments and demo programs.

11.19 replaceAll

This method is similar to replace, but calls INNER several times, once for each match. It is not yet fully documented. Please see pcre.bet comments and demo programs.

11.20 subPatterns

This is a readonly integer pattern, which tells you how many subpatterns your pattern has. Only useful if you are reading regular expressions from a config file or from the user, since otherwise you should know this figure already.

11.21 compilationError

This pattern is executed if your regular expression contains syntax errors. In this case it is not a good idea to call match or replace on that pattern.

12 PCRE specification

12.1 Name

pcre − Perl−compatible regular expressions.

12.2 Description

The PCRE library is a set of functions that implement regular expression pattern matching using the same syntax and semantics as Perl 5, with just a few differences (see below). The current implementation corresponds to Perl 5.005, with some additional features from the Perl development release.

This documentation is an edited version of the full documentation for the C language interface to the PCRE library. Some sections that were of no interest to BETA users were removed.

The functions **pcre_compile()**, **pcre_study()**, and **pcre_exec()** are used for compiling and matching regular expressions, while **pcre_copy_substring()**, **pcre_get_substring()**, and **pcre_get_substring_list()** are convenience functions for extracting captured substrings from a matched subject string. The function **pcre_maketables()** is used (optionally) to build a set of character tables in the current locale for passing to **pcre_compile()**.

The function **pcre_fullinfo()** is used to find out information about a compiled pattern; **pcre_info()** is an obsolete version which returns only some of the available information, but is retained for

backwards compatibility. The function **pcre_version()** returns a pointer to a string containing the version of PCRE and its date of release.

The global variables **pcre_malloc** and **pcre_free** initially contain the entry points of the standard **malloc()** and **free()** functions respectively. PCRE calls the memory management functions via these variables, so a calling program can replace them if it wishes to intercept the calls. This should be done before calling any PCRE functions.

12.3 Compiling a Pattern

The function **pcre_compile()** is called to compile a pattern into an internal form. The pattern is a C string terminated by a binary zero, and is passed in the argument *pattern*. A pointer to a single block of memory that is obtained via **pcre_malloc** is returned. This contains the compiled code and related data. The **pcre** type is defined for this for convenience, but in fact **pcre** is just a typedef for **void**, since the contents of the block are not externally defined. It is up to the caller to free the memory when it is no longer required.

The size of a compiled pattern is roughly proportional to the length of the pattern string, except that each character class (other than those containing just a single character, negated or not) requires 33 bytes, and repeat quantifiers with a minimum greater than one or a bounded maximum cause the relevant portions of the compiled pattern to be replicated.

The *options* argument contains independent bits that affect the compilation. It should be zero if no options are required. Some of the options, in particular, those that are compatible with Perl, can also be set and unset from within the pattern (see the detailed description of regular expressions below). For these options, the contents of the *options* argument specifies their initial settings at the start of compilation and execution. The PCRE_ANCHORED option can be set at the time of matching as well as at compile time.

If *errptr* is NULL, **pcre_compile()** returns NULL immediately. Otherwise, if compilation of a pattern fails, **pcre_compile()** returns NULL, and sets the variable pointed to by *errptr* to point to a textual error message. The offset from the start of the pattern to the character where the error was discovered is placed in the variable pointed to by *erroffset*, which must not be NULL. If it is, an immediate error is given.

If the final argument, *tableptr*, is NULL, PCRE uses a default set of character tables which are built when it is compiled, using the default C locale. Otherwise, *tableptr* must be the result of a call to **pcre_maketables()**. See the section on locale support below.

The following option bits are defined in the header file:

```
PCRE_ANCHORED
```

If this bit is set, the pattern is forced to be "anchored", that is, it is constrained to match only at the start of the string which is being searched (the "subject string"). This effect can also be achieved by appropriate constructs in the pattern itself, which is the only way to do it in Perl.

```
PCRE_CASELESS
```

If this bit is set, letters in the pattern match both upper and lower case letters. It is equivalent to Perl's /i option.

```
PCRE_DOLLAR_ENDONLY
```

If this bit is set, a dollar metacharacter in the pattern matches only at the end of the subject string.

Without this option, a dollar also matches immediately before the final character if it is a newline (but not before any other newlines). The PCRE_DOLLAR_ENDONLY option is ignored if PCRE_MULTILINE is set. There is no equivalent to this option in Perl.

PCRE_DOTALL

If this bit is set, a dot metacharater in the pattern matches all characters, including newlines. Without it, newlines are excluded. This option is equivalent to Perl's /s option. A negative class such as [^a] always matches a newline character, independent of the setting of this option.

PCRE_EXTENDED

If this bit is set, whitespace data characters in the pattern are totally ignored except when escaped or inside a character class, and characters between an unescaped # outside a character class and the next newline character, inclusive, are also ignored. This is equivalent to Perl's /x option, and makes it possible to include comments inside complicated patterns. Note, however, that this applies only to data characters. Whitespace characters may never appear within special character sequences in a pattern, for example within the sequence (?(which introduces a conditional subpattern.

PCRE_EXTRA

This option was invented in order to turn on additional functionality of PCRE that is incompatible with Perl, but it is currently of very little use. When set, any backslash in a pattern that is followed by a letter that has no special meaning causes an error, thus reserving these combinations for future expansion. By default, as in Perl, a backslash followed by a letter with no special meaning is treated as a literal. There are at present no other features controlled by this option. It can also be set by a (?X) option setting within a pattern.

PCRE_MULTILINE

By default, PCRE treats the subject string as consisting of a single "line" of characters (even if it actually contains several newlines). The "start of line" metacharacter (^) matches only at the start of the string, while the "end of line" metacharacter ($) matches only at the end of the string, or before a terminating newline (unless PCRE_DOLLAR_ENDONLY is set). This is the same as Perl.

When PCRE_MULTILINE it is set, the "start of line" and "end of line" constructs match immediately following or immediately before any newline in the subject string, respectively, as well as at the very start and end. This is equivalent to Perl's /m option. If there are no "\n" characters in a subject string, or no occurrences of ^ or $ in a pattern, setting PCRE_MULTILINE has no effect.

PCRE_UNGREEDY

This option inverts the "greediness" of the quantifiers so that they are not greedy by default, but become greedy if followed by "?". It is not compatible with Perl. It can also be set by a (?U) option setting within the pattern.

12.4 Studying a Pattern

When a pattern is going to be used several times, it is worth spending more time analyzing it in order to speed up the time taken for matching. The function **pcre_study()** takes a pointer to a compiled pattern as its first argument, and returns a pointer to a **pcre_extra** block (another **void** typedef) containing additional information about the pattern; this can be passed to **pcre_exec()**. If no additional information is available, NULL is returned.

The second argument contains option bits. At present, no options are defined for **pcre_study()**, and this argument should always be zero.

The third argument for **pcre_study()** is a pointer to an error message. If studying succeeds (even if no data is returned), the variable it points to is set to NULL. Otherwise it points to a textual error message.

At present, studying a pattern is useful only for non−anchored patterns that do not have a single fixed starting character. A bitmap of possible starting characters is created.

12.5 Locale Support

PCRE handles caseless matching, and determines whether characters are letters, digits, or whatever, by reference to a set of tables. The library contains a default set of tables which is created in the default C locale when PCRE is compiled. This is used when the final argument of **pcre_compile()** is NULL, and is sufficient for many applications.

An alternative set of tables can, however, be supplied. Such tables are built by calling the **pcre_maketables()** function, which has no arguments, in the relevant locale. The result can then be passed to **pcre_compile()** as often as necessary. For example, to build and use tables that are appropriate for the French locale (where accented characters with codes greater than 128 are treated as letters), the following code could be used:

```
setlocale(LC_CTYPE, "fr");
tables = pcre_maketables();
re = pcre_compile(..., tables);
```

The tables are built in memory that is obtained via **pcre_malloc**. The pointer that is passed to **pcre_compile** is saved with the compiled pattern, and the same tables are used via this pointer by **pcre_study()** and **pcre_exec()**. Thus for any single pattern, compilation, studying and matching all happen in the same locale, but different patterns can be compiled in different locales. It is the caller's responsibility to ensure that the memory containing the tables remains available for as long as it is needed.

12.6 Information About a Pattern

The **pcre_fullinfo()** function returns information about a compiled pattern. It replaces the obsolete **pcre_info()** function, which is nevertheless retained for backwards compability (and is documented below).

The first argument for **pcre_fullinfo()** is a pointer to the compiled pattern. The second argument is the result of **pcre_study()**, or NULL if the pattern was not studied. The third argument specifies which piece of information is required, while the fourth argument is a pointer to a variable to receive the data. The yield of the function is zero for success, or one of the following negative numbers:

```
PCRE_ERROR_NULL       the argument code was NULL
                      the argument where was NULL
PCRE_ERROR_BADMAGIC   the "magic number" was not found
PCRE_ERROR_BADOPTION  the value of what was invalid
```

The possible values for the third argument are defined in **pcre.h**, and are as follows:

```
PCRE_INFO_OPTIONS
```

Return a copy of the options with which the pattern was compiled. The fourth argument should

point to au **unsigned long int** variable. These option bits are those specified in the call to **pcre_compile()**, modified by any top−level option settings within the pattern itself, and with the PCRE_ANCHORED bit forcibly set if the form of the pattern implies that it can match only at the start of a subject string.

```
PCRE_INFO_SIZE
```

Return the size of the compiled pattern, that is, the value that was passed as the argument to **pcre_malloc()** when PCRE was getting memory in which to place the compiled data. The fourth argument should point to a **size_t** variable.

```
PCRE_INFO_CAPTURECOUNT
```

Return the number of capturing subpatterns in the pattern. The fourth argument should point to an \fbint\fR variable.

```
PCRE_INFO_BACKREFMAX
```

Return the number of the highest back reference in the pattern. The fourth argument should point to an **int** variable. Zero is returned if there are no back references.

```
PCRE_INFO_FIRSTCHAR
```

Return information about the first character of any matched string, for a non−anchored pattern. If there is a fixed first character, e.g. from a pattern such as (cat|cow|coyote), it is returned in the integer pointed to by *where*. Otherwise, if either

(a) the pattern was compiled with the PCRE_MULTILINE option, and every branch starts with "^", or

(b) every branch of the pattern starts with ".*" and PCRE_DOTALL is not set (if it were set, the pattern would be anchored),

−1 is returned, indicating that the pattern matches only at the start of a subject string or after any "\n" within the string. Otherwise −2 is returned. For anchored patterns, −2 is returned.

```
PCRE_INFO_FIRSTTABLE
```

If the pattern was studied, and this resulted in the construction of a 256−bit table indicating a fixed set of characters for the first character in any matching string, a pointer to the table is returned. Otherwise NULL is returned. The fourth argument should point to an **unsigned char *** variable.

```
PCRE_INFO_LASTLITERAL
```

For a non−anchored pattern, return the value of the rightmost literal character which must exist in any matched string, other than at its start. The fourth argument should point to an **int** variable. If there is no such character, or if the pattern is anchored, −1 is returned. For example, for the pattern /a\d+z\d+/ the returned value is 'z'.

The **pcre_info()** function is now obsolete because its interface is too restrictive to return all the available data about a compiled pattern. New programs should use **pcre_fullinfo()** instead. The yield of **pcre_info()** is the number of capturing subpatterns, or one of the following negative numbers:

```
PCRE_ERROR_NULL       the argument code was NULL
PCRE_ERROR_BADMAGIC   the "magic number" was not found
```

If the *optptr* argument is not NULL, a copy of the options with which the pattern was compiled is placed in the integer it points to (see PCRE_INFO_OPTIONS above).

If the pattern is not anchored and the *firstcharptr* argument is not NULL, it is used to pass back information about the first character of any matched string (see PCRE_INFO_FIRSTCHAR above).

12.7 Matching a Pattern

The function **pcre_exec()** is called to match a subject string against a pre−compiled pattern, which is passed in the *code* argument. If the pattern has been studied, the result of the study should be passed in the *extra* argument. Otherwise this must be NULL.

The PCRE_ANCHORED option can be passed in the *options* argument, whose unused bits must be zero. However, if a pattern was compiled with PCRE_ANCHORED, or turned out to be anchored by virtue of its contents, it cannot be made unachored at matching time.

There are also three further options that can be set only at matching time:

```
PCRE_NOTBOL
```

The first character of the string is not the beginning of a line, so the circumflex metacharacter should not match before it. Setting this without PCRE_MULTILINE (at compile time) causes circumflex never to match.

```
PCRE_NOTEOL
```

The end of the string is not the end of a line, so the dollar metacharacter should not match it nor (except in multiline mode) a newline immediately before it. Setting this without PCRE_MULTILINE (at compile time) causes dollar never to match.

```
PCRE_NOTEMPTY
```

An empty string is not considered to be a valid match if this option is set. If there are alternatives in the pattern, they are tried. If all the alternatives match the empty string, the entire match fails. For example, if the pattern

```
a?b?
```

is applied to a string not beginning with "a" or "b", it matches the empty string at the start of the subject. With PCRE_NOTEMPTY set, this match is not valid, so PCRE searches further into the string for occurrences of "a" or "b".

Perl has no direct equivalent of PCRE_NOTEMPTY, but it does make a special case of a pattern match of the empty string within its **split()** function, and when using the /g modifier. It is possible to emulate Perl's behaviour after matching a null string by first trying the match again at the same offset with PCRE_NOTEMPTY set, and then if that fails by advancing the starting offset (see below) and trying an ordinary match again.

The subject string is passed as a pointer in *subject*, a length in *length*, and a starting offset in *startoffset*. Unlike the pattern string, it may contain binary zero characters. When the starting offset is zero, the search for a match starts at the beginning of the subject, and this is by far the most common case.

A non−zero starting offset is useful when searching for another match in the same subject by calling **pcre_exec()** again after a previous success. Setting *startoffset* differs from just passing over

a shortened string and setting PCRE_NOTBOL in the case of a pattern that begins with any kind of lookbehind. For example, consider the pattern

```
\Biss\B
```

which finds occurrences of "iss" in the middle of words. (\B matches only if the current position in the subject is not a word boundary.) When applied to the string "Mississipi" the first call to **pcre_exec()** finds the first occurrence. If **pcre_exec()** is called again with just the remainder of the subject, namely "issipi", it does not match, because \B is always false at the start of the subject, which is deemed to be a word boundary. However, if **pcre_exec()** is passed the entire string again, but with *startoffset* set to 4, it finds the second occurrence of "iss" because it is able to look behind the starting point to discover that it is preceded by a letter.

If a non−zero starting offset is passed when the pattern is anchored, one attempt to match at the given offset is tried. This can only succeed if the pattern does not require the match to be at the start of the subject.

In general, a pattern matches a certain portion of the subject, and in addition, further substrings from the subject may be picked out by parts of the pattern. Following the usage in Jeffrey Friedl's book, this is called "capturing" in what follows, and the phrase "capturing subpattern" is used for a fragment of a pattern that picks out a substring. PCRE supports several other kinds of parenthesized subpattern that do not cause substrings to be captured.

Captured substrings are returned to the caller via a vector of integer offsets whose address is passed in *ovector*. The number of elements in the vector is passed in *ovecsize*. The first two−thirds of the vector is used to pass back captured substrings, each substring using a pair of integers. The remaining third of the vector is used as workspace by **pcre_exec()** while matching capturing subpatterns, and is not available for passing back information. The length passed in *ovecsize* should always be a multiple of three. If it is not, it is rounded down.

When a match has been successful, information about captured substrings is returned in pairs of integers, starting at the beginning of *ovector*, and continuing up to two−thirds of its length at the most. The first element of a pair is set to the offset of the first character in a substring, and the second is set to the offset of the first character after the end of a substring. The first pair, *ovector[0]* and *ovector[1]*, identify the portion of the subject string matched by the entire pattern. The next pair is used for the first capturing subpattern, and so on. The value returned by **pcre_exec()** is the number of pairs that have been set. If there are no capturing subpatterns, the return value from a successful match is 1, indicating that just the first pair of offsets has been set.

Some convenience functions are provided for extracting the captured substrings as separate strings. These are described in the following section.

It is possible for an capturing subpattern number *n+1* to match some part of the subject when subpattern *n* has not been used at all. For example, if the string "abc" is matched against the pattern (a|(z))(bc) subpatterns 1 and 3 are matched, but 2 is not. When this happens, both offset values corresponding to the unused subpattern are set to −1.

If a capturing subpattern is matched repeatedly, it is the last portion of the string that it matched that gets returned.

If the vector is too small to hold all the captured substrings, it is used as far as possible (up to two−thirds of its length), and the function returns a value of zero. In particular, if the substring offsets are not of interest, **pcre_exec()** may be called with *ovector* passed as NULL and *ovecsize* as zero. However, if the pattern contains back references and the *ovector* isn't big enough to remember the related substrings, PCRE has to get additional memory for use during matching.

Thus it is usually advisable to supply an *ovector*.

Note that **pcre_info()** can be used to find out how many capturing subpatterns there are in a compiled pattern. The smallest size for *ovector* that will allow for *n* captured substrings in addition to the offsets of the substring matched by the whole pattern is (*n*+1)*3.

If **pcre_exec()** fails, it returns a negative number. The following are defined in the header file:

```
PCRE_ERROR_NOMATCH        (−1)
```

The subject string did not match the pattern.

```
PCRE_ERROR_NULL           (−2)
```

Either *code* or *subject* was passed as NULL, or *ovector* was NULL and *ovecsize* was not zero.

```
PCRE_ERROR_BADOPTION      (−3)
```

An unrecognized bit was set in the *options* argument.

```
PCRE_ERROR_BADMAGIC       (−4)
```

PCRE stores a 4−byte "magic number" at the start of the compiled code, to catch the case when it is passed a junk pointer. This is the error it gives when the magic number isn't present.

```
PCRE_ERROR_UNKNOWN_NODE   (−5)
```

While running the pattern match, an unknown item was encountered in the compiled pattern. This error could be caused by a bug in PCRE or by overwriting of the compiled pattern.

```
PCRE_ERROR_NOMEMORY       (−6)
```

If a pattern contains back references, but the *ovector* that is passed to **pcre_exec()** is not big enough to remember the referenced substrings, PCRE gets a block of memory at the start of matching to use for this purpose. If the call via **pcre_malloc()** fails, this error is given. The memory is freed at the end of matching.

12.8 Extracting Captured Substrings

Captured substrings can be accessed directly by using the offsets returned by **pcre_exec()** in *ovector*. For convenience, the functions **pcre_copy_substring()**, **pcre_get_substring()**, and **pcre_get_substring_list()** are provided for extracting captured substrings as new, separate, zero−terminated strings. A substring that contains a binary zero is correctly extracted and has a further zero added on the end, but the result does not, of course, function as a C string.

The first three arguments are the same for all three functions: *subject* is the subject string which has just been successfully matched, *ovector* is a pointer to the vector of integer offsets that was passed to **pcre_exec()**, and *stringcount* is the number of substrings that were captured by the match, including the substring that matched the entire regular expression. This is the value returned by **pcre_exec** if it is greater than zero. If **pcre_exec()** returned zero, indicating that it ran out of space in *ovector*, the value passed as *stringcount* should be the size of the vector divided by three.

The functions **pcre_copy_substring()** and **pcre_get_substring()** extract a single substring, whose number is given as *stringnumber*. A value of zero extracts the substring that matched the

entire pattern, while higher values extract the captured substrings. For **pcre_copy_substring()**, the string is placed in *buffer*, whose length is given by *buffersize*, while for **pcre_get_substring()** a new block of store is obtained via **pcre_malloc**, and its address is returned via *stringptr*. The yield of the function is the length of the string, not including the terminating zero, or one of

```
PCRE_ERROR_NOMEMORY       (−6)
```

The buffer was too small for **pcre_copy_substring()**, or the attempt to get memory failed for **pcre_get_substring()**.

```
PCRE_ERROR_NOSUBSTRING    (−7)
```

There is no substring whose number is *stringnumber*.

The **pcre_get_substring_list()** function extracts all available substrings and builds a list of pointers to them. All this is done in a single block of memory which is obtained via **pcre_malloc**. The address of the memory block is returned via *listptr*, which is also the start of the list of string pointers. The end of the list is marked by a NULL pointer. The yield of the function is zero if all went well, or

```
PCRE_ERROR_NOMEMORY       (−6)
```

if the attempt to get the memory block failed.

When any of these functions encounter a substring that is unset, which can happen when capturing subpattern number *n+1* matches some part of the subject, but subpattern *n* has not been used at all, they return an empty string. This can be distinguished from a genuine zero−length substring by inspecting the appropriate offset in *ovector*, which is negative for unset substrings.

12.9 Limitations

There are some size limitations in PCRE but it is hoped that they will never in practice be relevant. The maximum length of a compiled pattern is 65539 (sic) bytes. All values in repeating quantifiers must be less than 65536. The maximum number of capturing subpatterns is 99. The maximum number of all parenthesized subpatterns, including capturing subpatterns, assertions, and other types of subpattern, is 200.

The maximum length of a subject string is the largest positive number that an integer variable can hold. However, PCRE uses recursion to handle subpatterns and indefinite repetition. This means that the available stack space may limit the size of a subject string that can be processed by certain patterns.

12.10 Differences from perl

The differences described here are with respect to Perl 5.005.

1. By default, a whitespace character is any character that the C library function **isspace()** recognizes, though it is possible to compile PCRE with alternative character type tables. Normally **isspace()** matches space, formfeed, newline, carriage return, horizontal tab, and vertical tab. Perl 5 no longer includes vertical tab in its set of whitespace characters. The \v escape that was in the Perl documentation for a long time was never in fact recognized. However, the character itself was treated as whitespace at least up to 5.002. In 5.004 and 5.005 it does not match \s.

2. PCRE does not allow repeat quantifiers on lookahead assertions. Perl permits them, but they do

not mean what you might think. For example, (?!a){3} does not assert that the next three characters are not "a". It just asserts that the next character is not "a" three times.

3. Capturing subpatterns that occur inside negative lookahead assertions are counted, but their entries in the offsets vector are never set. Perl sets its numerical variables from any such patterns that are matched before the assertion fails to match something (thereby succeeding), but only if the negative lookahead assertion contains just one branch.

4. Though binary zero characters are supported in the subject string, they are not allowed in a pattern string because it is passed as a normal C string, terminated by zero. The escape sequence "\0" can be used in the pattern to represent a binary zero.

5. The following Perl escape sequences are not supported: \l, \u, \L, \U, \E, \Q. In fact these are implemented by Perl's general string−handling and are not part of its pattern matching engine.

6. The Perl \G assertion is not supported as it is not relevant to single pattern matches.

7. Fairly obviously, PCRE does not support the (?{code}) and (?p{code}) constructions. However, there is some experimental support for recursive patterns using the non−Perl item (?R).

8. There are at the time of writing some oddities in Perl 5.005_02 concerned with the settings of captured strings when part of a pattern is repeated. For example, matching "aba" against the pattern /^(a(b)?)+$/ sets $2 to the value "b", but matching "aabbaa" against /^(aa(bb)?)+$/ leaves $2 unset. However, if the pattern is changed to /^(aa(b(b))?)+$/ then $2 (and $3) are set.

In Perl 5.004 $2 is set in both cases, and that is also true of PCRE. If in the future Perl changes to a consistent state that is different, PCRE may change to follow.

9. Another as yet unresolved discrepancy is that in Perl 5.005_02 the pattern /^(a)?(?(1)a|b)+$/ matches the string "a", whereas in PCRE it does not. However, in both Perl and PCRE /^(a)?a/ matched against "a" leaves $1 unset.

10. PCRE provides some extensions to the Perl regular expression facilities:

(a) Although lookbehind assertions must match fixed length strings, each alternative branch of a lookbehind assertion can match a different length of string. Perl 5.005 requires them all to have the same length.

(b) If PCRE_DOLLAR_ENDONLY is set and PCRE_MULTILINE is not set, the $ meta− character matches only at the very end of the string.

(c) If PCRE_EXTRA is set, a backslash followed by a letter with no special meaning is faulted.

(d) If PCRE_UNGREEDY is set, the greediness of the repetition quantifiers is inverted, that is, by default they are not greedy, but if followed by a question mark they are.

(e) PCRE_ANCHORED can be used to force a pattern to be tried only at the start of the subject.

(f) The PCRE_NOTBOL, PCRE_NOTEOL, and PCRE_NOTEMPTY options for **pcre_exec()** have no Perl equivalents.

(g) The (?R) construct allows for recursive pattern matching (Perl 5.6 can do this using the (?p{code}) construct, which PCRE cannot of course support.)

12.11 Regular Expression Details

The syntax and semantics of the regular expressions supported by PCRE are described below. Regular expressions are also described in the Perl documentation and in a number of other books, some of which have copious examples. Jeffrey Friedl's "Mastering Regular Expressions", published by O'Reilly (ISBN 1−56592−257), covers them in great detail. The description here is intended as reference documentation.

A regular expression is a pattern that is matched against a subject string from left to right. Most characters stand for themselves in a pattern, and match the corresponding characters in the subject. As a trivial example, the pattern

```
The quick brown fox
```

matches a portion of a subject string that is identical to itself. The power of regular expressions comes from the ability to include alternatives and repetitions in the pattern. These are encoded in the pattern by the use of *meta−characters*, which do not stand for themselves but instead are interpreted in some special way.

There are two different sets of meta−characters: those that are recognized anywhere in the pattern except within square brackets, and those that are recognized in square brackets. Outside square brackets, the meta−characters are as follows:

```
\      general escape character with several uses
^      assert start of subject (or line, in multiline mode)
$      assert end of subject (or line, in multiline mode)
.      match any character except newline (by default)
[      start character class definition
|      start of alternative branch
(      start subpattern
)      end subpattern
?      extends the meaning of (
       also 0 or 1 quantifier
       also quantifier minimizer
*      0 or more quantifier
+      1 or more quantifier
{      start min/max quantifier
```

Part of a pattern that is in square brackets is called a "character class". In a character class the only meta−characters are:

```
\      general escape character
^      negate the class, but only if the first character
−      indicates character range
]      terminates the character class
```

The following sections describe the use of each of the meta−characters.

12.12 Backslash

The backslash character has several uses. Firstly, if it is followed by a non−alphameric character, it takes away any special meaning that character may have. This use of backslash as an escape character applies both inside and outside character classes.

For example, if you want to match a "*" character, you write "*" in the pattern. This applies whether or not the following character would otherwise be interpreted as a meta−character, so it is always safe to precede a non−alphameric with "\" to specify that it stands for itself. In particular, if you want

to match a backslash, you write "\\".

If a pattern is compiled with the PCRE_EXTENDED option, whitespace in the pattern (other than in a character class) and characters between a "#" outside a character class and the next newline character are ignored. An escaping backslash can be used to include a whitespace or "#" character as part of the pattern.

A second use of backslash provides a way of encoding non−printing characters in patterns in a visible manner. There is no restriction on the appearance of non−printing characters, apart from the binary zero that terminates a pattern, but when a pattern is being prepared by text editing, it is usually easier to use one of the following escape sequences than the binary character it represents:

```
\a     alarm, that is, the BEL character (hex 07)
\cx    "control−x", where x is any character
\e     escape (hex 1B)
\f     formfeed (hex 0C)
\n     newline (hex 0A)
\r     carriage return (hex 0D)
\t     tab (hex 09)
\xhh   character with hex code hh
\ddd   character with octal code ddd, or backreference
```

The precise effect of "\cx" is as follows: if "x" is a lower case letter, it is converted to upper case. Then bit 6 of the character (hex 40) is inverted. Thus "\cz" becomes hex 1A, but "\c{" becomes hex 3B, while "\c;" becomes hex 7B.

After "\x", up to two hexadecimal digits are read (letters can be in upper or lower case).

After "\0" up to two further octal digits are read. In both cases, if there are fewer than two digits, just those that are present are used. Thus the sequence "\0\x\07" specifies two binary zeros followed by a BEL character. Make sure you supply two digits after the initial zero if the character that follows is itself an octal digit.

The handling of a backslash followed by a digit other than 0 is complicated. Outside a character class, PCRE reads it and any following digits as a decimal number. If the number is less than 10, or if there have been at least that many previous capturing left parentheses in the expression, the entire sequence is taken as a *back reference*. A description of how this works is given later, following the discussion of parenthesized subpatterns.

Inside a character class, or if the decimal number is greater than 9 and there have not been that many capturing subpatterns, PCRE re−reads up to three octal digits following the backslash, and generates a single byte from the least significant 8 bits of the value. Any subsequent digits stand for themselves. For example:

```
\040   is another way of writing a space
\40    is the same, provided there are fewer than 40
          previous capturing subpatterns
\7     is always a back reference
\11    might be a back reference, or another way of
          writing a tab
\011   is always a tab
\0113  is a tab followed by the character "3"
\113   is the character with octal code 113 (since there
          can be no more than 99 back references)
\377   is a byte consisting entirely of 1 bits
\81    is either a back reference, or a binary zero
          followed by the two characters "8" and "1"
```

Note that octal values of 100 or greater must not be introduced by a leading zero, because no more than three octal digits are ever read.

All the sequences that define a single byte value can be used both inside and outside character classes. In addition, inside a character class, the sequence "\b" is interpreted as the backspace character (hex 08). Outside a character class it has a different meaning (see below).

The third use of backslash is for specifying generic character types:

```
\d     any decimal digit
\D     any character that is not a decimal digit
\s     any whitespace character
\S     any character that is not a whitespace character
\w     any "word" character
\W     any "non−word" character
```

Each pair of escape sequences partitions the complete set of characters into two disjoint sets. Any given character matches one, and only one, of each pair.

A "word" character is any letter or digit or the underscore character, that is, any character which can be part of a Perl "word". The definition of letters and digits is controlled by PCRE's character tables, and may vary if locale− specific matching is taking place (see "Locale support" above). For example, in the "fr" (French) locale, some character codes greater than 128 are used for accented letters, and these are matched by \w.

These character type sequences can appear both inside and outside character classes. They each match one character of the appropriate type. If the current matching point is at the end of the subject string, all of them fail, since there is no character to match.

The fourth use of backslash is for certain simple assertions. An assertion specifies a condition that has to be met at a particular point in a match, without consuming any characters from the subject string. The use of subpatterns for more complicated assertions is described below. The backslashed assertions are

```
\b     word boundary
\B     not a word boundary
\A     start of subject (independent of multiline mode)
\Z     end of subject or newline at end (independent of multiline mode)
\z     end of subject (independent of multiline mode)
```

These assertions may not appear in character classes (but note that "\b" has a different meaning, namely the backspace character, inside a character class).

A word boundary is a position in the subject string where the current character and the previous character do not both match \w or \W (i.e. one matches \w and the other matches \W), or the start or end of the string if the first or last character matches \w, respectively.

The \A, \Z, and \z assertions differ from the traditional circumflex and dollar (described below) in that they only ever match at the very start and end of the subject string, whatever options are set. They are not affected by the PCRE_NOTBOL or PCRE_NOTEOL options. If the *startoffset* argument of **pcre_exec()** is non−zero, \A can never match. The difference between \Z and \z is that \Z matches before a newline that is the last character of the string as well as at the end of the string, whereas \z matches only at the end.

12.13 Circumflex and Dollar

Outside a character class, in the default matching mode, the circumflex character is an assertion which is true only if the current matching point is at the start of the subject string. If the *startoffset* argument of **pcre_exec()** is non−zero, circumflex can never match. Inside a character class, circumflex has an entirely different meaning (see below).

Circumflex need not be the first character of the pattern if a number of alternatives are involved, but it should be the first thing in each alternative in which it appears if the pattern is ever to match that branch. If all possible alternatives start with a circumflex, that is, if the pattern is constrained to match only at the start of the subject, it is said to be an "anchored" pattern. (There are also other constructs that can cause a pattern to be anchored.)

A dollar character is an assertion which is true only if the current matching point is at the end of the subject string, or immediately before a newline character that is the last character in the string (by default). Dollar need not be the last character of the pattern if a number of alternatives are involved, but it should be the last item in any branch in which it appears. Dollar has no special meaning in a character class.

The meaning of dollar can be changed so that it matches only at the very end of the string, by setting the PCRE_DOLLAR_ENDONLY option at compile or matching time. This does not affect the \Z assertion.

The meanings of the circumflex and dollar characters are changed if the PCRE_MULTILINE option is set. When this is the case, they match immediately after and immediately before an internal "\n" character, respectively, in addition to matching at the start and end of the subject string. For example, the pattern /^abc$/ matches the subject string "def\nabc" in multiline mode, but not otherwise. Consequently, patterns that are anchored in single line mode because all branches start with "^" are not anchored in multiline mode, and a match for circumflex is possible when the *startoffset* argument of **pcre_exec()** is non−zero. The PCRE_DOLLAR_ENDONLY option is ignored if PCRE_MULTILINE is set.

Note that the sequences \A, \Z, and \z can be used to match the start and end of the subject in both modes, and if all branches of a pattern start with \A is it always anchored, whether PCRE_MULTILINE is set or not.

12.14 Full Stop (Period, Dot)

Outside a character class, a dot in the pattern matches any one character in the subject, including a non−printing character, but not (by default) newline. If the PCRE_DOTALL option is set, dots match newlines as well. The handling of dot is entirely independent of the handling of circumflex and dollar, the only relationship being that they both involve newline characters. Dot has no special meaning in a character class.

12.15 Square Brackets

An opening square bracket introduces a character class, terminated by a closing square bracket. A closing square bracket on its own is not special. If a closing square bracket is required as a member of the class, it should be the first data character in the class (after an initial circumflex, if present) or escaped with a backslash.

A character class matches a single character in the subject; the character must be in the set of characters defined by the class, unless the first character in the class is a circumflex, in which case

the subject character must not be in the set defined by the class. If a circumflex is actually required as a member of the class, ensure it is not the first character, or escape it with a backslash.

For example, the character class [aeiou] matches any lower case vowel, while [^aeiou] matches any character that is not a lower case vowel. Note that a circumflex is just a convenient notation for specifying the characters which are in the class by enumerating those that are not. It is not an assertion: it still consumes a character from the subject string, and fails if the current pointer is at the end of the string.

When caseless matching is set, any letters in a class represent both their upper case and lower case versions, so for example, a caseless [aeiou] matches "A" as well as "a", and a caseless [^aeiou] does not match "A", whereas a caseful version would.

The newline character is never treated in any special way in character classes, whatever the setting of the PCRE_DOTALL or PCRE_MULTILINE options is. A class such as [^a] will always match a newline.

The minus (hyphen) character can be used to specify a range of characters in a character class. For example, [d−m] matches any letter between d and m, inclusive. If a minus character is required in a class, it must be escaped with a backslash or appear in a position where it cannot be interpreted as indicating a range, typically as the first or last character in the class.

It is not possible to have the literal character "]" as the end character of a range. A pattern such as [W−]46] is interpreted as a class of two characters ("W" and "−") followed by a literal string "46]", so it would match "W46]" or "−46]". However, if the "]" is escaped with a backslash it is interpreted as the end of range, so [W−\]46] is interpreted as a single class containing a range followed by two separate characters. The octal or hexadecimal representation of "]" can also be used to end a range.

Ranges operate in ASCII collating sequence. They can also be used for characters specified numerically, for example [\000−\037]. If a range that includes letters is used when caseless matching is set, it matches the letters in either case. For example, [W−c] is equivalent to [][\^_`wxyzabc], matched caselessly, and if character tables for the "fr" locale are in use, [\xc8−\xcb] matches accented E characters in both cases.

The character types \d, \D, \s, \S, \w, and \W may also appear in a character class, and add the characters that they match to the class. For example, [\dABCDEF] matches any hexadecimal digit. A circumflex can conveniently be used with the upper case character types to specify a more restricted set of characters than the matching lower case type. For example, the class [^\W_] matches any letter or digit, but not underscore.

All non−alphameric characters other than \, −, ^ (at the start) and the terminating] are non−special in character classes, but it does no harm if they are escaped.

12.16 Posix Character Classes

Perl 5.6 (not yet released at the time of writing) is going to support the POSIX notation for character classes, which uses names enclosed by [: and :] within the enclosing square brackets. PCRE supports this notation. For example,

```
[01[:alpha:]%]
```

matches "0", "1", any alphabetic character, or "%". The supported class names are

```
alnum    letters and digits
alpha    letters
ascii    character codes 0 − 127
cntrl    control characters
digit    decimal digits (same as \d)
graph    printing characters, excluding space
lower    lower case letters
print    printing characters, including space
punct    printing characters, excluding letters and digits
space    white space (same as \s)
upper    upper case letters
word     "word" characters (same as \w)
xdigit   hexadecimal digits
```

The names "ascii" and "word" are Perl extensions. Another Perl extension is negation, which is indicated by a ^ character after the colon. For example,

```
[12[:^digit:]]
```

matches "1", "2", or any non−digit. PCRE (and Perl) also recogize the POSIX syntax [.ch.] and [=ch=] where "ch" is a "collating element", but these are not supported, and an error is given if they are encountered.

12.17 Vertical Bar

Vertical bar characters are used to separate alternative patterns. For example, the pattern

```
gilbert|sullivan
```

matches either "gilbert" or "sullivan". Any number of alternatives may appear, and an empty alternative is permitted (matching the empty string). The matching process tries each alternative in turn, from left to right, and the first one that succeeds is used. If the alternatives are within a subpattern (defined below), "succeeds" means matching the rest of the main pattern as well as the alternative in the subpattern.

12.18 Internal Option Setting

The settings of PCRE_CASELESS, PCRE_MULTILINE, PCRE_DOTALL, and PCRE_EXTENDED can be changed from within the pattern by a sequence of Perl option letters enclosed between "(?" and ")". The option letters are

```
i  for PCRE_CASELESS
m  for PCRE_MULTILINE
s  for PCRE_DOTALL
x  for PCRE_EXTENDED
```

For example, (?im) sets caseless, multiline matching. It is also possible to unset these options by preceding the letter with a hyphen, and a combined setting and unsetting such as (?im−sx), which sets PCRE_CASELESS and PCRE_MULTILINE while unsetting PCRE_DOTALL and PCRE_EXTENDED, is also permitted. If a letter appears both before and after the hyphen, the option is unset.

The scope of these option changes depends on where in the pattern the setting occurs. For settings that are outside any subpattern (defined below), the effect is the same as if the options were set or unset at the start of matching. The following patterns all behave in exactly the same way:

```
(?i)abc
a(?i)bc
ab(?i)c
abc(?i)
```

which in turn is the same as compiling the pattern abc with PCRE_CASELESS set. In other words, such "top level" settings apply to the whole pattern (unless there are other changes inside subpatterns). If there is more than one setting of the same option at top level, the rightmost setting is used.

If an option change occurs inside a subpattern, the effect is different. This is a change of behaviour in Perl 5.005. An option change inside a subpattern affects only that part of the subpattern that follows it, so

```
(a(?i)b)c
```

matches abc and aBc and no other strings (assuming PCRE_CASELESS is not used). By this means, options can be made to have different settings in different parts of the pattern. Any changes made in one alternative do carry on into subsequent branches within the same subpattern. For example,

```
(a(?i)b|c)
```

matches "ab", "aB", "c", and "C", even though when matching "C" the first branch is abandoned before the option setting. This is because the effects of option settings happen at compile time. There would be some very weird behaviour otherwise.

The PCRE−specific options PCRE_UNGREEDY and PCRE_EXTRA can be changed in the same way as the Perl−compatible options by using the characters U and X respectively. The (?X) flag setting is special in that it must always occur earlier in the pattern than any of the additional features it turns on, even when it is at top level. It is best put at the start.

12.19 Subpatterns

Subpatterns are delimited by parentheses (round brackets), which can be nested. Marking part of a pattern as a subpattern does two things:

1. It localizes a set of alternatives. For example, the pattern

```
cat(aract|erpillar|)
```

matches one of the words "cat", "cataract", or "caterpillar". Without the parentheses, it would match "cataract", "erpillar" or the empty string.

2. It sets up the subpattern as a capturing subpattern (as defined above). When the whole pattern matches, that portion of the subject string that matched the subpattern is passed back to the caller via the *ovector* argument of **pcre_exec()**. Opening parentheses are counted from left to right (starting from 1) to obtain the numbers of the capturing subpatterns.

For example, if the string "the red king" is matched against the pattern

```
the ((red|white) (king|queen))
```

the captured substrings are "red king", "red", and "king", and are numbered 1, 2, and 3.

The fact that plain parentheses fulfil two functions is not always helpful. There are often times when

a grouping subpattern is required without a capturing requirement. If an opening parenthesis is followed by "?:", the subpattern does not do any capturing, and is not counted when computing the number of any subsequent capturing subpatterns. For example, if the string "the white queen" is matched against the pattern

```
the ((?:red|white) (king|queen))
```

the captured substrings are "white queen" and "queen", and are numbered 1 and 2. The maximum number of captured substrings is 99, and the maximum number of all subpatterns, both capturing and non−capturing, is 200.

As a convenient shorthand, if any option settings are required at the start of a non−capturing subpattern, the option letters may appear between the "?" and the ":". Thus the two patterns

```
(?i:saturday|sunday)
(?:(?i)saturday|sunday)
```

match exactly the same set of strings. Because alternative branches are tried from left to right, and options are not reset until the end of the subpattern is reached, an option setting in one branch does affect subsequent branches, so the above patterns match "SUNDAY" as well as "Saturday".

12.20 Repetition

Repetition is specified by quantifiers, which can follow any of the following items:

```
a single character, possibly escaped
the . metacharacter
a character class
a back reference (see next section)
a parenthesized subpattern (unless it is an assertion − see below)
```

The general repetition quantifier specifies a minimum and maximum number of permitted matches, by giving the two numbers in curly brackets (braces), separated by a comma. The numbers must be less than 65536, and the first must be less than or equal to the second. For example:

```
z{2,4}
```

matches "zz", "zzz", or "zzzz". A closing brace on its own is not a special character. If the second number is omitted, but the comma is present, there is no upper limit; if the second number and the comma are both omitted, the quantifier specifies an exact number of required matches. Thus

```
[aeiou]{3,}
```

matches at least 3 successive vowels, but may match many more, while

```
\d{8}
```

matches exactly 8 digits. An opening curly bracket that appears in a position where a quantifier is not allowed, or one that does not match the syntax of a quantifier, is taken as a literal character. For example, {,6} is not a quantifier, but a literal string of four characters.

The quantifier {0} is permitted, causing the expression to behave as if the previous item and the quantifier were not present.

For convenience (and historical compatibility) the three most common quantifiers have single−character abbreviations:

```
*    is equivalent to {0,}
+    is equivalent to {1,}
?    is equivalent to {0,1}
```

It is possible to construct infinite loops by following a subpattern that can match no characters with a quantifier that has no upper limit, for example:

```
(a?)*
```

Earlier versions of Perl and PCRE used to give an error at compile time for such patterns. However, because there are cases where this can be useful, such patterns are now accepted, but if any repetition of the subpattern does in fact match no characters, the loop is forcibly broken.

By default, the quantifiers are "greedy", that is, they match as much as possible (up to the maximum number of permitted times), without causing the rest of the pattern to fail. The classic example of where this gives problems is in trying to match comments in C programs. These appear between the sequences /* and */ and within the sequence, individual * and / characters may appear. An attempt to match C comments by applying the pattern

```
/\*.*\*/
```

to the string

```
/* first command */  not comment  /* second comment */
```

fails, because it matches the entire string due to the greediness of the .* item.

However, if a quantifier is followed by a question mark, it ceases to be greedy, and instead matches the minimum number of times possible, so the pattern

```
/\*.*?\*/
```

does the right thing with the C comments. The meaning of the various quantifiers is not otherwise changed, just the preferred number of matches. Do not confuse this use of question mark with its use as a quantifier in its own right. Because it has two uses, it can sometimes appear doubled, as in

```
\d??\d
```

which matches one digit by preference, but can match two if that is the only way the rest of the pattern matches.

If the PCRE_UNGREEDY option is set (an option which is not available in Perl), the quantifiers are not greedy by default, but individual ones can be made greedy by following them with a question mark. In other words, it inverts the default behaviour.

When a parenthesized subpattern is quantified with a minimum repeat count that is greater than 1 or with a limited maximum, more store is required for the compiled pattern, in proportion to the size of the minimum or maximum.

If a pattern starts with .* or .{0,} and the PCRE_DOTALL option (equivalent to Perl's /s) is set, thus allowing the . to match newlines, the pattern is implicitly anchored, because whatever follows will be tried against every character position in the subject string, so there is no point in retrying the overall match at any position after the first. PCRE treats such a pattern as though it were preceded by \A. In cases where it is known that the subject string contains no newlines, it is worth setting PCRE_DOTALL when the pattern begins with .* in order to obtain this optimization, or alternatively

using ^ to indicate anchoring explicitly.

When a capturing subpattern is repeated, the value captured is the substring that matched the final iteration. For example, after

```
(tweedle[dume]{3}\s*)+
```

has matched "tweedledum tweedledee" the value of the captured substring is "tweedledee". However, if there are nested capturing subpatterns, the corresponding captured values may have been set in previous iterations. For example, after

```
/(a|(b))+/
```

matches "aba" the value of the second captured substring is "b".

12.21 Back References

Outside a character class, a backslash followed by a digit greater than 0 (and possibly further digits) is a back reference to a capturing subpattern earlier (i.e. to its left) in the pattern, provided there have been that many previous capturing left parentheses.

However, if the decimal number following the backslash is less than 10, it is always taken as a back reference, and causes an error only if there are not that many capturing left parentheses in the entire pattern. In other words, the parentheses that are referenced need not be to the left of the reference for numbers less than 10. See the section entitled "Backslash" above for further details of the handling of digits following a backslash.

A back reference matches whatever actually matched the capturing subpattern in the current subject string, rather than anything matching the subpattern itself. So the pattern

```
(sens|respons)e and \1ibility
```

matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility". If caseful matching is in force at the time of the back reference, the case of letters is relevant. For example,

```
((?i)rah)\s+\1
```

matches "rah rah" and "RAH RAH", but not "RAH rah", even though the original capturing subpattern is matched caselessly.

There may be more than one back reference to the same subpattern. If a subpattern has not actually been used in a particular match, any back references to it always fail. For example, the pattern

```
(a|(bc))\2
```

always fails if it starts to match "a" rather than "bc". Because there may be up to 99 back references, all digits following the backslash are taken as part of a potential back reference number. If the pattern continues with a digit character, some delimiter must be used to terminate the back reference. If the PCRE_EXTENDED option is set, this can be whitespace. Otherwise an empty comment can be used.

A back reference that occurs inside the parentheses to which it refers fails when the subpattern is first used, so, for example, (a\1) never matches. However, such references can be useful inside

repeated subpatterns. For example, the pattern

```
(a|b\1)+
```

matches any number of "a"s and also "aba", "ababaa" etc. At each iteration of the subpattern, the back reference matches the character string corresponding to the previous iteration. In order for this to work, the pattern must be such that the first iteration does not need to match the back reference. This can be done using alternation, as in the example above, or by a quantifier with a minimum of zero.

12.22 Assertions

An assertion is a test on the characters following or preceding the current matching point that does not actually consume any characters. The simple assertions coded as \b, \B, \A, \Z, \z, ^ and $ are described above. More complicated assertions are coded as subpatterns. There are two kinds: those that look ahead of the current position in the subject string, and those that look behind it.

An assertion subpattern is matched in the normal way, except that it does not cause the current matching position to be changed. Lookahead assertions start with (?= for positive assertions and (?! for negative assertions. For example,

```
\w+(?=;)
```

matches a word followed by a semicolon, but does not include the semicolon in the match, and

```
foo(?!bar)
```

matches any occurrence of "foo" that is not followed by "bar". Note that the apparently similar pattern

```
(?!foo)bar
```

does not find an occurrence of "bar" that is preceded by something other than "foo"; it finds any occurrence of "bar" whatsoever, because the assertion (?!foo) is always true when the next three characters are "bar". A lookbehind assertion is needed to achieve this effect.

Lookbehind assertions start with (?<= for positive assertions and (?<! for negative assertions. For example,

```
(?60;!foo)bar
```

does find an occurrence of "bar" that is not preceded by "foo". The contents of a lookbehind assertion are restricted such that all the strings it matches must have a fixed length. However, if there are several alternatives, they do not all have to have the same fixed length. Thus

```
(?60;=bullock|donkey)
```

is permitted, but

```
(?60;!dogs?|cats?)
```

causes an error at compile time. Branches that match different length strings are permitted only at the top level of a lookbehind assertion. This is an extension compared with Perl 5.005, which requires all branches to match the same length of string. An assertion such as

```
(?60;=ab(c|de))
```

is not permitted, because its single top−level branch can match two different lengths, but it is acceptable if rewritten to use two top−level branches:

```
(?60;=abc|abde)
```

The implementation of lookbehind assertions is, for each alternative, to temporarily move the current position back by the fixed width and then try to match. If there are insufficient characters before the current position, the match is deemed to fail. Lookbehinds in conjunction with once−only subpatterns can be particularly useful for matching at the ends of strings; an example is given at the end of the section on once−only subpatterns.

Several assertions (of any sort) may occur in succession. For example,

```
(?60;=\d{3})(?60;!999)foo
```

matches "foo" preceded by three digits that are not "999". Notice that each of the assertions is applied independently at the same point in the subject string. First there is a check that the previous three characters are all digits, and then there is a check that the same three characters are not "999". This pattern does *not* match "foo" preceded by six characters, the first of which are digits and the last three of which are not "999". For example, it doesn't match "123abcfoo". A pattern to do that is

```
(?60;=\d{3}...)(?60;!999)foo
```

This time the first assertion looks at the preceding six characters, checking that the first three are digits, and then the second assertion checks that the preceding three characters are not "999".

Assertions can be nested in any combination. For example,

```
(?60;=(?60;!foo)bar)baz
```

matches an occurrence of "baz" that is preceded by "bar" which in turn is not preceded by "foo", while

```
(?60;=\d{3}(?!999)...)foo
```

is another pattern which matches "foo" preceded by three digits and any three characters that are not "999".

Assertion subpatterns are not capturing subpatterns, and may not be repeated, because it makes no sense to assert the same thing several times. If any kind of assertion contains capturing subpatterns within it, these are counted for the purposes of numbering the capturing subpatterns in the whole pattern. However, substring capturing is carried out only for positive assertions, because it does not make sense for negative assertions.

Assertions count towards the maximum of 200 parenthesized subpatterns.

12.23 Once−Only Subpatterns

With both maximizing and minimizing repetition, failure of what follows normally causes the repeated item to be re−evaluated to see if a different number of repeats allows the rest of the pattern to match. Sometimes it is useful to prevent this, either to change the nature of the match, or to cause it fail earlier than it otherwise might, when the author of the pattern knows there is no point

in carrying on.

Consider, for example, the pattern \d+foo when applied to the subject line

```
123456bar
```

After matching all 6 digits and then failing to match "foo", the normal action of the matcher is to try again with only 5 digits matching the \d+ item, and then with 4, and so on, before ultimately failing. Once−only subpatterns provide the means for specifying that once a portion of the pattern has matched, it is not to be re−evaluated in this way, so the matcher would give up immediately on failing to match "foo" the first time. The notation is another kind of special parenthesis, starting with (?> as in this example:

```
(?62;\d+)bar
```

This kind of parenthesis "locks up" the part of the pattern it contains once it has matched, and a failure further into the pattern is prevented from backtracking into it. Backtracking past it to previous items, however, works as normal.

An alternative description is that a subpattern of this type matches the string of characters that an identical standalone pattern would match, if anchored at the current point in the subject string.

Once−only subpatterns are not capturing subpatterns. Simple cases such as the above example can be thought of as a maximizing repeat that must swallow everything it can. So, while both \d+ and \d+? are prepared to adjust the number of digits they match in order to make the rest of the pattern match, (?>\d+) can only match an entire sequence of digits.

This construction can of course contain arbitrarily complicated subpatterns, and it can be nested.

Once−only subpatterns can be used in conjunction with lookbehind assertions to specify efficient matching at the end of the subject string. Consider a simple pattern such as

```
abcd$
```

when applied to a long string which does not match. Because matching proceeds from left to right, PCRE will look for each "a" in the subject and then see if what follows matches the rest of the pattern. If the pattern is specified as

```
^.*abcd$
```

the initial .* matches the entire string at first, but when this fails (because there is no following "a"), it backtracks to match all but the last character, then all but the last two characters, and so on. Once again the search for "a" covers the entire string, from right to left, so we are no better off. However, if the pattern is written as

```
^(?62;.*)(?60;=abcd)
```

there can be no backtracking for the .* item; it can match only the entire string. The subsequent lookbehind assertion does a single test on the last four characters. If it fails, the match fails immediately. For long strings, this approach makes a significant difference to the processing time.

When a pattern contains an unlimited repeat inside a subpattern that can itself be repeated an unlimited number of times, the use of a once−only subpattern is the only way to avoid some failing matches taking a very long time indeed. The pattern

```
(\D+|60;\d+62;)*[!?]
```

matches an unlimited number of substrings that either consist of non−digits, or digits enclosed in <>, followed by either ! or ?. When it matches, it runs quickly. However, if it is applied to

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

it takes a long time before reporting failure. This is because the string can be divided between the two repeats in a large number of ways, and all have to be tried. (The example used [!?] rather than a single character at the end, because both PCRE and Perl have an optimization that allows for fast failure when a single character is used. They remember the last single character that is required for a match, and fail early if it is not present in the string.) If the pattern is changed to

```
((?62;\D+)|60;\d+62;)*[!?]
```

sequences of non−digits cannot be broken, and failure happens quickly.

12.24 Conditional Subpatterns

It is possible to cause the matching process to obey a subpattern conditionally or to choose between two alternative subpatterns, depending on the result of an assertion, or whether a previous capturing subpattern matched or not. The two possible forms of conditional subpattern are

```
(?(condition)yes−pattern)
(?(condition)yes−pattern|no−pattern)
```

If the condition is satisfied, the yes−pattern is used; otherwise the no−pattern (if present) is used. If there are more than two alternatives in the subpattern, a compile−time error occurs.

There are two kinds of condition. If the text between the parentheses consists of a sequence of digits, the condition is satisfied if the capturing subpattern of that number has previously matched. Consider the following pattern, which contains non−significant white space to make it more readable (assume the PCRE_EXTENDED option) and to divide it into three parts for ease of discussion:

```
( \( )?    [^()]+    (?(1) \) )
```

The first part matches an optional opening parenthesis, and if that character is present, sets it as the first captured substring. The second part matches one or more characters that are not parentheses. The third part is a conditional subpattern that tests whether the first set of parentheses matched or not. If they did, that is, if subject started with an opening parenthesis, the condition is true, and so the yes−pattern is executed and a closing parenthesis is required. Otherwise, since no−pattern is not present, the subpattern matches nothing. In other words, this pattern matches a sequence of non−parentheses, optionally enclosed in parentheses.

If the condition is not a sequence of digits, it must be an assertion. This may be a positive or negative lookahead or lookbehind assertion. Consider this pattern, again containing non−significant white space, and with the two alternatives on the second line:

```
(?(?=[^a−z]*[a−z])
\d{2}−[a−z]{3}−\d{2}  |  \d{2}−\d{2}−\d{2} )
```

The condition is a positive lookahead assertion that matches an optional sequence of non−letters followed by a letter. In other words, it tests for the presence of at least one letter in the subject. If a letter is found, the subject is matched against the first alternative; otherwise it is matched against the second. This pattern matches strings in one of the two forms dd−aaa−dd or dd−dd−dd, where

aaa are letters and dd are digits.

12.25 Comments

The sequence (?# marks the start of a comment which continues up to the next closing parenthesis. Nested parentheses are not permitted. The characters that make up a comment play no part in the pattern matching at all.

If the PCRE_EXTENDED option is set, an unescaped # character outside a character class introduces a comment that continues up to the next newline character in the pattern.

12.26 Recursive Patterns

Consider the problem of matching a string in parentheses, allowing for unlimited nested parentheses. Without the use of recursion, the best that can be done is to use a pattern that matches up to some fixed depth of nesting. It is not possible to handle an arbitrary nesting depth. Perl 5.6 has provided an experimental facility that allows regular expressions to recurse (amongst other things). It does this by interpolating Perl code in the expression at run time, and the code can refer to the expression itself. A Perl pattern to solve the parentheses problem can be created like this:

```
$re = qr{\( (?: (?62;[^()]+) | (?p{$re}) )* \)}x;
```

The (?p{...}) item interpolates Perl code at run time, and in this case refers recursively to the pattern in which it appears. Obviously, PCRE cannot support the interpolation of Perl code. Instead, the special item (?R) is provided for the specific case of recursion. This PCRE pattern solves the parentheses problem (assume the PCRE_EXTENDED option is set so that white space is ignored):

```
\( ( (?62;[^()]+) | (?R) )* \)
```

First it matches an opening parenthesis. Then it matches any number of substrings which can either be a sequence of non−parentheses, or a recursive match of the pattern itself (i.e. a correctly parenthesized substring). Finally there is a closing parenthesis.

This particular example pattern contains nested unlimited repeats, and so the use of a once−only subpattern for matching strings of non−parentheses is important when applying the pattern to strings that do not match. For example, when it is applied to

```
(aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa()
```

it yields "no match" quickly. However, if a once−only subpattern is not used, the match runs for a very long time indeed because there are so many different ways the + and * repeats can carve up the subject, and all have to be tested before failure can be reported.

The values set for any capturing subpatterns are those from the outermost level of the recursion at which the subpattern value is set. If the pattern above is matched against

```
(ab(cd)ef)
```

the value for the capturing parentheses is "ef", which is the last value taken on at the top level. If additional parentheses are added, giving

```
\( ( ( (?62;[^()]+) | (?R) )* ) \)
   ^                        ^
   ^                        ^
```

the string they capture is "ab(cd)ef", the contents of the top level parentheses. If there are more than 15 capturing parentheses in a pattern, PCRE has to obtain extra memory to store data during a recursion, which it does by using **pcre_malloc**, freeing it via **pcre_free** afterwards. If no memory can be obtained, it saves data for the first 15 capturing parentheses only, as there is no way to give an out−of−memory error from within a recursion.

12.27 Performance

Certain items that may appear in patterns are more efficient than others. It is more efficient to use a character class like [aeiou] than a set of alternatives such as (a|e|i|o|u). In general, the simplest construction that provides the required behaviour is usually the most efficient. Jeffrey Friedl's book contains a lot of discussion about optimizing regular expressions for efficient performance.

When a pattern begins with .* and the PCRE_DOTALL option is set, the pattern is implicitly anchored by PCRE, since it can match only at the start of a subject string. However, if PCRE_DOTALL is not set, PCRE cannot make this optimization, because the . metacharacter does not then match a newline, and if the subject string contains newlines, the pattern may match from the character immediately following one of them instead of from the very start. For example, the pattern

```
(.*) second
```

matches the subject "first\nand second" (where \n stands for a newline character) with the first captured substring being "and". In order to do this, PCRE has to retry the match starting after every newline in the subject.

If you are using such a pattern with subject strings that do not contain newlines, the best performance is obtained by setting PCRE_DOTALL, or starting the pattern with ^.* to indicate explicit anchoring. That saves PCRE from having to scan along the subject looking for a newline to restart at.

Beware of patterns that contain nested indefinite repeats. These can take a long time to run when applied to a string that does not match. Consider the pattern fragment

```
(a+)*
```

This can match "aaaa" in 33 different ways, and this number increases very rapidly as the string gets longer. (The * repeat can match 0, 1, 2, 3, or 4 times, and for each of those cases other than 0, the + repeats can match different numbers of times.) When the remainder of the pattern is such that the entire match is going to fail, PCRE has in principle to try every possible variation, and this can take an extremely long time.

An optimization catches some of the more simple cases such as

```
(a+)*b
```

where a literal character follows. Before embarking on the standard matching procedure, PCRE checks that there is a "b" later in the subject string, and if there is not, it fails the match immediately. However, when there is no following literal this optimization cannot be used. You can see the difference by comparing the behaviour of

```
(a+)*\d
```

with the pattern above. The former gives a failure almost instantly when applied to a whole line of "a" characters, whereas the latter takes an appreciable time with strings longer than about 20

characters.

12.28 Author

Philip Hazel <ph10@cam.ac.uk>
University Computing Service,
New Museums Site,
Cambridge CB2 3QG, England.
Phone: +44 1223 334714

Last updated: 27 January 2000

13.1 Basicsystemenv Interface

```
ORIGIN 'betaenv';
LIB_DEF 'basicsystemenv' '../lib';
BODY 'private/basicsystemenvbody'
---LIB:attributes---
(*
 * COPYRIGHT
 *       Copyright Mjolner Informatics, 1992-96
 *       All rights reserved.
 *
 * This fragment contains abstract superpatterns for describing the
 * BETA concepts of concurrent systems.
 *
 * The basic ideas are
 *
 *      A. Components (coroutines) can be executed concurrently
 *
 *      B. A primitive semaphore pattern is available for
 *         syncronization.
 *
 *      C. An abstract pattern 'Monitor' similar to the monitor
 *         proposed by Hoare and Brinch-Hansen
 *
 *      D. An abstract pattern 'System' is defined. System defines
 *         communication between systems by means of synchronized
 *         rendezvous.  A concurreny imperative 'conc' is defined for
 *         systems.
 *
 * The abstractions defined here are identical to the ones described
 * in chapter 12 of the BETA book except for the following points:
 *
 * 1. The syntax of 'fork' is
 *      S[]->fork
 *    and NOT S.fork
 *
 * 2. The syntax of 'conc' is
 *      conc(# do S1[]->start; S2[]->start; S3[]->start #)
 *    and NOT conc(# do S1.start; S2.start; S3.start #)
 *
 * 4. THE CONCURRENCY IS SIMULATED In order to implement real
 *    concurreny, an interrupt mechanism must be implemented. This is
 *    currently NOT done. A component/system will thus keep the
 *    control until it makes an explicit or implicit SUSPEND.  An
 *    implicit SUSPEND is made when a component must wait for a
 *    semaphore, executes the pause pattern, executes the sleep
 *    pattern, or performs a blocking communication using the shellEnv
 *    distribution abstractions.  As the concurrency is simulated,
 *    there is no difference between the implementation of the alt and
 *    conc imperatives.
 *
 * 5. A program using concurrency must have the form:
 *      systemenv(# ... do ... #)
 *
 * 6. Concurrency and X-Windows/macenv/guienv
 *    User interface environments are usually event-driven in the
 *    sense that actions in the program are executed as a response to
 *    user input events.  To handle this, a number of separate
 *    implementations of SystemEnv exists for different user interface
 *    libraries:
 *
 *    Use systemenv.bet as origin for programs not using event-driven
 *    user-interface libraries.
 *
```

```
 *    Use ~beta/Xt/current/xsystemenv.bet as origin for programs using
 *    XtEnv, AwEnv or MotifEnv.
 *
 *    Use ~beta/guienv/current/guienvsystemenv.bet as origin for
 *    programs using GUIenv (Lidskvjalv).
 *
 *    See xsystemenv and guienvsystemenv for a description of using
 *    systemenv in conjunction with X and GUIenv programs,
 *    respectively.
 *
 *    See ~beta/macenv/current/macsystemenv for a description of using
 *    systemenv and macenv.
 *
 * For examples of using SystemEnv see the demo directory.
 *)
getSystemEnv:
  (* Returns the unique systemEnv instance running *)
  (# systemEnvType:< systemEnv;
     theSystemEnv: ^systemEnvType;
  do(* SystemEnv## −> objectPool.strucGet
     (# init::< (#
          do (failure,
              'Program:descriptor must be a subpattern of systemEnv')
               −> stop
     #)#) −> theSystemEnv[]; *)
     objectPool.get
     (# type::systemEnvType;
        init::
          (#
          do (failure,'Illegal use of systemenv. You may have precisely one systemenv instance!
               −>stop;
          #)
     #)−>theSystemEnv[];
  exit theSystemEnv[]
  #);
SystemEnv: SysHead
  (# <<SLOT systemlib:attributes >>;
     semaphore:
       (* P and V are the usual semaphore operations.
        *
        * tryP returns true if the P operation succeded. Returns false
        * if a P would block the caller. In that case the P operation
        * is not performed.
        *
        * Count returns the number of components waiting for the
        *         semaphore.
        *)
       (# P: @...;
          V: @...;
          tryP: @BooleanValue
            (# ... #);
          Count: @
            (# value: @Integer;
            ...
            exit value
            #);
          semRep: @...
       #);
     fork: @
       (* S is put into the queue of scheduled systems. The calling
        * system keeps control, i.e. is not preempted.
        *)
       (# first:  @...;
          second: @...;
          S: ^|SysHead
       enter S[]
```

```
  do first; second; none -> s[];
  #);
kill: @
  (* Kills S. If S is the active system, this is equivalent to a
   * direct suspend.
   *)
  (# S: ^|SysHead; doKill: @...
  enter S[]
  do doKill
  #);
pause: @
  (* Moves the calling system to the end of the queue of
   * scheduled systems.
   *)
  ...;
sleep: @
  (* Makes the calling system sleep at least time seconds.  If
   * time is 0 or negative, sleep has no effect.
   *)
  (# time: @Real
  enter time
  ...
  #);
sleepUntil:
  (* Makes the calling system sleep until at least time.  If
   * time is less than the current time, sleepUntil has no effect.
   *)
  (# time: @Real
  enter time
  ...
  #);
timeStamp:
  (# value: @Real;
  ...
  exit value
  #);
Monitor:
  (# (* idx+ *)
     Condition:
       (# q: @Semaphore;
          Wait: ...;
          Signal: ...;
       #);
     Wait:
       (# cond: @boolean
       do INNER;
          (if not cond then
              return; (* exit monitor *)
              pause;
              mutex.P; (* reentry of monitor *)
              restart Wait
          if)
       #);
     Entry: (# do mutex.P; INNER; return #);
     init:< (# do INNER; mutex.V; #);
     (* private:
      *
      * mutex controls entry to the Monitor.  urgent delays a
      * signalling process.
      *
      * return is executed by processes leaving the monitor.
      * Reactivates possible processes waiting for entry: delayed
      * signalling processes (urgent) have first priority
      *)
     mutex: @semaphore;
     urgent: @semaphore;
```

```
      return: @...;
    #);
  System: SysHead
    (# Port:
         (# mx,m: @Semaphore;
            entry: (# do m.P; INNER; mx.V #);
            accept: (# do m.V; mx.P #)
         #);
       RestrictedPort:
         (# mx, am: @Semaphore;
            Delayed: @...;
            accept:<
              (# ... #);
            acceptable:<
              (# OK: @Boolean; s: ^|sysHead enter s[] do INNER exit OK #);
            restrictedEntry:
              (# ... #);
         #);
       ObjectPort: RestrictedPort
         (# accept::< (# enter sender[] do none->sender[] #);
            acceptable::< (# ... #);
            entry: RestrictedEntry (# do INNER #);
            sender: ^|sysHead
         #);
       QualifiedPort: RestrictedPort
         (# accept::< (# enter sender## do none->sender## #);
            acceptable::< (# ... #);
            entry: RestrictedEntry(# do INNER #);
            sender: ##sysHead
         #);
       conc:
         (# start:
              (# s: ^|system
              enter s[]
              ...
              #);
            concPriv: @...
         do INNER; ...;
         #);
       alt: conc (# do INNER #);
       onKilled:<
         (* Called before this system terminates. *)
         (#
         do (if caller[]<>NONE then (* not the outermost system *)
                caller.dec; NONE -> caller[]
            if);
            INNER;
         #);
       caller: ^protectedInt;
    do INNER;
    #);
  deadLocked:< Exception
    (* This exception is called when all coroutines are blocked
     * and none are waiting for I/O.
     *)
    (#
    do INNER;
       (if not continue then
           'BasicSystemEnv: All coroutines blocked on semaphores.'
             -> msg.append;
       if);
    #);
  conc:
    (# start:
         (# s: ^|system
         enter s[]
```

```
            ...
            #);
          concPriv: @...
      do INNER; ...;
      #);
    alt:
      (* Same as conc as a consequence of non−preemtive scheduling.
       *)
      conc (# #);

    (* ATTRIBUTES FOR EVENT−DRIVEN WINDOWING ENVIRONMENTS
     *
     * These attributes are only used when combining SystemEnv with
     * an event−driven windowing environment. This demands an
     * alternative implementation than the standard SystemEnv
     * implementation. See the file: xsystemenv.bet
     *)
    windowEnvType:< Object;
    theWindowEnv: ^windowEnvType;
    setWindowEnv:< Object;

    (* PRIVATE
     *
     * Everything below is in principle private implementation stuff.
     *)
    private: @ ...;
    BasicScheduler: ...;
    theActive: ^|sysHead;
    ProtectedInt: IntegerObject
      (* Used in implementation of conc. *)
      (# mutex: @semaphore;
         atZero: @semaphore;
         dec:
           (#
           do mutex.P; (if (value−1−>value)=0 then atZero.V if); mutex.V;
           #);
         waitForZero: (# do atZero.P #);
         init: (# enter value do mutex.V #);
      #);
    initBeforeScheduler:<
      (* Called before the scheduler is activated and before
       * setWindowEnv and the systemenv INNER is called.
       *)
      Object;
  do ...;
    INNER
  #);

cyclicElm:
  (# s: ^|SysHead;
     next, prev: ^cyclicElm;
     due: @Real
       (* due is used by sleepingQueue. If zero, this element is
        * currently not in a sleepingQueue.
        *)
  #);
cyclicQueue:
  (# onDelete:< Object;
     onDel: @onDelete;
     onInsert:< Object;
     onIns: @onInsert;
     first, freeList: ^cyclicElm;
     insert: @
       (# s: ^|sysHead; new: ^cyclicElm;
       enter s[]
       ...
```

```
      exit new[]
      #);
    append: @
      (# elm: ^cyclicElm;
      enter elm[]
      ...
      #);
    prepend: @
      (# elm: ^cyclicElm;
      enter elm[]
      ...
      #);
    insertBefore: @
      (# new, old: ^cyclicElm;
      enter (new[],old[])
      ...
      #);
    getFirst: @
      (# elm: ^cyclicElm;
      ...
      exit elm[]
      #);
    delete: @
      (# elm: ^cyclicElm;
      enter elm[]
      ...
      exit elm[]
      #);
    remove: @
      (* elm should not be reused after remove. Use delete instead.
       *)
      (# elm: ^cyclicElm; s: ^|sysHead;
      enter elm[]
      ...
      exit s[]
      #);
    scan:
      (# current: ^cyclicElm;
      ...
      #);
    size: @Integer;
  #);
SysHead:
  (# shstatus: @Integer;
     lc: ^Object;     (* Last errorCatcher for distribution errors. *)
     ce: ^cyclicElm;  (* ce,q <> none => this(sysHead) is ce in q.  *)
     q: ^cyclicQueue;
  do INNER
  #);
(* SysHead.shstatus values: *)
SE_RUNNING:  (# exit 1 #);  (* Current system.       *)
SE_WAITING:  (# exit 2 #);  (* Blocked on semaphore. *)
SE_READY:    (# exit 3 #);  (* Ready to run.         *)
SE_SLEEPING: (# exit 4 #);  (* Sleeping.             *)
SE_KILLED:   (# exit 5 #)
```

13.2 Betaenv Interface

```
LIB_DEF 'betaenv' '../lib';
BODY 'private/betaenvbody';
(*
 * COPYRIGHT
 *       Copyright (C) Mjolner Informatics, 1984-2003
 *       All rights reserved.
 *
 * This fragment implements the very basic patterns, utilized by most
 * BETA programs
 *)
-- betaenv: descriptor --
(# <<SLOT lib: attributes>>;
   (*****************************************************************)
   (* The simple patterns for simple values and variables.  These
    * simple patterns are treated special by the compiler.
    *)
   integer: (* 32 bit signed long *) (# #);
   shortInt: (* 16 bit unsigned half *) (# #); (* do not use shortInt anymore,
                                                * use int16u instead.
                                                *)
   char: (* 8 bit unsigned byte *) (# #);
   boolean: (* 8 bit unsigned byte, values 0 or 1 *) (# #);
   false: boolean (* 8 bit unsigned byte with value 0 *) (# #);
   true: boolean (* 8 bit unsigned byte with value 1 *) (# #);
   real: (* double precision floating point number *) (# #);
   real32: (* single precision real for bytecode only *) (# #);
   int8: (* 8 bit signed integer *) (# #);
   int8u: (* 8 bit unsigned integer *) (# #);
   int16: (* 16 bit signed integer *) (# #);
   int16u: (* 16 bit unsigned integer.
            * int16u will eventually replace shortInt *) (# #);
   int32: (* 32 bit signed integer
           * int32 is semantically identical to integer *) (# #);
   int32u: (* 32 bit unsigned integer *) (# #);

   (* int64 and int64u are NOT yet implemented;
    * the compiler allows variables of these types,
    * but no operations, including assignment,
    * are implemented, so don't use them.
    *)
   int64: (* 64 bit signed integer *) (# #);
   int64u: (* 64 bit unsigned integer *) (# #);

   (* The pattern wchar is for experimenting with implementing
    * support for the UniCode character set. The name wchar
    * is preliminary. wchar is semantically identical to int16u.
    * Patterns wcharValue and wcharObject have also been introduced below
    *)
   wchar: (# #); (* 16 bit unsigend integer *)

   (* The pattern COM is a general super pattern for
    * objects that may used with Microsoft COM
    *)
   COM: (# #);

   (* Holder is general superpattern for holder-patterns used for
    * parameters in COM.
    *)
   Holder: (# adr: @integer #);

   object: (* General superpattern *)
     (# _struc:
```

```
        (* Exit a pattern reference for THIS(Object).
         * Is now obsolete: the new form obj## is preferred
         * to the old form obj.struc
         *)
        (#
        exit this(object)##
        #);
      _new:
        (* returns a new object, that is qualified exactly
         * as THIS(object)
         *)
        (# newObj: ^object; oType: ##object
        do this(object)##->oType##; &oType[]->newObj[]; INNER _new
        exit newObj[]
        #);
      _state:
        (* Pattern _state is for experimental purpose only
         * and using it may give undefined results
         *)
        (# S: ##object
        enter S##
        ...
        #)
  do INNER object
  #);
(* idx *)

(* The following patterns define 'real' patterns corresponding to
 * the predefined simple patterns
 *)
integerValue:  (# value: @integer do INNER integerValue exit value #);
integerObject: integerValue(# enter value do INNER integerObject #);

charValue:     (# value: @char do INNER charValue exit value #);
charObject:    charValue(# enter value do INNER charObject #);

wcharValue:    (# value: @wchar do INNER wcharValue exit value #);
wcharObject:   wcharValue(# enter value do INNER wcharObject #);

booleanValue:  (# value: @boolean do INNER booleanValue exit value #);
trueValue:     booleanValue(# do true->value; INNER trueValue #);
falseValue:    booleanValue(# do false->value; INNER falseValue #);

booleanObject: booleanValue(# enter value do INNER booleanObject #);
trueObject:    booleanObject(# do true->value; INNER trueObject #);
falseObject:   booleanObject(# do false->value; INNER falseObject #);

realValue: (# value: @real do INNER realValue exit value #);
realObject: realValue(# enter value do INNER realObject #);

textValue: (# value: ^text do INNER textValue exit value[] #);
textObject: textValue(# enter value[] do INNER textObject #);

(****  Integer limits *******************************************)
MaxInt8:   (# exit 0x7f #);
MinInt8:   (# exit 0x80 #);
MaxInt8u:  (# exit 0xff #);
MinInt8u:  (# exit 0x00 #);

MaxInt16:  (# exit 0x7fff #);
MinInt16:  (# exit 0x8000 #);
MaxInt16u: (# exit 0xffff #);
MinInt16u: (# exit 0x0000 #);

MaxInt32:  (# exit 0x7fffffff #);
MinInt32:  (# exit 0x80000000 #);
```

```
MaxInt32u: (# exit 0xffffffff #);
MinInt32u: (# exit 0x00000000 #);

MaxInt:    (# exit MaxInt32 #);
MinInt:    (# exit MinInt32 #);

MaxReal:   (# exit 1.797693134862315E+308 #);
MinReal:   (# exit 2.225073858507201E−308 #);

(****  Implementation dependent constants ************************)
infReal: (* Returns the real value 'Infinity' *)
  realValue(# ... #);

(*****  Functional patterns ************************************)
min: (* Returns the minimum of 2 integers *)
  (# a,b: @integer
  enter (a,b)
  do (if (a < b) then a−>b if)
  exit b
  #);
max: (* Returns the maximum of 2 integers *)
  (# a,b: @integer
  enter (a,b)
  do (if (a < b) then b−>a if)
  exit a
  #);
abs: (* Returns the absolute value of an integer *)
  (# n: @integer
  enter n
  do (if (n < 0) then −n−>n if)
  exit n
  #);

(*****  Simple standard input/output patterns *********************)
keyboard, screen: ^stream;
get:
  (# ch: @char;
     getC: ^stream.get
  do (if getC[]=NONE then &keyboard.get[]−>getC[] if);
     getC−>ch;
     INNER;
  exit ch
  #);
put:
  (# ch: @char;
     putC: ^stream.put
  enter ch
  do (if putC[] = NONE then &screen.put[] −> putC[] if);
     INNER;
     ch−>putC
  #);
newline:
  (# newL: ^stream.newline
  do (if newL[] = NONE then &screen.newline[] −> newL[] if);
     INNER;
     newL
  #);
putint:
  (# i: @integer;
     putI: ^stream.putInt
  enter i
  do (if puti[] = NONE then &screen.putint[] −> putI[] if);
     INNER;
     i−>putI;
  #);
puttext:
```

```
  (# t: ^text;
     putT: ^stream.puttext;
  enter t[]
  do (if putT[] = NONE then &screen.puttext[] −> putT[] if);
     INNER;
     t[]−>putT;
  #);
putline:
  (# t: ^text;
     putL: ^stream.putline;
  enter t[]
  do (if putL[] = NONE then &screen.putline[] −> putL[] if);
     INNER;
     t[]−>putL;
  #);
getint:
  (# i: @integer;
     getI: ^stream.getInt;
  do (if getI[] = NONE then &keyboard.getint[] −> getI[] if);
     INNER;
     getI −> i;
  exit i
  #);
getNonBlank:
  (# ch: @char;
     getNB: ^stream.getNonBlank;
  do (if getNB[] = NONE then &keyboard.getNonBlank[] −> getNB[] if);
     INNER;
     getNB −> ch;
  exit ch
  #);
scanAtom:
  (# scanA: ^stream.scanAtom;
  do (if scanA[] = NONE then &keyboard.scanAtom(# do INNER scanAtom #)[] −> scanA[] if);
     scanA;
  #);
getAtom:
  (# t: ^text;
     getA: ^stream.getAtom;
  do (if getA[] = NONE then &keyboard.getAtom[] −> getA[] if);
     INNER;
     getA −> t[];
  exit t[]
  #);
getline:
  (# t: ^text;
     getL: ^stream.getLine;
  do (if getL[] = NONE then &keyboard.getline[] −> getL[] if);
     INNER;
     getL −> t[];
  exit t[]
  #);


(*****  Control patterns ****************************************)
forTo: (* for 'inx' in [low:high] do INNER forTo *)
  (# low, high, inx: @integer;
  enter (low, high)
  ...
  #);
cycle: (* Executes INNER forever *)
  (# ... #);
loop:
  (# while:< booleanValue(# do true−>value; INNER while #);
     until:< booleanValue;
     whilecondition: @while;
```

```
      untilcondition: @until;
    ...
    #);
  qua:
    (* Pattern replacing the BETA language construct QUA.  To be
     * used as 't1[]->qua(# as::< Tn #)->t2[]'.  The 'qua' pattern
     * checks, whether 't1' is qualified by 'Tn'.  If not, the
     * 'quaError' exception is invoked.  Otherwise, a reference
     * qualified by 'Tn', and referring to the same object as 't1[]'
     * is referring, is returned.
     *)
    (# as:< object; R: ^object; thisObj: ^as;
       quaError:< exception
         (# do 'Qualification error'->msg.append; INNER quaError #)
    enter R[]
    ...
    exit thisObj[]
    #);

  (*****  Stream patterns ****************************************)
  stream:
    (# <<SLOT streamLib: attributes>>;
       length:< integerValue (* returns the length of THIS(stream) *)
         (#
         do -1->value; INNER length
         #);
       position: (* current position of THIS(stream) *)
         (#
         enter setPos
         exit getPos
         #);
       eos:< (* returns 'true' if THIS(stream) is at end-of-stream *)
         booleanValue;
       reset: (* sets 'position' to zero *)
         (#
         do 0->setPos
         exit THIS(stream)[]
         #);
       peek:< (* looks at the next character of THIS(stream) *)
         (# ch: @char
         do INNER peek
         exit ch
         #);
       get:< (* reads a character from THIS(stream) *)
         (# ch: @char
         do INNER get
         exit ch
         #);
       getNonBlank:
         (* Reads first non-whitespace character from THIS(stream).
          * If called at end-of-stream the character 'ascii.fs' is
          * returned
          *)
         (# ch: @char;
            skipblanks: @scanWhiteSpace;
            testEOS: @EOS;
            getCh: @get;
         ...
         exit ch
         #);
       getint: integerValue
         (* Reads an integer: skips whitespace characters and
          * returns the following digits.
          *
          * See numberio.bet for more numerical output operations
          *)
```

```
  (# syntaxError:< streamException
       (#
       ...
       #);
     geti: @...
  do geti; INNER getint
  #);
getAtom:<
  (* Returns the next atom (i.e. sequence of non−white
   * characters − skipping leading blanks)
   *)
  (# txt: ^text;
  do &text[]−>txt[]; INNER getAtom;
  exit txt[]
  #);
getline:<
  (* Reads a sequence of characters until nl−character
   * appears and returns the characters read.
   *)
  (# txt: ^text;
     missing_newline:< Object
       (* Called if last line of THIS(Stream) is
        * not terminated by a newline character.
        *);
  do &text[]−>txt[]; INNER getline
  exit txt[]
  #);
asInt:
  (* converts THIS(text) to an integer value, ignoring
   * leading and trailing whitespace.  See numberio.bet for
   * more numerical conversion operations.
   *)
  (# i: @integer;
     syntaxError:< streamException
       (#
       ...
       #);
  ...
  exit i
  #);
put:< (* writes a character to THIS(stream) *)
  (# ch: @char
  enter ch
  do INNER put
  exit THIS(stream)[]
  #);
newline: (* writes the nl−character *)
  (#
  do ascii.newline−>put
  exit THIS(stream)[]
  #);
putint:
  (* Writes an integer to THIS(stream); The format may be
   * controlled by the 'signed', 'blankSign', 'width',
   * 'adjustLeft' and 'zeroPadding' variable attributes.
   * 'width' is extended if it is too small.  Examples:
   * '10−>putint' yields: '10'; '10*pi−>putint(# do 10−>width;
   * true−>adjustLeft #)' yields: '10 '; and '10−>putint(# do
   * 10−>width; true−>zeroPadding #)' yields: '0000000010'.
   *
   * See numberio.bet for more numerical output operations
   *)
  (# n: @integer;
     signed: @boolean
       (* If integer is positive, a '+' will always be
        * displayed
```

```
        *);
      blankSign: @boolean
        (* If integer is positive, a ' ' space is displayed as
         * the sign.  Ignored if 'signed=true'
         *);
      width: @integer
        (* Minimum width *);
      adjustLeft: @boolean
        (* Specifies if the number is to be aligned left or
         * right, if padding of spaces is necessary to fill up
         * the specified width.
         *);
      zeroPadding: @boolean
        (* width is padded with leading zero instead of
         * spaces.  Ignored if 'adjustLeft=true'
         *);
      format:< (# do INNER format #);
      puti: @...
  enter n
  do 1->width; format; INNER putint; puti
  exit THIS(stream)[]
  #);
puttext:< (* Writes a text to THIS(stream). *)
  (# txt: ^text
  enter txt[]
  do (if txt[]<>NONE then INNER puttext if)
  exit THIS(stream)[]
  #);
putline:
  (* 'puttext' followed by 'newline' *)
  (# T: ^text; putT: @puttext; newL: @newline
  enter T[]
  do INNER putline; T[]->putT; newL
  exit THIS(stream)[]
  #);
scan:
  (* Scan chars from current position in THIS(stream) while
   * '(ch->while)=true'; perform INNER for each char being
   * scanned
   *)
  (# while:<
       (# ch: @char; value: @boolean
       enter ch
       do true->value; INNER while
       exit value
       #);
      ch: @char;
      whilecondition: @while;
      testEOS: @EOS;
      getPeek: @peek;
      getCh: @get;
  ...
  exit THIS(stream)[]
  #);
scanWhiteSpace: scan
  (* Scan whitespace characters *)
  (# while::< (# do ch->ascii.isWhiteSpace->value #)
  do INNER scanWhiteSpace
  exit THIS(stream)[]
  #);
scanAtom:
  (* Scan until first non-whitespace char.  Scan the next
   * sequence of non-whitespace chars.  Stop at first
   * whitespace char.  For each non-whitespace char an INNER
   * is performed. Usage: 'scanAtom(# do ch-><destination> #)'
   *)
```

```
        (# ch: @char;
        ...
        exit THIS(stream)[]
        #);
      scanToNl:
        (* Scan all chars in current line including newline char *)
        (# ch: @char; getCh: @get;
           missing_newline:< Object
             (* Called if last line of THIS(Stream) is
              * not terminated by a newline character.
              *);
        ...
        exit THIS(stream)[]
        #);
      streamException: exception
        (# do INNER streamException #);
      EOSerror:< streamException
        (* Raised from 'get' and 'peek' when attempted to read past
         * the end of the stream.
         *)
        (#
        do 'Attempt to read past end−of−stream'−>msg.putline;
           INNER EOSerror
        #);
      otherError:< streamException
        (* Raised when some other kind of stream error apart from
         * the one mentioned above occurs.
         *);
      getPos:< (* returns current position of THIS(Stream) *)
        integerValue;
      setPos:< (* sets current position in THIS(stream) to 'p' *)
        (# p: @integer
        enter p
        do INNER setPos
        exit THIS(stream)[]
        #)
    #); (* pattern stream *)

  (*****  Text pattern ********************************************)
  text: stream
    (* A text is a sequence of characters.  Let 'T: @text'. The
     * range of 'T' is '[1,T.length]'.  A text can be initialized by
     * executing 'T.clear' or by assigning it another (initialized)
     * text.  A text−constant has the form 'foo'.  The 'text' pattern
     * is primarily intended for small texts but there is no upper
     * limit in the size. However, most of the operations becomes
     * less efficient with larger texts.
     *)
    (# <<SLOT textLib: attributes>>;
       length::< (* Returns the length of THIS(text) *)
        (# do lgth−>value; INNER length #);
       eos::<
        (# ... #);
       empty:
        (# exit (lgth = 0) #);
       clear: (* Sets the length and position of THIS(text) to zero *)
        (#
        do 0−>pos−>lgth
        exit THIS(text)[]
        #);
       equal: booleanValue
        (* Tests if THIS(text) is equal to the entered text.  If
         * 'NCS' is further bound to 'trueObject', the comparison
         * will be done Non Case Sensitive.
         *)
        (# txt: ^text;
```

```
     NCS:< booleanObject
  enter txt[]
  ...
  #);
equalNCS: equal
  (* As 'equal', except the the comparison will be done Non
   * Case Sensitive
   *)
  (# NCS:: trueObject #);
less: booleanValue
  (* Tests whether the entered text 'T1[1: length]' is less
   * than 'THIS(text)[1: T1.length]'.  The lexicographical
   * ordering is used.
   *)
  (# T1: ^text
  enter T1[]
  ...
  #);
greater: booleanValue
  (* Tests whether the entered text 'T1[1: length]' is
   * greater than 'THIS(text)[1: T1.length]'.  The
   * lexicographical ordering is used.
   *)
  (# T1: ^text
  enter T1[]
  ...
  #);
peek::<
  (* Returns the character at current position; does not
   * update 'position'
   *)
  (# ... #);
get::<
  (* Returns the character at current position; increments
   * 'position'
   *)
  (# ... #);
inxGet: charValue
  (* Returns the character at position 'i' *)
  (# i: @integer;
     iget: @...
  enter i
  do iget
  #);
getAtom::<
  (* Returns the next atom (i.e. sequence of non−white
   * characters − skipping leading blanks)
   *)
  (# ... #);
getline::<
  (* Reads a sequence of characters until nl−character
   * appears and returns the characters read.
   *)
  (# ... #);
put::<
  (* writes the character 'ch' at current position in
   * THIS(text); increments 'position'
   *)
  (# ... #);
inxPut:
  (* Replaces the character at position 'i' *)
  (# ch: @char;
     i: @integer;
     iput: @...
  enter (ch,i)
  do iput
```

```
  exit THIS(text)[]
  #);
puttext::<
  (# ... #);
append:
  (* Appends a text to THIS(text); does not change 'position'
   *)
  (# T1: ^text
  enter T1[]
  ...
  exit THIS(text)[]
  #);
prepend:
  (* Inserts the text in 'T1' in front of THIS(text); updates
   * current position to 'position+T1.length' if 'position>0'
   *)
  (# T1: ^text
  enter T1[]
  ...
  exit THIS(text)[]
  #);
insert:
  (* Inserts a text before the character at position 'inx'.
   * Note: inx<1 means inx=1; inx>length means inx=length+1.
   * If 'position>=inx' then 'position+T1.length->position'.
   *)
  (# T1: ^text;
     inx: @integer
  enter (T1[],inx)
  ...
  exit THIS(text)[]
  #);
delete:
  (* Deletes THIS(text)[i: j]; updates current position:
   *      i<=position<j => i-1->position
   *      j<=position   => position-(j-i+1)->position
   *)
  (# i,j: @integer;
     deleteT: @...
  enter (i,j)
  do deleteT
  exit THIS(text)[]
  #);
makeLC: (* Converts all characters to lower case *)
  (# ...
  exit THIS(text)[]
  #);
makeUC:
  (* Converts all characters to upper case *)
  (# ...
  exit THIS(text)[]
  #);
sub:
  (* Returns a copy of THIS(text)[i:j].  If 'i<1', 'i' is
   * adjusted to 1. If 'j>length', 'j' is adjusted to
   * 'length'.  If (after adjustment) 'i>j', an empty text is
   * returned.
   *)
  (# i,j: @integer; T1: ^text;
     subI: @...
  enter (i,j)
  do subI
  exit T1[]
  #);
copy:
  (# T1: ^text;
```

```
      copyI: @...
    do copyI
    exit T1[]
    #);
  scanAll:
    (* Scans all the elements in THIS(text).  For 'ch' in '[1:
     * THIS(text).length]' do INNER
     *)
    (# ch: @char
    do (for i: lgth repeat T[i]->ch; INNER scanAll for)
    exit THIS(text)[]
    #);
  find:
    (* find all occurrences of the character 'ch' in
     * THIS(text), executing INNER for each occurrence found,
     * beginning at 'THIS(text).position'.  'inx' will contain
     * the position of each 'ch' in THIS(text).  If 'NCS' is
     * further bound to 'trueObject', the comparison will be
     * done Non Case Sensitive.  If 'from' is further bound, the
     * search will begin at position 'from'.
     *)
    (# ch: @char;
       inx: @integer;
       NCS:< booleanObject;
       from:< integerObject(# do pos->value; INNER from #)
    enter ch
    ...
    exit THIS(text)[]
    #);
  findAll: find
    (* As 'find', except that the entire text will be searched.
     * Replaces 'findCh' in previous versions of betaenv (v1.4
     * and earlier)
     *)
    (# from:: (# do 0->value #)
    do INNER findAll
    #);
  findText:
    (* find all occurrences of the 'txt' in THIS(text),
     * executing INNER for each occurrence found, beginning at
     * 'THIS(text).position'.  'inx' will contain the position
     * of the first character of each occurrence found
     * THIS(text).  If 'NCS' is further bound to 'trueObject',
     * the comparison will be done Non Case Sensitive.  If
     * 'from' is further bound, the search will begin at
     * position 'from'.
     *)
    (# txt: ^text;
       inx: @integer;
       NCS:< booleanObject;
       from:< integerObject(# do pos->value; INNER from #)
    enter txt[]
    ...
    exit THIS(text)[]
    #);
  findTextAll: findText
    (* As 'findText', except that the entire text will be
     * searched
     *)
    (# from:: (# do 0->value #)
    do INNER findTextAll
    #);
  extend:
    (* Extend THIS(text) with 'L' (undefined) chars. Notice
     * that it is only the representation of the THIS(text),
     * that is extended, the 'length' and 'position' are not
```

```
         * changed.
         *)
        (# L: @integer
        enter L do L->T.extend
        exit THIS(text)[]
        #);
      indexError:< streamException
        (* Raised from 'Check' when the index goes outside the
         * range of the text. Message: "Index error in text!".
         *)
        (# inx: @integer
        enter inx
        ...
        #);
      EOSerror::<
        (* Raised from 'get' and 'peek' when the end of the stream is
         * passed.
         *)
        (# ... #);
      otherError::<
        (* Raised when an error other than the Index-/EOSerror
         * occurs.
         *)
        (# ... #);
      setPos::<
       (# ... #);
      getPos::<
        (# do pos->value; INNER getPos #);
      (* Private attributes: !!OBS!! The 3 attributes 'T', 'lgth'
       * and 'pos' declared below MUST be the first data items
       * declared in 'stream' and 'text' since their addresses are
       * hardcoded into the compiler.
       *)
      T: [16] @char;
      lgth,pos: (* 16 is default size *) @integer;
      setT: (# enter T do T.range->lgth->pos #)
  enter setT
  exit T[1: lgth]
  #) (* Pattern text *);

(*****  ASCII character constants and attributes ******************)
ascii: @
  (# <<SLOT asciiLib: attributes>>;
     nul: (# exit 0 #);
     soh: (# exit 1 #);
     stx: (# exit 2 #);
     etx: (# exit 3 #);
     eot: (# exit 4 #);
     enq: (# exit 5 #);
     ack: (# exit 6 #);
     bel: (# exit 7 #);
     bs: (# exit 8 #);
     ht: (# exit 9 #);
     nl: (# exit 10 #);
     vt: (# exit 11 #);
     np: (# exit 12 #);
     cr: (# exit 13 #);
     so: (# exit 14 #);
     si: (# exit 15 #);
     dle: (# exit 16 #);
     dc1: (# exit 17 #);
     dc2: (# exit 18 #);
     dc3: (# exit 19 #);
     dc4: (# exit 20 #);
     nak: (# exit 21 #);
     syn: (# exit 22 #);
```

```
      etb: (# exit 23 #);
      can: (# exit 24 #);
      em: (# exit 25 #);
      sub: (# exit 26 #);
      esc: (# exit 27 #);
      fs: (# exit 28 #);
      gs: (# exit 29 #);
      rs: (# exit 30 #);
      us: (# exit 31 #);
      sp: (# exit 32 #);
      capA: (# exit 65 #);
      smalla: (# exit 97 #);
      del: (# exit 127 #);
      newline: @char; (* either 'lf' or 'cr' *)

      init: ...;
      upCase: @charObject
        (# ... #);
      lowCase: @charObject
        (# ... #);
      testChar: booleanValue
        (# ch: @char
        enter ch
        do INNER testchar
        #);
      isUpper: @testChar
        (# ... #);
      isLower: @testChar
        (# ... #);
      isDigit: @testChar
        (# ... #);
      isLetter: @testChar
        (# ... #);
      isSpace: @testChar
        (* True if 'ch' in {sp,cr,nl,np,ht,vt} *)
        (# ... #);
      isWhiteSpace: @testChar
        (* True if 'ch' is a whitespace char *)
        (# ... #);
      private: @...
  #);
(*****  Exception Patterns ************************************)
stop:
  (* Terminates program execution:
   *   termCode=normal       : normal termination;
   *   termCode=failure      : abnormal termination;
   *   termCode=failureTrace : abnormal termination with trace of
   *                           run-time stack on dump-file;
   *   termCode=dumpStack    : Trace of run-time stack on console
   *                           without termination.
   *
   * 'T' will be printed on the screen.
   *)
  (# termCode: @integer; T: ^text
  enter (termCode,T[])
  ...
  #);
normal: (# exit 0 #);
failure: (# exit -1 #);
failureTrace: (# exit -2 #);
dumpStack: (# exit -3 #);

(*****  Object Pool ******************************************)
objectPool: @
  (# <<SLOT objectPoolLib: attributes>>;
     get:
```

```
      (# type:< object;
         obj: ^type;
         exact:< booleanValue;
         init:< object(* Called if an object was created *)
      ...
      exit obj[]
      #);
    strucGet:
      (# type: ##object;
         obj: ^object;
         exact:< booleanValue;
         init:< object(* Called if an object was created *);
      enter type##
      ...
      exit obj[]
      #);
    scan:
      (* Scan through all objects in 'objectPool', (at least)
       * qualified by 'type'.
       *)
      (# type:< object;
         current: ^type;
         exact:< booleanValue;
      ...
      #);
    strucScan:
      (* Scan through all objects in 'objectPool', (at least)
       * qualified by 'type'
       *)
      (# type: ##object;
         current: ^object;
         exact:< booleanValue
      enter type##
      ...
      #);
    put:
      (* Puts a given object into 'objectPool'. If an object with
       * (at least) the qualification of the given object is
       * already present in 'objectPool', the exception
       * 'alreadyThere' is raised.
       *)
      (# obj: ^object;
         exact:< booleanValue;
         alreadyThere:< exception;
         putObj: @...
      enter obj[]
      do putObj
      #);
    private: @...;
  #);

(*****  Command line arguments **********************************)
argumentHandlerType:
  (#
    noOfArguments:<
      (* Return the number of arguments on command line.
       * The number includes the program name.
       *)
      integervalue;

    getArgByNumber:<
      (* Returns argument number argNo.
       * Number 1 is the program name,
       * number 2 is the first program argument, etc.
       *)
      (# argNo: @integer; theArg: ^text;
```

```
        enter argNo
        do INNER
        exit theArg[]
        #);
  #);

rawArgumentHandler: argumentHandlerType
  (#
     noOfArguments::
       (# ... #);
     getArgByNumber::
       (# ... #);
  #);
expandWildcardsArgumentHandler: argumentHandlerType
  (# private: @...;
     noOfArguments::
       (# ... #);
     getArgByNumber::
       (# ... #);
  #);

argumentHandler: ^argumentHandlerType;

(* Backwards compatible interface *)
noOfArguments: integervalue(# do argumentHandler.noOfArguments -> value #);
arguments:
  (# argNo: @integer; theArg: ^text;
  enter argNo
  do argNo -> argumentHandler.getArgByNumber -> theArg[]
  exit theArg[]
  #);

(* Explicit array − currently only supported by bytecode implementations *)
ArgVector: [0]^text;

(*****************************************************************)
(* External language interface: See file 'external.bet' for further
 * patterns.
 *)
External:
  (* Is only meaningful with interface to externals *)
  (# callC,callPascal,pascal,pascalTrap,callStd,
     cExternalEntry,pascalExternalEntry,stdExternalEntry: @text
  #);
cStruct:
  (* Super−pattern for describing structures which can be given
   * 'by refererence' (using the usual [] notation) to an external
   * function (e.g. a C function described as a specialization
   * of the above External pattern). See file external.bet for
   * supported operations on cStruct.
   *)
  (# <<SLOT cStructLib: attributes>>;
     (* 'R' is the bytestream containing THIS(cStruct).
      * MUST be declared as the first attribute
      *);
     R: [(byteSize−1) div 4 + 1] @integer;
     byteSize:<
       (* Number of bytes in THIS(cStruct) *)
       IntegerObject;

     BoundsExceeded:< Exception
       (* Raised if indexing outside range of R *)
       (# inx: @integer;
       enter inx
       ...
       #);
```

```
      chk: @(# inx: @integer enter inx ... #);
  #);
data:
  (* The 'data' pattern may be used for definining simple data
   * objects.  Data−objects have no 'type' information. They can
   * thus NOT be allocated dynamically in the BETA heap. They do
   * not have the overhead of extra attributes used for virtual
   * dispatch and garbage collection. One main use of data−objects
   * is as interface to external data such as 'cstruct'.  For
   * details see the manuals
   *)
  (# #);

(* Basic types used for bytecode platforms *)
ExternalClass:
  (# classname: @text;
  do INNER;
  #);
class:
  (# (*classname: @text;*) #);
proc:
  (# procname: @text; #);
static_proc: (# procname: @text #);
cons: (# procname: @text #);
static_cons: (# procname: @text #);

Structure:
  (# (* representing a descriptor sctrurure/ptn variable *)#);

doGC: (* will force a garbage collection to happen *)
  (# ... #);
machine_type:
  (* Exits a reference to a copy of a text indicating the machine
   * type in lowercase, e.g. 'sun4s', 'linux', 'nti'.
   *)
  (# T: @Text;
  ...
  exit T.copy
  #);

program: (* descriptor executed by this environment *)
  ...;
theProgram: ^|program;
theScheduler: ^|object
  (* Scheduler installed by 'basicSystemEnv' (if used in program) *);
(*************************************************************)
(* The following patterns are only used by the compiler and should
 * NOT be used for other purposes.
 *)
repetition:
  (# range: (* Returns the range of THIS(repetition) *)
       (# n: @integer
       exit n
       #);
     new:
       (* Allocates a new repetition of 'n' elements. The previous
        * elements in THIS(repetition) become inaccessible
        * hereafter
        *)
       (# n: @integer
       enter n
       #);
     extend:
       (* Extends THIS(repetition) by 'n' elements.  The existing
        * elements are retained.  The new elements are allocated
        * after the existing elements (i.e. with index from the
```

```
         * 'range+1')
         *)
        (# n: @integer
        enter n
        #)
  #);
state: (# #); (* Pattern STATE is for experimental purpose only
               * and using it may give undefined results
               *)
errorName: (# #);

exception:
  (# <<SLOT exceptionLib: attributes>>;
     msg:
       (* append text to this 'msg' variable to specify
        * the exception error message for
        * this(exception)
        *)
       @text;
     continue: @boolean
       (* the value of this variable determines the
        * control−flow behaviour of this(exception):
        *    true:  continue execution after exception
        *    false: terminate execution by calling
        *           'stop'; default
        *);
     error:
       (* used to define local exception conditions
        * which can be handled separately.  All error's
        * that are not handled separately will be
        * handled by this(exception)
        *)
       (# <<SLOT errorExceptionLib: attributes>>
       do false−>continue;
          INNER;
          '**** Error processing\n'−>msg.prepend;
          (if not continue then this(exception) if)
       #);
     _notify: error
       (* used to define local notification conditions
        * which can be handled separately.  All
        * 'notify's that are not handled separately
        * will be handled by this(exception)
        *)
       (# <<SLOT notifyExceptionLib: attributes>>
       do true−>continue; INNER
       #);
     propagate:<
       (* if further bound to trueObject, this(exception) allows
        * propagation (i.e. this(exception will _not_ terminate)
        *)
       (* This is to make exception backward compatible *)
     booleanValue;
     termCode: @integer;
       (* Arg. To pattern 'stop'; initial failureTrace *)

     ...
  #);

(* Notification is used to make the new exceptions backward compatible *)
notification: exception
  (# do true−>continue; INNER notification #);

unknown: exception
  (# <<SLOT unknownExceptionLib: attributes>>;
     original: ^object;
```

```
      do INNER
      #);

   try:
      (# <<SLOT tryExceptionLib: attributes>>;
         handler:<
           (* invoked automatically bu the try block in the search for a
            * when clause
            *)
           (# <<SLOT tryHandlerExceptionLib: attributes>>;
              when:
                (# <<SLOT tryHandlerWhenExceptionLib: attributes>>;
                   current: ^type;
                   type:< object;
                   predicate:< booleanValue;
                   continue:
                     (# ... #);
                   retry:
                     (# ... #);
                   propagate:
                     (# ... #);
                   abort:
                     (# ... #);
                do ...
                #);
              current: ^object; status: @(*private*)integer;
              private: @...
           enter current[]
           do ...
           exit status
           #);
         finally:< (# do INNER; #);
         name:< (# n: ^text do INNER exit n[] #);
         private: @...
      ...
      #);

   throw:
      (# <<SLOT throwExceptionLib: attributes>>;
         current: ^object;
         private: @...
      enter current[]
      ...
      #);

   init: ...;

   betaenvPrivate: @...;
   (*****************************************************************)

do (#
   ...
   #)
#)
```

13.3 Binfile Interface

```
ORIGIN '~beta/basiclib/file';
LIB_DEF 'binfile' '../lib';
BODY 'private/binfilebody';

(* binfile:
 *   These fragments declare attributes for direct reading/writing
 *   of various data sizes to a file. The data is written out exactly
 *   as is:
 *      64 -> aBinFile.putLong
 *   will write the number 0x00000020 to aBinFile, whereas
 *      64 -> aBinFile.putInt
 *   will write the two characters '6' (ascii 54) and '4' (ascii 52)
 *   to the file.
 *
 *   The operations putBytes and getBytes allow an arbitrary
 *   sequence of bytes to be written/read to/from a file.
 *   E.g.
 *      buffer: [1000]@char;
 *      putB: @aFile.putBytes;
 *   do ...
 *      (@@buffer[1],500) -> putB;
 *   This will write the first 500 characters from the buffer
 *   repetition to the file. NOTICE, that you must have a static
 *   instance of putBytes/getBytes when using them, since they require
 *   an address argument. If dynamic instances are used, a garbage-
 *   collection may be triggered, and the address argument would be
 *   illegal.
 *
 *   These operations are declared in FileLib, e.g. they become
 *   usable for any file, by just including this fragment file.
 *   However, on some platforms, the "binary" virtual of File
 *   MUST be further bound to TrueObject for these operations
 *   to work. The binfile pattern below adds this further binding.
 *
 *   You should remember this further binding if you are using
 *   these operations on a file, that is not a binfile.
 *
 *   Exceptions:
 *   If any of the put-operations fail, they raise the WriteError
 *   file-exception. If any of the get-operations fail, they raise
 *   the ReadError file-exception.
 *)

-- LIB: attributes --

binfile: file
  (# <<SLOT BinFileLib: attributes>>;
     binary :: trueobject
  #);

-- FileLib: attributes --
putdouble:
  (* Write binary representation of i (8 bytes) to file *)
  (# i: @real;
  enter i
  ...
  #);
putlong:
  (* Write binary representation of i (4 bytes) to file *)
  (# i: @integer;
  enter i
  ...
```

```
  #);
putshort:
  (* Write binary representation of i (2 bytes) to file *)
  (# i: @int16;
  enter i
  ...
  #);
putbyte:
  (* Write binary representation of i (1 byte) to file *)
  (# i: @char;
  enter i
  ...
  #);
getDouble:
  (* Read binary representation of i (8 bytes) from file *)
  (# i: @real;
  ...
  exit i
  #);
getLong:
  (* Read binary representation of i (4 bytes) from file *)
  (# i: @integer;
  ...
  exit i
  #);
getShort:
  (* Read binary representation of i (2 bytes) from file *)
  (# i: @int16;
  ...
  exit i
  #);
getByte:
  (* Read binary representation of i (1 byte) from file *)
  (# i: @char;
  ...
  exit i
  #);

putCharRep:
  (* Write num chars from R[1]..R[num] to file *)
  (# R: [0] @char;
     num: @integer;
  enter (R[], num)
  ...
  #);
getCharRep:
  (* Read num chars from file into a new repetition.
   * 'extra' can be used to preallocated that number of
   * chars at the end og the repetition during the same
   * operation.
    *)
  (# R: [0] @char;
     num, extra: @integer;
  enter (num, extra)
  ...
  exit R[]
  #);

(* Put and get of repetitions of SIGNED elements
 * using repetition references
 *)
putInt8Rep:
  (* Write num int8 from R[1]..R[num] to file *)
  (# R: [0] @int8;
     num: @integer;
  enter (R[], num)
```

```
  ...
  #);
getInt8Rep:
  (* Read num int8 from file into a new repetition.
   * 'extra' can be used to preallocated that number of
   * int8 at the end og the repetition during the same
   * operation.
    *)
  (# R: [0] @int8;
     num, extra: @integer;
  enter (num, extra)
  ...
  exit R[]
  #);
putInt16Rep:
  (* Write num int16s from R[1]..R[num] to file *)
  (# R: [0] @int16;
     num: @integer;
  enter (R[], num)
  ...
  #);
getInt16Rep:
  (* Read num int16s from file into a new repetition.
   * 'extra' can be used to preallocated that number of
   * int16s at the end og the repetition during the same
   * operation.
   *)
  (# R: [0] @int16;
     num, extra: @integer;
  enter (num, extra)
  ...
  exit R[]
  #);
putInt32Rep:
  (* Write num int32s from R[1]..R[num] to file *)
  (# R: [0] @int32;
     num: @integer;
  enter (R[], num)
  ...
  #);
getInt32Rep:
  (* Read num int32s from file into a new repetition.
   * 'extra' can be used to preallocated that number of
   * int32s at the end og the repetition during the same
   * operation.
   *)
  (# R: [0] @int32;
     num, extra: @integer;
  enter (num, extra)
  ...
  exit R[]
  #);

(* Put and get of repetitions of UNSIGNED elements
 * using repetition references
 *)
putInt8uRep:
  (* Write num int8u from R[1]..R[num] to file *)
  (# R: [0] @int8u;
     num: @integer;
  enter (R[], num)
  ...
  #);
getInt8uRep:
  (* Read num int8u from file into a new repetition.
   * 'extra' can be used to preallocated that number of
```

```
   * int8 at the end og the repetition during the same
   * operation.
    *)
  (# R: [0] @int8u;
     num, extra: @integer;
  enter (num, extra)
  ...
  exit R[]
  #);
putInt16uRep:
  (* Write num int16u from R[1]..R[num] to file *)
  (# R: [0] @int16u;
     num: @integer;
  enter (R[], num)
  ...
  #);
getInt16uRep:
  (* Read num int16u from file into a new repetition.
   * 'extra' can be used to preallocated that number of
   * int16s at the end og the repetition during the same
   * operation.
   *)
  (# R: [0] @int16u;
     num, extra: @integer;
  enter (num, extra)
  ...
  exit R[]
  #);
putInt32uRep:
  (* Write num int32u from R[1]..R[num] to file *)
  (# R: [0] @int32u;
     num: @integer;
  enter (R[], num)
  ...
  #);
getInt32uRep:
  (* Read num int32u from file into a new repetition.
   * 'extra' can be used to preallocated that number of
   * int32s at the end og the repetition during the same
   * operation.
   *)
  (# R: [0] @int32u;
     num, extra: @integer;
  enter (num, extra)
  ...
  exit R[]
  #);

(* Unsafe operations retained for backward compatibility *)

putBytes:
  (* Write num bytes to file from memory
   * starting at address addr.
   *)
  (# addr, num: @integer;
  enter (addr, num)
  ...
  #);
getBytes:
  (* Read in num bytes from file to memory
   * starting at address addr.
   *)
  (# addr, num: @integer;
  enter (addr, num)
  ...
  #)
```

13.4 Directory Interface

```
ORIGIN 'file';
LIB_DEF 'directory' '../lib';
BODY 'private/directorybody';
(*
 * COPYRIGHT
 *       Copyright (C) Mjolner Informatics, 1984-96
 *       All rights reserved.
 *
 *)
---- LIB: attributes ----
directory:
  (* Generalization of disk folder/directory.  Describes the list
   * aspects of directories and contains a DiskEntry item describing
   * the other properties of a directory.
   *)
  (#
     <<SLOT DirectoryLib: attributes>>;

     EntryDesc:< DiskEntry;
     entry: @EntryDesc
       (* The item holding most characterizing attributes of
        * THIS(directory)
        *);
     name: @
       (* convenient interface to entry.path *)
       (# read:
            (* Reads a directory name from the Keyboard *)
            (# do ... #);
       enter entry.path
       exit entry.path
       #);

     (* Directory exceptions *)

     DirException: Exception
       (* General directory exception *)
       (# do ...; INNER #);
     DirEntryException: DirException
       (* Error for an entry in the directory. Message:
        * "Operation failed on entry".
        *)
       (# entry: ^text enter entry[] ... #);
     NoSuchException: DirEntryException
       (* Raised on attempt to delete a file or directory that did
        * not exist in THIS(directory). Message: "Attempt to delete a
        * nonexisting entry."
        *)
       (# do ...; INNER #);
     EntryExistException: DirEntryException
       (* Raised on attempt to create a file or directory that
        * already existed in THIS(directory). Message: "Directory
        * entry already exist"
        *)
       (# do ...; INNER #);
     DirScanException: DirException
       (* Raised if a scan of THIS(directory) has failed.  Message:
        * "Scan of directory failed.", and an indication of why it
        * failed.
        *)
       (# do ...; INNER #);
     DirSearchException: DirException
       (* Raised if a find in THIS(directory) has failed.  Message:
```

```
   * "Search of directory failed.", and an indication of why it
   * failed.
   *)
  (# do ...; INNER #);
NotFoundException: DirException
  (* Raised if findEntry.select is used in findEntry.notFound,
   * or in other situations that findEntry.found[]=NONE. Message:
   * "Attempt to use 'select' in 'findEntry' when the candidate
   * was not found."
   *)
  (# do ...; INNER #);

(* Manipulations of THIS(directory) *)

touch: entry.touch
  (* If the disk entry does not exist, an empty directory will
   * be created.
   *)
  (# touchD: @...;
  do touchD
  #);
delete:
  (* Delete THIS(directory) *)
  (# nosuch:< NoSuchException
       (* Raised if there was no disk entry corresponding to
        * THIS(Directory)
        *);
     error:< entry.DiskEntryException
       (* Raised if other errors occurred *);
     deleteD: @...;
  do deleteD
  #);
createFile:
  (* Create a file named 'name' in THIS(Directory) *)
  (# name: ^text;
     newEntry: ^EntryDesc;
     exists:< EntryExistException
       (* Raised if en entry of that name already existed *);
     error:< DirEntryException
       (* Raised if other errors occurred *);
  enter name[]
  ...
  exit newEntry[]
  #);
deleteFile:
  (* Delete a file named 'name' in THIS(Directory) *)
  (# name: ^text;
     nosuch:< NoSuchException
       (* Raised if there was no disk entry in THIS(Directory)
        * named 'name'
        *);
     error:<DirEntryException
       (* Raised if other errors occured *)
  enter name[]
  ...
  #);
createDir:
  (* Create a directory named 'name' in THIS(Directory) *)
  (# name: ^text;
     newEntry: ^EntryDesc;
     exists:< EntryExistException
       (* Raised if en entry of that name already existed *);
     error:< DirEntryException
       (* Raised if other errors occurred *);
  enter name[]
  ...
```

```
  exit newEntry[]
  #);
deleteDir:
  (* Delete a directory named 'name' in THIS(Directory) *)
  (# name: ^text;
     nosuch:< NoSuchException
       (* Raised if there was no disk entry in THIS(Directory)
        * named 'name'
        *);
     error:< DirEntryException
       (* Raised if other errors occured *)
  enter name[]
  ...
  #);
noOfEntries: IntegerValue
  (* exit the number of entries in THIS(directory) *)
  (# error:<DirException;
  ...
  #);
empty: BooleanValue
  (* TRUE iff THIS(directory) is empty. Note that this does not
   * always imply NoOfEntries=0
   *)
  (# error:<DirException;
  ...
  #);
findEntry:
  (* Calls INNER if entry was found in THIS(directory), and
   * otherwise calls notFound
   *)
  (# <<SLOT DirFindLib: attributes>>;
     candidate: ^text;
     (* The name of the entry to search for *)
     foundDesc:< DiskEntry;
     (* Qualification of "found" *)
     found: ^foundDesc;
     (* Reference to entry, if found.  Notice that 'found.path'
      * is relative to THIS(directory).  The full path may be
      * obtained by 'foundFullPath' Also 'foundFullPath ->
      * found.path' may be needed before is queried for modtime
      * etc., if THIS(Directory) is not the current working
      * directory.
      *)
     foundFile:< File;
     (* Qualification of file generated in
      * select.whenfile.thefile
      *)
     foundDir:< Directory;
     (* Qualification of directory generated in
      * select.whendir.thedir
      *)
     foundFullPath: (* Fullpath of "found" *)
       (# p: ^text do ... exit p[] #);

     notfound:< (* Called if the entry was not found *)
       (# do INNER #);
     select:
       (* Used to distinguish between the various entries that
        * may be found
        *)
       (# error:< found.DiskEntryException;
          whenFile:<
            (* Called when the entry found is a file *)
            (# thefile:
                 (* Generate an instance of foundFile
                  * corresponding to the entry found. Notice that
```

```
                    * 'found' and 'f.entry' are two distinct
                    * objects; 'f.entry' has a full path, 'found'
                    * may or may not have a path relative to
                    * THIS(Directory).
                    *)
                   (# f: ^foundFile
                   do ...
                   exit f[]
                   #);
              do INNER
              #);
            whenDir:<
              (* Called when the entry found is a directory *)
              (# theDir:
                   (* Generate an instance of foundDir
                    * corresponding to the entry found. Notice that
                    * 'found' and 'd.entry' are two distinct
                    * objects; 'd.entry' has a full path, 'found'
                    * may or may not have a path relative to
                    * THIS(Directory).
                    *)
                   (# d: ^foundDir
                   do ...
                   exit d[]
                   #);
              do INNER
              #);
            whenOther:<
              (* Called when the entry found is neither a file nor
               * a directory
               *)
              (# do INNER #);
            selectImpl:< (* private *)
              (# selectedInInner: @boolean
              do ...;
                 INNER; ...;
              #)
         do selectImpl;
         #); (* select *)
       error:< DirSearchException
         (* Raised if the search fails *);
    enter candidate[]
    do ...;
    #); (* findEntry *)
  scanEntries:
    (* Calls INNER for each entry in THIS(directory) *)
    (# <<SLOT DirScanLib: attributes>>;
       longest: @integer;
       (* The length of the longest entry−name in THIS(directory)
        *)
       foundDesc:< DiskEntry;
       (* Qualification of "found" *)
       found: ^foundDesc;
       (* Reference to entry, if found.  Notice that 'found.path'
        * is relative to THIS(directory).  The full path may be
        * obtained by 'foundFullPath' Also 'foundFullPath −>
        * found.path' may be needed before is queried for modtime
        * etc., if THIS(Directory) is not the current working
        * directory.
        *)
       foundFile:< File;
       (* Qualification of file generated in
        * select.whenfile.thefile
        *)
       foundDir:< Directory;
       (* Qualification of directory generated in
```

```
          * select.whendir.thedir
          *)
        foundFullPath: (* Fullpath of "found"  *)
          (# p: ^text do ... exit p[] #);
        select:
          (* Used to distinguish between the various entries that
           * may be found
           *)
          (# error:< found.DiskEntryException;
             whenFile:<
               (* Called when the entry found is a file *)
               (# thefile:
                    (* Generate an instance of foundFile
                     * corresponding to the entry found. Notice that
                     * 'found' and 'f.entry' are two distinct
                     * objects; 'f.entry' has a full path, 'found'
                     * may or may not have a path relative to
                     * THIS(Directory).
                     *)
                    (# f: ^foundFile
                    do ...
                    exit f[]
                    #);
               do INNER
               #);
             whenDir:<
               (* Called when the entry found is a directory *)
               (# theDir:
                    (* Generate an instance of foundDir
                     * corresponding to the entry found. Notice that
                     * 'found' and 'd.entry' are two distinct
                     * objects; 'd.entry' has a full path, 'found'
                     * may or may not have a path relative to
                     * THIS(Directory).
                     *)
                    (# d: ^foundDir
                    do ...
                    exit d[]
                    #);
               do INNER
               #);
             whenOther:<
               (* Called when the entry found is neither a file nor
                * a directory
                *)
               (# do INNER #);
             selectImpl:< (* private *)
               (# selectedInInner: @boolean
               do ...;
                  INNER; ...;
               #)
          do selectImpl;
          #); (* select *)
        error:< DirScanException
          (* Raised if the scan fails *);
        (* idx− *) (* idx− *)
     do ...;
     #); (* scanEntries *)
   private: @...;
#)
```

13.5 External Interface

```
ORIGIN 'betaenv';
LIB_DEF 'external' '../lib';
-- CStructLib: attributes---
(*
 * COPYRIGHT
 *       Copyright Mjolner Informatics, 1992-99
 *       All rights reserved.
 *
 ****** Patterns for external interface *****
 *
 * In CStructLib, the operations on a cStruct are defined.
 * The pattern ExternalRecord is an interface to e.g. CStruct objects
 * allocated from C or other external languages.
 *)
GetByte:
  (# byteno: @int32;
  enter byteno->chk
  exit byteNo -> R.%getByte
  #);
PutByte:
  (# val: @int8;
     byteno: @int32;
  enter(byteno,val)
  do byteno->chk;
     (val,byteno) ->R.%putbyte
  #);
GetShort:
  (# byteno: @int32;
  enter byteno->chk
  exit (byteno div 2) ->R.%getShort
  #);
PutShort:
  (# val: @int16;
     byteno: @int32;
  enter (byteno,val)
  do byteno->chk;
     (val,byteno div 2) -> R.%putShort
  #);
GetSignedShort:
  (# byteno: @int32;
  enter byteno->chk
  exit (byteno div 2) ->R.%getSignedShort
  #);
GetLong:
  (# byteno: @int32;
  enter byteno->chk
  exit (byteno div 4) ->R.%getLong
  #);
PutLong:
  (# val: @int32;
     byteno: @int32
  enter (byteno,val)
  do byteno->chk;
     (val,byteno div 4) ->R.%putLong
  #);


CStructField:
  (* Used for declaring CStruct fields *)
  (# pos:< IntegerObject;
     p: @pos;
  #);
```

```
Byte: CStructField
  (# set: @(# val: @int8 enter val do (val,p) −>R.%putByte #);
  enter set
  exit p −>R.%getByte
  #);
Short: CStructField
  (# set: @(# val: @int16 enter val do (val,p div 2) −>R.%putShort #);
  enter set
  exit (p div 2) −>R.%getShort
  #);
SignedShort: CStructField
  (# set: @(# val: @int16 enter val do (val,p div 2) −>R.%putShort #);
  enter set
  exit (p div 2) −>R.%getSignedShort
  #);
Long: CStructField
  (# set: @(# val: @int32 enter val do (val,p div 4) −>R.%putLong #);
  enter set
  exit (p div 4)−>R.%getLong
  #);


−−LIB: attributes−−

(* Various C functions *)
malloc: External
  (# size: @integer;
     ptr: @integer;
  enter size
  exit ptr
  #);
memcpy: external
  (# s1, s2, nbytes: @int32;
  enter (s1, s2, nbytes)
  exit s1
  #);

ExternalRecord:
  (* Super−pattern for describing externally allocated record−structures.
   * A call to e.g. a C routine may often return a pointer to a CStruct.
   * By assigning such a pointer to the ptr−field of an externalRecord
   * object it is possible to interface to such an external CStruct.
   * Notice the difference to the CStruct pattern, which is typically used
   * to *provide* external code with a structure allocated in BETA.
   *)
  (# ptr: @int32; (* pointer to the externally allocated record *)
     GetByte:
       (# byteno: @int32;
       enter byteno
       exit %getByteAt (ptr+byteno)
       #);
     PutByte:
       (# val: @int8;
          byteno: @int32;
       enter(byteno,val)
       do val %putByteAt (ptr+byteno)
       #);
     GetShort:
       (# byteno: @int32;
       enter byteno
       exit %getShortAt (ptr+byteno)
       #);
     GetSignedShort:
       (# byteno: @int32;
       enter byteno
       exit %getSignedShortAt (ptr+byteno)
```

```
      #);
    PutShort:
      (# val: @int16;
         byteno: @int32;
      enter(byteno,val)
      do val %putShortAt (ptr+byteno)
      #);
    GetLong:
      (# byteno: @int32
      enter byteno
      exit %getLongAt (ptr+byteno)
      #);
    PutLong:
      (# val: @int32;
         byteno: @int32
      enter(byteno,val)
      do val %putLongAt (ptr+byteno)
      #);

    ExternalRecordField:
      (* For declaring fields in ExternalRecords *)
      (# pos:< IntegerValue;
         p: @pos;
      #);
    Byte: ExternalRecordField
      (# set: @(# val: @int8 enter val do val %putByteAt (ptr+p) #)
      enter set
      exit %getByteAt (ptr+p)
      #);
    Short: ExternalRecordField
      (# set: @(# val: @int16 enter val do val %putShortAt (ptr+p) #);
      enter set
      exit %getShortAt (ptr+p)
      #);
    SignedShort: ExternalRecordField
      (# set: @(# val: @int16 enter val do val %putShortAt (ptr+p) #);
      enter set
      exit %getSignedShortAt (ptr+p)
      #);
    Long: ExternalRecordField
      (# val: @int32;
         set: @(# enter val do val %putLongAt (ptr+p) #);
      enter set
      exit %getLongAt (ptr+p)
      #);
    DoubleLong: ExternalRecordField
      (# v1,v2: @int32;
         set: @(# enter(v1,v2)
               do v1 %putLongAt (ptr+p);
                  v2 %putLongAt (ptr+p+4);
               #);
      enter set
      exit (%getLongAt (ptr+p), %getLongAt (ptr+p+4))
      #);
  enter ptr
  do INNER
  exit ptr
  #) (* ExternalRecord *);

ExternalRepetition: ExternalRecord
  (# elementSize:<integerValue;
     init:
      (# enter new #);
     new:
      (# newrange: @int32
      enter newrange
```

```
  do free;
     newrange->range;
  #);
extend:
  (# extra: @int32;
     newptr, newrange, size: @int32;

  enter extra
  do (* is realloc available on all platforms? *)
     elementSize->size;
     (*'EXTEND: elementsize: ' -> puttext;
      * elementsize -> putint; ', range.r: ' -> puttext;
      * range.r->putint; newline;
      *)
     range.r+extra -> newrange;
     size*newrange -> malloc -> newptr;
     (if newPtr=0 then
         'ExternalRepetition.extend: malloc failed' -> screen.putline;
      else
         (if ptr<>0 then
             (* 'memcpy ' -> puttext; size*range.r->putint;
              * ' bytes.'->putline;
              *)
             (newptr, ptr, size*range.r) -> memcpy;
             free;
         if);
         newptr->ptr;
         newrange -> range.r;
     if);
  #);
range: @
  (# r: @int32;
  enter (# r2: @int32 enter r2 do r2->r->extend #)
  exit r
  #);
free:
  (# cfree: External
       (# ptr: @integer
       enter ptr
       do 'free' -> callC;
       #);
  do (if ptr<>0 then ptr -> cfree; 0->ptr; if);
     0 -> range.r;
  #);
inxPut:
  (* Only for elementSize in {1,2,4}. Inx zero based. No index check *)
  (# elm, inx: @int32;
  enter (elm, inx)
  do (if elementSize
      // 1 then elm %putByteAt (ptr+inx)
      // 2 then elm %putShortAt (ptr+2*inx)
      // 4 then elm %putLongAt (ptr+4*inx)
      else
         'ExternalRepetition.inxPut: Not for elementSize='->screen.puttext;
         elementSize->screen.putInt;
         screen.newline;
     if);
  #);
inxCopy:
  (* Copy elementSize bytes from "element" (which is assumed to
   * point to data of the same type as the external repetition elements)
   * to index number inx (counting from zero) in the external
   * repetition.
   * Can be used for any elementSize.
   *)
  (# inx: @int32;
```

```
            element: ^data;
            n: @integer;
          enter (inx, element[])
          do elementSize -> n;
            (ptr+inx*n, %getLongAt(@@element), n) -> memcpy
          #);
        inxGet:
          (* Only for elementSize in {1,2,4}. Inx zero based. No index check *)
          (# elm, inx: @int32;
          enter (inx)
          do (if elementSize
              // 1 then (%getByteAt (ptr+inx)) -> elm
              // 2 then (%getShortAt (ptr+2*inx)) -> elm;
              // 4 then (%getLongAt (ptr+4*inx)) -> elm;
              else
                 'ExternalRepetition.inxGet: Not for elementSize='->screen.puttext;
                 elementSize->screen.putInt;
                 screen.newline;
            if);
          exit elm
          #);
    #);

makeCBF: External
    (* Call this external to install a callback and get
     * an int32 pointer to it.
     *)
    (# pat: ##External;
       cb: @int32;
    enter pat##
    exit cb
    #);

freeCBF: External
    (* Call this external with an int32 pointer to an installed
     * callback (obtained via MakeCBF) when it is certain that the
     * callback will NOT be called again.
     * This will free BETA heap space associated with the callback.
     *)
    (# cbf: @int32;
    enter cbf
    #)
```

13.6 File Interface

```
ORIGIN 'betaenv';
LIB_DEF 'file' '../lib';
BODY 'private/filebody';
(*
 * COPYRIGHT
 *       Copyright (C) Mjolner Informatics, 1984-96
 *       All rights reserved.
 *
 * The BETA interface to disk-entries in a hierarchic file system
 * files, and directories is organised as follows:
 *
 *   DiskEntry:     Machine independent interface to entries like
 *                  file and directories on the disk
 *                  (file 'file.bet').
 *   UnixEntry:     Unix specific specialization of DiskEntry
 *                  (file 'unixfile.bet').
 *   MacEntry:      Macintosh specific specialization of DiskEntry
 *                  (file 'macfile.bet').
 *
 *   File:          Machine independent interface to disk files. Is
 *                  a specialization of Stream (file 'betaenv.bet'),
 *                  and contains a DiskEntry (file 'file.bet').
 *   UnixFile:      Unix specific specialization of File, which
 *                  contains a UnixEntry (file 'unixfile.bet').
 *   MacFile:       Macintosh  specific specialization of File,
 *                   which contains a MacEntry (file 'macfile.bet').
 *
 *   Directory:     Machine independent interface to
 *                  directories/folders. Contains a DiskEntry
 *                  (file 'directory.bet').
 *   UnixDirectory: Unix specific specialization of Directory,
 *                  which contains a UnixEntry
 *                  (file 'unixdirectory.bet').
 *   MacDirectory:  Macintosh specific specialization of Directory,
 *                  which contains a MacEntry
 *                  (file 'macdirectory.bet').
 *)
-- LIB: Attributes --
DiskEntry:
  (* Pattern describing various attributes of disk-entries like files
   * and directories in a hierarchic file system
   *)
  (# <<SLOT DiskEntryLib: attributes>>;

     (* DISK ENTRY EXCEPTIONS *)
     DiskEntryException: Exception
       (* General exception for disk entries *)
       (# ... #);
     DiskEntryExistsException: DiskEntryException
       (* Raised if a test for disk entry existence has
        * failed. Message: "Test for disk entry existence failed.",
        * and an indication of why it failed.
        *)
       (# ... #);
     DiskEntryModtimeException: DiskEntryException
       (* Raised if examination or setting of disk entry modtime has failed.
        * Message: "Examination/setting of disk entry modtime failed.", and an
        * indication of why it failed.
        *)
       (# ... #);
     DiskEntryTouchException: DiskEntryException
       (* Raised if touch of a disk entry has failed. Message: "Touch
```

```
   * of disk entry failed.", and an indication of why it failed.
   *)
  (# ... #);
DiskEntryRenameException: DiskEntryException
  (* Raised if rename of a disk entry has failed. Message:
   * "Rename of disk entry failed.", and an indication of why it
   * failed.
   *)
  (# ... #);
pathDesc:<
  (* A virtual descriptor for the full or relative path of
   * THIS(DiskEntry)
   *)
  (# head:<
       (* The head of the path, e.g. head of '/usr/smith/foo.bet'
        * is '/usr/smith'
        *)
       (# h: ^text
       ...
       exit h[]
       #);
     nameDesc:<
       (* The actual name−part of the path, e.g. name part of
        * '/usr/smith/foo.bet' is 'foo.bet'
        *)
       (# prefix:<
            (* exits the prefix part of the name, i.e. what is
             * before the last dot (.), e.g 'foo' for 'foo.bet'
             *)
            (# p: ^text
            ...
            exit p[]
            #);
          extension:<
            (* exits the extension part of the name, i.e. what
             * is after the last dot (.), e.g. 'bet' for 'foo.bet'
             *)
            (# e: ^text
            ...
            exit e[]
            #);
          suffix:<
            (* like extension, but includes the dot (.),
             * e.g. '.bet' for 'foo.bet'
             *)
            (# s: ^text
            ...
            exit s[]
            #);
          get:<
            (* exits "prefix.extension" *)
            (# n: ^text
            ...
            exit n[]
            #);
       exit get
       #);
     name: @nameDesc;
     set:<
       (* set the entire path *)
       (# p: ^text
       enter p[]
       ...
       #);
     get:<
       (* get the entire path *)
```

```
        (# p: ^text
        ...
        exit p[]
        #);
    enter set
    do INNER
    exit get
    #);
  path: @pathDesc;
  exists: BooleanValue
    (* exits a boolean indicating whether the disk entry
     * corresponding to the current setting of path actually exists
     *)
    (# error:< DiskEntryExistsException;
    ...
    #);
  modtime:
    (* exits an integer denoting the (system) time of the last
     *         modification
     *)
    (# time: @integer;
       error:<DiskEntryModTimeException;
    enter (# enter time ... #)
    exit (# ... exit time #)
    #);
  touch:
    (* Updates the modtime to the current (system) time. *)
    (# error:< DiskEntryTouchException;
    ...
    #);
  rename:
    (* Rename the disk entry. Changes the physical disk entry and
     * updates THIS(DiskEntry).path
     *)
    (# newpath: ^text;
       error:< DiskEntryRenameException;
    enter newpath[]
    ...
    #);
  size: IntegerValue
    (* exits the size of THIS(DiskEntry) in bytes *)
    (# error:<DiskEntryException;
    ...
    #);
  readable: BooleanValue
    (* exits true if THIS(DiskEntry) can be read *)
    (# error:< DiskEntryException;
    ...
    #);
  writeable: BooleanValue
    (* exits true if THIS(DiskEntry) can be written to *)
    (# error:< DiskEntryException;
       checkwrite: @...;
    do checkwrite
    #);
  isFile: BooleanValue
    (* True if THIS(DiskEntry) is a regular file *)
    (# error:< DiskEntryException;
    ...
    #);
  isDirectory: BooleanValue
    (* True if THIS(DiskEntry) is a directory *)
    (# error:<DiskEntryException;
    ...
    #);
```

```
     private: @...
  do INNER
  #); (* DiskEntry *)

(* Constants used for specifying mode to File.SetPos. *)
FromBeginning:
  (* Seeks relative to the beginning of a file.  Corresponds to
   * absolute positions in File[0:File.Length−1].
   *)
  (# exit 0 #);
FromCurrent:
  (* Seeks relative to the current position.   *)
  (# exit 1 #);
FromEnd:
  (* Seeks relative to the end of a file. *)
  (# exit 2 #);

File: Stream
  (* Generalization of disk file.  Describes the stream aspects of
   * files, providing buffered I/O, and contains a DiskEntry object
   * describing the other properties of a file.
   *)
  (# <<SLOT FileLib: attributes>>;

     EntryDesc:< DiskEntry;
     Entry: @EntryDesc
       (* The item holding most characterizing attributes of
        * THIS(file)
        *);
     name: @
       (* convenient interface to entry.path *)
       (# read:
            (* Reads the file name from the Keyboard *)
            (# ... #);
       enter entry.path
       exit entry.path
       #);
     Put::<  (# ... #);
     Get::<  (# ... #);
     Peek::< (# ... #);
     PutText::< (# ... #);
     GetAtom::< (# ... #);
     GetLine::< (# ... #);
     Length::<
       (* Returns the byte size of THIS(file). Notice that this is
        * not always the same as entry.size, which is how many bytes
        * THIS(file) occupies on the disk
        *)
       (# ... #);
     GetPos::<
       (* Returns current position of THIS(File) *)
       (# ... #);
     SetPos::<
       (* Sets position on THIS(file).  Enters position and mode. See
        * above for definition of constants to use as mode,
        * FromBeginning, FromCurrent, FromEnd.  Returns the absolute
        * position seeked to [0..File.Length−1].
        *)
       (# mode,newpos: @integer
       enter mode
       ...
       exit newpos
       #);
     Eos::< (# ... #);
     touch: entry.touch
       (* If the disk entry does not exist, an empty file will be
```

```
   * created.
   *)
  (# ... #);
delete:
  (* Deletes THIS(File) *)
  (# ... #);
binary:< booleanvalue
  (* THIS(File) is binary if value is true. On some systems a
   * non−binary (e.g. textual) file may behave differently, than
   * a binary file. A binary file is always treated as raw bytes,
   * whereas a non−binary file may treat some characters, notably
   * the end−of−line marker, differently.
   *);
openRead:
  (* opens THIS(File) for reading, starting at the beginning *)
  (# ... #);
openWrite:
  (* Opens THIS(File) for writing.  truncates the contents of
   * the disk file if it already existed, and creates the disk
   * file if not
   *)
  (# ... #);
openAppend:
  (* Opens THIS(File) for writing at the end.  Setpos cannot be
   * used to write other places than at the end.  Creates the
   * file if it did not exist.
   *)
  (# ... #);
openReadWrite:
  (* Opens THIS(File) for both reading and writing.  The file is
   * positioned at the beginning. To switch between writing and
   * reading an intermediate setpos may be necessary.
   *)
  (# ... #);
openReadAppend:
  (* Like OpenReadWrite, but positiones at the end *)
  (# ... #);
flush:
  (* Flushes THIS(File). Affects only files opened for output *)
  (# ... #);
close:  (* Closes THIS(File) *)
  (# ... #);

(* FILE EXCEPTIONS *)

FileException: StreamException
  (* General File exception *)
  (# m: ^text
  enter m[]
  ...
  #);
OpenException: FileException
  (* Raised if opening of a file has failed. Message: "Cannot
   *         open file".
   *)
  (# ... #);
AccessError:< OpenException
  (* Raised on attempt to access a file with insufficient
   * privilegies.  Message: "Insufficient access privilegies".
   *)
  (# ... #);
WriteError:< FileException
  (* Raised from Put, PutText and Flush on attempt to write on a
   * non−existing block. Message: "Write block error".
   *)
  (# ... #);
```

```
    ReadError:< FileException
      (* Raised from Get and Peek on attempt to read a non−existing
       * block. Message: "Read block error".
       *)
      (# ... #);
    EOSerror::<
      (# ... #);
    NoSuchFileError:< FileException
      (* Raised on attempt to open a non−existing file. Message:
       * "File does not exist"
       *)
      (# ... #);
    FileExistsError:< FileException
      (* Raised when creating an already existing file. Message:
       * "File does already exist".
       *)
      (# ... #);
    NoSpaceError:< FileException
      (* Raised when the file system is full. Message: "File system
       * is full".
       *)
      (# ... #);
    OtherError:< FileException
      (* Raised when errors other than the above occur *)
      (# ... #);
    private: @ ...;
#)
```

13.7 Filerep Interface

```
ORIGIN 'betaenv';
LIB_DEF 'filerep' '../lib';
BODY 'private/filerepbody'
---lib:attributes---
FileRep:
  (* A pattern consisting of a repetition R which may be
   * saved/restored in one chunk from a file; When saving, the
   * elements R[1:top-1] are saved; After restoring top is R.range
   *)
  (# <<SLOT FileRepLib: attributes>>;
     R: [1] @integer;
     top: @integer;
     Save:
       (# filename: ^text
       enter filename[]
       do ...
       #);
     Restore:
       (# filename: ^text
       enter filename[]
       do ...
       #)
  #)
```

13.8 Formatio Interface

```
ORIGIN 'betaenv';
LIB_DEF 'formatio' '../lib';
BODY 'private/formatioBody'
(*
 * COPYRIGHT
 *       Copyright (C) Mjolner Informatics, 1984-96
 *       All rights reserved.
 *
 *)
--- streamLib: attributes ---
getFormat: formatter
  (* getFormat accepts the following syntax for markers:
   *
   *      %[width][.[precision]]{dioxXrRbBfeEgGcsn%}
   *
   * where width is a (unsigned) decimal number (or '*'), and
   *   precision is a (unsigned) decimal number (or '*'),
   *
   * and where
   *     [ ... ] means that the enclodes is optional,
   * and { ... } means one if the enclosed characters.
   *
   * Width is only interpreted in conjunction with the 's' marker.
   * Precision is only interpreted in conjunction with the
   * 'r' and 'R' markers.
   *
   * For all but the 'c' marker, leading white space are skipped.
   *
   * The '%' marker means that a '%' is expected on the input stream.
   *
   * Actually, getFormat accepts the same syntax as putFormat (see
   * later):
   *
   *      %{-+ }[[0]width][.[[0]precision]]{dioxXrRbBfeEgGcsn%}
   *
   * but only the above part of that syntax is actually interpreted by
   * getFormat.  The reason for accepting the same syntax is to allow
   * a format string to be used both for getFormat and putFormat.
   *)
  (# <<SLOT getformatlib: attributes>>;
     width:
       (* sets the default precision to be used in future '*' width
        * specifications
        *)
       (# w: @integer enter w ... #);
     precision:
       (* sets the default precision to be used in future '*'
        * precision specifications
        *)
       (# p: @integer enter p ... #);
     marker: scanForMarker
       (# formatEOS:: (# do mark->missingMarker #)
       ...
       #);
     d: marker (* read a decimal number *)
       (# mark:: (# do 'd'->value #);
          value: @integer
       ...
       exit value
       #);
     i: marker
       (* read a number, either decimal, octal or hexadecimal using C
```

```
   * conventions: 0nnn implies octal, 0xnnn implies hexadecimal,
   * decimal otherwise.
   *)
  (# mark:: (# do 'i'->value #);
     value: @integer
  ...
  exit value
  #);
o: marker (* read an octal number *)
  (# mark:: (# do 'o'->value #);
     value: @integer
  ...
  exit value
  #);
x: marker (* read a hexadecimal number *)
  (# mark:: (# do 'x'->value #);
     value: @integer
  ...
  exit value
  #);
uX: marker (* read a hexadecimal number.  Identical to 'x' *)
  (# mark:: (# do 'X'->value #);
     value: @integer
  ...
  exit value
  #);
r: marker
  (* read a number in radix given by precision *)
  (# mark:: (# do 'r'->value #);
     value: @integer
  ...
  exit value
  #);
uR: marker
  (* read a number in radix given by precision.  Identical to
   * 'r'
   *)
  (# mark:: (# do 'R'->value #);
     value: @integer
  ...
  exit value
  #);
b: marker
  (* read a number as based number (i.e. in the bbxnnn format) *)
  (# mark:: (# do 'b'->value #);
     base, value: @integer
  ...
  exit (base,value)
  #);
uB: marker
  (* read a number as based number (i.e. in the bbxnnn format).
   * Identical to 'b'
   *)
  (# mark:: (# do 'B'->value #);
     base, value: @integer
  ...
  exit (base,value)
  #);
f: marker (* read a real *)
  (# mark:: (# do 'f'->value #);
     value: @real
  ...
  exit value
  #);
e: marker (* read a real.  Identical to 'f' *)
  (# mark:: (# do 'e'->value #);
```

```
          value: @real
        ...
        exit value
        #);
      uE: marker (* read a real.  Identical to 'f' *)
        (# mark:: (# do 'E'−>value #);
          value: @real
        ...
        exit value
        #);
      g: marker (* read a real.  Identical to 'f' *)
        (# mark:: (# do 'g'−>value #);
          value: @real
        ...
        exit value
        #);
      uG: marker (* read a real.  Identical to 'f' *)
        (# mark:: (# do 'G'−>value #);
          value: @real
        ...
        exit value
        #);
      c: marker (* read a single character *)
        (# mark:: (# do 'c'−>value #);
          value: @char
        ...
        exit value
        #);
      s: marker
        (* read a text atom (i.e. a sequence of non−white space
         * characters, terminated by a white space character − similar
         * to getAtom).  If width is non−zero, at most width characters
         * are read.  Otherwise, characters are read until next
         * whitespace character
         *)
        (# mark:: (# do 's'−>value #);
          value: ^text
        ...
        exit value[]
        #);
      n: marker
        (* return the number of characters read until this point *)
        (# mark:: (# do 'n'−>value #);
          value: @integer
        ...
        exit value
        #);
      match:: (* private *)
        (# ... #)
  do INNER getFormat
  #);

putFormat: formatter
  (* putFormat accepts the following syntax for markers:
   *
   *      %{−+ }[[0]width][.[[0]precision]]{dioxXrRbBfeEgGcsnt%}
   *
   * where width is a (unsigned) decimal number (or '*'), and
   *   precision is a (unsigned) decimal number (or '*') and where
   *
   *     [ ... ] means that the enclodes is optional,
   * and { ... } means one if the enclosed characters.
   *
   * '−' implies output leftjustified in field
   * '+' implies output numbers signed (always with leading '+' or
   *                                    '−')
```

```
 * ' ' implies output numbers with blank sign (i.e. leading ' ' if
 *                                                 positive)
 * '0' in front of either width or precision implies zero padding
 * width specifies the minimun width of the output field.
 * precision is interpreted for various things in the 'rRbBfeEgG'
 *     markers.
 *
 * The '%' marker means that a '%' is put on the output stream.
 *
 * The '%t' marker prints booleans, use it with tv.
 *)
(# <<SLOT putformatlib: attributes>>;
   width:
     (* sets the default precision to be used in future '*' width
      * specifications
      *)
     (# w: @integer enter w ... #);
   precision:
     (* sets the default precision to be used in future '*'
      * precision specifications
      *)
     (# p: @integer enter p ... #);
   marker: scanForMarker
     (# formatEOS:: (# do mark->missingMarker #)
     ...
     #);
   d: marker (* insert the integer *)
     (# mark:: (# do 'd'->value #);
        value: @integer
     enter value
     ...
     #);
   i: marker (* insert the integer. Identical to 'd' *)
     (# mark:: (# do 'i'->value #);
        value: @integer
     enter value
     ...
     #);
   y: marker (* insert the integer in binary *)
     (# mark:: (# do 'y'->value #);
        value: @integer
     enter value
     ...
     #);
   o: marker (* insert the integer in octal, at least 10 ciffers *)
     (# mark:: (# do 'o'->value #);
        value: @integer
     enter value
     ...
     #);
   x: marker (* insert the integer in hexadecimal, at least 8 ciffers *)
     (# mark:: (# do 'x'->value #);
        value: @integer
     enter value
     ...
     #);
   uX: marker
     (* insert the integer in hexadecimal, using uppercase letters
      *)
     (# mark:: (# do 'X'->value #);
        value: @integer
     enter value
     ...
     #);
   r: marker
     (* insert the integer in radix given by precision *)
```

```
  (# mark:: (# do 'r'->value #);
     value: @integer
  enter value
  ...
  #);
uR: marker
  (* insert the integer in radix given by precision, using
   * uppercase letters if radix>10
   *)
  (# mark:: (# do 'R'->value #);
     value: @integer
  enter value
  ...
  #);
b: marker
  (* insert the integer as based number (i.e. in the bbxnnn
   * format) with base given by precision
   *)
  (# mark:: (# do 'b'->value #);
     value: @integer
  enter value
  ...
  #);
uB: marker
  (* insert the integer as based number (i.e. in the bbxnnn
   * format) with base given by precision, using uppercase
   * letters if base>10
   *)
  (# mark:: (# do 'B'->value #);
     value: @integer
  enter value
  ...
  #);
f: marker (* insert the real in noexp style *)
  (# mark:: (# do 'f'->value #);
     value: @real
  enter value
  ...
  #);
e: marker (* insert the real in exp style *)
  (# mark:: (# do 'e'->value #);
     value: @real
  enter value
  ...
  #);
uE: marker (* insert the real in exp style and upcase 'E' *)
  (# mark:: (# do 'E'->value #);
     value: @real
  enter value
  ...
  #);
g: marker (* insert the real in plain style *)
  (# mark:: (# do 'g'->value #);
     value: @real
  enter value
  ...
  #);
uG: marker (* insert the real in plain style and upcase 'E' *)
  (# mark:: (# do 'G'->value #);
     value: @real
  enter value
  ...
  #);
c: marker (* insert the char *)
  (# mark:: (# do 'c'->value #);
     value: @char;
```

```
            putc: @(* Private *)...
          enter value
          do putc
          #);
        s: marker (* insert the text *)
          (# mark:: (# do 's'−>value #);
             value: ^text;
             puts: @(* Private *)...
          enter value[]
          do puts
          #);
        n: marker
          (* return the length of the result string until this point *)
          (# mark:: (# do 'n'−>value #);
             value: @integer
          ...
          exit value
          #);
        trueText:<(# t:^text; do 'true'−>t[]; INNER exit t[] #);
        falseText:<(# t:^text; do 'false'−>t[]; INNER exit t[] #);
        tv: marker (* Truth−value, prints trueText of falseText *)
          (# mark:: (# do 't'−>value #);
             localTrueText:<(# t:^text; enter t[] do INNER exit t[] #);
             localFalseText:<(# t:^text; enter t[] do INNER exit t[] #);
             value:@boolean
          enter value
          ...
          #);
        match:: (* private *)
          (# ... #)
    do INNER putFormat
    #);

  formatter: (* superpattern for putFormat and getFormat *)
    (# illegalFormat:< exception
          (# mark: @char
          enter mark
          ...
          #);
        missingMarker:< exception
          (# mark: @char
          enter mark
          ...
          #);
        missingField:< exception
          (# ... #);
        inputError:< exception
          (# chFound, chExpected: @char
          enter (chFound, chExpected)
          ...
          #);
        scanForMarker: (* Private *)
          (# mark:< charValue;
             formatEOS:< exception;
             fieldWidth, precisionSpec: @integer;
             leftFlag, signedFlag, blankFlag,
             alternativeFlag, zeroFlag, longFlag: @boolean
          ...
          #);
        match:< (* private *)
          (# ch: @char
          enter ch
          do INNER match
          #);
        private: @...;
        formatStr: ^text
```

```
  enter formatStr[]
  ...
  exit this(stream)[]
  #);

--- lib: attributes ---

getFormat: keyboard.getFormat(# do INNER getFormat #);

putFormat: screen.putFormat(# do INNER putFormat #)
```

13.9 Iget Interface

```
ORIGIN 'betaenv';
BODY 'private/igetbody';
(*
 * COPYRIGHT
 *       Copyright (C) Mjolner Informatics, 1984-96
 *       All rights reserved.
 *
 *)
--LIB: attributes--
iget:
  (* Pattern to get a single character from the keyboard immediately
   * without having to type <RETURN> after the character.
   *)
  (# ch: @char;
  ...
  exit ch
  #)
```

13.10 Math Interface

```
ORIGIN 'betaenv';
LIB_DEF 'math' '../lib';
(*
 * COPYRIGHT
 *       Copyright (C) Mjolner Informatics, 1984-96
 *       All rights reserved.
 *
 * Math.bet: mathematical functions
 *
 * This library provides mathematical patterns: trigonometric,
 * hyperbolic, exponential and logarithmic, floating point
 * manipulation and miscellaneous constants.
 *)
--- LIB: Attributes ---

(* miscellaneous constants *)

e:       (# Exit 2.7182818284590452354 #);
log2e:   (# Exit 1.4426950408889634074 #);
log10e:  (# Exit 0.43429448190325182765 #);
ln2:     (# Exit 0.69314718055994530942 #);
ln10:    (# Exit 2.30258509299404568402 #);
pi:      (# Exit 3.14159265358979323846 #);
pihalf:  (# Exit 1.57079632679489661923 #);
piforth: (# Exit 0.78539816339744830962 #);

(* trigonometric functions *)

acos: external
  (* returns the arccosine of x in radians, in the range 0 to pi.  If
   * x is a NaN (Not-a-Number) or if the absolute value of x exceeds
   * 1.0, acos(x) returns a NaN.  Invalid operation/DOMAIN error is
   * signaled if x is a NaN or if |x| > 1.0.
   *)
  (# x,res: @Real;
  enter x
  exit res
  #);
asin: external
  (* returns the arcsine of x in radians, in the range -pi/2 to pi/2.
   * If x is a NaN or if the absolute value of x exceeds 1.0, asin
   * returns a NaN.  Invalid operation/DOMAIN error is signaled if x
   * is a NaN or if |x| > 1.0.
   *)
  (# x,res: @Real;
  enter x
  exit res
  #);
atan: external
  (* returns the arctangent of x, in the range of -pi/2 to pi/2
   * radians.  If x = +-infinity, then atan(x) returns +-pi/2.  If x
   * is +- 0, then atan returns x. If x is a NaN, then atan returns a
   * NaN.  An invalid operation/DOMAIN error is signaled by atan only
   * if x is a NaN.
   *)
  (# x,res: @Real;
  enter x
  exit res
  #);
atan2: external
  (* returns the arctangent of y/x in radians, in the range -pi to
   * pi, using the signs of both arguments to determine the quadrant
```

```
   * of the return value.
   *
   * If x is a NaN or if y is a NaN or if both x and y are infinities,
   * atan2 returns a NaN.  If both x and y are zero, atan2 returns
   * zero.  Invalid operation/DOMAIN error is signaled by atan2 if
   * both x and y are infinite or if either x or y is a NaN.
   *)
  (# y,x,res: @Real;
  enter (y,x)
  exit res
  #);
cos: external
  (* computes the cosine of x, where x is expressed in radians.
   *
   * The cos function uses an argument reduction based on the
   * remainder function and pi.  The cos function is periodic with
   * respect to pi, so its period differs slightly from its
   * mathematical counterpart and diverges from its counterpart when
   * the argument becomes very large.
   *
   * If x is infinite or a NaN, then cos returns a NaN and signals
   * invalid/DOMAIN error.
   *)
  (# x,res: @Real;
  enter x
  exit res
  #);
sin: external
  (* computes the sine of x, where x is expressed in radians.
   *
   * The sin function uses an argument reduction based on the
   * remainder function and pi.  The sin function is periodic with
   * respect to pi, so its period differs slightly from its
   * mathematical counterpart and diverges from its counterpart when
   * the argument becomes very large.
   *
   * If x is infinite or a NaN, then sin returns a NaN and signals
   *    invalid/DOMAIN error.
   *)
  (# x,res: @Real;
  enter x
  exit res
  #);
tan: external
  (* computes the tangent of x, where x is expressed in radians.
   *
   * The tan function uses an argument reduction based on the
   * remainder function and pi The tan function is periodic with
   * respect to pi, so its period differs slightly from its
   * mathematical counterpart and diverges from its counterpart when
   * the argument becomes very large.
   *)
  (# x,res: @Real;
  enter x
  exit res
  #);

(* hyperbolic functions *)

cosh: external
  (* returns the hyberbolic cosine of x.
   *
   * If x is a NaN, cosh returns a NaN.
   *)
  (# x,res: @Real;
  enter x
```

```
  exit res
  #);
sinh: external
  (* returns the hyberbolic sine of x.
   *
   * If x is a NaN, sinh returns a NaN.
   *)
  (# x,res: @Real;
  enter x
  exit res
  #);
tanh: external
  (* returns the hyberbolic tangent of x.
   *
   * If x is a NaN, tanh returns a NaN.
   *)
  (# x,res: @Real;
  enter x
  exit res
  #);

(* exponential and logarithmic functions *)

exp: external
  (* returns the base−e or natural exponential e^x.
   *
   * Special cases for exp:
   *
   * If x = +infinity, then exp returns +infinity and does not signal
   * an exception.  If x = -infinity, then exp returns 0 and does not
   * signal an exception.  If x is a NaN, then exp returns a NaN.
   *)
  (# x,res: @Real;
  enter x
  exit res
  #);
ldexp: external
  (* returns the quantity x * 2^exp. *)
  (# x,res: @Real;
     exp: @Integer;
  enter (x,exp)
  exit res
  #);
log: external
  (* returns the base e or natural logarithm of its argument x.
   *
   * Special cases for log:
   *
   * If x is +infinity, then log returns +infinity and signals no
   * exceptions.  If x is 0, then log returns −infinity and signals
   * divide−by−zero.  If x < 0, then log returns a NaN and signals
   * invalid/DOMAIN error.
   *)
  (# x,res: @Real;
  enter x
  exit res
  #);
log10: external
  (* returns the base 10 logarithm of x.
   *
   * If x is a NaN or is negative, log10 returns a NaN. If x is
   * +infinity, log10(x) returns +infinity.  If x is zero, log10
   * returns −infinity and signals divide by zero/SING error.
   *)
  (# x,res: @Real;
  enter x
```

```
  exit res
  #);

(* floating point manipulation *)

modf: external
  (* returns the fractional part of x and stores the integral part
   * indirectly in the location pointed to by ipPtr.  Both the return
   * value and the value stored in ipPtr share the same sign as x.
   *
   * If x is infinite, modf returns a zero with the sign of x and sets
   * ipPtr to x.  If x is a NaN, mod returns a NaN and sets ipPtr to
   * the same NaN.
   *)
  (# x,res: @Real;
     ipPtr: @Integer;
  enter (x,ipPtr)
  exit res
  #);
pow: external
  (* returns x^y *)
  (# x,y,res: @Real;
  enter (x,y)
  exit res
  #);
sqrt: external
  (* computes the square root of x.
   *
   * Special cases for sqrt:
   *
   * If x is a NaN, sqrt returns a NaN and signals no exceptions.  If
   * x is a NaN or if x < 0, sqrt returns a NaN and signals invalid
   * operation/DOMAIN error.
   *)
  (# x,res: @Real;
  enter x
  exit res
  #);
ceil: external
  (* returns the smallest integer value (in real format) not less
   * than x.
   *
   * If x is a NaN, ceil returns a NaN.  If x is infinite, ceil
   * returns x.  Invalid operation is signaled by ceil if x is a NaN.
   * If x is a non−integral finite value, ceil signals inexact
   *)
  (# x,res: @Real;
  enter x
  exit res
  #);
fmin: (* Returns the minimum of 2 reals *)
  (# a,b: @real
  enter (a,b)
  do (if (a < b) then a−>b if)
  exit b
  #);
fmax: (* Returns the maximum of 2 reals *)
  (# a,b: @real
  enter (a,b)
  do (if (a < b) then b−>a if)
  exit a
  #);
fabs: external
  (* returns |x|, the absolute value of x *)
  (# x,res: @Real;
  enter x
```

```
  exit res
  #);
floor: external
  (* largest integer value (in real format) not greater than x.
   *
   * If x is a NaN, floor returns a NaN.  If x is infinite, floor
   * returns x. Invalid operation is signaled by floor if x is a NaN.
   * If x is a non−integral finite value, floor signals inexact.
   *)
  (# x,res: @Real;
  enter x
  exit res
  #);
fmod: external
  (* Whenever possible, the fmod pattern returns the number f with
   * the same sign as x, such that x = i*y + f for some integer i, and
   * |f| < |y|. If y is 0, fmod returns a NaN.
   *)
  (# x,y,f: @Real;
  enter (x,y)
  exit f
  #)
```

13.11 Mbs_version Interface

```
ORIGIN 'betaenv';
LIB_DEF 'mbs_version' '../lib';
(*
 * COPYRIGHT
 *       Copyright (C) Mjolner Informatics, 1984-2004
 *       All rights reserved.
 *)
-- LIB: attributes --
mbs_version:
  (* Defines the current version of the Mjolner BETA System
   * in various ways. Typical usage:
   *
   *   (if (mbs_version).major >= 4 then
   *       ...
   *   if);
   *)
  (# major:    (# exit 5 #);
     minor:    (# exit 4 #);
     revision: (# exit 0 #);
     release:  (# exit 050400 #);
     desc:     (# exit '5.4' #);
  exit THIS(mbs_version)[]
  #)
```

13.12 Numberio Interface

```
ORIGIN 'betaenv';
LIB_DEF 'numberio' '../lib';
BODY 'private/numberioBody'
(*
 * COPYRIGHT
 *       Copyright (C) Mjolner Informatics, 1992-96
 *       All rights reserved.
 *
 * This fragment implements the following stream operations:
 *    getNumber  * reads a number from THIS(stream).
 *               * The number may be either an integer,
 *               * a based number or a real number
 *    getBased   * reads a based number from THIS(stream)
 *    getRadix   * reads a based number from THIS(stream),
 *               * without the 'bbx' part
 *    getInteger * reads an integer number from THIS(stream)
 *    getReal    * reads a real number from THIS(stream)
 *    putReal    * appends a textual rep. of a real value to
 *               * THIS(stream)
 *    putBased   * appends a textual rep. of a integer value in a
 *               * particular to THIS(stream).  The textual
 *               * representation in in the given base
 *    putRadix   * as putBased, except that the 'bbx' part is
 *               * not printed
 *    getHex     * similar to getRadix with radix 16, but more efficient.
 *    putHex     * similar to putBased with base 16, but more efficient.
 *    getOctal   * similar to getOctal with base 8, but more efficient.
 *    putOctal   * similar to putBased with base 8, but more efficient.
 *    getBinary  * similar to getRadix with radix 2, but more efficient.
 *    putBinary  * similar to putBased with base 2, but more efficient.
 *    putByteHex    * like putHex, except that it only prints one byte.
 *    putByteBinary * like putBinary, except that it only prints one byte.
 *
 *    asNumber   * abstract pattern for the following asBased,
 *               * asRadix, and asReal operations
 *    asBased    * returns the based number present in
 *               * THIS(stream)
 *    asRadix    * returns the based number present in
 *               * THIS(stream), without the 'bbx' part
 *    asReal     * returns the real number present in
 *               * THIS(stream)
 *
 * The corresponding short-cuts for keyboard.getNumber, etc, and
 * screen.putReal, etc. are also included in this fragment.
 *
 * Since the asNumber operations does not make sence for keyboard,
 * no short-cuts are defined for these.
 *)
--- StreamLib: attributes ---
getNumber:
  (* getNumber reads a number from the current position of
   * this(stream).
   * The number is either an integer (in base 10), an integer with a
   *  given base, or a real.
   * Integer examples:  10, 0, 123
   * A based integer has the form <base>X<number>. Examples are:
   *    2X101          base=2, number= 4*1 + 2*0 + 1*1 = 5
   *    8X12           base=8, number= 8*1 + 1*2 = 10
   *    16x2A1         base=16, number= 256*2 + 16*10 + 1*1  = 673
   *     0x2A1         base=16, i.e. base=0 is interpreted as base=16
   * Examples of reals are:
   *   3.14, 3.14E-8, 3E+8
```

```
 * The following grammar defines the exact syntax of the numbers:
 *
 * N ::= {D}+                          Int             314
 *    |  {D}+ '.' {D}+                 real            3.14
 *    |  {D}+ '.' {D}+ 'E' E           real            3.14E8
 *                                     real            3.14E+8
 *                                     real            3.14E−8
 *
 *    |  {D}+  'E' E                   real            3E8
 *                                     real            3E+8
 *                                     real            3E−8
 *    |  'X' {D | L}+                  based           2X0101
 *                                     based           8x0845
 *                                     based           16xAF12
 * D ::= {'0' | ... | '9' }
 * L ::= {'A' | ... | 'Z'}
 * E ::= {D}+
 *    |  {D}+ '+' {D}+
 *    |  {D}+ '−' {D}+
 *
 * All letters may be in lower or upper case.
 * After the call, the stream is positioned
 * after the first char not in the number.
 *)
(# integerValue:<
    (* the number has the form
     *     x
     * value contains the integer value
     *)
    integerValuePtn;
  integerValuePtn:
    (# value: @integer enter value do INNER #);
  basedValue:<
    (* the number has the form
     *     bXy
     * base contains the base number
     * value contains the integer value (in base 10)
     *)
    basedValuePtn;
  basedValuePtn:
    (# base,value: @integer enter (base,value) do INNER #);
  realValue:<
    (* the  number has the form
     *       x.yEz
     *       l is the number of leading zero's in y. i.e. in
     *       3.0017E−12, x=3, y=17,l=2 and z=−12
     * value contains the real value
     *)
    realValuePtn;
  realValuePtn:
    (# x,y,l,z: @real; value: @real enter(x,y,l,z,value)
    do INNER #);
  syntaxError:< streamException
    (# peekCh: @char
    enter peekCh
    do 'getNumber: Syntax error − looking at: "'->msg.append;
       (if peekCh = ascii.nul then 'NUL'->msg.puttext
        else peekCh->msg.put if);
       '"'->msg.put; INNER #);
  baseError:< streamException
    (# base: @integer
    enter base
    do 'getNumber: Error in base − looking at: "'->msg.append;
       base->msg.putInt; '"'->msg.put; INNER #);
  valueError:< streamException
    (# peekCh: @char
```

```
      enter peekCh
      do 'getNumber: Illegal value type − looking at: "'->msg.append;
         peekCh->msg.put; '"'->msg.put; INNER #);
    overflow:< streamException
      (# value: @integer
      do 'getNumber: Overflow in integer− or based−value'->msg.append;
         INNER
      exit value
      #);
    underflow:< streamException
      (# value: @integer
      do 'getNumber: Underflow in integer− or based−value'->msg.append;
         INNER
      exit value
      #);
    EOSError:< streamException
      (#
      do 'getNumber: End of stream while reading number'->msg.append;
         INNER
      #);
    getn: @...
  do getn;
    INNER getNumber
  #);
getReal: getNumber
  (# r: @real;
    realValue::< (# do value->r #)
  do INNER getReal
  exit r
  #);
getBased: getNumber
  (# i, b: @integer;
    basedValue::< (# do value->i; base->b #)
  do INNER getBased
  exit (b,i)
  #);
getInteger: getNumber
  (# i: @integer;
    integerValue::< (# do value->i #)
  do INNER getInteger
  exit i
  #);
getRadix:
  (* gets a number in the specified radix.  GetRadix is similar to
   * getBased, except that is does NOT expect the 'bbx' prefix
   *)
  (# radix, value: @integer;
    radixError:< streamException
      (# radix: @integer
      enter radix
      ...
      #);
    getr: (* private *) @...
  enter radix
  do getr;
    INNER getRadix
  exit value
  #);

putBased:
  (* Takes a number and a base, and prints the number in that base.
   * If base is 0, base 16 is assumed, and the format "0xnnn" is used.
   * If base is negative, 1 or greater that 126, the baseError
   * exception is invoked.
   *
   * The format is default "bbxnnnn", where "bb" is the base (in
```

```
   * decimal), and "nnnn" is the number, printed in the base. "x"
   * separates the two parts.  The format may be controlled by the
   * signed, blankSign, upcase, uppercase, width, adjustLeft,
   * zeroPadding, noBasePrefix, baseWidth and baseZeroPadding variable
   * attributes. If noBasePrefix is true, the "bbx" part is omitted.
   *)
  (# value, base: @integer;
     baseError:< streamException
       (# base: @integer
       enter base
       do 'putBased: Illegal base: "'->msg.append;
          base->msg.putInt; '"'->msg.put; INNER #);
     (* The format may be further controlled by the signed, blankSign,
      * width, adjustLeft and zeroPadding variable attributes.
      * width is extended if it is too small.
      *
      * Examples:
      *   (10,10)->putBased
      *      yields: '10x10'
      *   (2,5)->putBased(# do 10->width; true->adjustLeft #);
      *      yields: '2x101     '
      *   (2,5)->putBased(# do 10->width; true->zeroPadding #);
      *      yields: '2x00000101'
      *)
     signed:
       (* If the number is positive, a '+' will always be displayed
        *)
       @boolean;
     blankSign:
       (* If the number is positive, a ' ' space is displayed as the
        * sign.  Ignored if signed=true
        *)
       @boolean;
     upcase: @boolean
       (* Specifies whether an upcase 'X' or a lowcase 'x' is the
        * be used in the 'bbx' part.
        *);
     uppercase: @boolean
       (* Specifies whether uppercase letters or lowercase letters
        * are used in the 'nnnn' part (for base>9).
        *);
     width: (* Minimum width *) @integer;
     adjustLeft: @boolean
       (* Specifies if the number is to be aligned left or right,
        * if padding of spaces is necessary to fill up the specified
        * width.
        *);
     zeroPadding:
       (* width is padded with leading zero instead of spaces.
        * Ignored if adjustLeft=true
        *)
       @boolean;
     noBasePrefix: (* If true, the 'bbx' part is omitted *)
       @boolean;
     baseWidth: (* minimun width for the 'bbx' part *)
       @integer;
     baseZeroPadding:
       (* baseWidth is padded with leading zero instead of spaces *)
       @boolean;
     format:< (# do INNER #);
     putb: @...
  enter (base, value)
  do INNER putBased; putb
  #);
putReal:
  (* Append a real to THIS(stream). The format may be controlled by
```

```
   * the style, signed, blankSign, precision, upcase, width,
   * adjustLeft and zeroPadding variable attributes
   *)
  (# r: @real;
     style: @integer
       (* Controls the style, and may be one of plain, exp and noexp
        * (noexp is the default)
        *);
     noexp: (* The notation [−]mmm.dddddd is used *)
       (# exit 0 #);
     exp: (* The notation [−]m.ddddddE[+|−]xx is used *)
       (# exit 1 #);
     plain:
       (* In this style, precision is the total number of digits in
        * the printed real (not the number of digits in the fraction,
        * as in the other styles).
        *
        * The exp or noexp style is used, dependent on the value being
        * printed.  Exp style is used only if the exponent is less
        * than −4 or greater than or equal to the precision; otherwise
        * the noexp notation is used.  Trailing zeros are not printed
        * as part of the fractional part and a decimal point is
        * printed if not followed by a digit
        *)
       (# exit 2 #);
     signed: (* If real is positive, a '+' will always be displayed *)
       @boolean;
     blankSign:
       (* If real is positive, a ' ' space is displayed as the sign.
        * Ignored if signed=true
        *)
       @boolean;
     precision: @integer
       (* The number of d's in the expressions above, default 6 *);
     upcase: @boolean
       (* Specifies whether an upcase 'E' or a lowcase 'e' is the
        * be used in the exp style.
        *);
     width: (* Minimum width *)
       @integer;
     adjustLeft: @boolean
       (* Specifies if the number is to be aligned left or right,
        * if padding of spaces is necessary to fill up the specified
        * width.
        *);
     zeroPadding:
       (* width is padded with leading zero instead of spaces.
        * Ignored if adjustLeft=true
        *)
       @boolean;
     (* Examples:
      *   10*pi −> putreal;
      *      yields: '31.415926'
      *   10*pi −> putreal(# do 10−>width; true−>adjustLeft #);
      *      yields: '31.415926   '
      *   10*pi −> putreal(# do exp−>style; true−>upcase #);
      *      yields: '3.1415926E+01'
      *   10*pi −> putreal(# do exp−>style; 2−>precision #);
      *      yields: '3.14e+01'
      *)
     format:< (# do INNER #);
     putr: @...
  enter r
  do 1−>width; 6−>precision; format; INNER putReal; putr
  #);
putRadix: putBased
```

```
  (#
  do true->noBasePrefix; INNER putRadix
  #);

putHex:
  (* prints a hexadecimal representation of x (as unsigned word) on
   * this(stream). Similar to
   * (16,x)->putRadix(# do true->zeroPadding; 8->width #)
   * but more efficient.
   *)
  (# uppercase: @boolean;
     width: @integer;
     zeroPadding: @boolean;
     x: @integer;
     format:< (# do INNER #);
     putH: (*private*)@...
  enter x
  do 8->width; true->zeroPadding; format; INNER putHex; putH
  #);

putByteHex:
  (* prints a hexadecimal representation of byte 'byte' in x (as
   * unsigned word) on this(stream)
   *)
  (# x: @integer;
     byte: @integer;
     putBH: (*private*)@...
  enter (x,byte)
  do INNER putByteHex; putBH
  #);

putOctal:
  (* prints a octal representation of x (as unsigned word) on
   * this (stream). Similar to
   * (8,x)->putRadix(# do true->zeroPadding;  #)
   * but more efficient.
   *)
  (# width: @integer;
     zeroPadding: @boolean;
     x: @integer;
     format:< (# do INNER #);
     putO:(*private*)@...
  enter x
  do true->zeroPadding; format; INNER putOctal; putO;
  #);

putBinary:
  (* prints a binary representation of x (as unsigned word) on
   * this(stream). Similar to
   * (2,x)->putRadix(# do true->zeroPadding; #)
   * but more efficient.
   *)
  (# width: @integer;
     zeroPadding: @boolean;
     x: @integer;
     format:< (# do INNER #);
     putB: (*private*)@...
  enter x
  do true->zeroPadding; format; INNER putBinary; putB
  #);

putByteBinary:
  (* prints a binary representation of byte 'byte' of x (as unsigned
   * word) on this(stream)
   *)
  (# x: @integer;
```

```
      byte: @integer;
      putBB: (*private*)@...
  enter (x, byte)
  do INNER putByteBinary; putBB
  #);

getHex:
  (* reads a hexadecimal number from this(stream) and returns the
   * value in x (as unsigned word). Similar to 16−>getRadix but more
   * efficient.
   *)
  (# x: @integer;
     noNumberError:< streamException
       (# peekCh: @char
       enter peekCh
       do 'getHex: the number begins with: "'−>msg.append;
          (if peekCh = ascii.nul then 'NUL'−>msg.puttext
           else peekCh−>msg.put if);
          '".  Not a legal hexadecimal digit'−>msg.puttext;
          INNER noNumberError
       #);
     getH: (*private*)...
  do INNER getHex; getH
  exit x
  #);

getOctal:
  (* reads a hexadecimal number from this(stream) and returns the
   * value in x (as unsigned word). Similar to 16−>getRadix but more
   * efficient.
   *)
  (# x: @integer;
    noNumberError:< streamException
       (# peekCh: @char
       enter peekCh
       do 'getHex: the number begins with: "'−>msg.append;
          (if peekCh = ascii.nul then 'NUL'−>msg.puttext
           else peekCh−>msg.put if);
          '".  Not a legal hexadecimal digit'−>msg.puttext;
          INNER noNumberError
       #);
     getO: (*private*)...
  do INNER getOctal; getO
  exit x
  #);

getBinary:
  (* reads a binary number from this(stream) and returns the value in
   * x (as unsigned word). Similar to 2−>getRadix but more efficient.
   *)
  (# x: @integer;
     noNumberError:< streamException
       (# peekCh: @char
       enter peekCh
       do 'getBinary: the number begins with: "'−>msg.append;
          (if peekCh = ascii.nul then 'NUL'−>msg.puttext
           else peekCh−>msg.put if);
          '".  Not a legal binary digit'−>msg.puttext;
          INNER noNumberError
       #);
     getB: (*private*)...
  do INNER getBinary; getB
  exit x
  #);

asNumber:
```

```
  (# syntaxError:< streamException
       (# peekCh: @char
       enter peekCh
       do 'asNumber: Syntax error − looking at: "'->msg.append;
          peekCh->msg.put; '"'->msg.put;
          INNER syntaxError
       #);
     baseError:< streamException
       (# base: @integer
       enter base
       do 'asNumber: Error in base − looking at: "'->msg.append;
          base->msg.putInt; '"'->msg.put;
          INNER baseError
       #);
     valueError:< streamException
       (# peekCh: @char
       enter peekCh
       do 'asNumber: Illegal value type − looking at: "'->msg.append;
          peekCh->msg.put; '"'->msg.put;
          INNER valueError
       #);
  do reset;
     INNER asNumber;
     ScanWhiteSpace; (if not eos then peek->syntaxError if)
  #);

asReal: asNumber
  (# r: @real
    ...
  exit r
  #);
asBased: asNumber
  (# i, b: @integer
    ...
  exit (b,i)
  #);
asRadix: asNumber
  (# radix, value: @integer
  enter radix
  ...
  exit value
  #);
asInteger: asNumber
  (# i: @integer
    ...
  exit i
  #)

--- lib: attributes ---

getNumber: keyboard.getNumber
  (# do INNER getNumber #);
getReal: keyboard.getReal
  (# do INNER getReal #);
getBased: keyboard.getBased
  (# do INNER getbased #);
getRadix: keyboard.getRadix
  (# do INNER getRadix #);
getInteger: keyboard.getInteger
  (# do INNER getInteger #);

putReal: screen.putReal
  (# do INNER putReal #);
putBased: screen.putBased
  (# do INNER putBased #);
putRadix: screen.putRadix
```

```
  (# do INNER putRadix #);

getHex: keyboard.getHex
  (# do INNER getHex #);
getOctal: keyboard.getOctal
  (# do INNER getOctal #);
getBinary: keyboard.getBinary
  (# do INNER getBinary #);

putHex: screen.putHex
  (# do INNER putHex #);
putByteHex: screen.putByteHex
  (# do INNER putByteHex #);
putOctal:screen.putOctal
  (# do INNER putOctal #);
putBinary: screen.putBinary
  (# do INNER putBinary #);
putByteBinary: screen.putByteBinary
  (# do INNER putByteBinary #);


real2ints:
  (* Extract the high− and low bits of the real r into (i1, i2) *)
  (# r: @real;
     i1, i2: @int32;
  enter r
  ...
  exit (i1, i2)
  #);

ints2real:
  (* Combine the high− and low bits in (i1, i2) to form the real r *)
  (# i1, i2: @int32;
     r: @real;
  enter (i1, i2)
  ...
  exit r
  #)
```

13.13 Random Interface

```
ORIGIN 'betaenv';
BODY 'private/randombody';
(*
 * COPYRIGHT
 *       Copyright (C) Mjolner Informatics, 1995-96
 *       All rights reserved.
 *)
--- lib: attributes ---

initgn: external
  (*
   * INIT-ialize current G-e-N-erator
   *      Reinitializes the state of the current generator
   *
   * Arguments
   *      isdtyp -> The state to which the generator is to be set
   *        isdtyp = -1 => sets the seeds to their initial value
   *        isdtyp = 0 => sets the seeds to the first value of the
   *                      current block
   *        isdtyp = 1 => sets the seeds to the first value of the
   *                      next block
   *
   * Method
   *      This is a transcription from Pascal to Fortran of routine
   * Init_Generator from the paper
   *      L'Ecuyer, P. and Cote, S. "Implementing a Random Number
   * Package with Splitting Facilities."  ACM Transactions on
   * Mathematical Software, 17:98-111 (1991)
   *)
  (# isdtyp: @integer
  enter (isdtyp)
  #);


gscgn: external
  (*
   * Get/Set GeNerator
   *      Gets or returns in G the number of the current generator
   *
   * Arguments
   *      getset -->> 0 Get 1 Set
   *      g <<-- Number of the current random number generator (1..32)
   *)
  (# getset, g: @integer
  enter (getset, g)
  #);


setsd: external
  (*
   * SET S-ee-D of current generator
   *      Resets the initial seed of the current generator to ISEED1
   *      and ISEED2. The seeds of the other generators remain
   *      unchanged.
   * Arguments
   *      iseed1 -> First integer seed
   *      iseed2 -> Second integer seed
   * Method
   *      This is a transcription from Pascal to Fortran of routine
   * Set_Seed from the paper L'Ecuyer, P. and Cote, S. "Implementing a
   * Random Number Package with Splitting Facilities."  ACM
   * Transactions on Mathematical Software, 17:98-111 (1991)
```

```
   *)
  (#  iseed1, iseed2: @integer
  enter (iseed1, iseed2)
  #);


getsd: external
  (*
   * GET SeeD
   *      Returns the value of two integer seeds of the current
   *      generator
   * Arguments
   *      iseed1 <− First integer seed of generator G
   *      iseed2 <− Second integer seed of generator G
   * Method
   *      This is a transcription from Pascal to Fortran of routine
   * Get_State from the paper L'Ecuyer, P. and Cote, S. "Implementing
   * a Random Number Package with Splitting Facilities."  ACM
   * Transactions on Mathematical Software, 17:98−111 (1991)
   *)
  (# iseed1, iseed2: @integer
  enter (iseed1, iseed2)
  #);


setall: external
  (*
   * SET ALL random number generators
   *      Sets the initial seed of generator 1 to ISEED1 and
   * ISEED2. The initial seeds of the other generators are set
   * accordingly, and all generators states are set to these seeds.
   *
   * Arguments
   *      iseed1 −> First of two integer seeds
   *      iseed2 −> Second of two integer seeds
   *
   * Method
   *      This is a transcription from Pascal to Fortran of routine
   * Set_Initial_Seed from the paper L'Ecuyer, P. and Cote,
   * S. "Implementing a Random Number Package with Splitting
   * Facilities."  ACM Transactions on Mathematical Software,
   * 17:98−111 (1991)
   *)
  (# iseed1, iseed2: @integer
  enter (iseed1, iseed2)
  #);


setant: external
  (*
   * SET ANTithetic Sets whether the current generator produces
   *      antithetic values.  If X is the value normally returned from
   *      a uniform [0,1] random number generator then 1 − X is the
   *      antithetic value. If X is the value normally returned from a
   *      uniform [0,N] random number generator then N − 1 − X is the
   *      antithetic value. All generators are initialized to NOT
   *      generate antithetic values.
   * Arguments
   *      qvalue −> nonzero if generator G is to generating antithetic
   *                values, otherwise zero
   * Method
   *      This is a transcription from Pascal to Fortran of routine
   * Set_Antithetic from the paper L'Ecuyer, P. and Cote,
   * S. "Implementing a Random Number Package with Splitting
   * Facilities."  ACM Transactions on Mathematical Software,
   * 17:98−111 (1991)
```

```
   *)
  (# qvalue: @integer
  enter qvalue
  #);


advnst: external
  (*
   * ADV−a−N−ce ST−ate
   *      Advances the state of the current generator by 2^K values
   *      and resets the initial seed to that value.
   * Arguments
   *      k −> The generator is advanced by2^K values
   * Method: routine Advance_State from the paper L'Ecuyer, P. and
   *      Cote, S. "Implementing a Random Number Package with
   *      Splitting Facilities."  ACM Transactions on Mathematical
   *      Software, 17:98−111 (1991)
   *)
  (# k: @integer
  enter k
  #);

(*=================================================================*)
(*===== Integer random number generators ==========================*)
(*=================================================================*)

ignlgi: external
  (*
   * Integer GeNerate LarGe Integer
   *      Returns a random integer following a uniform distribution
   * over (1, 2147483562) using the current generator.
   *
   * Method
   *      This is a transcription from Pascal to Fortran of routine
   * Random from the paper L'Ecuyer, P. and Cote, S. "Implementing a
   * Random Number Package with Splitting Facilities."  ACM
   * Transactions on Mathematical Software, 17:98−111 (1991)
   *)
  (# random: @integer
  exit random
  #);


ignuin: external
  (*
   * GeNerate Uniform INteger
   *      Generates an integer uniformly distributed between LOW and
   * HIGH.
   *
   * Arguments
   *      low −−> Low bound (inclusive) on integer value to be
   *              generated
   *      high −−> High bound (inclusive) on integer value to be
   *               generated
   *
   * Note
   *      If (HIGH−LOW) > 2,147,483,561 prints error message on * unit
   * and stops the program.
   *)
  (# low, high: @integer;
     random: @integer
  enter (low, high)
  exit random
  #);
```

```
ignbin: external
  (*
   * Integer GeNerate BINomial random deviate
   *      Generates a single random deviate from a binomial
   * distribution whose number of trials is N and whose probability of
   * an event in each trial is P.
   *
   * Arguments
   *      n −−> The number of trials in the binomial distribution from
   *            which a random deviate is to be generated.
   *      p −−> The probability of an event in each trial of the
   *            binomial distribution from which a random deviate is
   *            to be generated.
   *      ignbin <−− A random deviate yielding the number of events
   *            from N independent trials, each of which has a
   *            probability of event P.
   * Method
   *      This is algorithm BTPE from: Kachitvichyanukul, V. and
   * Schmeiser, B.  W.  Binomial Random Variate Generation.
   * Communications of the ACM, 31, 2 (February, 1988) 216.
   *)
  (# n: @integer;
     p: @real;
     random: @integer
  enter (n, p)
  exit random
  #);


ignnbn: external
  (*
   * Integer GeNerate Negative BiNomial random deviate
   *      Generates a single random deviate from a negative binomial
   * distribution.
   *
   * Arguments
   *      N −−> The number of trials in the negative binomial
   *            distribution from which a random deviate is to be
   *            generated.
   *      P −−> The probability of an event.
   * Method
   *      Algorithm from page 480 of Devroye, Luc, Non−Uniform Random
   * Variate Generation.  Springer−Verlag, New York, 1986.
   *)
  (# n: @integer;
     p: @real;
     random: @integer
  enter (n, p)
  exit random
  #);


ignpoi: external
  (*
   * GENerate POIsson random deviate
   *      Generates a single random deviate from a Poisson
   *      distribution with mean AV.
   *
   * Arguments
   *      mu −−> The mean of the Poisson distribution from which a
   *             random deviate is to be generated.
   *
   * Method
   *      Renames KPOIS from TOMS as slightly modified by BWB to use
   * RANF instead of SUNIF.  For details see: Ahrens, J.H. and Dieter,
   * U.  Computer Generation of Poisson Deviates From Modified Normal
```

```
   * Distributions.  ACM Trans. Math. Software, 8, 2 (June
   * 1982),163−179
   *)
  (# mu: @real;
     random: @integer
  enter mu
  exit random
  #);

(*=================================================================*)
(*===== Real random number generators ============================*)
(*=================================================================*)

ranf: external
  (*
   * RANnom number generator as a Function
   *      Returns a random floating point number from a uniform
   * distribution over 0 − 1 (endpoints of this interval are not
   * returned) using the current generator
   *
   * Method
   *      This is a transcription from Pascal to Fortran of routine
   * Uniform_01 from the paper L'Ecuyer, P. and Cote, S. "Implementing
   * a Random Number Package with Splitting Facilities."  ACM
   * Transactions on Mathematical Software, 17:98−111 (1991)
   *)
  (# random: @real
  exit random
  #);


genunf: external
  (*
   * GENerate UNiForm real
   *      Generates a real uniformly distributed between LOW and HIGH.
   *
   * Arguments low −−> Low bound (exclusive) on real value to be
   *      generated high −−> High bound (exclusive) on real value to
   *      be generated
   *)
  (# low, high: @real;
     random: @real
  enter (low, high)
  exit random
  #);


genbet: external
  (*
   * GeNerate BETa random deviate
   *      Returns a single random deviate from the beta distribution
   * with parameters A and B.  The density of the beta is x^(a−1) *
   * (1−x)^(b−1) / B(a,b) for 0 < x < 1
   *
   * Arguments
   *      aa −−> First parameter of the beta distribution
   *      bb −−> Second parameter of the beta distribution Method
   *
   * Method: described in R. C. H. Cheng Generating Beta Variatew with
   *      Nonintegral Shape Parameters Communications of the ACM,
   *      21:317−322 (1978) (Algorithms BB and BC)
   *)
  (# aa, bb: @real;
     random: @real
  enter (aa,bb)
  exit random
```

```
  #);


genchi: external
  (*
   * GENerate random value of CHIsquare variable
   *      Generates random deviate from the distribution of a
   * chisquare with DF degrees of freedom random variable.
   *
   * Arguments
   *      df −−> Degrees of freedom of the chisquare (Must be
   * positive)
   *
   * Method:
   *      Uses relation between chisquare and gamma.
   *)
  (# df: @real;
     random: @real
  enter df
  exit random
  #);


genexp: external
  (*
   * GENerate EXPonential random deviate
   *      Generates a single random deviate from an exponential
   * distribution with mean AV.
   *
   * Arguments
   *      av −−> The mean of the exponential distribution from which a
   *             random deviate is to be generated.  Method:
   *
   * Renames SEXPO from TOMS as slightly modified by BWB to use
   * RANF instead of SUNIF.
   * For details see: Ahrens, J.H. and Dieter,
   * U.  Computer Methods for Sampling From the Exponential and Normal
   * Distributions.  Comm.  ACM, 15,10 (Oct. 1972), 873 − 882.
   *)
  (# av: @real;
     random: @real
  enter av
  exit random
  #);


genf: external
  (*
   * GENerate random deviate from the F distribution
   *      Generates a random deviate from the F (variance ratio)
   * distribution with DFN degrees of freedom in the numerator and DFD
   * degrees of freedom in the denominator.
   *
   * Arguments
   *      dfn −−> Numerator degrees of freedom (Must be positive)
   *      dfd −−> Denominator degrees of freedom (Must be positive)
   *
   * Method
   *      Directly generates ratio of chisquare variates
   *)
  (# dfn, dfd: @real;
     random: @real
  enter (dfn, dfd)
  exit random
  #);
```

```
gengam: external
  (*  GENerates random deviates from GAMma distribution
   *      Generates random deviates from the gamma distribution whose
   * density is (A**R)/Gamma(R) * X**(R−1) * Exp(−A*X)
   *
   * Arguments
   *      a −−> Location parameter of Gamma distribution
   *      r −−> Shape parameter of Gamma distribution
   *
   * Method
   *      Renames SGAMMA from TOMS as slightly modified by BWB to use
   * RANF instead of SUNIF.  For details see: (Case R >= 1.0) Ahrens,
   * J.H. and Dieter, U.  Generating Gamma Variates by a Modified
   * Rejection Technique.  Comm. ACM, 25,1 (Jan. 1982), 47 − 54.
   * Algorithm GD (Case 0.0 <= R <= 1.0) Ahrens, J.H. and Dieter, U.
   * Computer Methods for Sampling from Gamma, Beta, Poisson and
   * Binomial Distributions.  Computing, 12 (1974), 223−246/ Adapted
   * algorithm GS.
   *)
  (# a, r: @real;
     random: @real
  enter (a, r)
  exit random
  #);


gennch: external
  (*  GENerate random value of Noncentral CHisquare variable
   *      Generates random deviate from the distribution of a
   * noncentral chisquare with DF degrees of freedom and noncentrality
   * parameter xnonc.
   *
   * Arguments
   *      df −−> Degrees of freedom of the chisquare (Must be > 1.0)
   *      xnonc −−> Noncentrality parameter of the chisquare (Must be
   *                >= 0.0)
   * Method
   *      Uses fact that noncentral chisquare is the sum of a
   * chisquare deviate with DF−1 degrees of freedom plus the square of
   * a normal deviate with mean XNONC and standard deviation 1.
   *)
  (# df, xnonc: @real;
     random: @real
  enter (df, xnonc)
  exit random
  #);


gennf: external
  (*  GENerate random deviate from the Noncentral F distribution
   *      Generates a random deviate from the noncentral F (variance
   * ratio) distribution with DFN degrees of freedom in the numerator,
   * and DFD degrees of freedom in the denominator, and noncentrality
   * parameter XNONC.
   *
   * Arguments
   *      dfn −−> Numerator degrees of freedom (Must be >= 1.0)
   *      dfd −−> Denominator degrees of freedom (Must be positive)
   *      xnonc −−> Noncentrality parameter (Must be nonnegative)
   *
   * Method
   *      Directly generates ratio of noncentral numerator chisquare
   * variate to central denominator chisquare variate.
   *)
  (# dfn, dfd, xnonc: @real;
```

```
     random: @real
  enter (dfn, dfd, xnonc)
  exit random
  #);


gennor: external
  (*
   * GENerate random deviate from a NORmal distribution
   *      Generates a single random deviate from a normal distribution
   * with mean, AV, and standard deviation, SD.
   *
   * Arguments
   *      av −−> Mean of the normal distribution.
   *      sd −−> Standard deviation of the normal distribution.
   *
   * Method
   *      Renames SNORM from TOMS as slightly modified by BWB to use
   * RANF instead of SUNIF.  For details see: Ahrens, J.H. and Dieter,
   * U.  Extensions of Forsythe's Method for Random Sampling from the
   * Normal Distribution.  Math. Comput., 27,124 (Oct. 1973), 927 −
   * 937.
   *)
  (# av, sd: @real;
     random: @real
  enter (av, sd)
  exit random
  #);


sexpo: external
  (*
   * Standard EXPonential distribution
   *
   * Method
   *      For details see: Ahrens, J.H. and Dieter, U.  Computer
   * Methods For Sampling From The Exponential And Normal
   * Distributions.  COMM. ACM, 15,10 (Oct. 1972), 873 − 882.  All
   * statement numbers correspond to the steps of algorithm 'SA' in
   * the above paper (slightly modified implementation) Modified by
   * Barry W. Brown, Feb 3, 1988 to use RANF instead of SUNIF.  The
   * argument IR thus goes away.
   *
   *      Q(N) = SUM(ALOG(2.0)**K/K!)  K=1,..,N , The highest N (here
   * 8) is determined by Q(N)=1.0 within standard precision
   *)
  (# random: @real
  exit random
  #);


sgamma: external
  (*
   * Standard GAMMA distribution
   *
   *      Sample from the GAMMA−(A)−distribution coefficients Q(K)
   *         − for Q0 = SUM(Q(K)*A**(−K)) coefficients A(K)
   *         − for Q = Q0+(T*T/2)*SUM(A(K)*V**K) coefficients E(K)
   *         − for EXP(Q)−1 = SUM(E(K)*Q**K)
   *
   * Arguments
   *      A −−−> Parameter (mean) of the standard GAMMA distribution
   * Method
   *      CASE A >= 1.0 !
   *      For details see: Ahrens, J.H. and Dieter, U.  Generating
   * GAMMA variates by a modified rejection technique.  COMM. ACM,
```

```
   * 25,1 (Jan.  1982), 47 − 54.  Step numbers correspond to algorithm
   * 'GD' in the above paper (straightforward implementation) Modified
   * by Barry W. Brown, Feb 3, 1988 to use RANF instead of SUNIF.  The
   * argument IR thus goes away.
   *      CASE 0.0 < A < 1.0 !
   *      For details see: Ahrens, J.H. and Dieter, U.  Computer
   * Methods for Sampling from GAMMA, BETA, Poisson and Binomial
   * Distributions.  COMPUTING, 12 (1974), 223 − 246.  (adapted
   * implementation of algorithm 'GS' in the above paper)
   *)
  (# a: @real;
     random: @real
  enter a
  exit random
  #);


snorm: external
  (*
   * Standard NORmal distribution
   *
   * Method
   *       For details see: Ahrens, J.H. and Dieter, U.  Extensions of
   * Forsythe's method for Random Sampling from the NORMAL
   * distribution.  MATH.  COMPUT., 27,124 (OCT. 1973), 927 − 937.
   * All statement numbers correspond to the steps of algorithm 'FL'
   * (M=5) in the above paper (slightly modified implementation)
   * Modified by Barry W. Brown, Feb 3, 1988 to use RANF instead of
   * SUNIF.  The argument IR thus goes away.
   *)
  (# random: @real
  exit random
  #)
```

13.14 Regexp Interface

```
ORIGIN 'betaenv';
LIB_DEF 'regexp' '../lib';
BODY 'private/regexplib';
(*
 * COPYRIGHT
 *       Copyright (C) Mjolner Informatics, 1992-96
 *       All rights reserved.
 *)
--- textlib: attributes ---
regexp_operation:
  (* generic superpattern for all regexp text operations:
   *     regexp_match, regexp_search, regexp_replace,
   *     regexp_replace_literally.
   * regexp_string: text string containing the regexp.
   * start: start position for match in THIS(text).
   *        Default: pos
   * limit: end position for match in THIS(text).
   *        Default: length
   * posToMatchEnd: if true, move THIS(text).pos to the end of the
   *                matched substring.
   *        Default: false
   * regs: structure for getting access to the matched substring.
   * noMatch: invoked if no matches are found.
   * regexpError: is invoked if syntax error occurs in the specified
   * regexp.
   *
   * value: true, if any match is found.
   *)
  (# regexp_string: ^text;
     start:< integerObject(# do pos -> value; INNER #);
     limit:< integerObject(# do length -> value; INNER #);
     posToMatchEnd:< booleanObject;
     regs: @regexp_registers;
     noMatch:< Notification;
     regexpError:< Exception(# do 'Syntax error in regular expression'->msg; INNER #);
     value: @boolean;
     private: @(* private *)...
  enter (#
        enter regexp_string[]
        do ...
        #)
  do INNER regexp_operation
  exit value
  #);

regexp_match: regexp_operation
(* Takes a regexp as enter parameter (in the form of a reference to a
 * text, containing the regexp.  Matches THIS(text) against the
 * regexp.  INNER is executed if THIS(text) matches the regexp, and
 * the virtual notification noMatch is invoked otherwise.  Returns
 * true if a match is found, false otherwise.  The regexp must be
 * found starting at the current position of THIS(text).
 *)
(# do ... #);

regexp_search: regexp_operation
(* Like regexp_match, except that the match is allowed to be found
 * anywhere between the current position and the end of THIS(text).
 *)
(# do ... #);

regexp_replace: regexp_search
```

```
(* Like regexp_search, except that it takes a second enter parameter,
 * replace_string.  Regexp_replace searches for the regexp, and
 * replaces the matched substring of THIS(text) with the replacement
 * string.  The replacement string may contain \0, \1, ..., \9,
 * representing the substring matched by the i'th parenthesis in the
 * regexp.  \0 represents the entire substring matched.  INNER is
 * executed after the replace have taken place.
 *)
(# replace_string: ^text;
enter replace_string[]
do ...
#);

regexp_replace_global:
  (* replaces all occurences of m with r in THIS(text) using
   * regexp_replace, starting from THIS(text).pos.
   *)
  (# m,r: ^Text; more: @boolean;
     replaceOp: @regexp_replace
     (# noMatch:: (# do false->more #);
        posToMatchEnd:: (# do true->value #)
     #);
  enter (m[],r[])
  do true->more;
     (m[],r[])->replaceOp;
     loop:
       (if more and not eos then
           replaceOp;
           restart loop
       if)
  #);

regexp_replace_literally: regexp_search
(* Like regexp_replace, except that the replacement string is taken
 * literally (i.e \0, \1, etc. are not substituted with any matched
 * substrings).
 *)
(# replace_string: ^text
enter replace_string[]
do ...
#);

regexp_replace_literally_global:
  (* replaces all occurences of m with r in THIS(text) using
   * regexp_replace_literally, starting from THIS(text).pos.
   *)
  (# m,r: ^Text; more: @boolean;
     replaceOp: @regexp_replace_literally
     (# noMatch:: (# do false->more #);
        posToMatchEnd:: (# do true->value #)
     #);
  enter (m[],r[])
  do true->more;
     (m[],r[])->replaceOp;
     loop:
       (if more and not eos then
           replaceOp;
           restart loop
       if)
  #);

--- lib: attributes ---
regexp_numberOfRegisters: (# exit 10 #);

regexp_registers: Cstruct
  (* Structure for accessing the substrings matched by some regexp. *)
```

```
(# getRegisterValue:
     (# regNr, value: @integer;
        pos:< integerValue;
        thePos: @pos (* private: for efficiency *)
     enter regNr
     do ...
     exit value
     #);
   start: @getRegisterValue
     (# pos::< (# do 0 -> value #) #);
   end: @getRegisterValue
     (# pos::< (# do 40 -> value #) #);
   byteSize::< (* private *) (# do regexp_numberOfRegisters*2*4 -> value #)
#)
```

13.15 Pcre Interface

```
ORIGIN 'betaenv';
BODY 'private/pcrebody';
(*
 * COPYRIGHT
 *       Copyright (C) Mjolner Informatics, 2000,2001,2002
 *       All rights reserved.
 *       Written by Erik Corry
 *)
--- lib: attributes ---

(*
 * Perl compatible regular expressions, based on Philip Hazel's PCRE
 * library.   See his documentation and perl documentation for details.
 * To activate the /i /x /m or /s options you can use the inline notation
 * (?x) notation either at the top level of the regular expression or
 * in a subexpression.  You can disable the options again with (?-x). You
 * can also use the comments below.
 *
 * See also pcreDemo.bet in the basiclib/demo/pcre directory for some uses
 * for this stuff.
 *)

(* HOW TO DO SOME TYPICAL PERL THINGS
 *
 * Here are a few things that are very easy to do in perl with the
 * equivalent using BETA's perl-compatible regular expression support.
 * As you can see, the BETA version is often a little longer - this is
 * the penalty you pay for having a general purpose language.  You can
 * save some space at the expense of readability and perhaps efficiency
 * by initialising the Pcre object inline.
 *
 * Assume
 * pre: @Pcre;
 * ok: @boolean;
 *
 * Desc:  Test whether a string matches a pattern
 * Perl:  if $sample =~ /trigger/ ...
 * BETA:  'trigger' -> pre;
 *        (if sample[] -> pre.match then ... if)
 * Alternative:
 *        (if sample[] -> ('trigger' -> Pcre).match then ... if)
 *
 * Desc:  Replace a text in a string with another text
 * Perl:  $sample =~ s/gun/pistol/;
 * BETA:  'gun' -> pre;
 *        (sample[], 'pistol') -> pre.replace -> (ok, sample[]);
 *
 * For /g use replaceAll instead of replace
 * For /e use rep, see HTMLise in pcreDemo in ~beta/basiclib/demo/pcre
 *
 * Desc:  Test for case insensitive match
 * Perl:  if $sample =~ /trigger/i ...
 * BETA:  '(?i)trigger' -> pre;
 *        (if sample[] -> pre.match then ... if);
 * Alternative:
 *        'trigger' -> pre (# options:: (# do CASELESS #) #);
 *        (if sample[] -> pre.match then ... if);
 * Likewise for /x
 *
 * Desc:  Split an input line three ways into fields using : as separator
 * Perl:  ($wordone, $wordtwo, $rest) = split(/:/, $sample, 3);
 * BETA:  sample[] -> (':' -> Pcre).matchAll
```

```
 *        (#
 *           post:: (# do sp1 -> wordone[];
 *                       sp2 -> wordtwo[];
 *                       rest3 -> rest[];
 *                  #)
 *        #)
 * Alternative:
 *        sample[] -> (':' -> Pcre).matchAll
 *        (# post:: (# do ways3 -> (wordone[], wordtwo[], rest[]) #) #);
 *)



Pcre:
  (#
     <<SLOT PcreLib: attributes>>;

     compilation_error:< Exception
       (# regexp: ^text;
          errortext: ^text;
       enter (regexp[],errortext[])
       ...
       #);

     (* Options:  See pcre.h and doc *)
     pcre_CASELESS:       (# exit   1 #);
     pcre_MULTILINE:      (# exit   2 #);
     pcre_DOTALL:         (# exit   4 #);
     pcre_EXTENDED:       (# exit   8 #);
     pcre_ANCHORED:       (# exit  16 #);
     pcre_DOLLAR_ENDONLY: (# exit  32 #);
     pcre_EXTRA:          (# exit  64 #);
     pcre_NOTBOL:         (# exit 128 #);
     pcre_NOTEOL:         (# exit 256 #);
     pcre_UNGREEDY:       (# exit 512 #);
     pcre_NOTEMPTY:       (# exit 1024 #);

     pcre_NONBETAOPTIONS: (# exit 65535 #);

     (* Only in BETA library version *)
     (* Use non-localised English char classes *)
     (* You have to set this when compiling the regexp, not when matching *)
     pcre_C_LOCALE:       (# exit 65536 #);
     (* Study the regular expression after compiling it *)
     (* You have to set this when compiling the regexp, not when matching *)
     pcre_DO_STUDY:       (# exit 131072 #);
     (* Give none instead of zero length strings for cases where there is no
      * match.  This is more correct, but you have to program more carefully
      * to avoid runtime errors.
      *)
     pcre_RETURN_NONE:        (# exit 262144 #);

     pcre_MATCHOPTIONS:       (# exit pcre_NOTBOL %Bor
                                 pcre_NOTEOL %Bor
                                 pcre_NOTEMPTY %Bor
                                 pcre_RETURN_NONE #);

     (* For internal use *)
     pcre_INFO_OPTIONS:      (# exit  0 #);
     pcre_INFO_SIZE:         (# exit  1 #);
     pcre_INFO_CAPTURECOUNT: (# exit  2 #);
     pcre_INFO_BACKREFMAX:   (# exit  3 #);
     pcre_INFO_FIRSTCHAR:    (# exit  4 #);
     pcre_INFO_FIRSTTABLE:   (# exit  5 #);
     pcre_INFO_LASTLITERAL:  (# exit  6 #);
```

```
pcre_ERROR_NOMATCH:     (# exit  −1 #);
pcre_ERROR_NULL:        (# exit  −2 #);
pcre_ERROR_BADOPTION:   (# exit  −3 #);
pcre_ERROR_BADMAGIC:    (# exit  −4 #);
pcre_ERROR_UNKNOWN_NODE:(# exit  −5 #);
pcre_ERROR_NOMEMORY:    (# exit  −6 #);
pcre_ERROR_NOSUBSTRING: (# exit  −7 #);

(* Private internal state *)
private: @...;

(* Read−only for users of pcre.  Tells you how many subpatterns your
 * pattern has.  Only useful if you are reading regular expressions from
 * a config file or from the user, since otherwise you should know this
 * figure already :−]
 *)
subPatterns: @Integer;

(* Specialise this in order to give options when compiling the
 * regular expression and default options when matching.
 *)
options:< integerValue
  (#
     (* Options:  See above *)
     CASELESS:       (# do value %Bor 1 −> value #);
     MULTILINE:      (# do value %Bor 2 −> value #);
     DOTALL:         (# do value %Bor 4 −> value #);
     EXTENDED:       (# do value %Bor 8 −> value #);
     ANCHORED:       (# do value %Bor 16 −> value #);
     DOLLAR_ENDONLY: (# do value %Bor 32 −> value #);
     EXTRA:          (# do value %Bor 64 −> value #);
     NOTBOL:         (# do value %Bor 128 −> value #);
     NOTEOL:         (# do value %Bor 256 −> value #);
     UNGREEDY:       (# do value %Bor 512 −> value #);
     NOTEMPTY:       (# do value %Bor 1024 −> value #);
     C_LOCALE:       (# do value %Bor 65536 −> value #);
     DO_STUDY:       (# do value %Bor 131072 −> value #);
     RETURN_NONE:    (# do value %Bor 262144 −> value #);
     clearCASELESS:       (# do value %Band (%Bnot 1) −> value #);
     clearMULTILINE:      (# do value %Band (%Bnot 2) −> value #);
     clearDOTALL:         (# do value %Band (%Bnot 4) −> value #);
     clearEXTENDED:       (# do value %Band (%Bnot 8) −> value #);
     clearANCHORED:       (# do value %Band (%Bnot 16) −> value #);
     clearDOLLAR_ENDONLY: (# do value %Band (%Bnot 32) −> value #);
     clearEXTRA:          (# do value %Band (%Bnot 64) −> value #);
     clearNOTBOL:         (# do value %Band (%Bnot 128) −> value #);
     clearNOTEOL:         (# do value %Band (%Bnot 256) −> value #);
     clearUNGREEDY:       (# do value %Band (%Bnot 512) −> value #);
     clearNOTEMPTY:       (# do value %Band (%Bnot 1024) −> value #);
     clearC_LOCALE:       (# do value %Band (%Bnot 65536) −> value #);
     clearDO_STUDY:       (# do value %Band (%Bnot 131072) −> value #);
     clearRETURN_NONE:    (# do value %Band (%Bnot 262144) −> value #);

  do 0 −> value;
     INNER;
  #);

init:
  (#
     exp: ^Text;

  enter exp[]
  ...
  #);

match:
```

```
(#
   result: @Integer;
   subMatchCounter: @Integer;
   nextSubMatchIndex:
     (#
     do subMatchCounter = subMatchCounter + 1;
     exit subMatchCounter
     #);

   (* Get (as an integer pair) the position of the text that matched
    * the regular expression in the original text.
    *)
   matchPos:
     (#
        start: @Integer;
        end: @Integer;
     ...
     exit (start, end)
     #);

   (* Get (as a text reference) the text that matched the regular
    * expression.
    *)
   matchText:
     (#
        result: ^Text;
     do
        matchPos -> subject.sub -> result[];
     exit result[]
     #);

   (* Get (as a text reference) the text before the text that matched
    * the regular expression.
    *)
   preMatchText:
     (#
        result: ^Text;
     ...
     exit result[]
     #);

   (* Get (as a text reference) the text after the text that matched
    * the regular expression.
    *)
   postMatchText:
     (#
        result: ^Text;
     ...
     exit result[]
     #);

   (* Get (as an integer pair) the position of the nth submatch in the
    * original text.  You get (0,0) if the nth subpattern didn't match.
    * (It is possible that the nth subpattern didn't match, even if
    * the pattern as a whole matched.  This is different from the
    * subpattern matching an empty string.)
    *)
   subMatchPos:
     (#
        index: @Integer;
        start: @Integer;
        end: @Integer;
     enter index
     ...
     exit (start, end)
     #);
```

```
(* Get (as an integer pair) the position of the next submatch in the
 * original text.  You get (0,0) if the next subpattern didn't match.
 * (It is possible that the nth subpattern didn't match, even if
 * the pattern as a whole matched.  This is different from the
 * subpattern matching an empty string.)
 *)
nextSubMatchPos:
  (#
  exit nextSubMatchIndex -> subMatchPos
  #);

(* Get (as a text reference) the position of the nth submatch in the
 * original text.  You get NONE if the nth subpattern didn't match and
 * you set the option.
 * (It is possible that the nth subpattern didn't match, even if
 * the pattern as a whole matched.  This is different from the
 * subpattern matching an empty string.)
 *)
subMatchText:
  (#
     index: @Integer;
     start: @Integer;
     end: @Integer;
     result: ^Text;
  enter index
  ...
  exit result[]
  #);

(* Get (as a text reference) the position of the next submatch in the
 * original text.  You get NONE if the next subpattern didn't match
 * and you set the option.
 * (It is possible that the nth subpattern didn't match, even if
 * the pattern as a whole matched.  This is different from the
 * subpattern matching an empty string.)
 *)
nextSubMatchText:
  (#
  exit nextSubMatchIndex -> subMatchText
  #);

(*
 * Shorthand methods to get a given matched subpattern
 * You get NONE if the given subpattern didn't match and you set the
 * option.
 * (It is possible that the subpattern didn't match, even if
 * the pattern as a whole matched.  This is different from the
 * subpattern matching an empty string.)
 *)
sub1: (# exit 1 -> subMatchText #);
sub2: (# exit 2 -> subMatchText #);
sub3: (# exit 3 -> subMatchText #);
sub4: (# exit 4 -> subMatchText #);
sub5: (# exit 5 -> subMatchText #);
sub6: (# exit 6 -> subMatchText #);
sub7: (# exit 7 -> subMatchText #);
sub8: (# exit 8 -> subMatchText #);
sub9: (# exit 9 -> subMatchText #);

(* Gets called if there is no match at all.  I'm sure you can think
 * of something useful to put here.
 *)
noMatch:<(# do INNER; #);

(* Specialise this in order to start at a position other than the
```

```
       * start of the string
       *)
      position:< integerValue
        (#
        do 1 −> value;
           INNER;
        #);

      (* Specialise this in order to give options when executing the
       * regular expression.  Doesn't work for options used to compile
       * the regular expression, you had to give them earlier.  If you
       * don't specialise this then you get the global options for this
       * pcre object.
       *)
      options:< integerValue
        (#
           (* Options:  See above
            * Only the options that are useful at match−time (as opposed to
            * init−time) are here
            *)
           NOTBOL:         (# do value %Bor 128 −> value #);
           NOTEOL:         (# do value %Bor 256 −> value #);
           NOTEMPTY:       (# do value %Bor 1024 −> value #);
           RETURN_NONE:    (# do value %Bor 262144 −> value #);
           clearNOTBOL:    (# do value %Band (%Bnot 128) −> value #);
           clearNOTEOL:    (# do value %Band (%Bnot 256) −> value #);
           clearNOTEMPTY:  (# do value %Band (%Bnot 1024) −> value #);
           clearRETURN_NONE:(# do value %Band (%Bnot 262144) −> value #);
        do THIS(pcre).options %Band pcre_MATCHOPTIONS −> value;
           INNER;
        #);

      (* Called before the first match is attempted
       *)
      pre:<
        (# do INNER; #);

      (* match
       * Enter a text reference into the regular expression.  Returns true or
       * false according to whether the text matched the expression.  Executes
       * INNER if there is a match.
       *)
      subject: ^Text;
      matched: @Boolean;
      opt: @Integer;
      psn: @Integer;
    enter subject[]
    ...
    exit (matched)
    #);

  (*
   * matchAll: match
   * Keeps matching as many times as possible until there are no more matches
   * or the end of the string is reached.  Returns true if at least one match
   * occurs.  Calls INNER for each match.
   *)
  matchAll: match
    (#
       privatema: @...;

       (* The number og matches we have so far.  This can be queried in split
        * (where it is always one less than the number of matches except the
        * last time) or in the INNER part of matchAll, where it is accurate.
        *)
       matches: @Integer;
```

```
pre::<
  (#
  ...
  #);

(* Get (as a text reference) the text after the previous match (if any)
 * but before the text that matched the regular expression this time
 * around.
 *)
splitText:
  (#
     result: ^Text;
  do
     splitPos -> subject.sub -> result[];
  exit result[]
  #);

(* Get (as an integer pair) the position of the text after the previous
 * match (if any) but before the text that matched the regular
 * expression this time around.
 *)
splitPos:
  (#
     start: @Integer;
     end: @Integer;
  ...
  exit (start, end)
  #);

(* Gets called once for each split and once at the end.  You can call
 * splitText and splitPos from here to do something with the split
 * strings.  Gets called only once if the pattern doesn't match at all.
 *)
split:<
  (#
     thismatch: @Integer;
  ...
  #);

(* Make sure split and post get called at least once even if there
 * is no match at all.  You can add code here if you want to do
 * something whenever there is no match at all.
 *)
noMatch::<
  (#
  ...
  #);

(* Gets called once at the end.  You can call
 * splitText and splitPos from here to do something with the rest
 * You can also call spn, sp1, sp2, etc to get the first,
 * second etc. split text.  Restn, rest1, rest2 are similar, but they
 * get the rest of the string from the start of the nth split text to
 * the end.
 *)
post:<
  (#
     spn:<
       (#
          num: @Integer;
          result: ^Text;
       enter num
       ...
       exit result[]
       #);
```

```
            sp1: (# exit 1−> spn #);
            sp2: (# exit 2−> spn #);
            sp3: (# exit 3−> spn #);
            sp4: (# exit 4−> spn #);
            sp5: (# exit 5−> spn #);
            sp6: (# exit 6−> spn #);
            sp7: (# exit 7−> spn #);
            sp8: (# exit 8−> spn #);
            sp9: (# exit 9−> spn #);
            restn:<
              (#
                 num: @Integer;
                 result: ^Text;
              enter num
              ...
              exit result[]
              #);
            rest1: (# exit 1−> restn #);
            rest2: (# exit 2−> restn #);
            rest3: (# exit 3−> restn #);
            rest4: (# exit 4−> restn #);
            rest5: (# exit 5−> restn #);
            rest6: (# exit 6−> restn #);
            rest7: (# exit 7−> restn #);
            rest8: (# exit 8−> restn #);
            rest9: (# exit 9−> restn #);
            ways2: (# exit (sp1, rest2) #);
            ways3: (# exit (sp1, sp2, rest3) #);
            ways4: (# exit (sp1, sp2, sp3, rest4) #);
            ways5: (# exit (sp1, sp2, sp3, sp4, rest5) #);
            ways6: (# exit (sp1, sp2, sp3, sp4, sp5, rest6) #);
            ways7: (# exit (sp1, sp2, sp3, sp4, sp5, sp6, rest7) #);
            ways8: (# exit (sp1, sp2, sp3, sp4, sp5, sp6, sp7, rest8) #);
            ways9: (# exit (sp1, sp2, sp3, sp4, sp5, sp6, sp7, sp8, rest9) #);
          do INNER;
          #);

    ...
    #);

(*
 * Replace: match
 * Enter a reference to a text and a replacement string.  Exits a sucess
 * boolean and a text reference to the new string.  If there is no match
 * then false, plus a reference to a copy of the original string is exited.
 *)
replace: match
    (#
       (* By overriding this you can put a different value in replacement,
        * so that the replacement text can be calculated dynamically (based
        * on eg. the contents or position of the matched or submatched texts).
        * (You can call matchText to get the text that matched)
        *)
       rep:< textObject;
       replacement: ^Text;
       new: ^Text;
       check:
         (#
         do (if new[] = NONE then subject.copy −> new[] if);
         exit new[]
         #);
    enter replacement[]
    ...
    exit check
    #);
```

```
   (*
    * ReplaceAll: matchAll
    * Enter a reference to a text and a replacement string.  Exits a sucess
    * boolean and a text reference to the new string.  If there is no match
    * then false, plus a reference to a copy of the original string is exited.
    *)
   replaceAll: matchAll
     (#
        (* By overriding this you can put a different value in replacement,
         * so that the replacement text can be calculated dynamically (based
         * on eg. the contents or position of the matched or submatched texts).
         * (You can call matchText to get the text that matched)
         *)
        rep:< textObject;

        post::<
          (#
          do (if new[]=NONE then &text[]->new[] if);
             splitText -> new.append;
             INNER;
          #);

        replacement: ^Text;
        new: ^Text;
     enter replacement[]
     do
        (if new[] = NONE then
            splitText -> new[];
            subject.lgth -> new.extend;
         else
            splitText -> new.append;
        if);
        replacement[] -> rep -> new.append;
        INNER;
     exit
        (#
        do
           (if new[] = NONE then subject.copy -> new[] if);
        exit new[]
        #)
     #);

   (* Pcre itself enters init (which takes a text reference and compiles it
    * to a regular expression) and exits a reference to itself, which lets you
    * dynamically create a regexp and call a method on it in one line
    *)
enter init
exit this(Pcre)[]
#)
```

13.16 Substreams Interface

```
ORIGIN 'betaenv';
BODY 'private/substreamsbody'
(*
 * COPYRIGHT
 *       Copyright (C) Mjolner Informatics, 1995-96
 *       All rights reserved.
 *
 * This fragment implements a substream and a subtext pattern
 * The operations of substream and subtext are essentially the same
 * as the operations on stream, respectively, text.  See these patterns
 * in betaenv for a description of the semantics of the operations.
 *)
--- lib: attributes ---
substream: stream
  (# <<SLOT substreamLib: attributes>>;
     streamType:< stream;
     stm: (* the attached stream *) ^streamType;
     high, low: @integer;
     init:
       (#
       enter (stm[], low, high)
       do check
       #);
     attach:
       (# enter stm[] #);
     range:
       (#
       enter (low,high)
       do check; high-low+1->lgth
       exit (low,high)
       #);
     check:
       (#
       do (if true
           // stm[]=NONE then notAttachedError
           // low>high then illegalRangeError
           // low<0 then illegalRangeError
           // low>stm.length then illegalRangeError
           // high>stm.length then illegalRangeError
          if)
       #);
     length::<(# ... #);
     eos::<
       (# ... #);
     empty: booleanValue
       (# ... #);
     copy:
       (# theCopy: ^substream;
          copyI: @...
       do copyI
       exit theCopy[]
       #);
     notAttachedError:< StreamException
       (* Raised when this(substream) is not attached to any stream.
        * Message: "The substream is not attached to any stream!".
        *)
       (# ... #);
     illegalRangeError:< StreamException
       (* Raised when the substream range specified does not lie
        * within the range of the attached stream.  Message: "The
        * substream range is not within the attached stream range!".
        *)
```

```
      (# ... #);
    indexError:< StreamException
      (* Raised when the index goes outside the range of the
       * substream.  Message: "Index error in stream! (index <num>)".
       *)
      (# inx: @integer
      enter inx ...
      #);
    EOSerror::<
      (* Raised from Get and Peek when the end of the substream is
       * passed.
       *)
      (# ... #);
    otherError::<
      (* Raised when an error other than the Index−/EOSerror occurs.
       *)
      (# ... #);
    setpos::<
      (# ... #);
    getpos::<
      (# ... #);
    pos, lgth: (* private *) @integer;
  enter (attach,range)
  #);

subtext: substream
  (# <<SLOT subtextLib: attributes>>;
    streamType::< text;
    put::<
      (# ... #);
    putText::<
      (# ... #);
    get::<
      (# ... #);
    getAtom::<
      (# ... #);
    getLine::<
      (# ... #);
    peek::<
      (# ... #);
    clear:
      (# ... #);
    inxGet: charValue
      (# i: @integer
      enter i
      ...
      #);
    inxPut:
      (# ch: @char; i: @integer
      enter (ch,i)
      ...
      #);
    append:
      (# S1: ^streamType; S1lgt: @integer
      enter S1[]
      ...
      #);
    prepend:
      (# S1: ^streamType; S1lgt: @integer
      enter S1[]
      ...
      #);
    scanAll:
      (# ch: @char
      ...
      #);
```

```
sub:
  (# i,j: @integer
  enter (i,j)
  exit (i,j)->stm.sub
  #);
insert:
  (# T1: ^streamType;
     inx: @integer;
     P: @integer
  enter (T1[],inx)
  ...
  #);
delete:
  (# i,j: @integer
  enter (i,j)
  ...
  #);
equal: booleanValue
  (# S1: ^streamType;
     S1pos, lgt: @integer;
     NCS:< booleanObject
  enter S1[]
  ...
  #);
equalNCS: equal
  (* As 'equal', except the the comparison will be done Non
   * Case Sensitive
   *)
  (# NCS:: trueObject #);
less: booleanValue
  (# S1: ^streamType;
     S1pos, lgt: @integer; ch, ch1: @char
  enter S1[]
  ...
  #);
greater: booleanValue
  (# S1: ^streamType;
     S1pos, lgt: @integer; ch, ch1: @char
  enter S1[]
  ...
  #);
makeLC:
  (# ... #);
makeUC:
  (# ... #);
find:
  (# ch: @char;
     inx: @integer;
     NCS:< booleanObject;
     from:< integerObject(# do pos->value; INNER from #)
  enter ch
  ...
  #);
findAll: find
  (# from:: (# do 0->value #)
  do INNER findAll
  #);
findText:
  (# txt: ^text;
     inx: @integer;
     NCS:< booleanObject;
     from:< integerObject(# do pos->value; INNER from #)
  enter txt[]
  ...
  #);
findTextAll: findText
```

```
      (# from:: (# do 0->value #)
      do INNER findTextAll
      #);
  exit stm.T[low:high]
  #);

putSubStream: screen.putSubstream(# do INNER putSubstream #)

--- textLib: attributes ---
subtxt:
  (# low,high: @integer;
     theSubtext: ^subtext
  enter (low,high)
  do &subtext[]->theSubtext[];
     (this(text)[],(low,high))->theSubtext;
     theSubtext.init
  exit theSubtext[]
  #)

--- streamLib: attributes ---
putSubstream:
  (# stm: ^substream;
     p: (*private*)@integer
  enter stm[]
  ...
  #)
```

13.17 Systemenv Interface

```
ORIGIN 'basicsystemenv';
BODY 'private/systemenvbody';
(*
 * COPYRIGHT
 *       Copyright (C) Mjolner Informatics, 1984-96
 *       All rights reserved.
 *
 * Use this fragment as the ORIGIN for concurrent BETA
 * programs NOT using X libraries or other UI libraries
 * with a central event-loop.
 *
 * Programs should look something like:
 *
 * ORIGIN 'systemenv';
 * --- program:descriptor ---
 * systemEnv:
 *   (# ...
 *   do ...
 *   #)
 *
 * For details about the concurrency abstractions,
 * see the file basicsystemenv.bet
 *)
```

13.18 TextUtils Interface

```
ORIGIN 'betaenv';
LIB_DEF 'textUtils' '../lib';
INCLUDE 'numberio'
(*
 * COPYRIGHT
 *       Copyright (C) Mjolner Informatics, 1996
 *       All rights reserved.
 *)
--- streamLib: attributes ---

getBoolean:
  (# trueValue:< (# value: ^text do 'true'->value[]; INNER exit value[] #);
     falseValue:< (# value: ^text do 'false'->value[]; INNER exit value[] #);
     syntaxError:< streamException
       (#
       do 'getBoolean: Syntax Error - looking at: "'->msg.append;
          t[]->msg.puttext; '"'->msg.putline; INNER
       #);
     value: @boolean;
     t: ^(* private *)text
  do getAtom->t[];
     (if true
      // t[]->(trueValue).equalNCS then true->value
      // t[]->(falseValue).equalNCS then false->value
      else syntaxError
     if);
     INNER getBoolean
  exit value
  #);
putBoolean:
  (# trueValue:< (# value: ^text do 'true'->value[]; INNER exit value[] #);
     falseValue:< (# value: ^text do 'false'->value[]; INNER exit value[] #);
     value: @boolean
  enter value
  do (if value then
         trueValue->puttext
      else
         falseValue->puttext
     if);
     INNER putBoolean
  #);

--- lib: attributes ---

getBoolean: keyboard.getBoolean
  (# do INNER getBoolean #);

putBoolean: screen.putBoolean
  (# do INNER putBoolean #);

--- textLib: attributes ---

set: (* makes THIS( stream) contain only the character: ch *)
  (# ch: @char
  enter ch
  do INNER set; clear; ch->put
  #);
setText:
  (* makes THIS( stream) contain only the text contained in 't' *)
  (# t: ^text
  enter t[]
  do INNER setText; clear; t[]->puttext
```

```
  #);
setInt:
  (* makes THIS( stream) contain only the textual representation of
   * the integer 'i'
   *)
  (# i: @integer
  enter i
  do INNER setInt; clear; i->putint
  #);
setBased:
  (* makes THIS( stream) contain only the textual representation of
   * the number 'value' in base 'base'
   *)
  (# base, value: @integer
  enter (base, value)
  do INNER setBased; clear; (base,value)->putBased
  #);
setReal:
  (* makes THIS( stream) contain only the textual representation of
   * the real number 'r'
   *)
  (# r: @real
  enter r
  do INNER setReal; clear; r->putreal
  #);
setBoolean:
  (* makes THIS( stream) contain only the textual representation of
   * the boolean 'value' (i.e. either 'true' og 'false')
   *)
  (# value: @boolean
  enter value
  do INNER setBoolean; clear; value->putBoolean
  #)
```

13.19 Texthash Interface

```
ORIGIN '~beta/basiclib/betaenv';
LIB_DEF 'texthash' '../lib';
--- lib:attributes ---
(* This is a pattern implementing a hash of a text into an integer.
 *
 * Usage:
 *      ph: @honeyman;
 *      ...
 * do ph.init;
 *    ...
 *    'Some text' -> ph.hash -> hashValue
 *
 * This version is a BETA implementation of a hash function found in:
 *
 *       C News Source, which contains the following statement:
 *
 * "dbz.c  V3.2
 *
 * Copyright 1988 Jon Zeeff (zeeff@b-tech.ann-arbor.mi.us)
 * You can use this code in any manner, as long as you leave my name on it
 * and don't hold me responsible for any problems with it.
 *
 * Hacked on by gdb@ninja.UUCP (David Butler); Sun Jun  5 00:27:08 CDT 1988
 *
 * Various improvments + INCORE by moraes@ai.toronto.edu (Mark Moraes)
 *
 * Major reworking by Henry Spencer as part of the C News project."
 *
 * The following text is the original comment before the hash function:
 *
 * This is a simplified version of the pathalias hashing function.
 * Thanks to Steve Belovin and Peter Honeyman
 *
 * hash a string into a long int.  31 bit crc (from andrew appel).
 * the crc table is computed at run time by crcinit() -- we could
 * precompute, but it takes 1 clock tick on a 750.
 *
 * This fast table calculation works only if POLY is a prime polynomial
 * in the field of integers modulo 2.  Since the coefficients of a
 * 32-bit polynomial won't fit in a 32-bit word, the high-order bit is
 * implicit.  IT MUST ALSO BE THE CASE that the coefficients of orders
 * 31 down to 25 are zero.  Happily, we have candidates, from
 * E. J.  Watson, "Primitive Polynomials (Mod 2)", Math. Comp. 16 (1962):
 *      x^32 + x^7 + x^5 + x^3 + x^2 + x^1 + x^0
 *      x^31 + x^3 + x^0
 *
 * We reverse the bits to get:
 *      11110101000000000000000000000001 but drop the last 1
 *         f   5   0   0   0   0   0   0
 *      01001000000000000000000000000001 ditto, for 31-bit crc
 *         4   8   0   0   0   0   0   0
 *)

honeyman: (* BETA VERSION *)
  (# POLY: (# exit 0x48000000 #);
     CrcTable: [128]@Integer;

     init:
       (# j,sum: @Integer;
       do
          (for i:128 repeat
               0->sum; 6->j;
```

```
            loop:
              (#
              do (if ((i−1) %Band (1 %sll j)) then
                     (sum %Bxor (POLY %srl j))−>sum;
                 if);
                 (if j>0 then j−1−>j; restart loop if);
              #);
            sum−>CrcTable[i];
         for);
      #);

   hash: @
      (# t: ^Text; sum: @Integer;
      enter t[]
      do 0−>sum;
         (for i:t.lgth repeat
            (sum %srl 7) %Bxor CrcTable[((sum %Bxor t.T[i]) %Band 0x7f) + 1]
              −>sum;
         for);
      exit sum
      #);
#)
```

13.20 Timedate Interface

```
ORIGIN '~beta/basiclib/betaenv';
BODY 'private/timedatebody';

(* This library defines a 'time' (and thereby a date) and a 'period'
 * concept.
 *
 * The library defines numerous operations on time and period values,
 * and time objects, including operations for comparisons, addition,
 * subtractions, printing and reading times and periods.
 *
 * Given a time, you can ask for the year, day, etc. components of
 * that time, giving the date-related informations of a time.
 *)

--- lib:attributes ---
time:
  (# <<SLOT timeLib:attributes>>;

     (* Get and set the individual attributes of the time.
      *
      * These are normalised, so that you can overflow attributes into
      * each other.  If you e.g. add 23 hours to a time representing
      * 7 o'clock one day, it becomes 6 o'clock the next day!  If you
      * subtract a day from the 1st of a month, you get the last day
      * of the previous month, and so on.
      *)

     year: (* set/get the year-component of this(time) *)
       (#
       enter (# value: @integer enter value ... #)
       exit (# value: @integer ... exit value #)
       #);
     month: (* set/get the month-component of this(time). Range: [1-12] *)
       (#
       enter (# value: @integer enter value ... #)
       exit (# value: @integer ... exit value #)
       #);
     day: (* set/get the day-component of this(time). Range: [1-31] *)
       (#
       enter (# value: @integer enter value ... #)
       exit (# value: @integer ... exit value #)
       #);
     hour: (* set/get the hour-component of this(time) *)
       (#
       enter (# value: @integer enter value ... #)
       exit (# value: @integer ... exit value #)
       #);
     minute: (* set/get the minute-component of this(time) *)
       (#
       enter (# value: @integer enter value ... #)
       exit (# value: @integer ... exit value #)
       #);
     sec: (* set/get the sec-component of this(time) *)
       (#
       enter (# value: @integer enter value ... #)
       exit (# value: @integer ... exit value #)
       #);
     weekday:
       (* Returns the weekday of this(time) as an integer.
        *    Monday=1, Tuesday=2, etc.Sunday=7
        *)
       (#
```

```
  exit (# value: @integer ... exit value #)
  #);
dayOfYear: integerValue
  (* returns the day of yeat of this(time) *)
  (# doy: @...
  do doy
  #);
weekOfYear: integerValue
  (* Returns the number of the week of this(time). Range: [1−53].
   *    previousYear  is invoked if the week is the last week
   *                  of the previous year
   *    followingYear is invoked if the week is the first week
   *                  of the following year
   *)
  (# previousYear:< notification;
     followingYear:< notification;
     woy: @...
  do woy
  #);
leapYear: booleanValue
  (* Returns true, iff this(time).year is a leap year *)
  (# ly: @...
  do ly
  #);

timePredicate: booleanValue(# other: @time enter other do INNER #);

sameYear:
  (* returns true, iff this(time).year = other.year *)
  timePredicate(# ... #);
sameMonth:
  (* returns true, if sameYear and this(time).month = other.month *)
  timePredicate(# ... #);
sameDay:
  (* returns true, if sameMonth and this(time).day = other.day *)
  timePredicate(# ... #);
sameHour:
  (* returns true, if sameDay and this(time).hour = other.hour *)
  timePredicate(# ... #);
sameMinute:
  (* returns true, if sameHour and this(time).minute = other.minute *)
  timePredicate(# ... #);
sameSec:
  (* returns true, if sameMinute and this(time).sec = other.sec *)
  timePredicate(# ... #);

equal:
  (* returns true, iff this(time) = other *)
  sameSec(# #);

before: (* returns true, iff this(time) < other *)
  timePredicate(# ... #);
after: (* returns true, iff this(time) > other *)
  timePredicate(# ... #);

between: booleanValue
  (* returns true, iff d1 <= this(time) <= d2 *)
  (# d1,d2: @time
  enter (d1,d2)
  ...
  #);

add: (* moves this(time) forward in time by period 'p' *)
  (# p: @period
  enter p
  ...
```

```
      #);

    sub: (* moves this(time) backwards in time by period 'p' *)
      (# p: @period
      enter p
      ...
      #);

    private: @...;

  enter (# year, month, day, hour, minute, sec: @integer;
        enter (year, month, day, hour, minute, sec)
        ... #)
  exit  (# year, month, day, hour, minute, sec: @integer;
        ...
        exit (year, month, day, hour, minute, sec)
        #)
  #);

timeNow: (* get the time right now! *)
  (# t: @time;
  ...
  exit t
  #);

timeNowRef:
  (* as timeNow, except that it returns a time object reference *)
  (# t: ^time
  do &time[]->t[]; timeNow->t;
  exit t[]
  #);

timeMin: (* Mon Nov 24 12:00:00 -4713 *)
  (# t: @time;
  ...
  exit t
  #);

timeMax: (* in principle infinity *)
  (# t: @time;
  ...
  exit t
  #);

timeRange: (* the range within which time objects will be normalized *)
  (# exit (timeMin, timeMax) #);

timeDifference:
  (* calculates the time difference (period) between the two times t1 and t2 *)
  (# t1, t2: @time; d: @period
  enter (t1,t2)
  ...
  exit d
  #);

period:
  (* Periods are used to represent a period of time.
   *
   * Periods are measured in days, hours, minutes and seconds, and is
   * as such independent of months and years, since these measurements
   * are not time invariant (the length of a year varies, and the same
   * applies for months (28, 29, 30 or 31 days).
   *
   * Periods can be used to represent the time difference between two
   * times, e.g. between Nov 28 22:45:15 1996 and Dec 3 23:15:30,
   * which is 05#00:30:15.
```

```
   *
   * Periods are also used to represent a duration, e.g. the duration
   * of a standard lecture, 00#00:45:00.
   *
   * You can add and subtract periods using pAdd and pSub
   * (e.g. '((0,01,30,00),p1)−>pAdd−>p2' adds one hour and 30 minutes
   * to the period 'p1' and returns the result in 'p2').
   *
   * And you can measure the time difference between two times 't1'
   * and 't2' by '(t1,t2)−>timeDifference−>p'.  Finally, you can move
   * a time 't' forward or backwards in time by a given period 'p' by
   * 'p−>t.add' and' p−>t.sub'.
   *)
  (# <<SLOT periodLib: attributes>>;
     days, hours, minutes, seconds: @integer;
  enter (days, hours, minutes, seconds)
  exit (days, hours, minutes, seconds)
  #);

pAdd: (* adds two periods and returns the result *)
  (# p1, p2, p3: @period
  enter (p1, p2)
  do p1.days+p2.days−>p3.days;
     p1.hours+p2.hours−>p3.hours;
     p1.minutes+p2.minutes−>p3.minutes;
     p1.seconds+p2.seconds−>p3.seconds;
  exit p3
  #);

pSub: (* subtracts two periods and returns the result *)
  (# p1, p2, p3: @period
  enter (p1, p2)
  do p1.days−p2.days−>p3.days;
     p1.hours−p2.hours−>p3.hours;
     p1.minutes−p2.minutes−>p3.minutes;
     p1.seconds−p2.seconds−>p3.seconds;
  exit p3
  #);

(* Input/Output utilities:
 *
 * Formats:
 *    Time:   Www Mmm Dd Yyyy Hh:Mm:Ss
 *    Date:   Www Mmm Dd Yyyy
 *    Clock:  Hh:Mm:Ss
 *    Period: Dd#Hh:Mm:Ss
 *
 * where Www  is the weekday (Mon, Tue, etc.)
 *       Mmm  is the month (Jan, Feb, etc.)
 *       Dd   is the date
 *       Hh   is hours
 *       Mm   is minutes
 *       Ss   is seconds
 *       Yyyy is the year
 *)

putTime: (* prints a time value on screen *)
  screen.putTime(# do INNER #);
getTime: (* reads a time value from the keyboard *)
  keyboard.getTime(# do INNER #);

putDate: (* prints the date portion of a time value on screen *)
  screen.putDate(# do INNER #);
getDate:
  (* reads a date value from the keyboard.  The clock part of the
   * time value will be 00:00:00
```

```
   *)
  keyboard.getDate(# do INNER #);

putClock: (* prints the clock portion of a time value on screen *)
  screen.putClock(# do INNER #);
getClock:
  (* reads a clock value from the keyboard.  The date part of the
   * time value will be Jan 01, 0000.
   *)
  keyboard.getClock(# do INNER #);

putPeriod: (* prints a period value on screen *)
  screen.putPeriod(# #);
getPeriod: (* reads a period value from the keyboard *)
  keyboard.getPeriod(# #);

--- streamLib:attributes ---

putTime:  (* prints a time value on this(stream) *)
  (# t: @time;
  enter t
  ...
  #);
getTime: (* reads a time value from this(stream) *)
  (# t: @time;
  ...
  exit t
  #);

putDate: (* prints the date portion of a time value on screen *)
  (# t: @time;
  enter t
  ...
  #);
getDate:
  (* reads a date value from the keyboard.  The clock part of the
   * time value will be 00:00:00
   *)
  (# t: @time;
  ...
  exit t
  #);

putClock: (* prints the clock portion of a time value on screen *)
  (# t: @time;
  enter t
  ...
  #);
getClock:
  (* reads a clock value from the keyboard.  The date part
   * of the time value will be Jan 01, 0000
   *)
  (# t: @time;
  ...
  exit t
  #);

putPeriod: (* prints a period value on this(stream) *)
  (# p: @period
  enter p
  ...
  #);
getPeriod: (* prints a period value on this(stream) *)
  (# p: @period
  ...
  exit p
```

```
#)
```

13.21 Timehandler Interface

```
ORIGIN 'basicsystemenv';
LIB_DEF 'timehandler' '../lib';
BODY 'private/timehandlerbody';
(*
 * COPYRIGHT
 *       Copyright (C) Mjolner Informatics, 1984-96
 *       All rights reserved.
 *
 *)
--- systemLib:attributes ---
timeHandler:
  (* timeHandler handles the setting and unsetting of timers.
   *
   * register takes a time to wait and an object reference. It returns
   * a unique id identifying the registration. If the registration is
   * not unregistered before the timer goes off, the object will be
   * executed.  Due to non-preemptive multitasking, it can only be
   * guaranteed that at least time seconds will elapse before obj is
   * executed.
   *
   * unregister takes a registration id and unregisters the
   * registration.  If this happens before the corresponding timer
   * goes off, the registered object will not be executed.
   *
   * init should be called before using the timeHandler.
   *)
  (# register:
       (# obj: ^Object; time: @Integer;
          id: @Integer;
       enter (obj[],time)
       ...
       exit id
       #);

     unregister:
       (# id: @Integer;
       enter id
       ...
       #);
     init:
       (#
       ...
       #);
     timeHandlerPrivate: @...;
  #)
```

13.22 Wtext Interface

```
ORIGIN 'betaenv';
BODY 'private/wtextbody'
(*
 * COPYRIGHT
 *       Copyright (C) Mjolner Informatics, 1997-98
 *       All rights reserved.
 *
 * This fragment implements a UniCode stream and text concept
 *)

---lib:attributes---
wStream:
  (# <<SLOT wStreamLib: attributes>>;
     length:< integerValue (* returns the length of THIS(wStream) *)
       (#
       do -1->value; INNER length
       #);
     position: (* current position of THIS(wStream) *)
       (#
       enter setPos
       exit getPos
       #);
     eos:< (* returns 'true' if THIS(wStream) is at end-of-wStream *)
       booleanValue;
     reset: (* sets 'position' to zero *)
       (#
       do 0->setPos
       exit THIS(wStream)[]
       #);
     peek:< (* looks at the next character of THIS(wStream) *)
       (# ch: @wchar
       do INNER peek
       exit ch
       #);
     get:< (* reads a character from THIS(wStream) *)
       (# ch: @wchar
       do INNER get
       exit ch
       #);
     getNonBlank:
       (* Reads first non-whitespace character from THIS(wStream).
        * If called at end-of-wStream the character 'ascii.fs' is
        * returned
        *)
       (# ch: @wchar;
          skipblanks: @scanWhiteSpace;
          testEOS: @EOS;
          getCh: @get;
       ...
       exit ch
       #);
     getint: integerValue
       (* Reads an integer: skips whitespace characters and
        * returns the following digits.
        *
        * See numberio.bet for more numerical output operations
        *)
       (# syntaxError:< wStreamException
            (#
            do 'getint: syntax error - looking at: "'->msg.append;
               peek->msg.put; '"'->msg.putline; INNER syntaxError
            #);
```

```
      geti: @...
    do geti; INNER getint
    #);
  getAtom:<
    (* Returns the next atom (i.e. sequence of non−white
     * characters − skipping leading blanks)
     *)
    (# txt: ^wtext;
    do &wText[]−>txt[]; INNER getAtom;
    exit txt[]
    #);
  getline:<
    (* Reads a sequence of characters until nl−character
     * appears and returns the characters read.
     *)
    (# txt: ^wText;
    do &wText[]−>txt[]; INNER getline
    exit txt[]
    #);
  asInt:
    (* converts THIS(wText) to an integer value, ignoring
     * leading and trailing whitespace.  See numberio.bet for
     * more numerical conversion operations.
     *)
    (# i: @integer;
       syntaxError:< wStreamException
         (# peekCh: @wchar
         enter peekCh
         do 'asInt: syntax error − looking at: "'−>msg.append;
            peekCh−>msg.put; '"'−>msg.put;
            INNER syntaxError
         #)
    ...
    exit i
    #);
  put:< (* writes a character to THIS(wStream) *)
    (# ch: @wchar
    enter ch
    do INNER put
    exit THIS(wStream)[]
    #);
  newline: (* writes the nl−character *)
    (#
    do ascii.newline−>put
    exit THIS(wStream)[]
    #);
  putint:
    (* Writes an integer to THIS(wStream); The format may be
     * controlled by the 'signed', 'blankSign', 'width',
     * 'adjustLeft' and 'zeroPadding' variable attributes.
     * 'width' is extended if it is too small.  Examples:
     * '10−>putint' yields: '10'; '10*pi−>putint(# do 10−>width;
     * true−>adjustLeft #)' yields: '10 '; and '10−>putint(# do
     * 10−>width; true−>zeroPadding #)' yields: '0000000010'.
     *
     * See numberio.bet for more numerical output operations
     *)
    (# n: @integer;
       signed: @boolean
         (* If integer is positive, a '+' will always be
          * displayed
          *);
       blankSign: @boolean
         (* If integer is positive, a ' ' space is displayed as
          * the sign.  Ignored if 'signed=true'
          *);
```

```
      width: @integer
        (* Minimum width *);
      adjustLeft: @boolean
        (* Specifies if the number is to be aligned left or
         * right, if padding of spaces is necessary to fill up
         * the specified width.
         *);
      zeroPadding: @boolean
        (* width is padded with leading zero instead of
         * spaces.  Ignored if 'adjustLeft=true'
         *);
      format:< (# do INNER format #);
      puti: @...
    enter n
    do 1->width; format; INNER putint; puti
    exit THIS(wStream)[]
    #);
  putText:< (* Writes a wText to THIS(wStream). *)
    (# txt: ^wText
    enter txt[]
    do (if txt[]<>NONE then INNER puttext if)
    exit THIS(wStream)[]
    #);
  putline:
    (* 'puttext' followed by 'newline' *)
    (# T: ^wText; putT: @puttext; newL: @newline
    enter T[]
    do INNER putline; T[]->putT; newL
    exit THIS(wStream)[]
    #);
  scan:
    (* Scan chars from current position in THIS(wStream) while
     * '(ch->while)=true'; perform INNER for each char being
     * scanned
     *)
    (# while:<
        (# ch: @wchar; value: @boolean
        enter ch
        do true->value; INNER while
        exit value
        #);
      ch: @wchar;
      whilecondition: @while;
      testEOS: @EOS;
      getPeek: @peek;
      getCh: @get;
    ...
    exit THIS(wStream)[]
    #);
  scanWhiteSpace: scan
    (* Scan whitespace characters *)
    (# while::< (# do ch->ascii.isWhiteSpace->value #)
    do INNER scanWhiteSpace
    exit THIS(wStream)[]
    #);
  scanAtom:
    (* Scan until first non-whitespace char.  Scan the next
     * sequence of non-whitespace chars.  Stop at first
     * whitespace char.  For each non-whitespace char an INNER
     * is performed. Usage: 'scanAtom(# do ch-><destination> #)'
     *)
    (# ch: @wchar;
    ...
    exit THIS(wStream)[]
    #);
  scanToNl:
```

```
      (* Scan all chars in current line including newline char *)
      (# ch: @wchar; getCh: @get;
      ...
      exit THIS(wStream)[]
      #);
    wStreamException: exception
      (# do INNER wStreamException #);
    EOSerror:< wStreamException
      (* Raised from 'get' and 'peek' when attempted to read past
       * the end of the wStream.
       *)
      (#
      do 'Attempt to read past end−of−wStream'−>msg.putline;
         INNER EOSerror
      #);
    otherError:< wStreamException
      (* Raised when some other kind of wStream error apart from
       * the one mentioned above occurs.
       *);
    getPos:< (* returns current position of THIS(wStream) *)
      integerValue;
    setPos:< (* sets current position in THIS(wStream) to 'p' *)
      (# p: @integer
      enter p
      do INNER setPos
      exit THIS(wStream)[]
      #)
  #); (* pattern wStream *)

(*****  wText pattern ******************************************)
wText: wStream
  (* A wText is a sequence of characters.  Let 'T: @wText'. The
   * range of 'T' is '[1,T.length]'.  A wText can be initialized by
   * executing 'T.clear' or by assigning it another (initialized)
   * wText.  A wText−constant has the form 'foo'.  The 'wText' pattern
   * is primarily intended for small wTexts but there is no upper
   * limit in the size. However, most of the operations becomes
   * less efficient with larger wTexts.
   *)
  (# <<SLOT wTextLib: attributes>>;
     length::< (* Returns the length of THIS(wText) *)
      (# do lgth−>value; INNER length #);
    eos::<
      (# ... #);
    empty:
      (# exit (lgth = 0) #);
    clear: (* Sets the length and position of THIS(wText) to zero *)
      (#
      do 0−>pos−>lgth
      exit THIS(wText)[]
      #);
    equal: booleanValue
      (* Tests if THIS(wText) is equal to the entered wText.  If
       * 'NCS' is further bound to 'trueObject', the comparison
       * will be done Non Case Sensitive.
       *)
      (# txt: ^wText;
         NCS:< booleanObject
      enter txt[]
      ...
      #);
    equalNCS: equal
      (* As 'equal', except the the comparison will be done Non
       * Case Sensitive
       *)
      (# NCS:: trueObject #);
```

```
less: booleanValue
  (* Tests whether the entered wText 'T1[1: length]' is less
   * than 'THIS(wText)[1: T1.length]'.  The lexicographical
   * ordering is used.
   *)
  (# T1: ^wText
  enter T1[]
  ...
  #);
greater: booleanValue
  (* Tests whether the entered wText 'T1[1: length]' is
   * greater than 'THIS(wText)[1: T1.length]'.  The
   * lexicographical ordering is used.
   *)
  (# T1: ^wText
  enter T1[]
  ...
  #);
peek::<
  (* Returns the character at current position; does not
   * update 'position'
   *)
  (# ... #);
get::<
  (* Returns the character at current position; increments
   * 'position'
   *)
  (# ... #);
inxGet: wcharValue
  (* Returns the character at position 'i' *)
  (# i: @integer;
     iget: @...
  enter i
  do iget
  #);
getAtom::<
  (* Returns the next atom (i.e. sequence of non−white
   * characters − skipping leading blanks)
   *)
  (# ... #);
getline::<
  (* Reads a sequence of characters until nl−character
   * appears and returns the characters read.
   *)
  (# ... #);
put::<
  (* writes the character 'ch' at current position in
   * THIS(wText); increments 'position'
   *)
  (# ... #);
inxPut:
  (* Replaces the character at position 'i' *)
  (# ch: @wchar;
     i: @integer;
     iput: @...
  enter (ch,i)
  do iput
  exit THIS(wText)[]
  #);
puttext::<
  (# ... #);
append:
  (* Appends a wText to THIS(wText); does not change 'position'
   *)
  (# T1: ^wText
  enter T1[]
```

```
  ...
  exit THIS(wText)[]
  #);
prepend:
  (* Inserts the wText in 'T1' in front of THIS(wText); updates
   * current position to 'position+T1.length' if 'position>0'
   *)
  (# T1: ^wText
  enter T1[]
  ...
  exit THIS(wText)[]
  #);
insert:
  (* Inserts a wText before the character at position 'inx'.
   * Note: inx<1 means inx=1; inx>length means inx=length+1.
   * If 'position>=inx' then 'position+T1.length->position'.
   *)
  (# T1: ^wText;
     inx: @integer
  enter (T1[],inx)
  ...
  exit THIS(wText)[]
  #);
delete:
  (* Deletes THIS(wText)[i: j]; updates current position:
   *      i<=position<j => i-1->position
   *      j<=position   => position-(j-i+1)->position
   *)
  (# i,j: @integer;
     deleteT: @...
  enter (i,j)
  do deleteT
  exit THIS(wText)[]
  #);
makeLC: (* Converts all characters to lower case *)
  (# ...
  exit THIS(wText)[]
  #);
makeUC:
  (* Converts all characters to upper case *)
  (# ...
  exit THIS(wText)[]
  #);
sub:
  (* Returns a copy of THIS(wText)[i:j].  If 'i<1', 'i' is
   * adjusted to 1. If 'j>length', 'j' is adjusted to
   * 'length'.  If (after adjustment) 'i>j', an empty wText is
   * returned.
   *)
  (# i,j: @integer; T1: ^wText;
     subI: @...
  enter (i,j)
  do subI
  exit T1[]
  #);
copy:
  (# T1: ^wText;
     copyI: @...
  do copyI
  exit T1[]
  #);
scanAll:
  (* Scans all the elements in THIS(wText).  For 'ch' in '[1:
   * THIS(wText).length]' do INNER
   *)
  (# ch: @wchar
```

```
  do (for i: lgth repeat T[i]−>ch; INNER scanAll for)
  exit THIS(wText)[]
  #);
find:
  (* find all occurrences of the character 'ch' in
   * THIS(wText), executing INNER for each occurrence found,
   * beginning at 'THIS(wText).position'.  'inx' will contain
   * the position of each 'ch' in THIS(wText).  If 'NCS' is
   * further bound to 'trueObject', the comparison will be
   * done Non Case Sensitive.  If 'from' is further bound, the
   * search will begin at position 'from'.
   *)
  (# ch: @wchar;
     inx: @integer;
     NCS:< booleanObject;
     from:< integerObject(# do pos−>value; INNER from #)
  enter ch
  ...
  exit THIS(wText)[]
  #);
findAll: find
  (* As 'find', except that the entire wText will be searched.
   * Replaces 'findCh' in previous versions of betaenv (v1.4
   * and earlier)
   *)
  (# from:: (# do 0−>value #)
  do INNER findAll
  #);
findwText:
  (* find all occurrences of the 'txt' in THIS(wText),
   * executing INNER for each occurrence found, beginning at
   * 'THIS(wText).position'.  'inx' will contain the position
   * of the first character of each occurrence found
   * THIS(wText).  If 'NCS' is further bound to 'trueObject',
   * the comparison will be done Non Case Sensitive.  If
   * 'from' is further bound, the search will begin at
   * position 'from'.
   *)
  (# txt: ^wText;
     inx: @integer;
     NCS:< booleanObject;
     from:< integerObject(# do pos−>value; INNER from #)
  enter txt[]
  ...
  exit THIS(wText)[]
  #);
findwTextAll: findwText
  (* As 'findwText', except that the entire wText will be
   * searched
   *)
  (# from:: (# do 0−>value #)
  do INNER findwTextAll
  #);
extend:
  (* Extend THIS(wText) with 'L' (undefined) chars. Notice
   * that it is only the representation of the THIS(wText),
   * that is extended, the 'length' and 'position' are not
   * changed.
   *)
  (# L: @integer
  enter L do L−>T.extend
  exit THIS(wText)[]
  #);
indexError:< wStreamException
  (* Raised from 'Check' when the index goes outside the
   * range of the wText. Message: "Index error in wText!".
```

```
        *)
      (# inx: @integer
      enter inx
      ...
      #);
    EOSerror::<
      (* Raised from 'get' and 'peek' when the end of the wStream is
       * passed.
       *)
      (# ... #);
    otherError::<
      (* Raised when an error other than the Index−/EOSerror
       * occurs.
       *)
      (# ... #);
    setPos::<
      (# ... #);
    getPos::<
      (# do pos−>value; INNER getPos #);
    (* Private attributes: !!OBS!! The 3 attributes 'T', 'lgth'
     * and 'pos' declared below MUST be the first data items
     * declared in 'wStream' and 'wText' since their addresses are
     * hardcoded into the compiler.
     *)
    T: [16] @wchar;
    lgth,pos: (* 16 is default size *) @integer;
    setT: (# enter T do T.range−>lgth−>pos #);
    setAscii:
      (# t: ^ text
      enter T[]
      do T.scanAll(#do ch −> put #)
      #);
    asAscii:
      (# T: @text
      do scanAll(#do ch −> T.put #)
      exit T[]
      #)
  enter setT
  exit T[1: lgth]
  #) (* Pattern wText *);

ascii2wText:
  (# T1: ^text; T2: @wText
  enter T1[]
  do T1.scanAll(#do ch −> T2.put #);
  exit T2[]
  #);
−−−textLib:attributes−−−
aswText:
  (# UT: @wText
  do scanAll(#do ch −> UT.put #)
  exit UT[]
  #)
```

Index

The entries in the alphabetic index consists of selected words and symbols from the body files of this manual − these are in **bold** font − as well as the identifiers defined in the public interfaces of the libraries − set in regular font.
In the manual, the entries, which can be found in the index are typeset like this. This can help localizing the identifier, when the link from the index if followed − especially in the case where the browser does not scroll the line to the top, e.g. because there is less than a page of text left.
In the small table of letters and symbols below, each entry links directly to the section of the index containing entries starting with the corresponding letter or symbol.

< _ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

<

_

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z