

# The Mjølner System

## Mjølner System Tutorial

Mjølner Informatics Report

MIA 94-24(1.4)

November 1997

Copyright © 1994-97 Mjølner Informatics ApS.  
All rights reserved.

No part of this document may be copied or distributed  
without the prior written permission of Mjølner Informatics

# Introduction

This manual is a tutorial on the Mjølnér System. The tutorial gives an introduction to the Mjølnér System environment to someone who already knows how to program in another (object-oriented) language and want to know how to do it in BETA.

Often a hidden strength of a language lies in the available libraries and the ease with which it can interact with the operating system or other software.

We present a sequence of tutorial programs designed to teach the essentials of BETA programming. The programs start with trivial objectives like printing out “Hello world” and doing arithmetic and gradually cover things like input/output, files, arrays, procedures, objects, containers, GUI, and persistence.

How to install the Mjølnér System, call the compiler, etc., will not be described in this tutorial. This information can be found the system manuals [MIA 90-2], [MIA 90-4], and [MIA 90-6].

Most of the tutorial programs developed in this tutorial, are supplied along with the MjølnérSystem in the `tutorial` demo directory.

Although it is not necessary, it will be an advantage for the reader to be familiar with the basic concepts of the BETA language. The BETA Language Introduction [MIA 94-26] gives an introduction to the BETA language. The BETA Book [Madsen 93] is the main reference for the BETA language and every programmer with intend to use BETA extensively should read this book.

## **Acknowledgment**

This tutorial is based on a set of tutorial notes written by Jean Vaucher, Professeur d'informatique, Université de Montréal, on sabbatical at CRIM, November 1993. Mjølnér Informatics are grateful to Jean Vaucher for taking the initiative in writing the first BETA tutorial notes. By permission of Jean Vaucher, we have adopted these notes for this tutorial and extended them with more sections containing examples on repetitions, texts, containers, persistence, and GUI programming.

# Contents

<b>Introduction.....</b>	<b>i</b>
<b>Contents.....</b>	<b>ii</b>
<b>List of Programs.....</b>	<b>iv</b>
<b>1 Hello World.....</b>	<b>1</b>
<b>2 Simple Types and Output.....</b>	<b>3</b>
<b>3 Arithmetic and Expressions.....</b>	<b>6</b>
3.1 Constants.....	6
3.2 Evaluations.....	6
<b>4 Multiplication Table.....</b>	<b>10</b>
<b>5 Loops and Functions.....</b>	<b>15</b>
<b>6 Assignment and Procedure Calls.....</b>	<b>17</b>
<b>7 Patterns and Variables.....</b>	<b>21</b>
7.1 Patterns and Variables.....	22
7.2 Attribute Access.....	22
<b>8 Dynamic Data Structures.....</b>	<b>24</b>
8.1 Primitive Types.....	27
<b>9 Repetitions (Arrays).....</b>	<b>28</b>
9.1 Multidimensional Repetitions.....	31
<b>10 Files.....</b>	<b>33</b>
10.1 Command-line arguments.....	34
10.2 Storing Characters in a Repetition.....	36
<b>11 Directory.....</b>	<b>38</b>
<b>12 Text Manipulation.....</b>	<b>40</b>
12.1 Advanced Formatted Input and Output.....	42
<b>13 Container Library.....</b>	<b>44</b>
13.1 List Example.....	44
13.2 HashTable Example.....	45
<b>14 Exceptions.....</b>	<b>48</b>
14.1 Examples Using Exception.....	48
<b>15 Access to External Functions and Data.....</b>	<b>50</b>
15.1 Example.....	50
<b>16 Using the Persistence Library.....</b>	<b>54</b>
<b>17 Graphical User Interface.....</b>	<b>56</b>
17.1 GUIEnv.....	56

17.2 MotifEnv.....	61
<b>18 Concurrent Library.....</b>	<b>64</b>
18.1 Example.....	64
18.2 Concurrency and User Interface Environments.....	66
<b>19 The Fragment System.....</b>	<b>72</b>
19.1 Interface and Implementation.....	74
<b>References.....</b>	<b>77</b>
<b>Index.....</b>	<b>79</b>

# List of Programs

Program 1: HelloWorld.bet.....	1
Program 2.1: SimpleTypesWithErrorsbet.....	3
Program 2.2: SimpleTypes.bet.....	5
Program 3: ExploreTypes.bet.....	8
Program 4.1: Multiplication1.bet.....	11
Program 4.2: Multiplication2.bet.....	13
Program 5: SquareRoot.bet.....	16
Program 6: MultipleAssigment.bet.....	19
Program 8: StaticAndDynamic.bet.....	27
Program 9.1: QuickSort.bet.....	30
Program 9.2: MultiplicationTable.bet.....	32
Program 10.1: CountChar.bet.....	34
Program 10.2: CountChar1.bet.....	35
Program 10.3: CountChar2.bet.....	37
Program 11: ListDir.bet.....	39
Program 12.1: FileCount.bet.....	41
Program 12.2: StaticAndDynamic2.bet.....	43
Program 13.1: SaveListDir.bet.....	45
Program 13.2: DirTable.bet.....	46
Program 15: Person.bet.....	53
Program 16.1: DirList.bet.....	54
Program 16.2: SaveListDir2.bet.....	54
Program 16.3: SaveListDir3.bet.....	55
Program 16.4: GetListDir.bet.....	55
Program 17.1: TextEditor.bet.....	59
Program 17.2: Hello.bet.....	62
Program 17.3: Scale.bet.....	62
Program 18.1: Seconds.bet.....	65
Program 18.2: Clock.bet.....	66
Program 18.3: ClockTextEditor.bet.....	69
Program 19.1: HelloWorld.bet.....	72
Program 19.2: HelloWorld with filled program slot.....	73
Program 19.3: putBoxed.bet.....	73
Program 19.4: HelloWorld.bet.....	73
Program 19.5: HelloWorld with filled program and LIB slot.....	74
Program 19.6: putBoxed.bet.....	75
Program 19.7: putBoxedBody.bet.....	75
Program 19.8: HelloWorld.bet.....	75
Program 19.9: HelloWorld with filled program and LIB slot.....	76

# 1 Hello World

We start with the simplest of all programs that prints the statement “Hello World” on the screen.

```
ORIGIN '~beta/basiclib/ v1.6/betaenv'
---- program: descriptor ----
(#)
(* HelloWorld.bet:
 * =====
 * Author: J.Vaucher
 *
 * Purpose:
 * This is the simplest program possible.
 * Being able to compile and run it shows that the
 * compiler exists and that PATHS and ALIASES have been
 * correctly set. It also brings out "meta-programming"
 * considerations like the "fragment" system.
 *)
do
  'Hello world !' -> putline
#)
```

## Program 1: HelloWorld.bet

This illustrates the basic syntax of a BETA program:

```
(#
  <declarations>
do
  <statements>
#)
```

In this case, there are no *declarations* and the *statement* part is merely a simple output statement.

**Declarations and Statements**

The bulk of the program text is in the form of a comment traditionally delimited by (\* and \*). In the BETA book [Madsen 93], comments are shown delimited by { and } but the compiler only recognizes the form shown in Program 1.

The first two lines of the program are not strictly BETA (but are essential for correct compilation). They are part of the *fragment* specification language that describes inter-relationships between the various BETA modules that compose a complete program. A brief introduction of the fragment system is given in the last section of this tutorial.

The first line formally identifies the *library* environment required by our program; in other words, it gives the file path name of the BETA module where all the basic functions and procedures (such as `putline`) have been defined. In this respect, it acts much like the `#include <stdio.h>` statement seen at the beginning of most C programs.

**betaenv**

The body of the program is a simple procedure call to do output. It is interesting to compare the syntax of BETA's procedure calls with that of other languages:

```
C++:      cout << "Hello world !";
C:        printf("Hello world !\n");
BETA:    'Hello world !' -> putline;
```

**Screen Output**

In BETA, the syntax of procedure calls is made identical to that of simple assignments (or message passing). Evaluation is strictly left to right: parameters are evaluated; then passed (->) to an object `putline` whose role is to output them to the screen. Finally, note that text constants are delimited by apostrophes ("").

The `betaenv` library and many of the other libraries used in this tutorial are documented in the MjølnerSystem manual [MIA 90-8]. **More information**

## 2 Simple Types and Output

In our next program we declare variables of the 4 basic types defined in BETA: integer, real, char and boolean. Values are assigned to these variables and they are written out. The first version of our program is shown in Program 2.1.

```
ORIGIN '~beta/basiclib/ v1.6/betaenv'
---- program: descriptor ----
(*****
*
* SimpleTypes1.bet: A program to show handling of simple types
*
* Note: this program will not compile
*
***** )
(# i,j,k: @integer;
  x,y,z: @real;
  c: @char;
  b1,b2: @boolean;
do l1l->i;
  10->j -> k;
  i + 3 * j->k ;

  3.1415->x;
  10e5->y;

  'X'->c;
  true->b1;

newline;
'Printing out integers'->putline;
' i = '->puttext; i->putint; newline;
' j = '->puttext; j->putint; newline;
' k = '->puttext; k->putint; newline;

newline;
'Printing out reals: '->putline;
' x = '->puttext; x->putreal; newline;
' y = '->puttext; y->putreal; newline;
' z = '->puttext; z->putreal; newline;

newline;
'Now for a character, C="'->puttext; C->Put;
'" and as a integer: '->puttext;
C->putint;  newline;

newline;
'Printing out booleans: '->putline;
' B1 = '->puttext; b1->putboolean; newline;
' B2 = '->puttext; b2->putboolean; newline; newline;

'Now for something very C-like.'->putline;
' 'A'+ 3->put: '->puttext;
'A' + 3->put;

newline;
#)
```

**Simple types**

**Program 2.1: SimpleTypesWithErrorsbet**



Again, one notes the initial *fragment* `ORIGIN` statement, a comment, and then the program.

The declarations are Pascal-like with the addition of the `@` character. In BETA, the same declaration syntax will be used for types, variables, classes and procedures. In this context, the `@` denotes a *static variable* declaration whereas a declaration *without* the `@` corresponds to a *type* declaration.

**Static Variables**

The first few statements after the `do` show arithmetic and assignment. Arithmetic expressions follow convention; the usual operators (`+`, `-`, `*`, `/` (or `div`) and `mod`) are provided. Assignment goes left to right following the arrow and multiple assignment is allowed.

The next few lines show the syntax of real, character and boolean constants. Note that characters use the same delimiters as text strings.

In the remainder of the program we do output. The procedures used are:

- `newline:` skips to a new line
- `puttext:` writes a text
- `putline:` same as `puttext` followed by `newline`
- `putreal:` outputs a real
- `put:` outputs a character
- `putboolean:` outputs a boolean

**Screen Output**

Characters are type compatible with integers and can be used interchangeably in expressions. An example of this is shown at the end of the program.

**Type Compatibility**

When we try to compile `SimpleTypesWithErrors.bet`, we get the following semantic error message:

```
putreal
****Name is not declared

putboolean
****Name is not declared
```

There is also a complete listing of the program text with the same error messages in `SimpleTypesWithErrors.lst` to help localize the errors. In this case the error is due to the fact that `putreal` is not in the standard environment. `putreal` is in a library that must be *included* in the program<sup>1</sup>. Likewise `putboolean` is in the `textUtils` library.

**Static Semantic Errors**

The Mjølner System comes with a library supporting a wide range of input and output for numbers (integers, based integers, reals, etc.) called `numberio`. In order to print reals on the screen, this library must be included. A library is included using a `fragment INCLUDE` statement:

```
ORIGIN '~beta/basiclib/ v1.6/betaenv';
INCLUDE '~beta/basiclib/ v1.6/numberio';
INCLUDE '~beta/basiclib/ v1.6/textUtils';
---- program: descriptor ----
(#
  i,j,k: @integer
```

**Simple types with numberio**

---

<sup>1</sup> `putreal` is not in the standard environment in order to minimize the size of the resulting binary executable for simple programs that only uses the basic environment. `numberio` contains operations like `put/getreal` and `put/getbased`.

```
.....  
< original program >  
...  
#)
```

### **Program 2.2: SimpleTypes.bet**

And the results are shown below.

```
Printing out integers  
i = 111  
j = 10  
k = 141
```

```
Printing out reals:  
x = 3.141500  
y = 1000000.000000  
z = 0.000000
```

```
Now for a character, C="X" and as a integer: 88
```

```
Printing out booleans:  
B1 = true  
B2 = false
```

```
Now for something very C-like.  
'A'+ 3->put: D
```

# 3 Arithmetic and Expressions

We will start with the simplest of all expressions:

## 3.1 Constants

BETA accepts constants in the usual formats:

### Integer

1, 1666, 0, -12

There is also a format (`<base>X<number>`) for integers in other bases. All the following constants represents **Based Integers**:

2x1011, 3x102, 16xB, 0xb, 11

Predefined constants exist for `MaxInt` and `MinInt` in `betaenv`.

### Reals

3.14159, 3E9, 3.14E-9

Predefined constants exist for `MaxReal` and `MinReal` in the `math` library.

### Booleans

true, false

### Text and Characters

'a', 'ABC'

- a character is a text of length 1.
- the text delimiter character can be represented by doubling: `''Hi''`, is the text `'Hi'`
- following the C convention, many useful control characters can be introduced by using the backslash: `'\t'` for tabulator and `'\n'` for newline.

**Control Characters**

## 3.2 Evaluations

In BETA, the term *evaluation* is used to refer to expressions, assignment statements and procedure calls.

For evaluations arithmetic, boolean and relational operators are provided. The operator precedence is similar to C with the `AND` and `OR` operators considered to be on a par with `*` and `+` respectively. This means that parentheses will often be needed to obtain the desired result. The priorities are shown below:

Relative Priority	Class	Operators	Operator priority
least	relational	=, <>, >, >=, <, <=	
	additive	+, -, or, xor	
	multiplication	*, /, div, mod, and	
highest	unary	not, -, +	

When a real value is assigned to an integer the fractional part is removed irrespective of the sign of the value. Various functions for manipulating reals, including rounding, **Math Library**

log, log10, sin, cos, tanh, sqrt, and power are available in the math library. This library also contains predefined constants such as PI, e, pihalf (PI/2), log2e (log(e) to base 2), log10e (log(e) to base 10), ln2 (natural log of 2), etc.

The max, min and abs functions are not generic; they do not work properly with reals since they will convert input values to integers. Instead the functions fmax, fmin and fabs in the math library could be used.

Finally, since characters are type compatible with integers and there is no type control at this level, some interesting evaluations may be done, such as:

**Type  
compatibility**

```
(i+1) * ('a' * i) / 4.33->putint
```

The program shown in Program 3 shows some of the things discussed here. First, it includes both the numberio and the math fragments required for the numeric work. It also shows our first procedure, tab, introduced to help simplify formatting the output.

```
ORIGIN '~beta/basiclib/ v1.6/betaenv';
INCLUDE '~beta/basiclib/ v1.6/numberio';
INCLUDE '~beta/basiclib/ v1.6/math';
---- program: descriptor ----
(#)

(* p3.bet: Exploring types and functions
   *****)

i,j,k: @integer;
x,y,z: @real;
tab: (# do '      '->puttext #);

do '\nEnvironment constants: \n\n'->puttext ;
  '\t MaxInt = '->puttext; maxint->putint;
  '\n\t MinInt = '->puttext; minint->putint;
  '\n\t MaxReal= '->puttext; maxreal->putreal (# do exp->style #);
  '\n\t MinReal= '->puttext; minreal->putreal (# do exp->style #);
  3->i; -10->j;
  '\n\n *** Simple functions *** \n'->puttext;
  '\n   I      J      max(I,J)  min(I,J)  abs(J) \n'->puttext;
  tab ;
  i->putint; tab;
  j->putint; tab;
  (i,j)->max->putint; tab; tab;
  (i,j)->min->putint; tab; tab;
  j->abs->putint;
  '\n\n *** Reals ***\n'->puttext;
  '\nX          Y          fmax(X,Y)          fmin(X,Y)          fabs(-
3.6) \n'->puttext;
  3.01->x; 3.78->y;
  x->putreal; tab;
  y->putreal; tab;
  (x,y)->fmax->putreal; tab; tab;
  (x,y)->fmin->putreal; tab; tab;
  -3.6->fabs->putreal;
  '\n\nPi = '->puttext; pi->putreal;
  '\ncos(Pi/4) = '->puttext;
  (Pi div 4)->cos->putreal;
  '\n\nMixing types: (i+1) * ('a' * i) / 4.33'->
putText; ' = '->putText;
  (i+1) * ('a' * i) / 4.33->putint;
  newline;
#)
```

**Program 3: ExploreTypes.bet**

In BETA, procedures, types and objects are treated in a unified manner as variations of a single concept, the *pattern*. The general syntax for such a pattern declaration is: **Pattern**

```
<names>: <descriptor>
```

In the simplest case, the object descriptor is what we have called a block which can contain declarations and imperatives. In the case of procedures which need parameters or functions which return results, there can also be *input* and *output* parameters but we leave that for later.

In our example, `tab` is defined by: **Procedure**

```
tab: (# do '    ' -> puttext #);
```

This has neither parameters nor a result. `tab` merely stands for the more lengthy statement which outputs a string of 4 blanks. Contrast this declarations with that of the variables for `i,j,...,z` which use `@` to indicate that space is to be reserved for the variables.

Returning to our program, we first print out the `maxint` and `minint` constants defined for both integers and reals. Note the use of *control* characters, e.g. `'\n'` in the titles to do some formatting.

Next, we show the use of the `min`, `max` and `abs` functions on integers. We also use the `tab` procedure to space out the printed results.

After that we apply the `fmin`, `fmax` and `fabs` to reals.

Then we print `PI`, one of the built-in constants defined in `math`, as well as the value of `cos(45 degrees)`. Note that angles must be expressed as radians.

To finish off, there is a mixed evaluation with integers, booleans, characters and real.

The results of executing `ExploreTypes` are shown below:

```
Environment constants:
```

```
MaxInt = 2147483647
MinInt = -2147483648
MaxReal= 1.797693e+308
MinReal= 2.225074e-308
```

```
*** Simple functions ***
```

```
  I      J   max(I,J)  min(I,J)  abs(J)
  3     -10      3      -10      10
```

```
*** Reals ***
```

```
 X           Y           fmax(X,Y)           fmin(X,Y)           fabs(-3.6)
3.010000     3.780000     3.780000           3.010000           3.600000
```

```
Pi = 3.141593
cos(Pi/4) = 0.707107
```

```
Mixing types: (i+1) * ('a' * i) / 4.33 = 268
```

For more sophisticated examples on use of reals and math functions, the reader should look at the demonstrations programs:

- `realtest.bet`
- `putreals.bet`

These are located in the reals demo directory of `basiclib`.

## 4 Multiplication Table

In this section, we will take on the simple task of printing out a multiplication table of the integers from 1 to 9. For the purpose of illustration we shall introduce the following concepts:

- procedure with parameters
- the `for` imperative
- the `if` imperative

for printing in fixed width columns. We shall also show the *correct* BETA way to do formatted output, involving introducing an extension to a virtual pattern.

### The `for` imperative

In BETA, all syntactic structures have a similar form of delimiters with opening and closing parentheses and the `for` is no exception:

```
Block:    (#    ...  #)
Comment: (*    ...  *)
For: (for ... for)
```

In opposition to other languages where the `for` can count up or down and the step can be varied, in BETA only a simple version exists going from 1 to  $N$  where  $N$  is an expression (or *evaluation* to use the BETA terminology). This follows the BETA minimalist philosophy of providing the strict minimum combined with powerful extension mechanisms.

More precisely, there are two forms of the `for`, depending on whether one wants to have access to the counting variable or not. These are: **Two kinds of `for`**

```
(for <evaluation> repeat <imperatives> for)
```

and

```
(for <var> : <evaluation> repeat <imperatives> for)
```

Both forms will be used in this section. Note that the loop variable `<var>` acts as a locally defined variable and is only accessible inside the `for` imperative. Thus the loop variable `<var>` need not be declared elsewhere.

Neglecting labels, the body of our program could have the following form:

```
(for i:9 repeat
  (for j:9 repeat
    i*j -> putint;
    2 -> tab;
  for);
  newline;
for);
```

Where `tab` is a procedure that outputs blanks to separate the integers printed with `putint`. In contrast with previous versions of `tab` which printed a fixed number of blanks, here we provide it with a parameter  $N$  to indicate the number of spaces to write. Above, `in2 -> tab`, we want to print out 2 spaces.

In BETA, parameters are considered to be local variables which are assigned values from an input list before executing the procedure body. Hence they are declared the same way as any other local variable. The number of actual parameters that must be supplied and which local variables will receive these values is specified by an enter list. `tab` can be defined as follows:

```
tab: (# N: @integer
      enter N
      do (for N repeat ' ' -> put for)
      #);
```

The enter list comes after the declarations and before the `do` part.

Here is the program:

```
ORIGIN '~beta/basiclib/ v1.6/betaenv'
---- program: descriptor ----
(#)
(* Multiplication Table
 *
 * Objectives:
 *   - use the FOR imperative
 *   - introduce a parameterized procedure
 *)

tab: (# N: @integer
      enter N
      do (for N repeat ' ' -> put for)
      #);

do '\n\t** Multiplication Table ** \n\n'-> puttext;
4 -> tab;
(for i: 9 repeat
  i -> putint; 2->tab;
for);
newline; newline;
(for i: 9 repeat
  i->Putint; 3 -> tab;
  (for j: 9 repeat
    i*j->putint ;
    2 -> tab;
  for);
  newline
for)
#)
```

#### Program 4.1: Multiplication1.bet

Since this program uses no reals, we have used the same simplified fragment statements that we used in our first program. The results are shown below:

```
** Multiplication Table **

1 2 3 4 5 6 7 8 9
1 1 2 3 4 5 6 7 8 9
2 2 4 6 8 10 12 14 16 18
3 3 6 9 12 15 18 21 24 27
4 4 8 12 16 20 24 28 32 36
5 5 10 15 20 25 30 35 40 45
6 6 12 18 24 30 36 42 48 54
7 7 14 21 28 35 42 49 56 63
8 8 16 24 32 40 48 56 64 72
9 9 18 27 36 45 54 63 72 81
```

In this output, the columns do not line up because some of the integers use require a single digit and others require 2 and `putint` prints with minimum space.

To overcome this, we will write a simple procedure to output an integer in a field of width `w`. By simple, we mean that it will only work with positive integers less than 1000. To determine the number of digits required to print an integer we use the `if` imperative. The `if` imperative has two forms:

```
(if <Exp> then
  <Imperatives>
else
  <Imperatives>
if);
```

**Simple if**

and

```
(if <Exp>
  // <Exp> then <Imperatives>
  // <Exp> then <Imperatives>
  // <Exp> then <Imperatives>
  ...
else
  <Imperatives>
if);
```

**General if**

A first crack at the code to compute the number of digits in an integer is:

```
(if true
  // i < 10 then 1 -> ND
  // i < 100 then 2 -> ND
  else 3 -> ND
if)
```

This is coded along the lines of the standard `if...else if...else if...` pattern of other languages such as Pascal or Simula but is not quite correct according to the strict definition of the `if` imperative, because the `if` does not specify that the alternatives will be evaluated in a sequential manner. Therefore if `I= 5`, `ND` could receive either 1 or 2 as a value. Although the code would probably work, it should be written as:

```
(if true
  //          i<10 then 1 -> ND
  // (i>=10) and (i<100) then 2 -> ND
  else 3 -> ND
if)
```

Remember that the parentheses are required in the second alternative due to the operator priorities.

Now we can design a procedure, `(N,W) -> Outint`, that will print an integer `N` right justified in a field of `w` spaces wide. This procedure will require a *list* of 2 parameters when it is called.

**Procedure arguments**

```
Outint:
  (# N,W: @integer;
  enter (N,W)
  do (if true
    //          N<10 then 1 -> ND
    // (N>=10) and (N<100) then 2 -> ND
    else 3 -> ND
    if);
  ND -> tab;
  N -> putint;
  #);
```

The complete modified program is given below and the output follows.



```

ORIGIN '~beta/basiclib/ v1.6/betaenv'
---- program: descriptor ----
(
  (* Multiplication Table 2
  *
  * Objectives:
  *   - use the IF and FOR imperatives
  *   - use procedures & parameters
  *)

tab: (# N: @integer;
      enter N
      do (for N repeat ' ' -> Put for)
      #);

Outint:
  (# N,W, ND : @integer;
  enter (N,W)
  do (if true
      //           N<10   then 1 -> ND
      // (N>=10) and (N<100) then 2 -> ND
      //                                     else 3 -> ND

      if);
      W-ND -> tab;
      N -> putint;
  #);

do
  '\n\t** Multiplication Table ** \n\n'-> puttext;

  4 -> tab;
  (for i: 10 repeat
    (i,4) -> Outint;
  for);
  newline; newline;

  (for i: 10 repeat
    (i,4) -> Outint;
    (for j: 10 repeat
      (i*j,4) -> Outint;
    for);
    newline
  for)
#)

```

#### Program 4.2: Multiplication2.bet

and the results:

```

** Multiplication Table **

  1  2  3  4  5  6  7  8  9 10
1  1  2  3  4  5  6  7  8  9 10
2  2  4  6  8 10 12 14 16 18 20
3  3  6  9 12 15 18 21 24 27 30
4  4  8 12 16 20 24 28 32 36 40
5  5 10 15 20 25 30 35 40 45 50
6  6 12 18 24 30 36 42 48 54 60
7  7 14 21 28 35 42 49 56 63 70
8  8 16 24 32 40 48 56 64 72 80
9  9 18 27 36 45 54 63 72 81 90
10 10 20 30 40 50 60 70 80 90 100

```

Finally, the way formatted output is done in the demo programs distributed with the Mjølnir System. It involves extending the virtual pattern `format` defined in `putint`. Thus the `OutInt` procedure of Program 4.1 should be rewritten as follows: **Formatted output**

```
Outint:
  (# N,W : @integer;
  enter (N,W)
  do N -> screen.putint(# format::(# do W -> width #)#)
  #);
```

and the results would be identical to those obtained previously. The latter procedure would be much more robust, accepting any positive or negative integer value. Program `Multiplication3.bet` uses this procedure. See also Section 12.1.

## 5 Loops and Functions

In this section, we take on the task of writing a function that computes square roots. This will lead us to consider conditional looping and the definitions of functions, or in BETA terms, patterns that can return values.

To compute a square root we will use Newton's algorithm where the computation is done by successive approximations until the error is less than a preset level (epsilon).

The usual way to program this involves a *while* loop. In C, the code to get the root of *x* could look like this:

```
Root= X;
while (abs(X-Root**2) > epsilon) {
    Root= (Root+X/root) / 2;
}
```

Assuming that *x* = 100, the calculated values of *Root* in each iteration will be:

```
150.5
226.2401
315.0255
410.8404
510.0326
610.0001
710.0000... and the loop ends.
```

Surprisingly, there is no *while* loop in BETA language. BETA is designed with a more fundamental (and extensible) feature as the ability to either leave or restart any *labeled* imperative. In the basic BETA environment *betaenv*, a loop pattern using these features is defined and can be used like:

```
X -> Root;
Loop(# while::(# do (X-Root*Root -> fabs) > X / 10E6 -> value #);
    do (Root + X/root) / 2 -> root;
    #);
```

**While loop  
in betaenv**

A *while* loop can also directly be implemented using a labeled imperative with a *restart*. The following BETA code is a repeated conditional imperative implemented using a *labeled if* with a *restart* imperative at the end:

```
X -> Root;
L: (if (X-Root*Root -> fabs) > X / 10E6 then
    0.5 * (Root + X div Root) -> Root;
    restart L;
if);
```

**While loop using  
labeled  
imperative and  
restart**

Here, the error, *epsilon*, is set at one part in a million. The complete program code is given in Program 5 below. Here the code for the Square Root has been put in the form of a procedure which returns a value: that is to say a *function*. As shown in the example, a function call has the following syntax:

```
i -> Sqrt -> Res;
```

We have already seen the notation *<parameters> -> <procedure>* whereby data is passed (as parameters) to the procedure object. In BETA, the same syntax is used to show that results are obtained from *Sqrt* and passed on to *Res*.

```

ORIGIN '~beta/basiclib/ v1.6/betaenv';
INCLUDE '~beta/basiclib/ v1.6/numberio'
        '~beta/basiclib/ v1.6/math';
---- program: descriptor ----
(#)

(* SquareRoot.bet: Exploring functions
 *                   + conditional loops
 *)

Res: @real;

Sqrt:
  (# X, Root: @real;
  enter X
  do X -> Root;
  Loop(# while::
        (# do (X-Root*Root -> fabs) > X / 10E6 -> value #)
        do (Root + X/root) / 2 -> root;
        #);
  exit Root
  #);

tab:
  (# N: @integer;
  enter N
  do (for N repeat ' '-> put for)
  #);

do '\n\t ** Functions and variables **\n' -> putline;
' I  sqrt(i) ' -> putline;
(for i:10 repeat
  i -> Sqrt -> Res;
  i -> putint(# format::(# do 2 -> width#)#);
  3-> tab;
  Res -> putreal;
  newline;
for)
#)

```

### Program 5: SquareRoot.bet

The declaration of `Sqrt` shows an `exit` list which defines the list of values obtained after executing the body of the function. In this case, there is only one value, a real, but BETA *functions* could return multiple results. **Function results**

The output obtained upon execution are shown below.

```

        ** Functions and variables **

I  sqrt(i)
1  1.000000
2  1.414214
3  1.732051
4  2.000000
5  2.236068
6  2.449490
7  2.645751
8  2.828427
9  3.000000
10 3.162278

```

## 6 Assignment and Procedure Calls

We have previously mentioned that BETA strives for minimalism along with orthogonality. So far we have hidden this fact by programming in a standard fashion and presenting the programs with traditional concepts such as variables, functions, assignment statements, etc. Now we shall begin the study of BETA's particularities by considering assignment.

We have already noticed that BETA evaluates expressions left to right and uses an unusual assignment operator, `->`. Our examples have shown simple examples of assignment such as:

```
i+1 -> i;
```

In BETA, we can do more: assignment is defined to operate on lists of values with single value assignment being a special case. Thus we can say:

```
(1,2,3) -> (i,j,k);
```

Which has the same effect as the series of simple assignments:

```
1 -> i; 2-> j; 3 -> k;
```

We can also cascade such assignments:

```
(a,b,c) -> (m,n,o) -> (x,y,z) ;
```

If we just consider the first items in the lists, the above statement means that we take the value of `a` and pass it on to `m`, then we take the value of `m` and we pass it on to `x`.

If the number or types of the items in lists do not match, an error is signaled:

```
(111, 222) -> i;
```

or

```
'string' -> i; (* where i is an integer *)
```

So far, this is all pretty obvious but, in BETA, the destination of an assignment is not restricted to being a simple variable or a list of such variables; the target of an assignment can also be a more complex object with an `enter` list. In that case, assignment takes place between the values in the list on the left and the variables named in the target's `enter` list.

If both the source `S` and target `T` of an assignment are complex objects, the assignment:

```
S -> T;
```

becomes a multiple assignment between the `exit` list of `S`, (`O1`, ..., `Oi`, ...) and the `enter` list of `T` (`I1`, ..., `Ii`, ...). In addition, the `do` part of `S` is executed before this multiple assignment and the `do` part of `T` is executed after. In other words, this happens:

1. Execute `do` of `S`
2. (`O1`, ..., `Oi`, ...) -> (`I1`, ..., `Ii`, ...)
3. Execute `do` of `T`

For cascaded assignments:

```
S -> T -> U;
```

Evaluations

Complex evaluations

We have:

1. Execute `s`
2. Output of `s` -> input of `T`
3. Execute `T`
4. Output of `T` -> input of `U`
5. Execute `U`

Note that the body of each object mentioned in the assignment is executed *once*. Of course, defining assignment between complex structures or objects in terms of assignment between individual `exit` expressions and `enter` variables is a *recursive* explanation which eventually leads to assignment between primitive objects like integers which has an obvious interpretation in terms of machine code. This is described in great detail in section 5.8 of the BETA Book.

Note that absence of a `do` part has no effect on assignment: it is equivalent to a *null* statement; *but* absence of an `enter` or `and exit` part has great importance. An object without an `exit` part cannot appear as a source in an assignment. Similarly, an object without an `enter` cannot appear as a target.

This property is exploited in the `math` fragment to define *read-only* objects or *constants*. These have only an `exit` list:

**Constants**

```
e: (# exit 2.7182818284590452354 #);
Pi: (# exit 3.14159265358979323846 #);
```

BETA's generalized definition of assignment means that there is no fundamental difference between assignments and procedure calls. Following from this argument is the fact that *procedure* declarations and *type* declarations will be syntactically identical. To illustrate this, consider the following declarations. The first is a *type* definition for a complex numbers with two `real` attributes. The second is a procedure that adds real numbers.

```
complex: (# Re,Im: @real;
          enter (Re,Im)
          exit (Re,Im)
          #);

add: (# A,B: @real;
      enter (A,B)
      exit A+B
      #);
```

**Procedure and  
type declarations**

These definitions have been purposely made alike. There is no `do` part in either; the computation for `add` being done by an expression in the `exit` list. Each has 2 local `real` attributes. Each has an `enter` list and an `exit` list, meaning that objects of the type `complex` and those of the type `add` can be assigned values and can provide values and thus can be used on both sides of an assignment statement. For example:

```
(1.0, 3.3) -> complex -> (x,y);
(1.0, 3.3) -> add -> x;
```

In the case of `complex`, the output value is an exact duplicate of its local state and of the input values: `complex` objects will be used mainly for their storage potential (as variables). With `add`, the output value is computed from the state (input) values: `add` objects are more useful for this computational aspect. In actual fact, `complex` would seldom be used directly in such an assignment; it would more properly be used as a model for variables which would in turn be used in assignments.

**Local state**

```
(# c1,c2: @complex;
do (1.0, 3.3) -> c1;
   c1 -> c2;
#)
```

The point, however, is that BETA does not distinguish between *types* and *functions*. It considers both as examples of a more fundamental concept: the object, which can be used for many things depending on how the programmer chooses to define and use it.

The program below shows the examples that we have talked about:

```
ORIGIN '~beta/basiclib/ v1.6/betaenv'
---- program: descriptor ----
(#
  (*   Multiple assignment and function calls   *)

  i,j,k : @integer;
  Nines: (# exit 99999 #);    (* Constant   *)

  complex: (# Re,Im: @integer;
            enter (Re,Im)
            exit  (Re,Im)
            #);
  add: (#  A,B: @integer;
        enter (A,B)
        exit A+B
        #);

do
  'Examples using multiple assignment and function calls' ->
    putline; newline;

  'Outputting a constant: Nines= ' -> puttext;
  Nines -> putint;
  newline; newline;

  '(1,2,3) -> (i,j,k); '-> putline; newline;

  (1,2,3) -> (i,j,k);
  ' I= '-> puttext;  i -> putint;
  ', J= '-> puttext;  j -> putint;
  ', K= '-> puttext;  k -> putint; newline;

  newline; '(i,j) -> (j,i):  ' -> puttext;

  (i,j) -> (j,i);
  ' I= '-> puttext;  i -> putint;
  ', J= '-> puttext;  j -> putint; newline;

  'Note that (x,y) -> (y,x) doesn't imply swap semantics.' ->
    putline; newline;

  ' (** More examples **) ' -> putline; newline;

  (111,999) -> complex -> (i,j);
  (111,999) -> add -> k ;

  ' I= '-> puttext;  i -> putint;
  ', J= '-> puttext;  j -> putint;
  ', K= '-> puttext;  k -> putint; newline;
#)
```

### Program 6: MultipleAssignment.beta

Now for its output:

Examples using multiple assignment and function calls

Outputting a constant: Nines= 99999

(1,2,3) -> (i,j,k);

I= 1, J= 2, K= 3

(i,j) -> (j,i): I= 1, J= 1

Note that (x,y) -> (y,x) doesn't imply swap semantics.

(\*\*\* More examples \*\*\*)

I= 111, J= 999, K= 1110



# 7 Patterns and Variables

In this section, we get a bit more formal with the BETA way of expressing object-oriented concepts. This section treats topics from chapter 3 of the BETA Book but with a slightly different approach.

Often object-oriented concepts are summarized as follows:

**Object-oriented concepts**

- 1) *Objects* are meant to represent the things that we see or talk about in the real world.
- 2) *Objects* have properties. These are often divided between state (attributes) and potential actions (services, methods or scripts).
- 3) *Objects* which have the same attributes and behave the same way are said to belong to the same *class*. A class definition specifies the attributes and actions common to all objects of the same class. The objects described by a class are said to be instances of that class.
- 4) Often it is useful to introduce the notion of class hierarchy to reflect various levels of similarity. The specification mechanism which allows incremental description of the similarities at each level is called *inheritance*.
- 5) Additionally, some languages model the fact that individual objects can operate in parallel. These *active* objects are sometimes called agents, processes or actors.

**Inheritance**

**Concurrency**

Actually, in this section, we will only talk about points 2 and 3. Point 1 is included for completeness and point 4 is not covered in this tutorial (see the BETA language introduction [MIA 94-26] or the BETA book [Madsen 93]). Point 5 will be covered later.

In BETA, objects are described by a syntactic construct called an *object-descriptor* (or *descriptor* for short). This has the form:

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

and is used to specify the local attributes and actions of an object (or class of objects).

An *unusual* feature of the BETA object-descriptor (compared to other object-oriented languages) is that, with `enter` and `exit` lists, it introduces the notion of *value* for objects. The different facets of an object are defined by the various (optional) parts of the object description:

```
(# <declarations>  
  enter <input list>  
  do    <imperatives>  
  exit  <output list>  
#)
```

**Descriptor**

This single concept of *descriptor* has been used to replace many diverse concepts that we are familiar with from traditional languages. In particular, it is used for procedures, functions, types, classes, macros and local blocks. These different roles are achieved by using the descriptor in different contexts, by combining it with other *operators* or by selective use of the internal parts.

## 7.1 Patterns and Variables

In BETA, a descriptor can be given a name. Thereafter, the name becomes a shorthand form for the full description. The association of name and descriptor is written amongst the declarations and is known as a *pattern declaration*. It has the following syntax:

```
<pattern-name>: <prefix> <descriptor>
```

**Pattern  
declaration**

Thereafter, the pattern name or the descriptor can be used interchangeably and the term *pattern* is used to mean either form. `prefix` is an optional name of a pattern that `pattern-name` inherits from.

As we have seen previously, variable declarations are characterized by the token:

```
<variable-names>: @<pattern>
```

**Static variable  
declarations**

Examples:

```
POINT: (# X,Y: @real;  
  enter (X,Y)  
  exit (X,Y)  
  #);  
  
P1,P2: @Point;  
P3: @(# X,Y: @real;  
  enter (X,Y)  exit (X,Y)  
  #);
```

P1 and P2 are considered to be objects of the same class. Note also that *case* is immaterial in BETA:POINT being the same as PPOINT or point.

Thereafter, the following assignments are allowed:

```
P1 -> p2;  
(0,0) -> p1 -> p2 -> p3;
```

Note that BETA uses structural equivalence in its value assignments and comparisons. This is not as strict as in other languages and assignment is possible between variables of any two patterns with similar `enter` and `exit` lists. This was shown above (with identical lists) and if we define another pattern `complex` with two reals as value, assignment will be allowed between `point` and `complex` objects.

**Structural  
equivalence**

```
C1: @(# Re,Im: @real;  
  enter (Re, Im)  
  exit (Re, Im)  
  #);
```

## 7.2 Attribute Access

So far we have concentrated on showing that BETA objects can behave either as classical data variables or as procedures. The example patterns that we showed often had local attributes (of primitive types) used either to implement *value* or to hold temporary results of computations.

BETA objects also function as *structured* (or *record*) data and the local attributes are accessible via the traditional *dot* notation. For the POINT P1 defined in the previous section, the examples below show how its local attributes X and Y can be both *read* and *set* directly:

**Structured data**

```
0 -> P1.X; 123 -> P1.Y;  
P1.X -> putint;
```

Notice, that the first line is equivalent to `(0,123) -> P1`.

The local attributes can have any *type* and could be *function* objects. Below, we use a modified `Point` pattern which has a third attribute `dist` to compute the distance from the origin.

```
Point: (# X,Y: @real;  
        dist: @(# exit (X*X + Y*Y) -> sqrt #);  
        enter (X,Y)  
        exit  (X,Y)  
        #);
```

**Local function**

```
P1,P2 : @Point;
```

This third attribute is *read-only* (because it has no `enter` list) but can be consulted just like `x` and `y`:

```
(3,4) -> P1;  
P1.dist -> putint>(* will print "5" *)
```

Many object-oriented languages enforce encapsulation by disallowing direct reading or writing of the local *variables* and restrict access to the invocation of the local *methods*. Smalltalk is the obvious example of this approach but even Simula, where by default all local data and methods are accessible, introduced a Hidden/Protected mechanism to allow protection. This is meant to enforce separation of the provided behavior from the details of implementation.

**Encapsulation**

In the BETA *language*, there is no provision for hiding the internal details of an object. The mechanisms for protection as well as those for modularization and configuration management are relegated to a separate *fragment system* which is described in the last section of this tutorial.

## 8 Dynamic Data Structures

The variables that we have declared so far (with @) are said to be *static objects* and the variable names are said to be *static references*

Advanced programming requires more than just static data structures. In particular, list processing is based on the notion of dynamically created objects linked by pointers. Recursive procedures also rely on dynamically allocated *activation records*. More to the point, objects are generally created on demand with the *new* operator.

In BETA, pointers or *dynamic references* as they are called are declared very much as in C using the \* token. Below, we declare *refA* and *refB* to be *references* to *Point* objects whereas the declaration for *P1* and *P2* cause storage space to be reserved for 2 points and associate (permanently) the names *P1* and *P2* to those points.

**Dynamic references**

```
refA, refB: ^Point;  
P1, P2: @Point;
```

Initially, *refA* and *refB* point nowhere and have the value *NONE* whereas *P1* and *P2* designate real live points. Thus, we can assign data to *P1* but not to *refA*.

```
P1 -> P2;      (* OK *)  
P1 -> refA;    (* run-time error because refA is NONE *)
```

This seems normal *but reread* these two imperatives carefully. Anyone having used pointers in other languages should notice that the BETA pointer concept is a little different than most other languages. We have used (correctly) the same notation for the *variable* and for the *pointer*. If this were C, with:

```
Point p1, p2;  
Point *refA, *refB;
```

then *refA* would represent the *address* of a *Point* and *\*refA* would be used to denote the *contents* of that *Point*. In C, the assignments would have read

```
p2 = p1;  
*refA = p1;
```

Now we can return to the BETA approach to dynamic data which is quite different from the traditional one.

In BETA, a pointer is treated as a *reference* which may point to different objects (or to *NONE*) at different times during execution whereas a variable is considered to be a *reference* which will always denote the same object. Thus both are *references* but one is *static* and the other *dynamic* and they will be used in the same way to access the data. The concept of pointer storage address is avoided.

**Static and dynamic references**

In BETA, simple use of a reference (static or dynamic) in an evaluation refers to the contents of the object referenced. Thus, assuming that *refA* and *refB* designate *Points* (and not *NONE*) then,

```
(0.0, 0.0) -> P1 -> refA -> refB -> P2 ;
```

means that the contents (or value) of each point is set to (0,0).

To manipulate references to objects and not just the contents, we need to use a *reference operator*. In BETA, this is a postfix operator written [] (read *box*). Thus the following imperative:

**Reference operator**

```
refA[] -> refB[];
```

has the effect that `refB` now points to (references) the same object as `refA`.

BETA's approach is the converse of C's: BETA uses a referencing operator and C uses a dereferencing operator.

Assignment Type	BETA	C
Content	<code>refA -&gt; refB;</code>	<code>*refB = *refA;</code>
Reference	<code>refA[] -&gt; refB[];</code>	<code>refB = refA;</code>

In BETA, it is also possible to make a dynamic reference denote a *static object*. This can also be done in C:

**Dynamic reference to static object**

```
BETA:    P1[] -> refA[]
C:       refA = &P1;
```

This is one way to give dynamic references values other than `NONE`. The other and more obvious one involves dynamic creation of new objects at run-time. In BETA, the *new* operator is written `&`. Thus,

**New operator**

```
&Point
```

causes a new point object to be created. Now, comes a delicate aspect.

To create a new object and to get the address of this new `Point`, we also need the reference operator:

```
&Point[] -> refA[];
```

As mentioned in section 3.2.3 of the BETA book, this is a subtle point:

“The difference between `&P` and `&P[]` is very important: the expression `&P` means ‘generate a new instance of `P` and *execute it*’; the expression `&P[]` means ‘generate a new instance of `P` *without* executing it and return a reference to this new object’.”

In C a `Point` is allocated like this:

```
refA = (Point*) malloc(sizeof(Point));
```

The following program shows the use of static and dynamic references. This uses a `Point` user-type with integer attributes. There are 2 static references, `P1` and `P2`, and a dynamic reference, `refA`. At various points in the program `refA` points to either `P1` or `P2` or to a dynamically allocated object. Note that access to `Points` via the static or dynamic variables is syntactically identical. We assign various values to the three references and use `dump` to show the contents of the first attributes of all three. This shows that effectively `refA` designates various `Points` during execution. At the end, we show the use of a dynamically generated `Point` in a cascaded assignment. In this case, the purpose is just to show that it can be done and what happens. Useful version of this dynamic generation will be shown later.

```
(111,333) -> &Point -> P1;
```

**Create and execute**

What happens here is that

- 1) a new `Point` object is created,
- 2) the values `(111,333)` are assigned to the variables in the `enter` list of the new Object,
- 3) the (empty)`do` part of the object is executed,

- 4) a value assignment is done between the `exit` list of the `Point` and the `enter` list of `P1` and
- 5) finally, the (empty) `do` part of `P1` is executed. The newly created `Point` is inaccessible and the space it occupies will be reclaimed by the garbage collector.

Instead using the reference operator gives:

```
(111,333) -> &Point[] -> refA[];
```

**Create and return  
a reference**

What happens here is that

- 1) a new `Point` object is created,
- 2) the values `(111,333)` are assigned to the variables in the `enter` list of the new object,
- 3) a reference assignment is done between newly created object and `refA`. `refA` now refers to the new object.

```
ORIGIN '~beta/basiclib/ v1.6/betaenv'
---- program: descriptor ----
(#
  (* Static and Dynamic references *)

  Point: (# X,Y: @integer;
          enter (X,Y)
          exit (X,Y)
          #);

  refA : ^Point;
  P1,P2: @Point;

Dump:
  (#
  do 'P1: ' -> puttext;
    P1.X -> screen.putint(# format::(# do 3->width#)#);
    ', P2: ' -> puttext; P2.X -> putint;
    ', refA: ' -> puttext; refA.X -> putint;
    newline;
  #);

do
  'Dynamic references' -> putline;
  newline;

  &Point[] -> refA[];

  (1,1) -> P1-> P2 -> refA;
  Dump;

  (2,2) -> P2;
  (3,3) -> refA;
  Dump;

  P1[] -> refA[]; Dump;
  P2[] -> refA[]; Dump;

  (1,1) -> P1; (2,2) -> P2; (3,3) -> refA;
  Dump;

  (111,333) -> &Point -> P1;
  Dump;
  Newline;
#)
```

**Program 8: StaticAndDynamic.bet**

The results from execution are shown below:

Dynamic references

```
P1: 1, P2: 1, refA: 1
P1: 1, P2: 2, refA: 3
P1: 1, P2: 2, refA: 1
P1: 1, P2: 2, refA: 2
P1: 1, P2: 3, refA: 3
P1: 111, P2: 3, refA: 3
```

## 8.1 Primitive Types

In the previous section, all examples of static and dynamic references dealt with a user defined type, `Point`. Could we have done the same thing with one of the four primitive types in BETA: `integer`, `char`, `boolean` or `real`? The answer is *no*. For these 4 types, it is not legal to apply neither the *new* nor the *reference* operators. Similarly, we shall see later that other limitations apply and these types cannot be used as *prefixes* for other object definitions.

Thus the following expressions are illegal:

```
RefInt: ^integer; (* ILLEGAL *)
&real -> ... (* ILLEGAL *)
```

The reason is that for safe pointer operation, each object that can be designated by a dynamic reference (pointer) needs extra space for administrative data. In the case of primitive types, this overhead can be overwhelming and it has been chosen to handle them differently (and more efficiently) than other patterns. This is the same approach used in Simula, Pascal and Eiffel.

In Smalltalk, another approach was used: the language tries to treat all objects (and types) in exactly the same way. This makes for a very powerful system but, in spite of considerable research, Smalltalk programs are still bulky and notoriously slow.

With the `&` operator, C allows pointers to anything and this is a major source of errors in C code that neither the compiler nor the run-time system can help to control.

For situations where one would like to use primitive objects in ways identical to user-defined objects, BETA has defined 4 special patterns: `IntegerObject`, `CharObject`, `RealObject` and `BooleanObject`. These are completely compatible with their primitive counterparts (assignment, comparison, etc.) but dynamic creation (`&`), the reference operator (`[]`), inheritance, etc., are allowed on them.

**Objects of  
primitive types**

# 9 Repetitions (Arrays)

In BETA arrays are called repetitions

```
A: [10] @integer;
```

This repetition describes a set of static references to integers. 10 is called the range of the repetition (the upper bound). In spite that the lower bound is always 1, repetitions are flexible since the upper range is accessible as a local attribute of the repetition, they can be assigned, extended and sub-range access is possible (slices).

**Flexible and extensible repetitions**

BETA repetitions compared to its C counterpart:

Language	BETA	C
Declaration	A: [10] @integer;	int [10] A;
Lower	1	0
Upper	A.range	9
Size	A.range	10
Access	A[i]	A[i]
Assignment	A -> B;	not possible
Extend	10 -> A.extend	not possible
Slices	A[2..3]	not possible

It should be noted that it is not possible to take the address of a repetition, i.e. A[] is illegal (legal in C as &A).

In the current Mjølner implementation, it is possible to declare repetitions of types: char, boolean, integer, real, and any object reference:

```
(# Record: (# ... #);
  A: [100] ^Record;
do &Record[] -> A[1][]; (* create a new instance of Record and
                        * assign it to first entry in A *)
  ...
#)
```

**Object reference repetitions**

Besides assigning values to the elements of a repetition, whole repetitions can be assigned to other repetitions regardless of their ranges, e.g.:

```
a: [10] @integer;
b: [1] @integer;
do (for i: A.range repeat (* initialize a *)
    i -> a[i]; (* put i into i'th position in repetition a *)
  for);
  14 -> b[1]; (* a is [1,2,3,4,5,6,7,8,9,10], and
              * b is [14]
              *)
  a -> b; (* make repetition assignment:
          * a is [1,2,3,4,5,6,7,8,9,10], and
          * b is [1,2,3,4,5,6,7,8,9,10]
          *)
```

The next program illustrates how to use repetitions in a simple sorting program called quick sort, originating from C.A.R Hoare. Given a repetition, one element is chosen

**Quick sort example**



and the others partitioned into two subsets: those less than and those greater than or equal to the partition element. The same process is then applied recursively to the two subsets. When a subset has fewer than two elements it does not need any sorting and the recursion stops.

In BETA it is illegal to use the reference operator on repetitions, and since the quick sort algorithm is inherently recursive with the repetition as function argument in each recursion, we face a problem. However, this problem is easily solved in BETA. We simply define a pattern containing a repetition, and using an object of this type as the argument to quick sort.

```

numberRepetition: (# r: [1] @Integer #);
qsort:
  (# nr: ^numberRepetition;
   enter (nr[], ...)
   do ....
  #);
numbers: @numberRepetition;
do
  ...
  qsort(numbers[],...);

```

So the limitation of not being allowed to take a reference to repetitions is easily circumvented.

The quick sort algorithm also uses a `swap` operation, that swaps two elements in the repetition. This operation can be define locally inside (statically nested inside) `qsort`, so `swap` can operate on the same repetition:

**Local swap  
function**

```

qsort:
  (# nr: ^numberRepetition;
   swap:
     (# i,j: @Integer;
      temp: @Integer;
      enter (i,j)
      do nr.r[i] -> temp;
         nr.r[j] -> nr.r[i];
         temp -> nr.r[j];
      #);
   enter (nr[], ...)
   do ...
  #);

```

The complete code including a loop for reading numbers to be sorted from the keyboard follows below:

```

ORIGIN '~beta/basiclib/ v1.6/betaenv';
---program: descriptor---
(# (* Hoare QuickSort program illustrating how to use
 * repetitions, simple pattern declarations,
 * block structure and recursion.
 *)
numberRepetition: (# r: [1] @Integer #);
qsort:
  (# nr: ^numberRepetition;
   left, right, last: @Integer;
   swap:
     (# i,j: @Integer;
      temp: @Integer;
      enter (i,j)
      do nr.r[i]->temp;
         nr.r[j]->nr.r[i];
         temp->nr.r[j];
      #);
   enter (nr[], left, right)

```

```

do L: (if left >= right then (* stop if rep. contains *)
      leave L;                (* fewer than two elements *)
      else
        (* move partition element to nr.r[1] *)
        (left, (left+right)/2) -> swap;
        left->last;
        (* partition *)
        (for i: right-left repeat
          (if nr.r[i+left] < nr.r[left] then
            last+1->last;
            (last,i+left) -> swap;
            if);
          for);
        (left,last) -> swap; (* restore partition elem. *)
        (nr[],left,last) -> qsort;
        (nr[],last+1,right) -> qsort;
        if);
      #);
numbers: @numberRepetition;
t: ^Text;
i: @Integer;
do
  (* initialize a repetition with numbers typed
   * by the user
   *)
  'Type some numbers: '->puttext;
  getline->t[]; (* read all what the user types until newline *)
  l->i;
  t.reset;
  L: (if not t.eos then
      (* parse the text;
       * assuming that the user only types numbers
       *)
      (if i>numbers.r.range then
        (* remember to extend the repetition *)
        numbers.r.range->numbers.r.extend;
        if);
      t.getint->numbers.r[i];
      i+1->i;
      restart L;
      if);

  (* sort the repetition *)
  (numbers[],1,i-1) -> qsort;

  'Sorted numbers: '->puttext;
  (for j: i-1 repeat
    numbers.r[j]->putint; ' '->put;
  for);
  newline;
#)

```

### Program 9.1: QuickSort.bet

Running the program and typing some numbers results in the following output:

```

nil% QuickSort
Type some numbers: 9 8 4 6 3 8 2 7 12 45 2 78 5 6 1 0 2
Sorted numbers: 0 1 2 2 2 3 4 5 6 6 7 8 8 9 12 45 78

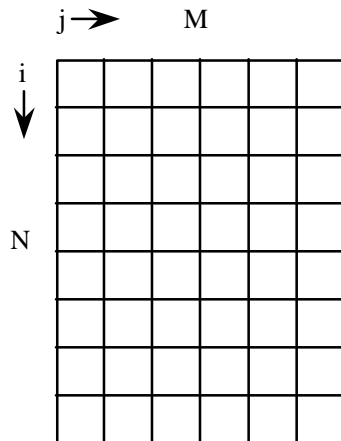
```

## 9.1 Multidimensional Repetitions

It is not possible to make multidimensional repetitions using the current version of the Mjølner System. However, multidimensional repetitions are easily constructed, e.g. a repetition with dimension  $N \times M$  can be declared like:

```
mul_table: [N*M] @Integer;
```

which is intended to realize a two-dimensional array of the form:



The following example shows how the multiplication table constructed in section 4 previously can be stored in a two-dimensional repetition:

```
ORIGIN '~beta/basiclib/ v1.6/betaenv'
---- program: descriptor ----
(#
  (* Multiplication Table 3
  *
  * Objective: Store values in a repetition
  *)
  N,M: @Integer;
do
  '\n\t** Multiplication Table ** \n\n'-> puttext;
  'Enter dimensions (NxM): '->puttext;
  getint -> N;
  getint -> M;
  (# mul_table: [N*M] @Integer;
  do (* build table *)
    (for i: N repeat
      (for j: M repeat
        i*j -> mul_table[(i-1)*M + j];
      for);
    for);

    (* print table *)
    newline;
    '    '->puttext;
    (for i: M repeat
      i -> screen.putint(# format::(# do 4-> width #)#);
    for);
    newline;
    (for i: N repeat
      i -> screen.putint(# format::(# do 4-> width #)#);
      (for j: M repeat
        mul_table[(i-1)*M + j]
        -> screen.putint(# format::(# do 4-> width #)#);
      for);
      newline;
    for)
  #);
#)
```

**Program 9.2: MultiplicationTable.bet**

# 10 Files

Our objective in this section is to open a file and analyze the characters that it contains. This means that we will be doing input for the first time. At first, we will merely count the characters in the file but we will also use command-line arguments to apply the program to various files.

File handling in BETA is quite painless. Files for both input and output are implemented through a single pattern: `file`. This pattern is not in the standard environment `betaenv` but in an extended library called `file` which includes `betaenv` so it is sufficient to replace `betaenv` by `file` in the `ORIGIN` statement.

**File Library**

A complete program skeleton to read a file called `data1` is as follows:

```
ORIGIN '~beta/basiclib/ v1.6/file'
--- program: descriptor ---
(# F: @File;
do 'data1' -> F.Name;
  F.openRead;

  (*... use F ...*)

  F.close;
#)
```

Here, `F` is declared as a `file` variable. The external file name is provided then `open-read` is invoked. After use, the `close` operation should be called. Here we have used only three attributes of the pattern `file`:

- Name
- OpenRead
- Close

**File attributes**

Other useful attributes/operations are:

- `OpenWrite`: creates an empty file or erases the current contents of an existing one
- `OpenAppend`: positions for writing at end of existing data
- `OpenReadWrite`: to allow `Get`, `Put` and `Pos` operations
- `Get`: returns the next character
- `Put`: writes or appends a character
- `Eos`: end-of-file check

The standard `while <not end-of-file>` loop for sequentially handling the contents of a file translates into the following BETA code:

```
Loop:
  (if not F.Eos then
    F.get -> ch; (* reading the next element *)
    ...
    restart Loop
  if);
```

**End-of-file loop**

Here is the whole program for counting the characters in the file `data1`. We have added visual feedback to the user by printing `!` on the screen for every 10 characters

read. Note that using the input/output predicates directly (i.e. put or get) without . notation (i.e. f.get) accesses the standard streams (keyboard and screen).

```
ORIGIN '~beta/basiclib/ v1.6/file'
---- program: descriptor ----

(# (*
 *   CountChar.bet: Simple file handling program
 *                       -Counting characters-
 *)

  inFile: @file;
  Ch:     @char;
  nc:     @integer;
do
  'data1' -> inFile.name; inFile.openRead; (* OPENING *)
  'Reading: '-> puttext;  inFile.name -> putline;
Loop:
  (if not inFile.eos then
    inFile.get -> ch;
    nc + 1 -> nc;
    (if nc mod 10 = 0 then '.' -> put if);
    restart Loop
  if);
  newline;
  nc -> putint; ' characters in file' -> putline;
  inFile.close;
#)
```

### Program 10.1: CountChar.bet

The output looks like this:

```
nil% CountChar
Reading: data1
....
41 characters in file
```

If the file data1 is not present, execution gives the following error message.

```
nil% CountChar

**** Exception processing
File exception for 'data1'
No such file

# Beta execution aborted: Stop is called
# Look at CountChar.dump'
```

**File Exception**

## 10.1 Command-line arguments

At present, our program only works with one file data1. It would be more useful if the name of the file could be specified by the user. A common way to allow this in UNIX is to pass the name of the files to be used as arguments on the command line used to invoke the program. For example, to count the characters in file data2, we would like to invoke count as follows:

```
% CountChar data2
```

The demo file programs that come with BETA all work this way. To do this, there are two useful functions in the standard environment which correspond to the UNIX C argc and argv variables. These are:

- noOfArguments : which returns the number of arguments on the command line and

- `arguments`: which given an integer parameter `N` returns the `n`'th argument on the command line. Remember that argument 1 is the name used for the program - `CountChar1` in our example - and that the one we want will be argument 2.

Below, we show how `count` has been modified to use the command argument. To keep things simple, we do not check the number of arguments or provide for an error message. For examples of how to do this, see the demo programs.

```

ORIGIN '~beta/basiclib/ v1.6/file'
---- program: descriptor ----

(# (* CountChar1.bet: Simple file handling program
   *           -Counting characters-
   *)

   inFile: @file;
   Ch:     @char;
   nc:     @integer;
do
  2 -> Arguments -> inFile.name;
  inFile.openRead; (* OPENING *)

  'Reading: '-> puttext;   inFile.name -> putline;
Loop:
  (if not inFile.eos then
    inFile.get -> ch;
    nc + 1 -> nc;
    (if nc mod 10 = 0 then '.' -> put if);
    restart Loop
  if);
  newline;
  nc -> putint; ' characters in file' -> putline;
  inFile.close;
#)

```

### Program 10.2: CountChar1.bet

And below we show the application of `countChar1` to the count program itself.

```

nil% CountChar1 CountChar1.bet
Reading: CountChar1.bet
.....
...
681 characters in file
nil%

```

A nice thing about passing file names as command line arguments is that the shell will expand the file name as expected. In particular, the `~` and `*` characters are interpreted correctly in the example below<sup>2</sup>:

```

nil% count ~/Beta/dat*
Reading: ../Beta/data1
....
41 characters in file

```

These also work:

- `count ~vaucher/Beta/data1`
- `count ../data1`
- `count ../Beta/file/../data1`

---

<sup>2</sup> For Unix shells only.

Were we to set the filename directly, it would be OK to include “.” and “..” in the path name but “~” would not be handled properly.

## 10.2 Storing Characters in a Repetition

The following example program illustrates how to count each occurrence of characters in the input file. The count for each character is stored in a repetition:

```
occurrences: [256]@Char;
```

using the assignment:

```
occurrences[Ch]+1->occurrences[Ch];
```

and the occurrences are printed using `for` and an `if` statement:

```
(for i:256 repeat
  (if occurrences[i]>0 then (* only print if > 0 *)
    i->put; (* notice how a char can be printed *)
    ': '->puttext;
    occurrences[i]->putint;
    newline;
  if);
for);
```

**Indexing using a char**

The complete program is as follows:

```
ORIGIN '~beta/basiclib/ v1.6/file'
---- program: descriptor ----

(# (* CountChar2.bet: Simple file handling program
  * -Counting occurrences of characters-
  *)

  inFile: @file;
  Ch: @char;
  nc: @integer;
  occurrences: [256]@Char;
do
  2 -> Arguments -> inFile.name ;
  inFile.openRead; (* OPENING *)

  Loop:
  (if not inFile.eos then
    inFile.Get -> Ch;
    nc+1->nc;
    occurrences[Ch]+1->occurrences[Ch];
    restart Loop
  if);

  newline;
  (for i:256 repeat
    (if occurrences[i]>0 then (* only print if > 0 *)
      i->put; (* notice how a char can be printed *)
      ': '->puttext;
      occurrences[i]->putint;
      newline;
    if);
  for);

  'Total '->puttext;
  nc -> putint;
  ' characters in file' -> putline;
  inFile.close;
#)
```

**Program 10.3: CountChar2.bet**

And below we show the application of CountChar2 to the CountChar2.bet program itself.

```
nil% CountChar2 CountChar2.bet
```

```
  : 241
 #: 2
' : 8
( : 8
) : 8
* : 11
+ : 2
- : 117
. : 7
/ : 7
0 : 2
1 : 3
2 : 3
4 : 1
5 : 2
6 : 2
: : 9
; : 21
> : 12
@ : 4
A : 1
C : 6
E : 1
F : 6
G : 3
I : 3
L : 3
N : 4
O : 2
P : 1
R : 2
S : 1
T : 1
[ : 5
] : 5
a : 20
b : 5
c : 34
d : 5
e : 55
f : 13
g : 6
h : 13
i : 41
l : 20
m : 5
n : 39
o : 26
p : 16
r : 34
s : 16
t : 31
u : 15
v : 1
w : 3
x : 2
y : 1
~ : 1
Total 952 characters in file
nil%
```



# 11 Directory

Directory handling is very similar to file handling in the Mjølner System. Files and directories have similar properties like `name`, `path`, etc. Files are special since the contents typically are characters that can be read and written, directories are special since the contents are files and directories. These similar and special properties are both modeled in the `file` and `directory` libraries.

**Directory and File**

When using the `directory` library, a directory is simply declared as

```
ORIGIN '~beta/basiclib/ v1.6/directory';
...
d: @directory;
...
```

A directory can be given a name:

```
'myDir' -> d.name;
```

**Directory attributes**

And it can be tested for existence, content, entries, and be scanned:

- `d.exists`: returns true if the directory exists
- `d.empty`: returns true if the directory has some content
- `d.noOfEntries`: returns the number of entries (files and directories) in the directory.
- `d.scanEntries`: calls `INNER` for each entry (found) in the directory.

The following program shows a simple use of `directory`: The directory with the path given as argument is scanned, and the names of all the entries are printed.

```
ORIGIN '~beta/basiclib/ v1.6/directory';
---program: descriptor---
(# d: @directory;
do (if noOfArguments <> 2 then
    'Usage: ' -> puttext; 1->arguments->puttext; ' path' -> putline;
    stop;
if);

(* set name of directory *)
2 -> arguments -> d.name;

(* print name of directory *)
newline;
d.name -> puttext;

(* test for content *)
(if d.empty then ' is empty.' -> putline;
 else ' is not empty.' -> putline;
if);

'It contains the following ' -> puttext;
d.noOfEntries -> putint;
' entries:' -> putline;

(* scan the entries *)
d.scanEntries
  (# (* found refers to the current entry *)
  do found.path -> putline;
```

```
        #);  
        newline;  
#)
```

### Program 11: ListDir.bet

This program also checks for the number of arguments. If the number of arguments is not 2, then an error message is printed, and `stop` is called. `stop` is defined in the basic environment `betaenv`, and when called, terminates the execution. **Stop operation**

Below we show the output of `ListDir` on the current working directory:

```
nil% ListDir .  
  
. is not empty.  
It contains the following 17 entries:  
.  
..  
CountChar.bet  
CountChar1.bet  
CountChar2.bet  
ExploreTypes.bet  
HelloWorld.bet  
ListDir  
ListDir.ast  
ListDir.bet  
MultipleAssignment.bet  
Multiplication1.bet  
Multiplication2.bet  
Multiplication3.bet  
SimpleTypes.bet  
SquareRoot.bet  
sun4s
```

## 12 Text Manipulation

The basic BETA environment defines a `Text` pattern for manipulating texts. Text constants have been used a lot in the previous examples. Here we explore more on the many facilities of the text concept. Constant texts can be assigned to text variables and texts can be added:

```
(# t: @text; (* declare t as a static ref. to a text object *)
  r: ^text; (* declare t as a dynamic ref. to a text object *)
  i: @integer;
do 'foo' -> t;          (* assign a constant to t = 'foo'  *)
  ' ' -> t.append;    (* append  one blank to t = 'foo ' *)
  ' ' -> t.prepend;  (* prepend one blank to t = ' foo ' *)
  t.length -> i;     (* assign the length of t to i (5) *)
  (2,4) -> t.sub -> r[]; (* get substring 'foo' from t *)
#)
```

Users do not have to bother about extending the text when adding or manipulating. The length of the text object will automatically be adjusted. Many functions on texts uses a *current position* in the text (`t.pos`). For example:

**Flexible and  
extensible text  
concept**

```
(# t: @text;
do 'foo'->t; (* sets pos to t.length *)
  'bar'->t.puttext; (* adds 'bar' after current pos: t='foobar'*)
  l->t.pos;
  'bar'->t.puttext; (* t = 'barbar' *)
#)
```

Texts sub-strings can be fetched and assigned to another text object *reference*, and texts can be inserted at a specified position:

```
(# t: @text; (* declare t as a static ref. to a text object *)
  r: ^text; (* declare t as a dynamic ref. to a text object *)
do 'foo' -> t;          (* assign a constant to t = 'foo' *)
  (2,4) -> t.sub -> r[]; (* get substring 'foo' from t *)
  ('bar',5) -> t.insert; (* insert substring 'bar' in t = 'foobar' *)
#)
```

Texts can be compared using the `equal` function.

```
(# t: @text;
  b: @boolean;
do ...
  'foo' -> t.equal -> b; (* case sensitive comparison *)
  'foo' -> t.equalNCS -> b; (* not case sensitive comparison *)
#)
```

**Comparing texts**

The following example program is an extended version of the character counting programs constructed before. The program can count either characters or lines in the input file. In addition to text comparison, the program uses two new features.

- `getline`: reads from input, i.e. what the user types. Waits until the user has typed a newline
- `ascii.newline`: `ascii` is an object defined in `betaenv` containing attributes for manipulating and comparing ASCII characters. `newline` is a generic definition of the newline character. `ascii` also contains conversion functions, e.g. `toLower`, definition of *white space*, e.g. `isWhiteSpace`, etc.

**ASCII object**

```

ORIGIN '~beta/basiclib/ v1.6/file'
---- program: descriptor ----

(# (* FileCount.bet: Simple file handling program
   *                               -Counting lines/characters-
   *)

   inFile: @file;
   Ch:      @char;
   nc:      @integer;
   answer: ^text;
   lines, chars: @Boolean;
do
  2->Arguments->inFile.name ;
  inFile.openRead; (* OPENING *)
  'Count what in ''->puttext;  inFile.name->puttext;
  '' (lines/chars)? '->puttext;
  (* read from keyboard - what the user types *)
  getline->answer[];
  (if true
   //('lines'->answer.equal) then true->lines;
   //('chars'->answer.equal) then true->chars;
   else
    'Unknown input'->putline;
    Stop; (* end execution *)
  if);
Loop:
  (if not inFile.eos then
   inFile.Get->Ch;
   (if true
    //lines then (if Ch//ascii.newline then nc + 1->nc if);
    //chars then nc + 1->nc;
   if);
   restart Loop
  if);
newline;
nc->putint;
(if true
 //lines then ' lines '->puttext;
 //chars then ' characters '->puttext;
if);
'in file ''->Puttext;
inFile.name->puttext;
''\n\n'->puttext;
inFile.close;
#)

```

### Program 12.1: FileCount.bet

The output running `FileCount` on itself is:

```

nil% FileCount FileCount.bet
Count what in 'FileCount.bet' (lines/chars)? lines

46 lines in file 'FileCount.bet'

nil% FileCount FileCount.bet
Count what in 'FileCount.bet' (lines/chars)? chars

1238 characters in file 'FileCount.bet'

```

Finally, the table below lists some of the useful attributes of texts:

<pre> t.length t.pos t.empty -&gt; b t.clear -&gt; b c -&gt; t.put t.get -&gt; c  t.peek -&gt; c  r[] -&gt; t.puttext r[] -&gt; t.prepend r[] -&gt; t.append i -&gt; t.putint  t.getint -&gt; i  t.getAtom -&gt; r[]  t.getLine -&gt; r[]  i -&gt; t.inxget -&gt; c (c,i) -&gt; t.inxput t.copy -&gt; r[] r[]-&gt;(t.copy).append-&gt;s[] r[]-&gt;(t.copy).prepend-&gt;s[] t.scanAtom(# do ... #)  t.scanAll(# do ... #)  (i,j) -&gt; t.sub -&gt; r[]  (i,j) -&gt; t.delete r[] -&gt; t.less  r[]-&gt;t.greater  t.makeLC t.makeUC c -&gt; findAll(# do ... #) t.EOSerror </pre>	<pre> Returns number of characters of text Returns current position Returns True if t is empty Sets the length to zero Appends the character c to t Returns the character at current position, and increments position by 1 Returns the character at current position, without updating the position Adds r to t starting at current position Prepends the text r to t Appends the text r to t Inserts the integer i to t starting at current position Reads the next integer from t starting at current position Reads characters until next white-space and returns the text Reads characters from t until next newline and returns that text. Returns the character at position i Replaces the character at position i Returns a copy of t Returns s[] where s = t cat r <sup>3</sup> Returns s[] where s = r cat t Scans from current position until next white-space and call INNER for each char Scans all the elements in t and calls IN - NER for each char Returns the text from position i to posi - tion j from t Deletes characters in the range i:j Tests whether r is less than t. Lexico - graphic ordering is used Tests whether r is greater than t. Lexico - graphic ordering is used Converts all characters to lower case Converts all characters to upper case Calls INNER for each occurrence of c in t Called when reading past length of the text </pre>	<b>Useful Text operations</b>
--	--	-----------------------------------

Please see the basic libraries manual [MIA 90-8] for more details about the text concept. **More information**

## 12.1 Advanced Formatted Input and Output

The Mjølner System also 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 defined in the `'~beta/basiclib/v1.6/formatio'` library.

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

**getFormat and  
putFormat**

---

<sup>3</sup> Actually this is an example of how to combine patterns that exits reference `append` is called on the reference returned by `copy`. This facility is called *computed remote*.

output), such as integer values. The markers are indicated in the string by a leading '%'. Following the '%' is the specification of the marker type.

In section 8 previously, the example program uses a complex `Dump` function to print out three numbers and some text. `putFormat` could have been used instead as illustrated in the following example.

```
ORIGIN '~beta/basiclib/ v1.6/formatio'
---- program: descriptor ----
(#
  (* Static and Dynamic references *)

  Point: (# X,Y: @integer;
          enter (X,Y)
          exit  (X,Y)
          #);

  refA : ^Point;
  P1,P2: @Point;

  Dump:
  (#
    do 'P1: %3d, P2: %d, refA: %d\n'->
      putFormat(# do P1.X -> d; P2.X -> d; refA.x -> d #)
    #);

do
  'Dynamic references' -> putline;
  newline;

  &Point[] -> refA[];

  (1,1) -> P1-> P2 -> refA;
  Dump;

  (2,2) -> P2;
  (3,3) -> refA;
  Dump;

  P1[] -> refA[]; Dump;
  P2[] -> refA[]; Dump;

  (1,1) -> P1; (2,2) -> P2; (3,3) -> refA;
  Dump;

  (111,333) -> &Point -> P1;
  Dump;
  Newline;
#)
```

### Program 12.2: StaticAndDynamic2.bet

The output is exactly the same as in section 8:

```
nil% StaticAndDynamic2
Dynamic references

P1: 1, P2: 1, refA: 1
P1: 1, P2: 2, refA: 3
P1: 1, P2: 2, refA: 1
P1: 1, P2: 2, refA: 2
P1: 1, P2: 3, refA: 3
P1: 111, P2: 3, refA: 3
```

# 13 Container Library

One of the strengths of the Mjølner System is the large set of available libraries. One of very useful libraries, is the container libraries. The container library supports a number of different ways to store data: sets, multisets, lists, hashtables, stacks, etc. Here we will show how to use `theList` and `theHashTable`.

## 13.1 List Example

The list library is available in the `'~beta/containers/ v1.6/list'`, thus this library must be included when using lists.

We will use the directory example above, and make a list to store the entries of the directory.

A list is simply declared as follows:

```
dirList: List
  (# element:: Text #);
```

Here we declare a pattern named `dirList` that inherits from the `list` pattern in the `'~beta/containers/ v1.6/list'` library. We specify the type of the elements in the list by extending the virtual pattern `element` to be the type of `text`. For details about the BETA concept of virtual patterns, see the BETA language introduction [MIA 94-26].

Virtual patterns  
in BETA

The `dirList` pattern inherits an operation from the `list` pattern that can be used to add elements, so we can simply add an element to the list by:

```
dirname[] -> dirList.append;
```

Lists have several other operations. Some useful operations are briefly described in the table below. Please see the container manual [MIA 92-22] for more details.

<code>l.clear</code>	Removes all elements currently in the list, making it empty
<code>l.empty -&gt; b</code>	Returns true if the list is empty
<code>l.size -&gt; i</code>	Returns the number of elements currently in the list
<code>equal:: (# ... #)</code>	Defines the equality test used by the implementation of the different operations. Users of list must further bind <code>equal</code> to contain the proper equality test for the specified element type. Default equality test for equal references (i.e. the same object)
<code>e[] -&gt; l.has</code>	Takes an element, and checks whether it is in the list
<code>l.scan(# ... #)</code>	Scans through the list, invoking <code>INNER</code> for each element in the list. In each turn of the scan, "current" refers to the current element in the list.
<code>l.copy-&gt; ll[]</code>	Default copy is one-level (shallow) copying. I.e. copying the list and all objects in the list.
<code>elm[]-&gt;l.prepend</code>	insert <code>elm</code> as first element
<code>elm[]-&gt;l.append</code>	insert <code>elm</code> as last element

Useful  
operations on  
List

In the table above, it is mentioned, that the equality test should always be defined. For our directory list this can be done like:

```
dirList: List
  (# element:: Text;
    equal::
      (# (* since the element type is text simply test whether the
          * two text strings are equal
          *)
        do left[]->right.equal->value;
        #);
  #);
```

**Specialize List**

The extended directory listing program can be as follows:

```
ORIGIN '~beta/basiclib/ v1.6/directory';
INCLUDE '~beta/containers/ v1.6/list';
---program: descriptor---
(# dirList: @List
  (# element:: Text;
    equal::
      (# (* since out element type is text simply test whether the
          * two text strings are equal
          *)
        do left[]->right.equal->value;
        #);
  #);
d: @directory;
do (if noOfArguments <> 2 then
    'Usage: ' -> puttext; 1->arguments->puttext; ' path' -> putline;
    stop;
  if);
  (* set name of directory *)
  2 -> arguments -> d.name;
  (* print name of directory *)
  newline;
  (* initialize list *)
  dirList.init;
  (* scan the entries and append to list *)
  d.scanEntries
  (# (* found refers to the current entry *)
    do found.path -> dirList.append;
    #);
  (* dirList now contains all the names of the entries in the
  * directory
  *)
  dirList.scan
  (# (* current refers to the current text element *)
    do current[] -> putline; (* print the text *)
    #);
#)
```

### Program 13.1: SaveListDir.bet

Later we shall see how this list can be saved (persistent) and used in another program.

## 13.2 HashTable Example

We could also choose to save the file list in a hash table. A hash table is typically used to store objects that should be retrieved fast from the table. In order to store an object in a hash table it is necessary to define a hash function that given an element returns a value that can be used in the hash table implementation. A good hash function for our file list could be



```

hashFunction::
  (# (* scan all characters in the filename and compute a value
      * for the hash function
      *)
  do name.scanAll(# do value*100 + ch -> value #);
  #);

```

The hashtable can then be defined as follows:

```

dirTable: @ hashTable
  (# element:: Text;
    hashFunction::
      (#
        do e.scanAll(# do value*100 + ch -> value #);
        #);
  #);

```

And the complete program using this hash table:

```

ORIGIN '~beta/basiclib/ v1.6/directory';
INCLUDE '~beta/containers/ v1.6/hashTable;
---program: descriptor---
(# dirTable: @ hashTable
  (# element:: Text;
    hashFunction::
      (#
        do e.scanAll(# do value*100 + ch -> value #);
        #);
  #);
d: @directory;
do (if noOfArguments <> 2 then
  'Usage: ' -> puttext; 1->arguments->puttext; ' path' -> putline;
  stop;
  if);
(* set name of directory *)
2 -> arguments -> d.name;
(* print name of directory *)
newline;
(* initialize table *)
dirTable.init;
(* scan the entries and append to list *)
d.scanEntries
  (# (* found refers to the current entry *)
  do found.path -> dirTable.insert;
  #);
(* dirTable now contains all the names of the entries in the
  * directory
  *)
dirList.scan
  (# (* current refers to the current text element *)
  do current[] -> putline; (* print the text *)
  #);

(* print hashtable statistics on screen*)
'\nStatistics: '->screen.putline;
dirTable.statistics(# do screen[]->print #);
#)

```

### Program 13.2: DirTable.bet

Running this program on the current directory gives the following output:

```

nil% DirTable .

StaticAndDynamic.bet
FileCount.bet
ExploreTypes.bet
CountChar2.bet

```

```
CountChar1.bet
CountChar.bet
Multiplication3.bet
Multiplication2.bet
Multiplication1.bet
MultiplicationTable.bet
DirTable.bet
DirTable.ast
HelloWorld.bet
SaveListDir.bet
QuickSort.bet
SquareRoot.bet
..
.
MultipleAssignment.bet
DirTable
sun4s
ListDir.bet
SimpleTypes.bet

Statistics:
Histogram: (0,2,1,3,0,0,0,0,2,3,0,3,2,0,0,2,0,0,0,1,2,2,2,0,1)
Maximum Collisions: 3
Minimum Collisions: 0
Average Collisions: 2
```

More information and more examples using the other containers in this library can be [More information](#) found the MjølnerSystem manual [MIA 92-22].

# 14 Exceptions

The pattern `exception` defined in `'~beta/basiclib/ v1.6/betaenv'` is used as a superpattern for all exceptions in the Mjølner System. The default action of an exception is to stop the program execution and print an informative error message on the screen. In addition, the file `<programname>.dump` contains a dump of the call stack. `Exception` uses 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. If you wish to prevent the program execution from being stopped in order to handle the exception during execution, the boolean attribute `continue` of `exception` must be set to `true`.

**Exception  
pattern**

The exceptions are often defined as a virtual pattern of other patterns (such as the `file` pattern, discussed below).

In order to differentiate between potential fatal exceptions and more harmless exceptions, the `notification` pattern is also defined in `betaenv` defined as:

**Notification  
pattern**

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

## 14.1 Examples Using Exception

In order to illustrate the use of exceptions, let us return to the previous file example. Without using the exception handling facilities an attempt to open a non-existing file produced the following error messages:

```
nil% CountChar

**** Exception processing
File exception for 'data1'
No such file

# Beta execution aborted: Stop is called
# Look at CountChar.dump'
```

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

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. The binding of `noSpaceError` shows that a message can be added to `msg`.

```
ORIGIN '~beta/basiclib/ v1.6/file'
(# inFile: @file
  (# noSuchFileError:: (* continue execution *)
    (# do true->continue; false->OK #)#);
  outFile: @file
  (# noSpaceError:: (* extend exception; put message to msg *)
    (# do 'It is time to delete garbage!'->msg.putline #)#);
  OK: @boolean;

do 'in.bet' -> inFile.name;
true -> OK;
openFile: (* labeled block *)
(#
do inFile.openRead;
  (if not OK then
```

```

        'File does not exist!' -> screen.putline;
        'Type input file name: ' -> screen.puttext;
        inFile.readFileName;
        true -> OK;
        restart openFile (* restart labeled block *)
if)#);

'out.bet' -> outFile.name;
outFile.openWrite;
readFile:
(#
do (if not inFile.eos then
    false -> inFile.gettext -> outFile.puttext;
    outFile.newline;
    restart readFile
    else leave readFile
if)#);
inFile.close;
outFile.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 possibility to proceed with another file name.

In case of disk space exhausted, the following message will be printed on the screen before the program execution is stopped:

```

**** Exception processing
Error in file 'in.bet'
File system is full
It is time to delete garbage!

```

The first line is from the general pattern `exception`, the second and the third lines are from the binding of `noSpaceError` in file and the fourth line is from the binding above, i.e. at the user level.

# 15 Access to External Functions and Data

The Mjølner System allows a tight integration between the BETA language and routines and data structures, originating from the C language. Many of the libraries in the Mjølner System (such as the interface to the X Window System) is based on this tight integration.

The integration allows for two types of integration, namely integration of routines, and integration of data structures. The facilities give the BETA programmer the possibility to invoke routines, written in C, and for accessing data structures, allocated in C. Moreover, the facilities also works the other way around, namely by allowing BETA patterns to be invoked (instantiated) from C routines, and BETA objects to be manipulated by C routines.

**Integration of routines and data**  
**Callbacks from C to BETA**

## 15.1 Example

Imagine that we have a database with person records. The database has a C interface and we like to use the database in BETA.

The following C declarations and functions illustrates a simplified database:

```
typedef struct Person {
    long ID;
    char *firstname,*surname;
    char sex; /* m(ale) or f(emale) */
} Person;
```

```
#define MaxPersons 200
```

```
Person Persons[MaxPersons];
```

```
Person *getPerson(long ID) {
    if (ID>=0 && ID<MaxPersons)
        return &Persons[ID];
    else
        return 0;
}
```

```
long putPerson(long ID, char * firstname, char* surname, char sex) {
    if (ID>=0 && ID<MaxPersons) {
        Persons[ID].ID=ID;
        Persons[ID].firstname=firstname;
        Persons[ID].surname=surname;
        Persons[ID].sex=sex;
        return 1;
    } else {
        return 0;
    }
}
```

**The C database interface**

We can then interface to the two functions and the Person struct by the following external and externalRecord declarations:

```
getPerson: external
    (# ID: @integer;
```

```

        ptr: @integer;
    enter ID
    exit ptr
    #);
putPerson: external
    (# ID: @integer;
     firstname, surname: [1]@char;
     sex: @char;
     result: @boolean;
    enter (ID,firstname,surname,sex)
    exit result
    #);
Person: externalRecord
    (# ID: @long(# pos::(# do 0-> value #)#);
     firstname: @long(# pos::(# do 4-> value #)#);
     surname: @long(# pos::(# do 8-> value #)#); (* char ptr *)
     sex: @byte(# pos::(# do 12-> value #)#);
    #);

```

Interfacing to C routines are done by specifying the `external` pattern as the superpattern for the BETA pattern, which, when invoked, should invoke the C routine. The name of the entry call of the C routine should be the same as the name of the BETA pattern. 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. The pattern:

**external  
pattern**

```

getPerson: external
    (# ID: @integer;
     ptr: @integer;
    enter ID
    exit ptr
    #);

```

describes the interface to an external C function with the name `getPerson`.

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 externally. `externalRecord` is the means for specifying a BETA interface into some data structures, allocated externally. The pattern:

**cStruct and  
externalRecord**

```

Person: externalRecord
    (# ID: @long(# pos::(# do 0-> value #)#);
     firstname: @long(# pos::(# do 4-> value #)#);
     surname: @long(# pos::(# do 8-> value #)#); (* char ptr *)
     sex: @byte(# pos::(# do 12-> value #)#);
    #);

```

describes an interface to an external allocated structure (`Person`) with four fields.

We can create a person by calling `putPerson` like this:

```
(117, 'Roger', 'Smith', 'm')->putPerson
```

We can get a person from the database by:

```
117 -> getPerson -> aPerson.ptr;
```

Notice, that we must assign to the `ptr` attribute of the `externalRecord Person`.

The `Person` can now be examined like any other BETA object, except for the “string” declarations `firstname` and `surname`. These refers to C strings. The Mjølnér System includes a `cString` library for easy interface to these C strings, so we simply make a small operation to print out the strings:

**cString**

```

putCString:
  (# cstr: @cString;
  enter cstr
  do cstr.get -> puttext;
  #);

```

Finally we must specify where to find the C object file we are interfacing to. This is **OBJFILE** done using **OBJFILE** specification. The specification:

```

OBJFILE nti      '$/cperson.obj'
        mac      '$/cperson.obj'
        default '$/cperson.o';

```

means that we should link with the file `cperson.o` located in the subdirectory with the name of the platform—the same name as the code subdirectory ('`$`' expands to name of platform).

The C object file can also be created automatically by using make files. The **MAKE** specification:

```

MAKE     nti      'person_nti.make'
        mac      'person_mac.make'
        default 'person_unix.make';

```

describes for each platform which make file must be used. The Unix version looks like:

```

$(MACHINETYPE)/cperson.o: cperson.c
  $(CC) -c -o $(MACHINETYPE)/cperson.o cperson.c

```

And now the complete program:

```

ORIGIN '~beta/basiclib/ v1.6/external';
INCLUDE '~beta/sysutils/ v1.6/cstring';
OBJFILE nti      '$/cperson.obj'
        mac      '$/cperson.obj'
        default '$/cperson.o';
MAKE     nti      'person_nti.make'
        mac      'person_mac.make'
        default 'person_unix.make';
--program: descriptor--
(
  #
  getPerson: external
  (# ID: @integer;
  ptr: @integer;
  enter ID
  exit ptr
  #);
  putPerson: external
  (# ID: @integer;
  firstname, surname: [1]@char;
  sex: @char;
  result: @boolean;
  enter (ID,firstname,surname,sex)
  exit result
  #);
  Person: ExternalRecord
  (# ID: @long(# pos::(# do 0-> value #)#);
  firstname: @long(# pos::(# do 4-> value #)#);
  surname: @long(# pos::(# do 8-> value #)#); (* pointers to
text *)
  sex: @byte(# pos::(# do 12-> value #)#);
  #);
  putCString:
  (# cstr: @cString;
  enter cstr
  do cstr.get -> puttext;

```

```

        #);
aPerson: @Person;
do
(* store a person in C-database *)
(if not ((117, 'Roger', 'Smith', 'm')->putPerson) then
    'Failed to store person'->putline; stop;
if);
(* get person from C-database *)
117 -> getPerson -> aPerson.ptr;
(if aPerson.ptr = 0 then
    'Failed to retrieve person' -> putline; stop;
if);
'Person: ' -> puttext;
aPerson.ID -> putint;
' ' -> put;
'Name: '' -> puttext;
aPerson.firstname -> putCString;
' ' -> put;
aPerson.surname -> putCString;
'' ' -> puttext;
'Sex: ' -> puttext;
aPerson.sex-> put;
newline;
#)

```

### Program 15: Person.bet

Output of running the program is:

```

nil% Person
Person: 117 Name: 'Roger Smith' Sex: m

```



# 16 Using the Persistence Library

The persistence library can be used to save your data on the disk for later use in another program execution. Any object created can be saved using the persistence library. The patterns defining the objects do not have to be extended in any way before the objects can be saved. Imagine that we like to save the character count in the previous example for usage in another program. The pattern definition of the `directoryList` can be described in a separate file (called `dirList.bet`) as follows:

```
ORIGIN '~beta/containers/ v1.6/list
--- lib: Attributes ---
  directoryList: List
  (# ...
  #);
```

**Making a library**

## Program 16.1: DirList.bet

Notice, that we do not define a program fragment in this file, instead we define attributes only. A file describing simple pattern declarations only can use the slot called `lib` defined in the `betaenv` environment (see section 19 below about the fragment system, for more details). The declarations in the `DirList` file can be used by including the file in the program. Thus the program listed Program 13.1 can be changed like:

```
ORIGIN '~beta/basiclib/ v1.6/file';
INCLUDE 'DirList'
---- program: descriptor ----
(# dir: @directory;
  dirList: ^directoryList;
do &directoryList[] -> dirList[];
  ...
#)
```

## Program 16.2: SaveListDir2.bet

We can now save `DirList` using the persistent store, The persistent store is available as a library in the file `'~beta/persistentstore /v1.6/persistentstore`. By including this file we can use the `persistentstore` pattern to save the list. `persistentstore` has the following useful operations:

- `persistentstore.create` : given a text create a persistent store with that name
- `persistentstore.openWrite` : given a name opens the persistent store with read and write permission. `openRead` opens a store with read permission only
- `persistentstore.get` : given a name and a pattern variable, returns an object in the storage with that type
- `persistentstore.put` : given a name and an object, stores that object in the persistent store
- `persistentstore.close` : closes the persistent store

**Operations on a persistent store**

The following program is similar to the one above, except that it stores the `dirList` in a persistent store.

```

ORIGIN '~beta/basiclib/ v1.6/file';
INCLUDE '~beta/persistentstore/ v1.6/persistentstore;
INCLUDE 'DirList'
---- program: descriptor ----

(# (* Saving the file names in a persistent store *)

    dir: @directory;
    theStore: @persistentstore;
    dirList: ^directoryList;
do &directoryList[] -> dirList[];

    ... (* Program 13.1 *)

    'fileStore'->theStore.create;
    (dirList[], 'myList')->theStore.put;
    theStore.close;
#)

```

### Program 16.3: SaveListDir3.bet

The persistent store is now located in the file directory `fileStore`.

Finally, we can make a program that reads the list, and examines the data:

```

ORIGIN '~beta/basiclib/ v1.6/file';
INCLUDE '~beta/persistentstore/ v1.6/persistentstore;
INCLUDE 'DirList'
---- PROGRAM: descriptor ----

(# (* Reading counted occurrences of characters
    * from a persistent store
    *)
    theStore: @persistentstore;
    dirList: ^directoryList;
do
    'fileStore'->theStore.openWrite;
    ('myList', directoryList##)->theStore.get->dirList[];
    dirList.scan(# ... #);
    ...
    theStore.close;
#)

```

### Program 16.4: GetListDir.bet

A complete description of the facilities in the persistent store library can be found in **More information** [MIA 91-20].

# 17 Graphical User Interface

In this section we will show how to use two different graphical user interface libraries. The first example uses the device independent library called GUIEnv and the second example uses the X11 specific MotifEnv library.

## 17.1 GUIEnv

GUIEnv is a device independent graphical user interface library, intended for making applications with graphical user interfaces running on:

- Macintosh
- X Window System(Motif Widgets)
- Windows(Win32)

GUIEnv realizes user interfaces of many different look-and-feels. GUIEnv allows construction of portable user interfaces in such a way that the look-and-feel of the applications, will conform to the standardized look-and-feel of the specific platform.

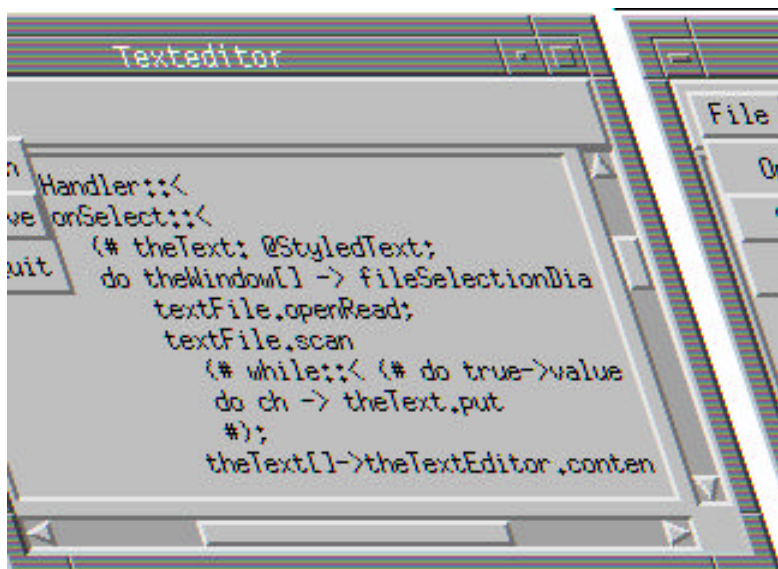
The basic GUI library is defined in the file '~beta/guienv/ v1.6/guienv'. An application with a graphical user interface, thus must have origin in this file. The guienv library defines a pattern also called guienv in which all the user interface attributes, operations, and patterns are available. So every GUIEnv application typically has the following outline:

```
ORIGIN '~beta/guienv/ v1.6/guienv';
-- program: descriptor --
guienv
  (# (* write GUI code here *)
  #)
```

**guienv  
application  
outline**

Our task will be to develop a simple texteditor that can open a text file, edit the file and save the file. We would like an application that looks like the following (Motif version):

**Texteditor  
example**



We will need a window to display the text in. `guienv` defines a `window` pattern, that we can use. We would like that this window is opened when the application starts up, so in the `do` part of the specialization of `guienv` we open the window. **Window**

```
guienv
(# theWindow: @window
  (# ...
  #);
do theWindow.open;
#);
```

Inside the window we want a `textEditor` that can contain the text from the file. `guienv` supplies a `textEditor` for this purpose: **TextEditor**

```
theWindow: @window
  (# theTextEditor: @textEditor
    (# open::
      (#
        do theWindow.size -> Size;
        True -> bindBottom; True -> bindRight
      #)
    #);
  open::(# do theTextEditor.open #);
#);
do theWindow.open;
```

The lines:

```
open::
  (#
  do theWindow.size -> Size;
  True -> bindBottom; True -> bindRight
  #);
```

means that we extend the `open` virtual of `textEditor`, set the size of the `textEditor` to be same size as the window, and bind the `textEditor` to the bottom and right corners of the window.

The next thing we need to do, to complete the user interface is to make a menu. The application should have a menu with three items: open a file, save the file, and quit the application. `guienv` supplies a standard menubar on each window for this purpose. We extend the standard menubar with one menu called `File`, and we make three items in this menu called `Open`, `Save`, and `Quit`. **File menu**

```
menubarType::
  (# fileMenu: @menu
    (# openItem: @menuItem
      (# ... #);
    saveItem: @menuItem
      (# ... #);
    quitItem: @menuItem
      (# ... #);
    open::
      (#
        do 'File' -> name;
        openItem.open; openItem[] -> append;
        saveItem.open; saveItem[] -> append;
        quitItem.open; quitItem[] -> append;
      #)#);
  open::(# do fileMenu.open; fileMenu[] -> append #);
#);
```

Like the `textEditor`, we extend the `open` virtual of the `File` menu to open and append the three items and to give the menu a title.

Each menuitem has two virtuals that needs to be extended: `eventHandler` and `open`. **Quit item**  
 For the quit item we do the following:

```
quitItem: @menuitem
  (# eventHandler::
    (# onSelect::(# do Terminate #)
      #);
    open::(# do 'Quit' -> name #);
  #);
```

The `eventHandler` has a virtual called `onSelect` that is invoked whenever this menuitem is selected. We call `Terminate` (defined in `guienv`) to stop execution. The `open` virtual is extended to give the item a name.

Finally, we need to do some file handling in the `open` and `save` items. The `open` item **Open item** does the following when selected

```
onSelect::
  (# theText: @StyledText;
  do theWindow[] -> fileSelectionDialog -> textFile.name;
  textFile.openRead;
  textFile.scan
    (# while:: (# do true->value #);
    do ch -> theText.put
    #);
  theText[]->theTextEditor.contents.contents;
  textFile.close;
  #)
```

First we call `fileSelectionDialog` that opens a standard file open dialog and returns **Standard file open** a name of a file that we can open (we ignore errors, pressing cancel, etc.). We open the file, read all the file content into a `StyledText` and sets the `StyledText` as the content of the `textEditor`. `StyledText` is a specialization of `Text` with specification of face, font, size, etc.

The `save` item does the following when selected: **Save item**

```
onSelect::
  (# theText: @Text;
  do textFile.openWrite;
  theTextEditor.contents.contents->textFile.puttext;
  textFile.close;
  #)
```

Because `StyledText` is a specialization of `Text` we can write the `StyledText` contents of the `TextEditor` directly to the file using the `puttext` operation.

We open the same file, write the `textEditor` content into the file and close the file.

The complete code needed for this application is shown below.

```
ORIGIN '~beta/guienv/ v1.6/guienv';
INCLUDE '~beta/guienv/ v1.6/fields'
      '~beta/guienv/ v1.6/stddialogs'
      '~beta/basiclib/ v1.6/file';
-- program: descriptor --
guienv (* inherit from guienv *)
(# theWindow: @window (* make a window *)
  (# menuBarType:: (* extend the menuBar *)
    (# fileMenu: @menu (* make a file menu *)
      (# textFile: @file; (* the file we open and save *)
        openItem: @menuitem (* make an open item *)
        (# eventHandler::
          (* extend the virtual that is called when
            * this menu item is selected *)
          (# onSelect::
            (# theText: @StyledText;
```

**The complete code**

```

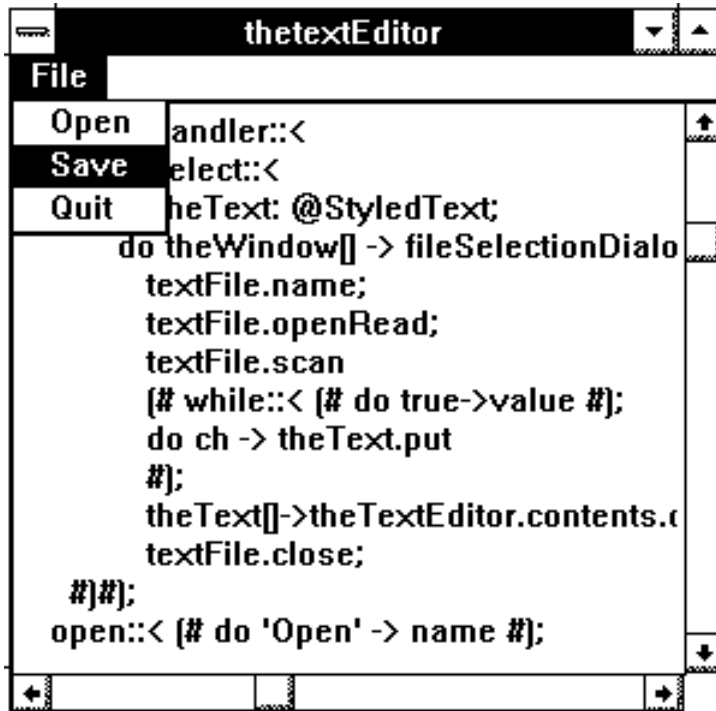
do theWindow[] -> fileSelectionDialog
  -> textFile.name;
  textFile.openRead;
  textFile.scan
  (# while:: (# do true->value #);
  do ch -> theText.put
  #);
  theText[]->
    theTextEditor.contents.contents;
  textFile.close;
  #)#);
  open:: (# do 'Open' -> name #);
#);
saveItem: @menuItem (* make a save item *)
  (# eventHandler::
  (# onSelect::
  (# theText: @Text;
  do textFile.openWrite;
  theTextEditor.contents.contents->
  textFile.puttext;
  textFile.close;
  #)#);
  open:: (# do 'Save' -> name #);
#);
quitItem: @menuItem (* make a quit item *)
  (# eventHandler::
  (# onSelect:: (# do Terminate #) #);
  open:: (# do 'Quit' -> name #);
  #);
open:: (* extend the open virtual of filemenu
  * to open the items *)
  (#
  do 'File' -> name;
  openItem.open; openItem[] -> append;
  saveItem.open; saveItem[] -> append;
  quitItem.open; quitItem[] -> append;
  #)#);
open:: (* extend the open virtual of the menubar
  * to open the filemenu *)
  (# do fileMenu.open; fileMenu[] -> append #);
#);
thetextEditor: @textEditor (* our text editor *)
  (# open:: (* extend the open virtual
  * to set the size and the placement *)
  (# do theWindow.size -> Size;
  True -> bindBottom; True -> bindRight
  #)#);
open:: (* extend the window open virtual
  * to open the textEditor *)
  (# do thetextEditor.open #);
#);
do theWindow.open; (* open the window when the appl. start up *)
#)

```

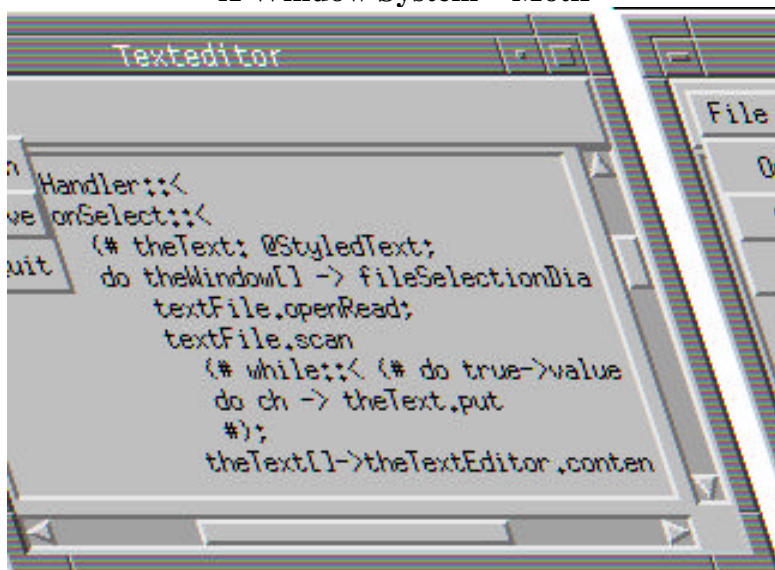
### Program 17.1: TextEditor.bet

The three screen snapshots following below show how this application appears on Windows NT, Motif (X11), and Macintosh after the program has loaded its own source code for editing, and with the menu opened.

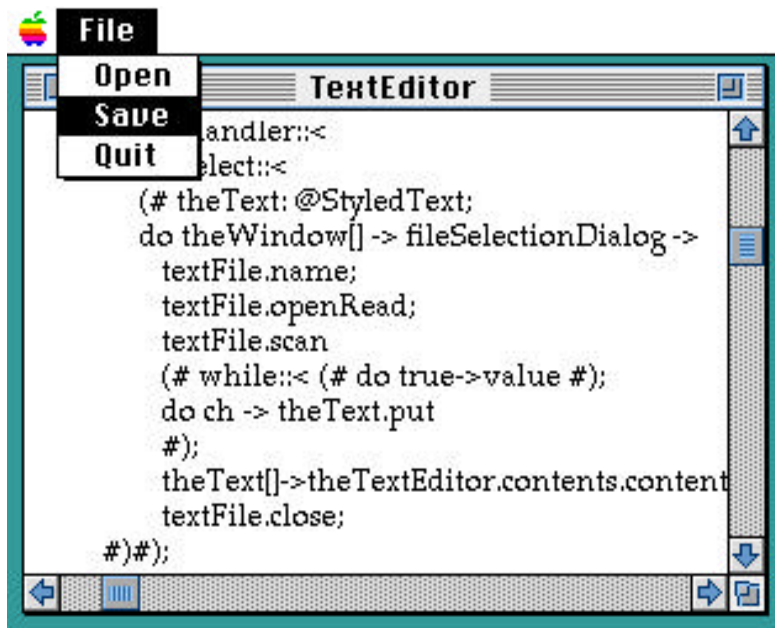
### Windows NT



### X Window System—Motif



## Macintosh



More details and examples about the GUIEnv libraries can be found in [MIA 94-27]. [More information](#)

### 17.2 MotifEnv

In this subsection we will show how to use the Motif specific user interface library. Motif is a very large user interface toolkit, with user interface elements not easily abstracted into a general device independent framework, so the Mjølnær system includes a library for making Motif specific applications.

#### Xt Toolkits

Xt is a C-library top of Xlib, the low-level interface used in programming X Window System applications. The purpose of Xt is to provide an object-oriented layer that supports user-interface abstractions (windows, scrollbars, commands buttons, menus) called widgets. A widget is a reusable, configurable piece of C-code that operates independently of the application except through prearranged interactions.

Xt contains the basic functionality to support widgets, i.e. an architectural model for widgets that allow them to be written and used in an object-oriented fashion. Xt also contains a small core set of widgets.

**Widgets**

A widget set is a collection of widgets build on top of Xt that provide commonly used user-interface components tied together with a consistent appearance and user interface. Several different widget sets from various sources exist. The Athena widget set is one example. Others are Motif from Open Software Foundation (OSF) and OPEN WINDOWS from Sun and AT&T.

The Motif widget set contains many user-interface components, including scroll bars, menus, buttons, dialogs, and a wide variety of composite widgets. Motif has conventions about the use of its widgets and gadgets, that lead to a consistent look among all applications using Motif. Along with each Motif license comes an *OSF/Motif Style Guide* with the documentation. This document contains recommendations for application design and layout.

**OSF/Motif**

The Mjølnær System comes with object-oriented interfaces to these libraries, XtEnv, AwEnv and MotifEnv.



## Using the MotifEnv Fragment

The `motifenv` file in `'~beta/Xt/v 1.10/MotifEnv'` simply defines the `MotifEnv` pattern for applications using the BETA interface to Motif. For each widget/gadget wanted in the application, the files in the directory `'~beta/Xt/v 1.10/motif/'`, defining the BETA interface to it, must be explicitly included.

**MotifEnv**

An application using `motifenv` thus typically has the following outline:

```
ORIGIN '~beta/Xt/v1. 10/MotifEnv';
INCLUDE '~beta/Xt/v 1.10/motif/rowcolumn'
        '~beta/Xt/v 1.10/motif/pushbutton'
-- PROGRAM: descriptor --
MotifEnv
  (# ...
  do ...
  #)
```

In this case the program is using the `rowcolumn` and `pushbutton` widgets.

The following small program shows how to make the traditional "Hello world" program using a Motif Label widget:

**hello.bet**

```
ORIGIN '~beta/Xt/v 1.10/motifenv'
INCLUDE '~beta/Xt/v 1.10/motif/label'
-- program: descriptor --
MotifEnv
  (# hello: @Label;
  do hello.init;
    'Hello world' -> hello.labelString;
  #)
```



**Program 17.2: Hello.bet**

The border of the window, with grips for resizing the window, and the title bar with buttons, is added by the window manager, in this case *mwm*, the Motif Window Manager. The actual Label widget is the one showing the "Hello world" text.

The following example shows how to use the special purpose Motif Scale widget, useful for adding, e.g., a potentiometer-like control to a panel of controls.

**Scale.bet**

```
ORIGIN '~beta/Xt/current/motifenv':
INCLUDE '~beta/Xt/current/motif/scale'
-- program: descriptor --
MotifEnv
  (# volume: @Scale
    (# init::
      (#
        do 0 -> minimum;
          100 -> maximum;
          XmVERTICAL -> orientation;
          'Volume' -> titleString;
          true -> showValue;
        #);
      valueChangedCallback::
        (#
          do 'New volume: ' -> screen.puttext;
            data.value -> screen.putint;
            screen.newline;
          #);
    #);
  do volume.init;
  #)
```



**Program 17.3: Scale.bet**

The `valueChangedCallback` is called after the scale has been changed. In this case the callback is extended and prints out the new value of the `Scale` (`ta.value`).

More details and examples about the X11 libraries can be found in [MIA 91-16].

**More information**

# 18 Concurrent Library

Concurrent programming in BETA is supported by the `systemenv` library. This library contains patterns for describing the BETA concepts of concurrent systems. The basic ideas are:

1. Components (`coroutines`) can be executed concurrently. **Concurrent coroutines**
2. A primitive `semaphore` pattern is available for synchronization. The operations on a `semaphore` is executed as an indivisible unit.
3. An abstract pattern `Monitor` similar to the monitor proposed by Hoare and Brinch-Hansen.
4. 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`. **Pattern System**

## 18.1 Example

The following example of using the `systemenv` library makes three concurrent coroutines that each sleeps for a specified number of seconds and then prints out the seconds elapsed since startup. All three systems inherit from a generic system:

```
everyNthSecond: System everyNthSecond
  (* inherit from System: can run concurrently *)
  (# N:< IntegerValue;
   now: @Integer;
   do cycle
     (#
      do INNER everyNthSecond;
      N -> sleep; (* sleep for N seconds *)
      now+N -> now; (* accumulate time *)
     #)
  #)
```

`everyNthSecond` inherits from `system`, i.e. it is able to run concurrently. The `do`-part consists of a loop that calls `INNER`, then sleeps for `N` seconds, and when activated again, updates the time, and calls `INNER` again. Notice, that `N` is defined as an `IntegerValue`.

Now we can make a coroutine that inherits from `everyNthSecond` like this:

```
fourth: @| everyNthSecond
  (* a co-routine that inherits from everyNthSecond *)
  (# N:: (# do 4 -> value #); (* sleep for 4 seconds *)
   do 'fourth: ' -> puttext; now -> screen.putint; newline
  #)
```

A coroutine is declared using the `'|'` symbol. The declaration `'@|'` means that we declare `fourth` to be a static reference to a coroutine. The `do`-part will be called every 4'th second since `fourth` inherits from `everyNthSecond` and extends the `IntegerValue N` to be 4. **Declaring a coroutine**

Finally, we need to start the coroutines concurrently. This is done by starting the coroutine inside a `conc` pattern like this:

```

conc (* execute concurrently: *)
  (# do ... fourth[]->start; ... #)

```

The execution of `conc` will not terminate until all the systems executed inside it has terminated.

The complete program with three concurrent systems is shown in Program 18.1. Notice, that the program never terminates.

```

ORIGIN '~beta/basiclib/ v1.6/systemenv'
--- program: descriptor ---
systemenv (* inherits from systemenv *)
(
  (* everyNthSecond calls INNER every N'th second. *)
  everyNthSecond: System
    (* inherit from System: can run concurrently *)
    (# N:< IntegerValue;
      now: @Integer;
      do cycle
        (#
          do INNER everyNthSecond;
            N -> sleep; (* sleep for N seconds *)
            now+N -> now; (* accumulate time *)
          #);
    #);
  every: @| everyNthSecond
    (* a co-routine that inherits from everyNthSecond *)
    (# N::< (# do 1 -> value #); (* sleep for 1 second *)
      do 'every: ' -> puttext; now -> screen.putint; newline;
    #);
  fourth: @| everyNthSecond
    (* a co-routine that inherits from everyNthSecond *)
    (# N::< (# do 4 -> value #); (* sleep for 4 seconds *)
      do 'fourth: ' -> puttext; now -> screen.putint; newline;
    #);
  eighth: @| everyNthSecond
    (* a co-routine that inherits from everyNthSecond *)
    (# N::< (# do 8 -> value #); (* sleep for 8 seconds *)
      do 'eighth: ' -> puttext; now -> screen.putint; newline;
    #);
do
  conc (* execute concurrently: *)
    (# do every[]->start; fourth[]->start; eighth[]->start #);
  (* terminates when all systems stops. In this case: never *)
#)

```

### Program 18.1: Seconds.bet

Output of running Program 18.1 for 20 seconds:

```

nil% Seconds
every: 0
fourth: 0
eighth: 0
every: 1
every: 2
every: 3
fourth: 4
every: 4
every: 5
every: 6
every: 7
eighth: 8
fourth: 8
every: 8
every: 9
every: 10

```

```

every: 11
fourth: 12
every: 12
every: 13
every: 14
every: 15
eighth: 16
fourth: 16
every: 16
every: 17
every: 18
every: 19
fourth: 20

```

## 18.2 Concurrency and User Interface Environments

Graphical 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` exist for the different user interface libraries, such as `asMotifEnv` and `guienv`:

Use `~beta/basiclib/ v1.6/systemenv` as origin for programs not using event-driven user-interface libraries.

Use `~beta/Xt/ v1.10/xsystemenv` as origin for programs using the Motif user interface library.

Use `~beta/guienv/ v1.6/guienvsystemenv` as origin for programs using GUIEnv interface library.

Please note, that programs should only use *one* of the `systemenv`, `xsystemenv`, and `guienvsystemenv` fragments.

Suppose that we like to extend the `texteditor` above with a clock that should be updated every second. A clock can easily be made using the basic `systemenv`: **A simple clock**

```

ORIGIN '~beta/basiclib/ v1.6/systemenv' ;
INCLUDE '~beta/sysutils/ v1.6/time'
-- program: descriptor --
systemenv
( #
  updateClock: @|System
  ( #
    do cycle
      ( #
        do 1 -> sleep;
          systemtime -> formattime -> putline
        #)
      #)
  do updateClock[] -> fork
  #)

```

### Program 18.2: Clock.bet

Here we simply print out the current system time on the screen. Notice, that we have included a new library called `time` in `'~beta/sysutils/ v1.6/time'`. This library contains facilities for getting the date and time, time usage, and for formatting times for nice printing. Running the program shown above gives the following result: **time library**

```

Tue Aug 23 11:48:35 1994
Tue Aug 23 11:48:36 1994
Tue Aug 23 11:48:37 1994
Tue Aug 23 11:48:38 1994
Tue Aug 23 11:48:39 1994

```

```

Tue Aug 23 11:48:40 1994
Tue Aug 23 11:48:41 1994
Tue Aug 23 11:48:42 1994
Tue Aug 23 11:48:43 1994
Tue Aug 23 11:48:44 1994
Tue Aug 23 11:48:45 1994
Tue Aug 23 11:48:46 1994
Tue Aug 23 11:48:47 1994
Tue Aug 23 11:48:48 1994
Tue Aug 23 11:48:49 1994

```

Now we want to integrate this clock in our GUIEnv texteditor program, so we can always see the time in the low left corner of the window. We need to use the `'~beta/guienv/ v1.6/guienvsystemenv '`:

```

ORIGIN  '~beta/guienv/ v1.6/guienvsystemenv';
INCLUDE '~beta/guienv/ v1.6/fields'
        '~beta/guienv/ v1.6/stddialogs'
        '~beta/basiclib/ v1.6/file'
        '~beta/sysutils/ v1.6/time'
-- program: descriptor --
systemenv
(#
  setWindowEnv:: (* tell systemenv that myguienv is the
                  * the graphical user interface
                  *)
    (# do myguienv[] -> theWindowEnv[] #);

  updateClock: @|System
    (#
      do cycle
        (#
          do 1 -> sleep;
            systemtime -> formattime -> ... ;
              (* put time into the clock *)
        #);
    #);

  myguienv: @guienv (* inherit from guienv *)
    (# (* guienv code as before *)
    #);

do (* fork updateClock as a separate system *)
  updateClock[] -> fork;
#)

```

We need to specify to `systemenv` what graphical user interface system we are using. This is done by extending the virtual `setWindowEnv` like the following:

```

setWindowEnv::(# do myguienv[] -> theWindowEnv[] #); setWindowEnv

```

In order to do that, we have changed the `guienv` into a static object called `myguienv`. `myguienv` will automatically be started by `systemenv`.

Finally, we need to create a user interface element that can show the time. We use a `staticText`, that we position below the `TextEditor` field:

```

clock: @staticText staticText
  (# open::
    (# w,h: @integer;
      do systemtime -> formattime -> label;
        theWindow.size -> (w,h);
        (5,h-16) -> position; (50,15) -> size;
        True -> BindBottom; False -> BindTop;
    #);
  #);

```

The complete program is:

Complete code

```
ORIGIN '~beta/guienv/ v1.6/guienvsystemenv';
INCLUDE '~beta/guienv/ v1.6/fields'
        '~beta/guienv/ v1.6/stddialogs'
        '~beta/basiclib/ v1.6/file'
        '~beta/sysutils/ v1.6/time';
-- program: Descriptor --
systemEnv
(#
  setWindowEnv::< (# do myguienv[]->theWindowEnv[] #);
  updateClock: @|System
    (#
      do
        cycle
          (# theText: @StyledText;
            do l->sleep;
              systemtime->formattime-> myguienv.theWindow.clock.label;
            #);
          #);
  myguienv: @guienv (* inherit from guienv *)
    (# theWindow: @window (* make a window *)
      (# menubarType:: (* extend the menubar *)
        (# fileMenu: @menu (* make a file menu *)
          (# textFile: @file;
            openItem: @menuItem (* make an open item *)
              (# eventHandler::
                (* extend the virtual that is called *)
                (# onSelect::
                  (* this menu item is selected *)
                  (# theText: @StyledText;
                    do theWindow[]->
                      fileSelectionDialog->
                        textFile.name;
                      textFile.openRead;
                      textFile.scan
                        (# while::(#do true->value#);
                          do ch->theText.put
                            #);
                      theText[]->
                        theTextEditor.contents.
                          contents;
                      textFile.close;
                    #)#);
                open:: (# do 'Open'->name #);
              #);
            saveItem: @menuItem (* make a save item *)
              (# eventHandler::
                (* extend the virtual that is called *)
                (# onSelect::
                  (* this menu item is selected *)
                  (# theText: @Text;
                    do
                      textFile.openWrite;
                      theTextEditor.contents.
                        contents->
                          textFile.puttext;
                      textFile.close;
                    #)
                  #);
                open:: (# do 'Save'->name #);
              #);
            quitItem: @menuItem (* make a quit item *)
              (# eventHandler::
                (# onSelect:: (# do Terminate #) #);
                open:: (# do 'Quit'->name #);
              #);
```

```

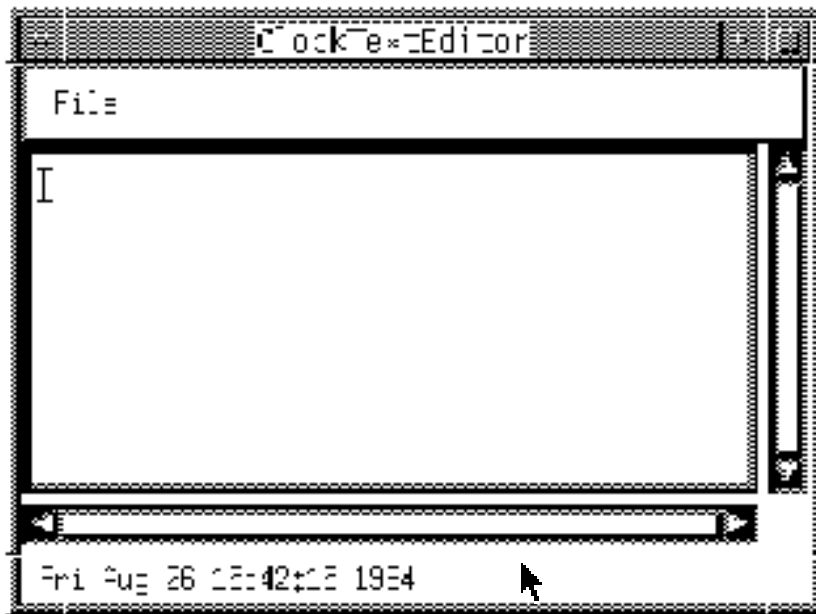
        open::
        (* extend the open virtual of filemenu
        * to open the items *)
        (#
        do 'File'->name;
        openItem.open;
        openItem[]->append;
        saveItem.open;
        saveItem[]->append;
        quitItem.open;
        quitItem[]->append;
        #)
    #);
    open::
    (* extend the open virtual of the menubar
    * to open the filemenu *)
    (# do fileMenu.open; fileMenu[]->append #);
    #);
thetextEditor: @textEditor (* our text editor *)
    (# open::
    (* extend the open virtual to set the size
    * and the placement *)
    (# w,h: @integer;
    do theWindow.size->(w,h);
    (w,h-20)->Size;
    True->bindBottom;
    True->bindRight
    #);
    #);
clock: @staticText
    (# open::
    (# w,h: @integer;
    do systemtime->formattime->label;
    theWindow.size->(w,h);
    (5,h-16)->position;
    (300,15)->size;
    True->BindBottom;
    False->BindTop;
    #);
    #);
    open::
    (* extend the window open virtual
    * to open the textEditor *)
    (# do thetextEditor.open; clock.open; #);
    #);
do theWindow.open;
    (* open the window when the application start up *)
    #)
do updateClock[]->fork;
#)

```

### Program 18.3: ClockTextEditor.bet

The following figure shows a snapshot of the program running on Motif:





### 18.3 Changes from the Original Design

The abstractions defined here are based on the ones described in chapter 12 of the BETA book. The implementation is identical to the design in the BETA book, except for the following changes:

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 #)`.

3. The syntax of `alt` is

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

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

This implementation of `systemenv` includes some new facilities, not described in the BETA book: **New facilities**

4. `semaphore` had an additional attribute `tryP`, which is a non-blocking call of.
5. In addition to `s[]->fork`, `s[]->kill` is possible, and in addition to `pause`, `100->sleep` is possible.
6. `system` has a new virtual attribute, `onKilled`, that is invoked before the system terminates
7. `systemenv` has a new virtual attribute, `deadlocked`, that is invoked if all processes are deadlocked.
8. 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.

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 **The Concurrency is Simulated**

makes an explicit or implicit `SUSPEND`. An implicit `SUSPEND` is made when a component must wait for a semaphore, or executes the `pause` and `sleep` patterns.

The `systemenv` libraries are thoroughly described in the manual [MIA 94-25]

**More information**

# 19 The Fragment System

Every BETA program uses the Fragment System. A fragment can be viewed as a piece of a BETA program – a module. Fragments are organized in files. A file may consist of one or more fragments.

The basic BETA environment, called `betaenv`, supplies basic BETA patterns, such as integer, char, boolean, and text. In order to use these basic patterns, the program must specify that the `betaenv` environment is to be used. The following example illustrates how:

```
ORIGIN '~beta/basiclib/ v1.6/betaenv'  
---program: descriptor---  
(#  
do 'Hello World!'->putline;  
#)
```

**Program 19.1: HelloWorld.bet**

The example consists of two parts, the specification of `ORIGIN` and the descriptor<sup>4</sup> called `program`.

The specification of `ORIGIN` tells that the program uses the fragment file `~beta/basiclib/ v1.6/betaenv`. The descriptor `program` tells that following the line

```
---program: descriptor---
```

**program slot**

comes a BETA descriptor, i.e. `(# ... #)`, that will be named `program`. The name is used to identify the descriptor for the purpose of binding it to an unbound hole in the `betaenv` environment. A simple `betaenv` environment could have the following outline:

```
(* The basic BETA environment betaenv *)  
(# ...  
  put: (# c: @char; enter c do ... #);  
  puttext: (# t: ^text; enter t[] do ... #);  
  putline: (# t: ^text; enter t[] do t[]->puttext; newline #);  
  newline: (# do ... #);  
  text: (# ... #);  
  ...  
  <<SLOT LIB: Attributes>>  
  ...  
do (* initialize for execution *)  
  <<SLOT program: descriptor>>  
  (* terminate execution *)  
#)
```

**betaenv**

The `betaenv` environment consists of a single descriptor with two holes—slots. One named `program` of type `descriptor` and one named `LIB` of type `Attributes`.

**program and LIB slots**

---

<sup>4</sup> `descriptor` is an alias for `ObjectDescriptor`, i.e. `--program:ObjectDescriptor--` is also legal.

The `program` slot is empty and can be *filled* (or *bound*) by a BETA program by defining an descriptor like:

```
---program: descriptor---
(# ... #)
```

as illustrated above. Every BETA program must have exactly *one* such construct in order to fill the empty slot in `betaenv`.

Filling a slot can be compared to a textual replacement<sup>5</sup>. The Hello World example program above, thus replaces the `program` slot in `betaenv`, resulting in the following expanded BETA program: **Filling slots**

```
(* The basic BETA environment betaenv *)
(# ...
  put: (# c: @char; enter c do ... #);
  puttext: (# t: ^text; enter t[] do ... #);
  putline: (# t: ^text; enter t[] do t[]->puttext; newline #);
  newline: (# do ... #);
  text: (# ... #);
  ...
do (* initialize for execution *)
  (#
  do 'Hello World!'->putline;
  #)
  (* terminate execution *)
#)
```

### Program 19.2: HelloWorld with filled program slot

The `LIB` slot can be used to define libraries that may be used in other BETA programs. If we want to add an operation called `putBoxed` to the basic environment, we can fill the `LIB` slot:

```
ORIGIN '~beta/basiclib/ v1.6/betaenv'
---LIB: Attributes---
  putBoxed:
    (* print the text with a box surrounding:
     * 'text'->putBoxed results in '[text]'
     *)
    (# t: ^Text;
    enter t[]
    do '['->put; t[]-> puttext; ']'->put;
    #);
```

**Defining a library**

### Program 19.3: putBoxed.bet

The `HelloWorld` program can then use this library by including it:

```
ORIGIN '~beta/basiclib/ v1.6/betaenv';
INCLUDE 'putBoxed'
---program: descriptor---
(#
do 'Hello World!'->putBoxed;
#)
```

**Using a library**

### Program 19.4: HelloWorld.bet

Resulting in the following output:

```
[Hello World!]
```

---

<sup>5</sup> Textual replacement is not exactly correct due to the scope rules. Please see the BETA book chapter 17 for a description of these rules.

The `HelloWorld` example program using the `putBoxed` library, results in the following expanded BETA program:

```
(* The basic BETA environment betaenv *)
(# ...
  put: (# c: @char; enter c do ... #);
  puttext: (# t: ^text; enter t[] do ... #);
  putline: (# t: ^text; enter t[] do t[]->puttext; newline #);
  newline: (# do ... #);
  text: (# ... #);
  putBoxed:
    (* print the text with a box surrounding:
      * 'text'->putBoxed results in '[text]'
      *)
    (# t: ^Text;
     enter t[]
     do '['->put; t[]-> puttext; ']'->put;
     #);
  ...
do (* initialize for execution *)
  (#
  do 'Hello World!'->putline;
  #)
  (* terminate execution *)
#)
```

#### Program 19.5: HelloWorldwith filled program and LIB slot

The program can be made even more simpler by having `ORIGIN` in the `putBoxed.bet` file:

```
ORIGIN 'putBoxed'
---program: descriptor---
(#
do 'Hello World!'->putBoxed;
#)
```

This works because the `putBoxed.bet` has `ORIGIN` in the `betaenv` environment, so the `HelloWorld` program will also have access to the `betaenv` environment.

The `---LIB: Attributes---` may be multiply specified in the same file or in different files. The way to make libraries in BETA is thus to define the pattern declarations in a fragment called `LIB`. The file containing the `---LIB: Attributes---` fragment can then be included in your program and the declarations can be used.

### 19.1 Interface and Implementation

The fragment system can be used to separate *interface* from *implementation*. In the `putBoxed` example above we included the implementation of the operation in the interface. We can move the implementation of `putBoxed` to another file using a `dopart` slot. This is specified as follows:

```
putBoxed:
  (* print the text with a box surrounding:
    * 'text'->putBoxed results in '[text]'
    *)
  (# t: ^Text;
   enter t[]
   <<SLOT putBoxed: dopart>>
   #);
```

**Interface**

Here we have described only the interface of `putBoxed`, i.e. it can be seen that the operation takes a text as a argument (and the comment states that the operation will print

the text with a surrounding box). The implementation is hidden. The implementation can be described in a `dopart` fragment:

```
---putBoxed: dopart---
do '['->put; t[]-> puttext; ']'->put;
```

**Implementation**

In order to make things work we must specify where the implementation can be found. This is done using `ABODY` specification in the `putBoxed.bet` file:

```
ORIGIN '~beta/basiclib/ v1.6/betaenv';
BODY 'putBoxedBody'
---LIB: Attributes---
putBoxed:
(* print the text with a box surrounding:
 * 'text'->putBoxed results in '[text]'
 *)
(# t: ^Text;
enter t[]
<<SLOT putBoxed: dopart>>
#);
```

### Program 19.6: putBoxed.bet

The file with the `---putBoxed: dopart---` fragment must specify where the `dopart` fragment is to be filled. This is done using the `ORIGIN`:

```
ORIGIN 'putBoxed'
---putBoxed: dopart---
do '['->put; t[]-> puttext; ']'->put;
```

### Program 19.7: putBoxedBody.bet

Another major advantage of separating the implementation from the interface is *separate compilation*. The `putBoxed.bet` and the `putBoxedBody.bet` file can be separately compiled, and the `putBoxedBody.bet` file can be changed and recompiled without recompiling the interface file `putBoxed.bet` or any of the programs that are using the library `putBoxed`.

**Separate compilation**

The `HelloWorld` program using the `putBoxed` library has not changed:

```
ORIGIN '~beta/basiclib/ v1.6/betaenv';
INCLUDE 'putBoxed'
---program: descriptor---
(#
do 'Hello World!'->putBoxed;
#)
```

### Program 19.8: HelloWorld.bet

And the expanded BETA program, using the files: `betaenv`, `HelloWorld`, `putBoxed`, and `putBoxedBody` is (exactly as above):

```
(* The basic BETA environment betaenv *)
(# ...
put: (# c: @char; enter c do ... #);
puttext: (# t: ^text; enter t[] do ... #);
putline: (# t: ^text; enter t[] do t[]->puttext; newline #);
newline: (# do ... #);
text: (# ... #);
putBoxed:
(* print the text with a box surrounding:
 * 'text'->putBoxed results in '[text]'
 *)
(# t: ^Text;
```

```
enter t[]
do '['->put; t[]-> puttext; ']'->put;
#);
...
do (* initialize for execution *)
(#
do 'Hello World!'->putline;
#)
(* terminate execution *)
#)
```

**Program 19.9: HelloWorldwith filled program and LIB slot**

The fragment system is described in abstract terms in the BETA book [Madsen 93]. **More information** That description also suggests many ideas of how to use the fragment system. The current implementation of the fragment system is described in the compiler manual [MIA 90-2].

# References

- [Knudsen 94] J. L. Knudsen, M. Löfgren, O. L. Madsen, B. Magnusson (eds.): *Object-Oriented Environments – The Mjølner Approach*, Prentice Hall, 1994, ISBN 0-13-009291-6.
- [Madsen 93] O. L. Madsen, B. Møller-Pedersen, K. Nygaard: *Object-Oriented Programming in the BETA Programming Language*, Addison-Wesley, 1993, ISBN 0-201-62430-3
- [MIA 90-2] Mjølner Informatics: *The Mjølner System: BETA Compiler Reference Manual* Mjølner Infomatics Report MIA 90-2.
- [MIA 90-4] Mjølner Informatics: *The Mjølner System: Using BETA on UNIX Systems*, Mjølner Informatics Report MIA 90-4.
- [MIA 90-6] Mjølner Informatics: *The Mjølner System: Using BETA on the Macintosh*, Mjølner Informatics Report MIA 90-6.
- [MIA 90-8] Mjølner Informatics: *The Mjølner System: Basic Libraries, Reference Manual*, Mjølner Infomatics Report MIA 90-8
- [MIA 91-16] Mjølner Informatics: *The Mjølner System—X Window System Libraries*, Mjølner port MIA 91-16.
- [MIA 91-20] Mjølner Informatics: *The Mjølner System – Persistent Store*, Mjølner
- [MIA 92-22] Mjølner Informatics: *The Mjølner System – Container Libraries, Reference Manual*, Mjølner Report MIA 92-22.
- [MIA 94-25] Mjølner Informatics: *The Mjølner System – Distribution* Mjølner
- [MIA 94-26] Mjølner Informatics: *The Mjølner System – BETA Language Introduction* Mjølner 94-26.
- [MIA 94-27] Mjølner Informatics: *The Mjølner System – GUIEnv Librarues* Mjølner





# Index

---

## A

alt 68

---

## B

betaenv.....1; 70  
betaenv.....2  
BODY.....73  
BooleanObject.....26  
Booleans.....6

---

## C

Characters.....6  
CharObject.....26  
Command-line arguments.....32  
Comparing texts.....38  
Complex evaluations.....16  
conc.....68  
Concurrency.....20  
Concurrent.....62  
Concurrent coroutines.....62  
Constants.....6; 17  
container.....42  
Control Characters.....6  
coroutine.....62  
Create and execute.....24  
Create and return a reference.....25  
cString.....49  
cStruct.....49

---

## D

Declarations.....1  
Declaring a coroutine.....62  
Defining a library.....71  
Descriptor.....20  
Directory.....36  
Directory attributes.....36  
Dynamic reference to static object.....24  
Dynamic references.....23

---

## E

Encapsulation.....22  
*evaluation*.....6  
Evaluations.....16  
Exception.....46  
external pattern.....49  
externalRecord.....49

---

---

## F

File.....31  
File attributes.....31  
File Exception.....32  
File menu.....55  
Filling slots.....71  
for9  
for imperative.....9  
fork and not S.fork.....68  
Formatted Input and Output.....40  
Formatted output.....12  
*fragment system*.....22; 70  
Function results.....15

---

## G

Graphical User Interface.....54  
GUIEnv.....54

---

## H

hashtable.....43

---

## I

if 11  
if imperative.....9  
Implementation.....72  
INCLUDE.....71  
INCLUDE.....4  
Inheritance.....20  
Integer.....6  
IntegerObject.....26  
Interface.....72

---

## L

labeled imperative.....14  
LIB slots.....70  
List.....42  
Local function.....22  
Local state.....17

---

## M

Macintosh.....54  
Making a library.....52  
Math Library.....6  
Monitor.....62

MotifEnv.....	59; 60
Multidimensional Repetitions.....	29

---

## N

New operator.....	24
Notification.....	46
numberio.....	4

---

## O

Object reference repetitions.....	27
<i>object-descriptor</i> .....	20
Object-oriented concepts.....	20
OBJFILE.....	50
Operations on a persistent store.....	52
operations on List.....	42
Operator priority.....	6
ORIGIN.....	70
ORIGIN.....	4
OSF/Motif.....	59

---

## P

Parameter list.....	10
Pattern .....	8
<i>pattern declaration</i> .....	21
persistence .....	52
Primitive Types.....	26
Procedure.....	8; 17
Procedure arguments.....	11
program slot.....	70

---

## Q

Quick sort.....	27
-----------------	----

---

## R

RealObject.....	26
Reals.....	6
recursion .....	28
Reference operator.....	23
repetitions .....	27
restart.....	14

---

## S

Screen Output.....	2; 4
semaphore.....	62; 69
Separate compilation.....	73
Simple types.....	3
Standard file open .....	56
Statements.....	1
Static and dynamic references.....	23
Static Semantic Errors.....	4
Static variable declarations.....	21
Static Variables.....	4
statically nested.....	28
Stop.....	37
Structural equivalence.....	21
Structured data.....	21
System .....	62
systemenv.....	62

---

## T

Text .....	38
Text .....	6
Text operations.....	40
TextEditor.....	55
type.....	17
Type Compatibility.....	4; 7

---

## U

Using a library.....	71
----------------------	----

---

## V

Vaucher.....	i
--------------	---

---

## W

While loop.....	14
Widgets.....	59
Window.....	55
Windows.....	54

---

## X

X Window System.....	54
Xt Toolkits.....	59