The Mjølner BETA System
BETA Compiler

Reference Manual

Mjølner Informatics Report

MIA 90-02(1.5)

August 1996

Table of Contents

1 Introduction

This manual describes version 5.2 of the BETA compiler (corresponding to release 4.0 of the Mjølner BETA System). The compiler implements most parts of the BETA language as described in [Madsen93]. There are, however, some implementation restrictions.

The user should read section 5 for a description of the implementation restrictions and deviations from [Madsen93]

See Appendix C for an overview of new features in v5.2 of the compiler. The BETA compiler is accompanied by a large collection of libraries and application frameworks. This includes a text concept, and libraries for input/output on keyboard, screen and files, a user interface package, a library of well-known datastructures, and a metaprogramming system. The Mjølner BETA System is available for Macintosh (at least 68020 CPU, 8 Mbyte RAM, MPW 3.2 or later), UNIX workstations such as Sun-4 (SPARC running SunOS or Solaris), HP-9000 series 400, 700, Silicon Graphics running IRIX 5.3, and PC's running Windows NT, Windows 95 or Linux.

On Macintosh the user interface system is implemented on top of the Macintosh Toolbox. For Macintosh there is also a library that interfaces directly to the Toolbox.

On UNIX, the user interface system is implemented on top of the X Window System (X11R3 or later). A number of UNIX facilities can be accessed via a UNIX library.

On Windows 95 and Windows NT, the user interface system is implemented on top of WIN32.

A general interface to C and assembly language is part of the libraries/compiler.

The rest of this manual is organized as follows: Section 2 describes the simplest way of using the compiler. Section 3 describes the organization of the basic BETA libraries. Section 4 describes the files generated by the compiler. Section 5 describes various deviations in the implementation of BETA. Section 6 describes the implementation of the fragment system. Section 7 describes compile- and run-time errors. These sections contain useful information for all users.

The remaining sections are only for advanced users. In section 8, a number of different arguments to the compiler are described. In section 9, it is described how to instantiate machine dependent configurations of a program. In section 10 it is described how code is generated for multiple machines.

2 Simple Use of the Compiler

The following is an example of a very small BETA program.

A BETA program

```
ORIGIN '~beta/basiclib/v1.5/betaenv'
--- PROGRAM: descriptor ---
(#
do 'Welcome to Mjolner' -> putLine
#)
```

Only the part between `(# ... #)` is BETA. The ORIGIN specification:

ORIGIN

```
ORIGIN '~beta/basiclib/v1.5/betaenv'
```

describes that version 1.5[1] of the fragment `betaenv` from the BETA basic library (`basiclib`) is used.

The fragment name and category:

Program slot

```
--- PROGRAM: descriptor ---
```

describes that the BETA program is filled into a slot in `betaenv` called `PROGRAM`. The BETA compiler is integrated with the Mjølner BETA fragment system. The above BETA program is an example of a BETA fragment.

Assume that the above BETA fragment is located in the file `foo.bet`. The BETA fragment may then be compiled by issuing the command

Compiling

```
beta foo.bet
```

which will compile, assemble[2] and link the BETA fragment. The final object code will be in the file `foo`, which may be executed.

More information

How to invoke the compiler depends on whether Macintosh, PC or UNIX is used. Details about the different variants of the BETA compiler may be found in [MIA90-6] for Macintosh, [MIA94-32] for Windows, and in [MIA90-4] for UNIX.

[1] The actual version to be used depends on the current release installed at the available hardware.

[2] On some platforms, binary code is generated directly. In this case, the assembly phase is omitted.

3 The BETA Library

The BETA library is a collection of patterns and objects that include input/output, a text concept, the user interface toolkit, the metaprogramming system, a container library, a system library, etc. The library is organized as fragments.

basiclib

One part of the library contains the basic patterns and objects which are used by most programs. This basic BETA library is called `basiclib` and is described in [MIA90-8], which also describes the interface to C and assembly language.

The library `basiclib` contains a number of different fragments groups containing basic patterns, a text concept, various functions and control patterns, a file concept, etc. One of these fragment groups is `betaenv`, which contains the basic patterns, the text concept, other basic patterns and objects representing the screen and the keyboard. All BETA programs must use `betaenv`, which has the form:

betaenv

```
(# ...
   (* A lot of useful patterns *)
   ...
   <<SLOT LIB: attributes>>
   ...
   program: <<SLOT program: descriptor>>
   theProgram: ^|program;

do ...
   &|program[] -> theProgram[];
   theProgram;
   ...
#)
```

The `LIB` slot describes where most libraries are inserted. The `program` slot describes where an ordinary user program is inserted (see section 6 for more explanation of this).

Location of libraries

On UNIX, the BETA library is often located in the directory `/usr/local/lib/beta`.

For Macintosh, the convention is that the BETA library is located in a folder called **`beta`**.

In the rest of this manual, we assume that the basic library is located in `/usr/local/lib/beta`. We also use the UNIX convention for denoting directories with the character `/` to separate directory and file names.

The Mjølner BETA System contains directories for the various libraries. The basic library `basiclib` is e.g. located in the directory:

```
/usr/local/lib/beta/basiclib
```

The directory for a library contains directories corresponding to different versions of the library. Version `1.5` of `betaenv` is contained in the directory

```
/usr/local/lib/beta/basiclib/v1.5
```

This directory contains the fragment groups constituting `basiclib`. Instead of referring to a specific version, it is possible to refer to the current official version by means of the name `current`. (This is not possible on Windows and Macintosh).

The Mjølner BETA System accepts the following abbreviation for the BETA library:

~beta

`~beta` denotes `/usr/local/lib/beta`

The meaning of `~beta` can be changed by using the `BETALIB` environment variable, see [MIA 90-04].

To sum up, the file containing the current version of `betaenv` may be referred to by:

```
~beta/basiclib/current/betaenv
```

A user-program using `betaenv` may then look as follows:

```
ORIGIN '~beta/basiclib/current/betaenv'
--- PROGRAM: descriptor ---
(#
do 'Welcome to Mjolner' -> PutLine
#)
```

Please note, that on Windows and Macintosh the separator in `ORIGIN` specifications is also `/`. See section 6.2.

Assume that the above program resides on the file `foo.bet`. The program may then be compiled by issuing the command:

```
beta foo.bet
```

The file `foo` will now contain an executable version of `foo.bet`.

When developing the program, it may be an advantage to invoke the compiler as

```
beta -r foo.bet
```

Repeating mode

This will run the compiler in repeating mode. After having translated the fragments specified in the argument list, if in repeating mode, the compiler prompts the user for the name of another fragment to be translated. Hitting `<RETURN>` in this case will recompiler the program last compiled. See section 8 for a survey of the legal command line options.[3]

More information

Please consult the BETA tutorial [MIA 94-24] for a quick survey of the BETA language and the basic libraries.

[3] This is currently not possible on Windows and Macintosh.

4 Files Generated by the Compiler

For each fragment file, a number of other files may be produced by the compiler; let `foo.bet` be a BETA fragment. Then

- **List files** `foo.lst` contains information about possible syntactic and static semantic errors. If such errors occur, then the file contains a pretty-print of the fragment with an indication of the error(s). See section 7 for further information about error handling. Possible semantic error messages are listed in appendix A.
- **Abstract syntax tree files** `foo.ast` or `foo.astL` contains the abstract syntax tree representation of the compiled source code for big-endian and little-endian architectures, respectively. The AST files are used by many tools in the Mjølner BETA System.
- **Assembler files** `foo..s` contains the generated assembly code for the compiled source code[4]. Assembly files are located in subdirectories named according to the machine type, to which the source code has been compiled. Currently, the directories `sun4`, `sun4s`, `hpux9mc`, `hpux9pa`, `nti`, `linux`, sgi, and `mac` can be created. These directories are automatically created by the compiler, if not present already. The assembly file is usually deleted by the compiler after assembly.
- **Object files** `foo.o` contains the object code generated by the assembler. Like `foo..s`, this file is placed in a subdirectory.
- **Debug files** `foo..db` contains information used by the debugger Valhalla when debugging the `foo` fragment. See [MIA 92-12]. Like `foo..s`, this file is placed in a subdirectory.
- `foo..gs` and `foo..go` are generated instead of `foo..s` and `foo..o` if `foo` is compiled with debug info on. This is a temporary solution and these files will not be generated in a future release of the compiler.

The above list of files is generated for each fragment group that is included in a program. In addition, the following two files are generated for each program:

- **Executable** `foo` containing the executable code for the program.
- **Job file** `foo..job` containing directives for assembly and linking. Like `foo..s`, this file is placed in a subdirectory. This file is usually deleted by the compiler after linking.

For some implementations (e.g. Windows NT) other extensions than `..s` and `.o` may be used.

[4] On some platforms, binary machine code is generated directly. In this case, no assembly file is generated.

5 Implementation Deviations

5.1 The BETA Book

The BETA book

The BETA language is described thoroughly in [Madsen93].

It is prerequisite to be familiar with [Madsen93] in order to use the Mjølner BETA System.

This book is currently the only definition of the BETA language, but a precise language definition is being worked on.

A short introduction to BETA and the Mjølner BETA System may also be found in the Mjølner BETA Tutorial [MIA 94-24] and in [Knudsen94].

The BETA grammar is given in appendix B.

There are a few of deviations from [Madsen93] in the current implementation of BETA. These deviations are described below.

5.2 Restrictions

Restrictions

1. The `integer` operations `+`, `-`, `*`, `div`, `mod`, `=`, <>, etc. will work on 32 bits.
2. Assignment between instances of `integer`, and `real` is allowed. In assignments of reals to integers the values are truncated.

   Assignment between instances of `integer` and `char` is allowed. Character constants have their ASCII char value. Assignment of an arbitrary `integer` value to `char` instances may thus give meaningless results.

   Assignment between instances of `integer` and `boolean` is allowed, but will give a warning. In a future release these assignments will not be allowed and will give an error. The patterns `true` and `false` have the values `1` and `0` respectively. Assignment of an arbitrary `integer` value to boolean instances may thus give meaningless results.

   The following table shows legal combinations of operands and the result type.

   Entries not shown are illegal. Entries marked with ***** are illegal. Entries marked with **!** will give a warning, and will become illegal in a future release.

Abbreviations:

int	means	**integer**
bool	means	**boolean**
iref	means	**item reference**
cref	means	**component reference**
sref	means	**structure reference**

NONE is both an **iref**, a **cref** and an **sref**.

Assignment Compatibility

For assignment and binary operators, the rows and columns of the tables show left and right operands respectively, and the elements of the tables show the result type.

Assignment: ->

	int	**char**	**real**	**bool**	**iref**	**cref**	**sref**
int	**int**	**char**	**real**	**!**	*	*	*
char	**int**	**char**	*	*	*	*	*
real	**int**	*	**real**	*	*	*	*
bool	**!**	*	*	**bool**	*	*	*
iref	*	*	*	*	**iref**	*	*
cref	*	*	*	*	*	**cref**	*
sref	*	*	*	*	*	*	**sref**

3. The relational operators `=`, `<>`, `<`, etc. do only work for the basic patterns `integer`, `real`, `boolean`, and `char` and for references (only `=`, `<>`) I.e. `E1 = E2`, where `E1` and `E2` are instances of some user-defined pattern will not work.

Comparison Operator Compatibility

The following tables show legal combinations of operands and the result type. The notation is explained in item 2 above.

Binary operators: = , <>

	int	**char**	**real**	**bool**	**iref**	**cref**	**sref**
int	**bool**	**bool**	**bool**	**!**	*	*	*
char	**bool**	**bool**	**bool**	*	*	*	*
real	**bool**	**bool**	**bool**	*	*	*	*
bool	**!**	*	*	**bool**	*	*	*
iref	*	*	*	*	**bool**	*	*
cref	*	*	*	*	*	**bool**	*
sref	*	*	*	*	*	*	**bool**

Binary operators: <, <=, >, >=

	int	**char**	**real**	**bool**	**sref**
int	**bool**	**bool**	**bool**	*	*
char	**bool**	**bool**	**bool**	*	*
real	**bool**	**bool**	**bool**	*	*
bool	*	*	*	**bool**	*
sref	*	*	*	*	**bool**

Arithmetic/ Logical Operator Compatibility

4. Arithmetic/logical operators.

The following tables show legal combinations of operands and the result type. The notation is explained in item 2 above.

Binary operators: +, -, *, div

	int	**char**	**real**
int	**int**	**int**	**real**
char	**int**	**int**	*
real	**real**	*	**real**

Binary operator: mod

	int	**char**
int	**int**	**int**
char	**int**	**int**

Binary operator: /

	int	**char**	**real**
int	**real**	**real**	**real**
char	**real**	**real**	*
real	**real**	*	**real**

char is likely to be eliminated as a legal operand for **/** in a future version.

Binary operators: and, or, xor

	bool
bool	**bool**

Unary operators: + -

int, **char**, **real** result type is the same as operand type

Unary operator: not

bool result type is **bool**

5. In `if`-imperatives

   ```
   (if E0 // E1 then ... // E2 then ... if)
   ```

   the exit-lists of `E0`, `E1`, `E2`, ... must consist of exactly one `integer`, `real`, `char`, `boolean` or reference.

6. Inserted items, i.e.,

   ```
   do ...; P; ...
   ```

   ([Madsen93], section 5.10.2) are implemented as dynamic items (`&P`). However, the user is urged to use dynamic items for recursion in order to ensure compatability with future releases.

7. Inserted components, i.e.,

   ```
   do ...; |(# ... #); ...
   ```

   ([Madsen93], section 5.10.3) have not been implemented.

8. Virtual superpatterns, i.e.,

   ```
   A::< (# ... #); (* Where A is some virtual *)
   B: A(# ... #)
   ```

   have not been implemented.

   By using a *final binding*, this problem may often be overcome like this:

   ```
   A:: (# ... #); (* A is no longer virtual *)
   B: A(# ... #)
   ```

   The situation may also occur in a more indirect way:

   ```
   graph:
     (# node:< (# ... #);
        nodeList: @list(# element::< node #);
        ...
     #);
   ```

   Here the virtual further binding of `element` in `list` is not allowed, since `node` is itself virtual.

   The current version of the compiler will allow final binding using a pattern that is itself virtual. That is, you can do this:

   ```
   graph:
     (# node:< (# ... #);
        nodeList: @list(# element:: node #);
        ...
     #);
   ```

   General virtual prefixes behave much like multiple inheritance and will not be implemented in the near future.

9. The labelled compound imperative

   ```
   A: (L: imp1; imp2; ...; impN :L)
   ```

   has been eliminated from the language. Instead the following construct may be used:

   ```
   A: (# do imp1; imp2; ... ; impN; #)
   ```

   Inserted items with no declarations and no superpattern will be inlined in the enclosing code. There will thus be no execution overhead compared to the old (never implemented) labelled compound imperative statement.

10. Consider the following example:

```
A: (# X: ^P; (* reference to item qualified by P *)
      B: ^|P (* reference to component qualified by P *)
   do ...
      this(P)[] -> X[];   (* legal use of this(P)[] *)
      this(P)[] -> R[];   (* illegal use of this(P)[] *)
   #)
```

The illegal use is due to the fact that `this(p)[]` is considered a reference to an item object and not a component object.

11. In declarations like:

```
P: <AD>(# ... #);
X: @<AD>;
Y: ^<AD>;
```

it is checked that `<AD>` is a *static* denotation, where *static* is defined as follows:

- A name `A` is always static
- In a remote-name `R.A`, `R` must be a static object
- Use of `THIS(A).T` is static
- Only in `Y: ^P.T`, can `P` be a pattern
- Denotations using `R[e]`, and `(foo).bar` are *not* static

This means that e.g. descriptors like:

```
R[e].A(# ... #)
(foo).bar(# ... #)
R.P(# ... #) where 'R' is a dynamic ref.
```

are only allowed in *imperatives*.

For `Y: ^R.P` where `R` is a dynamic reference, the compiler will currently report a warning and suggest to use

`Y: ^A.P` where `A` is the qualification of R.

Note: that when using `--noWarnQua`, this warning will *not* be printed. A future release may change the warning to an error.

12. There are some deviations with respect to the implementation of concurrency. Please consult [MIA90-8] before using the concurrency.

13. It is in general not possible to use `leave P` or `restart P` where `P` is a pattern. `P` must in general be a label. *However*, the following has been implemented:

```
P: (#
   do
   ...
   leave P;
   ...
   restart P;
   ...
   #)
```

Leave/restart from an inserted item, however, is *not* supported by the current version of the compiler:

```
P: (#
   do
   ...
      (#
      do
         ...
         leave P; (* ILLEGAL *)
      ...
         restart P; (* ILLEGAL *)
         ...
      #)
   ...
   #)
```

14. A pattern where the object descriptor is described as a slot cannot be used as a super-pattern. I.e. the following is illegal:

    ```
    A: <<SLOT Pdesc: descriptor>>;
    B: P(# ... #);  (* illegal *)
    ```

    Instead the following can often be used:

    ```
    C: (# do <<SLOT Pdesc: descriptor>> #)
    D: P(# ... #); (* legal *)
    ```

15. The `Program` pattern as described in the chapter on exception handling in [Madsen93] has not been implemented.
16. There are some restrictions on the use of fragments as described in section 6 below.

5.3 Extensions

5.3.1String Literals as References

Extensions

The pattern `Text` enters and exits a char-repetition. This means, that a text may be initialized using constant strings as follows:

```
   t: @text;
do 'hello' -> t;
```

Many operations involving texts, however, takes *references* to texts as enter/exit parameters. This is mainly for efficiency reasons.

To allow easy invocation of such operations on string literals, the following is *also* allowed:

```
   t: ^text;
do 'hello' -> t[];
```

The semantics of this is, that a text object is instantiated, initialized by the constant string, and finally assigned to the text reference.

5.3.2 Simple If

Often the following If statement is used:

```
b: @boolean;
do (if b//TRUE
    then ...
    else ...
   if);
```

The current version of the compiler supports an extension to the BETA language called Simple If. This extension means, that the case-selector `//` may be omitted, if the evaluation on the left hand side exits a boolean. That is, the above may be written

```
b: @boolean;
do (if b
    then ...
    else ...
   if);
```

Like in the general `if`-statement, the `else` part if optional.

5.3.3 Xor Primitive

An `xor` primitive is supported as a basic operation on booleans. That is

```
b1, b2, b3: @boolean
do b1 xor b2 -> b3
```

is possible.

5.4.4Short-circuit Boolean Expressions

Boolean expressions are implemented as short-circuit.

That is, in

`B1 or B2`	`B2` is *not* evaluated if `B1` is true
`B1 and B2`	`B2` is *not* evaluated if `B1` is false

6 The Fragment System

The Mjølner BETA System is based on the notion of fragment. The fragment system must be used for splitting a large program into smaller units (fragments). The fragment system is used to support modularization, separation of interface and implementation parts, variant control and separate compilation. It is highly recommended to use the fragment system, since this may improve the structure of the program.

The principles of the fragment system are described in [Madsen93]. In the following it is assumed that the reader is familiar with this description.

The description in [Madsen93] is slightly more idealized than the actual implementation in the Mjølner BETA System:

In [Madsen93], the syntax of the fragment language is given in terms of diagrams. The fragment language implemented by the Mjølner BETA System has a textual syntax.

Implemented categories

In the Mjølner BETA System, slots have only been implemented for the syntactic categories `<<DoPart>>`, `<<objectDescriptor>>` and `<<attributes>>`.

Attributes restrictions

A fragment form of the category `<<attributes>>`, may only contain pattern declarations. It cannot contain any other kind of declarations, including virtual pattern declarations, virtual pattern bindings, static or dynamic declarations.

The alias `descriptor` can be used instead of `objectDescriptor`.

In the rest of this section, details of the Mjølner BETA System implementation of fragments are given.

Fragment group

In the current system, fragments are organized in groups. A group is stored as a file. The BETA compiler accepts a BETA program in the form of one or more files. Each file must contain a group of fragments (i.e. one or more fragments).

6.1 Fragment Language Syntax

In the following some of the examples of fragments from [Madsen93] will be given followed by the syntax used by the Mjølner BETA System. The first example shows the simplest possible BETA fragment-group:

Graphical syntax

```
NAME 'mini1'
ORIGIN 'betaenv'
PROGRAM: descriptor
(#
do 'Hello world!' -> PutLine
#)
```

The fragment-group is stored in the file `mini1.bet`, which is also the name of the fragment-group. The following syntax is is used by the Mjølner BETA System:

Textual syntax

```
ORIGIN '~beta/basiclib/v1.5/betaenv'
-- program: descriptor --
(#
do 'Hello world!'->PutLine
#)
```

The origin `betaenv` has been expanded into a complete file name for `betaenv`.

The next example is an example defining a library fragment:

Library

```
NAME 'mylib'
ORIGIN 'betaenv'
LIB: attributes
Hello: (# do 'Hello' -> PutText #);
World: (# do 'World' -> PutText #)
```

This fragment is stored in a file `mylib.bet` and the corresponding syntax in the Mjølner BETA System is:

```
ORIGIN '~beta/basiclib/v1.5/betaenv'
-- LIB: attributes --
Hello: (# do 'Hello' -> PutText #);
World: (# do 'World' -> PutText #)
```

The following fragments is an example of a fragment including the above defined library:

Using the library

```
NAME 'mini2'
ORIGIN 'betaenv'
INCLUDE 'mylib'
PROGRAM: descriptor
(#
do Hello; World; newLine
#)
```

This fragment is stored in a file `mini2.bet` and has the following syntax:

```
ORIGIN '~beta/basiclib/v1.5/betaenv';
INCLUDE 'mylib';
-- program: descriptor --
(#
do Hello; World; newLine
#)
```

The following example shows a fragment with a body:

Body

```
NAME 'textlib'
ORIGIN 'betaenv'
INCLUDE 'mylib'
LIB: attributes
SpreadText:
   {A blank is inserted between all chars in the text 'T'}
   (# T: @text
   enter T
   <<SLOT SpreadText:DoPart>>
   exit T
   #);
BreakIntoLines:
   {'T' refers to a text which is to be split into lines.}
   {'w' is the width of the lines.}
   (# T: ^ Text; w: @ Integer
   enter(T[],w)
   <<SLOT BreakIntoLines: DoPart>>
   #)
```

It is stored in a file `textlib.bet` and has the following syntax:

```
ORIGIN '~beta/basiclib/v1.5/betaenv';
BODY 'textlibbody';
---LIB: attributes---
SpreadText:
  (* A blank is inserted between all chars in the text 'T' *)
  (# T: @text
  enter T
  <<SLOT SpreadText: DoPart>>
  exit T
  #);
BreakIntoLines:
  (* 'T' refers to the text to be split into lines. *)
  (* 'w' is the width of the lines. *)
  (# T: ^ Text; w: @ Integer
  enter(T[],w)
  <<SLOT BreakIntoLines: DoPart>>
  #)
```

The body of `textlib` is shown in the next example:

```
NAME 'textlibbody'
ORIGIN 'textlib'
SpreadText: DoPart
do (# L: @integer
   do (for i: (T.length->L)-1 repeat
           (' ',L-i+1) -> T.InsertCh
      for)
   #)
BreakIntoLines: DoPart
do T.scan
   (# sepInx,i,l: @integer;
   do i+1->i; l+1->l;
      (if (ch<=' ' ) then i->sepInx if);
      (if l=w then
          (nl,sepInx)->T.InxPut;
          i-sepInx->l
      if);
   #);
   T.newline;
```

This fragment is stored in a file `textlibbody.bet`. The corresponding syntax is:

```
ORIGIN 'textlib'
-- Spreadtext: DoPart --
do (# L: @Integer
   do ...
   #)
--BreakIntoLines: DoPart --
do ...
```

Notice, that when local variables are needed in a `DoPart` slot, it may be necessary to make an inserted item in the `DoPart`. Alternatively a `Private` descriptor slot may be declared in the interface, and the `L` attribute moved to the `Private` fragment, which should then be placed in `textlibbody.bet` too.

General fragment file structure

Finally a general outline of a fragment group with several include, body and fragments is shown in the next example:

`NAME F`
`ORIGIN G`
`INCLUDE A1`
`INCLUDE A2`
`...`
`INCLUDE Am`
`BODY B1`
`BODY B2`
`...`
`BODY Bk`
`F1: S1`
`ff1`
`F2: S2`
`ff2`
`...`
`Fn: Sn`
`ffn`

This fragment group is stored in a file `F.bet` and the syntax is:

```
ORIGIN 'G';
INCLUDE 'A1' 'A2'... 'Am;
BODY 'B1' 'B2' ...  'Bk';
Prop1; Prop2; ... Propl
-- F1: S1 --
ff1
-- F2: S2 --
ff2
...
-- Fn: Sn --
ffn
```

`Prop1`, `Prop2`, ..., `Propl` are *properties* that may be defined for a fragment. Formally the `ORIGIN`, `INCLUDE`, and `BODY` parts are also properties. In section 6.3 a list of possible properties is given.

6.2 Fragment Denotations

In the examples above, terms like

```
INCLUDE '~beta/basiclib/v1.5/betaenv'
```

were used. Below we will use the term `FragmentDenotation` for the “fragment path” given in, e.g., the `INCLUDE` property. The other properties, that accept `FragmentDenotations` as arguments are explained in section 6.3.

Fragment denotation

Notice that a `FragmentDenotation` is *not* the same as a file name, although it resembles a UNIX file path, and although it normally corresponds directly to a (set of) file(s):

1. The separator in the `FragmentDenotation` is always the ‘`/`’ character, e.g., also for BETA programs on the Macintosh, where ‘`:`’ is used for *file* paths.

   ‘/ ‘ separator

2. As explained in section 3, the notation ‘`~beta`’ is legal in `FragmentDenotations` on all platforms, and simply means “the place BETA is installed”. As mentioned, the meaning of ‘`~beta`’ can be controlled by using the `BETALIB` environment variable, please consult [MIA 90-04], [MIA 94-34], and [MIA 90-06] for details.

   ~beta

3. The notation ‘`.`’ means ‘current directory/folder’ on all platforms, and the notation ‘`..`’ means ‘father directory/folder’, i.e. the directory containing a given directory.
4. It is not allowed to specify an extension (e.g. ‘`.bet`’ or ‘`.ast`’) in a `FragmentDenotation`.

There are some restrictions in the legal fragment *file* names, which also apply to the `FragmentDenotations`, please see section 6.6.

6.3 Fragment Properties

The fragment system allows arbitrary properties to be associated with fragments. The BETA compiler recognizes the following properties: For most users, only `ORIGIN`, `INCLUDE`, and `BODY` are relevant.

`ORIGIN` *`<TextConst>`*

The origin of a fragment is a fragment which is used when binding fragment-forms to slots.

`INCLUDE` *`<StringList>`*

Specifies one or more fragments that are always included when using this fragment.

`BODY` *`<StringList>`*

Specifies one or more fragments that fills the slots in this fragment file, but are not visible.

`MDBODY` *`<MachineSpecificationList>`*

Specifies one or more machine dependent fragments that fills the slots in this fragment file dependent on the machine type. See section 9 for further description.

`OBJFILE` *`<MachineSpecificationList>`*

The object file is included in the linker-directive. This is typically an External library which is interfaced to via the External interface described in [MIA90-8]. See also section 9.

`BETARUN` *`<MachineSpecificationList>`*

The standard BETA run-time system is replaced with the one in the object-file. See also section 9.

`MAKE` *`<MachineSpecificationList>`*

Specifies one or more makefiles to be executed before linking. See also section 9. The Makefile is executed relative to the directory, where the file containing the `MAKE` property is placed.

`RESOURCE` *`<MachineSpecificationList>`*

Specifies one or more resource files to be included in the applicaiton. Only used on Macintosh and Windows NT platforms. See also section 9.

`LIBFILE` *`<MachineSpecificationList>`*

Is similar to `OBJFILE`, but specifies inclusion of a library. See also section 9.

`LINKOPT` *`<MachineSpecificationList>`*

Machine dependent options to append to link directive for programs using the fragment. Only used on UNIX platforms. See also section 9.

`ON` *`n1 n2 ... nk`*

The compiler switches n1 n2 ... nk (positive numbers) are set. See also section 8.

`OFF` *`n1 n2 ... nk`*

The compiler switches n1 n2 ... nk (positive numbers) are cleared. See also section 8.

The terms *`<MachineSpecificationList>`*, *`<StringList>`*, and *`<TextConst>`* are syntactically explained in section 6.5.

6.4 Modularization of Data Structures

This section gives some advices that can be used when modularizing data structures. Consider the following program library (`stack.bet`):

```
ORIGIN '~beta/basiclib/v1.5/betaenv'
--- Lib: attributes ---
stack:
 (# element:< object;
    A: [100] ^element;
    top: @integer;
    push:
      (# e: ^element;
      enter e[]
      do top+1->top;
         e[] -> A[top][];
      #);
    pop:
      (# e: ^element;
      do A[top][] -> e[];
         top-1->top;
      exit e[]
      #);
    top:
      (# e: ^element;
      do A[top][]->e[];
      exit e[]
      #);
 #)
```

If we want to separate the interface and the implementation, this can be modularized in the following way:

Separating the interface

Introduce the following SLOTs:

```
ORIGIN '~beta/basiclib/v1.5/betaenv';
BODY 'stackImpl'
--- Lib: attributes ---
stack:
 (# element:< object;
    private: @<<SLOT private: descriptor>>;
    push:
     (# e: ^element;
     enter e[]
     <<SLOT pushBody: DoPart>>
     #);
    pop:
     (# e: ^element;
     <<SLOT popBody: DoPart>>
     exit e[]
     #);
    top:
     (# e: ^element;
     <<SLOT topBody: DoPart>>
     exit e[]
     #);
 #)
```

Create a new fragment file `stackImpl.bet`:

```
ORIGIN 'stack';
-- private: descriptor --
(# A: [100] ^element;
   top: @integer;
#)
-- pushBody: DoPart --
do private.top+1->private.top;
   e[] -> private.A[private.top][];
-- popBody: DoPart --
do private.A[private.top][] -> e[];
   private.top-1->private.top;
-- topBody: DoPart --
do private.A[private.top][]->e[]
```

The reason why the data representation (`A` and `Top`) is put into a `descriptor` slot instead of an `attributes` slot is that `attributes` slots may only contain patterns, no static items (objects) or object references. This is due to the implementation of separate compilation. Therefore it is necessary to put static items into an attribute (in this case `private`) that is declared by means of a `descriptor` slot. Because of this all accesses to the representation must be done via the `private` variable (see `pushBody`, `popBody` and `topBody`). Notice that the parameters are visible in the interface. If the operations had local variables they should not be shown in the interface.

6.4 Modularization with INNER

Programs fragments with do-parts that contain an `INNER` imperative e.g.:

```
ORIGIN '~beta/basiclib/v1.5/betaenv';
--- lib: attributes ---
A: (# do imp1; imp2; INNER; imp3 #)
```

can be modularized in the following two ways depending on whether the `INNER` imperative should be visible in the interface or not.

If the `INNER` is preferred visible in the interface, the interface fragment could look like (`fooLib1.bet`):

Interface

```
ORIGIN '~beta/basiclib/v1.5/betaenv';
BODY 'fooImpl1'
-- lib: attributes --
A: (#
   do <<SLOT imp12slot: descriptor>>;
      INNER;
      <<SLOT imp3slot: descriptor>>
   #)
```

and the implementation fragment (`fooImpl1.bet`):

Implementation

```
ORIGIN 'fooLib1'
-- imp12slot: descriptor --
(# do imp1; imp2 #)
-- imp3slot: descriptor --
(# do imp3 #)
```

In this case a `DoPart` slot might be used instead (`fooLib2.bet`):

Using DoPart slot

```
ORIGIN '~beta/basiclib/v1.5/betaenv';
BODY 'fooImpl2'
-- lib: attributes --
A: (# <<SLOT imp12slot: DoPart>> #)
```

with the implementation fragment (`fooImpl2.bet`):

```
ORIGIN 'fooLib2'
-- imp12slot: DoPart --
do imp1; imp2; INNER; imp3
```

Using do-parts like this, then although the `INNER` is not visible in the interface, the A pattern may still be specialized and behave as if the `INNER` was in the interface. Notice, that when specializing a pattern with no `INNER` in the do-part, the compiler will normally complain about this. But when the pattern being specialized contains a SLOT, the compiler will assume, that the SLOT contains an `INNER`. Thus it is possible to specialize the `A` pattern in `foolib2`.

But if the `INNER` imperative is placed “inside” some structure e.g.:

```
A: (#
   do (if E1
       // E2 then INNER
       // E3 then imp
      if)
   #)
```

you might not want to show the `if` imperative in the interface. In this case a variant of the `INNER` construct may be used, in which case the interface fragment could be (`fooLib3.bet`):

```
ORIGIN '~beta/basiclib/v1.5/betaenv';
BODY 'fooImpl3'
--- lib: attributes ---
A: (# do <<SLOT Abody: descriptor>> #);
```

and the implementation fragment (`fooImpl3.bet`):

```
ORIGIN 'fooLib3'
--- Abody: descriptor ---
(#
do (if E1
    // E2 then INNER A
    // E3 then imp
   if)
#)
```

If a “normal” `INNER` had been used instead of `INNER A`, it would mean that specializations of the pattern containing the `INNER` in the do-part combine the actions at this point. But the pattern containing the `INNER` in the do-part, in this case would be the anonymous pattern in the `ABody` descriptor fragment. By using `INNER A`, it is ensured, that the control flow descents to the specialization of `A` although the `INNER` is inside the `ABody` descriptor.

A `DoPart` slot could also be used here, as in the previous example.

6.5 Formal Syntax of Fragment Language

The formal syntax of the BETA fragment-system is:

Fragment Grammar

```
<TranslationUnit> ::= <Properties> <FormPart>
<FormPart> ::* <FormDef>
<FormDef> ::= -- <FormDefinition>
<FormDefinition> ::| <DescriptorForm> | <AttributesForm> | <dopart_form>
<DescriptorForm> ::= <NameDcl> : descriptor -- <ObjectDescriptor>
<AttributesForm> ::= <NameDcl> : attributes -- <Attributes>
<DopartForm> ::= <NameDcl> : dopart -- <DoPart>
```

Property Grammar

```
<Properties> ::= <PropertyList>
<PropertyList> ::+ <PropertyOpt> ';'
<PropertyOpt> ::? <Property>
<Property>  ::| <ORIGIN>
              | <INCLUDE>
              | <BODY>
              | <MDBODY>
              | <OBJFILE>
              | <LIBFILE>
              | <LINKOPT>
              | <BETARUN>
              | <MAKE>
              | <RESOURCE>
              | <ON>
              | <OFF>
              | <Other>
<ORIGIN> ::= 'ORIGIN' <TextConst>
<INCLUDE> ::= 'INCLUDE' <StringList>
<BODY> ::= 'BODY' <StringList>
<MDBODY> ::= 'MDBODY' <MachineSpecificationList>
<OBJFILE> ::= 'OBJFILE' <MachineSpecificationList>
<LIBFILE> ::= 'LIBFILE' <MachineSpecificationList>
<LINKOPT> ::= 'LINKOPT' <MachineSpecificationList>
<BETARUN> ::= 'BETARUN' <MachineSpecificationList>
<MAKE> ::= 'MAKE' <MachineSpecificationList>
<RESOURCE> ::= 'RESOURCE' <MachineSpecificationList>
<ON> ::= 'ON' <IntegerList>
<OFF> ::= 'OFF' <IntegerList>
<StringList>::+ <TextConst>
<IntegerList>::+ <IntegerConst>
<MachineSpecificationList>::+ <MachineSpecification>
<MachineSpecification> ::= <Machine> <StringList>
<Machine> ::| <NameApl> | <Default>
<Default> ::= 'default'
<Other> ::= <NameDcl> <PropertyValueList>
<PropertyValueList> ::* <PropertyValue>
<PropertyValue> ::= <Value>
<Value> ::| <NameDcl> | <IntegerConst> | <TextConst>
<NameDcl> ::= <NameDecl>
<NameApl> ::= <NameAppl>
<TextConst> ::= <String>
<IntegerConst> ::= <Const>
```

Note that the symbol `--` may consist of two or more dashes (`-`), and that the old style INCLUDE and fragment syntax (`--INCLUDE fragment`) are not described by this grammar. This old-style INCLUDE syntax is likely to be removed in a future release.

6.6 File Name Restrictions

Because of implementations details, the current version of the fragment system imposes the following restrictions on file names used for BETA programs.

1. It is not allowed for a program to use two files with the same name, say `foo.bet` (ignoring case), which both contains fragments of category `Attributes`.
2. It is not allowed for a program to use a file named, say, `foo.bet`, if `foo.bet` contains a fragment of category `Attributes`, and if there is a SLOT of category `ObjectDescriptor`/`Descriptor` or `DoPart` named `foo` in any of the files involved in the program. Again case is irrelevant.
3. It is not allowed to use the '`-`' (dash) character in fragment file names.

'-' is illegal in file names

4. Because the `FragmentDenotation` separator character is '`/`' it is not allowed to use the '`/`' in fragment file names, not even on platforms where the file system would allow it.
5. In general, it is advisable to restrict the characters used in the fragment file names to be: `a-z`, `A-Z`, `0-9`, and '`_`'. If other characters are used in the fragment file names, there is a danger, that the supporting tools (such as linkers) will complain.

Symptoms

The symptom on breaking rule 1 or 2 is typically a "`Multiple defined symbol M1FOO`" and the like, in the linking phase, the symptom for breaking rule 3 is that the compiler / Valhalla [MIA 92-12] / Sif [MIA 90-11] may become confused. Finally the symptom on breaking rule 5 may be a complaint from the assembler about illegal characters.

Except for rule 3, these restrictions only apply to the *file* names. The *directories / Folders* containing the files, may be freely named.

7 Error Handling

BETA programs containing errors will cause error messages during compilation. Error messages may appear during syntax analysis, static semantic analysis, code generation and assembly/linking. In addition various forms of system errors may occur.

7.1 Syntax Errors

A syntax error is given when there are errors in the context free syntax of the BETA program. These includes missing semicolons, non-matching brackets, etc. Such errors are printed on the screen and may look as follows:

```
Parse errors
#   1 ORIGIN '~beta/basiclib/v1.5/betaenv'
#   2 --PROGRAM: descriptor--
#   3 (# T: (# #);
#   4    X: [100) @integer;
# ************* ^
#  Expected symbols: >= mod < <= = % <> > -> * ] div +  /
   xor or and
 File "syntaxerror.bet"; Line 4
#   3 (# T: (# #);
#   4    X: [100) @integer;
#   5 do (for i: X.range repeat
#   6         3->X[i];
#   7    if)
# ******* ^
#  Expected symbols: _NAME_ _KONST_ _STRING_ none not @@
   restart leave ; (# % & ( this +  inner for tos suspend
 File "syntaxerror.bet"; Line 7
```

The error message shows that there are syntax errors in lines 4 and 7. In line 4 the arrow(`^`) points at the place where an illegal symbol is met. The compiler gives a list of acceptable symbols. In this case `)` should have been a `]`. In line 7, the `if` should have been a `for`.

7.2 Static Semantic Errors

Static semantic errors appear in situations where a name is used without being declared, where a pattern name is used as an object, etc. Each error found is printed on the screen with a small indication of the context. After the checking, a pretty print of the fragment including a precise indication of the error is generated on the `lst`-file (see section 4)[5]

In appendix A, the semantic error messages that may be reported by the compiler are listed.

7.3 Assembler and Linker Errors

Errors may also appear during assembling and linking. The following type of errors may appear:

- The assembler/linker complains about a corrupt `..s` or `.o` file. This may happen if the compilation/assembly has been interrupted for some reason leaving an incomplete file. This can usually be handled by forcing a recompilation of the corresponding BETA file. (Delete the `..s` *and* `.o` files in question)
- The disk may run full during assembling or linking. Restart compilation after having obtained more disk space.

See also section 6.6.

7.4 System Errors

Two kinds of system errors may appear: (1) Errors in the compiler, and (2) error situations in the operating systems. Most times a meaningful error message is given in these situations, but due to the nature of these errors this is not always the case.

Compiler errors should be reported to Mjølner Informatics ApS. This can be done in one of three ways:

1. **Via electronic mail using the Internet address**
   support@mjolner.dk
2. **By sending a fax to Mjølner Informatics ApS at**
   +45 86 20 12 22
3. **By issuing an ordinary mail to the address**
   Mjølner Informatics
   Science Park Aarhus,
   Gustav Wieds Vej 10
   DK-8000 Århus C
   Denmark

---

[5] Some semantic errors may cause the compiler to fail without generating a pretty print. There should however always be an error indication on the screen. In case the compiler fails during checking and it is not obvious for what reason, it is possible to trace the checking of declarations and imperatives using the option `--traceCheck` (see section 8). However, this may generate a large amount of output on the screen. The compiler may also fail during code generation. These errors may be traced using option `--traceCode`. However, tracing errors in this way should rarely be needed.

Operating system errors are often due to local problems. Examples of such errors may be: insufficient access to files, no more disc space, file server inaccessible, etc.

7.5 Run-time Errors

Run-time errors are errors in the program detected during its execution. In this case an error message is given and a dump of the call stack of objects is generated on the file `foo.dump` if the program is named `foo`.

Consider the following fragments (note that the name of the fragments are complete UNIX file paths)

`/usr/smith/mylib.bet`:

```
ORIGIN '~beta/basiclib/v1.5/betaenv'
--LIB: attributes--
lib1: (# do INNER #);
lib2: lib1
  (# T: (# x: @integer #);
     R: ^T
  do (* &T[]->R[] *)
     111->R.x; (* R[] is NONE *)
     INNER
  #);
lib3: lib2(# do 'hello'->putLine #)
```

`/usr/smith/runtimeerr.bet`:

```
ORIGIN '~beta/basiclib/v1.5/betaenv';
INCLUDE 'mylib'
--PROGRAM: descriptor--
(# foo1: (# do foo2 #);
   foo2: (# do foo.foo3 #);
   foo: @(# foo3: (# do lib3 #)#)
do foo1
#)
```

Execution of this program on a sun4 machine will result in the following `runtimeerr.dump` file:

.dump file

```
Beta execution aborted: Reference is none.

Call chain: (sun4)

  item lib3#<lib2#>lib1# in /usr/smith/mylib
    -- BETAENV-~ in ~beta/basiclib/v1.5/betaenv
  item <foo3#> in /usr/smith/runtimeerr
    -- foo# in /usr/smith/runtimeerr
  item <foo2#> in /usr/smith/runtimeerr
    -- PROGRAM-~ in /usr/smith/runtimeerr
  item <foo1#> in /usr/smith/runtimeerr
    -- PROGRAM-~ in /usr/smith/runtimeerr
  comp <PROGRAM-~> in /usr/smith/runtimeerr

  basic component in ~beta/basiclib/v1.5/betaenv
```

The information in `runtimeerr.dump` has the following meaning:

- The activation stack of invoked objects is shown. Each element of the stack is shown as two lines. The object and its statically enclosing object.
- For each object, the name of the file where it is defined is also shown.

- From the above file it can be seen that the error occurred in an instance of `lib3`. The description `lib3#<lib2#>lib1` shows the superpattern chain of `lib3`. The braces (<,>) indicates that the error occurred in the `do`-part of `lib2`.
- The symbol immediately after the name of an object shows its kind. The different possibilities are:

  # The descriptor belongs to a pattern, e.g. `P: (#...#)`

  ~ Singular named descriptor, e.g. `X: @(# ... #)`

  * Singular unnamed descriptor, e.g. `...; (# ... #);...`

  - Descriptor SLOT.

  Notice that, e.g. the PROGRAM SLOT is marked with both - and ~ since a descriptor SLOT gives rise to a singular named descriptor.
- It can be seen that `lib3` was called from `foo3`, which was called from `foo2`, which was called from `foo1`, etc. The bottom most objects are defined in `be-taenv`
- For each active object its enclosing object is shown, on a line starting with “`--`” The encloser of e.g. `foo3` is `foo`. The rest of the objects have enclosers, which are slots.
- For each object, the corresponding fragment file is shown. The pattern `lib3` is defined in the file
  `/usr/smith/mylib`

8 Compiler Arguments

When activating the BETA compiler, the following command line arguments are valid.

Most options have both a "`--<name>`" and a "`--no<name>`" form: Activate the option using "`--<name>`"; deactivate the option using "`--no<name>`". In the listing below, the activating form is shown first (and explained), if both exist for an option.

Shortcuts

For most options, there is a short (one-character) option for the non-default form. One-character options allow multiple option characters after the "`-`" (e.g. "`-qwd`").

Case sensitiveness

Long option names are case insensitive, whereas one-character options are case sensitive.

A star (*****) in the listings below indicates the *default* option.

--help	**-h**	Show a brief overview of the legal command line options
--repeat	**-r**	Run compiler in repeating mode. After having translated the fragments specified in the argument list, if in repeating mode, the compiler prompts the user for the name of another fragment to be translated: `Type Fragment File Name:` This interaction is continued until the compiler is explicitly killed, e.g. by sending a `control-C` or the end-of-stream character to the compiler process. The compiler may also be given additional options at the prompt, e.g. you may type `--nolink foo.bet` to translate `foo.bet`, but avoid linking of it. If no new fragments are specified at the prompt, the compiler will retranslate the last fragment it has translated when `<RETURN>` it typed. By using repeating mode, the compiler saves time when analyzing dependencies between fragments, since fragments are saved in memory between compilations.
--noRepeat	*	
--link	*	Link program
--noLink	**-x**	
--static		Use static linking
--dynamic	*	Use dynamic linking
--list	*	Generate .lst file, if semantic errors
--noList	**-l**	

--debug	*	Generate debug info to enable debugging. Include debugging information in the generated code. This is used by the BETA debugger—valhalla. On the other hand, using `--noDebug` reduces the size of the executable files by 30-50%, and also speeds up linking time.
--noDebug	**-d**	
--code	*	Generate code
--noCode	**-c**	
--checkQua	*	Generate runtime checks for QUA errors
--noCheckQua	**-Q**	
--checkNone	*	Generate runtime checks for NONE references
--noCheckNone	**-N**	
--checkIndex	*	Generate runtime checks for repetition index out of range
--noCheckIndex	**-I**	
--warn	*	Generate warnings
--noWarn	**-w**	
--warnQua	*	Generate warnings about runtime QUA checks
--noWarnQua	**-q**	
--verbose		Verbose compiler info output
--quiet	*	Only compiler info on parse, check, etc.
--mute		No compiler info output
--traceCheck		Trace the compiler during semantic checking
--noTraceCheck		
--traceCode		Trace the compiler during code generation
--noTraceCode		
--out	**-o**	Specify name to use for resulting executable image
--preserve	**-p**	Preserve generated `.job` and assembly files
--noPreserve	*	
--job	*	Execute the `..job` file
--noJob	**-j**	
--switch	**-s**	Set/unset one or more compiler switches. The `-s` option makes it possible to define one or more so-called compiler switches. Switches are specified as integers on the command line after `--switch` or `-s`, possibly terminated by a `0` (zero). Switches are used for a number of purposes: parameterization of the compiler, debugging, testing etc. The most interesting switches with respect to parameterization are listed below; notice that some of them may also be set as ordinary options.

- **5**: Suppress code generation. I.e. only semantic checking is performed. This switch will also set switch 33. Same as **-c**.
- **6**: Suppress linking. Same as **-x**.
- **14**: Do not generate run-time checks for NONE-references. Same as **-N**

- **15**: Do not generate run-time checks for index-errors. Same as **-I**.
- **18**: Preserve assembly- and job-files. Same as **-p**.
- **19**: Suppress notification of insertion of run-time checks for qualification errors in reference assignment. Same as **-q**.
- **21**: Continue translation after semantic errors.
- **23**: Preserve job-files.
- **32**: Do not produce .lst file in case of semantic errors. Same as **-l**.
- **33**: Do not execute .job file. Same as **-j**.
- **37**: Do not generate debugging information. Same as **-d**.
- **42**: Do not generate run-time checks for qualification errors in reference assignment. Same as **-Q.**
- **191**: Print each descriptor just before it is checked.
- **192**: Print each declaration just before it is checked.
- **193**: Print each imperative just before it is checked.
- **308**: Print each declaration just before code is generated for it.
- **311**: Print each imperative just before code is generated for it.

--linkOpts	Specify text string to be appended to the link directive
fragment1 … fragmentN	Arguments other than the above mentioned options are treated as the names of fragments to be translated by the compiler. It should be noted that for an option to take effect in the translation of a fragment whose name is passed as argument to the compiler, the option must appear *before* the fragment name in the argument list.

9 Machine Dependent Configurations

In this section, the terminology of the fragment system is used freely without further explanation. The fragment system has been extended to support generic software descriptions. The same generic software description may be used to instantiate configurations for different machines. The term "machine" covers a CPU and an operating system running on that CPU.

Generic properties

The concept of generic software descriptions is implemented by means of special "generic properties". Normally, a property has exactly *one* associated set of values. A generic property has *a number of* such value-sets. The idea is that the programmer can specify a value-set for each machine. These value-sets are the ones termed `<MachineSpecificationList>` in the formal specification of properties in section 6.3 and 6.5. As an example:

```
OBJFILE  sun4    'xlib.o'
         linux   'zlib.o'
         default 'wlib.o'
```

Configurations

OBJFILE is the name of a generic property. The OBJFILE property is used for inclusion in the linkage phase of external object files, e.g. produced by a C compiler. A generic property specification should be seen as a kind of "switch/case" statement. The semantics of the above OBJFILE property is that when instantiating a configuration for the machine `sun4`, the value `xlib.o` is chosen. This means that the object file `xlib.o` is included when linking a configuration for a `sun4` machine. Similarly for `linux` machines. The `default` literal indicates that when instantiating configurations for machines *other* than `sun4` or `linux` , the object file `wlib.o` should be included.

Besides OBJFILE, there are the following generic properties: MAKE, BETARUN, LIBFILE, LINKOPT, RESOURCE, and MDBODY. For all of these properties, the relation between machine symbols and value-sets are specified in the same manner as described above. To be precise, the following algorithm is used when instantiating a configuration for a specific machine type, say A.

1. If A matches any of the machine symbols of the generic property, the value-set associated with that particular machine symbol is chosen. If no match is possible, proceed with step 2.
2. If the symbol `default` is specified as machine symbol, the associated value-set is chosen. If not, a warning is issued.

The only distinction between the different generic properties is in the interpretation of the elements of the chosen value-set. For OBJFILE, the value-set is interpreted as external object files. MAKE is meant to point out a number of so-called makefiles. These are executed just prior to the linkage phase. A makefile is often used to keep the included object files up to date with respect to the source files from which they originate. For BETARUN, the value-sets must contain exactly one element, and this element denotes the runtime system to be used in the resulting configuration. With respect to LIBFILE, the elements of the value-sets are interpreted as external libraries, e.g. the X11 library, to be included in the linkage phase. The chosen value-set in an MDBODY property denotes ordinary BETA fragments to be treated as if they had been specified by means of a normal BODY property. The MDBODY property may thus be used to specify that a fragment appears in a number of machine dependent variants. Finally, the LINKOPT property denotes arguments to append to the link-directive in the linking phase of compilations. Finally, the RESOURCE property is used (only on PC and Macintosh) to specify a set of resource files to add to the application.

Cross-compilation

Configurations are instantiated by the compiler, by default for the machine on which the compilation takes place. It is possible to instantiate a configuration for a machine other than the one, on which the compilation is performed (“cross-compilation”). This requires extensions to the Mjølner BETA System; please contact Mjølner Informatics if this is needed.

10 Code Generation for Multiple Machines

When instantiating a configuration for some machine, a number of object files are produced by the compiler - one for each fragment contributing to the configuration. On most architectures, the compiler actually generates symbolic assembly code, and this code is turned into object files by means of the native assembler. The native linker is used to produce an executable image for the machine in question on basis of these object files.

10.1 Placement of Object Code

Different machines normally use different formats for object files. The files containing object code and symbolic assembly code are always placed in a sub-directory relative to the directory containing the common source code. A sub-directory is created for each special object file format. Currently the following subdirectories are used:

`sun4`	SUN-4 (SPARC) running SunOS 4.x
`sun4s`	SUN-4 (SPARC) running Solaris 2.x
`hpux9pa`	HP 9000/700 running HP-UX 9.x
`hpux9mc`	HP 9000/300-400 running HP UX 9.x
`sgi`	Silicon Graphics (MIPS) running IRIX 5.3
`linux`	PC running Linux 1.0 or later
`nti`	PC running Windows NT
`mac`	Macintosh mc680x0, MPW 3.2 or later

For executable images to be activated “directly”, without prefixing their name with the name of a sub-directory, executable images are placed in the same directory as the common source files. It is however possible to control the naming of the executable images. This is done by means of the -o option to the compiler.

10.2 Macro Expansion

Consider this use of the MDBODY property:

```
MDBODY default './$/betaenvbody_$'
```

The symbol `$` is expanded by the compiler. It is expanded to the name of the subdirectory into which the generated code will be placed. That is, if code is generated for a `mac` (Macintosh) machine, the above expands to `./mac/betaenvbody_mac`. This may be a convenient short-hand, but may also make is possible to instantiate configurations for new machines without changing the original source code.

Bibliography

[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 90-4]	Mjølner Informatics: *The Mjølner BETA System: Using BETA on UNIX Systems*, Mjølner Informatics Report MIA 90-4.
[MIA 90-6]	Mjølner Informatics: *The Mjølner BETA System: Using BETA on the Macintosh*, Mjølner Informatics Report MIA 90-6.
[MIA 90-8]	Mjølner Informatics: *The Mjølner BETA System: Basic Libraries*, *Reference Manual*, Mjølner Informatics Report MIA 90-8
[MIA 90-11]	Mjølner Informatics: *Sif – A Hyper Structure Editor, Tutorial and Reference Manual* Mjølner Informatics Report MIA 90-11.
[MIA 92-12]	Mjølner Informatics: *The Mjølner BETA System – The BETA Source-level Debugger – Users's Guide*, Mjølner Informatics Report MIA 92-12
[MIA 94-24]	Mjølner Informatics: *The Mjølner BETA System – The Mjølner BETA System Tutorial* MjølnerInformatics Report MIA 94-24.
[MIA 94-34]	Mjølner Informatics: *The Mjølner BETA System – Using on Windows 95 or Windows NT* MjølnerInformatics Report MIA 94-34.

Appendix A. Semantic Errors and Warnings

A.1 Semantic Errors

The following is a list of semantic error messages that may be reported by the compiler. See also section 7.2.

1. Name is declared more than once
2. Name is not declared
3. Attribute is not declared
4. A pattern is expected here
5. An item is expected here
6. A repetition is expected here
7. A simple evaluation cannot be assigned
8. The lists have different lengths
9. The lists have different lengths
10. In "leave P" or " restart P", "P" must be an enclosing label
    or enclosing pattern
11. Illegal assignment/comparison of value, reference or repetition
12. Only a single name is allowed here
13. Attempt to bind V which is not virtual (V ::< T)
14. In V ::< T, T does not have a correct qualification
15. An object is expected here
16. A basic pattern cannot be used as a super-pattern
17. A virtual pattern or a pattern defined as a descriptor slot cannot
    be used as super-pattern
18. A string of length 1 is a char - NOT a text
19. Illegal recursion in the definition of a pattern.
    One of the following type of errors have occurred:
    (1) There may be a circle in the super-pattern chain:
    ```
    A: C(# ... #); B: A(# ... #); C: B(# ... #)
    ```

(2) The pattern may direct or indirectly contain a static instance of itself:

```
P: (# ...; X: @P; ... do ... #)
```

(3) The pattern may directly or indirectly contain an inserted instance of itself:

```
P: (# ... do ...; P(# ... #); ... #)or
A: (# ... P: (# R: ^A; ... do ...; R.P(# ... #); ... #) ... #)
```

20. Incompatible qualifications in assignment/comparison
21. Only simple values or references may be compared
22. Only simple values may appear in unary expressions
23. Fatal error: virtual binding not found
27. The descriptor is both used as item and component
28. Static size of descriptor is larger than 32760 bytes
29. Illegal recursion in object-description
30. Illegal assignment to constant value/reference or repetition
31. Only pattern-declarations may appear in a fragment of category 'attributes'
32. A virtual qualification must be a pattern name or a descriptor
33. A virtual pattern or descriptor-slot cannot be used as a component
34. An enter/exit parameter of an "external" must be one of:
    integer,char,real,integer-repetitions,char-repetition,
    subpattern of cstruct,variable-subpattern of external
35. An "external" can only have one exit parameter
36. A sub-pattern of "external" cannot be used as super-pattern
37. The DO-part of an "external" should be empty
38. A repetition/for-imp range must be an integer, char or boolean evaluation
39. A simple pattern cannot be used here
40. Unknown inline primitive
41. The superpattern of this descriptor has no INNER
42. Attempt to bind a virtual in a descriptor with no superpattern
43. The qualification of a variable pattern must be a pattern
44. A pattern-, virtual-, variable-pattern, or reference is expected here
45. A repetition name is expected here
46. In "this(P)" or "inner P", P must be the name of an enclosing pattern
47. An unexpanded nonterminal must be a SLOT
48. A super-pattern must be a simple pattern or a simple
    pattern attribute of a static object
49. A simple pattern or virtual pattern cannot be assigned

a structure reference

50. A structure reference can only be assigned to/compared with another structure reference
51. Only integer,char,boolean, real objects and references can be compared in an if-imperative
52. Rename declaration has NOT been implemented
53. Syntax error in number
54. Name not declared. There is no corresponding virtual declaration
55. A pattern with a do-part slot cannot be used as a super-pattern
56. The QUA construct has not been implemented
57. A basic pattern like integer, real, char, boolean, false, and true cannot be used as a super-pattern
58. In a list being assigned to and being assigned from as in
    ...->(E1,E2,...En)-> ...
    the elements may not be patterns
59. The enter-parameters of an external call must be supplied
60. The left-side of the assignment/comparison has no (exit-)list or the right-side has no (enter-)list
61. An element of the left-side/right-side of the assignment/comparison has no (exit-)list or (enter-)list
62. The Left-side of the assignment/comparison has no (exit-)list
63. An element of the left-side of the assignment/comparison has no (exit-)list
64. The right-side of the assignment/comparison has no (enter-)list
65. An element of the right-side of the assignment/comparison has no (enter-)list
66. A simple value (integer,boolean,char,real) cannot be assigned/compared to/with a list
67. An object with no exit-list is being assigned/compared to a reference.

    The left-side may be missing a "[]" or the right-side may have a superfluous "[]"
68. An element with no exit-list in the left-side list is being assigned/compared to a reference on the right-side

    The left-side may be missing a "[]" or the right-side may have a superfluous "[]"
69. A reference is being assigned/compared to an object with no enter-list

    The right-side may be missing a "[]" or the left-side may have a superfluous "[]"
70. A reference is being assigned/compared to an element on the right-side with no enter-list

    The right-side may be missing a "[]" or the left-side may have a superfluous "[]"
80. "inner P" is only legal in the do-part of the pattern "P"
81. In a computed-remote, "(EV).X","EV" cannot be an evaluation-list

82. In a computed-remote, "(EV).X", "EV" must have one exit-element,
    which must be a reference
83. In a computed-remote, "(EV).X", "EV" is not a legal evaluation
84. "Extend" and "new" must have an enter-parameter
85. "leave P" or "restart P", where "P" is a pattern,
    is only legal in the do-part of "P"
87. A repetition index must be an integer-evaluation
88. The base of this number is too large
89. A subpattern of "data" may only have declarations of the forms:
    "X: ^T" where "T" is subpattern of "data", or
    "X: @T" where "T" is integer,shortint,char,boolean,real
        or subpattern of "data"
90. A subpattern of "data" may not have a do-part
91. A boolean evaluation is expected here
92. Primitive operation appears in wrong context
93. It is not possible to obtain a structure reference for a basic pattern
    like integer, real, char, boolean, false, and true or instances of these
94. A virtual pattern cannot be bound to a basic pattern like
    integer, real, char, boolean, false, and true
96. In "X: ^<AD>.P", "Y: @<AD>.P", "<AD>" cannot be:
        a repetition element as in "R[e].P"
        a computed remote as in"(R).P"
    It must be a static object
98. A sub-pattern of "external" must be defined as a pattern
100. In "V ::< T", "T" must be a non-virtual pattern
101. In "V :: T", "T" must be a pattern
102. A cycle has been detected in the super-chain of the virtual/final binding
103. Incompatible types of binary operator
104. Incompatible left- and right-side of assignment
105. Illegal assignment to constant, literal or expression
107. A virtual cannot be bound to a slot
108. Illegal use of the "&"-operator
110. Illegal recursion in exit list:
     a pattern is referred directly or indirectly in its own exit list
111. Illegal recursion in enter list:
     a pattern is referred directly or indirectly in its own enter list
112. External entry point has a blank- or control character
113. There is a circle in the super-pattern chain

A.2 Semantic Warnings

24. A run-time qualification check will be generated here
25. Repetition of static components is not implemented
26. Repetition of non simple patterns is not implemented
86. "leave P" and "restart P", where "P" is a pattern,
    are currently not allowed in internal descriptors of "P"
95. In "X: ^R.P", "Y: @R.P", or "Z: @R.P(#...#),
    "R" should NOT be a dynamic reference!
    For "X: ^R.P", consider using "X: ^T.P",
    where "T" is the pattern qualifying "R" ("R: ^T").
    A future release may consider this to be a semantic error.
97. An "inner" in a singular object will never be executed
99. Final binding to a virtual pattern is a new facility
    in this version of the compiler.
    Please report any problems to support@mjolner.dk
106. Assignment/comparison between boolean and integer
109. Text has a null-char. All chars after the null-char are ignored

Appendix B. The BETA Grammar

This appendix contains a listing of a grammar describing the BETA language accepted by the compiler. The grammar formalism used in the Mjølner BETA System is a variant of context free grammars. A *structured context free grammar* is a context free grammar (CFG) where the rules (productions) satisfy a certain structure. See [MIA90-8] for a description of structured context free grammars.

```
<BetaForm>         ::| <DescriptorForm>
                     | <AttributesForm>
<DescriptorForm>   ::= <ObjectDescriptor>
<AttributesForm>   ::= <Attributes>
<ObjectDescriptor> ::= <PrefixOpt> <MainPart>
<MainPart>         ::= '(#' <Attributes> <ActionPart> '#)'
<Attributes>       ::+ <AttributeDeclOpt> ';'
<PrefixOpt>        ::? <Prefix>
<Prefix>           ::= <AttributeDenotation>
<AttributeDeclOpt> ::? <AttributeDecl>
<AttributeDecl>    ::| <PatternDecl>
                     | <SimpleDecl>
                     | <RepetitionDecl>
                     | <VirtualDecl>
                     | <BindingDecl>
                     | <FinalDecl>
<PatternDecl>      ::= <Names> ':' <ObjectDescriptor>
<SimpleDecl>       ::= <Names> ':' <referenceSpecification>
<RepetitionDecl>   ::= <Names> ':' '[' <index> ']' <referenceSpecification>
<VirtualDecl>      ::= <Names> ':' '<' <ObjectSpecification>
<BindingDecl>      ::= <Names> ':' ':' '<' <ObjectSpecification>
<FinalDecl>        ::= <Names> ':' ':' <ObjectSpecification>
<VariablePattern>  ::=  '##' <AttributeDenotation>
<referenceSpecification> ::| <StaticItem>
                           | <DynamicItem>
                           | <StaticComponent>
                           | <DynamicComponent>
                           | <VariablePattern>
<StaticItem>       ::= '@' <ObjectSpecification>
<DynamicItem>      ::= '^' <AttributeDenotation>
<StaticComponent>  ::= '@' '|' <ObjectSpecification>
<DynamicComponent> ::= '^' '|' <AttributeDenotation>
<ObjectSpecification>    ::| <ObjectDescriptor>
                           | <AttributeDenotation>
<Index>            ::| <SimpleIndex>
                     | <NamedIndex>
<NamedIndex>       ::= <NameDcl> ':' <Evaluation>
<ActionPart>       ::= <EnterPartOpt> <DoPartOpt> <ExitPartOpt>
<EnterPartOpt>     ::? <EnterPart>
<DoPartOpt>        ::? <DoPart>
<ExitPartOpt>      ::? <ExitPart>
<EnterPart>        ::= 'enter' <Evaluation>
<DoPart>           ::= 'do' <Imperatives>
```

```
<ExitPart>          ::= 'exit' <Evaluation>
<Imperatives>       ::+ <ImpOpt> ';'
<ImpOpt>            ::? <Imp>
<Imp>               ::| <LabelledImp>
                      | <ForImp>
                      | <SimpleIfImp>
                      | <GeneralIfImp>
                      | <LeaveImp>
                      | <RestartImp>
                      | <InnerImp>
                      | <SuspendImp>
                      | <Evaluation>
<LabelledImp>       ::= <NameDcl> ':' <Imp>
<ForImp>            ::=
          '(' 'for' <Index> 'repeat' <Imperatives> 'for' ')'
<GeneralIfImp>      ::=
          '(' 'if' <Evaluation> <Alternatives> <ElsePartOpt> 'if' ')'
<SimpleIfImp>       ::=
          '(' 'if' <Evaluation> 'then' <Imperatives> <ElsePartOpt> 'if'
<LeaveImp>          ::= 'leave' <NameApl>
<RestartImp>        ::= 'restart' <NameApl>
<InnerImp>          ::= 'inner' <NameAplOpt>
<NameAplOpt>        ::? <NameApl>
<SuspendImp>        ::= 'suspend'
<Alternatives>      ::+ <Alternative>
<Alternative>       ::= <Selections> 'then' <Imperatives>
<Selections>        ::+ <Selection>
<Selection>         ::| <CaseSelection>
<CaseSelection>     ::= '//' <evaluation>
<ElsePartOpt>       ::? <ElsePart>
<ElsePart>          ::= 'else' <Imperatives>
<Evaluations>       ::+ <Evaluation> ','
<Evaluation>        ::| <Expression>
                      | <AssignmentEvaluation>
<AssignmentEvaluation> ::= <Evaluation> '->' <Transaction>
<Transaction>       ::| <ObjectEvaluation>
                        | <ObjectReference>
                      | <EvalList>
                      | <StructureReference>
<ObjectEvaluation> ::| <InsertedItem>
                      | <reference>
<Reference>         ::| <ObjectDenotation>
                      | <DynamicObjectGeneration>
<DynamicObjectGeneration>    ::| <DynamicItemGeneration>
                               | <DynamicComponentGeneration>
<InsertedItem>               ::= <ObjectDescriptor>
<ObjectDenotation>           ::= <AttributeDenotation>
<ObjectReference>            ::= <Reference> '[]'
<StructureReference>         ::= <AttributeDenotation> '##'
<EvalList>                   ::= '(' <Evaluations> ')'
<DynamicItemGeneration>      ::= '&' <ObjectSpecification>
<DynamicComponentGeneration> ::= '&' '|' <ObjectSpecification>

<AttributeDenotation>        ::| <NameApl>
                               | <Remote>
                               | <ComputedRemote>
                               | <Indexed>
                               | <ThisObject>
<Remote>             ::= <AttributeDenotation> '.' <NameApl>
<ComputedRemote>     ::= '(' <Evaluations> ')' '.' <NameApl>
<Indexed>            ::= <AttributeDenotation> '[' <Evaluation> ']'
<ThisObject>         ::= 'this' '(' <NameApl> ')'
<Expression>         ::| <RelationalExp> | <SimpleExp>
<RelationalExp>      ::| <EqExp> | <LtExp> | <LeExp>
                       | <GtExp> | <GeExp> | <NeExp>
<SimpleExp>          ::| <AddExp> | <SignedTerm> | <Term>
```

```
<AddExp>            ::| <PlusExp> | <MinusExp> | <OrExp> | <XorExp>
<SignedTerm>        ::| <unaryPlusExp> | <unaryMinusexp>
<Term>              ::| <MulExp> | <Factor>
<MulExp>            ::| <TimesExp> | <RealDivExp> | <IntDivExp>
                      | <ModExp> | <AndExp>
<EqExp>             ::= <Operand1:SimpleExp> '=' <Operand2:SimpleExp>
<LtExp>             ::= <Operand1:SimpleExp> '<' <Operand2:SimpleExp>
<LeExp>             ::= <Operand1:SimpleExp> '<=' <Operand2:SimpleExp>
<GtExp>             ::= <Operand1:SimpleExp> '>' <Operand2:SimpleExp>
<GeExp>             ::= <Operand1:SimpleExp> '>=' <Operand2:SimpleExp>
<NeExp>             ::= <Operand1:SimpleExp> '<>' <Operand2:SimpleExp>
<PlusExp>           ::= <SimpleExp> '+' <Term>
<MinusExp>          ::= <SimpleExp> '-' <Term>
<OrExp>             ::= <SimpleExp> 'or' <Term>
<XorExp>            ::= <SimpleExp> 'xor' <Term>
<unaryPlusExp>      ::= '+' <Term>
<unaryMinusExp>     ::= '-' <Term>
<TimesExp>          ::= <Term> '*' <Factor>
<RealDivExp>        ::= <Term> '/' <Factor>
<IntDivExp>         ::= <Term> 'div' <Factor>
<ModExp>            ::= <Term> 'mod' <Factor>
<AndExp>            ::= <Term> 'and' <Factor>
<Factor>            ::| <TextConst>
                      | <IntegerConst>
                      | <NotExp>
                      | <NoneExp>
                      | <RepetitionSlice>
                      | <Transaction>
<RepetitionSlice>   ::= <AttributeDenotation>
                        '[' <Low:Evaluation> ':' <High:Evaluation> ']'
<notExp>            ::= 'not' <factor>
<noneExp>           ::= 'none'
<Names>             ::+ <NameDcl> ','
<NameDcl>           ::= <NameDecl>
<NameApl>           ::= <NameAppl>
<SimpleEntry>       ::? <TextConst>
<TextConst>         ::= <String>
<IntegerConst>      ::= <Const>
<SimpleIndex>       ::= <Evaluation>
```

Appendix C. New Features in v5.2

The following new features have been implemented in version 5.2 of the compiler, compared to version 5.1.

C.1 New Platforms

A lot of efforts have been put into porting the compiler into some new platforms:

- A final version for Silicon Graphics MIPS is now available.
- The linux compiler now generates binary code directly.
- Work is going on the make a binary compiler for Windows NT and Windows 95 too.
- Work is going on to make native binary code generation for PowerPC based macintoshes.

This work has caused a lot of changes to the interior of the compiler and runtime-system. These changes should be transparent to the user, though.

C.2 ## now allowed for objects

You may now use `P##` as an alternative to `P.struc`, when `P` is an *object*. Previously `##` was only allowed for *patterns*.

C.3 `CC` set in UNIX job files

The job files on UNIX platforms now set the `CC` environment variable to a suitable default value before executing the `MAKE` commands.

Thus `$(CC)` may now be used in the make files on UNIX platforms.

C.4 Check for bound SLOTs.

In general the compiler will only attempt to link, if a PROGRAM slot has been found in the dependency graph (this feature was introduced in v5.1 of the compiler, but the implementation was buggy).

If SLOTs of category `DoPart` or `Descriptor` in the dependency graph are not bound, and linking would otherwise have happened, the compiler now issues a warning, and does not attempt to link. This prevents the kind of error that could give an "Undefined Reference" error at link time in v5.1 of the compiler.

Likewise, if two or more fragments tries to bind the same SLOT, the compiler will give a warning. This prevents the kind of error, that could give an "Multiply Defined Symbol" error at link time in v5.1 of the compiler.

C.5 Interfragment leave/restart

The compiler now supports interfragment leave/restart as in

```
foo.bet:

ORIGIN '~beta/basiclib/v1.5/betaenv';
BODY 'foobody';
--PROGRAM:descriptor---
(# do L: <<SLOT LL:descriptor>> #)


foobody.bet:

ORIGIN 'foo';
--LL:descriptor--
(# do leave L #)
```

This feature did not work in previous versions of the compiler.

C.6 Generalized special characters in string literals

The following special characters are now allowed in BETA string literals. Some of them, e.g. `\t`, also worked in previous versions too.

`\a`	alert (bell) character	`\v`	vertical tab
`\b`	backspace	`\\`	backslash
`\f`	formfeed	`\?`	question mark
`\n`	newline	`\'`	single quote
`\r`	carriage return	`\"`	double quote
`\t`	horizontal tab	`\ooo`	octal number

Notice that you may now use `\'` as an alternative to `''` to include a literal quote in a string. E.g.: `'Tom\'sCottage'`. This has the consequence, though, that to type the backslach character, you must now do it as: `'\\'` instead of the previous way: `'\'`

`\ooo` can also be `\o` or `\oo`, provided that the character immediately following it is not a digit.

Index