90-02: Beta Compiler - Reference Manual

Table of Contents

Copyright Notice

Mjłlner Informatics Report
90-02
August 1999

BETA Compiler Reference Manual

1. Introduction

This manual describes version 5.4 of the BETA compiler (corresponding to release 5.0 of the Mjølner System). The compiler implements most parts of the BETA language as described in [MMN93]. There are, however, some changes that have been made to BETA since the publication of [MMN93]. These changes are described in:

- BETA Language Modifications

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 compile- and run-time errors. These sections contain useful information for all users.

The remaining sections are only for advanced users. In section 6, a number of different arguments to the compiler are described. In section 7, it is described how to instantiate machine dependent configurations of a program. In section 8 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.

hello.bet

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

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

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

describes that the fragment `betaenv` from the BETA basic library (basiclib) is used.

The fragment name and category:

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

describes that the BETA program is filled into a slot in betaenv called PROGRAM. The BETA compiler is integrated with the Mjølner 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

```
beta foo.bet
```

which will compile, assemble, and link the BETA fragment. On most platforms binary object code is generated directly. In this case the assembly phase is omitted. For the HP platforms assembly code is generated and assembled. The final object code will be in the file `foo`, which may be executed.

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 [MIA 99-36].

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.

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.bet

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

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.

When using the Mjølner System, the BETA library must be installed on the file system of the computer. BETA fragments will need to refer to fragments in the BETA library. Since the location of the BETA library may differ from machine to machine, a fragment may denote the BETA library by means of the name:

```
~beta
```

The file containing the betaenv fragment may then be denoted by

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

On many UNIX systems, the BETA library is often placed in the directory:

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

In this case a fragment denotation like

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

then refers to the file

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

The meaning of `~beta` can be changed by using the `BETALIB` environment variable, see [MIA 99-36].

A program using betaenv may then look as follows:

hello.bet

```
ORIGIN '~beta/basiclib/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
```

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. (This is currently not possible on Windows and Macintosh.)

Please consult the BETA tutorial [MIA94-26] for a quick survey of the BETA language and the basic libraries.

4. Files Generated by the Compiler

For each fragment file, a number of other files and directories may be produced by the compiler.For the BETA fragment `foo.bet` the following files and directories are produced: Then

File: `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. The compiler also prints short error messages on the screen during compilation.

File: `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 System.

One of the following directories depending on the platform: (The extensions `..s` and `.o` differ on the various platforms.)

`sun4s`: on a SUN Sparc running Solaris

`sgi`: on a Silicon Graphics MIPS running Unix

`linux`: on a PC running Linux

`hpux9pa`: on a HP PA Risc running HP UX

`nti-gnu`: on a Windows PC based on GNU tools

`nti-ms`: on a Windows PC based on Microsoft tools

`ppcmac`: on a Power Macintosh

The code directory contains the following files:

File: `foo..s`

contains the generated assembly code for the compiled source code (As mentioned assembly code is only generated for HPUX9PA) The assembly file is usually deleted by the compiler after assembly.

File: `foo.o`

contains the object code generated by the compiler or assembler.

File: `foo.db`

contains information used by the debugger Valhalla when debugging the foo fragment. See [MIA90-12].

File: `foo..job`

containing directives for assembly and linking. This file is usually deleted by the compiler after linking.

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

File: `foo`

containing the executable code for the program.

5. 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.

5.1 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/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.

5.2 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).

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.

Semantic error messages and warnings that may be reported by the compiler are listed in

Semantic error messages and warnings

5.3 Check for bound SLOTs.

In general the compiler will only attempt to link, if a PROGRAM slot has been found in the dependency graph.

If SLOTs of category DoPart or Descriptor in the dependency graph are not bound, and linking would otherwise have happened, the compiler issues a warning, and does not attempt to link.

Likewise, if two or more fragments tries to bind the same SLOT, the compiler will give a warning.

5.4 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 linker may report errors such as "Undefined Reference" or "Multiply Defined Symbol". This may be due to violations of the restrictions mentioned in section 6.
- The disk may run full during assembling or linking. Restart compilation after having obtained more disk space.

See also section 6.6.

5.5 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
See [MIA 99-36], section 4.5

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.

5.6 Run-time Errors

Runtime 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.

See

BETA runtime error stack dump

for details.

6. 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.

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").

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 debuggerÑvalhalla. On the other hand, using --noDebug forces the linker to strip the application , which reduces the size of the executable files by 30-50%, and also speeds up linking time. The actual machinelevel code generated for the BETA program is identical with or without debug info.???

--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.

7. 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.

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  sun4s   ’xlib.o’
         linux   ’zlib.o’
         default ’wlib.o’
```

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 sun4s, the value xlib.o is chosen. This means that the object file xlib.o is included when linking a configuration for a sun4s machine. Similarly for linux machines. The default literal indicates that when instantiating configurations for machines *other* than sun4s 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.

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 System; please contact Mjølner Informatics if this is needed.

9.1 BUILD Property

The BUILD property unifies the OBJFILE and MAKE properties. The BUILD property is used to specify rules for keeping external (i.e. non-BETA) sources up to date, and to include the external files in the link directive.

Syntax:

```
BUILD <machine> '<objectfile>'
     '<dep1>' '<dep2>' ... '<depN>'
     '<command>';
```

where

<machine> is the target machine specification (see MDBODY description)

<objectfile> is the external objectfile to include and possibly maintain. A $$ in this specification is expanded to the machine type. This is unlike other properties, like MDBODY, where a single $ is

expanded to the machine type. If a backslash (\) or a newline must be included literally in the specification, it must be quoted with backslash.

```
<dep1>
<dep2>
...
<depN>
```

Are source files, that the <objectfile> depends on.

<command> is a command (sequence) that is executed by the compiler as it is, except for the following substitutions:

$$ is expanded to the machine type, as explained above.

$0 is expanded to <objectfile>

$1 is expanded to <dep1>

$2 is expanded to <dep2>

...

$N is expanded to <depN>

If a backslash (\) or a newline must be included literally in the commands, it must be quoted with backslash.

Functionality

The <objectfile> is included in the link directive. The compiler will execute <commands> if and only if

a. <objectfile> does not exist

or

b. any of the files <dep1>, <dep2>, ... <depN> are newer than an existing <objectfile>

The compiler will execute <commands> from the directory in which the file containing the BUILD property resides.

Example

If the object file foo.o (foo.obj) is to be generated from the foo.c file in the external directory, but also depends on the foo.h file in the external directory, you could specify this as:

```
BUILD nti     '$$/foo.obj' 'external/foo.c' 'external/foo.h'
              'cl -c $1 -Fo$0 -nologo -w -O2 -Zd -Zp4'
      ppcmac  ':$$:foo.obj' ':external:foo.c' ':external:foo.h'
                 'MrC -D MAC -o $0 $1'
      default '$$/foo.o'   'external/foo.c' 'external/foo.h'
              '$CC -o $0 $1';
```

Notice, that regular environment variables may be used in the <commands> specification, e.g. in the default (UNIX) specification, the variable CC are used (on UNIX, this is always set to an appropriate value in the job-file).

More than one build pr. fragment

In general, more than one build pr. fragment will not work. The reason is that the meta-programming system combines all build directives into one directive (property). This means that:

```
BUILD sun4s 'cc1' default 'cc2';
BUILD sun4s 'cc3' default 'cc4'
```

means the same as:

```
BUILD sun4s 'cc1' default 'cc2' sun4s 'cc3' default 'cc4'
```

BUILD 'executes' all entries for a given plaform. This means that the 2 sun4s entries will be executed for sun4s. The 2 default entries will be executed for all other platforms.

If the following two build entries are used:

```
BUILD sun4s 'cc1' default 'cc2';
BUILD sgi   'cc3' default 'cc4'
```

ONLY cc1 will be executed for sun4s and ONLY cc3 for sgi, and the 2 default entries for all other platforms. This is probably not what is intedended: For sun4s you would expect cc1 and cc4 to be executed and sgi, cc2 and cc3.

To fix this, changes in the meaning of properties in MPS must be changed.

Summary: more than one build directive in a fragment file will work if they have the same structure with respect to platform.

Alternative

It would be easy to implementa build directives on the form:

```
BUILD1 sun4s 'cc1' default 'cc2';
BUILD2 sgi   'cc3' default 'cc4'
```

This will give different properties and the compiler can be made to handle this. It is, however, not an ideal solution.

8. 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.

8.1 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:

```
sun4s    SUN-4 (SPARC) running Solaris 2.x
hpux9pa  HP 9000/700 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 or Windows 95
ppcmac   Power   Macintosh 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.

8.2 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 ppcmac (Macintosh) machine, the above expands to ./ppcmac/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.

Appendix A. Semantic Errors and Warnings

Semantic error messages and warnings that may be reported by the compiler are listed in:

Semantic error messages and warnings

Appendix B. The BETA Grammar

An HTML interface to the BETA grammar can be found in

BETA Grammar.

Semantic Error Messages

Error 1:

```
Name is declared more than once
```

Error 2:

```
Name is not declared
```

Error 3:

```
Attribute is not declared
```

Error 4:

```
A pattern is expected here
```

Error 5:

```
An item is expected here
```

Error 6:

```
A repetition is expected here
```

Error 7:

```
A simple evaluation cannot be assigned
```

Error 8:

```
The lists have different lengths
```

Error 9:

```
The lists have different lengths
```

Error 10:

```
In "leave P" or " restart P", "P" must be an enclosing label
or enclosing pattern
```

Error 11:

```
Illegal assignment/comparison of value, reference or
repetition
```

Error 12:

```
Only a single name is alloved here
```

Error 13:

```
Attempt to bind V which is not virtual ( V ::T)
```

Error 14:

```
In V ::T, T does not have a correct qualification
```

Error 15:

```
An object is expected here
```

Error 16:

```
A basic pattern cannot be used as a super-pattern
```

Error 17:

```
A virtual pattern or a pattern defined as a descriptor slot
cannot
be used as super-pattern
```

Error 18:

```
A string of length 1 is a char - NOT a text
```

Error 19:

```
Illegal recursion in the definition of a pattern.
One of the following type of errors have occured:
(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(# ... #); ... #) ... #)
```

Error 20:

```
Incompatible qualifications in assignment/comparison
```

Error 21:

```
Only simple values or references may be compared
```

Error 22:

```
Only simple values may appear in unary expressions
```

Error 23:

```
Fatal error: virtual binding not found
```

Error 27:

```
The descriptor is both used as item and component
```

Error 28:

```
Static size of descriptor is larger than 32760 bytes
```

Error 29:

```
Illegal recursion in object-description
```

Error 30:

```
Illegal assignment to constant value/reference or repetition
```

Error 31:

```
Only pattern-declarations may appear in a fragment of
category 'attributes'
```

Error 32:

```
A virtual qualification must be a pattern name or a descriptor
```

Error 33:

```
A virtual pattern or descriptor-slot cannot be used as a
component
```

Error 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
```

Error 35:

```
An "external" can only have one exit parameter
```

Error 36:

```
A sub-pattern of "external" cannot be used as super-pattern
```

Error 37:

```
The DO-part of an "external" should be empty
```

Error 38:

```
A repetition/for-imp range must be an integer, char or
boolean evaluation
```

Error 39:

A simple pattern cannot be used here

Error 40:

Unkown inline primitive

Error 41:

The superpattern of this descriptor has no INNER

Error 42:

Attempt to bind a virtual in a descriptor with no superpattern

Error 43:

The qualification of a variable pattern must be a pattern

Error 44:

A pattern-, virtual-, variable-pattern, or reference is expected here

Error 45:

A repetition name is expected here

Error 46:

In "this(P)" or "inner P", P must be the name of an enclosing pattern

Error 47:

An unexpanded nonterminal must be a SLOT

Error 48:

A super-pattern must be a simple pattern or a simple pattern attribute of a static object

Error 49:

A simple pattern or virtual pattern cannot be assigned a structure reference

Error 50:

A structure reference can only be assigned to/compared with another structure reference

Error 51:

Only integer,char,boolean, real objects and references can be

```
compared in an if-imperative
```

Error 52:

```
Rename declaration has NOT been implemented
```

Error 53:

```
Syntax error in number
```

Error 54:

```
Name not declared. There is no corresponding virtual
declaration
```

Error 55:

```
A pattern with a do-part slot cannot be used as a
super-pattern
```

Error 56:

```
The QUA construct has not been implemented
```

Error 57:

```
A basic pattern like integer, real, char, boolean, false, and
true
cannot be used as a super-pattern
```

Error 58:

```
In a list being assigned to and being assigned from as in
...->(E1,E2,...En)-> ...
the elements may not be patterns
```

Error 59:

```
The enter-parameters of an external call must be supplied
```

Error 60:

```
The left-side of the assignment/comparison has no (exit-)list
or the right-side has no (enter-)list
```

Error 61:

```
An element of the left-side/right-side of the
assignment/comparison
has no (exit-)list or (enter-)list
```

Error 62:

```
The Left-side of the assignment/comparison has no (exit-)list
```

Error 63:

```
An element of the left-side of the assignment/comparison has
no (exit-)list
```

Error 64:

```
The right-side of the assignment/comparison has no
(enter-)list
```

Error 65:

```
An element of the right-side of the assignment/comparison has
no (enter-)list
```

Error 66:

```
A simple value (integer,boolean,char,real) cannot be
assigned/compared
to/with a list
```

Error 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 superflous "[]"
```

Error 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 superflous "[]"
```

Error 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 superflous "[]"
```

Error 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 superflous "[]"
```

Error 80:

```
"inner P" is only legal in the do-part of the pattern "P"
```

Error 81:

```
In a computed-remote, "(EV).X","EV" cannot be an
evaluation-list
```

Error 82:

```
In a computed-remote, "(EV).X", "EV" must have one
exit-element,
which must be a reference
```

Error 83:

```
In a computed-remote, "(EV).X", "EV" is not a legal evaluation
```

Error 84:

```
"Extend" and "new" must have an enter-parameter
```

Error 85:

```
"leave P" or "restart P", where "P" is a pattern,
is only legal in the do-part of "P"
```

Error 87:

```
A repetition index must be an integer-evaluation
```

Error 88:

```
The base of this number is too large
```

Error 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"
```

Error 90:

```
A subpattern of "data" may not have a do-part
```

Error 91:

```
A boolean evaluation is expected here
```

Error 92:

```
Primitive operation appears in wrong context
```

Error 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
```

Error 94:

```
A virtual pattern cannot be bound to a basic pattern like
integer, real, char, boolean, false, and true
```

Error 96:

```
In "X: ^.P", "Y: @.P", "" cannot be:
a repetition element as in "R[e].P"
a computed remote as in"(R).P"
It must be a static object
```

Error 98:

```
A sub-pattern of "external" must be defined as a pattern
```

Error 100:

```
In "V ::T", "T" must be a non-virtual pattern
```

Error 101:

```
In "V :: T", "T" must be a pattern
```

Error 102:

```
A cycle has been detected in the super-chain of the
virtual/final binding
```

Error 103:

```
Incompatible types of binary operator
```

Error 104:

```
Incompatible left- and right-side of assignment
```

Error 105:

```
Illegal assignment to constant, literal or expression
```

Error 107:

```
A virtual cannot be bound to a slot
```

Error 108:

```
Illegal use of the "
```

Error 110:

```
Illegal recursion in exit list:
a pattern is referred direcly or indirectly in its own exit
list
```

Error 111:

```
Illegal recursion in enter list:
a pattern is referred direcly or indirectly in its own enter
list
```

Error 112:

```
External entry point has a blank- or control chararacter
```

Error 113:

```
There is a circle in the super-pattern chain
```

Error 114:

```
Illegal operator "!"
```

Error 115:

```
Use of "@@" in combination with object executions in external
calls is insecure
```

Error 116:

```
A DATA object can only be a part-object of a COM object or a
DATA object
```

Error 117:

```
A COM virtual pattern cannot be a part object
```

Error 118:

```
"Object" may NOT be used to specify a virtual in a COM pattern
```

Error 119:

```
A virtual COM pattern is used somewhere else as a non-virtual
pattern-
```

Error 120:

```
A DATA object can only be used as a part object in a COM- or
DATA-object
```

Error 121:

```
Illegal recursion in computed remote like in:
( where
```

```
P: (# ... exit P #)
instead of
P: (# ... exit P[] #)
```

Error 122:

```
It is not possible to obtain a reference to a basic pattern
like integer, real, boolean and char
```

Error 123:

```
Underflow in number
```

Error 124:

```
Overflow in number
```

Semantic Warnings

Warning 24:

```
A run-time qualification check will be generated here
```

Warning 25:

```
Repetition of static components is not implemented
```

Warning 26:

```
Repetition of non simple patterns is not implemented
```

Warning 86:

```
"leave P" and "restart P", where "P" is a pattern,
are currently not allowed in internal descriptors of "P"
```

Warning 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.
```

Warning 97:

```
An "inner" in a singular object will never be executed
```

Warning 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
```

Warning 106:

```
Assignment/comparison between boolean and integer
```

Warning 109:

```
Text has a null-char. All chars after the null-char are
ignored
```

Warning 125:

```
Enter/exit-parts are NOT implemented for mainPart forms
```

Warning 126:

```
Int64 numbers are NOT implemented as arguments/return values
for external routines
```

Semantic Error- and Warning Messages

' Mjølner Informatics