An overview of the Mjølner BETA System*

Lars Bak,[†] Jørgen Lindskov Knudsen,[‡]
Ole Lehrmann Madsen[†‡], Claus Nørgaard[†],
Elmer Sandvad[†]

April 1991

*Presented at: *Conference on Software Engineering Environments,* 25-27 March 1991, Aberystwyth, Wales, Great Britain.

[†]Mjølner Informatics ApS, Sience Park Aarhus, Gustav Wiedsvej 10, DK-8000 Århus C, Denmark, Phone: +45 86 20 20 00, Fax: +45 86 20 12 22, E-mail: larsp@mjolner.dk, cn@mjolner.dk, ess@mjolner.dk

[‡]Computer Science Department, Aarhus University, Ny Munkegade 116, DK-8000 Århus C, Denmark, Phone: +45 86 12 71 88, Fax: + 45 86 13 57 25, E-mail: jlknudsen@daimi.aau.dk, olmadsen@daimi.aau.dk

Abstract

The Mjøner BETA System is an *integrated* and *interactive* programming environment with support for industrial object oriented programming. The Mjølner BETA System is a result of the Scandinavian research project Mjølner.

The integration of the various tools in the Mjølner BETA System is established by insisting that all tools in the system utilizes one single representation of the program. This representation is abstract syntax trees (ASTs). All manipulations of the ASTs by the various tools are done, utilizing the metaprogramming system, which defines an interface to the AST, and ways to manipulate the AST.

The Mjølner BETA System includes an implementation of the BETA programming language. In addition it includes a set of grammar-based tools, which can be used for any formal language that is defined by a context-free grammar. The grammar-based tools include a hyper structure editor, a metaprogramming system, and a fragment system. Finally, the system includes a system browser, a source level debugger, a graphics system, an user interface system, an application builder, a general interface to the underlying operating system and interface to external routines.

The BETA programming language is a block structured, strongly typed object oriented programming language. The language supports procedural, object oriented, concurrent and to some extent functional programming.

The hyper structure editor is an integrated text and syntax-directed editor with extensive facilities for abstract presentation, browsing, and hypertext.

The fragment system makes it possible to split a program into arbitrary modules called fragments. The fragment system supports separate compilation and separation of interface and implementation fragments.

The metaprogramming system defines a unique representation of programs in the form of a set of classes defined by the abstract syntax of the language. This makes it possible to write programs that manipulate other programs.

The Mjølner BETA System is grammar-based, implying that most components of the system exists in the form of tool generators, that given a context-free grammar for a language, will generate a language specific tool.

All tools in the Mjølner BETA System (including the compiler) are written in BETA (except the run-time system and a few other routines written in C and assembly language).

1 Introduction

The Mjølner BETA System is a highly integrated programming environment for object oriented programming. The objective is to support development of large, efficient industrial programs. The Mjølner BETA System is a result of the Scandinavian research project Mjølner.

The Mjølner BETA System includes an implementation of the BETA programming language [8, 7]. In addition it includes a set of grammar-based tools, which can be used for any formal language that is defined by a context-free grammar. The grammar-based tools include a hyper structure editor, a metaprogramming system, and a fragment system.

The BETA programming language is a block structured, strongly typed object oriented programming language. The language supports procedural, object oriented, concurrent and to some extent functional programming.

The hyper structure editor is an integrated text and syntax-directed editor with extensive facilities for abstract presentation, browsing, and hypertext. The hypertext facility combined with structure editing of documents makes the editor particularly well suited to support program documentation. Furthermore, an graphical extension to the hyper structure editor is currently being developed such that the editor can be used as an integrated diagram editor, too. The diagram editor will inherit all facilities from the hyper structure editor.

The fragment system makes it possible to split a program into axbitrary modules called fragments. This is used to share code in the form of libraries. In addition it is used to split a program into an interface part and an implementation part. For a class it is thus possible to separate the description of the interface from the description of the implementation. Since a fragment may have several different implementation parts, the fragment system also supports having several variants of a program.

The metaprogramming system defines a unique representation of programs in the form of a set of classes defined by the abstract syntax of the language. This makes it possible to write programs that manipulate other programs. In addition application programs may manipulate programs by means of this representation. The metaprogramming system has been designed to allow for "Lisp-like" representation of programs as data.

The system includes a system browser, a source level debugger, a graphics system, an user interface system, an application builder, a general interface to the underlying operating system and interface to external routines.

The Mjølner BETA System is grammar-based, implying that most components of the system exists in the form of tool generators, that given a context-free grammar for a language, will generate a language specific tool.

All tools in the Mjølner BETA System (including the compiler) are written in BETA (except the run-time system and a few other routines written in C and assembly language). The Mjølner BETA System is implemented under Macintosh and UNIX workstations. The Macintosh version is implemented under MacOS and MPW for MacII series work-stations. The UNIX versions uses the X Window System and is available for SUN3 series workstations, Apollo DN3500 series workstations and HP9000 series workstations.

2 Overview of the Mjølner BETA System

The Mjølner BETA System is an *integated* and *interactive* programming environment with support for industrial object oriented programming.

The integration of the various tools in the Mjølner BETA System is established by insisting that all tools in the system utilizes one single representation of the program. This representation is abstract syntax trees (ASTs). All manipulations of the ASTs by the various tools are done, utilizing the metaprogramniing system, which defines an interface to the AST, and ways to manipulate the AST. The overall structure of the Mjølner BETA System is illustrated in figure 1.

The Mjølner BETA System is based on the notion of *program fragment* (or just *fragment*). The notion of fragments is based on the context-free grammar for the programming language. In principle, any sentence derived from any nonterminal in the grammar may be a fragment. Non-terminals are the natural units of the syntax-directed editor. Fragments and nonterminals are the units of manipulation in major parts of the entire system, and the BETA compiler translates BETA fragments into native code. The system is also using fragments as a powerful notion of separate compilation, that enables the system to ensure full consistency across compilation units. This

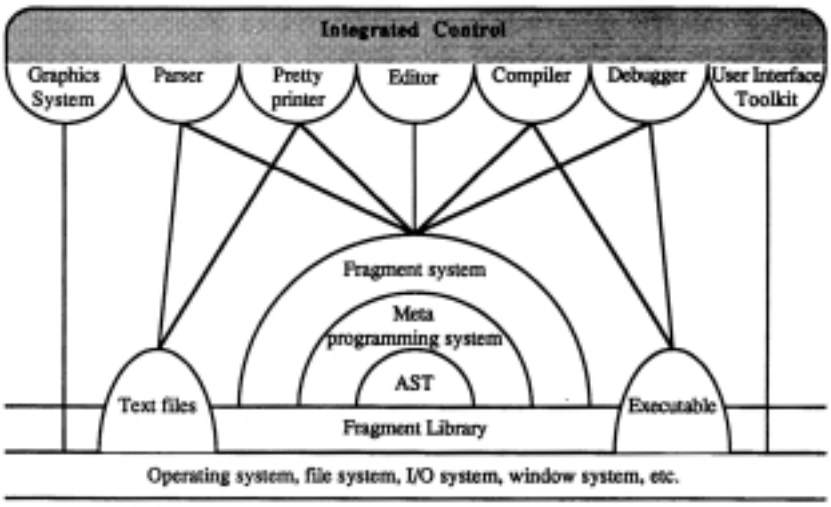


Figure 1: Mjølner BETA System overview

enables the programmer to make changes to the program, and the system will only compile or check the affected compilation units.

The hyper structure editor is a highly interactive tool for manipulating fragments. The user interface toolkit and graphics system is available to the application programmer in order to support the development of highly interactive and graphical applications by means of the Mjøner BETA System.

3 A Grammar-based System

Major parts of the system (e.g. editor, parser, pretty-printer, metaprogramming system, fragment system) are grammar-based in the sense that tool generators exists that given a specific grammar for a language will define a specific tool, that is able to manipulate programs written in that specific language. Such language specific tools have been generated for the BETA language, and form the basis for the Mjølner BETA System. Furthermore, the generators have been used to create tools for Simula, Modula, SQL, OSDL, Pascal, FelixPascal and others.

A variant of context-free grammars, called *structured context-free grammars*, are used for specifying the context-free syntax. Structured context-free

grammars are like ordinary BNF grammars except that productions must be one of:

```
<A>  ::= w0 <t1:A1> w1 ... <tn:  An> wn   construction rule
<A>  ::| <A1> w1 | <A2> |... <An>          alternation rule
<A>  ::* <B> w                             list zero rule
<A>  ::+ <B> w                              list one rule
<A>  ::?  <B> w                            optional rule
```

where `<A>` denotes nonterminals, `<t:B>` denotes nonterminals with a tag-name, and `w` denotes terminals. A construction rule specifies that the nonterminal on the left-hand side of the rule may be replaced by the string `w0 <t1:A1> w1 ... <tn:An> wn`. An alternation rule specifies that the nonterminal on the left-hand side of the rule may be replaced by either of the nonterminals `<A1>, ..., <An>` on the right-hand side. A list zero rule specifies that the nonterrninal on the left-hand side of the rule may be replaced by zero or more instances of the nonterminal on the right-hand side, separated by the string w (i.e. nothing, `<B>, <B> w <B>, <B> w <B> w <B>`, etc.). A list one rule is like the list zero rule, except that there must be at least one element in the list. An optional rule specifies that the nonterminal on the left-hand side of the rule may either be replaced with nothing or by the nonterminal on the right-hand side. A nonterminal may only appear once on the left-hand side of a production.

In the rest of this paper, we will use the BETA language as the vehicle for all examples and describe the grammar-based tools by describing the BETA specfic tools, generated from the tool generators. Nearly all that is said about these tools apply to all tools being generated from the tool generators. The exceptions are the static semantic parts and the integration with the BETA compiler. This implies, that in order to create a highly integrated environment based on these tool generators (like the BETA specific environment), one needs cooperation with the compiler.

An example of a BETA program is given in figure 2. The program defines three identifiers: `Private`, `Push` and `Pop`. `Private` is a static object (caused by the @), and `Push` and `Pop` are routines. `Push` takes one `integer` argument. Special grammar symbols are shown enclosed by `<<` and `>>` These symbols are called *placeholders*. Placeholders are always associated with a nonterminal of the grammar. A placeholder may have a tag-name in order to be able to distinguish between several instances of the same nonterminal in

a program. I.e. `<<Push:Descriptor>>` is a placeholder, where `Descriptor` specifies the syntactic category and `Push` is the tag-name.

```
(#
  Private: @<<SLOT Private:Descriptor>>;

  Push: (# e: @integer
        enter e
        do <<SLOT Push:Descriptor>>
        #);
  Pop: (# <<Attributes>> <<...ActionPart...>> #);
  <<Attributes>>
#)
```

Figure 2: Example of a BETA program

Generally, programs may contain placeholders of three different types: *nonterminals, slots* or *contractions*. Nonterminals and slots denote unexpanded nonterminals of the underlying grammar, whereas contractions denote expanded nonterminals.

Nonterminals (e.g. `<<Attributes>>`) are indications that these parts of the program have not been specified yet. The ability to handle nonterminals in a program is the means for allowing syntax-directed editing.

Slots (e.g. `<<SLOT Push:Descriptor>>`) are indications of parts of the program that deliberately have been kept open. Slots are the means for modularization of a program in order to support information hiding and separate compilation. The program parts to by located in slots will reside in another fragment. The fragment system for BETA (which takes care of slots) will be described in section 4.

Contractions (e.g. `<<...ActionPart...>>`) are placeholders indicating that this part of the program, derived from the nonterminal, is not shown. I.e. contractions are a means for suppressing details that are present in the program. Note that contractions may suppress other placeholders, i.e. within a contraction, nonterminals, slots and other contractions may reside.

4 The Fragment System

The foundation of the fragment system is syntaxdirected program modularization as described in [6]. Syntax-directed program modularization is a

very general principle for program modularization. The fragment system is a concrete, but limited implementation of syntax-directed program modularization, applied to the BETA programming language. The aim of the fragment system is to support modularization, information hiding and separate compilation of programs.

The basic level of the environment is the fragment library, that is the storage system for fragments. The fragment system is language independent implying that modularization, information hiding and separate compilation need not be defined as part of the programming language. The fragment system supports:

Modularization and Information Hiding: The fragment system enables the programmer to define the granularity of modularization and information hiding that suits his particular problem. Fragments are the modules of the Mjølner BETA System and fragments are any legal derivation, implying that both coarse- and fine-grained modularization is possible.

Separation of the interface and the implementation: The programmer may define fragments containing the interface definition of a program part and other fragments decking the implementation part. One implication hereof is that the interface of a class may be separated from its implementation.

Variant control: There may be several implementation fragments corresponding to, say, a class specification. This facility may be used to support variants of a program.

Separate compilation: Fragments are the basic entities handled by the compiler. Fragments may be separately compiled. When a fragment is compiled, all fragments it is depending on will be checked for modifications in the source code since last compilation of these fragments, and if such changes have been made, these fragments will be automatically recompiled.

Code sharing: In general the mechanism is useful for splitting programs into parts that may be shared by several other program parts. This is a useful and orthogonal feature to the class/sub-class mechanism.

4.1 Fragments

The modularization language is called the *fragment language*, since it describes the organization of programs in terms of *fragments*. (The notion of fragment will be introduced below.) The fragment language is used for communicating with the fragment system, which is the component of the Mjølner BETA System that handles storing and manipulation of fragments. The terms fragment language and fragment system are used interchangeably when this causes no confusion.

The fragment language is independent of the language that is used for specifying the programs that are manipulated by the fragment system. The principles behind the fragment language can be used to describe modularization of most programming languages. The fragment language is *grammar-based*. The idea is that any correct sequence of terminal and nonterminal symbols defined by the grammar is a legal module (i.e. fragment). The fragment language describes how such strings may be combined into larger strings. The fragment language is presented here using a graphical syntax, though the fragment language also has a textual syntax which is currently used by the Mjølner BETA System. A future version of the Mjølner BETA System will include support for a graphical syntax like the one used in this paper.[1]

4.1.1 Forms

A string of terminal and nonterminal symbols derived from a nonterminal `A` is called an `A`-*form*[2] or sometimes just a *form*. The derived strings in figure 3 are all examples of forms.

Forms are the basic elements used to define modules in the Mjølner BETA System. Consider e.g. the forms 2 and 3 in figure 3. By substituting the `DoPart` nonterminal of form 2 by form 3 we get the form in figure 4.

[1] In addition to the use of graphical syntax, the fragment language described in this paper is slightly more general than the actual implementation. For details, see Mjølner BETA System manuals.

[2] In formal language theory, this is called a *sentential form*.

	Nonterminal	Derived string
1.	<<Attributes>>	P: (# a,b: @char #);
2.	<<Descriptor>>	(# a,b,c: @char enter(a,b) <<DoPart>> exit c #)
3.	<<DoPart>>	do a*b->c

Figure 3: Nonterminals and corresponding derived forms

```
Foo: (# a,b,c: @char
     enter(a,b,c)
     do a*b->c
     exit c
     #)
```

Figure 4: The FOO form

4.1.2 Slots

In the Mjøner BETA System, several tools manipulate forms, but not all nonterminals are necessarily to be used by the fragment system. The nonterminals used by the fragment language are the *slots* since they define openings where other forms may be inserted.

Forms may contain slots, and the fragment langage contains constructs for binding slot names with forms, thus combining forms into composite forms and eventually complete programs.

Slot names and program names belong to different languages. There is thus no possibility of confusing program names and slot names. In figure 5 there is a routine called `Push` and a slot called `Push`. As we shall see later it is convenient to use identical names in this manner.

4.1.3 Fragment Form

In the fragment language, each form must be given a name and its syntactic category must be specified. A *fragment form* is a form associated with a name and a syntactic category. Figure 5 shows a fragment form. `Counter` is the name of the fragment form, `Attributes` is the syntactic category, and `Counter: (# ... #)` is a form (i.e. a string of terminal and nonterminal

symbols derived from `Attributes`).

```
<<form Counter:Attributes>>
Counter: (#
  Up: (# n: @ Integer
       enter n
         <<SLOT Up:DoPart>>
       #);
  Down: (# n: @ Integer
           <<SLOT Down:DoPart>>
         exit n
         #)
#)
```

Figure 5: `Counter` fragment form

4.1.4 Fragment Group

It is often convenient to defme a set of logically related fragment forms together. For this purpose it is possible to define a group of fragments, called a *fragment group*.[3] The syntax of a fragment group is shown in figure 6. It defines two fragment forms. The name of fragment forms are `Up` and `Down`, both with syntactic categories `DoPart` and the actual fragment forms are do `n + 7 -> n` and do `n - 5 -> n`.

```
<<form Up:DoPart>>
do n+7->n
<<form Down:DoPart>>
do n-5->n
```

Figure 6: A fragment group

4.1.5 Fragment Library

The fragment system handles the storing of fragments in a library, called *The fragment library*. The fragment library is usually implemented on top of a file system or a data base system. The fragment language refers to fragments

[3]The term *fragment* will be used to refer to either a fragment form or a fragment group. No confusion should be possible

stored in the fragment library. A fragment resides in a specific location in the fragment library. Fragments are named using a hierarchical naming scheme in the style of UNIX or Macintosh file systems. The location of a fragment is given by means of a hierarchical name. The name `/home/smith/Counter` denotes a fragment `Counter`. `Counter` resides in the directory `/home/smith`. In the following examples, the location of a fragment will often be given together with the definition of the fragment as shown in figure 7.

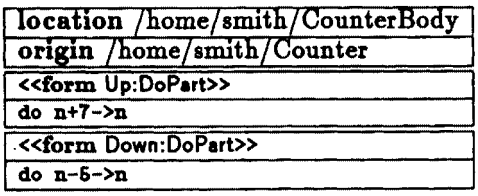


Figure 7: The `Counterbody` group

4.1.6 Origin of a Fragment

The *origin part* of a fragment specifies a fragment that is used when binding fragment forms to slots. Consider figure 7. The origin of `CounterBody` is the fragment `/home/smith/Counter` (c.f. figure 7). The origin must have free slots corresponding to `Up` and `Down`. The origin construct specifies that the fragment forms `Up` and `Down` are substituted for the corresponding slots in `Counter`. The result of this substitution is a form, called the *extent* of the fragment. The extent of `Counter` is shown in figure 8.

```
Counter: (#
  Up: (# n: @ Integer
      enter n
      do n+7->n
      #);
  Down: (# n: @ Integer
        do n-5->n
        exit n
        #)
#)
```

Figure 8: Extent of `Counterbody`

Extent

A fragment defines a unique form, called the *extent* of the fragment. The extent of the above fragment is a combination of `CounterBody` and `Counter`. The combination is obtained by filling in the slots in the origin with the corresponding fragment forms.

4.1.7 The Basic Environment

The Mjølner BETA System provides a basic environment that defines the most important standard patterns and objects. In addition, this environment initiates and terminates the execution of enay BETA program. The basic BETA environment is the fragment `betaenv`[4] shown in figure 9. This fragment defines a number of standard patterns, In addition, the fragment has two slots: /textttProgram and `Lib`.

```
location betaenv
<<form betaenv:Descriptor>>
{*** The basic BETA environment ***}
(# Put: (# ch: @ Char enter ch ... #);
   PutInt: (# n: @Integer enter n do ... #);
   PutText: (# T: @Text enter T do ...#);
   NewLine: (# ... #);
   PutLine: (# T:@Text enter T do T->puttext; newLine #);
   Text: ...;
   File: ...;
   Integer: (# ... #);
   Char: (# ... #);
   ... {Definition of other standard attributes}
   <<SLOT Lib: Attributes>>
do {Initialize for execution}
   <<SLOT Program:Descriptor>>;
   {Terminate execution}
#)
```

Figure 9: The basic BETA environment.

[4]In the rest of this paper, simple names (i.e. without any directory specified) are used for specifying locations of fragments

Simple programs

A complete BETA program that makes use of `betaenv` may be defined by specifying the `Program` slot. The fragment form in figure 10 is an example of a very simple BETA program.

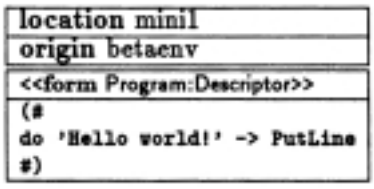


Figure 10: The basic BETA environment.

The extent of the fragment `mini1` is shown in figure 11. The `Program` fragment has been substituted for the `Program` slot in `betaenv`. In the `Program` fragment it is therefore possible to use any name which is visible at the point of the `Program` slot in `betaenv`. `PutLine` is visible at the `Program` slot and is therefore visible in the `Program` fragment. It would also have been posible to make use of patterns like `Integer, Char, Text` etc.

```
{*** The basic BETA environment ***}
(# ...
   PutLine:...
   ...
do {Initialize for execution}
   (#
   do 'Hello world!' -> PutLine
   #)
   {Terminate execution}
#)
```

Figure 11: Extent of `mini1`

Simple libraries

The `Lib` slot in betaenv is intended for making a set of general patterns to be used by other programs. The difference between such a library and a program is that the library is a list of patterns whereas the program is a single object descriptor. Figure 12 contains an example of a library consisting of

two patterns. By substituting the `Lib` slot in `betaenv` with the `Lib` fragment form, we obtain figure 13.

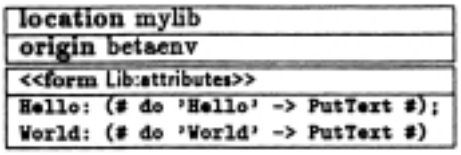


Figure 12: `mylib` library

```
{*** The basic BETA environment ***}
(# ...
   Hello: (# do 'Hello' -> PutText #);
   World: (# do 'World' -> PutText #)
do {Initialize for execution}
   <<SLOT Program:Descriptor>>;
   {Terminate execution}
#)
```

Figure 13: Extent of `mylib`

The library patterns are inserted at the point of the `Lib` slot. This means that in the `Lib` fragment form it is possible to see all names visible at the point of the `Lib` slot in `betaenv`. Note that the extent of `mylib` is not an executable program, since the `Program` slot has not been defined.

4.1.8 Include

When making libraries like `mylib`, we need a mechanism for combining several fragments into one fragment. The **include** construct makes this possible. Figure 14 contains a program that makes use of the library `mylib`.

The effect of **include** `mylib` is that the patterns defined in `myLib` can be used in the `Program` fragment form. Formally, the fragment forms of `mylib` become part of the fragment `mini2`. In the above example, `mini2` may be understood as a fragment group consisting of the fragment forms in `mylib` and the `Program` fragment form. This implies that the extent of `mini2` is obtained by substituting the `Lib` slot in `betaenv` by the `Lib` fragment form in `mylib` and by substituting the `Program` slot in `betaenv` by the `Program` fragment form in `mini2`. This gives the form in figure 15.

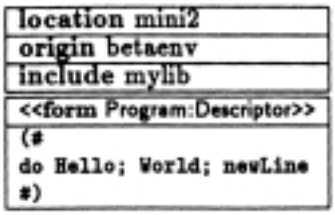


Figure 14: `mini2` program

```
{*** The basic BETA environment ***}
(# ...
   Hello: (# do 'Hello' -> PutText #);
   World: (# do 'World' -> PutText #)
do {Initialize for execution}
   (#
   do Hello; World; newLine
   #)
   {Terminate execution}
#)
```

Figure 15: Extent of `mini2`

Since the patterns in `mylib` are inserted at the point of the `Lib` slot, they are visible at the point of the `Program` slot. This is where the `Program` fragment form in `mini2` is inserted. I.e. the patterns `Hello` and `World` are visible inside the `Program` fragment form.

A fragment form may have more than one include. This makes it possible to use several library fragments in the same fragment (c.f. section 3.4).

4.1.9 Body

When defining a fragment it is often desirable to be able to specify one or more fragments that always must be included when using the fragment. This is often the case when a fragment is separated into an interface fragment and one or more implementation fragments. Here we introduce the construct for specifying this; but delay further explanation until section 3.2. The **body** construct specifies a fragment that is always part of the extent. Consider figure 16. The counter fragment has a body specification that specifies that a fragment called `counterbody` is always part of the extent of `counter`.

The `counterbody` fragment could be described as in figure 17. The counter fragment could be used as illustrated in figure 18.

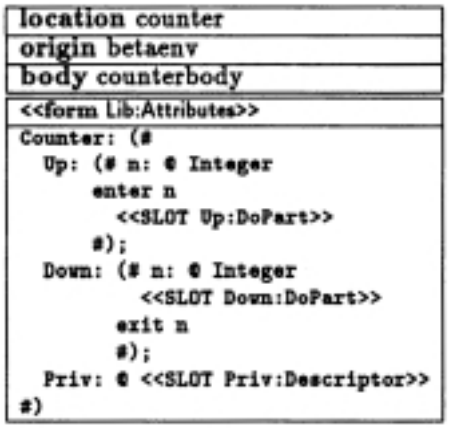


Figure 16: Using body

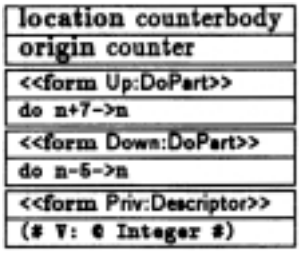


Figure 17: A `body` fragment

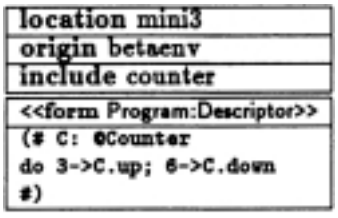
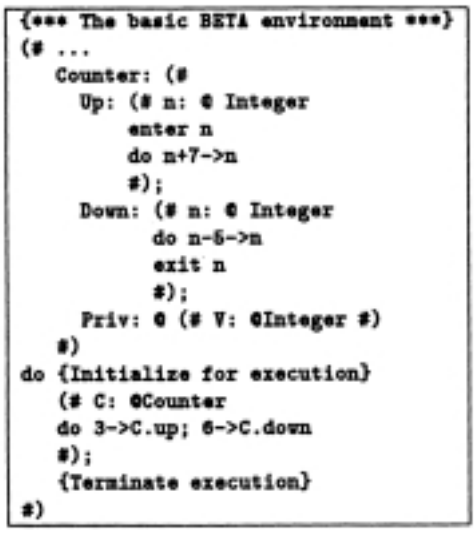


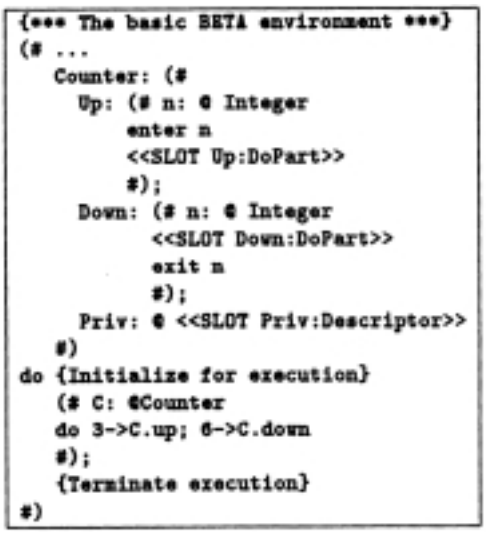
Figure 18: `mini3` program

The extent of `mini3` is obtained by combining the `Program` fragment form in `mini3`, **origin** `betaenv` and **include** `counter`. In addition, the



```
{*** The basic BETA environment ***}
(# ...
   Counter: (#
     Up: (# n: @ Integer
          enter n
          do n+7->n
          #);
     Down: (# n: @ Integer
            do n-5->n
            exit n
            #);
      Priv: @ (# V: @Integer #)
   #)
do {Initialize for execution}
   (# C: @Counter
   do 3->C.up; 6->C.down
   #);
   {Terminate execution}
#)
```

Figure 19: Extent of `mini3`



```
{*** The basic BETA environment ***}
(# ...
   Counter: (#
     Up: (# n: @ Integer
          enter n
          <<SLOT Up:DoPart>>
          #);
     Down: (# n: @ Integer
            <<SLOT Down:DoPart>>
            exit n
            #);
      Priv: @ <<SLOT Priv:Descriptor>>
   #)
do {Initialize for execution}
   (# C: @Counter
   do 3->C.up; 6->C.down
   #);
   {Terminate execution}
#)
```

Figure 20: Domain of `mini3`

body `counterbody` in `counter` implies that the `counterBody` fragment is also included in the extent. The resulting form looks as in figure 19.

As stated earlier, the patterns defined in `Lib` are visible in `mini3` through the use of **include**. However, the `counterbody` is not visible from

`mini3`. This means that an evaluation like `C.Priv.V+1->C.Prim.V` is not possible within `mini3`. That is even if the extent of `mini3` includes the `counterbody` fragment, it is not visible within `mini3`.

Domain

The *domain* of a fragment `F` is the part of the extent of `F` which is visible within `F`. The domain of `F` consists of the fragment forms in `F`, plus the domain of the origin of `F` plus the domain of possible included fragments. The domain of `mini3` is the form shown in figure 20. The domain of `mini3` is constructed as follows:

- The domain of `mini3` consists of the `Program` fragment form in `mini3` plus the domain of `betaenv` (its origin), plus the domain of the included fragment `counter` (c.f. figure 20)
- The domain of `betaenv` is the form in figure 9.
- The domain of the `counter` fragment consists of the form defining the pattern `Counter`, plus the domain of `betaenv`. Note that the **body** part of `Counter` does not contribute to the domain.

4.2 Separation of Interface and Implementation

Encapsulation and separation or interface and implementation saves compilation time. In the Mjølner BETA System, as in many other systems, fragments (modules) can be separately compiled. A change in an implementation module can then be made without recompilation of the interface module and modules using the interface module. This can yield significant savings in compilation time. On the other hand, a change in an interface module implies that all modules using it must be recompiled. This can be extremely time consuming. The fragment system manages these dependencies between fragments automatically and the BETA compiler utilizes this information to reduce recompilations to a minimum.

Programming takes place at different abstraction levels. The interface part of a module describes a view of objects and patterns meaningful at the abstraction level where the module is used. The implementation level

describes how objects and patterns at the interface level are realized using other objects and patterns.

The fragment language supports encapsulation and separation of interface and implementation. One fragment defines the interface while others define the implementation. The `Counter` fragment in figure 16 is an example of one such interface fragment, and the `CounterBody` fragment in figure 17 is an example of an implementation fragment. The fragment system ensures (in cooperation with the compiler), that the fragment `mini3` in figure 18 cannot utilize the information located in the implementation fragment (`CounterBody`).

4.2.1 Abstract Data Types

One of the fundamental concepts in program development is the notion of *abstract data type*. In the context of BETA, an abstract data type is a class pattern whose instances are completely characterized by a set of (procedure) pattern attributes — sometimes referred to as its operations. These operations constitute the outside view of the objects whereas reference attributes and details of the pattern attributes belong to the inside view (the implementation).

The fragments in figure 21 and figure 23 shows an example of an abstract data type in BETA. The fragments define the interface and implementation of a stack of text references. A stack is completely characterized by its operations Push, Pop, New and isEmpty. The `stack` may be used as shown in figure 22. Note, that `libuser2` is not a complete program, since no implementations for `stack` are specified yet.

Since the domain of `stack` does not include its implementation, the stack can only be used by means of its operations. It is good practice to define most class patterns as "abstract data types", i.e. restrict their interface to be pattern operations. In some languages, e.g. Smalltalk, class patterns are always abstract data types.

4.3 Alternative Implementations

It is possible to have several implementations of a given interface module. In general this means that different fragments may define different bindings for

```
location stack
origin betaenv
<<form Lib: Attributes>>
Stack:
  (# Priv: @<<SLOT Priv:Descriptor>>;
     Push:
       (# e: ^ Text
       enter e[]
       <<SLOT Push: DoPart>>
       #);
     Pop:
       (# e: ^ Text
       <<SLOT Pop: DoPart>>
       exit e[]
       #);
     New: (# <<SLOT New: DoPart>> #);
     isEmpty:
       (# Result: @ Boolean
       <<SLOT isEmpty: DoPart>>
       exit Result
       #)
  #)
```

Figure 21: The interface part of pattern `Stack`

slots in a given fragment.

Suppose that we want to define an alternate implementation of the `stack` from the previous section. In the alternate implementation, `stack` objects are represented as linked lists. The list implementation is shown in figure 24.

Selecting the proper `stack` implementation is done by means of a **body** specification as illustrated in figure 26. Naturally, this **body** specification could have been given in `libuser2`, but then would all `libuser2` programs be using the `arraystack` implementation.

4.4 Using Several Libraries

The examples of libraries until now have only shown how to use one library from a program. The syntactic category of a slot like `<<SLOT Lib: attributes>>` describes a list of declarations. It is thus possible to bind an arbitrary number of `Lib` fragments to such a slot. Figure 25 shows a fragment that includes two libraries.

```
location libuser2
origin betaenv
include stack
<<form program:Descriptor>>
(# T: @Text; S: @Stack
do  'To be or not to be' -> T;
    T.reset;
    Get:
      cycle
      (# Ti: ^ text
      do &Text[] -> Ti[];
         T.getText-> Ti;
         (if Ti.empty // True then
            leave Get if);
         Ti[] -> S.push
      #);
   Print:
     cycle
     (# Ti: ^ Text
     do (if S.isEmpty // true then
           leave Print if);
        S.pop -> Ti[];
        Ti -> puttext; ' ' -> put
     #)
#)
```

Figure 22: A fragment using the `stack` interface

```
location arraystack
origin stack
<<form Priv:Descriptor>>
(# A: [100] ^ Text;
   Top : @integer
#)
<<form Push:DoPart>>
do Priv.top+1 -> Priv.top;
   e[] -> Priv.A[Priv.top][]
<<form Pop:DoPart>>
do Priv.A[Priv.top][] -> e[];
   Priv.top-1 -> Priv.top
<<form new:DoPart>>
do 0->Priv.top
<<form isEmpty: DoPart>>
do (0 = Priv.Top) -> result
```

Figure 23: Array implementation of `Stack`

```
location listStack
origin stack
<<form Priv:Descriptor>>
(# head: ^ elm;
   elm: (# T:^ Text; next: ^ elm #)
#)
<<form Push:DoPart>>
do (# R: ^ Priv.elm
   do &Priv.elm[] -> R[];
      Priv.head[] -> R.next[];
      e[] -> R.T[];
      R[] -> Priv.head[];
   #)
<<form Pop:DoPart>>
do Priv.head.T[] -> e[];
   Priv.head.next[] -> Priv.head[];
<<form new:DoPart>>
do NONE -> Priv.head[]
<<form isEmpty:DoPart>>
do (NONE=Priv.head[]) -> result;
```

Figure 24: List implementation of `stack`

```
location libuser3
origin betaenv
include stack
include counter
body liststack
<<form program:Descriptor>>
(# S: @ Stack;
   C: @ Counter;
do 3 -> C.up;
   'Top of stack' -> S.push;
   ...
#)
```

Figure 25: Example of a fragment using more than one fragment

```
location libuser4
origin libuser2
body arrayStack
```

Figure 26: A complete `stack` program using the array implementation

4.5 Program Variants

Often several variants of a given program are needed. This is usually the case if variants of a given program have to exist for several computers. The major

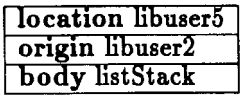


Figure 27: A complete `stack` program using the list implementation

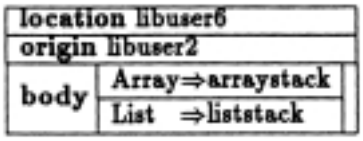


Figure 28: A complete `stack` program with two compile-time variants

part of the program is often the same for each computer. For maintenance purposes it is highly desirable to have only one version of the common part. In the Mjølner BETA System, program variants can be handled in the same way as alternative implementations of a module are handled. That is, different variants of a module bind some of the slots differently (as illustrated in figure 26 and 27).

However, often one wants to specify that a given fragment can have several variants (i.e. alternative **body** specifications), and postpone the decision on which variant to select until compile-time. This is done by a special **body** specification (as shown in figure 28). When compiling a program, it is possible to specify a set of tokens (e.g. `Array`) and the compiler will then select those bodies, that correspond to the specified tokens (e.g. `arraystack` for token `Array`). Program variants can in this way be maintained and produced efficiently.

4.6 Current implementation

The most important limitations of the present implementation of the fragment system in Mjøner BETA System are:

It does not support all nonterminals of the BETA grammar. Only two nonterminals of the BETA grammar are supported, namely `<Attributes>` and `<Descriptor>`.[5]

The naming scheme for fragments is a direct reflection of the file sys-

[5]This restriction is not a serious limitation in practice

tem, such that locations are spectications of files in the underlying hierarchical file system.

The selective **body** directive is currently only implemented for machine dependent tokens like `Sun3, HP, Mac`, etc. with the compiler automatically selecting the proper variant (identical to the machine type the compiler is running) unless the compiler is asked to cross-compile to another machine type.

It is important to note, that the fragment system in the current Mjøner BETA System implementation is able to handle large program, and supports the separate compilation of these large systems. The best proof of this is that the entire Mjølner BETA System is managed using the fragment system. The system consists of more than 1.000 fragments, distributed in more than 220 fragment groups, with a total of more 64.000 lines of BETA code. The fragment system is used in the Mjølner BETA System implementation to enable the automatic administration of machine specific fragments for four different target architectures on the same physical file system.

A full description of the fragment system as implemented in Mjølner BETA System can be found in the Mjøner BETA System manuals.

5 The Hyper Structure Editor

The editor has the following characteristics:

Structure and text editing: The syntax-directed editor is fully integrated with a text editor, allowing the user to switch freely between structure and text editing.

Incremental Parsing: The editor makes use of an incremental parsing algorithm when switching from textual editing to syntax-directed editing in order to ensure that the text being edited (by text editing) is syntactically valid within the given context. Only the edited text needs to be considered by the incremental parser to ensure this consistency.

Adaptive pretty-printing: The internal representation of a program is an abstract syntax tree (AST) as defined by the Mjølner BETA System. Pretty-printing, i.e. textual presentation of the AST is done adaptively,

which means that as much as possible is printed on each line in the editing window, The adaptive pretty-printing can be controlled by the user by tailoring the editor to conform with his needs.

Hyper Linking: The editor supports various types of hypertext facilities. These facilities include program semantical links, links to comments, links to other program fragments, links to text documents, etc. The linking facilities are fully symmetric.

Annotation: Comments in a program are handled as links to text objects. Instead of mixing comments with the program text, comments are presented and manipulated in separate text editing windows. Comments may also be included directly in the program text in the form of ordinary program comments.

Abstract presentation and browsing: Abstract presentation is provided by supporting placeholders of type contraction. Abstract presentation is similar to the ability to compress a whole sentence into one word (holophrastic). Abstract presentation has two applications: browsing and documentation. Browsing is supported by letting the user selectively go into further details of contractions. Printing a program at different abstraction levels provides good documentation facilities for e.g. functional specifications.

Grammar-based: The editor is grammar-based, which means that it may support any language that can be described by means of a context-free grammar.

Metaprogramming system: The editor is built upon the metaprogramming system, which is available for the user (see section 6). The user has the possibility to program her own metaprogramming tools. Due to a high degree of tailorability in the editor, it is possible to integrate such tools with the editor or simply to add functionality to the editor. This tailorability is available in the editor by providing special hooks to be expanded by the user and in general by the object oriented implementation language (BETA) of the editor [11].

The editor makes use of almost all other tools in the system: the BETA compiler, the pretty-printer, the parser, the user interface toolkit, the metaprogramming system, and the fragment system.

5.1 Syntax-directed Editing

Syntax-directed editing is supported through placeholders of kind nonterminals. During the editing of a program, nonterminals may appear, either specified explicitly by the user or as a result of another syntax-directed editing command. When a nonterminal is selected, the editor offers the possible derivations of that nonterminal as templates (i.e. the right-hand sides of all productions with the selected nonterminal on left side). This is done in a pop-up menu. If one of these productions are selected, the editor replaces the nonterminal with the right-hand of the selected production and editing may continue by selecting nonterminals in this template or by other editing tasks.

If a terminal in the program is selected, the smallest sentential form containing this terminal is selected. If a sentential form is selected, it may be deleted (i.e. replaced with the nonterminal from which it was originally derived). Special treatment is offered far nonterminals that are defined by optional or list zero productions.

5.2 Text Editing

Textual editing can be performed at any time instead of structural editing. Any sentential form can be selected for text editing, and when the textual editing of the sentential form is completed, the modified text is parsed according to the syntactic category of the sentential form. The parsing of the modified text is done using incremental parsing techniques.

5.3 Hyper Structure Editing

The editor supports hyper structure editing in four different ways:

Abstract Presentation: Most non-trivial fragments are normally too big to fit into a window on the screen, even if the window occupies the whole screen space. Abstract presentation can be considered as supporting intrafragment organizational links. The user has the possibility manually to substitute any structure in the fragment by a contraction, which acts as a link to the suppressed details. Abstract presentation of

a program fragment or a documentation fragment has several advantages:

Overview: ; It provides an overview of the document. The whole document can be surveyed at once in one window without scrolling through pages of text. This facility is also known in some word processing systems as outlining.

Browsing: Browsing is done by interactively detailing parts of an abstract presentation. If the document is a technical report with chapters and sections and the like, the highest abstraction level can actually be an interactive table of contents.

Documentation: Snapshots of a program at different abstraction levels can be very useful for documentation purposes. The user can select an appropriate abstraction level by detailing or abstracting the relevant constructs of the document and save the actual abstraction level including comments on textual form.

Annotations: Comments in a program are handled by means of links to simple text objects, so-called annotations. Any point in the program can be linked to a text object. If a construct in a program fragment is selected, a text window can be opened and the annotation can be entered. After finishing the annotation a special annotation mark ($*$) is inserted in the construct to indicate a link from the construct to a text object. Whenever the user selects a program construct with a annotation link, a text window can be activated (e.g. by double clicking with the mouse) and the annotation can be read or modified. Annotations may also be included directly in the program text in the form of ordinary program comments.

Program Semantical Linkage: The program semantical links are used to reflect the static semantic information of a program. For example definition-use relationships and super-/subclass relationships. Such relationships are automatically deductible from the program. In the Mjøner BETA System, program semantical links are set up by the checker. These links are used in the checking and coding processes, but are also available to the user in the editor. When a construct is selected in a program fragment a menu presents the available links from

that construct (if any). Note that program semantical relationships go across the fragment structure.

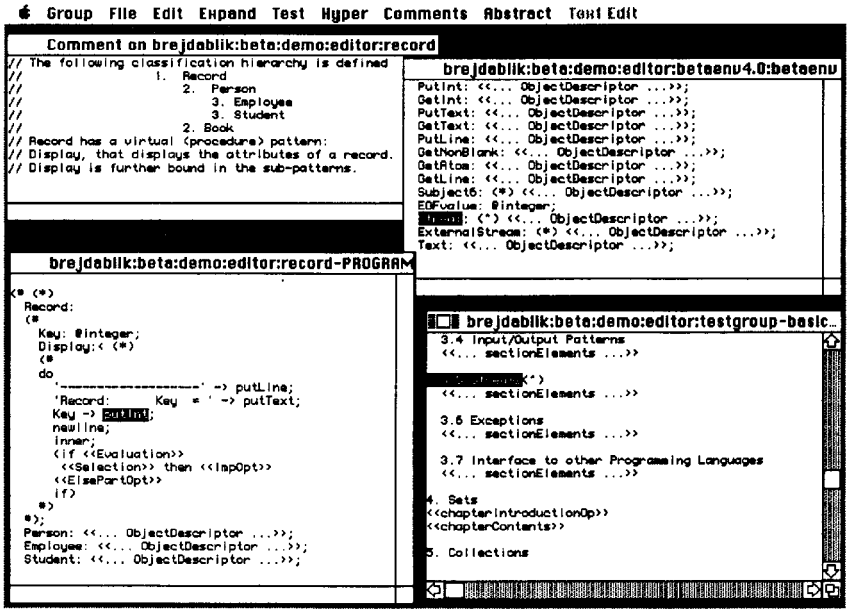


Figure 29: Editor User Interface

Interactive program analysis is normally not considered being part of program documentation, but language specific inspection of a program is often useful when trying to understand it. This kind of program traversal can be considered as non-hierarchical browsing.

Documentation Linkage: Documentation links are used to support all other kinds of relationships between documentation fragments mutually and between documentation fragments and program fragments. This link type is manually created by the user. The documentation link type is the basic mechanism for supporting integration of program and documentation. Any point in a program or fragment can be linked to another point in the same or another fragment. When a construct in a fragment is selected, the construct can be marked as a link source. The link destination is chosen by selecting another construct in the same or another fragment (possibly after activating an editor instance on the destination fragment) and then make it the link destination. A

link mark (^) is inserted in the source construct as well as the destination construct. A descriptive text can be associated with either end of the link. The hypertext facilities of the editor is based on the ideas presented in [13].

Figure 29 illustrates these linkage facilities. The lower left window is an editor window, showing the program being edited. The upper left window shows the comment annotation that is indicated just before `Record` in the editor window. Having followed the program semantic links from `putInt` in the editor window, have resulted in the upper right window, showing part of the basic BETA library. `PutInt` is defined in the first line shown (which was automatically selected when the `putInt` link was followed). In the upper right window, the hyper link shown at `Stream` has been followed, resulting in a document editor being opened in the lower right window, where the destination link is shown immediately after `Stream` in the text. Notice that the lower right window is a structure editor on the documentation. The structure editor for documentation is based on the hyper structure editor with the facilities for abstract presentation (which in this case will work like a outline processor), browsing, linking, etc.

With the graphical extensions currently being developed (and discussed shortly in section 7.8, the Mjølner BETA System will support an integrated editor for text, programs and documentation with support for graphical, structure, and text editing of all three types of documents, and with full support for linkage between internals of all three document types. Furthermore, the editor supports abstract presentation, browsing and annotation of all three document types.

5.4 Tailorability

The editor is implemented in BETA and the advanced user is supposed to have programming experience in BETA. If the user wishes to make "real" extensions to the editor, some knowledge of the metaprogramming system and the BETA user interface toolkit is required.

The tailorability of the editor is obtained mainly by applying object oriented design principles extensively throughout the entire editor. In fact,

the editor is designed as a class, and major tailorability is often made by creating a subclass of the editor and specify the tailoring in the subclass. The BETA editor is an example hereof, see section 5.5. For a more detailed discussion of tailorability of the Mjølner BETA System, see [11].

5.5 The BETA Editor

The BETA editor is an example of extensive tailoring of the general structure editor in order to fully support handling BETA programs. The BETA specific extension of the hyper structure editor provides the following additional facilities:

Automatic abstract presentation of BETA programs: When editor instances are activated with BETA program fragments, these are presented abstractly. The abstraction levels are object descriptors, attribute lists and imperative lists. Contractions explicitly defined within the AST are preserved across editing sessions.

Integration with the BETA compiler: Each BETA editor instance is able to activate the BETA compiler and to activate the resulting executable BETA program. Notice that the compiler does not have to perform lexical and syntactical analysis but uses the AST that the editor has produced. When the compiler is activated, the current fragment is marked as changed. The whole fragment group, which this fragment is a part of, is "delivered" to the compiler. If there are static semantic errors, the user can browse through these.

Simple static program analysis: The checker part of the BETA compiler sets up static semantic information in the AST, (e.g. references from name applications to the corresponding declaration). This information can be used from the editor by means of the program semantic linkage facilities. E.g. if a name application is selected, the corresponding name declaration can be found. If the name declaration is located in another program fragment, the editor is able to open another editor instance on it.

A more thorough description of the hyper structure editor can be found in the Mjølner BETA System manuals.

6 The Metaprogramming System

One of the objectives of the Mjøner project has been to obtain some of the flexibility of Lisp environments. One of the most significant features of Lisp is the fact that Lisp functions are represented and manipulated as data. This makes Lisp an excellent language for writing programs that manipulate other programs, i.e. metaprograms. In a program environment it is essential to have strong support for metaprogramming.

All metaprogramming tools in the Mjølner BETA System manipulate programs through a common representation that is abstract syntax trees (ASTs). The common representation eases the integration of different tools in the environment e.g. the static semantic checker and the code generator. This well-defined representation also facilitate construction of metaprograms seen from the users point of view.

An AST is modeled as an instance of a class. The classes describing the ASTs are organized in a class hierarchy, which makes it possible to access an AST at 3 levels:

Tree level: The most general set of classes describe the AST as a traditional data structure in the form of a tree with the usual operations. The operations defined by the classes from the tree level includes operations to enable the substitution of one subtree in the AST with any other AST, irrespectively of the syntactic vadility of that substitution. This level is used by tools that need not know details about the AST. Using only the tree level enables the editor tool to be language independent. Accessing ASTs at this level corresponds to manipulate *S-expressions* in Lisp.

Context-free level: This level imposes a context-free structure on the AST. The operations defined by the classes from the context-free level includes operations to substitute one subtree of an AST with another AST with ensurence that the substitution corresponds to a syntactically legal substitution of the grammar. Tools using this level allow only syntactically legal operation on the AST. The context-free level is generated from the grammar of the language. The set of classes are subclasses of the classes from tree level. The interface is uniquely determined by the grammar and the generated classes need not be consulted

since the grammar may function as a specification of the interface.

Semantic level: The semantic level makes it possible to add semantic attributes and operations to the AST. As an example the static semantic checker of the BETA compiler use this level to decorate the ASTs with static semantic information.

6.1 ASTs and Classes

The tree level classes are predefined classes, corresponding to the five types of productions: construction, alternation, list zero, list one and optional. Given a specific grammar, the context-free level is defined by subclasses of these tree level classes, since any nonterminal will be modeled by a class that is a subclass of the tree level class corresponding to the production rule applied in the grammar.

```
<Block>    ::= begin <DclPart: DclLst>
               do <ImpPart: ImpLst> end
<DclLst>   ::* <Dcl> ;
<Dcl>      ::| <VarDcl> | <ProcDcl>
<VarDcl>   ::= var <Name: NameDcl>: <VarType: Type>
<ProcDcl>  ::= proc <Name: NameDcl> <Body: Block>
<ImpLst>   ::* <Imp> ;
<Imp>      ::= if <Cond: Exp> then <ThenPart: ImpLst>
                                   else <ElsePart: ImpLst>
               endif
<Assign>   ::= <Var: NameAppl> := <Value: Exp>
<ProcCall> ::= <Proc: NameAppl>
```

Figure 30: Simple grammar

The tree level classes are predefined and context-free level classes are generated automatically from the grammar specification. The semantic level is somewhat different. Instead of defining the semantic level as subclasses to the classes from the context-free level, the semantic level is defined by augmenting the automatically defined context-free level classes with attributes containing the semantic information.

To illustrate the correspondence between a grammar and the generated the class hierarchy, a simple grammar is given in figure 30.

The nonterminals `<NameAppl>`, `<NameDcl>`, `<Type>`, and `<Exp>` will not be defined. This grammar gives rise to the class hierarchy in figure 31. Indentation defines the subclass relation and the attributes of each class are shown in parentheses. The attributes of the classes `Cons` and `List` are defined by the metaprogramming system. In these classes, several mechanisms for manipulating the AST are also defined.

For a more thorough description of the metaprogramming system, see [9]. The metaprogramming system is among others inspired by the GRAMS system [3].

```
Cons (...)
  Block (DclPart: DclLst, ImpPart: ImpLst)
  Dcl
    VarDcl (Name: NameDcl, VarType: Type)
    ProcDcl (Name: NameDcl, Body: Block)
  Imp
    IfImp (Cond: Exp, ThenPart: ImpLst, ElsePart: ImpLst)
    Assign (Var: NameAppl, Value: Exp)
    ProcCall (Proc: NameAppl)
List (...)
  DclLst
  ImpLst
```

Figure 31: Simple grammar hierarchy

7 Other Tools in the Mjølner BETA System

To complete the overview of the Mjølner BETA System, the remaining parts of the system are shortly described.

7.1 The BETA Compiler

The BETA programming language supports the object oriented perspective on programming and contains comprehensive facilities for procedural and functional programming. Research is going on with the aim of including

constraint oriented constructs. BETA replaces classes, procedures, functions and types by a single abstraction mechanism called the *pattern*. It generalizes virtual procedures to virtual patterns, streamlines linguistic notions such as nesting and block structure, and provides a unified framework for sequential, coroutine, and concurrent execution. BETA is a modern language in the SIMULA tradition. The resulting language is smaller than SIMULA in spite of being considerably more expressive. A full description of the BETA programming language is outside the scope of this paper. The language is described in detail in [8, 7].

The compiler for BETA is an effective implementation of the BETA language (except concurrency). The major effort have been put into creating a production compiler for the sequential and alternation parts of the language in order to offer an effective implementation vehicle for the Mjølner BETA System.

The main components of the BETA compiler are the *semantic analyzer* and the *code generator*. The semantic analyzer checks the correctness of the context sensitive syntax (static semantics) of an AST, and performs storage allocation. The code generator translates an AST into executable code (native machine code). The code generator is divided into two components: the *synthesizer* and the *coder*. The synthesizer contains a machine independent model of the code generation, and the coder takes care of the machine dependent parts of the code generation. The synthesizer is the largest part of the code generator. This implies that porting the compiler to another machine can be done with a reasonable effort.

A symbol table is constructed during semantic analysis. The symbol table is defined by means of the semantic level of the metaprogramming system. I.e. the AST decorated with semantic attributes. In this way the symbol table information is an integrated part of the AST and thereby available for other tools accessing the AST (e.g. the editor).

In order to manipulate the ASTs, the compiler makes extensive use of the metaprogramming system. Furthermore, in order to generate ASTs from textual program representations, the compiler makes use of the parser. Finally, the compiler makes use of the pretty-printer to generate a textual representation of parts of the AST (e.g. in order to indicate program errors).

The runtime system for the BETA language is based on garbage collection. The garbage collection scheme is based on generation scavenging.

The compiler uses the fragment system to enable programs to be divided into smaller fragments for separate compilation. The compiler makes an automatic dependency analysis on the fragment structure. When a fragment has been changed, the system keeps track of the dependent fragments that must be recompiled.

7.2 The Source Level Debugger

Mjølner BETA System contains a source level debugger for the BETA language. It contains facilities for specifying break-points, single stepping, inspection of object states, inspection of the run-time organization, etc. The debugger is available both with a command-driven interface and with a graphical interface.

7.3 Parser and Pretty-Printer

Two tools exist for converting between textual and abstract syntax tree representations of a program. Both tools are grammar-based and can be applied to any language with a context-free grammar. The *parser* translates a text stream into an AST and the *pretty-printer* translates an AST into a text stream. The parser is based on LALR(1) parsing algorithms and the BOBS compiler generator system [5]. The pretty-printer is an adaptive pretty-printer based on the adaptive algorithxn presented in [12]. The pretty-printer is using a pretty-printing specification to guide the format of the output. For each production in the grammar, pretty-printing directives are given on the layout of sentences, derived from that nonterminal. This specification can be specified by the user. Both tools are used by the compiler and the editor.

7.4 The BETA User Interface Toolkit

Two object oriented user interface toolkits are available: `MacEnv` for the Macintosh Toolbox and `XtEnv` for the X Window System. They provide high level interaction concepts, such as hierarchical windows, icons, menues, dialog boxes, etc. The reason for having two different toolkits is that we want application writers to be aware of the limitations and to be able to utilize

the strengths of each user interface system. It is planned to create specializations of `XtEnv` to support Motif and Open-Look/OpenWindows in the future. Furthermore, it is planned to create a platform independent user interface toolkit for easy portability of applications (using the program variant facility of the fragment system to select proper implementation platform). However, it is forseen that this toolkit will be some sort of minimal toolkit for creating relative simple applications.

7.5 The Bifrost Graphics System

The Bifrost graphics system [2] is a device independent, interactive, extensible, and tailorable graphics system based on the stencil & paint imaging model. The graphics system supports graphics modeling, interaction with graphics (creation, reshaping, translation, scaling, and rotation), graphics contexts (local, shared and global), and automatic damage repair.

Besides being a fragment library available for programmers, a MacDraw-like drawing application has been build based on Bifrost.

7.6 The Ensemble

The Mjøner BETA System has an object oriented interface to the operating system called the ensemble. The ensemble has three major goals:

1. To access the file system.
2. To control processes in the operating system.
3. To communicate between these processes.

The ensemble is written in BETA, and is therefore accessible to application programmers. The ensemble consists of a operating system independent part and several operating system dependent parts. A UNIX ensemble is extensively used in the UNIX implementations of the Mjølner BETA System, and a MacOS ensemble is used in the Macintosh implementation. For a full description of the Mjølner BETA System manuals.

7.7 Interface to Other Languages

Besides the ensemble interface to the operating system, there exists interfaces to routines written in C and Pascal. This interface enables the programmer to invoke routines written in these languages. Furthermore, there are interfaces to data structures in C (`struct`) and in Pascal (`record`). Furthermore, there are support for call-back from C or Pascal routines to BETA routines.

These interfaces defines a clean interface to these languages, and enables reuse of existing systems, written in other languages. An interface exists also for specifying machine code as part of the actions of an object. These interfaces are sufficient general that interfaces to other stack-based languages can be created with reasonable effort.

The existence of the ensemble and the extern routine interface makes the Mjølner BETA System highly portable across platforms.

7.8 Graphical Hyper Structure Editor

In order to support graphical editing of programs and fragments, and in order to support object oriented analysis and design, a graphical editor is being developed, partly based on a proposal for a syntax-directed graphical editor with extensive support for object oriented analysis and design, see [14, 15]. The editor is developed on top of the hyper structure editor, and will therefore facilitate abstract browsing, documentation, and hyper linking within graphical documents as well as symmetric linkage between programs, documentation and design diagrams. Furthermore, since the editor will use the same underlying structure for programs, documentation and design diagrams, consistency between these documents can easily be ensured. This tool will be a valuable CASE tool for object oriented analysis and design.

7.9 Application Builder

An application builder for the integrated construction of the user interface and functionality of an application is being developed. The user interface is constructed using direct manipulation, and the application builder generates BETA code that makes it easy for the designer to insert his own application code using the hyper structure editor. The generated code and the applica-

tion code are kept separated using the fragment system. Furthermore, the application builder will enable the modification of the interface code through the application builder while preserving the application code.

7.10 Future Developments

The Mjølner BETA System will be further developed in the future. Several new developments are under consideration:

Incremental Compilation: The present system supports only separate compilation. Incremental compilation techniques, as those developed in the Swedish part of the Mjølner project [10] will be examined in order to investigate their application in the Mjølner BETA System.

Dynamic Linking: Dynamic linking is one technique to minimize startup time for large programs and to minimize storage consumption of programs.

Persistent Objects: One of the very active research areas within object oriented programming is persistent objects. Presently, we are examining several approaches to persistent objects in the Mjølner BETA System, one of which is the use of extensible programs as the vehicle for persistence [1].

Extensible Programs: The ability to specify additional program fragments during run time of a program, and compile and link that fragment into the running program is currently under investigation [1] . Currently an experimental implementation of this technique have been completed with promising results.

Acknowledgements

The work reported here has been partly supported the Nordic Mjølner project. The Mjølner project [4] is funded by the participating organizations and supported by grants from *The Nordic Fund for Technology und Industrial Development*. The Danish team consisted of the authors, Ole Agesen, Peter Andersen, Karen Borup, Svend Frølund, Kim Jensen Møller, Claus H.

Pedersen and Per Fack Sørensen. The BETA language design team consists of Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen and Kristen Nygaard. Finally, we would like to express our thanks to Apple Computer for supporting the Macintosh implementation, and to Apollo Computer and Hewlett-Packard for support at various stages in the project.

References

[1] O. Agesen, S. Frølund, M.H. Olsen: *Persistent and Shared Objects in BETA*, Computer Science Department Technical Report IR-89, Aarhus University, April 1989.

[2] P. Andersen, K. Jensen Møller, J. Rask: *Bifrost: An Interactive Object Oriented Device Independent Graphics System*, Computer Science Department Technical Report IR-100, Aarhus University, February 1991.

[3] R.D. Cameron, M. Robert Ito: *Grammar-based Definition of Metaprogramming Systems*, ACM Transactions on Programming Languages and Systems, Vol. 6, No. 1, January 1984.

[4] H.P. Dahle, M. Løfgren, O.L. Madsen, B. Magnusson (eds): *The Mjølner Project*, In Proceedings of EUROSOFT '87, London, June 1987.

[5] S.H. Eriksen, B.B. Jensen, B.B. Kristensen, O.L. Madsen: *The BOBS System*, Computer Science Department Technical Report PB-71, Aarhus University, March 1977.

[6] B.B. Kristensen, O.L Madsen, B. Møller-Pedersen, K. Nygaard: *Syntax-directed Program Modularization*, In P. Degano, E. Sandewall (Eds.): *Interactive Computing Systems*, North-Holland, 1983

[7] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard: *Object Oriented Programming in the BETA Programming Language*, Book Draft, Computer Science Department, Aarhus University, January 1991.

[8] O.L. Madsen, B. Møller-Pedersen: *Basic Principles of the BETA Programming Languuge*, In Gordon Blair et. al (Eds.): *Object Oriented Languages, Systems and Applications*, Pitman, 1991.

[9] O.L. Madsen, C. Nørgaard: *An Object Oriented Metaprogramming System*, In B.D. Shriver (ed.): *Hawaii International Conference on System Sciences – 21*, IEEE, January 1988.

[10] B. Magnusson et al.: *An Overview of the Mjølner/ORM Environment: Incremental Language and Software Development*, TOOLS'90, Technology of Object Oriented Languages and Systems, Paris, 1990.

[11] C. Nørgaand, E. Sandvad: *Reusability and Tailorability in the Mjølner BETA System*, TOOLS'89: Technology of Object Oriented Languages and Systems, Paris, Nov. 1989.

[12] D.C. Oppen: *Prettyprinting*, ACM Transactions on Programming Languages and Systems, Vol. 2 No. 4, Oct. 1980.

[13] E. Sandvad: *Hypertext in an Object Oriented Programming Environment*, Woodman'89: Workshop on Object Oriented Document Manipulation, Rennes, May 1989.

[14] E. Sandvad: *Syntax-directed Graphical Editing*, Computer Science Department, Aarhus University, June 1989 (DRAFT).

[15] E. Sandvad: *Object Oriented Development — Integrating Analysis, Design and Implementation*, Computer Science Department Technical Report PB-302, Aarhus University, April 1990.