ISSN 0105-8517

# An Object-Oriented Metaprogramming System

Ole Lehrmann Madsen
Claus Nørgaard

DAIMI PB – 236
November 1987

# An Object-Oriented Metaprogramming System *

**Ole Lehrmann Madsen**
Computer Science Department,
Aarhus University, Denmark
(e-mail : olm@daimi.dk)

**Claus Nørgaard**
Institute of Electronic Systems,
Aalborg University Center, Denmark
(e-mail : cn@sysware.dk)

October 1987

**Abstract**

A number of tools in the Mjølner programming environment are *metaprograms*, i.e. programs that manipulate other programs. The metaprogramming system is grammar based in the sense that a metaprogramming tool may be generated from the grammar of any language. For each syntactic category of the language, a corresponding class is generated. The syntactic hierarchy of the grammar is mapped into a corresponding class hierarchy. This object-oriented representation of programs is further exploited by including a set of more general classes that view a program as an abstract syntax tree and by allowing the user to add semantic attributes in subclasses.

*To be presented at *Hawaii International Conference on System Sciences — 21, January 5-8, 1988*

## 1 Introduction

The highly interactive, integrated and incremental programming environments of many Lisp systems, such as Interlisp [Interlisp], are in many aspects much more advanced than most environments for development of production software. According to Abelson and Sussman [Abelson & Sussman 85] the most significant feature of Lisp is the fact that Lisp procedures can themselves be represented and manipulated as Lisp data. This makes Lisp an excellent language for writing *metaprograms*, i.e. programs that manipulate other programs, such as programming environments.

Lisp has mainly been successful for development of prototype systems using the *exploratory programming style* [Sheil 83]. However, Lisp is not considered an alternative for development of production programs. Some of the reasons are the lack of structure and security. There is currently a number of efforts going on to include some of the advantages of Lisp systems into environments for production programming. The Mjølner project [Mjølner 86] is such an activity. This paper describes the principles of metaprogramming in Mjølner.

Mjølner is a programming environment that supports design, implementation and maintenance of large production programs. In such an environment support for structure and security is essential. Mjølner is primarily aimed at supporting the object-oriented programming style. Initially the environment will support Standard Simula [Simula] and BETA [BETA 87]. In contrast to Smalltalk [Smalltalk], Simula and BETA are mainly intended for production programming. They belong to the Algol family with respect to block structure, type checking and lexical (static) name binding.

All metaprogramming tools in Mjølner manipulate programs through a common representation that is abstract syntax trees (ASTs). It was decided that for a language supported by the system, the corresponding ASTs should be instances of a well-defined data type. In most non-Lisp based systems, the data types defining ASTs are internal parts of the various tools, such as compilers, debuggers etc. This makes it almost impossible for a user to write a metaprogram using the same AST-definition.

There is no commonly agreed definition of abstract syntax tree, which implies that each language implementor selects his own definition. A context free grammar for a language induces an abstract syntax that may be used to give an AST-definition. In Mjølner, the representation of a program as an AST is defined by means of a context free grammar for the language. In addition there is a set of rules that specify how the context free grammar is mapped into a set of data types. The context free grammar is then part of the specification of the environment.

The ASTs defined by the context free grammar may be described as Lisp S-expressions. An example of a Pascal statement and a corresponding AST in the form of an S-expression is

```
while p <> q do if p < q then q := q − p else p := p − q

(while (<> p q)
       (if (< p q)
           (:= q (− q p))
           (:= p (− p q))))
```

S-expressions could in fact be used for manipulation of an AST. In order to do this in Simula, BETA, or other languages, predefined classes (types) modelling S-expressions could be included in the environment.

Not all S-expressions do however constitute correct programs. In order for an S-expression to be an AST, a certain context free structure must be satisfied. E.g. the S-expression "(while (if *p*) (else *q*))" does not correspond to an AST for a Pascal program, even though it is a well defined tree structure.

An object-oriented model of the ASTs has been developed in Mjølner. An AST is modelled as an instance of a class. There is a class corresponding to each syntactic category (nonterminal) of the grammar. ASTs derived from a syntactic category are then modelled as instances of the corresponding class. The class *IfImp* corresponds to the syntactic category <*IfImp*>. Instances of class *IfImp* then model ASTs that may be derived from <*IfImp*>.

The grammar hierarchy is modelled by a corresponding class hierarchy. E.g. if the nonterminal <*Imp*> may derive <*IfImp*>, < *WhileImp*> etc., then the class *Imp* will be a super-class of *IfImp*. The class hierarchy is derived automatically from the context free grammar. In order for this to work properly, the context free grammar must obey a certain structure.

Using the metaprogramming system, there is a well defined representation of programs in the form of ASTs. This implies that the various Mjølner tools and other metaprograms all are able to use the same representation of programs.

The *grammar based* interface described above results in a set of classes for each language. A metaprogram using the grammar based interface will thus be *language specific* since it uses the set of classes generated from the grammar of the actual language. A number of tools are language specific in the sense that usually one exist for each language. Examples of tools that benefit from using the grammar based interface are: semantic checkers, program analyzers, interpreters, browsers, graphical presentation tools, transformation tools.

For certain types of metaprograms it may be inconvenient to use the grammar based interface, since it implies grammar based information to be hard-coded in the programs. If manipulation of the AST could only take place through this interface, it would be necessary to write such tools for every new language. This is of course not acceptable. Examples of such tools are table-driven parsers and syntax-directed editors.

In order to support both types of tools, the AST in the Mjølner environment may be accessed at 3 levels.

1. **Tree level.** Here the AST is viewed as a tree. This corresponds to S-expressions.

2. **Context free level.** This is the grammar based interface generated automatically from the grammar. This level corresponds to S-expressions where a context free structure is imposed, together with functions for accessing the components of the AST.

3. **Semantic level.** At this level semantic attributes may be added to the AST. The attributes are tool dependent and usually reflect context sensitive aspects of the language.

The 3 levels are also modelled by a class hierarchy. A generated context free level is a subclass of tree level, and a semantic level is a subclass of the context free level for the language in question.

The Mjølner metaprogramming system is mainly an application of ideas found in Aleph [Winograd 83] and GRAMPS [Cameron & Ito 84]. Aleph is a general specification language that among others includes predefined classes for the abstract syntax of Aleph. GRAMPS is a metaprogramming system where the grammar is mapped into Pascal types. By using Pascal it is not possible to model the grammar hierarchy in the data types like it is done in Aleph and Mjølner. In Mjølner an attempt has been made to view traditional tools like editor, compiler and debugger as metaprograms in general. The advantage of this is that all tools including user programs access programs through a common representation. This lead to the integration of the grammar based interfaces with the tree level and semantic level described above.

Like the Mjølner metaprogramming system, the Cornell Program Synthesizer [Teitelbaum & Reps 81] may be used to generate part of a programming environment for a language. The purpose of the Mjølner metaprogramming system and the Synthesizer are however different. The Synthesizer is a tool for implementing language-based editors. The Mjølner metaprogramming system provides a foundation for such an editor generator. An editor generator has been implemented on top of the metaprogramming system. The difference between the Mjølner editor generator and the Synthesizer is that the Synthesizer takes a specification in form of an attribute grammar whereas the Mjølner editor generator is based on the context-free grammar formalism used by the metaprogramming system. The advantage of using an attribute grammar is the ability to specify the static semantics in a declarative style. In Mjølner the static semantic checkers are programmed by means of the metaprogramming system. In theory the metaprogramming system could be based on attribute grammars too. This has not been done due to the requirement that Mjølner must support production programming. However, experiments with a restricted form of attribute grammar for specifying static semantics, take place in the Mjølner project ([Hedin 86]). It is currently an open question, if a practical system can be based on attribute grammars in general. Since the metaprogramming system is based on a programming language a large number of applications other than language-based editors may be implemented. Of course the generality of attribute grammars implies that in theory the Synthesizer may be used for a large number of applications too.

In [BETA 85] a so-called *descriptor algebra* for manipulation of program fragments was presented. New language constructs for expressing the descriptor algebra were presented. The Mjølner metaprogramming system shows that the principles behind the descriptor algebra may be realized within an object-oriented framework without introducing new language constructs.

The implementation language of the metaprogramming system is BETA. This means that all metaprograms are written in BETA. Metaprograms can manipulate ASTs of any context free language. BETA will thus be used in this paper to describe the metaprogramming system, and the relevant parts of BETA will be described before use.

## 2 Structured Context Free Grammars

The grammar formalism used in Mjølner is a variant of context free grammars. The main reason for introducing this formalism is to make it possible automatically to generate class definitions from a grammar. The grammar formalism used in Mjølner is inspired by [Cameron & Ito 84]. There exist a number of different formalisms (see e.g. [Nørmark 87]) that could be reasonable alternatives.

A *structured context free grammar* is a context free grammar (CFG) where the rules (productions) satisfy a certain structure.

Each nonterminal must be defined by exactly one of the following rules:

1. An *alternation rule* has the following form:

   <*A0*> :: | <*A1*> | <*A2*> | ... | <*An*>

   where <*A0*>, <*A1*>, ..., <*An*> are nonterminal symbols. The rule specifies that <*A0*> derives one of <*A1*>, <*A2*>, ..., or <*An*>.

2. A *constructor rule* has the following form:

   <*A0*> ::= $w_0$ <*t1:A1*> $w_1$ ... <*tn:An*> $w_n$

   where <*A0*>, <*t1:A1*>, ..., <*tn:An*> are nonterminal symbols and $w_0, w_1, ..., w_n$ are possibly empty strings of terminal symbols. This rule describes that <*A0*> derives the string

   $w_0$ <*A1*> $w_1$ ... <*An*> $w_n$

   A nonterminal on the right side of the rule has the form <*t:A*> where *t* is a tag-name and *A* is the syntactic category. Tag-names are used to distinguish between nonterminals belonging to the same syntactic category. Consequently all tag-names in a rule must be different. If no tag-name is provided the name of the syntactic category is used as a tag-name.

3. A *list rule* has one of the following forms:

   <*A*> ::+ <*B*> *w*

   <*A*> ::* <*B*> *w*

where *<B>* is a nonterminal and *w* is a possibly empty string of terminal symbols. The nonterminal *<A>* generates a list of *<B>*'s separated by *w*'s:

*<B> w <B> w ... w <B>*

The +-rule specifies that at least one element is generated; the *-rule specifies that the list may be empty.

There exists four predefined nonterminal symbols named *<NameDecl>*, *<NameAppl>*, *<String>* and *<Const>*. These nonterminals are called *lexem-symbols*. They derive identifiers, character-strings and integer constants. A lexem-symbol may also has a tag-name, like *<Title:NameAppl>*.

## Example of Structured CFG

Below an example of a structured CFG is given.

**Grammar** *Small* :

```
<Block> ::= begin <DclPart:DclLst>
            do <ImpPart:ImpLst> end

<Dcl> :: | <VarDcl> | <ProcDcl>

<VarDcl> ::= var <Name:NameDecl> : <VarType:Type>

<ProcDcl> ::= proc <Name:NameDecl> <Body:Block>

<Imp> :: | <IfImp> | <AssignmentImp> | <ProcCall>

<IfImp> ::= if <Condition:Exp>
            then <ThenPart: ImpLst>
            else <ElsePart: ImpLst> endif

<AssignmentImp> ::= <Var:NameAppl> := <Value:Exp>

<ProcCall> ::= <Proc:NameAppl>

<DclLst> :: * <Dcl> ;

<ImpLst> :: * <Imp> ;
```

The nonterminals *<Type>* and *<Exp>* will not be defined.

The limitations on the rules which can be used in a structured CFG do not restrict the class of languages that can be described. Any context free language may be generated by a structured CFG. It may perhaps be awkward to be forced to follow the rules. On the other hand being forced to structure a grammar using the rules often results in a more readable grammar.

The syntactic categories of a structured CFG may be organized into a classification hierarchy according to the set of strings being generated. The hierarchy mainly derives from the alternation rules of the grammar. The hierarchy for the example grammar is:

- *Cons*
  - *Block*
  - *Dcl*
    - *VarDcl*
    - *ProcDcl*
  - *Imp*
    - *IfImp*
    - *AssignmentImp*
    - *ProcCall*
- *List*
  - *DclLst*
  - *ImpLst*

The categories *Cons* and *List* generalize all categories according to the rule type that defines the category. <*Imp*> is a *super-category* of <*IfImp*> since any string generated by <*IfImp*> may be generated by <*Imp*>.

The *super-category* of a given syntactic category $A$ is defined as follows :

- if <*A*> appears on the right side of an alternation rule of the form

  <*B*> :: | ... | <*A*> | ... | ...

  then the super-category of $A$ is $B$.

- if <*A*> appears in a list rule in one of the forms

  <*A*> ::+ <*B*> ...
  <*A*> ::* <*B*> ...

  then the super-category of $A$ is *List*.

- otherwise the super-category of $A$ is *Cons*.

The inheritance hierarchy of the generated classes of the context free level is the same as the classification hierarchy of the syntactic categories. In general a syntactic category may have more than one super-category. This corresponds to multiple inheritance in object-oriented languages. Since Simula and BETA currently do not support multiple inheritance, there is the additional restriction that the hierarchy must be tree structured.

## 3 The Tree Level

As mentioned in the introduction certain tools like syntax directed editors are usually table-driven in the sense that the code is independent of the actual grammar. The AST is manipulated as an ordinary tree. The context free level must then be integrated with a level where the AST is viewed as an ordinary tree. This is straight forward using subclassing. The classes generated from the grammar are all subclasses of the general classes *Cons* and *List*. These classes are actually subclasses of more general classes describing ASTs as ordinary trees. In this section these general classes are described. The general classes are called the *tree level*.

At the *tree level* an AST is modelled as an instance of the class *AST*. The class *AST* is further specialized into a number of sub-classes. Some of these sub-classes correspond to the rule types of a structured CFG. The tree level corresponds to an ordinary data type for a tree. The specialization hierarchy for the classes defined in the tree level is

- *AST*
  - *Rule*
    - *Cons*
    - *List*
  - *Lexem*
    - *LexemText*
      - *NameAppl*
      - *NameDecl*
      - *String*
    - *Const*

The following is a verbal description of these classes:

**AST** describes all ASTs. Operations of this class are *Symbol* which returns the nonterminal symbol of the AST, *Father* returns the father, *Comment* updates or returns an associated comment, *Copy* returns a copy, and *Match* performs matching of trees.

**Rule** describes all interior nodes. *GetSon* returns a son at a given position, *PutSon* updates a son at a given position, *SuffixWalk* performs a preorder traversal of the tree with this node as root

**Cons** describes all nodes derived by a constructor rule. The operations *Son1*, *Son2*, ... updates or returns sons. Each $Son_i$ have a local attribute *Cat* describing the class of the son.

**List** describes all nodes derived by a list rule. *ElmCat* describes the class of the elements of the list, *NoOfSons* returns the number of elements in the list, *Scan* iterates over the elements in the list, *Delete* deletes an element with a given position and *Insert* inserts an element at a given position.

**Lexem** describes all nodes derived by one of the predefined nonterminals.

**LexemText** describes leaves having textual contents. *GetText* returns the textual contents, *PutText* updates the textual contents.

**NameAppl** describes all name applications. *GetDecl* returns the declaration of this name application. This information is context-sensitive and if used it must be set up by a language specific tool. *SetDecl* tells where the application is declared.

**NameDecl** describes all name declarations. *ScanUsage* iterates over the name applications that use this declaration.

**String** describes all strings.

**Const** describes all integer constants. *GetValue* returns the value of the constant, *PutValue* updates the value of the constant.

In addition the procedure *NewAST* may be used for creating an AST instance given a syntactic category.

## 4 The BETA Programming Language

The metaprogramming system relies heavily on the power of the BETA language. It is however outside the scope of this paper to give a detailed account of BETA. The reader is referred to [BETA 87,BETA 88][1] for a description of BETA. A brief introduction to a subset of BETA will however be given below. The goal is that the reader should be able to understand the principles of the metaprogramming system without necessarily understanding all the details of the examples[2].

The BETA equivalent to a class in Simula and Smalltalk is called a pattern. The declaration of a pattern $P$ has the following form:

```
P: P0 (# Dcl1; Dcl2; ... Dcln
      enter In
      do Imp
      exit Out
      #)
```

where *P0* is a possible super-pattern of $P$. $Dcl_1$, $Dcl_2$, ..., $Dcl_n$ are declarations of attributes. An attribute may be either a *reference* or a *pattern*. *In* corresponds to input parameters. *Imp* (the *do-part*) is an imperative describing actions to be executed when the object is executed as a procedure, coroutine or concurrent process. *Out* corresponds to output parameters.

$P$ is said to be a *direct sub-pattern* of *P0*. A pattern $P'$ is a *sub-pattern* of $P''$ if it is a direct sub-pattern of $P''$ or if it is a sub-pattern of a possible super-pattern of $P''$.

References may be either *static* or *dynamic*. A static reference denotes an object that is generated as part of the object containing the reference as an attribute. A dynamic reference is a variable that may denote different objects during its lifetime. Dynamic references correspond to instance variables in Smalltalk. In the following example *R1* is declared as a static reference to a $P$ object and *R2* is declared as a dynamic reference to $P$ objects.

```
R1: @P;  {A static reference}
R2: ↑ P;  {A dynamic reference}
```

*R2* will initially have the value NONE. A major difference between Smalltalk and BETA (and Simula) is that references in BETA (and Simula) are qualified by a pattern name. The qualification restricts the possible objects that may be be referred to by the reference. A static reference

[1][BETA 88] is included in the proceedings from this conference

[2]The reader may want to skip this section during a first reading and perhaps return when forced by the details of the examples!

denotes an instance of the qualifying pattern. A dynamic reference may denote instances of the qualifying pattern and instances of sub-patterns of the qualifying patterns.

An expression of the form *R*.*x* denotes the *x* attribute of the object *R*. Such an expression is only legal if *x* is declared as an attribute of the qualifying pattern of *R*. This may be checked at compile-time.

It is possible to test if a reference denotes an object belonging to a sub-pattern of the qualifying pattern. The expression (*R* **is** *P1*) has the value true if *R* denotes an instance of *P*1, otherwise the value is false. The expression (*R* **qua** *P1*) is a so-called *instantaneous qualification* of *R*. The value of the expression is a reference to the object denoted by *R*, but qualified as a *P*1 pattern. If *R* does not denote an instance of *P*1, the evaluation constitutes a run-time error.

As mentioned a pattern is the equivalent of a Smalltalk class. A pattern is however a generalization of abstraction mechanisms such as class, procedure, function and type. The equivalent of a Smalltalk method will thus be a pattern attribute. A pattern attribute may however be used for other purposes than as a method.

A pattern attribute may be declared as a *virtual pattern*. A virtual pattern declaration gives only a partial description of the pattern. The description of a virtual pattern may be extended in sub-patterns of the pattern of which it is an attribute. Virtual patterns will be further explained below.

The following example is the pattern *List* from the tree level of the previous section.

```
List: Rule
(# ElmCat :< AST;  {A virtual pattern}

   NoOfSons: (# no: @Integer ... exit no#);

   Delete:
   (# SonNo: @Integer
   enter SonNo
   do ...
   #);

   Insert:
   (# SonNo: @Integer; ASon: ↑ ElmCat
   enter (SonNo, ASon)
   do ...
```

```
        #);

        Scan:
        (# ThisElm: ↑ ElmCat
        do (for i: NoOfSons repeat
                i → GetSon → ThisElm□;
                inner
            for)
        #)
    #)
```

The description of *List* includes five pattern attributes *NoOfSons, ElmCat, Insert, Delete* and *Scan*. *ElmCat* is declared as a virtual pattern attribute.

*NoOfSons, Insert* and *Delete* are examples of pattern attributes used as procedures/methods. The following example shows how to call a pattern attribute as a procedure/method.

```
L: ↑ List;  {L denotes a List object}

3 → L.Delete;  {Delete the third element of the list}
(3,T) → L.Insert;  {Insert T after the third element}
```

The *Scan* attribute is an example of a pattern used as a control abstraction. Consider a pattern

```
LScan: L.Scan(# do Imp'#)
```

An execution of *LScan* will step through the elements of the list and execute *Imp'* for each element in the list *L*. The dynamic reference *ThisElm* of *Scan* is an index variable that denotes the current element. Notice that the super-pattern of *LScan* is the *Scan* attribute of the object denoted by *L*.

The actions taking place when executing an instance of *LScan* is a combination of the do-part of *Scan* and the do-part of *LScan*. Execution starts with the do-part of *Scan*. An occurrence of **inner** implies execution of the do-part of *LScan*.

It is not necessary to declare a pattern for each list to be Scanned. The description of a pattern may be used directly as an imperative in the following way

```
L.Scan(# do ThisElm.Display#)
```

assuming that instances of pattern *AST* have a *Display* attribute.

As mentioned the context free level of the metaprogramming system will contain a class corresponding to each syntactic category of the grammar. Assume that there is a pattern *Imp* corresponding to ASTs generated from the nonterminal *Imp*. It is then possible to define a pattern *ImpList* that classifies lists of instances of *Imp*. Pattern *ImpList* may be declared as a sub-pattern of the general list pattern *List*. The description of the virtual pattern will be extended to be an *Imp*. This implies that the elements of the list must at least be instances of the pattern *Imp*. Note that the virtual pattern *ElmCat* has thus been used as a formal type parameter.

```
ImpList: List(# ElmCat ::< Imp#)

IL: ↑ ImpList

IL.Scan
(# do {ThisElm denotes an instance of pattern Imp} #)
```

The declaration of *ElmCat* in pattern *List* has the form *ElmCat* :< *AST*. This specifies that *ElmCat* will at least have all the properties described for the pattern *AST*. *ElmCat* may consequently only be bound to sub-patterns of *AST*. The pattern *Imp* thus has to be a sub-pattern of *AST*.

Virtual patterns may be used to achieve the effect of dynamic binding of methods in Smalltalk. Assume that there are patterns *AssignmentImp*, *IfImp* and *ProcCall* corresponding to ASTs generated from the respective nonterminals. These patterns may be sub-patterns of the pattern *Imp*. Assume that a tool for counting the number of imperatives in an instance of *ImpList* is to be implemented. In addition the tool should count the number of different types of imperatives. The tool may be implemented by adding a virtual pattern attribute *Count* to the pattern *Imp*. The definition of *Count* may then be extended in the sub-patterns.

```
Imp: Cons
(# Count :< (# do ImpCount.Add1; inner #)
#);
AssignmentImp: Imp
(# ...
   Count ::< (# do AssignmentCount.Add1#)
#);
```

```
...
IL.Scan(# do ThisElm.Count#)
```

Note that the description of the virtual pattern *Count* is given directly instead of referring to an existing pattern as in *ElmCat* :< *AST*. The virtual pattern *Count* in *AssignmentImp* will be a sub-pattern of the *Count* pattern in *Imp*.

If *ThisElm* in *IL.Scan*... denotes an instance of *AssignmentImp* then *ThisElm.Count* will result in execution of *impCount.Add1* followed by *assignmentCount.Add1*.

## 5 The Context Free Level

The context free level has explicit knowledge about the grammar for the language. For each nonterminal $A$ of the grammar, a corresponding pattern is automatically generated, depending of the defining rule for $A$. For each rule type described in section 2, the list below describes the corresponding generated classes.

1. Alternation: A pattern of the following form is generated:

   ```
   A:  P (# #)
   ```

   where $P$ is the pattern corresponding to the super-category of $A$. The pattern $P$ is thus the super-pattern for $A$.

2. Constructor: A pattern of the following form is generated:

   ```
   A: P
   (# t1: Son1(# Cat ::< A1 #);
      t2: Son2(# Cat ::< A2 #);
      ........
      tn: Sonn(# Cat ::< An #)
   #)
   ```

   where $P$ is the super-category of $A$. There is an attribute corresponding to each nonterminal on the right side of the rule. The name of an attribute ($t_i$) is the same as the corresponding tagname.

   If $T$ is an instance of $A$ then $T.t_i$ denotes the i'th sub-AST. The *Soni* patterns have enter and exit parameters such that an instance of the pattern *Ai* can be inserted as the i'th sub-ast by ... $\rightarrow T.t_i$ and will be delivered by $T.ti \rightarrow$ ...

3. List: A pattern of the following form is generated:

   *A*: *List* (# *ElmCat* ::< *B* #)

   where *B* is the name of the nonterminal on the right side of the rule. The super-pattern is *List* as the super-category of *A* is *List*.

Constructor rules are thus mapped into an aggregation hierarchy and alternation rules into an inheritance hierarchy.

By using the context free level it is not possible for a programmer to construct an AST that violates the context free syntax.

In addition patterns are generated which provide easy creation of new ASTs from existing ones. For each nonterminal *A* of the grammar a generator named *NewA* is generated. For nonterminals defined by a constructor rule, *NewA* will as enter parameter take as many ASTs as there are nonterminals on the right side of the rule. It will then exit an *A*-ast with the enter parameters as sons.

## Example of structured CFG (continued)

The patterns generated for the example grammar are:

```
Small : TreeLevel
(#
  Block :  Cons
  (# DclPart: Son1(# Cat ::< DclLst #);
     ImpPart: Son2(# Cat ::< ImpLst #)
  #);

  Dcl:  Cons (# #);

  VarDcl :  Dcl
  (# Name: Son1(# Cat ::< NameDecl #);
     VarType: Son2(# Cat ::< Type #)
  #);

  ProcDcl :  Dcl
  (# Name: Son1(# Cat ::< NameDecl #);
     Body: Son2(# Cat ::< Block #)
  #);
```

```
   Imp: Cons (# #);

   IfImp : Imp
   (# Condition: Son1(# Cat ::< Exp #);
      ThenPart: Son2(# Cat ::< Imp #);
      ElsePart: Son3(# Cat ::< Imp #)
   #);

   ProcCall : Imp
   (# Proc: Son1(# Cat ::< NameAppl #)
   #);

   AssignmentImp : Imp
   (# Var: Son1(# Cat ::< NameAppl #);
      Value: Son2(# Cat ::< Exp #)
   #);

   DclLst: List (# ElmCat ::< Dcl #);
   ImpLst: List (# ElmCat ::< Imp #);

   { generators for new ASTs }
   NewBlock :
   (# DclPart : ↑ DclLst; ImpPart : ↑ ImpLst;
      TheBlock : ↑ TheBlock
   enter (DclPart,ImpPart)
   do ...
   exit TheBlock
   #)
   ...{ more generators }
#)
```

## 6 Using the metaprogramming system

Consider the references

```
    P : ↑ ProcDcl;
    B : ↑ Block
```

*P.Body* refers to the block of *P*, and after executing the assignment *P.Body* → *B*, *B* will refer to this block. *P.Name.GetText* will return the name of the procedure as a text.

Consider a tool for investigating the contents of a block, where part of the investigation is to count the number of imperatives in the block.

In addition the number of different types of imperatives will be counted.

This tool may be implemented by adding the operation *Investigate* to the pattern *Block*. *Investigate* makes use of the virtual operation *Count* which is added to the pattern *Imp*. *Count* is further specialized in the sub-patterns of *Imp*. The attributes that are added to the context free level are called *semantic attributes*.

```
Small : Treelevel
(# ImpCount, AssignmentCount,
   ProcCallCount, IfCount : @Integer;

   Block: Cons
   (# ...
      Investigate :
      (#
      do 0 → ImpCount → AssignmentCount
            → ProcCallCount → IfCount;
         ImpPart.Scan(# do ThisElm.Count #);
         ... { Use ImpCount, ProcCallCount, ... }
      #);
   #);
   ...
   Imp : Cons
   (# Count :< (# do ImpCount.Add1; inner #)
   #);

   IfImp : Imp
   (# ...
      Count ::<
      (#
      do IfCount.Add1;
         ThenPart.Scan(# do ThisElm.Count #);
         ElsePart.Scan(# do ThisElm.Count #);
      #);
   #);

   AssignmentImp : Imp
   (# ...
      Count ::< (# do AssignmentCount.Add1 #)
   #);

   ProcCall : Imp
   (# ...
      Count ::< (# do ProcCallCount.Add1 #);
   #);
   ...
#);
```

*B* can now be investigated by *B.Investigate*;

In spite of the limited usefulness of the above example it gives a flavour of how semantic attributes may be added to the generated classes. Tools like a semantic analyzer, a code generator, a program interpreter, a browser, presentation tools, program analyzers, transformation tools benefit from the possibility to add semantic attributes.

The next example will demonstrate how the syntax directed editor of Mjølner can be extended to provide the user of the editor with *transformations.*

The editor is an ordinary syntax directed editor which presents an AST in a window by means of a prettyprinter, it allows the user to navigate in the AST and to edit it. The pattern describing the editor has the outline

```
Sde :
(# Grammar  :< TreeLevel;
   G : @ Grammar;

   Root, CurrentSelection : ↑ G.AST;

   ... { a lot of other stuff }
#)
```

*Grammar* describes which grammar is actually used. An editor for Pascal may be constructed by binding the context free level generated for Pascal to *Grammar.* The reference *Root* denotes the program fragment being edited by the user. *CurrentSelection* denotes the sub-AST which is the current focus of the user.

To extend the editor with transformations the pattern *SdeWithTransformations* is declared as a sub-pattern to *Sde. SdeWithTransformations* declares the pattern *Transformation*, which has three virtual operations *Init*, *EnablingCondition* and *Perform* and a static reference *Name.*

*SdeWithTransformations* keeps a list containing an instance of each sub-pattern of *Transformation.* This list is created by means of initialization operations not shown here.

When the user selects a new node in the tree *EnablingCondition* will be tested for all transformations. The *Names* of those that are enabled will be presented to the user in a menu, and if the user selects one of the items in this menu, *Perform* for the corresponding transformation will be called.

```
SdeWithTransformations : Sde
(# ...
   Transformation :
   (#
      Name : @ Text; { to be presented in the menu}

      Init :< (# do ... inner ; ... #);
      { to be called when instance is created }
      ...
      EnablingCondition :<
      { virtual operation to test if transformation
        is applicable for the current selection
        of the editor.}
      (# Enabled : @ Boolean
      do inner
      exit Enabled
      #);

      Perform :<
      { operation to be performed if the user selects
        this transformation.}
      (# do inner #);
   #);
   ...
#);
```

Assume a grammar for Pascal has been written, structured as *Small*, and including the rule

```
<WhileImp> ::= while <Condition:Exp>
               do <DoPart:ImpLst>
```

A syntax directed editor for Pascal with a transformation that will allow the user to transform an *IfImp* into a *WhileImp* could be created by the pattern

```
PascalEditor : SdeWithTransformations
(# Grammar ::< PascalGrammar;

   IfToWhileTransformation : Transformation
   (# Init ::< (# do 'IfImp to WhileImp' → Name #)
        EnablingCondition ::<
        (#
        do (CurrentSelection is G.IfImp) → Enabled
        #);

        Perform ::<
        (#
        do ((CurrentSelection qua G.IfImp).Condition,
            (CurrentSelection qua G.IfImp).ThenPart)
            → G.NewWhileImp → ReplaceCurrentSelection;
        #);
    #);

    { more Pascal-transformations }
#);
```

Note that an instantaneous qualification (see section 4) of *CurrentSelection* as an *IfImp* reference is made inside *Perform*. This is necessary in order to access the attributes of *IfImp*. The pattern *ReplaceCurrentSelection* used by *Perform* is an attribute of pattern *Sde*.

The *IfToWhileTransformation* is a simple tree-match transformation. In the same way more advanced context-sensitive transformations could be added to the Pascal-editor. A transformation that extends a <*Procedure-Identifier*> with a template for the list of actual parameters is an example of this. This list could be generated with the correct number of parameters, and the parameters could be specialized such that a <*Variable*>-nonterminal is inserted if the formal parameter is a <*Var-Parameter*>, an <*Exp*>-nonterminal if it is a <*Value-Parameter*>, etc.

## 7 The Semantic Level

As indicated by the investigation example in the previous section it is often useful for tools to be able to add attributes (operations, data) to the patterns of the context free level. A simple way to add semantic attributes is to let the tool programmer textually edit the patterns of the context free level. In a programming environment with many grammars and tools this is not satisfactory from a maintenance point of view. If semantic attributes have to be manually inserted into the patterns this has to be done each time changes are made to the grammar. From a structuring point of view it would be an advantage if the definition of semantic attributes could be kept separate from the generated patterns.

One solution could be to extend the grammar format to include definition of semantic attributes. But this would be inflexible since it would mean that every time a new tool needs a new attribute, the grammar has to be changed.

Another solution would be to let the tool programmer define sub-patterns of the generated patterns of the context free level. The semantic attributes could then be added in these sub-patterns. But ordinary sub-patterns of the generated patterns will not work. Other tools that do not know about these sub-patterns, e.g. the editor, will create instances of the patterns they know about, and these instances will thus not have the added attributes. This means that a tool programmer who adds new sub-patterns cannot use ASTs generated by other tools.

The solution that has been chosen is to make further use of the virtual mechanism of BETA. The patterns generated for the context-free level are actually generated as virtual patters. They thus have the form

```
    A :< P (# ... #)
```

instead of

```
    A : P (# ... #)
```

which was described in section 5. The context free level of the example grammar would then look as follows:

```
Small : TreeLevel
(# ...
   Imp :< Cons(# #);

   IfImp :< Imp
   (# Condition: Son1(# Cat ::< Exp #);
      ThenPart: Son2(# Cat ::< Imp #);
      ElsePart: Son3(# Cat ::< Imp #)
   #);
   ...
#);
```

Semantic attributes may now be added by creating a sub-pattern of *Small* and extend the specialization of the generated patterns.

The investigation tool of the previous section would then have the form

```
InvestigateSmall : Small
(# ImpCount,... : @ Integer;

   Imp ::<
   (# Count :< (# do ImpCount.Add1; inner #);
   #);

   IfImp ::<
   (# Count ::< (# do ... #);
   #);
   ...
#);
```

New attributes can be added by other tools using the same scheme but declaring sub-patterns of *InvestigateSmall* instead of *Small.*

## 8 Status & Conclusion

A prototype of the metaprogramming system has been in use in the Mjølner project since October 1986. It has been used in a number of tools, including a syntax directed editor [Borup & Sandvad 86], a prettyprinter, a mini-browser for BETA, a transformation system [Berg et al. 88], and a Masterscope-like ([Interlisp]) system for BETA programs called *Betalyzer* [Hagemann & Pedersen 87]. The metaprogramming system has been integrated with a parser which creates an AST from a text.

Grammars are described by a *metagrammar*, and the *generator* which generates the context free level for an application grammar uses the context free level of the metagrammar to access the grammar.

The system has also been used for implementing a mail handler [Sørgaard 87]. The mail handler is an example of an application that goes beyond the original intentions with the metaprogramming system. The different types of mail are described by a grammar. This grammar has then be used for generating a set of classes. In general it turns out that the system may be used for generating an implementation of classes where the interface may be described by a grammar.

The experience with using an object-oriented framework for modelling ASTs has so far been satisfactory. Sub-classing has made it possible to integrate the tree level, the context free level and the semantic level into a consistent interface to the AST. This permits tools applicable for all programming languages and more language specific tools to use the same ASTs.

The context free level provides the programmer with much more structure and security than the tree level alone. Consider the declaration *P*: ↑ *ProcDcl*. The body part of *P* may be denoted by *P*.*Body*. If only the tree level is used, the corresponding declaration and denotation will be *P*: ↑ *AST* and *P*.*son2*. This is first of all less readable. In addition it is the programmers responsibility that *P* actually denotes an AST for <*ProcDcl*> otherwise *P*.*son2* will not return an AST for <*Block*>. This type of errors occurred frequently during the development of the BETA compiler, which only makes use of the tree level. (The metaprogramming system has been developed after the BETA compiler.)

With respect to metaprogramming, the system allows for the same degree of flexibility as is available in Lisp. By having a unique well defined representation of programs it is easy for a Mjølner user to implement his own tools. Since the whole Mjølner environment is organized as a set of classes and objects the possibilities for extended existing tools are in general good.

Finally it may be stressed that the grammar of a given language is a complete specification of the AST interface at the context free level. If a programmer is familiar with the rules for generating classes from a grammar, then the grammar alone may be used as an interface specification.

Further development of the metaprogramming system will take place during the final period of the Mjølner project. This includes a better support for specifying nonterminals that may derive the empty string. For the time being there is an *Optional* rule type for this purpose, but it needs some improvements

*Acknowledgement.* A number of individuals have contributed to the work described in this paper. We wish to thank the members of the Mjølner and Scala group and other users of the system for their constructive critique and assistance during the development of the system. We would also like to thank Görel Hedin, Jørgen Lindskov Knudsen, Bent Bruun Kristensen, Elmer Sandvad, Pål Sørgaard and the anonymous referees for many helpful comments on this paper. This work has been supported by the Mjølner project and *The Danish Natural Science Research Council*, FTU grant no. 5.17.5.1.25.

## References

[Abelson & Sussman 85] H. Abelson and G.J. Sussman with J. Sussman : *Structure and Interpretation of Computer Programs*, The MIT Press 1985.

[Berg et al. 88] F. Berg, M. Larsen, T. Pedersen: *Master thesis*, Computer Science Department, Aarhus University, (in preparation).

[BETA 85] B.B. Kristensen, O.L Madsen, B. Møller-Pedersen, K. Nygaard: *An Algebra for Program Fragments*, Proceedings ACM SIGPLAN 85 Symposium on Programming Languages and Programming Environments, Seattle, Washington, June 1985.

[BETA 87] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard: The BETA Programming Language — Part 1: Abstraction Mechanisms — Part 2: Multi-Sequential Execution. *In: B.D. Shriver, P.Wegner (ed.),* Research Directions in Object Oriented Programming, *MIT Press, 1987.*

[BETA 88] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard: Coroutine Sequencing in BETA. *To be presented at: Hawaii International Conference on System Sciences - 21, January 5-8, 1988.*

[Borup & Sandvad 86] K. Borup, E. Sandvad: *Editor Specification* Mjølner Report DK-SYS-10.2, Sysware/Aarhus University, September 1986.

[Cameron & Ito 84] 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, pp. 20-54

[Hagemann & Pedersen 87] A. Hagemann, B. Møller-Pedersen: *Betalyzer, A MasterScope for BETA*, Unpublished manuscript, Norwegian Computing Center, 1987.

[Hedin 86] G. Hedin: *Incremental Semantic Analysis in Mjølner* Mjølner Report no. S-LTH-12.1, Lund Institute of Techonology, December 1986.

[Interlisp] W. Teitelman, L. Masinter: *The Interlisp Programming Environment*, Computer, 14:4, April 1981.

[Mjølner 86] H.P. Dahle, M. Löfgren, O.L. Madsen, B. Magnusson (eds.): *The Mjølner Project — A Highly Efficient Programming Environment for Industrial Use.* Mølner Report No. 1, Aarhus, Lund, Malmö, Oslo, 1986.

The Mjølner Project is a joint venture between Elektrisk Bureau A.S., Oslo, Telelogic AB, Malmö, and Sysware Aps., Aarhus in cooperation with The Norwegian Computing Center, Oslo, Lund Institute of Technology, Lund, Aarhus University, Aarhus and Aalborg University Centre, Aalborg. The project has been initiated by *Nordforsk, Copenhagen*, (the Nordic Cooperative Organization for Applied Research) which also coordinates the project. The project is partly funded by a grant from *The Nordic Fund for Technology and Industrial Development.*

[Nørmark 87] K. Nørmark: *Transformations and Abstract Presentations in a Language Development Environment*, Ph.D. Thesis, Computer Science Department, Aarhus University, DAIMI PB-222, Feb. 1987.

[Sheil 83] B. Sheil : *Power Tools for programmers*, Datamation, Vol. 29, No. 2, February 1983

[Simula] *Data Processing - Programming Languages - SIMULA*, Swedish Standard SS 63 61 14, 1987, ISBN 91-7162-234-9.

[Smalltalk] A. Goldberg, D. Robson: *SMALLTALK-80: The Language and its Implementation*, Addison Wesley 1983

[Sørgaard 87] P. Sørgaard: *Heimdal - A Mjølner Mail handler* Mjølner Report DK-SYS 15.1, AArhus University, November 1986.

[Teitelbaum & Reps 81] T. Teitelbaum, T. Reps: *The Synthesizer Generator.* Proceedings ACM SIGSOFT/SIGPLAN 84 Software Engineering Symposium on Practical Software Developments, Pittsburgh, Penn., ACM SIGPLAN Notices 19,5, May 1984.

[Winograd 83] T. Winograd : *The Aleph specification language*, Stanford University, (unpublished) 1983.