The Mjølner BETA System
Metaprogramming System
Reference Manual

Mjølner Informatics Report
MIA 91-14(1.3)
August 1996

Contents

The Metaprogramming System

Metaprograms

A number of tools in the Mjølner BETA System 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 pattern is generated. The syntactic hierarchy of the grammar is mapped into a corresponding pattern hierarchy. This object-oriented representation of programs is further exploited by including a set of more general patterns that view a program as an abstract syntax tree and by allowing the user to add semantic attributes in sub-patterns.

Grammar based

The Mjølner BETA System 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. The Mjølner BETA System is primarily aimed at supporting the object-oriented programming style.

Common representation: Abstract Syntax Trees (AST)

All metaprogramming tools in the Mjølner BETA System 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. 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 the Mjølner BETA System, 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.

S-expression

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 patterns (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.

AST modelled using patterns

An object-oriented model of the ASTs has been developed as part of the Mjølner BETA System. An AST is modelled as an instance of a pattern. There is a pattern corresponding to each syntactic category (nonterminal) of the grammar. ASTs derived from a syntactic category are then modelled as instances of the corresponding pattern.

The pattern `IfImp` corresponds to the syntactic category `<IfImp>`. Instances of pattern `IfImp` then model ASTs that may be derived from `<IfImp>`.

The grammar hierarchy is modelled by a corresponding pattern hierarchy. E.g. if the nonterminal `<Imp>` may derive `<IfImp>`, `<WhileImp>` etc., then the pattern `Imp` will be a super-pattern of `IfImp`. The pattern 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 BETA System tools and other metaprograms all are able to use the same representation of programs.

Grammar based interface

The *grammar-based* interface described above results in a set of patterns for each language. A metaprogram using the grammar-based interface will thus be *language specific* since it uses the set of patterns generated from the grammar of the actual language. A number of tools are language specific in the sense that usually one exists for each language. Examples of tools that benefit from using the grammar-based interface are: semantic checkers, program analysers, 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 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 metaprogramming system may be accessed at three levels.

Three level interface

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 three levels are also modelled by a pattern hierarchy. A generated context-free level is a subpattern of tree level, and a semantic level is a subpattern of the context-free level for the language in question.

In the metaprogramming system, 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 leads to the integration of the grammar-based interfaces with the tree level and semantic level described above.

The implementation language of the metaprogramming system is BETA. This means that all metaprograms are written in BETA. However, metaprograms can manipulate ASTs of any context-free language.

Yggdrasil

The metaprogramming system is also known under the name *Yggdrasil*. This name is used in a few operations, and in some of the diagnostic messages from the system. The name originates in the Nordic mythodology, where Yggdrasil is the name of the *"Tree of Life"*.

1 Introduction to Context-Free Grammers

The theory of formal grammars is very extensive, and we will only describe here the very basics of formal grammars. Several textbooks dealing with formal grammars exists and their application as a basis for describing programming languages.

Formal grammars may be divided into several classes, of which the most important classes are the *right-linear* grammars, the *context-free* grammars and the *context-sensitive* grammars. Right-linear grammars are identical to regular expressions.

The most important result is that context-free grammars can be generated and recognized effectively by automated tools such as parsers, whereas context-sensitive grammars are undecidable in general.

Context-free grammars play an important role in the definition of programming languages and is therefore also the foundation for the grammar-based tools of the metaprogramming system. Before going into details with the particular grammar formalism used in the metaprogramming system, we will give a quick introduction to context-free grammars.

Context-free grammar

Any context-free grammar is defined by means of a set of *terminals*, a set of *nonterminals*, a set of *productions* (also called *rules*), and one *startsymbol*. A terminal is a string of characters (e.g. BEGIN in a Pascal grammar). A nonterminal is a special symbol that may derive other symbols. Productions are the means for specifying the rules for deriving sentences and sentential forms from the grammar. We say that the grammar is able to derive a set (possible indefinite) of *sentences*. A sentence consists solely of terminals (e.g. a Pascal program is a sentence derived from a Pascal grammar). A *sentential form* is like a sentence, except that nonterminals may be present in a sentential form. I.e. a sentential form is not fully derived (the remaining nonterminals have not been expanded).

Terminals are denoted by w_i, nonterminals by $\langle A_i \rangle$, productions by $\langle A \rangle ::= w_0 \langle B_0 \rangle w_1 \langle B_1 \rangle \ldots$. The startsymbol is a nonterminal, from which all derivations according to the grammar are defined to constitute the language, defined by the grammar. There is one special symbol, *empty*, which stands for the empty terminal (string).

To illustrate these concepts, a small part of the Pascal grammar is given[1]:

A grammar

```
<program>      ::= program <name>
                     <declarations>
                   begin
                     <statements>
                   end.
```

[1] Please note, that this grammar specification is not fully consistent with the actual grammar specification syntax of the metaprogramming system (see chapter 7). The primary difference is, that terminals in the actual grammars to be used by the metaprogramming system must be enclosed in single quotes (e.g. `'begin'` instead of **`begin`**.

```
<statements>   ::= <statement> ; <statements>
<statements>   ::= empty
<statement>    ::= if <condition>
                   then <statement>
                   else <statement>
<statement>    ::= <name> := <value>
<condition>    ::= <name>
<declarations> ::= <declaration> ; <declarations>
<declarations> ::= empty
<declaration>  ::= <name> : <name>
```

<name> may be any identifier, <value> may be true or false

In this grammar, program, begin, end, ".", ";", ":=", ":", if, then, and else are terminals, whereas <program>, <name>, <declarations>, <declaration>, <statements>, <statement>, and <condition> are all nonterminals. The nonterminal <program> is the startsymbol.

This grammar may e.g. generate the sentence:

Examples

```
program P
V:T;
begin V := true; end.
```

and the sentential form:

```
program P
V:T;
begin if <condition> then <statement> else V := false; end.
```

Derivation

From this sentential form, several derivations are possible. Derivations are defined by the productions of the grammar. A derivation consists of substituting a nonterminal in the sentential form with the right-side of one of the productions having this nonterminal on the left-side of the "::=" symbol of the production. This implies that <condition> may be substituted with any legal name and <statement> with either an assignment statement or an if statement.

It is important to note, that if one takes another nonterminal as the startsymbol, the language that can be derived from that nonterminal, will be a sublanguage of the original language. That is, if we choose <statements> as the startsymbol, we will get the sublanguage of legal statements in Pascal.

The above discussion is taken in terms of derivations (i.e. the grammar generating the possible sentences defined by the grammar). The observations are equally important when we are talking about parsing a string of characters, and evaluate whether or not the string is a legal sentence according to the grammar.

1.1 Structured Context-Free Grammars

The grammar formalism used in the metaprogramming system is a variant of context-free grammars. The main reason for introducing this formalism is to make it possible automatically to generate pattern definitions from a grammar.

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

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

Rules in structured context-free grammar

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>::= w0 <t1:A1> w1 ... <tn:An> wn
   ```

   where `<A0>`, `<t1:A1>`, ..., `<tn:An>` are nonterminal symbols and `w0`, `w1`, ..., `wn` are possibly empty strings of terminal symbols. This rule describes that `<A0>` derives the string `w0 <A1> w1... <An>`. 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.

4. An *optional rule* has the following form:

   ```
   <A>::? <B>
   ```

   where `<B>` is a nonterminal. The nonterminal `<A>` may generate either the same strings as `<B>` may generate, or the empty string (i.e. nothing).

Predefined nonterminals

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 have a tag-name, like `<Title:NameAppl>`.

LALR(1)

A nonterminal may only appear once on the left-hand side of a rule, and the complete grammar must be LALR(1)[2]. The limitations on the rules which can be used in a

---

[2] However, this restriction is only neceaasry if the grammar is to be used for parsing purposes (see chapter 7 for more details).

structured context-free grammar do not restrict the class of languages that can be described. Any context-free language may be generated by a structured context-free grammar. 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.

Abstract syntax tree (AST)

There is one important representation for sentential forms and sentences of any context-free grammar, the *abstract syntax tree*. An abstract syntax tree (for short AST) represents how a sentential form (or sentence) has been derived from the grammar (or how it has been constructed by the parser). The AST does not represent the terminals of the grammar, only the involved nonterminals. The nodes in an AST are productions and the branches in the AST signifies derivations of the nonterminals involved in the production in that node. The leaves of the AST are lexems, if the AST represents sentences. If the AST represents a sentential form, leaves may also be nonterminals. ASTs are a very convenient representation of programs and many manipulations of programs may be specified as manipulations on the underlying AST.

Mapping context-free grammar into data types

A context-free grammar for a language induces an abstract syntax that may be used to give an AST-definition. In the Mjølner BETA System, 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 *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`.

- If <A> appears in an optional rule:

  ```
  <A> ::? <B>
  ```

  then no category `A` is defined (to be discussed later).

- Otherwise the super-category of `A` is `Cons`.

The inheritance hierarchy of the generated patterns 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 BETA currently does not support multiple inheritance, there is the additional restriction that the hierarchy must be tree structured. That is, the following grammar will not be a legal grammar:

```
<A> ::| <B> | <C>
<B> ::+ <D>
<C> ::= <E> terminal <F>
```

since `<B>` will have both `List` and `A` as super-category.

1.1.1 Example of Structured Context-Free Grammar

Below an example of a structured context-free grammar is given[3].

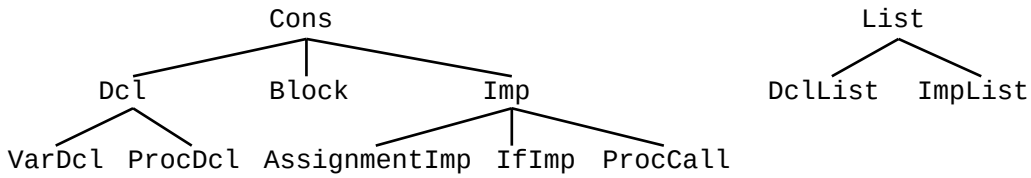
Example grammar

```
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 syntactic categories of a structured context-free grammar 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:

Hierarchy



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>`.

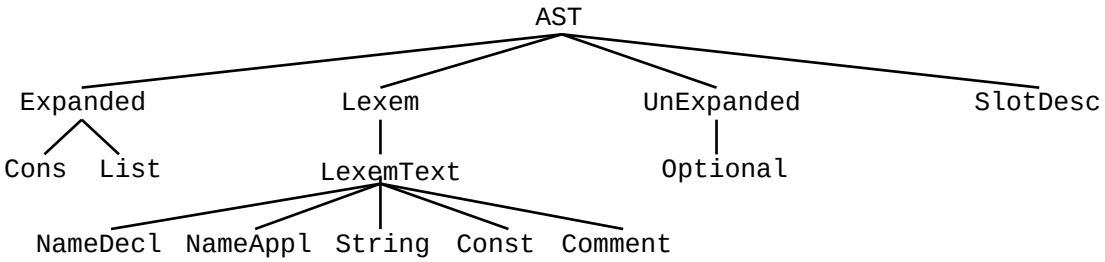
[3] Please note, that this grammar specification (as the previous grammar) is not fully consistent with the actual grammar specification syntax of the metaprogramming system (see chapter 7).

2 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 therefore be integrated with a level where the AST is viewed as an ordinary tree. This is straight forward using subclassing. The patterns generated from the grammar are all subpatterns of the general patterns Cons and List. These patterns are actually subpatterns of more general patterns describing ASTs as ordinary trees. In this section these general patterns are described. The general patterns are called the tree level.

AST is modelled as an instance of the pattern AST

At the *tree level*, an AST is modelled as an instance of the pattern `AST`. The pattern `AST` is further specialized into a number of sub-patterns. Some of these sub-patterns correspond to the rule types of a structured context-free grammar. The tree level corresponds to an ordinary data type for a tree. The specialization hierarchy for the patterns defined in the tree level is:



The following is a verbal description of these patterns:

Tree level patterns

`AST` describes all ASTs. Operations of this pattern are: `Symbol` returns the nonterminal symbol of the AST. `Kind` returns the type of the AST (e.g. `List`). `Father` returns the father. `SonNo` returns the index of this AST in the list of sons of the `father` of this AST. `NextBrother` returns the next brother of this AST in the list of sons of the `father` of this AST. `AddComment` updates an associated comment, `GetComment` returns an associated comment, and `HasComment` tests whether a comment is associated with this AST. `HasSemanticError` and `SemanticError` is used by language-dependent tools to mark ASTs with information on semantic errors in the AST. `Copy` returns a copy. `Match, Equal` and `Lt` performs different kinds of comparisons of ASTs, and `copy` enables copying an entire AST. Finally, `NearestCommonAncestor` returns the nearest common ancestor of this AST and some other AST. `GetAttribute`, `putAttribute`, `getNodeAttribute`, `putNodeAttribute`, `getSlotAttribute`, `putSlotAttribute`, `getSlotNodeAttribute` and `putSlotNodeAttribute`, are operations for accessing and changing the different attributes, that are specified for this kind of ASTs. The number of attributes are defined as part of the grammar specification. Finally, the putCommentProp and getCommentProp is used for associating properties directly with individual ASTs. For a description of properties, see chapter 7.

`Expanded` describes ASTs that are expanded into trees (i.e. something has been derived from the nonterminal of this AST). `Get` returns a son at a given position. `Put` updates a son at a given position. `NoOfSons` returns the number of sons of

this AST. `Scan` iterates over the sons in this AST. `SuffixWalk` and `SuffixWalkforProd` perform preorder traversal of the tree with this node as root. `Insert` will insert an AST before a given son. Finally, `expanded` defines `getson1`, ..., `getson9` and `putson1`, ..., `putson9` for direct access to the given son number.

`Cons` describes all nodes derived by a constructor rule. `Delete` deletes a son at a given position.

`List` describes all nodes derived by a list rule. `SonCat` describes the category of the ASTs of the list. `Delete` deletes an AST with a given position and `Insert` inserts an AST at a given position. `Append` inserts an AST as the last son in the list. `NewScan` iterates over the ASTs in this list.

`Lexem` describes all nodes derived by one of the predefined nonterminals (e.g. Const). No special operations.

`LexemText` describes leaves having textual contents. `GetText` returns the textual contents. `PutText` updates the textual contents. `CurLength` returns the length of the textual contents. `GetChar` and `putChar` allows for accessing and changing the individual chars in the textual contents. `Clear` empties the textual contents.

`NameDecl` describes all name declarations. `ScanUsage` iterates over the name applications that use this declaration. This information is context-sensitive and if used it must be set up by a language specific tool. `AddUsage` and `removeUsage` is used by these language-specific tools to set-up and remove such informations.

`NameAppl` describes all name applications. `GetDecl` returns a reference to the AST, containing the declaration of this name application. This information is context-sensitive and if used it must be set up by a language specific tool. `NextUsage` returns the next usage (reference to an AST) of this nameAppl, if any exists. `DeclSet` tells whether any language-specific tools have setup declaration information for this name application.

`String` describes all strings. No special operations.

`Const` describes all numeric constants. `GetValue` returns the value of the constant, `PutValue` updates the value of the constant.

`Comment` describes comments associated with ASTs. `CommentType` returns the type of this comment.

`UnExpanded` describes ASTs where nothing yet has been derived from the nonterminal of this AST. `NonterminalSymbol` returns the kind of nonterminal associated with this AST (Note that `this(AST).symbol` returns `unExpanded`). The kind of nonterminals are defined in the `kinds` object. The available kinds are: `kinds.interior`, `kinds.unExpanded`, `kinds.optional`, `kinds.nameDecl`, etc).

`Optional` describes ASTs that are unexpanded, and where the nonterminal symbol of this AST is defined by an optional rule. No special operations.

`SlotDesc` describes AST representing SLOTs. More on SLOTs later.

3 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 on the defining rule for A. For each rule type described in section 2, the list below describes the corresponding generated patterns.

Patterns in the context-free level

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
      (#
        getT1: getson1(# #)
        putT1: putson1(# #)
        getT2: getson2(# #)
        putT2: putson2(# #)
        ...
        getTn: getsonn(# #)
        putTn: putsonn(# #)
      #)
   ```

   where `P` is the super-category of `A`. There is an attribute corresponding to each nonterminal on the right side of the rule. The suffix of the `get-` and `put-` attributes (`Ti`) is the same as the corresponding tag-name.

   If `T` is an instance of `A`, the i'th sub-AST can be accessed and changed through the `get-` and `putTi` operations. The `put-` and `getsoni` patterns have enter (respectively exit) parameters such that an AST can be inserted as the i'th sub-AST by ... → `T.putTi` and will be delivered by `T.getTi` → ...

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

   ```
   A: List(# sonCat::< 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 a composition hierarchy and alternation rules into a classification hierarchy.

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

3.1 The Structure of the Context-free Level Interface

The above mentioned patterns that are generated by the metaprogramming system, are all declared local to a specialization of the `treelevel` pattern. That is, the structure of the context-free level interface is:

```
ORIGIN '~beta/ast/v5.1/astlevel'
--- astInterfaceLib: attributes ---
grammarName: treelevel
  (# ...
     (* declaration of the patterns from the grammar *)
     ...
     init::< (# do ... (* some initializations *) ... #)
  #);
```

`Treelevel` contains a local AST, `grammarAST`, decribing the grammar and a number of patterns (e.g. `newAST`, `newLexemText`, `newConst`, etc.) for instantiating new ASTs from this grammar, for identifying the version and name of the grammar, and facilities for parsing a text representation into a fragment. Finally, `treelevel` contains a `parser` attribute which can be used for parsing a text stream into an AST from the grammar in `grammarAST`.

The `applGram` fragment contains a specialization of `treeLevel`, called `applGram`. `ApplGram` makes the necessary setup for using the `treelevel` interface to any grammar.

3.2 Example of Structured Context-Free Grammar, cont.

The patterns generated for the example grammar are:

```
Small: TreeLevel
  (# Block: Cons(# getDclPart: getson1(# #);
                   putDclPart: putSon1(# #);
                   getImpPart: getson2(# #);
                   putImpPart: putson2(# #)
                #)
     Dcl: Cons(# #);
     VarDcl: Dcl(# getName:    getson1(# #);
                   putName:    putson1(# #);
                   getVarType: getson2(# #);
                   putVarType: putson2(# #)
                #)
     ProcDcl: Dcl(# getName: getson1(# #);
                    putName: putson1(# #);
                    getBody: getson2(# #);
                    putBody: putson2(# #)
                    #)
```

```
      Imp: Cons(# #);
      IfImp: Imp(# getCondition: getson1(# #);
                   putCondition: putson1(# #);
                   getThenPart:  getson2(# #);
                   putThenPart:  putson2(# #);
                   getElsePart:  getson3(# #);
                   putElsePart:  putson3(# #)
                #)
      ProcCall: Imp(# getProc: getson1(# #);
                      putProc: putson1(# #)
                   #)
      AssignmentImp: Imp(# getVar:   getson1(# #);
                           putVar:   putson1(# #);
                           getValue: getson2(# #);
                           putValue: putson2(# #)
                        #)
      DclLst: List(# #);
      ImpLst: List(# #);
      ...
   #)
```

3.3 Using the Context-Free Level

Consider the references:

```
P: ^ProcDcl; B: ^Block; N: ^nameDecl
```

`P.getBody` refers to the block of `P`, and after executing the assignment `P.getBody -> B[]`, `B` will refer to this block. `P.getName->N[]; N.GetText` will return the name of the procedure as a text.

Count the number of imperatives in the block

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`.

```
Small: Treelevel
  (# ImpCount, AssignmentCount,
     ProcCallCount, ICount: @Integer;
     Block: Cons
       (# Investigate:
            (# do
               0 → ImpCount → AssignmentCount
                 → ProcCallCount → IfCount;
               ImpPart.Scan(# do Current.Count #);
               ... (* Use ImpCount, ProcCallCount, ... *)
            #);
       #);
     ...
     Imp: Cons
       (# Count:< (# do ImpCount.Add1; inner #) #);
     IfImp: Imp
       (# Count::<
            (# do
               IfCount.Add1;
               ThenPart.Scan(#do Current.Count #);
```

```
                ElsePart.Scan(#do Current.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 patterns. 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.

Syntax directed editor

The next example will demonstrate how the syntax directed editor of the Mjølner BETA System 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 pretty-printer, 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: (* Presented in the menu *) @Text;
          Init:< (* Called when an instance is created *)
            (# do ... inner; ... #);
          ...
          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## = G.IfImp##)
                    -> Enabled
            #);
          Perform::<
            (# theIfImp: ^G.IfImp;
               theWhileImp: ^G.WhileImp;
               frag: ^fragmentForm;
            do CurrentSelection[] -> theIfImp[];
               (whileImp, frag[]) -> newAst
                 -> theWhileImp[];
               theIfImp.getCondition
                 -> theWhileImp.putCondition;
               theWhileImp -> ReplaceCurrentSelection;
            #);
       #);
     ... (* More Pascal-transformations *)
  #);
```

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.

Structure and security

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.getBody`. If only the tree level is used, the corresponding declaration and denotation will be `P: ^AST` and `P.getson2`. This is less readable and it is solely the responsibility of the programmer that `P` actually denotes an AST for `<ProcDcl>` otherwise `P.getson2` will not return an AST for `<Block>`.

4 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.

Semantic attributes

The semantic level of the metaprogramming system is therefore defined as part of the grammar. The semantic level allows the specification of the number of semantic attributes of each syntactic category in the grammar. The metaprogramming system will then ensure that the proper memory space is allocated for these attributes, and their values will be maintained by the system and stored along with the AST.

Encoded bytestream

It is important to know, that the persistent parts of the ASTs (i.e. the information stored in the files) is not in the form of BETA objects (instances of the AST patterns). The storage is instead in the form of an encoded bytestream, which also is the runtime representation of the ASTs, and the AST patterns are merely interfaces to this compact representation. This also implies that information in various types of objects as part of the instances of the AST patterns will not be stored when the AST is stored onto the disk. Such information is transient and cannot be shared with other tools. Since the semantic information is to be shared between tools, and stored onto the persistent representation of ASTs, it is specified in the grammar (in the attributes part, see chapter 6).

These persistent attributes of ASTs are accessed through the `putAttribute` and `getAttribute` of ASTs. The attributes are integer-valued and indexed, and may be used for any purpose.

Attribute slot

To enable the specification of transient properties of AST to be used at the semantic level, the metaprogramming system offers facilities for specifying that SLOTs should be inserted at various places in the generated context-free level interface. First of all, if a nonterminal is mentioned in the attribute part of the grammar, an attribute slot is automatically inserted in the pattern generated for that nonterminal (see chapter 6 and appendix 3). Secondly, the options part of the grammar may specify an identifier in the `substanceSlot` option in the options part of the grammar, and the result is that a descriptor slot with that name is inserted in the `treelevel` subpattern for the grammar (see chapter 6 and appendix 3).

These slots are used for specifying the transient properties in separate fragments, such that changes in the grammar (and thereby regeneration of the context-free level interface, does not destroy any semantic specifications. See appendix 3 for an example of this usage of semantic level slots.

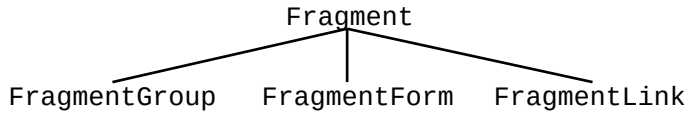
5 The Fragment System Interface

Storing ASTs using the fragment system

The metaprogramming system also contains an interface to the Fragment System of the Mjølner BETA System. The fragment system enables the management and manipulation of ASTs located on different files. The functionality of the fragment system allows the splitting of an AST into an number of sub-ASTs (sub-trees) by allowing some interior nodes in the original AST to be replaced by special nodes, called `slotDesc`. The AST, originally positioned at the position of that node, may by located on a totally different file (possibly along with other sub-ASTs). All these ASTs are called fragments. Fragments are named in an hierarchical name space, similar to the UNIX hierarchical file system. The full name of a fragment has the following structure: $/name_1/name_2/.../name_n/frag$, where $/name_1/name_2/.../name_n$ is called the path of the fragments and `frag` is called the local name of the fragment. The path of a fragment is the same as the full name of the father of the fragment. Fragments are usually located in files on the file system. Fragments are stored in two different formats: textual representation and binary representation. The binary representation is considered the essential representation of fragments, and the purpose of the textual representation is human reading, printing or parsing into the binary representation. If the fragment is created using tools based on the metaprogramming system, there is no reason for storing the textual representation since the binary representation contains all necessary information.

The fragment system interface of the metaprogramming system is used as the basis for the fragment system for BETA [MIA90-2].

The fragment system interface consists essentially of four major patterns:



`Fragment` is an abstract pattern for the three other patterns, implementing the following operations: `Name` accesses the local name of the fragment and `fullName` accesses the full name of the fragment. `DiskFilename` refers to the name of the file containing the binary representation, and `textFilename` refers to the file containing the textual representation. `Father` returns a reference to the father of this fragment, and `type` refers to the fragment type (group, form, or link). `Init` initializes the fragment, `reset` resets the fragment as if it has just been parsed from the textual representation, and `close` closes the fragment. `ModTime` returns the time for the last change to this fragment, `changed` indicates whether this fragment has been changed, `markAsChanged` sets the changed mark, and `checkDiskRepresentation` will check if the disk representation has been changed. Fragments may have properties associated with them. These properties are kept in the `prop` attribute, which is a list of properties, where each property carries one value. In a subsequent section, properties are discussed in more detail.

`Fragment` also contains a few operations, related to the BETA specific fragment system: `Origin` refers to the origin of the BETA fragmens, `bind` binds a fragment within this BETA fragment and `bindToOrigin` binds this fragment to its origin.

`FragmentForm` represents one single (sub-)AST. It contains the following operations: `Category` refering to the syntactic category of the root node of the AST in this fragmentForm. `Print` makes an almost readable dump of the fragmentForm. `Root` refers to the AST of this fragmentForm, `grammar` refers to the grammar, describing the grammar of the language of this fragmentForm, and `scanSlots` iterates through all slots in this fragmentForm. Finally, the BETA specific attribute `binding` refers to the slot to which this fragmentForm have been bound.

`FragmentGroup` represents a group of (sub-)ASTs. `FragmentList` is the list, containing all the fragments in this group. `Open` makes it possible to get access to a single fragment in the group, `scan` makes it possible to iterate through all the fragments of this group, and `parse` makes it possible to parse a file into a fragment group. `DefaultGrammar` refers to the grammar used for this `fragmentGroup`. `SetupOrigin`, `getBinding` and `getBETAbindings` can be used for accessing the slot bindings.

`FragmentLink` represents a link to some other fragment (e.g. the INCLUDE link in the BETA fragment system). It contains a reference to the fragment, and the various names for that link.

The fragment system interface also influences the AST interface a few places. First of all, an AST has a `frag` attribute, which refers to the fragment containing this AST. Secondly, `unExpanded` has the attributes `isSlot` and `theSlot`, where `isSlot` is true if this node in the AST represents a slot, and `theSlot` contains information (name and syntactic category) on the slot. Slots is inserted in the AST in the following way:

```
(# aSlot: ^slotDesc; anUnexpanded: ^unExpanded;
do ... -> newUnExpanded -> anUnexpanded[];
   newSlot -> aSlot[];
   'foo' -> aSlot.name;
   aSlot[] -> theUnexpanded.theSlot
#)
```

6 Generating a Metaprogramming Interface

Many of the tools in the Mjølner BETA System are available as generators, such that given a specific grammar for some language, new program development tools may be generated. These tools will offer extensive support of the specific language, such as parsing, pretty-printing, hyper structure editing (including syntax directed editing), and modularization etc. as offered by the fragment system. Finally, the meta programming system is available for that language.

This section will describe how to construct a structured context-free grammar, and generate a new set of program development tools that supports this language. The grammar-based tools are Parser, Pretty-printer, Editor, Fragment system, and the meta programming system.

6.1 Constructing a Grammar

We are assuming that an ordinary context-free grammar is given for the selected language. Strategies for constructing such a grammar can be found in most textbooks, that deals with compiler construction. In order for the grammar to be useful, it has to fullfill the following requirements:

1. The grammar must be converted into a structured context-free grammar. Since any context-free grammar can be converted into a structured grammar, this step should not cause major difficulties (except possibly for the restriction that the resulting hierarchy *must* be tree-structured, as discussed in chapter 2).
2. The resulting structured context-free grammar has to be LALR(1). This requirement may be ignored, if there is no need for generating a parser for that language (i.e. all programs in that language will be manipulated, using the editor and the meta programming system). Unless otherwise explicitly noted, we will, in the following, always be refering to the structured context-free grammar when we are discussing the grammar.
3. Along with the grammar, several additional properties of the grammar need to be specified, namely unused predefined nonterminals, the comment symbols of the grammar, the string symbol of the grammar, etc. All these options are described in chapter 6. We will discuss here only the most commonly used options.

The grammar definition is divided into five parts: the *fragment* part, tthe *naming* part, the *options* part, the *rules* part, and the *attributes* part: The naming part specifies the name of the grammar, the options part specifies the valid options of that grammar, the rules part contains the productions of the grammar, and the attributes part specifies for

each nonterminal, the number of semantic attributes defined at the semantic level for that nonterminal. The options part and the attributes part are optional parts and may thus be absent from the grammar definition. The meta grammar describing the language for grammar specification is given in appendix 1.

6.1.1 The Fragment Part

In order to make the grammar specification readable for the various grammar tools, it must start with a fragment form specification. The format of a fragment form specification for a grammar is:

-- *name*: aGrammar: metagrammar --

where *name* must be the name chosen for this grammar.

6.1.2 The Naming Part

The grammar definition must begin by naming the grammar. This is done in the *naming* part of the grammar. The naming part consists of one clause:

`Grammar` *name*`:`

where *name* must be the name chosen for this grammar.

6.1.3 The Options Part

The *options* part of the grammar contains various settings of variables, that control the way in which the grammar processor treats the productions in the rules part, and other issues.

Each option is specified by the name of the option followed by "=" followed by the value of the option. The valid options are:

- **version:** Defines the `version` number of the grammar. The version number is used by the meta programming system to ensure that different AST's handled in a meta program are using the same version of the grammar. Default is 'undefined'.
- **astVersion:** Defines the version of the metaprogramming system to be used for this grammar. Default is the same version as the version used for the generator.
- **comBegin:** Defines the string that signifies a beginning of comment in the language of the grammar. Default is '`(*`'.
- **comEnd:** Defines the string that signifies the end of comment in the language of the grammar. If the string is the empty string, end of line acts as the comment end string. Default is '`*)`'
- **stringChar:** Defines the string literal enclosing symbol (e.g. in a Pascal grammar, `stringChar` will be `"`). containing programs, written in the language of this grammar. Default is `'` (single quote)
- **Unused Lexem Terminals:** If not all lexem terminals (e.g. `<nameApp>` are used in the grammar, these should be marked as unused. This is done by using the name of the lexem terminal as the option name and associate the value `unused` to it (e.g `nameApp = unused`). Default is none.
- **substanceSlot:** Specifies an identifiers: `id`. The result of specifying this identifier is, tha the metaprogramming system will generate the following attribute in the generared context-free interface fragment: `id: <<SLOT id: descritor>>`.
- **subOf:** Specifies the pattern name to be used as the superpattern for the context free level patterns. Default is 'treelevel'.

- **BobsOptions:** String containing a comma separated list of options to be passed to the BOBS compiler-compiler. Default is '32,34,59'.
- **splitOnFiles:** Specifies that the generated BETA patterns, interfacing to the context free level should be split on the given number of files. Default is 1.
- **suffix:** Defines the file name suffix, that is expected on files, containing programs in the syntax of this grammar. Default is '`.text`'.
- **startsymbol:** Defines the startsymbol of the grammar. If no startsymbol option is defined for the grammar, the nonterminal on the left-hand side of the first production of the rules part of the grammar is chosen as the startsymbol of the grammar.

6.1.4 The Rules Part

The rules part of the grammar contains the specifications of the productions of the grammar.

The productions must follow the structure of a structured context-free grammar, as described ealier. Terminals have the form `'w'`, i.e. a string enclosed in single quotes (e.g. `'enter'`). Nonterminals has either the form `<A>` or `<t:A>`, where `t` is a tag-name, and `A` is the syntactical category. If no tag-name is provided, the name of the syntactic category is used as the default tag-name.

The complete grammar must be LALR(1), if a parser needs to be generated by the metaprogramming system.

6.1.5 The Attributes Part

The attributes part of the grammar is a specification of the additional memory, the metaprogramming system needs to allocate in order to be able to handle the semantic attributes defined at the semantic level of the grammar.

The attributes part is a list of

```
<nonterminalName>: number,
```

where `<nonterminalName>` is the name of a nonterminal of the grammar, and `number` is the size of the semantic attributes defined for this nonterminal. These semantic attributes are saved as part of the AST, when it is stored on some file. Please note, that for efficiency reasons, the number of attributes must be even (or zero).

As a side effect of specifying the nonterminal in the attributes part, that the generated context-free level interface pattern for that pattern will contain an attributes slot:

```
<<SLOT nonterminalNameAttributes: attributes>>
```

6.1.6 An Example Grammar

The following grammar will be used in the following to illustrate the various tools:

```
-- mylang: aGrammar: metagrammar --
grammar mylang:
rule
<module>        ::= 'module' <module:id> ';' <importOpt>
                    'begin' <statement> 'end';
<id>            ::= <nameDecl>;
<importOpt>     ::? <import>;
<import>        ::= 'import' <nameList> ';';
<nameList>      ::+ <nameDecl> ',';
<statement>     ::| <if> | <while> | <procCall>;
<if>            ::= 'if' <condition:exp>
                    'then' <thenPart:statement>
                    'else' <elsePart:statement> 'endif';
```

```
<while>         ::= 'while' <condition:exp>
                    'do' <statementList> 'end';
<statementList> ::* <statement> ';';
<exp>           ::| <expProcCall> | <text> | <number>;
<text>          ::= <string>;
<number>        ::= <const>;
<expProcCall>   ::= <procCall>;
<procCall>      ::= <nameAppl> '(' ')'
```

6.2 Generating the Grammar-Based Information

After having constructed the grammar, several grammar analysis tools needs to be invoked in order to analyse the grammar and generate the necessary information for the various tools. In the following discussion, we will assume the grammar is named `mylang` and the grammar is residing on the file `mylang-meta.gram`.

The naming conventions used here are mandatory:
A grammar must reside on a file with a name that is the name of the grammar (as specified in the `Grammar` clause of the grammar) followed by `-meta.gram`.

6.2.1 Generating the Metaprogramming Interface

In order to generate the predefined patterns that constitute the tree- and context-free level interface to the AST's generated by the grammar, we have to invoke the `generator` tool:

```
generator mylang
```

The `generator` checks whether the grammar is a valid structured context-free grammar, and generates the following files:

- `mylang-meta.ast`: This file contains an AST of the grammar itself. This AST is in accordance with the metagrammar specification given in appendix 1.
- `mylangcfl.bet`: This file contains BETA patterns, constituting the BETA interface (as described above) to the context-free level of AST's that will be generated by the parser (or other tools, such as the editor). If the grammar options (discussed later) specify that these patterns should be split on several files, the files `mylang2.bet`, `mylang3.bet`, etc. will also exist.
- `mylang-parser.bobs`: This file contains the grammar in a special format to be used by the parser generator (see below).

6.2.2 Generating Parser and Parser Tables

The next step is to analyse the grammar (to check that the grammar is LALR(1) and otherwise well-formed). This is done by the `bobsit` tool:

```
bobsit mylang
```

Besides analysing the grammar, the `bobsit` tool generates the file:

- `mylang-parser.btab`: This file contains the parser tables, needed by `mylang-parser.bet`.

`Bobsit` may find errors in the grammar (such as the grammar not being LALR(1), nonterminals that cannot be reached from the startsymbol, etc). If a parser is not to be used at all for the `mylang` grammar (i.e. all AST's of the grammar is generated by grammar-based tools) these errors may be ignored.

`Bobsit` is a revised version of the BOBS compiler generator.

6.2.3 Generating Pretty-printer Specification

Having analysed and checked the grammar, generated the AST interface to the grammar, generated the BETA interface to the AST's of the language, and generated the parser and parser tables, the next step is to generate a pretty-printer specification for the language. This is done by the `makepretty` tool:

```
makepretty mylang
```

`Makepretty` generates the default pretty-printer specification for the grammar `mylang` on the file:

- `mylang-pretty.pgram`

This default pretty-printer specification is often not the best possible pretty-printer specification for the given grammar. The default pretty-printer specification is therefore often modified in order give a better reflection of the semantical structure of the language. These modifications are done manually and discussed later.

6.2.4 Generating Pretty-printer Specification Tables

As the final step in generating the grammar-based information to be used by the various grammar-based tools, the pretty-printer specification tables need to be generated. This in done by the `morepretty` tool:

```
morepretty mylang
```

`Morepretty` analyses and checks the pretty-printer specification on the file `mylang-pretty.pgram` and generates the pretty-printer specification tables on the file:

- `mylang-pretty.ptbl`

6.2.5 Generating the Grammar-based Information Easily

In order to make it easier to run these four tools in the right sequence, a utility tool is available:

```
dogram mylang
```

which runs `generator`, `bobsit`, `makepretty` and `morepretty` in that sequence. Please note, that the `dogram` tool will overwrite any existing manually edited pretty-printer specifications. If these should be retained, either run the three other tools (e.g. except `makepretty`) manually, or copy the manually edited pretty-printer specification file to a safe place before invoking `dogram`.

6.2.6 Registering the new grammar

The grammar is now ready for being used by the different grammar-based tools in the Mjølner BETA System. The grammar-based tools uses a particular searching strategy, when trying to locate the grammar to be used for interpreting a given file (textual source file, of a group file, containing the ASTs). This searching strategy is implemented in the `findGrammar` fragment, described later. This strategy is the following:

1) First try to locate the grammar in the current directory.
2) Then try to find the grammar among the grammars specified in one of the grammar specification files:

a) First try among the grammars defined in the `MBSgrammars.text` file located in the current directory.

b) Then try among the grammars defined in the `MBSgrammars.text` file located in the HOME directory of the user.

c) Then try among the grammars defined in the `MBSgrammars_DEMO.text` file located in the `~beta` directory.

d) Finally try among the grammars defined in the `MBSgrammars_STD.text` file located in the `~beta` directory.

The first grammar found in this sequence will be used. The format of these grammar specification files are:

```
[[
-- INCLUDE 'filename of grammar'
-- INCLUDE 'filename of grammar'
... etc. ...
-- INCLUDE 'filename of grammar'
]]
```

The filename of the grammar should not include the `-meta` suffix. If we assume that your new grammar `mylang` is located in the `~you/mylang` directory, you can specify the grammar by inserting the following line in one of the grammar specification files:

```
-- INCLUDE '~you/mylang/mylang'
```

In order to have your new grammar being usable by the grammar-based tools, you therefore either have to have the grammar files located in the current directory, or have the grammar specified in one of the above mentioned grammar specification files. Since the `grammarsDEMO.text` and `grammarsSTD.text` files are located in the `~beta` directory, these files will not usually be modifyable by the normal users. You will therefore most often be specifying your grammar in one of the `.MBSgrammars.text` files mentioned above.

Important note for version 5.1: In this release, it is necessary to specify the grammar also in the file

`MBSgrammarsExt.text`

to make the grammar usable for the grammar-based tools (Sif, Freja, Frigg, Valhalla).

6.2.7 Using the Pretty-printer and the Hyper-structure Editor

After having specified your grammar, you will be able to use the new grammar with the pretty-printer and the hyper-structure editor.

In order to make a pretty-print of the `mylang` program on the file `tst.mylang`, the `pp` tool must be used:

```
pp tst.mylang > tst.pp
```

This will pretty-print the file `tst.mylang` (or `tst.ast`) and deliver the pretty-print on the file `tst.pp`. `pp` accepts two options:

`-p ppSpecFile`: name of a pretty-print specification to be used for this pretty-print

`-d ppDepth`: the depth of the AST to be pretty-printed. Can be used for making abstract interface descriptions.

The grammar is now also ready for use by the hyper-structure editor **`Sif`**. Usage of `mylang` in `Sif` is described in detail in the Sif manual [MIA 90-11]. However, in order to be able to utilize the automatic contraction facilities of `Sif`, the grammar specification needs to be augmented with one additional section, namely definition of the

`contractionCategories` property. This property is specified at the very beginning of the grammar specification (in the form of a fragment property). As an example, we can define the `contractionCategories` property for the `mylang` grammar:

```
contractioncategories
     module
     import
     if
     while;

-- mylang: aGrammar: metagrammar --

grammar mylang:
rule
<module>         ::= 'module' <module:id> ';' <importOpt>
                     'begin' <statement> 'end';
<id>             ::= <nameDecl>;
<importOpt>      ::? <import>;

etc. as previously
```

In this example, we have specified the rules `module`, `import`, `if` and `while` as the contractions categories. This implies, that `Sif` automatically will contract these parts when displaying a program derived from `mylang`. Please refer to the `Sif`.

Please note, that we in this example shows the entire file, including the fragment syntax needed:

```
-- mylang: aGrammar: metagrammar --
```

Note, that `aGrammar` and `metagrammar` are mandatory names, whereas `mylang` can be freely chosen.

6.2.8 Modifying the Pretty-print Specification

If you want to modify the specification, then modify the file: `mylang-pretty.pgram` either by means of a text editor or the pretty-print specification language-editor. After modifying the pretty-print specification `morepretty` is used again to create new tables for the editor. Note that if the grammar is modified, then the file `mylang-pretty.pgram` must be updated accordingly, e.g. a new production in the grammar might require a new production in the pretty-print specification grammar. In order to be able to modify the pretty-print specification the user must know the pretty-print algorithm.

The Pretty-print Algorithm

The pretty-print algorithm is an adaptive pretty-printer, i.e. the pretty-printer always tries to print as much as possible on each line. If the pretty-printed text cannot fit on one line, the pretty-print specification tells where to break the line. For each production in the grammar, there is a specification of how to pretty-print that production. Furthermore it is indicated where to associate a possible comment.

The pretty-printer algorithm takes as input a stream of tokens. A token is a text string, a break or a block.

- a text string is a sequence of characters
- a break specifies, where a line may be broken
- a block is specified by means of two delimitor tokens: [and].

A stream is defined by

1. a text string is a stream and
2. if s_1, s_2, ... s_n is a stream then [s_1 <break> s_2 <break> .. <break> s_n] is also a stream.

The algorithm gives a text string as output. The text string has a fixed maximal width. The block concept is used in the following way: the algorithm tries to break onto different lines as few blocks as possible according to the maximal width. If a block cannot fit on one line the block has to be broken. The breaks are used to specify where to break the block. A break has a length and an indention. The length specifies the number of single space characters to be written if the block is not broken. The indention specifies the number single space characters to be written (relative to the surrounding block) at the beginning of a new line if the block is broken.

There exist two types of blocks: consistent and inconsistent blocks. If a consistent block cannot be written on one line the substreams of the block will be written on separate lines. I.e. all breaks in the block will imply a line break. If an inconsistent block cannot be written on one line the substreams are only written on a separate line if they cannot be written on the rest of the current line.

The reason for the distinction between consistent and inconsistent blocks is that one might prefer:

```
        if <<condition:exp>> then
               <<thenPart:statement>>
        else
               <<elsePart:statement>>
        endif
to      if <<condition:exp>> then <<thenPart:statement>> else
        <<elsePart:statement>> endif
and     import a,b,
               c
to[4]   import a,
               b,
               c
```

The Pretty-print Specification

The pretty-printer is based on the algorithm just described, but because the internal representation of a document is an abstract syntax tree, the input stream to the pretty-printer algorithm must be generated from the AST. This step is called unparsing. The structure of an AST is described by means of a grammar. The pretty-print specification defines for each production in the grammar how it shall be unparsed.

Only those productions, that result in nodes in the AST, has a corresponding specification. The productions are constructor and list productions as described previously.

The grammar for pretty-print specifications is described in appendix 2.

The pretty-print specification of a constructor production has the following form:

```
<Constructor> ::= <ProductionName:nameAppl> '=' <Stream:ItemList>;
<ItemList>    ::* <Item>;
<Item>        ::| <Terminal> | <NontTerm> | <Break>
                  | <Block> | <CommentPlace>
```

The `ProductionName` is the name of the syntactic category on the left side of the corresponding production in the language grammar. The items in the stream can be terminals, nonterminals, breaks, blocks or comment places:

- A *terminal* can be a terminal symbol from the corresponding production. This is specified by `T:n`, where `n` is the terminal number in the production. A terminal can also be an explicit terminal symbol: `abc`.

---

[4] Naturally, we are assuming that the line width is not sufficient to have the entire
```
import a, b, c
```
on one single line

- A *nonterminal* is referring to the nonterminal symbol in the corresponding production or, if the nonterminal has been expanded, to the underlying sub-AST. Like terminals the nonterminals are numbered (`N:n`).
- A *break* is specified by `$<length>,<indention>`. The meaning of `length` and `indention` is described above. The default break `$$` has length 0 and indention 1.
- A *consistent block* is specified by `[c ... ]` and an *inconsistent block* is specified by `[i ... ]`.
- The *comment place* character `*` is used to indicate where to pretty-print the comment that may be associated with the node (corresponding to the production) in the AST. A comment place character shall be positioned after a terminal symbol and cannot be used in a list production.

The pretty-print specification of a list production has the following form:

```
<ListProd> ::= <ProductionName:nameAppl> '=' '(' <ListSpec> ')';
<ListSpec> ::= <Beginning:ItemList>
               '{' <BlockType> <Separator:ItemList> '}'
               <Ending:ItemList>
```

The list production specifies what is going to be pretty-printed before the list, between the list elements and after the list. A list is always surrounded by a block. Note the the block delimitors for lists are `{` and `}`.

The pretty-print specification must always start with a fragment form soecification (similar to the grammar specification). For a pretty-print specification the syntax is:

-- *name*: prettyprint: prettyprint --

where *name* must be the name chosen for this grammar.

An Example of Modifying the Pretty-print Specification

As mentioned the `makepretty` script generates a default pretty-print specification. This section illustrates the default pretty-print specification of the sample grammar and how the specification can be improved to obtain a more "pretty" pretty-printing.

The default pretty-print specification for `mylang` (`mylang-pretty.pgram`) might looks like[5]:

```
-- mylang: prettyprint: prettyprint --
PrettyPrintScheme mylangSpec
for mylang:
module   =  T:1 $$ * $$ N:1 $$ T:2 $$ N:2 $$ T:3 $$ N:3 $$ T:4;
id   =  N:1;
importOpt=  N:1 ;
import   =  T:1 $$ * $$ N:1 $$ T:2;
nameList =  ( {c T:1 $$ } );
statement=  N:1 ;
if   =  T:1 $$ * $$ N:1 $$ T:2 $$ N:2 $$ T:3 $$ N:3 $$ T:4;
while    =  T:1 $$ * $$ N:1 $$ T:2 $$ N:2 $$ T:3;
statementList =  ( {c T:1 $$ } );
exp  =  N:1 ;
text =  N:1;
number   =  N:1;
expProcCall   =  N:1;
procCall =  N:1 $$ T:1 $$ * $$ T:2
```

---

[5] This default pretty-print speficication is subject to changes. The default pretty-print specification on your system might differ from the one shown here. Especially will all construction nonterminals have consistent blocks surrounding their right-hand pretty-print specifications.

As can be seen the default specification uses default breaks; there are no blocks except in lists; the block type of a list is consistent and only the list separator is specified; comments are only associated with constructor productions and always after the first terminal symbol.

Let us look at the following source file:

```
-- test: module: mylang --
module pip;
(* Copyright 1992*)
(* Mjølner Informatics *)
import dyt, baat, olsen; (* some imported operations *)
begin
  while olsen() do
     if (* the test condition will be filled out later *)
<<condition:exp>>
     then
        dyt()
     else if <<condition:exp>> then baat() else <<elsePart:statemen
endif
     endif;
     <<statement>>
  end
end
```

Using the default pretty-printer specification as generated by the `makepretty` tool (see above) results in the following pretty-print[6]:

```
-- test: module: mylang --
module (* Copyright 1992 *)
(* Mjølner Informatics *) pip ; import
(* some imported operations *) dyt, baat, olsen ; begin while
olsen ( ) do
if
(* the test condition will be filled out later *)
<<condition: exp>>
then
dyt
(
)
else
if
<<condition: exp>>
then
baat
(
)
else
<<elsePart: statement>>
endif
endif;
<<statement>> end end
```

Because there are no blocks in the specification, the pretty-printer tries to write as much as possible on each line as can be seen on the first three lines of the program. Note that no blocks in the specification has the same effect as one surrounding inconsistent block. After the third line each item is written on a separate line. This is caused by the statementList, that introduces a consistent block of statements.

---

[6] Since the default pretty-print specification might differ on your system, the same applies for this pretty-print example.

This pretty-printing is certainly not very pretty, but only a few modifications of the specifiation improves the layout drastically. Consider the following modified specification:

```
-- mylang: prettyprint: prettyprint --
PrettyPrintScheme mylangSpec
for mylang:
module   =  [c [i T:1 $$ N:1 T:2 ] $$ * $$ N:2 $$ [c T:3 $1,2
N:3 $$ T:4] ];
id  =  N:1;
importOpt=  N:1 ;
import   =  [i T:1 $$ N:1 T:2 $$ *];
nameList =  ( {i T:1 $$ } );
statement=  N:1 ;
if  =  [c [i T:1 $$ * $1,3 N:1 $$ T:2 ] $1,3 N:2 $$ [i T:3 $1,3
N:3] $$ T:4 ];
while    =  [c [i T:1 $$ * $$ N:1 $$ T:2 ] $1,3 N:2 $$ T:3];
statementList =  ( {c T:1 $$ } );
exp  =  N:1 ;
text =  N:1;
number   =  N:1;
expProcCall   =  N:1;
procCall =  N:1 T:1 T:2 $$ *
```

The `test` program now looks much nicer:

```
-- test: module: mylang --
module pip;
(* Copyright 1992 *)
(* Mjølner Informatics *)
import dyt, baat, olsen; (* some imported operations *)
begin
  while olsen() do
     if (* the test condition will be filled out later *)
        <<condition: exp>> then
        dyt()
     else
        if <<condition: exp>> then
           baat()
        else <<elsePart: statement>>
        endif
     endif;
     <<statement>>
  end
end
```

The following changes have been made: In the `module` production some blocks have been introduced: one that surrounds the whole production, one that surrounds the header of the module and one that surrounds the body of the module. It is important that the block types of the first and the last block are consistent. The break between the module name and the ":" have been removed. The comment character have been moved. The break between the `begin` keyword and the `<statement>` has been changed to an indention of two characters.

In the `nameList` production, the block type has been changed to inconsistent. The `import` production is surrounded by an inconsistent block in order to "overload" the consistent block of the start production. In the `if` and `while` productions, blocks has been introduced and the indention has been changed. In the `procCall` production, the breaks has been removed.

The reader is recommended to try to understand the effect of these modifications in order to gain insight into the workings of the pretty-print specification.

6.3 Format of Source Files

The source files to be read by the different grammar tools (e.g. `pp` and `sif`) must all start by giving a specification of the fragment form (similar to grammar specifications and pretty-printer specifications).

The fragment form syntax for source files is:

```
-- name: category: grammar --
```

where *`grammar`* be the name for the grammar for this source file. *`Category`* must be the name of a construction nonterminal in the grammar, and finally *`name`* is the name chosen for this fragment form (mostly not significant). If ": *`grammar`*" is omitted, the BETA grammar is assumed.

Please note, that it is important that *`category`* is the name of a construction nonterminal, i.e. a nonterminal specified by a construction rule:

```
category ::= ...;
```

7 Fragment and AST Properties

The metaprogramming system enables associating properties with each fragment and with the individual nodes in an AST. The fragment properties are defined as part of the fragment syntax (e.g. the ORIGIN and INCLUDE properties). The AST properties may be attached to the nodes by metaprogramming tools.

The properties are defined as instances of the `propertyList` pattern (defined in the `property` fragment). Properties are lists of (*name*, *parameterList*) pairs. Name may be any text string, and parameterList is a list of values, where each value may be an integer, a textstring or a name (also a textstring).

The attributes of `propertyList` are: `addProp`, `findProp`, `deleteProp`, `scanProp` and `getProp`.

The fragment properties are available through the `prop` attribute of the `fragment` pattern. The `AST` properties may be accessed through the following attributes of the AST pattern: `setCommentProp`, `getCommentProp` and `hasCommentProp`. `SetCommentProp` makes it possible to associate a `propertyList` with an `AST`, and `getCommentProp` makes it possible to gain access to the `propertyList` associated with an `AST`, and finally `hasCommentProp` is used for testing whether a `propertyList` is currently associated with an `AST`.

Please note, that the current implementation of properties of ASTs implies that the properties replaces any comments, which might have been associated with the AST.

8 The Metaprogramming System Libraries

The metaprogramming system consists of a number of fragments: `astlevel`, `applgram`, `findgrammar`, `property`, `metagrammarcfl` and `metagramsemAtt`:

- **`astlevel.bet`** contains the metaprogramming interface to the ASTs and fragments, as described above.
- **`applgram.bet`** contains one single pattern (subpattern of `treelevel`), which defines the proper initializations etc. for utilizing the treelevel interface for any given grammar.
- **`findgrammar.bet`** contains the `grammarFinder` used in the Mjølner BETA System (i.e. `findgrammar` seeks for grammars the places where the Mjølner BETA System locates its grammars).
- **`metagrammarcfl.bet`** contains the context-free level interface for the metagrammar, thus defining the interface to any grammar information, maintained by the Mjølner BETA System. `metagrammarcfl` is used by tools that needs to know about the structure of the grammar for the ASTs they are working on (e.g. the `metaGrammarcfl` interface is used by the Sif editor to find out about the valid derivations of a given nonterminal).
- **`metagramsematt.bet`** contains the semantic level interface for the metagrammar, and defines the interface to the available semantic information for grammars (e.g. options).

9 Interface Description for the Metaprogramming System

Astlevel.bet

```
ORIGIN '~beta/basiclib/v1.5/betaenv';
INCLUDE '~beta/basiclib/v1.5/file'
        '~beta/sysutils/v1.5/pathhandler'
        '~beta/containers/v1.5/hashTable'
        'property';
BODY 'private/astPrivate';
(*
 * COPYRIGHT
 *       Copyright (C) Mjolner Informatics, 1986-93
 *       All rights reserved.
 *)
-- LIB: Attributes --
(* This fragment contains the tree level interface to the abstract syntax trees
 * and interface to the fragment library.
 *) (* idx: 2 *)
astInterface:
  (#
     <<SLOT astInterfaceLib:Attributes>>;
     yggdrasilVersion:
     (* describes the version of THIS(astInterface) *) (#  exit 'v5.1' #);
     ast:
     (* Basic class, which is the super-pattern of all patterns describing
      * abstract syntax trees.  Ast's are stored in a special purpose format
      * which is internally allocated in a repetition.
      *)
       (#
          <<SLOT astLib:Attributes>>;
          frag: (* where THIS(ast) belongs *)
            ^fragmentForm;
          symbol: (* the nonterminal symbol of THIS(ast) *)
            (# lab: @integer
            enter (#  enter lab ... #)
            exit
              (#
              ...
              exit lab
              #)
            #);
```

```
father:
(* return the father of THIS(ast) or NONE, if we are in the root
  (#
  exit (# as: ^ast ... exit as[] #)
  #);
nextBrother:
  (# brother: ^ast
  ...
  exit brother[]
  #);
sonNo:
(* returns the sonNo of THIS(ast) in the father node *)
  (# inx,finx,son: @integer
  ...
  exit son
  #);
kind:
(* return the subCategory of ast this node is *)
  (#
  exit (# kind: @integer ... exit kind #)
  #);
equal:
(* determines if THIS(ast) and another ast-reference points to th
 * same ast. This operations is to be used instead of testing
 * reference-equivalence directly: instead of testing
 *      a1,a2: ^ast;
 *      (if a1[]=a2[] then ... if);
 * you must test
 *      (if (a1[]->a2.equal) then ... if)
 *)
  (# comparedAst: ^ast;
  enter comparedAst[]
  exit (# eq: @boolean ... exit eq #)
  #);
nearestCommonAncestor:
(* find the nearest common ancestor of THIS(ast) and the ast
 * entered
 *)
  (# testAst,nca: ^ast; testSonNo,mySonNo: @integer
  enter testAst[]
  do ...
  exit
  (nca[],testSonNo,mySonNo)
  (* TestSonNo is the number of the son where father-chain of the
   * entered ast differs.  MySonNo is the number of the son where
   * father-chain THIS(ast) differs
   *)
  #);
lt:
(* Determine whether the ast entered or THIS(ast) will be met fir
 * in a preorder traversal of the tree. Return true if the ast
 * entered comes first
 *)
  (# testAst: ^ast; testSonNo,mySonNo: @integer
  enter testAst[]
  do ...
  exit (testSonNo < mySonNo)
  #);
putAttribute:
(* save an integer value as an attribute to THIS(ast) *)
  (# val,attributNo: @integer;
  enter (val,attributNo)
  ...
```

```
  #);
getAttribute:
(* get an integer-valued attribute *)
  (# attributNo,val: @integer;
  enter attributNo
  ...
  exit val
  #);
putNodeAttribute:
(* save an ast-reference as an attribute to THIS(ast) *)
  (# val: ^ast; attributNo: @integer
  enter (val[],attributno)
  ...
  #);
getNodeAttribute:
(* get an ast-reference - valued attribute *)
  (# attributNo: @integer; val: ^ast
  enter attributno
  ...
  exit val[]
  #);
putSlotAttribute:
(* save an integer value as an attribute to THIS(ast) *)
  (# val,attributNo: @integer;
  enter (val,attributNo)
  ...
  #);
getSlotAttribute:
(* get an integer-valued attribute *)
  (# attributNo,val: @integer;
  enter attributNo
  ...
  exit val
  #);
putSlotNodeAttribute:
(* save an ast-reference as an attribute to THIS(ast) *)
  (# val: ^ast; attributNo: @integer
  enter (val[],attributno)
  ...
  #);
getSlotNodeAttribute: (* get an ast-reference - valued attribute *)
  (# attributNo: @integer; val: ^ast
  enter attributno
  ...
  exit val[]
  #);
addComment:
(* add a commment to THIS(ast). Overwrites existing comments *)
  (# l: ^lexemText;
  enter l[]
  ...
  #);
getComment:
(* return the comment associated with THIS(ast) *)
  (#
  exit
    (# as: ^ast ... exit as[] #)
  #);
getNextComment: @|
(*
 * This is a special operation that only should be used by
 * the prettyprinter
 * A comment c for at subAST is organized as follows:
```

```
 * c = c1 c2 ... cn, where the positions of the ci's are:
 * c1 son1 c2 son2 c3 .... cn sonn cn+1
 * each ci can be further divided into a subsequence of comments
 * that must be prettyprinted separately.
 * NextComment scans all subcomments one of the time.
 * A call of nextComment returns the next subcomment in the
 * sequence of comments belonging to THIS(ast).
 *
 * if n is -2 the whole comment is empty and subcomment is none
 * if n is -1 the subcomment is empty and 'subcomment' is none
 * if n is 0 there is only one comment between the two sons
 * or it is the last subcomment
 * if n is 1 there are more than one subcomment and
 * 'subcomment' contains the current one
 * if n is 2 the whole comment has been scanned,
 * 'subcomment' contains the last one
 *
 * The representation of the comment looks like this:
 * ' xxx 21 yyy 2 zzz 21 aaa 2'
 *
 * where 1 (ascii 1) is the separator between the
 * subcomments and
 * 2 (ascii 2) is the subsequence separator
 *
 * and it should be prettyprinted like this:
 * [* xxx *] son1 [* yyy *] [* zzz *] son2 [* aaa *]
 *)
  (# subcomment: ^text; n: @integer
  do ...
  exit (subcomment[],n)
  #);
insertSubcomments:
(* This is a special operation that only should be used by
 * the editor
 * Inserts the subcomments with index inx (1..n)
 * Subcomments must include subsequence separators.
 * THIS(ast) must already have a comment.
 * An empty comment with separators
 * can be created using the prettyprinter.
 *)
  (# subcomments: ^text; inx: @integer
  enter (subcomments[],inx)
  do ...
  #);
setSubcomments:
(* This is a special operation that only should be used by
 * the editor
 * Sets the subcomments corresponding to index inx (1..n)
 * Subcomments must include subsequence separators.
 * If subcomments is empty, the existing subcomments
 * at index inx are deleted.
 * THIS(ast) must already have a comment.
 * An empty comment with separators
 * can be created using the prettyprinter.
 *)
  (# subcomments: ^text; inx: @integer
  enter (subcomments[],inx)
  do ...
  #);
getSubcomments:
(* This is a special operation that only should be used by
 * the editor
 * Returns subcomments with index inx (1..n), including
```

```
 * subsequence separators.
 * If the node has no comment or the subcomments are empty
 * the empty string is returned.
 *)
  (# subcomments: ^text; inx: @integer
  enter (inx)
  do ...
  exit subcomments[]
  #);
scanComments:
(*
 * A comment c for at subAST is organized as follows:
 * c = c1 c2 ... cn, where the positions of the ci's are:
 * c1 son1 c2 son2 c3 .... cn sonn cn+1
 * Each ci can be further divided into comments that must be
 * prettyprinted separately.
 * ScanComment scans all subcomments one of the time
 * calling INNER for each subcomment.
 * 'current' contains the current subcomment with indexes
 * inx (1..n, the ci number) and subinx (1..n, the number in
 * the subsequence)
 *)
  (# current: ^text; inx,subinx: @integer
  do ...
  #);
insertSubcomment:
(* Inserts subcomment with indexes inx and subinx
 * THIS(ast) must already have a comment.
 * An empty comment with separators
 * can be created using the prettyprinter.
 *)
  (# subcomment: ^text; inx,subinx: @integer
  enter (subcomment[],inx,subinx)
  do ...
  #);
setSubcomment:
(* Sets subcomment with indexes inx and subinx,
 * If subcomment is empty, the existing subcomment is deleted.
 * THIS(ast) must already have a comment.
 * An empty comment with separators
 * can be created using the prettyprinter.
 *)
  (# subcomment: ^text; inx,subinx: @integer
  enter (subcomment[],inx,subinx)
  do ...
  #);
getSubcomment:
(* Returns subcomment with indexes inx and subinx,
 * if the node has no comment or the subcomment is empty
 * the empty string is returned
 *)
  (# subcomment: ^text; inx,subinx: @integer
  enter (inx,subinx)
  do ...
  exit subcomment[]
  #);
hasComment:
(* tells if there is a comment associated with THIS(ast) *)
  (# has: @boolean ... exit has #);
hasCommentProp:
  (#
  exit (typeOfComment = 17)
  #);
```

```
getCommentProp:
  (# prop: ^propertyList;
  do ...
  exit prop[]
  #);
setCommentProp:
  (# prop: ^propertyList;
  enter (prop[])
  do ...
  #);
typeOfComment:
(* sets or returns the type of THIS(comment) *)
  (# type: @integer
  enter
    (#  enter type ... #)
  exit
    (#
    ...
    exit type
    #)
  #);
dump:< (* do a nearly human readable dump of THIS(ast) to a strea
  (# level: @integer; dmp: ^stream;
  enter (level,dmp[])
  ...
  #);
copy:
(* make a copy of THIS(ast) with all sons. The enter-parameter
 * tells which fragmentForm the copy shall belong to
 *)
  (#
     copyFrag: ^fragmentForm;
     astOverflow:< astInterfaceException;
     startingParsing:< (# do INNER #);
     fragmentNotExisting:< astInterfaceException
       (#  do true->continue; INNER #);
     grammarNotFound:< astInterfaceException;
     badFormat:< astInterfaceException;
     parseErrors:< astInterfaceException;
     fatalParseError:< astInterfaceException
       (# errNo: @integer enter errNo do INNER #);
     doubleFormDeclaration:< astInterfaceException;
     readAccessError:< astInterfaceException;
     writeAccessError:< astInterfaceException;
     writeAccessOnLstFileError:< astInterfaceException;
     EOSError:< astInterfaceException;
     noSuchFileError:< astInterfaceException;
     fileExistsError:< astInterfaceException;
     noSpaceLeftError:< astInterfaceException;
     otherFileError:< astInterfaceException;
     copyCatcher: @handler (* private *)
       (#  ... #)
  enter copyFrag[]
  exit
    (# as: ^ast
    do ...
    exit as[]
    #)
  #);
match:<
(* pattern-matching. Returns true if the entered ast match
 * THIS(ast)
 *)
```

```
  (# doesMatch: @boolean; treeToMatch: ^ast
  enter treeTomatch[]
  ...
  exit doesMatch
  #);
hasSemanticError:
(* returns true if THIS(ast) has semantic errors *)
  (#
  enter
    (# b: @boolean
    enter b
    do (@@ frag.a[index],b)->tos'%PutBits[1,1]'
    #)
  exit frag.a[index]->tos'%GetBits[1,1]'
  #);
semanticError: (* if hasSemanticError, this is the errorNumber *)
  (#
  enter
    (# errorNumber: @integer
    enter errorNumber
    ...
    #)
  exit
    (#
       errorNumber:
         @integer;

    ...
    exit errorNumber
    #)
  #);
stopYggdrasil:< astException;
astException: astInterfaceException
  (#
  do INNER ;
     msg.newLine;
     ' index = '->msg.puttext;
     (index)->msg.putInt;
     ' symbol = '->msg.puttext;
     (symbol)->msg.putInt;

  #);
<<SLOT astPrivateLib:Attributes>>;
index:
(* Private: architecture of an ast:
 *
 *           |      ....    |
 *           ----------------
 * index ->  |    prodno     |
 *           ----------------
 *           |  next brother | (if negative: -index to father)
 *           ----------------
 *           |   first son   | (for lexems: pointer to text)
 *           ----------------
 *           |first attribute|
 *           ----------------
 *           |     ....     |
 *
 *) @integer;
bit7: (* Private *)
  (# b: @boolean
  enter
    (#  enter b do (@@ frag.a[index],b)->tos'%PutBits[0,1]' #)
```

```
      exit frag.a[index]->tos'%GetBits[0,1]'->b
      #);
    copyPrivate:< (* Private *)
      (#
         theCopy: ^ast; theCopyInx: @integer; copyFrag: ^fragmentForm
      enter copyFrag[]
      ...
      exit theCopyInx
      #);

  do INNER
  #);
expanded: ast
(* this pattern describes all expanded ast *)
  (#
    <<SLOT expandedLib:Attributes>>;
    noOfsons:
    (* return the number of sons of THIS(expanded) *)
      (# sons: @integer;
      do ...
      exit sons
      #);
    get:
    (* get a son with a given son-number *)
      (# i: @integer;
      enter i
      exit
        (# as: ^ast ... exit as[] #)
      #);
    put:
    (* sets the entered ast to be a son of this son with a given
     * son-number
     *)
      (#
         i: @integer;
         s: ^ast;
         notSameFragment:< astException
         (* exception called if the entered ast is not in same fragme
          * as THIS(expanded)
          *)
           (#
           do INNER ;
              'Error in put. Inserted ast is not from same fragmentFc
                ->msg.putline;

           #);

      enter (i,s[])
      do ...
      #);
    scan:
    (* iterates over all sons *)
      (# current: ^ast; currentSonNo: @integer;
      do ...
      #);
    suffixWalk:
    (* make a preorder traversal of the tree with THIS(expanded) as
     * root. cutIf can be used to cut the traversal of some sub-ast's
     *)
      (#
         cutIf:<
           (# prod: @integer; toCut: @boolean
           enter prod
```

```
        do false->toCut; INNER
        exit toCut
        #);
      current: (* the ast-iterator *) ^ast;

  do ...
  #);
suffixWalkforProd:
(* make a preorder traversal of the tree with THIS(expanded) as
 * root.  Will only call INNER for ast's which have the symbol
 * 'prod'. cutIf can be used to cut the traversal of some sub-ast's
 *)
  (#
     scanCat:< ast;
     cutIf:<
       (# prod: @integer; toCut: @boolean
       enter prod
       do false->toCut; INNER
       exit toCut
       #);
     current: (* the ast-iterator *) ^scanCat;
     prod: @integer;

  enter prod
  do ...
  #);
insert:
(* insert an ast before a son with the given son-number. Must
 * externally only be called for lists
 *)
  (#
     i: @integer;
     s: ^ast;
     notSameFragment:< astException
     (* exception called if the entered ast is not in same fragment
      * as THIS(expanded)
      *)
       (#
       do INNER ;
          'Error in put. inserted ast is not from same fragmentForm '
            ->msg.putline;

       #);

  enter (i,s[])
  do ...
  #);
getson1:
(* optimized version of getson1: (# exit 1 -> get #) *)
  (#
  exit
  frag.a[index+1]->tos'%getShort[1]'->tos'%ShiftLeft[1]'
    ->frag.indexToNode
  #);
getson2: (#  exit 2->get #);
getson3: (#  exit 3->get #);
getson4: (#  exit 4->get #);
getson5: (#  exit 5->get #);
getson6: (#  exit 6->get #);
getson7: (#  exit 7->get #);
getson8: (#  exit 8->get #);
getson9: (#  exit 9->get #);
putson1: (# a: ^ast enter a[] do (1,a[])->put #);
```

```
      putson2: (# a: ^ast enter a[] do (2,a[])->put #);
      putson3: (# a: ^ast enter a[] do (3,a[])->put #);
      putson4: (# a: ^ast enter a[] do (4,a[])->put #);
      putson5: (# a: ^ast enter a[] do (5,a[])->put #);
      putson6: (# a: ^ast enter a[] do (6,a[])->put #);
      putson7: (# a: ^ast enter a[] do (7,a[])->put #);
      putson8: (# a: ^ast enter a[] do (8,a[])->put #);
      putson9: (# a: ^ast enter a[] do (9,a[])->put #);
      <<SLOT expandedPrivate:Attributes>>;
      dump::< (* Private *)
        (#  do ... #);
      match::< (* Private *)
        (#
        do ...
        #);
      copyPrivate::< (* Private *)
        (#  do ... #);

  do INNER ;
  #);
cons: expanded
(* describes ast's derived from a constructor-production *)
  (#
      <<SLOT consLib:Attributes>>;
      delete:
      (* delete a son with the given son-number. Inserts an unExpanded
       * instead
       *)
        (# sonnr: @integer;
        enter sonnr
        do ...
        #);
      dump::< (* Private *)
        (#
        do 'CONS'->dmp.puttext; INNER
        #)
  #);
list: expanded
(* describes ast's derived from a list-production *)
  (#
      <<SLOT listLib:Attributes>>;
      sonCat:< ast;
      newScan: (* iterates over all sons *)
        (# predefined:< (# current: ^Ast enter current[] do INNER #);
           a: ^ast;
           current: ^sonCat;
           currentSonNo: @integer;
        do ...
        #);
      append:
      (* append a son to the list *)
        (# a: ^ast;  enter a[] do (noOfSons+1,a[])->insert;  #);
      delete: (* delete the son with the given son-number from the list
        (# sonnr: @integer;
        enter sonnr
        do ...
        #);
      dump::< (* Private *)
        (#
        do 'LIST'->dmp.puttext;
           INNER
        #);
```

```
  #);
lexem: ast
(* describes all ast's derived from one of the predefined nonterminals *)
  (# <<SLOT lexemLib:Attributes>> #);
lexemText: lexem
(* describes all ast's having textual contents *)
  (#
     <<SLOT lexemTextLib:Attributes>>;
     getText: (* get the textual content *)
       (# t: ^text;
       do &text[]->t[]; ...
       exit t[]
       #);
     putText:
     (* set the textual content *)
       (# t: ^text;
       enter t[]
       do ...
       #);
     clear:
     (* clear the textual content *)
      ...;
     getChar: (* get a char *)
       (# index: @integer; ch: @char
       enter index
       do ...
       exit ch
       #);
     putChar:
     (* append a char to the textual content *)
       (# c: @char;
       enter c
       do ...
       #);
     curLength:
     (* sets or returns the length of the textual contents *)
       (# l: @integer
       enter
         (#
         enter l
         do ...
         #)
       exit
         (#
         ...
         exit l
         #)
       #);
     <<SLOT lexemTextPrivate:Attributes>>;
     dump::< (* Private *)
       (#
       do INNER ;
          '^'->dmp.put;
          getText->dmp.puttext
       #);
     copyPrivate::< (* Private *)
       (# theLexCopy: ^lexemText
       do theCopy[]->theLexCopy[];
          getText->theLexCopy.puttext;
          INNER
       #);
     match::< (* Private *)
       (#
```

```
            theMatchLexem: ^lexemText;
            theT,theMatchText: ^text;

          ...
          #)
    #);
nameDecl: lexemText
(* describes ast's derived from the predefined nonterminal <nameDecl>
    (#
       <<SLOT nameDeclLib:Attributes>>;
       addUsage:
       (* add an usage of THIS(nameDecl) *)
         (# user: ^nameAppl;
         enter user[]
         do ...
         #);
       removeUsage:
       (* marks that a nameAppl does not use THIS(nameDecl) any more *)
         (# userAppl: ^nameAppl;
         enter userAppl[]
         do ...
         #);
       scanUsage:
       (* iterates over all usages of THIS(nameDecl) *)
         (#
            current: ^nameAppl;
            user: @integer;
            currentFragmentForm: ^fragmentForm;

         ...
         #);

    exit prodNo.nameDecl
    #);
nameAppl: lexemText
(* describes ast derived from the predefined nonterminal <nameAppl> *)
    (#
       <<SLOT nameApplLib:Attributes>>;
       getDecl:
         (# user: @integer; declAst: ^ast;
         do ...
         exit declAst[]
         #);
       declSet:
         (# b: @boolean ... exit b #);
       nextUsage:
         (# nextNameAppl: ^nameAppl
         ...
         exit nextNameAppl[]
         #);

    exit prodNo.nameAppl
    #);
string: lexemText
(* describes ast derived from the predefined nonterminal <string> *)
    (# <<SLOT stringLib:Attributes>> exit prodNo.string #);
comment: lexemText
    (#
       <<SLOT commentLib:Attributes>>;
       commentType:
         (# type: @integer
         enter
           (#  enter type ... #)
```

```
        exit
          (#
          ...
          exit type
          #)
        #);
      copyPrivate::< (* Private *)
        (#  ... #);

  exit prodNo.comment
  #);
const: lexemText
(* describes ast derived from the predefined nonterminal <const> *)
  (#
     <<SLOT constLib:Attributes>>;
     newConstType:
       (# b: @boolean
       ...
       exit b
       #);
     putValue:
       (# val: @integer;
       enter val
       do ...
       #);
     getValue:
       (# val: @integer;
       do ...
       exit val
       #);
     dump::< (* Private *)
       (#  do INNER ; '&'->dmp.put; getText->dmp.putText #);
     copyPrivate::< (* Private *)
       (# theCnCopy: ^const;
       do theCopy[]->theCnCopy[]; getText->theCnCopy.putText;
       #);

  exit prodNo.const
  #);
unExpanded: ast (* describes ast's which have not been derived yet *)
  (#
     <<SLOT unExpandedLib:Attributes>>;
     nonterminalSymbol:
     (* describes which symbol, THIS(unExpanded) may derive.
      * THIS(unexpanded).symbol returns prodNo.unExpanded
      *)
       (#
       enter
         (# val: @integer
         enter val
         do (@@ frag.a[index+1],val)->tos'%putShort[1]'
         #)
       exit frag.a[index+1]->tos'%GetSignedShort[1]'
       #);
     isSlot: bit7 (#  #);
     theSlot:
       (#
       enter
         (# o: ^slotDesc
         enter o[]
         ...
         #)
       exit
```

```
            (# sd: ^slotDesc
            ...
            exit sd[]
            #)
          #);
        sy: (* Private *) @integer;
        dump::< (* Private *)  (#  ... #);
        copyPrivate::< (* Private *)
          (#  do ... #);

    do prodNo.unExpanded->sy;
       INNER
    exit sy
    #);
  optional: unExpanded
  (* nodes in the tree which are empty (for optionals) are generated as
   * instances of 'optional'
   *)
    (#
       <<SLOT optionalLib:Attributes>>;
       dump::< (* Private *)
         (#  do '#'->dmp.put; INNER #);

    do prodNo.optional->sy
    #);
  slotDesc: ast
    (#
       <<SLOT slotDescLib:Attributes>>;
       name:
         (#
         enter
           (# t: ^text;
           enter t[]
           do ...
           #)
         exit
           (# c: ^comment
           ...
           exit c.getText
           #)
         #);
       category:
         (# f: ^unExpanded do father->f[];  exit f.nonterminalSymbol #);
       isBound: (* Private *) @boolean;
       node: (* Private *)
         (# father: @integer; ff: ^fragmentForm
         ...
         exit (father,ff[])
         #);
       copyPrivate::< (* Private *)
         (#
         do ...
         #);
       dump::< (* Private *)  (#  ... #);

    exit prodNo.slotDesc
    #);
  nonterminalSymbol:
  (* may be used to describe symbol numbers *)
    (#
       <<SLOT nonterminalSymbolLib:Attributes>>;
       symbol: @integer;
       predefined:
```

```
        (#
        exit (symbol <= 0)
        #);
      isLexem:
        (#
        exit
        ((symbol < - 2) and
         (symbol > - 7))
        #)
  enter symbol
  exit symbol
  #);
(*--------------------- Fragment patterns -----------------------------*)
formType: (#  exit 0 #);
groupType: (#  exit 1 #);
linkType: (#  exit 2 #);
fragment:
(* Abstract super-pattern for fragments.  A fragment has a unique
 * identification in form of a hierarchical name: '/foo1/foo2/.../foon';
 * '/foo1/foo2/...' is called the path of the fragment; 'foo' is called
 * the (local) name.  Only name needs to be stored since the path can be
 * fetched recursively from the father.
 *)
  (#
     <<SLOT fragmentLib:Attributes>>;
     name:
     (* exit the local name of THIS(fragment) *)
       (#  enter nameT[] exit nameT[] #);
     fullName: (* exit the full name (path/name) of THIS(fragment) *)
       (# n: ^Text ... exit n[] #);
     father:
       (#
       enter fatherR[]
       exit fatherR[]
       #);
     isOpen:
     (* returns true if THIS(fragment) has been opened *) @boolean;
     close:< (* Close THIS(fragment) *)
       (#
       do (if changed then markAsChanged if);
          INNER ;
          false->isOpen
       #);
     type: (* returns one of formType, groupType, linkType *)
       (#  exit fragType #);
     init:<
       (#
       do &propertyList[]->prop[]; prop.init; false->changed; INNER
       #);
     reset:<
     (* reset fragmentForm to be as if it has just been parsed up *)
       (# do INNER #);
     modtime: (* time of last visit of file-representation *) @integer;
     markAsChanged: (* call this when you want to save some changes *)
       (#
          astOverflow:< astInterfaceException;
          startingParsing:< (# do INNER #);
          fragmentNotExisting:< astInterfaceException
            (#  do true->continue; INNER #);
          grammarNotFound:< astInterfaceException;
          badFormat:< astInterfaceException;
          parseErrors:< astInterfaceException;
          fatalParseError:< astInterfaceException
```

```
      (# errNo: @integer enter errNo do INNER #);
    doubleFormDeclaration:< astInterfaceException;
    readAccessError:< astInterfaceException;
    writeAccessError:< astInterfaceException;
    writeAccessOnLstFileError:< astInterfaceException;
    EOSError:< astInterfaceException;
    noSuchFileError:< astInterfaceException;
    fileExistsError:< astInterfaceException;
    noSpaceLeftError:< astInterfaceException;
    otherFileError:< astInterfaceException;
    markAsChangedCatcher: @handler (* Private *)
      (#  ... #)
  do ...
  #);
changed: @boolean;
checkDiskRepresentation:<
(* called when it should be checked, if the disk-representation
 * of the fragment have been changed by another fragment.  If it
 * have, the internal state of the fragment is updated according
 * the disk-representation
 *)
  (# haveBeenChanged: @boolean; error: ^stream
  enter error[]
  do ...
  exit
  haveBeenChanged
  #);
diskFileName:< (* returns the filename of the disk-representatior
  (# t: ^text do &text[]->t[]; INNER exit t[] #);
textFileName:<
(* returns the file-name of the text-representation of
 * THIS(fragment)
 *) (# t: ^text do &text[]->t[]; INNER exit t[] #);
origin: (#  enter originR[] exit originR[] #);
bind:< (* bind the fragment f inside THIS(fragment) *)
  (# f: ^fragmentForm; op: ^slotDesc
  enter f[]
  ...
  exit op[]
  #);
bindToOrigin:
  (#
     f: ^FragmentForm; op: ^slotDesc
  enter f[]
  ...
  exit op[]
  #);
setupOrigin:
  (# error: ^stream
  enter error[]
  do ...
  #);
prop: ^propertyList;
pack:<
(* Private: pack representation into byte stream *)
  (# do INNER #);
unpack:< (* Private: unpack rep. from bytestream *)
  (# error: ^stream enter error[] do INNER #);
bindMark:
(* Private: true => attempting to bind slots in THIS(fragment) *)
  @boolean;
nameT: (* Private *) ^text;
fullNameT: (* Private *) ^text;
```

```
      fatherR: (* Private: the enclosing group *) ^fragmentGroup;
      fragType: (* Private *) @integer;
      originR: (* Private: Attribute where THIS(fragment) 'belongs' *)
        ^fragment;
      ffNameSeparatorChar: (* Private *) (#  exit '-' #);
      catcher: handler (* Private *)
        (#  ... #);

  do INNER
  #);
newFragmentLink:
(* returns a new instance of fragmentLink *)
  (# l: ^fragmentLink do &fragmentLink[]->l[]; l.init;  exit l[] #);
fragmentLink: fragment
(* This type of fragment is a link to another fragment *)
  (#
     <<SLOT fragmentLinkLib:Attributes>>;
     f: ^fragment;
     fullNameOfLink: ^text;
     localName: ^text;
     diskFileName::<
       (#
       do fatherR.diskFileName->t[];
          t->t
       #);
     init::<
       (#
       ...
       #);
     close::< (* Private *)
       (#
       do (if (f[] <> none) then f.close if); none ->f[]
       #);
     unpack::< (* Private *)
       (#  do ... #);

  #);
newFragmentGroup:
(* returns a new instance of fragmentGroup *)
  (# g: ^fragmentGroup do &fragmentGroup[]->g[]; g.init;  exit g[] #);
fragmentGroup: fragment (* This is a group of fragments *)
  (#
     <<SLOT fragmentGroupLib:Attributes>>;
     scan:
       (# current: ^fragment
       do fragmentList.scan
            (#
            do (if current.f[]=none then
                   screen[]
                     ->current.open;

               if);
               current.f[]
                 ->
                   THIS(scan).
                     current[];
               INNER scan
            #)
       #);
     scanSlots:
       (# current: ^slotDesc
       do fragmentList.scan
            (# ff: ^fragmentForm;
```

```
        do (if current.type=formType then
               screen[]
                 ->current.open
                 ->ff[];
               ff.scanSlots
                 (#
                 do current[]
                      ->
                        THIS(
                        scanSlots).current[];
                    INNER scanSlots
                 #)
            if)
        #)
  #);
open:
(* This operation opens a local fragment, localPath, of this grou
 * LocalPath may be a local name of the form 'foo' or a local pat
 * 'foo1/foo2/.../foon' which will be interpreted local to this
 * group
 *)
  (#
     localPath: ^text;
     f: ^fragment;
     g: ^fragmentGroup;
     error: ^stream;
     groupInx,dirInx: @integer;
     astOverflow:< astInterfaceException;
     startingParsing:< (# do INNER #);
     fragmentNotExisting:< astInterfaceException
       (#  do true->continue; INNER #);
     grammarNotFound:< astInterfaceException;
     badFormat:< astInterfaceException;
     parseErrors:< astInterfaceException;
     fatalParseError:< astInterfaceException
       (# errNo: @integer enter errNo do INNER #);
     doubleFormDeclaration:< astInterfaceException;
     readAccessError:< astInterfaceException;
     writeAccessError:< astInterfaceException;
     writeAccessOnLstFileError:< astInterfaceException;
     EOSError:< astInterfaceException;
     noSuchFileError:< astInterfaceException;
     fileExistsError:< astInterfaceException;
     noSpaceLeftError:< astInterfaceException;
     otherFileError:< astInterfaceException;
     openCatcher: @handler (* Private *)
       (#  ... #);

  enter
  (localPath[],
   error[])
  do ...
  exit f[]
  #);
alreadyOpen: booleanValue
(* returns true, if the local fragment, localPath, of this group
 * already open in this group.  LocalPath is as in open.
 * INNER is executed if localPath is part of THIS(fragmentGroup),
 * and f[] refers to the opened fragment.
 *)
  (# localPath: ^text; f: ^fragment
  enter localPath[]
  do ...
```

```
  #);
close::<
  (#
  ...
  #);
namedClose:
  (# localPath: ^text; f: ^fragment
  enter localPath[]
  ...
  #);
fragmentListElement:
  (#
     f: ^fragment;
     type: @integer;
     name: ^text;
     localName,
       fullNameOfLink:
     (* ought to be in a subpattern, Only o.k. for link-type *)
       ^text;
     open:
       (# error: ^stream
       enter error[]
       ...
       exit f[]
       #);
     <<SLOT fragmentListElementPrivate:Attributes>>
  #);
fragmentList:
  ^fragmentListDescription;
fragmentListDescription: containerList
  (#
     element::< fragmentListElement;
     deleteLocalName: (* delete the fragment with the local name n *)
       (# n: ^text (* the local path *)
       enter n[]
       ...
       #);
     find:
     (* find a local fragment. If the fragment is not open return
      * NONE
      *)
       (# n: ^text (* the local path *) ; r: ^fragment
       enter n[]
       ...
       exit r[]
       #);
     open:
     (* Find a local fragment. If the fragment is not open then
      * open it
      *)
       (#
          f: ^fragment;
          n: ^text;
          e: ^element;
          error: ^stream;
          removeHeadingSlashes:
          (* this routine removes '/' 's at the head of a
           * text
           *)
            (# t: ^text; ch: @char
            enter t[]
            do 0->t.setPos;
               loop:
```

```
                    (if (t.get->ch)='/' then restart loop if);
                    (if (t.pos > 1) then
                        (1,t.pos-1)->t.delete
                    if)
                  exit t
                  #);

            enter (n[],error[])
            ...
            exit f[]
            #);
          insertFragment:
            (#
               f: ^fragment;
               newElement: ^element;
               alreadyThere:< (* exception, which may be called *)
                astInterfaceException;
               astOverflow:< astInterfaceException;
               startingParsing:< (# do INNER #);
               fragmentNotExisting:< astInterfaceException
                 (#  do true->continue; INNER #);
               grammarNotFound:< astInterfaceException;
               badFormat:< astInterfaceException;
               parseErrors:< astInterfaceException;
               fatalParseError:< astInterfaceException
                 (# errNo: @integer enter errNo do INNER #);
               doubleFormDeclaration:< astInterfaceException;
               readAccessError:< astInterfaceException;
               writeAccessError:< astInterfaceException;
               writeAccessOnLstFileError:< astInterfaceException;
               EOSError:< astInterfaceException;
               noSuchFileError:< astInterfaceException;
               fileExistsError:< astInterfaceException;
               noSpaceLeftError:< astInterfaceException;
               otherFileError:< astInterfaceException;
               addFragmentHandler: @handler (* Private *)
                 (#
                 ...
                 #)
            enter f[]
            do ...
            #);
          addFragment: insertFragment (#  do newElement[]->append #);
          insertFragmentBefore: insertFragment
            (# before: ^theCellType
            enter before[]
            do (newElement[],before[])->insertBefore
            #);
          insertFragmentAfter: insertFragment
            (# after: ^theCellType
            enter after[]
            do (newElement[],after[])->insertAfter
            #);
          <<SLOT fragmentListDescriptorPrivate:Attributes>>
        #);
      defaultGrammar:
        ^treeLevel;
      saveAs: (* save THIS(FragmentGroup) using the name fullname *)
        (# fullname: ^Text;
           astOverflow:< astInterfaceException;
           startingParsing:< (# do INNER #);
           fragmentNotExisting:< astInterfaceException
             (#  do true->continue; INNER #);
```

```
      grammarNotFound:< astInterfaceException;
      badFormat:< astInterfaceException;
      parseErrors:< astInterfaceException;
      fatalParseError:< astInterfaceException
        (# errNo: @integer enter errNo do INNER #);
      doubleFormDeclaration:< astInterfaceException;
      readAccessError:< astInterfaceException;
      writeAccessError:< astInterfaceException;
      writeAccessOnLstFileError:< astInterfaceException;
      EOSError:< astInterfaceException;
      noSuchFileError:< astInterfaceException;
      fileExistsError:< astInterfaceException;
      noSpaceLeftError:< astInterfaceException;
      otherFileError:< astInterfaceException;
      saveAsCatcher: @handler (* Private *)
        (#  ... #)
  enter fullname[]
  do ...
  #);
saveBackup:
(* save THIS(FragmentGroup) using the name diskFileName+ext *)
  (# ext: ^Text;
     astOverflow:< astInterfaceException;
     startingParsing:< (# do INNER #);
     fragmentNotExisting:< astInterfaceException
       (#  do true->continue; INNER #);
     grammarNotFound:< astInterfaceException;
     badFormat:< astInterfaceException;
     parseErrors:< astInterfaceException;
     fatalParseError:< astInterfaceException
       (# errNo: @integer enter errNo do INNER #);
     doubleFormDeclaration:< astInterfaceException;
     readAccessError:< astInterfaceException;
     writeAccessError:< astInterfaceException;
     writeAccessOnLstFileError:< astInterfaceException;
     EOSError:< astInterfaceException;
     noSuchFileError:< astInterfaceException;
     fileExistsError:< astInterfaceException;
     noSpaceLeftError:< astInterfaceException;
     otherFileError:< astInterfaceException;
     saveBackupCatcher: @handler (* Private *)
       (#  ... #)
   enter ext[]
  do ...
  #);
restoreBackup:
(* restore THIS(FragmentGroup) using the name diskFileName+ext *)
  (# ext: ^Text;
     astOverflow:< astInterfaceException;
     startingParsing:< (# do INNER #);
     fragmentNotExisting:< astInterfaceException
       (#  do true->continue; INNER #);
     grammarNotFound:< astInterfaceException;
     badFormat:< astInterfaceException;
     parseErrors:< astInterfaceException;
     fatalParseError:< astInterfaceException
       (# errNo: @integer enter errNo do INNER #);
     doubleFormDeclaration:< astInterfaceException;
     readAccessError:< astInterfaceException;
     writeAccessError:< astInterfaceException;
     writeAccessOnLstFileError:< astInterfaceException;
     EOSError:< astInterfaceException;
     noSuchFileError:< astInterfaceException;
```

```
      fileExistsError:< astInterfaceException;
      noSpaceLeftError:< astInterfaceException;
      otherFileError:< astInterfaceException;
      restoreBackupCatcher: @handler (* Private *)
        (#  ... #)
    enter ext[]
    do ...
    #);
  diskFileName::<
    (#
    ...
    #);
  textFileName::< (#  ... #);
  isRealOpen:
    (# opened: @Boolean;
    ...
    exit opened
    #);
  realOpen: (* only to be used by the compiler *)
    (# astOverflow:< astInterfaceException;
      startingParsing:< (# do INNER #);
      fragmentNotExisting:< astInterfaceException
        (#  do true->continue; INNER #);
      grammarNotFound:< astInterfaceException;
      badFormat:< astInterfaceException;
      parseErrors:< astInterfaceException;
      fatalParseError:< astInterfaceException
        (# errNo: @integer enter errNo do INNER #);
      doubleFormDeclaration:< astInterfaceException;
      readAccessError:< astInterfaceException;
      writeAccessError:< astInterfaceException;
      writeAccessOnLstFileError:< astInterfaceException;
      EOSError:< astInterfaceException;
      noSuchFileError:< astInterfaceException;
      fileExistsError:< astInterfaceException;
      noSpaceLeftError:< astInterfaceException;
      otherFileError:< astInterfaceException;
      realOpenCatcher: @handler (* Private *)
        (#  ... #)
    do ...
    #);
  parse: (* for parsing a fragmentGroup *)
    (#
      groupParser:
       ...;
      parseErrors:< (* exception called if parse-errors *)
       astInterfaceException;
      fatalParseError:< astInterfaceException
        (# errNo: @integer enter errNo do INNER #);
      doubleFormDeclaration:<
      (* exception called if two fragmentForms with the same name
       astInterfaceException;
      input: @file;
      error: ^stream;
      ok: @boolean
    enter (input.name,error[])
    do groupParser
    exit ok
    #);
  init::<  (#  ... #);
  bind::<
    (#
    do ...
```

```
  #);
getBinding:
(* Get the bindings of the slot within THIS(fragmentGroup).  All
 * bindings are delivered.  For each binding, found is called.  The
 * elements of THIS(fragmentGroup) must be fragmentForms or
 * fragmentLinks to such.
 *)
  (#
     mark: @
       (#
          f: ^fragmentGroup;
          inserted: @boolean;
          scan:
            (# current: ^fragmentGroup;
            do ...
            #);
          elm:
          (* Private *) (# f: ^fragmentGroup; succ: ^elm #);
          head: (* Private *) ^elm;

       enter f[]
       ...
       exit inserted
       #);
     markRelatedFragments:<
       (# f: ^fragment;
       enter f[]
       do (if (f[] <> none) then INNER if)
       #);
     found:<
       (# theBinding: ^fragmentForm
       enter theBinding[]
       do trace.getBinding
            ->tracer
              (#
              do 'binding found '->dmp.puttext;
                 theBinding.fullName->dmp.puttext;

              #);
          INNER
       #);
     sl: ^slotDesc
  enter sl[]
  ...
  #);
getBETABindings:
 getBinding
(* get bindings the BETA way; search origin and include fragments
 *)
  (#
     markRelatedFragments::<
       (#
       do ...
       #);

  #);
(********** PRIVATE PART ********************)
pack::< (* Private *)
  (#  ... #);
unpack::< (* Private *)
  (#
  do ...
  #);
```

```
      checkDiskRepresentation::< (* Private *)
        (#  do init; error[]->unpack #);
      isDirectory:
      (* Private: true if the group is not a 'real' group but a
       * directory
       *) @boolean;
      backupExt: ^Text;
      controller: @ (* used by the control module in the compiler *)
        (#
           status: @integer;
           ancestorTime: @integer;
           ancestorsChecked: @boolean;
           doneCheck: @boolean;
           groupT: @Integer;
           printName: ^text;

        #);

    #)
    (* fragmentGroup *)
    ;
  newFragmentForm: (* returns a new instance of fragmentForm *)
    (# g: ^treeLevel; f: ^fragmentForm
    enter g[]
    do &fragmentForm[]->f[]; g[]->f.grammar[]; f.init;
    exit f[]
    #);
  fragmentForm: fragment
  (* This is the basic form of a fragment defined by means of a general
   * sentential form
   *)
    (#
      <<SLOT fragmentFormLib:Attributes>>;
      category:
        (# sy: @integer
        ...
        exit sy
        #);
      theGsForm: (#  exit (root.index,THIS(fragmentForm)[]) #);
      fragNode: (#  exit (0,THIS(fragmentForm)[]) #);
      print:
        (#
        do 'Print called of fragmentForm '->screen.puttext;
           fullName->screen.puttext;
           screen.newLine;

        #);
      binding: (* The SLOT bound by THIS(fragmentForm) *) ^slotDesc;
      modificationStatus: @integer;
      root:
      (* the root symbol of the ast kept in the array.  Set by the
       * parser
       *) ^ast;
      recomputeSlotChain:
        (#  do ...;  #);
      scanSlots:
      (* access operations: scan all SLOTs in THIS(fragmentForm) *)
        (# inx: @integer; current: ^slotDesc;
        ...
        #);
      grammar: ^treeLevel;
      indexToNode:
        (#
```

```
                inx: @integer;
                as: ^ast;
                indexOutOfRange:<
                 astInterfaceException;
                noSuchSymbol:< astInterfaceException;
                grammarGenRefArrayError:< astInterfaceException;

              enter inx
              do ...
              exit as[]
              #);
            <<SLOT fragmentFormPrivate:Attributes>>;
            a: (* Private *) [initialLength] @integer;
            curtop: (* Private: current heapTop in the array a *) @integer;
            initialLength:< (* Private *)
              (# max: @integer do 200->max; INNER exit max #);
            firstSlot:
            (* Private: The index of the first SLOT in the array a. The SLOTs
             * are linked together through the 'usage-field' of SLOTs
             *) @integer;
            diskFileName::< (* Private *)
              (#  do fatherR.diskFileName->t[] #);
            textFileName::< (* Private *)
              (#  do fatherR.textFileName->t[] #);
            import: @ (* Private *)
            (* An indexed collection of fragments referred by
             * THIS(fragmentForm)
             *)
              (#
                 impL: ^list;
                 inxC: @integer;
                 element: (# n: ^text; f: ^fragmentForm #);
                 list:
                   (#
                      noOfElements:<(# nu: @integer;  do 10->nu; INNER exit nu
#);
                      l: [noOfElements] ^element;

                    #);
                 <<SLOT fragmentFormImportPrivate:Attributes>>
              #);
            rootInx: @integer;
            init::< (* Private *)
              (#
              ...
              #);
            reset::< (* Private *)
              (#  ... #);

          #);
        astFileExtension:
        (* exits the filename extension for AST files on the particular
         * architecture (the extension differs e.g. for big- and little endian
         * architectures).  See e.g. initialization in astBody.bet
         *) (#  exit astFileExt[] #);
        parserFileExtension:
        (* exits the filename extension for parser table files on the particular
         * architecture (the extension differs e.g. for big- and little endian
         * architectures).  See e.g. initialization in astBody.bet
         *) (#  exit parserFileExt[] #);
        ppFileExtension:
        (* exits the filename extension for pretty-printer table files on the
         * particular architecture (the extension differs e.g. for big- and
```

```
 * little endian architectures).  See e.g. initialization in astBody.b
 *) (#  exit ppFileExt[] #);
astFileExt: (* Private *) ^text;
parserFileExt: (* Private *) ^text;
ppFileExt: (* Private *) ^text;
(************** END The Fragment Library END **************)
(* top: * The top root of the fragment library *
 *        @fragmentGroup;
 *)
top: @
  (#
     init: (#  ... #);
     groupTable: @HashTable
       (#
          element::
            (#
               fullname: ^Text;
               g: ^FragmentGroup;
               open:
                 (# error: ^Stream;
                 enter error[]
                 ...
                 exit g[]
                 #);

            #);
          dummy: @Element;
          hashFunction::
            (# inx: @Integer;
            do L:
               (for i: 13 repeat
                 e.fullname.lgth-i+1->inx;
                 (if inx < 1 then leave L if);
                 e.fullname.T[inx]+value->value;

               for)
            #);
          equal::
            (#  do left.fullname[]->right.fullname.equal->value #);
          rangeInitial::  (#  do 117->value #);
          find:
          (* find a fragment group. If the fragment is not open returr
           * NONE
           *)
            (# fullName: ^text (* the path *) ; g: ^fragmentGroup
            enter fullName[]
            ...
            exit g[]
            #);
          open:
          (* Find a local fragment. If the fragment is not open then
           * open it
           *)
            (#
               g: ^fragmentgroup;
               fullName: ^text;
               e: ^element;
               error: ^stream;
               removeHeadingSlashes:
               (* this routine removes '/' 's at the head of a
                * text
                *)
                 (# t: ^text; ch: @char
```

```
            enter t[]
            do 0->t.setPos;
               loop:
               (if (t.get->ch)='/' then restart loop if);
               (if (t.pos > 1) then
                   (1,t.pos-1)->t.delete
               if)
            exit t
            #);

      enter (fullName[],error[])
      ...
      exit g[]
      #);
    <<SLOT topTablePrivate:Attributes>>
  #);
open:
(* This operation opens a fragmentgroup file: fileName
 *)
  (#
     fileName: ^text;
     g: ^fragmentGroup;
     f: ^fragment;
     error: ^stream;
     astOverflow:< astInterfaceException;
     startingParsing:< (# do INNER #);
     fragmentNotExisting:< astInterfaceException
       (#  do true->continue; INNER #);
     grammarNotFound:< astInterfaceException;
     badFormat:< astInterfaceException;
     parseErrors:< astInterfaceException;
     fatalParseError:< astInterfaceException
       (# errNo: @integer enter errNo do INNER #);
     doubleFormDeclaration:< astInterfaceException;
     readAccessError:< astInterfaceException;
     writeAccessError:< astInterfaceException;
     writeAccessOnLstFileError:< astInterfaceException;
     EOSError:< astInterfaceException;
     noSuchFileError:< astInterfaceException;
     fileExistsError:< astInterfaceException;
     noSpaceLeftError:< astInterfaceException;
     otherFileError:< astInterfaceException;
     openCatcher: @handler (* Private *)
       (#  ... #);

  enter (fileName[],error[])
  do ...;
     g[]->f[];

  exit f[]
  #);
newGroup: (* make a new group with top as father *)
  (#
     fullname: ^Text;
     fg: ^FragmentGroup;
     alreadyOpen:< astInterfaceException;

  enter fullname[]
  ...
  exit fg[]
  #);
close: (* close FragmentGroup fg *)
  (# fg: ^fragmentGroup;
```

```
        enter fg[]
        do ...
        #);
      delete:
      (* delete FragmentGroup fg *)
        (# fg: ^fragmentGroup;
        enter fg[]
        do ...
        #);
      insert:
      (* insert a FragmentGroup into top table *)
        (# fg: ^fragmentGroup;
        enter fg[]
        do ...
        #);
      isOpen:
      (* return Group fullname if it is already open, otherwise NONE *)
        (# fullname: ^Text; fg: ^FragmentGroup;
        enter fullname[]
        ...
        exit fg[]
        #);
      topGroup: ^FragmentGroup;
      catcher: handler (* Private *)
        (#  ... #);

  #);
(* end of top *)
parseSymbolDescriptor:
  (#
     terminals: (* is dynamically expanded *) [1]
     ^text;
     nonterminals: (* is dynamically expanded *) [1] @integer;

  #);
errorReporter:
(* error-reporter pattern. Create a specialization of this pattern if
 * you want to do your own error-reporting
 *)
  (#
     frag: ^fragment;
     errorStream: ^stream;
     beforeFirstError:< object;
     afterLastError:< object;
     forEachError:<
       (#
          streamPos,startLineNo: @integer;
          errorLines:
          (* 1, 2 or 3 lines of text before the
           * error.  Approx. 100 chars
           *) @text;
          errorPos: (* the pos in errorLines of the error *) @integer;
          legalSymbols: ^parseSymbolDescriptor
       enter
       (streamPos,startLineNo,errorLines,errorPos (*inx*) ,legalSymbol
       do INNER
       #);

  #);
theErrorReporter:
(* the error reporter which will be called from the fragmentGroupparse
 * or from fragmentForm.parser.errorReport
 *) ^errorReporter;
```

```
isReferenced:
(* determine if the fragment is referenced anywhere among the open
 * fragments
 *)
  (# fx: ^fragment; found: @boolean
  enter fx[]
  ...
  exit found
  #);
treeLevel:
(* prefix for descriptions of grammars *)
  (#
     <<SLOT treeLevelLib:Attributes>>;
     grammarAst:
     (* if not NONE this point to the form of the ast describing the
      * grammar
      *) ^fragmentForm;
     symbolToName: (* gives a human-readable name for a symbol-number *)
       (# symbol: @integer; t: ^text;
       enter symbol
       do &text[]->t[]; ...
       exit t[]
       #);
     symbolToAst:
       (#
          symbol: @integer;
          as: ^ast;

       enter symbol
       ...
       exit as[]
       #);
     newAst: (* returns a new instance of ast *)
       (# prod: @integer; as: ^ast; frag: ^fragmentForm;
       enter (prod,frag[])
       do ...
       exit as[]
       #);
     newAstWithoutSons:
       (#
          prod: @integer;
          as: ^ast;
          frag: ^fragmentForm;

       enter (prod,frag[])
       ...
       exit as[]
       #);
     newLexemText: (* returns a new instance of lexemText *)
       (#
          length: @integer;
          prod: @integer;
          frag: ^fragmentForm;
          inx,base: @integer;

       enter
         (#
         enter (prod,length,frag[])
         ...
         #)
       exit
         (# as: ^ast
         ...
```

```
    exit as[]
    #)
  #);
newConst: (* returns a new instance of const *)
  (# c: ^const; frag: ^fragmentForm
  enter frag[]
  ...
  exit c[]
  #);
newUnexpanded:
(* returns a new instance of unExpanded *)
  (# s: ^unExpanded; syncatNo: @integer; frag: ^fragmentForm
  enter (syncatNo,frag[])
  ...
  exit s[]
  #);
newOptional:
(* returns a new instance of optional *)
  (# s: ^optional; syncatNo: @integer; frag: ^fragmentForm;
  enter (syncatNo,frag[])
  ...
  exit s[]
  #);
newSlot:
(* returns a new instance of slotDesc *)
  (# s: ^slotDesc; frag: ^fragmentForm
  enter frag[]
  ...
  exit s[]
  #);
version:< (* returns the grammar version *)
 integerObject (#  do undefinedVersion->value; INNER #);
grammarIdentification:< (* the grammar name *)
  (# theGrammarName: ^text
  do &text[]->theGrammarName[];
     undefinedGrammarName->theGrammarName;
     INNER
  exit theGrammarName[]
  #);
suffix:<
(* the file-name extension used for files containing programs
 * derived from this grammar.  Default extension is '.text'.
 *)
  (# theSuffix: ^text
  do &text[]->theSuffix[]; '.text'->theSuffix; INNER
  exit theSuffix[]
  #);
init:<
  (#
  do THIS(treeLevel)[]->grammartable.insert;
     false->parser.haveBeenInitialized;
     (for i: genRefArray.range repeat
       &referenceGenerator[]->genRefArray[i][]
     for);
     INNER
  #);
parser: @parse;
parse:
  (#
     errorReport:
     (* produce an errorReport on stream if the last parse did nc
      * succeed
      *)
```

```
  (# input,error: ^stream;
  enter (input[],error[])
  do ...
  #);
findSymbolNo:
(* given a text-string, find the nonterminal-symbol, that has
 * that name
 *)
  (# symbol: ^text; no: @integer
  enter symbol[]
  ...
  exit no
  #);
input,error: ^stream;
goalSymbol:
  @nonterminalSymbol;
frag: ^fragmentForm;
ok,haveBeenInitialized: @boolean;
parseEndPos: @integer;
lastCh: @char;
privatePart: @...;
initialize:
  (#
     fileName: ^text;
     isEos:<
     (* '--' may be considered as end-of-stream *) TrueObject;
     longLexems:<
     (* the lexems may be long (multi-word lexems) *)
      FalseObject;
     dashNames:< (* dash '-' may be allowed in indentifiers *)
      FalseObject;
     caseSensitive:< (* allows keywords to be case sensitive *)
      FalseObject;
     EOLasComEnd:< (* EOL is also accepted as end-of-comment *)
      FalseObject
  enter fileName[]
  ...
  #);
doParse:
  (#
     astOverflow:<
      astInterfaceException;
     startingParsing:< (# do INNER #);
     fragmentNotExisting:< astInterfaceException
       (#  do true->continue; INNER #);
     grammarNotFound:< astInterfaceException;
     badFormat:< astInterfaceException;
     parseErrors:< astInterfaceException;
     fatalParseError:< astInterfaceException
       (# errNo: @integer enter errNo do INNER #);
     doubleFormDeclaration:< astInterfaceException;
     readAccessError:< astInterfaceException;
     writeAccessError:< astInterfaceException;
     writeAccessOnLstFileError:< astInterfaceException;
     EOSError:< astInterfaceException;
     noSuchFileError:< astInterfaceException;
     fileExistsError:< astInterfaceException;
     noSpaceLeftError:< astInterfaceException;
     otherFileError:< astInterfaceException;
     catcher: handler (* Private *)
       (#  ... #);
     parseCatcher: @handler
     (* Private *)
```

```
                    (#  ... #);

                  enter
                  (goalSymbol,input[],
                   error[],frag[])
                  do ...
                  exit ok
                  #);
                commentId:
                (* declared to be able to get the value of comment inside th
                 * comment-binding in the parser
                 *) (#  exit comment #);

            enter (goalSymbol,input[],error[],frag[])
            do doParse;
            exit ok
            #);
          <<SLOT treeLevelPrivate:Attributes>>;
          kindArray: (* Private *)
            [maxProductions] @integer;
          nodeClassArray: (* Private *) [maxProductions] @integer;
          sonArray: (* Private *) [maxProductions] @integer;
          roomArray: (* Private *) [maxProductions] @integer;
          genRefArray: (* Private *) [maxProductions] ^referenceGenerator;
          prettyPrinter: (* Private *) ^object;
          maxProductions:< integerObject (* Private *)
            (#  do 400->value; INNER #);

        #);
      kinds: @
        (#
          interior: (#  exit 1 #);
          unExpanded: (#  exit 2 #);
          optional: (#  exit 3 #);
          nameAppl: (#  exit 4 #);
          nameDecl: (#  exit 5 #);
          string: (#  exit 6 #);
          const: (#  exit 7 #);
          comment: (#  exit 8 #);
          slotDesc: (#  exit 9 #);
          list: (* this will only be returned by 'nodeClass' *)
            (#  exit 117 #);
          cons: (* this will only be returned by 'nodeClass' *)
            (#  exit 118 #);
          dummy: (* temporary declaration. Is never returned *)
            (#  exit - 317 #)
        #);
      prodNo: @
        (#
          unExpanded: (#  exit - 1 #);
          optional: (#  exit - 2 #);
          nameAppl: (#  exit - 3 #);
          nameDecl: (#  exit - 4 #);
          const: (#  exit - 5 #);
          string: (#  exit - 6 #);
          comment: (#  exit - 7 #);
          slotDesc: (#  exit - 8 #)
        #);
      CommentSeparator1: (#  exit 1 #);
      (* Separation of comments *)
      CommentSeparator2: (#  exit 2 #);
      (* Separation of comments in same son *)
      CommentSeparator3: (#  exit 3 #);
```

```
(* Separation of comments in properties *)
CommentSieve: [256] @Char;
printComment:
  (# comment: ^Text; output: ^Stream;
  enter (comment[],output[])
  ...
  #);
undefinedGrammarName:
(* describes unknown grammars *) (#  exit '????' #);
undefinedVersion: (* describes unknown versions of grammars *)
  (#  exit - 1 #);
grammarTable: @
  (#
     BETA,propertyGrammar,meta,pretty:
     (* some different grammars, which might by instantiated by the
      * application
      *) ^treeLevel;
     noOfKnownGrammars: @integer;
     scan:
       (# current: ^treeLevel
       do (for i: noOfKnownGrammars repeat
            t[i][]->current[]; INNER
          for)
       #);
     find:
       (#
          grammarName: ^text;
          inx: @integer;
          thename: @text;
          ifNotFound:< astInterfaceException
          (* exception called if grammar not found *)
            (#
            do INNER ;
               'Grammar "'->msg.puttext;
               grammarName[]->msg.puttext;
               '" not loaded'->msg.putline;
               'Going to stop'->msg.putline;

            #);
          error: ^stream
       enter (grammarName[],error[])
       do ...
       exit t[inx][]
       #);
     t: [10] (* Private *)
     ^treeLevel;
     insert: (* Private *)
       (# theGrammar: ^treeLevel
       enter theGrammar[]
       do ...
       #);
     insertMetagrammar:
     (* Private: an instance of metaGrammar must be inserted into
      * grammarTable before any usages of grammarTable
      *)
       (#
       enter meta[]
       ...
       #)
  #);
grammarFinder:
(* create subpatterns of this pattern to implement your strategy for
 * looking-up grammars.  The fragment: findGrammar.bet contains such a
```

```
 * subpattern, implementing the standard look-up method used in the
 * Mjolner BETA System
 *)
  (#
     grammar: ^text;
     installed: @boolean;
     registerGrammars:< (* invoked to register the grammars *)
       (# error: ^stream
       enter error[]
       ...
       #);
     registeredGrammars:<
     (* may return a fragmentGroup containing the registered grammars
       (# grammarsGroup: ^fragmentGroup
       do INNER
       exit grammarsGroup[]
       #);
     error: ^stream
  enter
  (grammar[],error[])
  (* here the look-up for a grammar should takes place.  Either by
   * looking somehow amoung the previously registered grammars, or by
   * using some dynamic grammar look-up method
   *)
  ...
  exit installed
  (* true if new grammar installed in grammarTable *)
  #);
defaultGrammarFinder:<
(* default grammarFinder installed by astLevelInit *) grammarFinder;
grammarMissing:
(* called when a grammar is missing.
 * grammarMissing.registerGrammars is invoked in astLevelInit
 *) ^grammarFinder;
thePathHandler: @fileNameConverter;
stripPathName:
(* Strips last filename from a path specification in order to
 * conform with the new pathHandler.
 *)
  (# PN,newPN: ^text; ix: @integer;
  enter PN[]
  do directoryChar->PN.findAll(#  do inx->ix #);
     (if ix=0 then
         none ->newPN[]
      else
     (* terminating directoryChar is not removed due to 'strange'
      * behavior in localPath
      *)
         (1,ix)->PN.sub->newPN[]
     if)
  exit newPN[]
  #);
expandToFullPath:
  (# name: ^text;
  enter name[]
  exit
  (name[],currentDirectory)->thePathHandler.convertFilePath
  #);
offendingFormName:
(* set in case of a doubleDeclaration in fragmentForm names *) ^text;
trace: @
(* different tracing possibilities. I.e. to trace open of
 * fragments use
```

```
 *      (trace.fragmentOpen,true) -> trace.set;
 * To activate tracing through the BETA compiler,
 * set compileroption=number given here+400
 * (e.g. "beta -s 490 ..." to activate trace of slot bindings).
 * The trace will be delivered on the stream trace.str.  This may be
 * set by e.g.:
 *      traceFile[] -> trace.output;
 * By default, trace is delivered on screen.
 *)
  (#
     fragmentOpen: (#  exit 1 #);
     onParse: (#  exit 2 #);
     topOpen: (#  exit 3 #);
     fragmentClose: (#  exit 4 #);
     topClose: (#  exit 4 #);
     compactOpen: (#  exit 10 #);
     grammars: (#  exit 20 #);
     parsingComments: (#  exit 30 #);
     getnextComment: (#  exit 31 #);
     editingComments: (#  exit 32 #);
     parser: (#  exit 50 #);
     getBinding: (#  exit 90 #);
     getBindingMark: (#  exit 91 #);
     set: (* call this to trace something in the astInterface *)
       (# no: @integer; on: @boolean;  enter (no,on) do on->d[no] #);
     output: (#  enter str[] #);
     str: (* Private *) ^stream;
     d: (* Private *) [100] @boolean;

  #);
options: @
(* different options available.  I.e. to set these options use
 *       true -> options.forceParse
 * and to test whether these options are set, use
 *      (if options.forceParse ... if)
 *)
  (#
     forceParse: (#  enter option[1] exit option[1] #);
     option: (* Private *) [10] @boolean
  #);
astInterfaceNotification:
 notification
  (# m: ^text
  enter m[]
  ...
  #);
astInterfaceException: exception
  (# m: ^text
  enter m[]
  ...
  #);
astInterfaceError:< astInterfaceException;
astLevelInit:
  (#
  do ...
  #);
(********** PRIVATE PART ********************)
referenceGenerator: (* Private *)
  (# as: ^ast do INNER exit as[] #);
genUnExpanded: (* Private *) @referenceGenerator
  (#  do &unExpanded[]->as[] #);
genOptional: (* Private *) @referenceGenerator
  (#  do &optional[]->as[] #);
```

```
offset: @
(* Private: the following constants are private constants to ast, whic
 * tells where in array A relative from 'index' different information
 * can be found
 *)
  (#
     attribute: (#  exit 2 #);
     slotAttribute: (#  exit 3 #);
     commentType: (#  exit 2 #);
     usage: (#  exit 2 #);
     sizePerNode:
     (* tells how many entries in A is needed per node (not including
      * extra attributes)
      *) (#  exit 2 #);
     sizePerUnExpanded: (#  exit 2 #);
     sizePerNameAppl: (#  exit 2 #);
     sizePerNameDecl: (* must be equal to sizePerNameAppl *)
       (#  exit 2 #);
     sizePerString: (#  exit 2 #);
     sizePerConst: (#  exit 4 #);
     sizePerComment: (#  exit 4 #);
     sizePerSlotDesc: (#  exit 12 #);

  #);
groupBlackNumber: (* Private *)
(* magic number. To be used to recognize group-files *)
  (#  exit 131453937 #);
errorNumbers: @ (* Private *)
  (#
     noReadAccess: (#  exit 1 #);
     noWriteAccess: (#  exit 2 #);
     notExisting: (#  exit 3 #);
     badFormat: (#  exit 4 #);
     parseErrors: (#  exit 5 #);
     grammarNotFound: (#  exit 6 #);
     arrayTooBig: (#  exit 7 #);
     noSpaceLeft: (#  exit 8 #);
     writeAccessOnLstFileError: (#  exit 9 #);
     doubleFormDeclaration: (#  exit 10 #);
     EOSError: (#  exit 14 #);
     noSuchFile: (#  exit 15 #);
     fileExists: (#  exit 16 #);
     otherFileError: (#  exit 18 #);
     fatalParseError:
     (* The error numbers between 101 and 199 are exclusively allocate
      * for BOBS fatal parse error numbers.  The original BOBS error
      * number is this (no-100):
      *) (# no: @integer enter no exit (100 < no) and (no < 200) #);

  #);
notificationNumbers: @ (* Private *)
  (# startingParsing: (#  exit 201 #) #);
handler: (* Private *)
  (# no: @integer; msg: ^text enter (no,msg[]) do INNER #);
theCatcher: ^handler (* Private *) ;
maxdepth: (* Private: maximal elements in a stack *) (#  exit 50 #);
stak: (* Private *)
  (#
     stakOverflowException: astInterfaceException
       (#  do INNER ; 'error: stack overrun'->msg.putline #);
     a: [maxdepth] @integer;
     topindex: @integer;
     init: (#  do 0->topindex #);
```

```
          push:
            (# e: @integer
            enter e
            do (if topIndex=maxDepth then stakOverflowException if);
               e->a[topindex+1->topindex]
            #);
          pop:
            (# e: @integer;
            do a[topindex]->e; topindex-1->topIndex;
            exit e
            #);
          empty: (#  exit (topindex = 0) #);

      #);
     (* The following category defines some constants used as values for super
      * attributes in metagrammar-ast's
      *)
     super: @ (* Private *)
       (#
          undefined: (#  exit - 10 #);
          cons: (#  exit - 11 #);
          list: (#  exit 99999 #)
       #);
     tracer: (* Private *)
       (# traceNo: @integer; dmp: ^stream
       enter traceNo
       do (if trace.d[traceNo] then
              (if trace.str[]=none then
                  screen[]->dmp[]
               else
                  trace.str[]->dmp[]
              if);
              'Trace: '->dmp.puttext;
              (traceNo)->dmp.putInt;
              ' '->dmp.put;
              INNER ;
              dmp.newLine;

          if)
       #);
     silentTracer: (* Private *)
       (# traceNo: @integer; dmp: ^stream
       enter traceNo
       do (if trace.d[traceNo] then
              (if trace.str[]=none then screen[]->dmp[]
               else
                  trace.str[]->dmp[]
              if);
              INNER
          if)
       #);
     repS: (* Private *) ^repetitionStream;
     doRealOpen:
     (* Private: if this boolean is false, unpack of fragments will only
      * read in part of the fragment description.  Should only be used by the
      * BETA compiler
      *) @boolean;
     useModificationStatus: (* Private *) @boolean;

  do astLevelInit; INNER ;
  #);
containerList: list
(* Private: Empty specialization of the list pattern defined in the
```

```
 * containers library.  It is only defined to circumvent name-clash betweer
 * the list pattern defined in containers, and the list pattern defined her
 * in astInterface.
 *) (#  #);
```

ApplGram.bet

```
ORIGIN 'astlevel';
(*
 * COPYRIGHT
 *       Copyright (C) Mjolner Informatics, 1986-93
 *       All rights reserved.
 *)
INCLUDE 'metagrammarcfl';
INCLUDE 'metagramsematt'; (* OLM: 2.4.93*)
BODY 'private/applGramBody'
--- astInterfaceLib:attributes ---
applgram: treeLevel
  (# meta: ^metagrammar;
     pl: ^meta.productionList;
     grammarName,applGramSuffix: ^text;
     grammarIdentification::<
       (# do (if grammarName[]=NONE then NONE -> theGrammarName[]
           else grammarName -> theGrammarName
       if) #);
     suffix::< (# do applGramSuffix -> theSuffix; INNER #);
     (* impossible to define maxproductions to anything better before init
      * have been called
      *)
     version::<
       (# grammar: ^meta.Agrammar;
       do grammarAst.root[]->grammar[];
          ('version',undefinedversion)->grammar.getOptionValue->value;
          INNER
       #);
     init::<
       (# do &text[]->applGramSuffix[];
          '.text' -> applGramSuffix;
          INNER; (* to be able to define grammarAst *)
          ...
       #);
  #); (* of applgram *)
```

Findgrammar.bet

```
ORIGIN 'astlevel';
INCLUDE 'applgram';
BODY 'private/findGrammarBody'
(*
 * COPYRIGHT
 *       Copyright (C) Mjolner Informatics, 1986-93
 *       All rights reserved.
 *)
--- astInterfaceLib: attributes ---
```

```
findGrammar: grammarFinder
  (# notFound:< astInterfaceException
       (* invoked if the grammar cannot be found *)
       (# ... #);
     noParserAvailable:< astInterfaceNotification
       (* notification invoked if no parser is available for this grammar *)
       (# ... #);
     noRegisteredGrammars :< astInterfaceException
       (* invoked if no grammars have been registered.  If grammars are
        * registered during this exception, and control is returned to
        * findGrammar, the registered grammars will be used. *)
       (# ... #);
     accessError:< astInterfaceException
       (* invoked if any access error occurs during the registering of
        * grammars
        *);
     startParsing:<
       (* invoked if parsing is done during the registering of grammars *)
       (# do INNER #);
     registerGrammars::<
       (# accessError:< astInterfaceException
            (* invoked if any access error occurs during the registering of
             * grammars
             *);
          startParsing:<
            (* invoked if parsing is done during the registering of grammars *)
            (# do INNER #);
          registerGrammarsCatcher: (* Private *) @handler
          (# ... #);
       do ...; INNER
       #);
     registeredGrammars::< (# ... #);
     grammarsPATH:<
       (* the name of the file in which the valid grammars are specified.
        * Used by registerGrammars
        *)
       (# grammars: ^text
       ...
       exit grammars[]
       #);
     metaGrammarFile:<
       (* the name of the file in which the  meta-grammar is specified *)
       (# metaGrammar: ^text
       ...
       exit metaGrammar[]
       #);
     private: @ ...;
     newGrammar: (* Private *) ^applGram;
     grammarWithPath: (* Private *) ^text;
     grammarGroup, grammarDefsGroup: (* Private *) ^fragmentGroup;
     bobsFile: (* Private *) @file;
     findGrammarCatcher: (* Private *) @handler
     (# ... #)
  do ...;
  #);
```

Metagrammarcfl.bet

```
ORIGIN 'astlevel'
(*
 * COPYRIGHT
 *       Copyright (C) Mjolner Informatics, 1986-93
 *       All rights reserved.
 *)
--- astInterfaceLib: attributes ---
metagrammar: TreeLevel
  (# <<SLOT metagrammarAttributes: Attributes>>;
     Prod: cons(# <<SLOT prodAttributes: attributes>> #);
     ConsElem: cons(# <<SLOT consElemAttributes: attributes>> #);
     Lst: Prod(# #);
     optionSpecification: cons(# #);
     singleOption: optionSpecification(# #);
     AGrammar: cons
       (# getGrammarName: getson1(# #);
          putGrammarName: putson1(# #);
          getOptionOp: getson2(# #);
          putOptionOp: putson2(# #);
          getProductionList: getson3(# #);
          putProductionList: putson3(# #);
          getAttributeOp: getson4(# #);
          putAttributeOp: putson4(# #);
          <<SLOT AGrammarAttributes: attributes >>
       exit 1
       #);
     GrammarName: cons
       (# getNameDecl: getson1(# #);
          putNameDecl: putson1(# #);
       exit 2
       #);
     ProductionList: list(# sonCat::< prod; exit 3 #);
     LeftSide: cons
       (# getSynDeclName: getson1(# #);
          putSynDeclName: putson1(# #);
          <<SLOT leftsideAttributes: attributes>>
       exit 5
       #);
     Alternation: Prod
       (# getLeftSide: getson1(# #);
          putLeftSide: putson1(# #);
          getSynCatList: getson2(# #);
          putSynCatList: putson2(# #);
       exit 6
       #);
     SynCatList: list(# sonCat::< synCat exit 7 #);
     Constructor: Prod
       (# getLeftSide: getson1(# #);
          putLeftSide: putson1(# #);
          getConsElemList: getson2(# #);
          putConsElemList: putson2(# #);
       exit 8
       #);
     ConsElemList: list(# sonCat::< consElem exit 9 #);
     TaggedSyn: ConsElem
       (# getTagName: getson1(# #);
          putTagName: putson1(# #);
          getSynName: getson2(# #);
          putSynName: putson2(# #);
```

```
      <<SLOT taggedSyncatAttributes: attributes>>
  exit 11
  #);
SynCat: ConsElem
  (# getSynName: getson1(# #);
     putSynName: putson1(# #);
     <<SLOT syncatAttributes: attributes>>
  exit 12
  #);
ErrorSpec: ConsElem
  (#
  exit 13
  #);
ListOne: Lst
  (# getLeftSide: getson1(# #);
     putLeftSide: putson1(# #);
     getSynCat: getson2(# #);
     putSynCat: putson2(# #);
     getTermOp: getson3(# #);
     putTermOp: putson3(# #);
  exit 15
  #);
ListZero: Lst
  (# getLeftSide: getson1(# #);
     putLeftSide: putson1(# #);
     getSynCat: getson2(# #);
     putSynCat: putson2(# #);
     getTermOp: getson3(# #);
     putTermOp: putson3(# #);
  exit 16
  #);
Opt: Prod
  (# getLeftSide: getson1(# #);
     putLeftSide: putson1(# #);
     getSynCat: getson2(# #);
     putSynCat: putson2(# #);
  exit 18
  #);
Dummy: Prod
  (# getLeftSide: getson1(# #);
     putLeftSide: putson1(# #);
     getSynCat: getson2(# #);
     putSynCat: putson2(# #);
  exit 19
  #);
Term: ConsElem
  (# getString: getson1(# #);
     putString: putson1(# #);
     <<SLOT terminalAttributes: attributes>>
  exit 23
  #);
SynName: cons
  (# getNameAppl: getson1(# #);
     putNameAppl: putson1(# #);
     <<SLOT synNameAttributes: attributes>>
  exit 20
  #);
TagName: cons
  (# getNameDecl: getson1(# #);
     putNameDecl: putson1(# #);
  exit 21
  #);
SynDeclName: cons
```

```
  (# getNameDecl: getson1(# #);
     putNameDecl: putson1(# #);
  exit 22
  #);
OptionPart: cons
  (# getoptionList: getson1(# #);
     putoptionList: putson1(# #);
  exit 25
  #);
optionList: list(# sonCat::< optionElement exit 26 #);
optionElement: cons
  (# getoptionName: getson1(# #);
     putoptionName: putson1(# #);
     getoptionSpecification: getson2(# #);
     putoptionSpecification: putson2(# #);
  exit 27
  #);
optionSpecLst: optionSpecification
  (# getoptionSpecList: getson1(# #);
     putoptionSpecList: putson1(# #);
  exit 29
  #);
optionSpecList: list(# sonCat::< singleOption exit 30 #);
optionName: singleOption
  (# getNameAppl: getson1(# #);
     putNameAppl: putson1(# #);
  exit 32
  #);
optionConst: singleOption
  (# getConst: getson1(# #);
     putConst: putson1(# #);
  exit 33
  #);
optionString: singleOption
  (# getString: getson1(# #);
     putString: putson1(# #);
  exit 34
  #);
AttributePart: cons
  (# getattriblist: getson1(# #);
     putattriblist: putson1(# #);
  exit 36
  #);
AttribList: list(# sonCat::< Attrib exit 37 #);
Attrib: cons
  (# getSynCat: getson1(# #);
     putSynCat: putson1(# #);
     getNoOfAttributes: getson2(# #);
     putNoOfAttributes: putson2(# #);
  exit 38
  #);
NoOfAttributes: cons
  (# getconst: getson1(# #);
     putconst: putson1(# #);
  exit 39
  #);
errorProd: Prod
  (# exit 40 #);
optionError: singleOption
  (# exit 41 #);
grammarIdentification::< (# do 'metagrammar' -> theGrammarName #);
version::< (# do 4 -> value #);
suffix::< (# do '.gram' -> theSuffix #);
```

```
     maxProductions::< (# do 41 -> value #);
     init::<
       (# do … #);
  #)
```

Metagramsematt.bet

```
ORIGIN 'metagrammarcfl'
(*
 * COPYRIGHT
 *       Copyright (C) Mjolner Informatics, 1986-93
 *       All rights reserved.
 *)
--- prodAttributes: attributes ---
leftSide:
  (#
  enter putson1
  exit getson1
  #);
getSynDeclText:
  (# ls: ^this(metagrammar).leftSide;
     sd: ^SynDeclName;
     n: ^NameDecl;
  do getson1 -> ls[];
     ls.GetSynDeclName -> sd[];
     sd.getNameDecl -> n[];
  exit n.getText
  #);
superValue:
  (# ls: ^this(metagrammar).leftSide;
  do getson1 -> ls[];
  exit ls.superValue
  #);
superProd:
  (# prodNo: @integer;
     theProd: ^prod;
     predefined:< object;
     f: ^productionList
  do superValue -> prodNo;
     (if (prodNo>0) then
         father -> f[];
         prodNo -> f.get -> theProd[]
      else predefined
     if)
  exit theProd[]
  #)

--- leftsideAttributes: attributes ---
getSynDeclText:
  (# sd: ^SynDeclName;
     n: ^nameDecl;
  do GetSynDeclName -> sd[];
     sd.getNameDecl -> n[];
  exit n.getText
  #);
superValue:
  (# value: @integer
  enter (# enter value do (value,1) -> putAttribute #)
```

```
  exit 1 -> getAttribute
  #);
attributeSize:
  (# value: @integer
  enter (# enter value do (value,2) -> putAttribute #)
  exit 2 -> getAttribute
  #);

--- taggedSyncatAttributes: attributes ---
getSynText:
  (# sn: ^synName;
     n: ^nameappl;
  do getSynName -> sn[];
     sn.getNameAppl -> n[];
  exit n.getText
  #);
getTagText:
  (# tn: ^tagName;
     n: ^nameDecl;
  do getTagName -> tn[];
     tn.getNameDecl -> n[];
  exit n.getText
  #);

--- terminalAttributes: attributes ---
getText:
  (# s: ^string
  do getString -> s[];
  exit s.getText
  #)

--- syncatAttributes: attributes ---
getSynText:
  (# sn: ^synName;
     s: ^nameappl;
  do getSynName -> sn[];
     sn.getNameAppl -> s[];
  exit s.getText
  #);

--- synNameAttributes: attributes ---
dclRef:
  (# value: @integer
  enter (# enter value do (value,1) -> putAttribute #)
  exit 1 -> getAttribute
  #);
dclRefProd:
  (# prodNo: @integer;
     theProd: ^prod;
     predefined:< object;
     theGrammar: ^agrammar;
  do dclRef -> prodNo;
     (if (prodNo>0) then
         frag.root[] -> theGrammar[];
         prodNo -> theGrammar.getProd -> theProd[]
      else predefined
     if)
  exit theProd[]
  #);
getSynText:
  (# s: ^nameappl;
  do getNameAppl -> s[];
  exit s.getText
```

```
  #);

--- AGrammarAttributes: attributes ---
getProd:
  (# prodNo: @integer;
     theProd: ^prod;
     pl: ^productionList;
  enter prodNo
  do getProductionList -> pl[];
     prodNo -> pl.get -> theProd[]
  exit theProd[]
  #);
OptionSet:
  (# t: @text;
     optSpec: ^optionSpecification;
     opPart: ^optionPart;
     op: ^ast;
     optList: ^optionList;
  enter t
  do getOptionOp -> op[];
     (if (op.symbol=optionPart) then
         op[] -> opPart[];
         opPart.getOptionList -> optList[];
     if);
     (if (optList[]<>none) then
         scan: optList.scan
           (#
              optionEl: ^optionElement;
              optName: ^optionName;
              ap: ^nameAppl;
           do current[] -> optionEl[];
              optionEl.getOptionName -> optName[];
              optName.getNameAppl -> ap[];
              (if (ap.getText -> t.equalNCS) then
                  optionEl.getOptionSpecification -> optSpec[];
                  leave scan;
              if);
           #)
     if)
  exit OptSpec[]
  #);
checkOption:
  (# errortext: @text;
     spec: ^optionSpecification;
     checkedSymbol: @integer;
     as: ^ast;
  enter (spec[],errortext,checkedSymbol)
  do (if (spec.symbol=checkedSymbol) then
         spec.getson1 -> as[];
      else
         'ERROR in option-Specification for ' -> putText;
         errorText[] -> putText;
         screen.newLine;
         'Expected symbol: ' -> screen.putText;
         checkedSymbol -> screen.putInt;
         screen.newLine;
         'Found symbol: ' -> screen.putText;
         spec.symbol -> screen.putInt;
         screen.newLine;
     if)
  exit as[]
  #);
GetOptionValue:
```

```
  (# value: @integer;
     optionName: @text;
     c: ^const;
     optSpec: ^optionSpecification;
  enter (optionName,value)
  do (if ((optionName -> optionSet -> optSpec[])<>none) then
         (optSpec[],OptionName,optionConst) -> checkOption -> c[];
         c.getValue -> value;
     if);
  exit value
  #);
GetOptionName:
  (# value: @text;
     theOptionName: @text;
     c: ^lexemText;
     optSpec: ^optionSpecification;
  enter (theOptionName,value)
  do (if ((theOptionName -> optionSet -> optSpec[])<>none) then
         (optSpec[],theOptionName,optionName) -> checkOption -> c[];
         c.getText -> (# t: ^text enter t[] exit t #) -> value;
     if);
  exit value
  #);
GetOptionString:
  (# value: @text;
     optionName: @text;
     c: ^lexemText;
     optSpec: ^optionSpecification;
  enter (optionName,value)
  do (if ((optionName -> optionSet -> optSpec[])<>none) then
         (optSpec[],OptionName,optionString) -> checkOption -> c[];
         c.getText -> (# t: ^text enter t[] exit t #) -> value;
     if);
  exit value
  #);
```

Property.bet

```
ORIGIN '~beta/basiclib/v1.5/betaenv';
INCLUDE '~beta/containers/v1.5/list';
INCLUDE '~beta/basiclib/v1.5/repstream';
BODY 'private/propertyBody'
(*
 * COPYRIGHT
 *       Copyright (C) Mjolner Informatics, 1986-93
 *       All rights reserved.
 *)

--- LIB: attributes ---
constType: (# exit 1 #);
StringType: (# exit 2 #);
nameType: (# exit 3 #);
parValue:
  (# repSave:< (* Private *)
       (# f: ^repetitionStream enter f[] do INNER #)
  #);
constElement: parValue
  (# c: @integer;
```

```
      repsave::< (* Private *) (# do c -> f.putInt #)
  #);
StringElement: parValue
  (# s: @text;
      repsave::< (* Private *) (# do s[] -> f.putText #)
  #);
nameElement: parValue
  (# n: @text;
      repsave::< (* Private *) (# do n[] -> f.putText #)
  #);
propertyList:
  (# propElement:
       (# prop: @text;
          par: @parameterList;
          <<SLOT propertyPropertyListPropElementRepresentationPrivate:
attributes>>;
       #);
     propList: @list
       (# element::< propElement;
          <<SLOT propertyPropertyListPropListRepresentationPrivate:
attributes>>;
       #);
     init: (# do propList.init #);
     <<SLOT propertyPropertyListRepresentationPrivate: attributes>>;
     addProp:
       (# propName: ^text;
          newPropElement: ^propElement;
          ifPropExist:< (# delete: @boolean do true -> delete; INNER exit
delete #);
          newPar:
            (# parType:< parValue;
               val: ^parType;
               par: ^parElement;
            do &parElement[] -> par[];
               &parType[] -> par.val[] -> val[];
               INNER;
               par[] -> newPropElement.par.append
            #);
          addString: newPar
            (# parType::< stringElement;
               s: ^text;
            enter s[]
            do stringType -> par.type; s -> val.s;
            #);
          addName: newPar
            (# parType::< nameElement;
               n: ^text
            enter n[] do nameType -> par.type; n -> val.n;
            #);
          addConst: newPar
            (# parType::< constElement;
               c: @integer
            enter c do constType -> par.type; c -> val.c;
            #);
       enter propName[]
       ...
       #);
     findProp:
       (# name: ^text;
          l: ^proplist.element
       enter name[]
       ...
       exit l[]
```

```
      #);
    deleteProp:
      (# prop: ^text;
      enter prop[]
      do ...
      #);
    scanProp:
      (# currentParList: ^parameterList;
         doProp:<
           (# prop: ^text;
              getName:
                (# notAName:< (# do INNER #);
                   name: ^text;
                do ...
                exit name[]
                #);
              getConst:
                (# notAConst:< (# do INNER #);
                   const: @integer
                do ...
                exit const
                #);
              getString:
                (# notAString:< (# do INNER #);
                   string: ^text
                do ...
                exit string[]
                #);
              scanParameters:
                (# doConst:< (# c: @integer enter c do INNER #);
                   doString:< (# s: ^text enter s[] do INNER #);
                   doName:< (# n: ^text enter n[] do INNER #);
                ...
                #)

           enter prop[]
           do INNER
           #);
      ...
      #);
    GetProp: ScanProp
      (# doProp::< (# do (if (prop[]->P.equalNCS) then INNER if)#);
         P: ^text
      enter P[]
      #);
  #);
parameterList: list(# element::< parElement #);
parElement:
  (# val: ^parvalue;
     type: @integer;
     <<SLOT propertyParElementLocalsPrivate: attributes>>;
  #);
```

Propertycfl.bet

```
ORIGIN '~beta/mps/v5.1/astlevel'
--- astInterfaceLib: attributes---
```

```
property:  TreeLevel
(#
Property: cons(# #);

Machine: cons(# #);

Value: cons(# #);

Properties: cons
(#
  getPropertyList: getson1(##);
  putPropertyList: putson1(##);
exit 1 #);

PropertyList: list
(#
  soncat::< Property;
 exit 2 #);

ORIGIN: Property
(#
  getTextConst: getson1(##);
  putTextConst: putson1(##);
exit 5 #);

INCLUDE: Property
(#
  getStringList: getson1(##);
  putStringList: putson1(##);
exit 6 #);

BODY: Property
(#
  getStringList: getson1(##);
  putStringList: putson1(##);
exit 7 #);

MDBODY: Property
(#
  getMachineSpecificationList: getson1(##);
  putMachineSpecificationList: putson1(##);
exit 8 #);

OBJFILE: Property
(#
  getMachineSpecificationList: getson1(##);
  putMachineSpecificationList: putson1(##);
exit 9 #);

LIBFILE: Property
(#
  getMachineSpecificationList: getson1(##);
  putMachineSpecificationList: putson1(##);
exit 10 #);

LINKOPT: Property
(#
  getMachineSpecificationList: getson1(##);
  putMachineSpecificationList: putson1(##);
exit 11 #);

BETARUN: Property
(#
```

```
  getMachineSpecificationList: getson1(##);
  putMachineSpecificationList: putson1(##);
exit 12 #);

MAKE: Property
(#
  getMachineSpecificationList: getson1(##);
  putMachineSpecificationList: putson1(##);
exit 13 #);

BUILD: Property
(#
  getMachineSpecificationList: getson1(##);
  putMachineSpecificationList: putson1(##);
exit 14 #);

RESOURCE: Property
(#
  getMachineSpecificationList: getson1(##);
  putMachineSpecificationList: putson1(##);
exit 15 #);

ON: Property
(#
  getIntegerList: getson1(##);
  putIntegerList: putson1(##);
exit 16 #);

OFF: Property
(#
  getIntegerList: getson1(##);
  putIntegerList: putson1(##);
exit 17 #);

StringList: list
(#
  soncat::< TextConst;
 exit 18 #);

IntegerList: list
(#
  soncat::< IntegerConst;
 exit 19 #);

MachineSpecificationList: list
(#
  soncat::< MachineSpecification;
 exit 20 #);

MachineSpecification: cons
(#
  getMachine: getson1(##);
  putMachine: putson1(##);
  getStringList: getson2(##);
  putStringList: putson2(##);
exit 21 #);

Default: Machine
(#
exit 23 #);

Other: Property
(#
```

```
  getNameDcl: getson1(##);
  putNameDcl: putson1(##);
  getPropertyValueList: getson2(##);
  putPropertyValueList: putson2(##);
exit 24 #);

PropertyValueList: list
(#
  soncat::< PropertyValue;
 exit 25 #);

PropertyValue: cons
(#
  getValue: getson1(##);
  putValue: putson1(##);
exit 26 #);

NameDcl: Value
(#
  getNameDecl: getson1(##);
  putNameDecl: putson1(##);
exit 28 #);

NameApl: Machine
(#
  getNameAppl: getson1(##);
  putNameAppl: putson1(##);
exit 29 #);

TextConst: Value
(#
  getString: getson1(##);
  putString: putson1(##);
exit 30 #);

IntegerConst: Value
(#
  getConst: getson1(##);
  putConst: putson1(##);
exit 31 #);

  grammarIdentification::< (# do 'property' -> theGrammarName #);

  version::< (# do 4 -> value #);

  suffix::< (# do '.prop' -> theSuffix #);

  maxproductions::< (# do 31 -> value #);

  init::< (# do … #);
```

Appendix 1: The Metagrammar

To illustrate the grammar definition language of the metaprogramming system, we have included the grammar for the grammar definition language itself. This metagrammar is available to the user of the metaprogramming system, making it possible to use hyper structure editing on language grammars, and making it possible to construct other tools that can manipulate grammars.

File: `metagram.gram`:

```
--- metagrammar : Agrammar : metagrammar ---
Grammar metagrammar :
Option
  version    = 4
  suffix= '.gram'

  BobsOption = '32,34'
  comBegin   = '(*'
  comEnd     = '*)'
  stringChar = ''' '
Rule

  <AGrammar>  ::= 'Grammar' <GrammarName> ':' <OptionOp>
                  'Rule' <ProductionList> <AttributeOp>;
  <GrammarName> ::= <NameDecl>;
  <ProductionList>::+ <Prod> ';';

  <Prod>        ::|<Alternation>|<Constructor>|<Lst>
                  |<Opt>|<Dummy>|<ErrorProd>;

  <LeftSide>    ::= '<' <SynDeclName> '>';

  <Alternation>    ::= <LeftSide> '::|' <SynCatList>;
  <SynCatList>     ::+ <SynCat> '|';

  <Constructor>    ::= <LeftSide> '::=' <ConsElemList>;
  <ConsElemList>::+ <ConsElem>;
  <ConsElem>  ::| <TaggedSyn> | <SynCat> | <Term> | <ErrorSpec>;
  <TaggedSyn>   ::= '<' <TagName> ':' <SynName> '>';
  <SynCat>      ::= '<' <SynName> '>';
  <ErrorSpec>   ::= 'error';

  <Lst>       ::| <ListOne> | <ListZero>;
  <ListOne>   ::= <LeftSide> '::+' <SynCat> <TermOp>;
  <ListZero>  ::= <LeftSide> '::*' <SynCat> <TermOp>;
  <TermOp>      ::? <Term>;

  <Opt>         ::= <LeftSide> '::?' <SynCat>;

  <Dummy>       ::= <LeftSide> '::'  <SynCat>;
```

```
  <SynName>     ::= <NameAppl>;
  <TagName>     ::= <NameDecl>;
  <SynDeclName> ::= <NameDecl>;
  <Term>        ::= <String>;

  <OptionOp>    ::? <OptionPart>;
  <OptionPart>  ::= 'option' <optionList>;
  <optionList>  ::+ <optionElement>;
  <optionElement> ::= <optionName> '=' <optionSpecification>;
  <optionSpecification> ::| <singleOption> | <optionSpecLst>;
  <optionSpecLst>   ::= '(' <optionSpecList> ')';
  <optionSpecList>  ::+ <singleOption>;

  <singleOption>::| <optionName> | <optionConst>
                  | <optionString> | <optionError>;
  <optionName>  ::= <NameAppl>;
  <optionConst> ::= <Const>;
  <optionString>::= <String>;

  <AttributeOp>    ::? <AttributePart>;
  <AttributePart>::= 'attribute' <attriblist>;
  <AttribList>  ::* <Attrib>;

  <Attrib>    ::= <SynCat> ':' <NoOfAttributes>;
  <NoOfAttributes> ::=  <const>;
  <errorProd> ::= Error;
  <optionError> ::= Error

Attribute

  <LeftSide> : 2
  <SynName>  : 1
```

Appendix 2: The Pretty-print Specification Grammar

File: prettyprint.gram:

```
-- prettyprint : Agrammar : metagrammar --
Grammar prettyprint:

option
   suffix='.pgram'
   bobsoptions = '25, 32, 34'
   combegin    = '(*'
   comEnd      = '*)'
   stringChar  = ''''

rule

   <PrettyPrint>   ::= 'PrettyPrintScheme' <SchemeName:nameDecl>
                       'for' <GrammarName:nameDecl> ':' <ProductionList>;
   <ProductionList>::* <Production> ';' ;

   <Production>    ::| <Constructor> | <ListProd> ;
   <Constructor>   ::= <ProductionName:nameAppl> '=' <Stream:ItemList>;
   <ListProd>      ::= <ProductionName:nameAppl> '=' '(' <ListSpec> ')';

   <ItemList>      ::* <Item>;
   <Item>          ::| <Terminal> | <NonTerm> | <Break> | <Block>
               | <CommentPlace>;

   <Terminal>      ::| <DefaultTerm> | <AltTerm> ;
   <DefaultTerm>   ::= 'T' ':' <TerminalNo:const>;
   <AltTerm>       ::= <AlternativeTerminal:String> ;

   <NonTerm>       ::= 'N' ':' <NonTerminalNo:const>;

   <Break>         ::| <DefaultBreak> | <AltBreak> ;
   <DefaultBreak>  ::= '$$';
   <AltBreak>      ::= '$' <Space:const> ',' <Indention:const>;

   <Block>         ::= '[' <BlockType> <ItemList> ']';

   (* comments must only be specified after terminals! *)
   <CommentPlace>  ::= '*';

   <ListSpec>      ::= <Beginning:ItemList>
                       '{' <BlockType> <Separator:ItemList> '}'
                <Ending:ItemLIst> ;
```

```
   <BlockType>     ::| <Consistent> | <InConsistent> ;
   <Consistent>    ::= 'c';
   <InConsistent>  ::= 'i'

Attribute
   <Constructor> : 1
   <ListProd>    : 1
   <DefaultTerm> : 2
   <AltTerm>     : 2
   <NonTerm>     : 2
   <ListSpec>    : 2
```

Appendix 3: Expression Grammar Example

This appendix contains an example of use of the metaprogramming system for generating an expression calculator, enabling the user to enter expressions from the keyboard, which are then parser and the resulting AST is then evaluated by an interpreter (actually a recursive traversal of the AST, eveluating sunexpressions), and the calculated result is then printed on the screen. To give an impression of the application, the following is an example of an execution of the calculator (the underlined text is entered by the user — `expreval` is the name of the calculator):

```
expreval
Eval? 3+4
7
Eval? (3+4)*(20+2)/4
38
Eval? (3+4
PARSE-ERRORS
#   1 (3+4
# ****** ^
#  Expected symbols:  / mod ) * +
Eval? 22 mod 5
2
Eval? .
```

This application consists of five files:

- **`expr-meta.gram`**: Contains the grammar specification for the valid expressions.
- **`expr-pretty.pgram`**: Contains the pretty-printer specification.
- **`exprcfl.bet`**: Contains the generated context-free level interface.
- **`exprsematt.bet`**: Contains the additional semantic level interface. Primerily the `eval` routine.
- **`expreval.bet`**: Contains the initialization, and keyboard and screen handling code.

We will in the following present the five files along with a few comments on the important aspects of the particular file.

The Expression Grammar

This file contains the grammar that are used to check the syntax of the expressions of the calculator. The grammar is a fairly ordinary expression grammar, except that assignment statements are part of the legal syntax of the calculator, making the use of variables valid in the calculator. Please note the declaration of the substanceSlot and the attribute part.

File: `expr-meta.gram`:

```
--- expr: AGrammar: metagrammar ---
Grammar expr:
option
    string = unused
    substanceSlot = dcAtt
Rule
  <stat> ::| <assignment> | <evalStatement> | <quit>;
  <assignment> ::= <name : nameDecl> '=' <expression>;
  <evalStatement> ::= <expression>;
  <quit> ::= '.';
  <expression> ::| <Term> | <addExpression>;
  <term> ::| <factor> | <multExpression>;
  <factor> ::| <number> | <bracketExpression> | <variable>;
  <bracketExpression> ::= '(' <expression> ')';
  <MultExpression> ::= <Operand1 : Term> <MultOperator> <Operand2 : Factor>
  <AddExpression> ::= <Operand1 : Expression> <AddOperator> <Operand2 : Ter
  <MultOperator> ::| <TimesOp> | <DivOp> | <ModOp>;
  <AddOperator> ::| <PlusOp> | <MinusOp>;
  <Number> ::= <Const>;
  <Variable> ::= <NameAppl>;
  <TimesOp> ::= '*';
  <DivOp> ::= '/';
  <ModOp> ::= 'mod';
  <plusOp> ::= '+';
  <minusOp> ::= '-'
attribute
  (* the following definitions will trigger the generator to make
   *  semantic attribute slots for the generated context free level
   *)
  <expression> : 0
```

The Expression Pretty-Print Grammar

This file contains the pretty-pring grammar, used by the calculator. Strictly speaking this pretty-print grammar is not used by the calculator.

File: `expr-pretty.pgram`:

```
--- expr : prettyprint : prettyprint ---
PrettyPrintScheme exprSpec
for expr:
stat =  N:1 ;
assignment    = [c  N:1 $1,0 T:1 $1,0 * $1,2 N:2];
evalStatement =  N:1;
quit =  T:1 $1,0 *;
expression    =  N:1 ;
term =  N:1 ;
factor    =  N:1 ;
bracketExpression  = [c  T:1 $1,0 * $1,2 N:1 $1,0 T:2];
MultExpression     = [c  N:1 $1,2 N:2 $1,2 N:3];
AddExpression = [c  N:1 $1,2 N:2 $1,2 N:3];
MultOperator  =  N:1 ;
AddOperator   =  N:1 ;
Number    =  N:1;
Variable  =  N:1;
TimesOp   =  T:1 $1,0 *;
DivOp     =  T:1 $1,0 *;
ModOp     =  T:1 $1,0 *;
plusOp    =  T:1 $1,0 *;
minusOp   =  T:1 $1,0 *
```

The Expression Context-Free Level Interface

This file contains the generated context-free level interface. Please note the effects of the substanceSlot and attribute part specifications in the grammar. The `init` routine only contains initializations that can be ignored.

File: `exprcfl.bet:`

```
ORIGIN '~beta/mps/v5.1/astlevel'
--- astInterfaceLib: attributes---
expr: TreeLevel
  (# stat: cons
       (# <<SLOT statAttributes: attributes>> #);
     expression: cons
       (# <<SLOT expressionAttributes: attributes>> #);
     term: expression
       (# #);
     factor: term
       (# #);
     MultOperator: cons
       (# #);
     AddOperator: cons
       (# #);
     assignment: stat
       (# getname: getson1(# #);
          putname: putson1(# #);
          getexpression: getson2(# #);
          putexpression: putson2(# #);
       exit 2
       #);
     evalStatement: stat
       (# getexpression: getson1(# #);
          putexpression: putson1(# #);
       exit 3
       #);
     quit: stat
       (# exit 4 #);
     bracketExpression: factor
       (# getexpression: getson1(# #);
          putexpression: putson1(# #);
       exit 8
       #);
     MultExpression: term
       (# getOperand1: getson1(# #);
          putOperand1: putson1(# #);
          getMultOperator: getson2(# #);
          putMultOperator: putson2(# #);
          getOperand2: getson3(# #);
          putOperand2: putson3(# #);
       exit 9
       #);
     AddExpression: expression
       (# getOperand1: getson1(# #);
          putOperand1: putson1(# #);
          getAddOperator: getson2(# #);
          putAddOperator: putson2(# #);
          getOperand2: getson3(# #);
          putOperand2: putson3(# #);
       exit 10
       #);
     Number: factor
       (# getConst: getson1(# #);
          putConst: putson1(# #);
```

```
      exit 13
      #);
    Variable: factor
      (# getNameAppl: getson1(# #);
         putNameAppl: putson1(# #);
      exit 14
      #);
    TimesOp: MultOperator
      (# exit 15 #);
    DivOp: MultOperator
      (# exit 16 #);
    ModOp: MultOperator
      (# exit 17 #);
    plusOp: AddOperator
      (# exit 18 #);
    minusOp: AddOperator
      (# exit 19 #);
    grammarIdentification::<
      (# do 'expr'->theGrammarName #);
    version::<
      (# do -1->value #);
    suffix::<
      (# do '.text'->theSuffix #);
    maxproductions::<
      (# do 19->value #);
    dcAtt: @
      <<SLOT dcAtt: descriptor>>;
    init::<
      (# ... #);
  #)
```

The Expression Semantic Level Interface

This file contains the semantic level interface, written for the calculator. Please note the utilization of the SLOTs, generated as the result of the substanceSlot and attribute part of the grammar.

File: `exprsematt.bet:`

```
ORIGIN 'exprcfl';
INCLUDE '~beta/containers/v1.5/hashTable'
--- expressionAttributes: attributes ---
eval:
  (# value: @integer ;
     n: ^number;
     cnst: ^const;
     m: ^multExpression;
     a: ^addExpression;
     anAst: ^ast;
     be: ^bracketExpression;
     e1,e2: ^expression;
     var: ^variable;
     na: ^nameAppl;
  do (if symbol
      //bracketExpression then
         this(expression)[] -> be[]; be.getExpression -> e1[]; e1.eval -> v
      //multexpression then
         this(expression)[] -> m[];
         m.getOperand1 -> e1[]; m.getoperand2 -> e2[];
         m.getMultOperator -> anAst[];
         (if anAst.symbol
          //timesop then e1.eval * e2.eval -> value
          //divop then e1.eval div e2.eval -> value
```

```
          //modOp then e1.eval mod e2.eval -> value
         if);
       //addexpression then
         this(expression)[] -> a[];
         a.getOperand1 -> e1[]; a.getOperand2 -> e2[];
         a.getAddOperator -> anAst[];
         (if anAst.symbol
          //plusOp then e1.eval + e2.eval -> value
          //minusOp then e1.eval - e2.eval -> value
         if)
       //number then this(expression)[]->n[];
                    n.getConst->cnst[]; cnst.getValue -> value
       //variable then this(expression)[] -> var[];
         var.getnameAppl -> na[];
         (# e: ^dcAtt.symbolTable.element
         do na[] -> dcAtt.symbolTable.findKey -> e[];
            (if e[]//none then
                na.getText -> screen.putText;
                ' is not declared ' -> screen.putLine;
             else e.e.eval -> value
            if);
         #);
      if)
  exit value
  #)

--- dcAtt: descriptor ---
(# symbolTable: @hashTable
     (# element::< (# id: ^lexemText; e: ^expression #);
        hashFunction::<
          (# t: ^text
          do e.id.getText -> t[];
             t.scan(# do (ch->ascii.lowCase)+133*value -> value #);
          #);
        equal::<
          (# equalText:
               (# t1,t2: ^text enter (t1[],t2[]) exit t1[] -> t2.equalNCS #)
          do (left.id.getText,right.id.getText) -> equalText -> value
          #);
        findKey:
          (# e: @element; found: ^element
          enter e.id[]
          do scan(# where::< (# do (e[],current[]) -> equal -> value #)
                  do current[] -> found[] #)
          exit found[]
          #);
     #);
   init: (# do symbolTable.init #);
#)

------ statAttributes: attributes ------
run:
  (# expr: ^expression;
     eval: ^evalStatement;
     let: ^assignment;
     elm: ^dcAtt.symbolTable.element
  do (if symbol
      //assignment then
         this(stat)[] -> let[];
         &dcAtt.symbolTable.element[] -> elm[];
         let.getName -> elm.id[];
         let.getExpression -> elm.e[];
         elm[] -> dcAtt.symbolTable.insert;
```

```
      //evalStatement then
         this(stat)[] -> eval[];
         eval.getExpression -> expr[];
         expr.eval -> screen.putInt;
         screen.newLine
      //quit then (normal,'') -> stop
     if);
```

The Expression Evaluator Program

This file contains the initialization of the metaprogramming system and the handling of the keybroard and screen. Please note the use of the parser, the handling of parse errors, and the evaluation of the ASTs, resulting from successful parsing of the input.

File: expreval.bet:

```
ORIGIN 'exprcfl';
INCLUDE 'exprsematt'
--- program: descriptor ---
(# (* This is a small demo-program of how to use the MetaProgrammingSystem.
    * The program implements a small desc-calculator a la dc in unix.  The
    * grammar for expressions is on the file expr-meta.gram.  The generated
    * context free level is on exprCfl.  Exprsematt contains semantic
    * attributes for expr.
    *)
   ast: @astinterface;
   expr: @ast.expr; (* the cfl of the grammar *)
   exprFragment: ^ast.fragmentForm;
   evalString: ^text;
   stat: ^expr.stat;
   ok: @boolean;
   btabFile: ^text;
do ast.astLevelInit; (* initialize astlevel *)
   expr.init;        (* and the context free level of the generated grammar
   'expr-parser' -> btabFile[];
   ast.parserFileExtension->btabFile.puttext;
   btabFile[] -> expr.parser.initialize; (* and the parser *)
   expr[] -> ast.newFragmentForm -> exprFragment[]
   (* create a fragmentform which can contain the asts *);
   cycle
   (# do
      'Eval? ' -> screen.putText;
      keyBoard.getLine -> evalString[]; (* read a string from keyboard *)
      evalString.newLine; (* add a newline to the string *)
      0 -> evalString.setPos;  (* reset evalString to start *)
      (1,evalString[],screen[],exprFragment[]) -> expr.parser -> ok;
      (* 1: goalSymbol,
       * evalString: input,
       * exprFragment: the fragmentform to contains the asts
       *)
      (if ok
       //false then
          'PARSE-ERRORS' -> screen.putLine;
          0 -> evalString.setPos; (* reset evalString to start *)
          (evalString[],screen[]) -> expr.parser.ErrorReport;
       else  (* there was no parse-errors *)
          exprFragment.root[] -> stat[];
          (* the parser returns the root of the parsed ast in fragment.root
          stat.run;
      if);
   #)
```

Index

The entries in the index are the identifiers defined in the public interface of the libraries: The minor level entries refer to identifiers defined local to the identifier of the major level entry. For those index entries referring to patterns with super- or subpatterns within the library, these patterns are specified in special sections of the minor level index for that identifier.

N

R

S