A Reusable Object-Oriented Approach to Formal Specifications of Programming Languages

Marjan Mernik, Mitja Lenič, Enis Avdičaušević, Viljem Žumer

University of Maribor
Faculty of Electrical Engineering and Computer Science
Smetanova 17, 2000 Maribor, Slovenia
marjan.mernik@uni-mb.si

RESUME.

ABSTRACT. Formal methods for describing programming language semantics are an excellent tool for programming language design and development. However, they are not widely used since they are not modular, extensible and reusable. In the paper a new modular, extensible and reusable approach for specifying programming languages with attribute grammars is presented. The concepts of object-oriented programming, i.e. templates, multiple inheritance, and object-oriented implementation of semantic domains, are integrated with attribute grammars. With the proposed approach a language designer has the chance to design incrementally a language or reuse some fragments from other programming language specifications. The proposed approach has been implemented in our compiler generator tool LISA ver. 2.0.

MOTS-CLES :

KEY WORDS : language specifications, reusability, extensibility, attribute grammars, compiler generators

1 Introduction

Compiler construction is often mentioned as one of the few really systematically managed disciplines. There is a long tradition of producing compilers, underlying theories are well understood and there exist many application generators which automatically produce a compiler or an interpreter from the formal programming language specification. Formal methods for programming language description also serve as a tool for programming language development and design. The programming languages which have been designed with one of the various formal methods have a better syntax and semantics, less exceptions and are easier to learn. However, despite of the obvious advantages, none of the currently most widely used formal methods for programming language description, such as denotational semantics, operational semantics, axiomatic semantics, or attribute grammars, have gained more popularity and general use. Some reasons for this are: semantics is much more difficult to describe than the syntax, and semantics description is not easy to read. We are convinced that the real problem with these formal methods is the lack of modularity, extensibility and reusability. Since programming language design is an iterative process, designers need partial descriptions that can be easily reused, extended and modified. In the paper a new approach to modularity, extensibility and reusability of attribute grammars is presented. Attribute grammars have been introduced by D.E. Knuth [KNU 68] and since then are proven to be useful in specifying the semantics of programming languages, in automatic constructing of compilers/interpreters, in specifying and generating interactive programming environments and in many other areas. Attribute grammars are a generalization of context-free grammars in which each symbol has an associated set of attributes that carry semantic information, and with each production a set of semantic rules with attribute computation is associated. In attribute grammars semantics is given in a declarative rather than an operational notation. However, ordinary attribute notation has deficiencies which become apparent in specifications for real programming languages. Such specifications are large, unstructured and are hard to understand, modify and maintain. Therefore, in recent years, the research on attribute grammars has shifted from theoretical investigations to improving attribute grammars as a methodology, and to designing systems with major emphasis on pragmatic issues [PAA 95]. In this paper a new approach that overcomes the drawbacks of ordinary attribute grammars is presented. The concepts from object-oriented programming, i.e. template, multiple inheritance, and object-oriented implementation of semantic domains, are integrated with attribute grammars. With templates we are able to describe the semantic rules which are independent of grammar production rules. Specifications with templates are reusable, shorter and easier to read and maintain. With multiple inheritance we are able to organize specifications in such a way that specifications can be inherited and specialized from ancestor specifications. Multiple attribute grammar inheritance supports reusability and extensibility of attribute grammars. With object-oriented implementation of semantic domains we are able to reuse various semantic domains in implementing different programming languages by inheritance or aggregation.

The paper is an extension of the paper [MER 98]. Reusable syntax and semantic specifications are presented in sections 2 and 3. Further on, in section 4, our implementation and experience are briefly described. In section 5 related work is described. It is followed by the conclusion in section 7. A part of specifications for PLM language is presented in appendix. We assume that the reader is familiar with attribute grammars.

2 Reusable syntax specifications

In formal methods for programming language description (regular expressions, BNF and attribute grammars) we are looking for components which can be reused in the specification for another programming language. In formal specifications of programming language it is easy to notice that the main part of such a specification consists of the semantic specification and the much smaller part consists of BNF and regular expressions. Therefore, our attention in searching for reusable components was mainly focused on semantic specifications. Since semantics can not be defined without the syntax, also the reusable components in syntax are looked for. However, we found that regular definitions are already reusable components and can be used in other regular expressions. In the existing BNF notation such reusable components were not found. But we agree that in BNF there are patterns which can be reusable. Such reusable patterns are iteration and option which already exist in EBNF, and associativity and priority of operators, list of items and many others. A common rule of thumb to make components reusable is to make components generic. For such reusable pattern in BNF we propose a template which can be instantiated many times and which generates different production rules. In the following example a template for the left associative operator is presented.

```
template <nonterminal LEVEL1, LEVEL2,  terminal OP>
  rule LEFT_ASSOC_OP {
     LEVEL1  ::= LEVEL2 __LEVEL ;
     __LEVEL ::= OP LEVEL2 __LEVEL
             |   /* epsilon */
             ;
}
```

In the rule within the template two different groups of nonterminal grammar symbols appear. The formal parameter in the template interface which matches with the concrete grammar sequence of symbols at template instantiation belongs to the first group. Other nonterminal symbols are fresh symbols which have not yet occurred in grammars and are denoted, for better reading, with "__" before the name. The above template can be instantiated in the following manner:

```
LEFT_ASSOC_OP<E; T; ″+″>
LEFT_ASSOC_OP<T; F; ″*″>
```

and generates the following productions:

```
E  ::= T EE
EE ::= ″+″ T EE | ε
T  ::= F TT
TT ::= ″*″ F TT | ε
```

Meta operations in EBNF, option and iteration, are only a special case of the proposed approach. They can be also described with the template in the following manner:

```
template <nonterminal ALFA>
  rule OPTIONAL {
       __ALFA ::= ALFA
               |   /* epsilon */
               ;
}

template <nonterminal ALFA>
  rule ITERATION {
       __ALFA ::= ALFA __ALFA
              |   /* epsilon */
              ;
}
```

The following production in EBNF notation

```
PARAMS ::= [ID {”,” ID}]
```

can be also described with a template as

```
PARAMS ::= OPTIONAL<ID ITERATION<”,” ID>>
```

Templates for various syntactic patterns are very useful in parser generation since various details are abstracted. However, this kind of template becomes less suitable when the semantic part is also included. Since similar syntactic structures with a similar semantic meaning are rare. Exceptions are arithmetic expressions. Therefore another kind of templates is proposed where a template is an abstraction over attribute computation.

3 Reusable semantic specifications

Our main attention in searching for reusable components in formal language description was focused on semantic specifications, from which the greatest benefits were expected [MER 97]. In semantic specifications three possibilities of reuse have been found. The first one are semantic rules associated with production rules of context free grammars. Reusable patterns of semantic rules are handled in a similar manner as reusable patterns in BNF by templates. Next possibilities of reuse arise when multiple inheritance is integrated with attribute grammars. In this case the whole attribute grammar is subject to inheritance and specialisation. On the other hand we observe the usefulness of reusable semantic domains.

3.1 Templates

When studying semantic specifications for various programming languages common patterns can be noticed. Patterns like value distribution, list distribution, value construction, list construction, bucket brigade, propagate value and many others are independent of the structure of production rules. They are also independent of a number of attribute occurrences. Such patterns are described with templates which are a new approach in the attempt to make attribute grammars more reusable. A template in attribute grammars is a polymorphic abstraction of a semantic rule parameterized with attribute occurrences which can be associated with many production rules with different nonterminal and terminal symbols. Since a nonterminal symbol can be considered as a class in object-oriented attribute grammars [KOS 91], a template in attribute grammars is a kind of polymorphism. Further, at template instantiation appropriate semantic rules are generated at compiler generation time which is similar to templates in object-oriented languages where the code is generated at compile time. For example, left associative operators have the following common pattern in semantic rules:

```
E ::= E + T
 { E[0].value = E[1].value + T.value; }
T ::= T * F
 { T[0].value = T[1].value * F.value; }
```

The pattern above can be described with the template:

```
template <attributes level0_val, level1_val,
                     nextlevel_val, function op>
 compute left_assoc_op {
  level0_val=op(level1_val,nextlevel_val);
}
```

The semantic specification for both expressions can now be described in the following way:

```
E ::= E + T
 { left_assoc_op<E[0].value; E[1].value; T.value; plus> }
T ::= T * F
 { left_assoc_op<T[0].value; T[1].value; F.value; mult> }
```

The pattern above is introductory and describes the basic idea behind the templates. A more general template should also be independent of a number of attribute occurrences which participate in semantic rules. For this purpose a variable list of arguments is proposed in a manner similar to other programming languages like Lisp and C++. As an example, a value distribution pattern (fig. 1) is described as:

```
Y ::= X1 X2 ... XN
 { X1.in = Y.in; X2.in = Y.in;   ...
   XN.in = Y.in; }
```

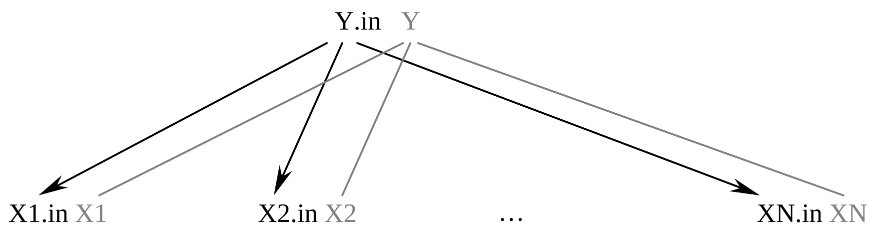

Figure. 1. **Value distribution pattern**

A template describing the value distribution pattern is:

```
template <attributes Y_in, X_in*>
 compute value_distribution {
    { X_in* = Y_in; }
}
```

The formal argument `X_in*` in the template `value_distribution` is a variable list of arguments. Such arguments are denoted with an asterisk after the name. At the template instantiation a part of semantic rules enclosed with braces is generated for each argument in the variable list. Therefore the semantic rule below

```
BLOCK ::= DEC STMTS
{value_distribution<BLOCK.static_level;
       DEC.static_level, STMTS.static_level>}
```

generates the following rules in the compiler generation time:

```
DEC.static_level = BLOCK.static_level;
STMTS.static_level = BLOCK.static_level;
```

Together with a variable list of arguments some functions are defined which can be used for variable list manipulation:

- `first()`, returns the first argument in the variable list
- `last()`, returns the last argument in the variable list
- `succ()`, returns the successor in the variable list
- `pred()`, returns the predecessor in the variable list

A successor for the last argument and a predecessor for the first argument do not exist. The usage of the above defined functions is presented in the next example. A propagate pattern (fig. 2) is described as:

```
Y ::= X1 X2 ... XN
 { X1.in = Y.in; X2.in = X1.out; ... XN.in = XN-1.out; }
```

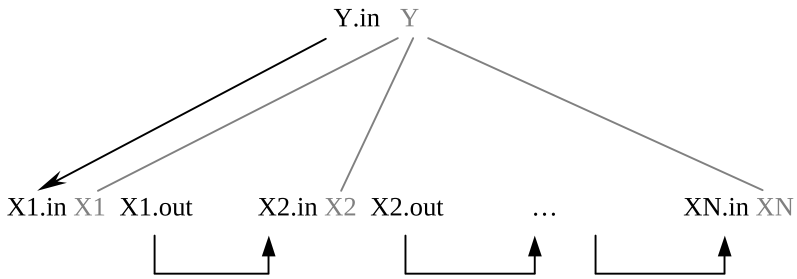

Figure 2. **Propagate pattern**

A template describing the propagate pattern is:

```
template <attributes Y_in, X_in*, X_out*>
  compute propagate {
    first(X_in*) = Y_in;
    { X_in* = pred(X_out*); }
}
```

Templates would become even more powerful if they were nested as presented in the next example. A pattern bucket brigade left (fig. 3) is described as:

```
Y ::= X1 X2 ... XN
 { X1.in = Y.in; X2.in = X1.out; ...
   XN.in = XN-1.out; Y.out = XN.out; }
```

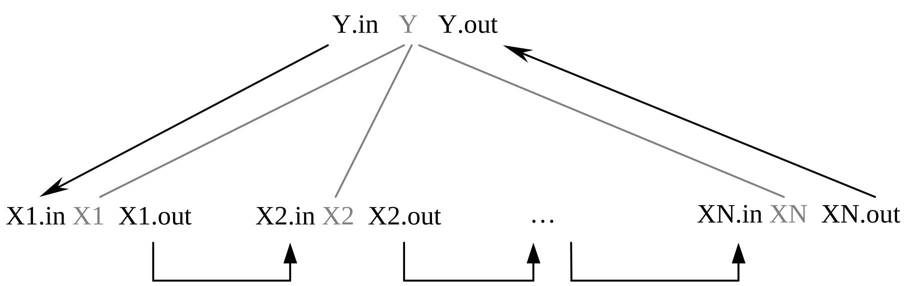

Figure 3. **Pattern bucket brigade left**

A template describing the pattern bucket brigade left with a nested template is:

```
template <attributes Y_in, Y_out, X_in*, X_out*>
 compute bucket_brigade_left {
    propagate<Y_in; X_in*; X_out*>
    Y_out = last(X_out*);
}
```

One of the drawbacks of attribute grammars pointed out by several researchers are the less readable semantic rules since essential computations are mixed with a lot of copy and propagation rules. To understand the work that has been accomplished by semantic rules can take a lot of time, as shown in the next example.

```
DECLS ::= DECL ";" DECLS
{ DECL.is_global=DECLS[0].is_global;
  DECL.inEnv=DECLS[0].inEnv;
  DECLS[1].is_global=DECLS[0].is_global;
  DECLS[0].outEnv=DECLS[1].outEnv;
  DECLS[1].inEnv=DECL.outEnv; }
```

Specifications with templates are clearly much more readable.

```
bucket_brigade_left<DECLS[0].inEnv; DECLS[0].outEnv;
                    DECL.inEnv, DECLS[1].inEnv;
                    DECL.outEnv, DECLS[1].outEnv>
value_distribution<DECLS[0].is_global;
                   DECL.is_global, DECLS[1].is_global>
```

The example shows that in complex semantic rules the decomposition of attribute computation is also possible (fig 4). In the semantic rule above, attribute computations are composed of bucket brigade and value distribution patterns.

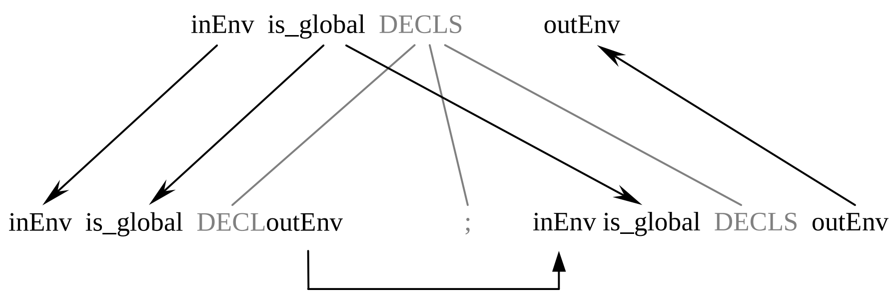

Figure 4. **Particular pattern for declarations**

Note that at a template instantiation arguments in the variable list are delimited by a comma ",", while others are delimited by a semicolon ";".

Templates which describe reusable syntactic patterns and those which describe reusable semantic patterns can also be merged as is presented in following example.

```
template <nonterminal LEVEL1, LEVEL2, terminal OP,
          attributes val, function f, type attr_type>

 attributes attr_type inval, outval;
 rule LEFT_ASSOC_OP {
  LEVEL1 ::= LEVEL2 __LEVEL compute {
     LEVEL1.val=__LEVEL.outval);
    __LEVEL.inval=LEVEL2.val; }
          ;
```

```
  __LEVEL ::= OP LEVEL2 __LEVEL  compute {
     __LEVEL[1].inval=f(LEVEL2.val, __LEVEL[0].inval);
     __LEVEL[0].outval= __LEVEL[1].outval; }
       ;
  __LEVEL ::= /* epsilon */    compute {
     __LEVEL.outval=__LEVEL.inval;}
           ;
 }
```

The template above, where fresh nonterminals and attributes of fresh nonterminals are completely abstracted out, can be instantiated as:

```
LEFT_ASSOC_OP<E; T; ″+″; val; plus; int>
LEFT_ASSOC_OP<T; F; ″*″; val; multiply; int>
```

We have already gained some experience in using templates. Their benefits are:

- specifications are more readable and maintainable since templates are at higher abstraction levels than assignment statements,
- specifications are reusable since the templates are independent of the structure of grammar productions, and
- language designers can create their own templates.

The implementation of templates is similar to the implementation in the programming language C++. Templates have macro semantics. Semantic rules in templates are not verified for correctness, only the result of their instantiation is.

3.2 Multiple Inheritance

The language design process should be supported by modularity and abstraction in a manner that allows incremental changes as easily as possible. This is one of the strategic directions of further research on programming languages [HAN 97]. Inheritance is a language mechanism that allows new definitions to be based on the existing ones. A new specification inherits the properties of its ancestors, and may introduce new properties that extend, modify or defeat its inherited properties. When a new concept is added/removed in/from a language, not only the semantic part is changed, but also the syntax rules and the lexicon are modified. Therefore, such incremental modifications usually do not preserve upward language compatibility. A language designer needs a formal method which enables incremental changes and usage of specification fragments from various programming languages. We accomplish these goals by introducing the object-oriented concepts, i.e. multiple inheritance and templates, into attribute grammars. Grammar inheritance is introduced in [AKS 91] where only single inheritance of production rules is possible. We extend this idea to multiple attribute grammar inheritance in a manner similar to object-oriented languages. Multiple inheritance offers considerably more possibilities for incremental modification. On the other hand, also conceptual and technical intricacies are introduced. Multiple attribute grammar inheritance is a structural organization of attribute grammars where the attribute grammar inherits the specifications from ancestor attribute grammars, may add new specifications, may override some specifications from ancestors or even defeat some ancestor specifications. With inheritance we can extend the lexical, syntax and semantic parts of programming language specification. Therefore, regular definitions, production rules, attributes, semantic rules and operations on semantic domains can be inherited, specialized or overriden from ancestor specifications. Language is specified in the following manner:

```
Language L1 extends L2, ..., LN {
  lexicon {
    [overrides | extends] regular definition
    ...
  }
  attributes type A1, ..., AM
              ...
  rule Z [extends Y | overrides Y] {
    X ::= X11 X12 ... X1p  compute {
      semantic functions }
      |
      ...
      |
          Xr1 Xr2 ... Xrt compute {
      semantic functions }
      ;
  }
  ...
  method [overrides | extends] M {
    operations on semantic domains
  }
  ...
}
```

In object-oriented languages the properties that consist of instance variables and methods are subject to modification. Since in attribute grammars semantic rules are tightly coupled with particular production rules, properties in multiple attribute grammar inheritance consist of lexical regular definitions, attribute definitions, generalized syntax rules which encapsulate semantic rules and operations on semantic domains. Our primary block suitable for modification is a generalized syntax rule. Properties should have names; no-name properties can not be specialised and modified. Therefore, the basic block for modification must be named. In our case we used the name rule which consisted of the production rule and the associated semantic rules. The production rule may have many alternatives which can be added or removed in modification process. We do not restrict the language designer to use a new name rule for each alternative. However, this is the preferred method to achieve maximum flexibility. This is also the reason why we could not simply use the name of the left hand nonterminal. In the case of many alternatives there is no way to determine which alternative is modified or even defeated.

Let us look at the self intuitive example of a simple calculator language. The language is gradually extended from the simplest form to a language with more features. In the first attempt of specifying the calculator language only adding and subtraction of digits are possible. Let us call it AddSubCalc.

```
Language AddSubCalc {
 lexicon {
   digit        [0-9]
   operator     ″+″ | ″-″
   whiteSpace   [\ \n\t]+
 }
 attributes int val;
 rule EXP {
  E ::= E ″+″ T  compute
        {left_assoc_op<E[0].val; E[1].val; T.val; plus>}
     |  E ″-″ T  compute
       {left_assoc_op<E[0].val; E[1].val; T.val; minus>}
     |  T        compute {E.val=T.val;}
     ;
  }
 rule TERM {
  T ::= digit compute
          {T.val=Value(digit.lex_val);}
    ;
  }
}
```

Next, the rules for multiplying are added. The language MulCalc is an extension of the language AddSubCalc where regular definitions `digit` and `whitespace`, attribute `val`, production `EXP` are inherited, the regular definition `operator` is extended, the regular definition `separator` and production `FACTOR` are added and the ancestor rule `TERM` is overridden.

```
Language MulCalc extends AddSubCalc {
 lexicon {
  extend AddSubCalc.operator    ″*″
                    separator   ″(″ | ″)″
 }
 rule TERM1 overrides AddSubCalc.TERM {
  T ::= T ″*″ F  compute
    {left_assoc_op<T[0].val; T[1].val; F.val; multiply>}
    ;
 }
 rule TERM2 {
  T ::= F  compute {T.val=F.val;}
     ;
 }
 rule FACTOR {
  F ::= ″(″ E ″)″  compute {F.val=E.val;} ;
  F ::= digit      compute {F.val=Value(digit.lex_val);} ;
 }
}
```

Let us suppose that the language Dec for declarations already exists.

```
Language Dec {
 lexicon {
   ident        [a-z][a-z0-9]*
   int          [0-9]+
   operator     "="
 }
 attributes Table consts;
 rule DECL {
  Decl ::= ident = int Decl compute {
    Decl[0].consts = Insert(Decl[1].consts,
                    ident.lex_val, Value(int.lex_val));}
      | /* epsilon */ compute { Decl.consts = NewTable(); }
      ;
  }
}
```

The calculator language with declarations DecCalc now inherits rules from both languages.

```
Language DecCalc extends MulCalc, Dec {
  rule Prog {
   P ::= Decl E  compute {
             P.val = E.val; E.consts = Decl.consts; }
  }
  rule EXP extends MulCalc.EXP {
    E ::= E "+" T  compute {/* extend semantics*/
      value_distribution<E[0].consts; E[1].consts, T.consts>}
      |  E "-" T  compute {/* extend semantics*/
      value_distribution<E[0].consts; E[1].consts, T.consts>}
      | T         compute {/*extend semantics*/
      value_distribution<E.consts; T.consts>}
      ;
  }
  rule TERM1 extends MulCalc.TERM1 {
   compute {/* extend semantics */
     value_distribution<T[0].consts; T[1].consts, F.consts>}
  }
  rule TERM2 extends MulCalc.TERM2 {
   compute {/*extend semantics*/
      value_distribution<T.consts; F.consts>}
  }
  rule FACTOR overrides MulCalc.FACTOR {
   F ::= ident compute {
          F.val = GetValue(F.consts, ident.lex_val); }
     |  int    compute { F.val = Value(int.lex_val); }
     |  "(" E ")" compute { F.val = E.val;
                            E.consts = F.consts; }
     ;
  }
}
```

In semantic rules, which are an extension from ancestor rules such as `EXP` and `TERM1`, only new attribute computations are added. In these new attribute computations the symbol table is distributed. Since the rule `FACTOR` is overriden, computations for the attribute `val` must be written again.

The concrete realisation of inheritance depends on many low level implementation issues such as the use of pseudo-variables self and super, delegation or concatenation, direction and completion of message lookup and ordered or unordered inheritance [TAI 96]. In our case the subject of inheritance is a specification which is further used in compiler generation time to automatically generate the compiler. In its basic form, inheritance can be viewed as record combination (in our case as specification combination) which can be implemented either by delegation or concatenation. We used delegation (in first release concatenation) with asymmetric descendant-driven direction of lookup [TAI 96]. A problematic issue of multiple inheritance are name collisions. An unordered inheritance approach is used where name collisions are considered as errors. Such a case arises when two or more ancestor specifications use the same name of the property. In this case all ambiguities must be resolved explicitly. The name resolution process is described with the following algorithm.

Input:

Id – name of the property to be resolved
L – current language
overridenIdentifiers(*E*) – set of overriden identifiers for language *E*
supers(*E*) – set of ancestor languages for language *E*

Output:

Result – set of resolved properties

method resolve(Id, *L*)
  ToVisit := {*L*}
  Result := ∅
  do
    LookIn := *ToVisit*
    ToVisit := ∅
    for ∀ *E* ∈ *LookIn* **do**
      if (Id ∈ *E*) **then**
        if (Id ∉ *overridenIdentifiers*(*E*)) **then** *Result* := *Result* ∪ Id
      else
        ToVisit := *ToVisit* ∪ *supers*(*E*)
      endif
    endfor
  while (To*Visit* ≠ ∅)
return *Result*

If someone should like to make counter parts with object-oriented languages, then the left hand nonterminal roughly corresponds to the method and the right hand part of production corresponds to the method body. Each nonterminal symbol in the right part of production can be seen as method invocation or message lookup. And the whole language specification can be seen as a class.

```
E ::= self.E ″+″ self.T
  compute { E[0].val = E[1].val + T.val; }
```

Specifications are compiled to the monolithic form for further processing. A part of generated monolithic form for the language MulCalc is (name resolving is written in the comments):

```
AddSubCalc.digit          [0-9]
AddSubCalc.operator       "+" | "-"
AddSubCalc.whitespace     [\ \n\t]+
MulCalc.operator          "+" | "-" | "*"
MulCalc.separator         "(" | ")"
AddSubCalc.E ::= AddSubCalc.E "+" MulCalc.T
// self.E→MulCalc.E→AddSubCalc.E "+" self.T→MulCalc.T
 compute { AddSubCalc.E[0].val =
              AddSubCalc.E[1].val + MulCalc.T[0].val; }
. . .
MulCalc.T ::= MulCalc.F    // self.F→MulCalc.F
 compute { MulCalc.T[0].val = MulCalc.F[0].val; }
MulCalc.F ::= "(" AddSubCalc.E ")"
// self.E→MulCalc.E→AddSubCalc.E
 compute { MulCalc.F[0].val = AddSubCalc.E[0].val; }
MulCalc.F ::= AddSubCalc.digit
// self.digit→MulCalc.digit→ AddSubCalc.digit
 compute { MulCalc.F[0].val =
              Value(AddSubCalc.digit[0].lex_val);}
```

We have already gained some experience in using multiple attribute grammars inheritence. Their benefits are:

- specifications are extensible since the language designer writes only new and specialized specifications,
- specifications are reusable since specifications are inherited from ancestor specifications, and
- the language designer can construct the programming language specification from multiple specifications.

3.3 Reusable Semantic Domains

The last possibilities of reuse found in semantic specifications are reusable semantic domains. Attributes which represent semantic information belong to various semantic domains [MEY 90] and can also be objects having all properties of an object-oriented programming language. They can be inherited and specialised depending upon semantic necessities of the language. Attributes can be objects of classes derived from classes specified in the library with an already defined behaviour. In this way the attribute behaviour is inherited from its base class. Some of the typical attributes that can be reused are environment, store, configuration, etc. In the previous example the symbol table can be implemented as an instance of library class and hence no implementation for operations such as Insert, GetValue and NewTable is needed. With this approach we are able to reuse various attribute domains in implementing different programming languages [MER 96].

The benefits of object-oriented attribute domain implementation are:

- a domain can be inherited and specialised according to language design necessities,
- in specifying new languages an attribute domain can be reused with inheritance or aggregation,
- internal details of attribute domains are abstracted out and therefore not exposed to semantic rules,
- changes in the implementation of an attribute domain have no influence on semantic rules.

4 Implementation and Experience

We have developed a compiler/interpreter generator tool LISA [MER 95, ŽUM 97] which operates in MS Windows environment and automatically produces a compiler or an interpreter in C++ from the attribute grammar. The tool LISA can be classified as a kind of application generator [KRU 92] where the complete software system design is reused, algorithms and data structures are automatically selected so that the software development can concentrate on what the system should do rather than how it is done. The knowledge of lexical, syntax and semantic analysis is embedded in the compiler/interpreter generator, which is reused each time a compiler is generated. However, we went a step further and reused also parts of language specifications. Since there was interest from various institutions to publicly use the tool on various platforms we have implemented a new version of the tool with many improvements. The tool LISA ver. 2.0 is written in the object-oriented programming language Java where different platform concerns become obsolete. Also the generated compiler/interpreter and attribute computations in semantic rules are written in Java for the same reasons. In the new version the template and multiple inheritance approach are implemented. We have also improved the lexical, syntax and semantic analyzers which can operate standalone. The generated compiler/interpreter are also visualized in a manner similar to [AND 88]. Users of the generated compiler/interpreter also have the possibility to observe visually the work of lexical, syntax and semantic analyzers by watching the animation of finite state automata (fig 5), parse tree (fig 6) and semantic tree (fig 7).

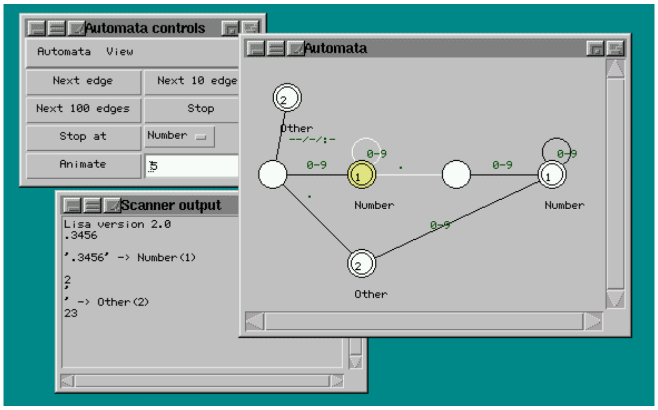

Figure 5. **Animation of finite state machine**

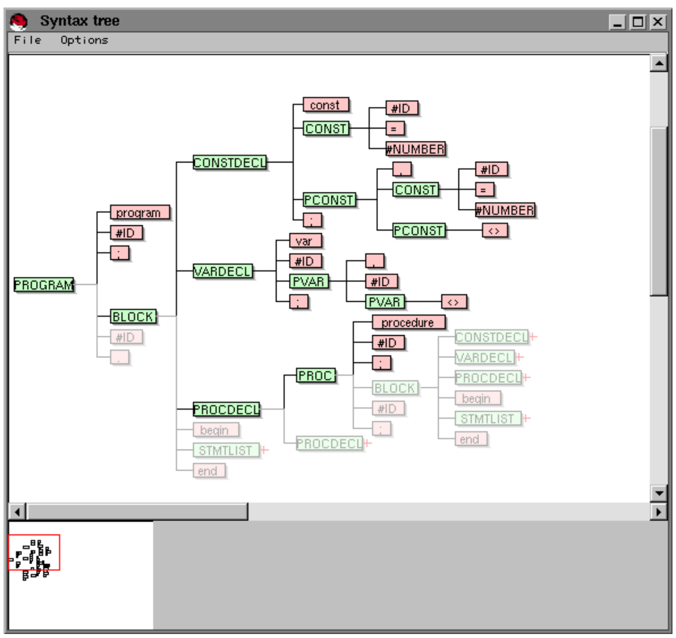

Figure 6. **Animation of syntax tree**

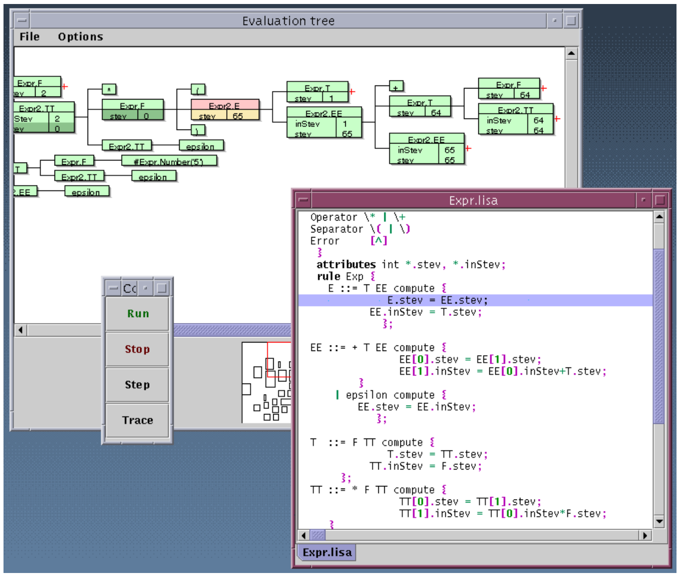

Figure 7. **Animation of semantic tree**

With the described approach we are able to reuse parts of semantic specifications of one programming language to define another. We have designed a simple functional language Fun which has been further extended to the imperative language Imp. From this example some indicators on reuse are obtained (Table 1). Two simple metrics are considered for measure reuse; LOC (lines of code) and NBP (number of basic parts). NBP is the number of basic parts which constitute semantic specifications. In the case of monolithic attribute grammars the basic parts are assignment statements, and in the case of attribute grammars with templates, the basic parts are templates, too.

Table 1: **LOC and NBP metrics**

	monolithic structure	**templates**	**templates and inheritance**
LOC(Fun)	99 lines (100%)	88 lines (88.8%)	88 lines (88.8%)
NBP(Fun)	54 assign. (100%)	34 temp. (62.9%)	34 temp. (62.9%)
LOC(Imp)	142 lines (100%)	132 lines (92.9%)	78 lines (54.9%)
NBP(Imp)	81 assign. (100%)	51 temp. (62.9%)	32 temp. (39.5%)

The semantic specification for the language Fun in monolithic attribute grammars is written in 99 lines, and in 88 lines with templates. The semantic specification for the language Imp is written in 142 lines in monolithic attribute grammars, in 132 lines with templates, and in 78 lines with multiple inheritance. Therefore, semantic specifications are shorter approximately by 10% with templates and with inheritance by 40%. In our opinion NBP is a more realistic metric since the number of lines depends on different styles, and moreover, templates are blocks on a higher abstraction level than assignment statements which are also more easily managed by the language designer. The NBP metric shows that 40% of improvements are achieved with templates and 60% with multiple inheritance.

Here, our approach is presented on the extended example of the PLM language [WIR 76]. Formal specifications of the PLM language are incrementally developed and are presented in appendix. First of all the language of constants is specified. This language can be reused whenever constant definitions are part of the language. Our first increment of the PLM language is the language PLM_1 where declarations and assignment statements can be written only in the main program. The next increment is the language PLM_2 where if and while statements are added, which introduces labels into the object code. Also constant definitions are added by multiple inheritance of language Constants. Then procedures and call statement are added to the final language PLM_3. The hierarchy of languages is presented in fig. 8.

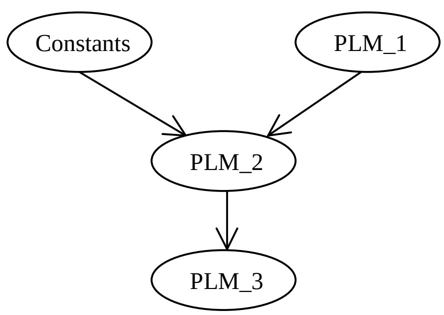

Figure 8. **The hierarchy of PLM languages**

An example of the program in the language PLM_3 and its semantically equivalent object code which is produced by automatically generated compiler (without comments) is:

```
program main;
const m = 7, n=85;
var x,y;
 procedure multiply;
  var a,b;
   begin
    a := x; b := m;
    while b>0 do
     if odd b then x := x + a; end;
     a := 2*a; b := b/2;
    end;
   end multiply;
```

```
begin
 x := m; call multiply;
end main.
```

```
                // goto subroutine main
                jmp    main

                // subroutine multiply
multiply
                // push on stack the value of global variable x
                // which is 1 static link up and with offset 2
                psh    1      2
                // pop the value of the stack and store to
                // local variable a (static link=0, offset=0)
                pop    0      0
                // push on stack the value of constant m
                // (static link=1, offset=0)
                psh    1      0
                // store to local variable b
                // (static link=0, offset=1)
                pop    0      1
                // logical expression of while loop
L_multiply0
                // push on stack the value of variable b
                psh    0      1
                // push on stack the value of literal 0
                lit    0
                // pop from stack 2 values, compare them for greater
                grt
                // negation of logical value
                not
                // if pop(stack)= true then goto L_multiply1
                jpc    L_multiply1
                // body of while loop
                // logical expression of if statement
                // push on stack the value of variable b
                psh    0      1
                // test of modulo 2
                odd
                // negation of logical value
                not
                // if pop(stack)= true then goto L_multiply2
                jpc    L_multiply2
                // true branch of if statement
                // push on stack the value of variable x
                psh    1      2
                // push on stack the value of variable a
                psh    0      0
                // pop from stack 2 values, add them and
                // push on stack new value
                add
                // pop the value of the stack and store to variable x
```

```
               pop    1      2
L_multiply2
               // push on stack the value of literal 2
               lit    2
               // push on stack the value of variable a
               psh    0      0
               // pop from stack 2 values, multiply them and
               // push on stack new value
               mul
               // pop the value of the stack and store to variable a
               pop    0      0
               // push on stack the value of variable b
               psh    0      1
               // push on stack the value of literal 2
               lit    2
               // pop from stack 2 values, divide them and
               // push on stack new value
               div
               // pop the value of the stack and store to variable b
               pop    0      1
               // goto label L_multiply0
               jmp    L_multiply0
               // return from subroutine multiply
L_multiply1    ret


               // subroutine main
main
               // push on stack the value of constant m
               psh    0      0
               // pop the value of the stack and store to variable x
               pop    0      2
               // call of subroutine multiply
               cal    1      multiply
               // return from subroutine main
               ret
```

5 Related Work

At present, many researchers are working on modular, extensible and reusable semantic specifications [DON 94, VAI 96] from which compilers or interpreters can be automatically produced. Modular attribute grammars MAG [DUE 90] are proposed as a solution to attribute pragmatic problems discussed above. The whole language specification consists of several MAGs. A single MAG is a set of patterns and associated templates. For each match between a production and pattern a set of attribute computations is generated. Both, the matching and the generation process are further constrained to generate only useful and meaningful attribute computations. As in our template approach also MAGs specify the semantic rules for sets of productions rather than for a particular production. For example, the following MAG modules are equivalent to our template bucket_brigade_left.

```
module in
  'goal → A …
    A.in = 0;
  A → B …
   B.in = A.in;
  A → … B C …
   C.in = B.out;
```

```
module out
 A →  … B  …
   B.out = B.in;
 A →  … B
  A.out = B.out;
 A →
  A.out = A.in;
```

We are convinced that our template approach offers a better abstraction of attribute computation since our template is a generic module parameterized by attribute instances, which is not the case with MAG modules. Also, in our approach the attribute computation generation is explicitly stated by the designer, and in MAG by the pattern matching process which is very difficult to follow. These difficulties with MAG were already observed by authors. On the other hand, MAG also does not have a counterpart to our multiple inheritance approach.

The last observation is also true in the case of regular right-part attribute grammars [JUL 84]. In regular right-part attribute grammars regular operators for concatenation, alternation and repetition are extended with the constancy operator and the distributed bucket brigade operator to describe frequently patterns in semantic rules. Our templates are more general since designers can create many other useful templates according to language design necessity.

We borrowed the idea of grammar inheritance from [AKS 91], and extended it to multiple attribute grammar inheritance. The difference between the approaches is also in the granularity of modification. In the approach of [AKS 91] modification is possible only for the whole production rule, since the name of the property is left hand nonterminal. The minor difference between the approaches is in the class of attribute grammars. We accept general absolute non-circular attribute grammars, while in [AKS 91] only S-attributed grammar is accepted. In our opinion the main weakness of the approach described in [AKS 91] is that it does not help the designer in the case when languages have similar semantics and a totally different syntax. We can handle this problem with the proposed templates.

In object-oriented attribute grammars [KOS 91] the concepts of class and class hierarchies have been introduced where nonterminals act as classes and class hierarchies have been derived from the context free grammar. Inheritance could be applied to attributes, attribute computations and syntactic patterns within one attribute grammar. We achieved a similar effect with templates. It is also well known that inherited attributes and class hierarchies produce some conflicts on well-definedness of attribute grammars and hence multiple inheritance is not allowed, and also inherited attributes can not be used in dynamic classes. In our approach a different view is chosen where the whole attribute grammar is a class without the above mentioned conflicts.

In our opinion the only widely accepted approach with reusability of attribute grammars is the approach presented in [KAS 94] and incorporated in the Eli compiler generator where, with remote attribute access and inheritance, an attribution module is defined which can be reused in a variety of applications. But with this approach the attribution module can be only constructed for those attribute computations where the attribute depends only on remote attributes. In this case computation is associated to a symbol rather than to production. With the inheritance described in [KAS 94] an attribute computation can be further independent from symbols used in particular language definitions. Our templates are similar to their attribution module but without such restriction. Attribute computation is also independent of grammar symbols since our templates are parameterized. Therefore their attribution module and inheritance are roughly equivalent to our template approach. In addition, with our multiple inheritance a language specification can be also extended from ancestor specifications.

6 Conclusion

In the paper a new approach to specifying programming languages is presented. The template is a new concept introduced into attribute grammars representing an abstraction over attribute computation which is independent of the structure of grammar productions. We also extended the idea of grammar inheritance to multiple attribute grammar inheritance. Our experience with templates and multiple inheritance shows that templates and multiple inheritance are useful in managing the complexity, reusability and extensibility of attribute grammars. Huge specifications become much shorter and are easier to read and maintain. On the other hand, with this approach it is also possible to reuse parts of programming languages specifications to define another. The benefit of the proposed approach is that it can extend the behaviour simply by adding new specifications while the conventional approach requires scavenging of existing specifications. The proposed approach is implemented in the compiler generator tool LISA ver. 2.0.

References

[AND 88] Andrews K., Henry R., Yamamoto W., "Design and Implementation of the UW Illustrated Compiler", *Proceedings of the Sigplan'88 Conference on Programming Language Design and Implementation*, pp. 105 - 114, 1988.

[AKS 91] Aksit M., Mostert R., Haverkort B., Grammar Inheritance, Technical Report, Department of Computer Science, University of Twente, 1991.

[DON 94] Dong J.S., Duke R., Rose G., "An Object-Oriented Approach to the Semantics of Programming Languages", *In Proc. 17th Australian Computer Science Conference*, pp. 767 - 775, 1994.

[DUE 90] Dueck G.D.P., Cormack G.V., "Modular Attribute Grammars", *Computer Journal*, Vol. 33, No. 2, pp. 164 - 172, 1990.

[HAN 97] Hankin C., Nielson H. R., Palsberg J., "Position Statements on Strategic Directions for Research on Programming Languages", *ACM Sigplan Notices*, Vol. 32, No. 1, pp. 59 - 65, 1997.

[JUL 84] Julling K., DeRemer F., "Regular right-part attribute grammars", *Proceedings of SIGPLAN'84 Symposium on Compiler Construction, ACM SIGPLAN Notices*, Vol. 19, No. 6, pp. 171 - 178, 1984.

[KAS 94] Kastens U., Waite W.M., "Modularity and reusability in attribute grammars", *Acta Informatica*, Vol. 31, 1994, pp. 601 - 627, 1994.

[KNU 68] Knuth D. E., "Semantics of contex-free languages", *Math. Syst. Theory*, Vol. 2, No. 2, pp. 127 - 145, 1968..

[KOS 91] Koskimies K., "Object-Orientation in Attribute Grammars", *Proceeding of the International Summer School SAGA*, Praga, Lecture Notes in Computer Science Vol. 545, pp. 297 - 329, 1991.

[KRU 92] Krueger C.W., "Software Reuse", *ACM Computing Surveys*, Vol. 24, No. 2, 1992.

[MER 95] Mernik M., Korbar N., Žumer V., "LISA: A Tool for Automatic Language Implementation", *ACM Sigplan Notices*, Vol. 30, No. 4, pp. 71 - 79, 1995.

[MER 96] Mernik M., Korbar N. Žumer V., "Object-Oriented Evaluation Paradigm in Attribute Grammar Based System LISA", *Journal of Computing and Information*, Vol. 2, No. 1, pp. 1233 – 1242, Special issue Proceedings of the 8th International Conference on Computing and Information ICCI'96, ISSN 1201-8511, 1996.

[MER 97] Mernik M., Žumer V., "Reusability of formal specifications in programming language description", *8th Annual Workshop on Software Reuse*, WISR-8, Columbus, Ohio, pp. 1 – 4, 1997.

[MER 98] Mernik M., Lenič M., Avdičaušević E., Žumer V., "The Template and Multiple Inheritance Approach into Attribute Grammars", *Proceedings of the International Conference on Computer Languages ICCL-98*, Chicago, pp. 102 - 110, 1998.

[MEY 90] Meyer B., *Introduction to the Theory of Programming Languages*, Prentice Hall, 1990.

[PAA 95] Paakki J., "Attribute Grammar Paradigms - A High-Level Methodology in Language Implementation", *ACM Computing Surveys*, Vol. 27, No. 2, pp. 196 - 255, 1995.

[TAI 96] Taivalsaari A., "On the Notion of Inheritance", *ACM Computer Surveys*, Vol. 28, No. 3, pp. 438 - 479, 1996.

[VAI 96] Vaidyanathan V., Bryant B. R., Formal Semantics Reuse for Different Programming Languages. Technical Report CIS-TR-96-010, Department of Computer and Information Sciences, University of Alabama at Birmingham, 1996.

[WIR 76] Wirth N., *Algorithms + Data Structures = Programs*, Prentice Hall, 1976.

[ŽUM 97] Žumer V., Korbar N., Mernik M., "Automatic Implementation of Programming Languages using Object Oriented Approach", *Journal of Systems Architecture*, Vol. 43, No. 1-5, pp. 203 - 210, 1997.

Appendix

```
// language of constants
language Constants {

 // regular definitions
 lexicon {
   RESERVE        const
   ID             [a-zA-Z_][a-zA-Z0-9_]*
   NUMBER         [0-9]+
   Operator       \=
   Delimiter      ,
   WhiteSpace     [\ \0x0A\0x0D\0x09]+
   ignore #WhiteSpace
 }

 // type definitions of attributes
 attributes
          Hashtable *.inSymbolTable, *.outSymbolTable;
          String *.procName;

 rule CONSTDECL {
   // sequence of constant definitions
   CONSTDECL ::= const CONST PCONST \; compute { ... }
             | /* epsilon */    compute { ... }
             ;
 }
 rule PCONST {
  // sequence of constant definitions
  PCONST ::= , CONST PCONST    compute {
     // distribute procedure name
     value_distribution<PCONST[0].procName;
                  CONST.procName, PCONST[1].procName>
     // collect constant definitions
     bucket_brigade_left<PCONST[0].inSymbolTable;
                         PCONST[0].outSymbolTable;
        CONST.inSymbolTable, PCONST[1].inSymbolTable;
        CONST.outSymbolTable,PCONST[1].outSymbolTable> }
          | /* epsilon */   compute {
    PCONST.outSymbolTable = PCONST.inSymbolTable; };
 }
 rule CONST {
  // constant definition
  CONST ::= #ID = #NUMBER   compute {
    // put in symbol table new entry of the form
    // (procedure name, constant name, constant vale)
    CONST.outSymbolTable = insert(CONST.inSymbolTable,
       CONST.procName, #ID.value(), #NUMBER.value()); };
 }

 // methods
}  // language Constants
```

```
language PLM_1 {

 // regular definitions
 lexicon {
     RESERVE        program | var | begin | end | odd
     ID             [a-zA-Z_][a-zA-Z0-9_]*
     NUMBER         [0-9]+
     Operator       \+ | \- | \:\= | \* | \< | \> | \= |
                    \/ | \#
     Delimiter      ; | . | \( | \) |,
     WhiteSpace     [\ \0x0A\0x0D\0x09]+
     ignore #WhiteSpace
  }

 // type definitions of attributes
 attributes Vector *.inCode, *.outCode;
             Hashtable *.inSymbolTable, *.outSymbolTable;
             String *.op, *.procName, *.code;
            boolean *.neg;

 rule PROGRAM {
  // program consist only from block which can not
  // contain procedures, therefore monolithic block
  // structure is used
  PROGRAM ::= program #ID \; BLOCK #ID \.    compute {
    BLOCK.inSymbolTable =
         subroutine(new Hashtable(), #ID.value());
    BLOCK.inCode = genCode(new Vector(), "jmp",
                        #ID.value());
    BLOCK.procName = #ID.value();
    PROGRAM.code = getObjectCode(BLOCK.outCode); } ;
 }
 rule BLOCK {
  // block consists from variable declarations and
  // statements
  BLOCK ::= VARDECL begin STMTLIST end      compute {
    // propagate symbol table through variable
    // declarations and statements
    propagate<BLOCK.inSymbolTable;
         VARDECL.inSymbolTable,STMTLIST.inSymbolTable;
         VARDECL.outSymbolTable>
    // distribute procedure name
    value_distribution<BLOCK.procName;
                 VARDECL.procName,STMTLIST.procName>
    // code is collected through statement list
    bucket_brigade_left<
              genCode(BLOCK.inCode,BLOCK.procName);
              BLOCK.outCode;
              STMTLIST.inCode;
              genCode(STMTLIST.outCode,"ret")> };
 }
```

```
 rule VARDECL { ... }

 rule PVAR { ... }

 rule STMTLIST {
  // sequence of statements
  STMTLIST ::= PSTMT \; STMTLIST   compute { ... };
 }

 rule PSTMT {
  // option of statement
  PSTMT ::= STMT    compute {
    // code for statement is added to code list
    bucket_brigade_left<PSTMT.inCode; PSTMT.outCode;
                        STMT.inCode;
                        STMT.outCode>
    // distribute procedure name
    value_distribution<PSTMT.procName; STMT.procName>
    // distribute symbol table
    value_distribution<PSTMT.inSymbolTable;
                       STMT.inSymbolTable>  }
       | /* epsilon */    compute { ... };
 }

 rule STMT {
  // only assigment statements are allowed
  STMT ::= #ID \:= EXPR   compute {
     // add following code of assignment to code list
     //    code for expression
     //    pop 0 offset
     bucket_brigade_left<STMT.inCode; STMT.outCode;
                         EXPR.inCode;
              genCode(EXPR.outCode, "pop", "0",
              offset(STMT.inSymbolTable,
                     STMT.procName,#ID.value()))>
    // distribute symbol table to expressions
    value_distribution<STMT.inSymbolTable;
                       EXPR.inSymbolTable>
    // distribute procedure name
    value_distribution<STMT.procName; EXPR.procName>};
 }

 rule EXPR { ... }
 ...

// methods
....

}  // language PLM_1
```

```
// adding if, while statements and constants
language PLM_2 extends PLM_1, Constants {

 // regular definitions
 lexicon {
   extends PLM_1.RESERVE if | while | do | #Constants.RESERVE
 }

 // type definitions of attributes
 attributes
   int *.inLabel, *.outLabel;

rule BLOCK overrides BLOCK {
  // now block consists from constant and variable
  //declarations, and statements
  BLOCK ::= CONSTDECL VARDECL begin STMTLIST end    compute {
    // propagate symbol table through constant, variable
    // declarations and statements
    propagate<BLOCK.inSymbolTable;
         CONSTDECL.inSymbolTable, VARDECL.inSymbolTable,
                                STMTLIST.inSymbolTable;
        CONSTDECL.outSymbolTable,VARDECL.outSymbolTable>
    // distribute procedure name
    value_distribution<BLOCK.procName;
                CONSTDECL.procName,VARDECL.procName,
                STMTLIST.procName>
    // code is collected through statement list
    bucket_brigade_left<
               genCode(BLOCK.inCode,BLOCK.procName);
                    BLOCK.outCode;
                    STMTLIST.inCode;
                genCode(STMTLIST.outCode,"ret")> };
    // if and while statements introduce labels
    STMTLIST.inLabel = 0; };
 }

rule STMTLIST extends STMTLIST { ... }

rule PSTMT extends PSTMT {
  PSTMT ::= STMT    compute {
    // extend semantic rules for labels
    bucket_brigade_left<PSTMT.inLabel; PSTMT.outLabel;
                    STMT.inLabel;
                    STMT.outLabel> }
      | /* epsilon */   compute { ... };
 }

 rule STMT extends STMT  {
  STMT ::= #ID \:= EXPR    compute {
      // extend semantic rules for labels
      STMT.outLabel = STMT.inLabel; };
 }
```

```
 rule STMT1 {
   // add if statement to statements
   STMT ::= if COND then STMTLIST end     compute {
     // add following code for if statement to code list
     //            code for condition
     //            not
     //            jpc labelX
     //            code for statement
     //   labelX
      bucket_brigade_left<STMT.inCode; STMT.outCode;
                         COND.inCode, STMTLIST.inCode;
          genCode(genCode(COND.outCode, "not"),
     "jpc", getLabel(STMT.procName, STMT.inLabel)),
          genCode(STMTLIST.outCode,
             getLabel(STMT.procName, STMT.inLabel))>
      // next available label is greater by 1
      bucket_brigade_left<STMT.inLabel+1; STMT.outLabel;
                         STMTLIST.inLabel;
                         STMTLIST.outLabel>
      // distribute symbol table
      value_distribution<STMT.inSymbolTable;
             STMTLIST.inSymbolTable, COND.inSymbolTable>
      // distribute procedure name
      value_distribution<STMT.ProcName;
                    STMTLIST.ProcName, COND.ProcName> };
 }

 rule STMT2  {
  STMT ::= while COND do STMTLIST end   compute { ... };
 }

} // language PLM_2
```

```
// adding procedure and call statement
language PLM_3 extends PLM_2 {

 // regular definitions
 lexicon {
   extends RESERVE procedure | call
 }

rule PROGRAM extends PLM_2.PROGRAM {
  compute {  // procedures introduce nested block structure
             // implementation of symbol table is different
   BLOCK.inSymbolTable =
      subroutine(new Hashtable(), #ID.value(), "null", 0); }
 }

rule BLOCK overrides PLM_2.BLOCK {
   // now block consists from constant, variable and
   // procedure declarations, and statements
   BLOCK ::= CONSTDECL VARDECL PROCDECL begin STMTLIST end
    compute {
    // propagate symbol table through constant, variable and
    // procedure declarations, and statements
    propagate<BLOCK.inSymbolTable;
        CONSTDECL.inSymbolTable, VARDECL.inSymbolTable,
              PROCDECL.inSymbolTable, STMTLIST.inSymbolTable;
        CONSTDECL.outSymbolTable, VARDECL.outSymbolTable,
               PROCDECL.outSymbolTable>
    // distribute procedure name
    value_distribution<BLOCK.procName;
      CONSTDECL.procName,VARDECL.procName,PROCDECL.procName,
                                          STMTLIST.procName>
    // code is collected through procedure declaration
    // and statement list
    bucket_brigade_left<BLOCK.inCode; BLOCK.outCode;
                        PROCDECL.inCode, STMTLIST.inCode;
                    genCode(PROCDECL.outCode,BLOCK.procName),
                         genCode(STMTLIST.outCode,"ret")>
      STMTLIST.inLabel = 0; };
 }

 rule PROCDECL {
   PROCDECL ::= PROC PROCDECL     compute { ... }
             | /* epsilon */
     compute { ... };
}

 rule PROC {
   // procedure declaration
  PROC ::= procedure #ID \; BLOCK #ID \;   compute {
     // add new procedure to symbol table and collect others
     // procedures
     bucket_brigade_left<subroutine(PROC.inSymbolTable,
```

```
          #ID.value(),
          getStaticLink(PROC.inSymbolTable, PROC.procName),
        1+getNestingLevel(PROC.inSymbolTable, PROC.procName));
                         PROC.outSymbolTable;
                         BLOCK.inSymbolTable;
                         BLOCK.inSymbolTable>
     // code is collected through block
     bucket_brigade_left<PROC.inCode; PROC.outCode;
                         BLOCK.inCode;
                         BLOCK.outCode>
     BLOCK.procName = #ID.value(); };
 }

 rule STMT3 extends PLM_2.STMT {
   // add call statement to statements
   STMT ::= call #ID     compute {
      // generate following code
      //    call procedure_distance procedure_name
      STMT.outCode = genCode(STMT.inCode, "cal",
          distanceProc(STMT.inSymbolTable,
           STMT.procName,#ID.value()),#ID.value());
      STMT.outLabel = STMT.inLabel; };
 }

 // methods
 method overrides Procedure {
  class Procedure {
   Vector vars, values;
   String staticLink;
   int nestingLevel;
   public String toString(){
     return staticLink+"  "+vars + "  "+nestingLevel; }
  }
 }

 method overrides subroutine {
  import java.util.*;
  Hashtable subroutine(Hashtable symbolTable, String procName,
                       String staticLink, int nestingLevel)  {
   if (!symbolTable.containsKey(procName)) {
     Procedure proc = new Procedure();
     proc.vars = new Vector();
     proc.values = new Vector();
     proc.staticLink = staticLink;
     proc.nestingLevel = nestingLevel;
     symbolTable.put(procName, proc); }
   else { System.out.println("Duplicate procedure name"); }
   return symbolTable;
  }
 }

} // language PLM_3
```