Multiple Attribute Grammar Inheritance

Marjan Mernik, Mitja Lenič, Enis Avdičaušević and Viljem Žumer

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

Abstract

The language design process should be supported by modularity and abstraction in a manner that allows incremental changes as easily as possible. To at least partially fulfill this ambitious goal a new object-oriented attribute grammar specification language which supports multiple attribute grammar inheritance is 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 or may override some specifications from ancestor specifications. 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 multiple attribute grammar inheritance is first introduced using an example, and thereafter by a formal model. The proposed approach is successfully implemented in the compiler/interpreter generator tool LISA ver. 2.0.

1. Introduction

We have developed a compiler/interpreter generator tool LISA which automatically produces a compiler or an interpreter from the ordinary attribute grammar specifications [1, 2]. But in this version of the tool, incremental language development was not supported, so the language designer had to design new languages from scratch or by scavenging old specifications. Other deficiencies of ordinary attribute grammars become apparent in specifications for real programming languages. Such specifications are large and unstructured, and are hard to understand, modify and maintain. Yet worse, small modifications of some parts in the specifications have widespread effects on the other parts of the specifications. Therefore specifications are not modular, extensible and reusable. Compared to modern programming languages, such as object-oriented or functional languages, the attribute grammar specification languages are far less advanced, specifically concerning the possibilities of abstraction, modularization, extensibility and reusability. Therefore, the integration of specification languages with various programming paradigms has developed in recent years. A detailed survey of attribute grammar based specification languages is given in [3]. We applied inheritance, a characteristic feature of object-oriented programming, in attribute grammars. A new object-oriented specification language with the paradigm $\boxed{\text{Attribute grammar} = \text{Class}}$, which is not included in [3], is presented in the paper. In [4] the new concept is introduced only in the informal manner through examples of a simple calculator language. We have incrementally designed various small programming languages, such as COOL and PLM, with multiple attribute grammar inheritance. Our experience with these non-trivial examples shows that multiple inheritance in attribute grammars is useful in managing the complexity, reusability and extensibility of attribute grammars. The benefit of this approach is also that for each language increment a compiler can be generated and the language tested. In this paper the reasons for introducing multiple inheritance into attribute grammars and the formal definition of multiple attribute grammar inheritance are presented. The multiple attribute grammar inheritance approach is successfully implemented in the newly developed version of the tool LISA ver. 2.0.

2. Background

Attribute grammars have been introduced by D.E. Knuth and since then have proved to be very 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 [5, 6, 7] 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. An attribute grammar consists of:

- A context-free grammar $G = (T, N, S, P)$, where T and N are the set of terminal symbols and nonterminal symbols; $S \in N$ is the start symbol, which doesn't appear on the right side of any production rule; and P is the set of productions. Now set $V = T \cup N$.
- A set of attributes $A(X)$ for each nonterminal symbol $X \in N$. $A(X)$ is divided into two mutually disjoint subsets, $I(X)$ of inherited attributes and $S(X)$ of synthesized attributes. Now set $A = \bigcup A(X)$. Let $Type$ denote a set of semantic domains. For each $a \in A(X)$, $a : type \in Type$ is defined which is the set of possible values of a.
- A set of semantic rules R. Semantic rules are defined within the scope of a single production. A production $p \in P, p : X_0 \rightarrow X_1 \ldots X_n$ $(n \geq 0)$ has an attribute occurrence $X_i.a$ if $a \in A(X_i)$, $0 \leq i \leq n$. A finite set of semantic rules R_p is associated with the production p with exactly one rule for each synthesized attribute occurrence $X_0.a$ and exactly one rule for each inherited attribute occurrence $X_i.a, 1 \leq i \leq n$. Thus R_p is a collection of rules of the form $X_i.a = f(y_1, \ldots, y_k), k \geq 0$, where $y_j, 1 \leq j \leq k$, is an attribute occurrence in p and f is a semantic function. In the rule $X_i.a = f(y_1, \ldots, y_k)$, the occurrence $X_i.a$ depends on each attribute occurrence $y_j, 1 \leq j \leq k$. Now set $R = \bigcup R_p$. For each production $p \in P, p : X_0 \rightarrow X_1 \ldots X_n$ $(n \geq 0)$ the set of defining occurrences of attributes is $DefAttr(p) = \{X_i.a | X_i.a = f(\ldots) \in R_p\}$. An attribute $X.a$ is called synthesized $(X.a \in S(X))$ if there exists a production $p : X \rightarrow X_1 \ldots X_n$ and $X.a \in DefAttr(p)$. It is called inherited $(X.a \in I(X))$ if there exists a production $q : Y \rightarrow X_1 \ldots X \ldots X_n$ and $X.a \in DefAttr(q)$.

Therefore, an attribute grammar is a triple $AG = (G, A, R)$ which consists of a context free grammar G, a finite set of attributes A and a finite set of semantic rules R.

3. Reasons for Introducing Multiple Inheritance into Attribute Grammars

The language design process should be supported by modularity and abstraction in a manner that allows to make incremental changes as easily as possible. This is one of the strategic directions of further research on programming languages. When introducing a new concept the designer has difficulties in integrating it into the language in an easy way. Therefore inheritance can be very helpful since it 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 is the semantic part changed, but the syntax rules and the lexicon may also need to be 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 [4]. Let us look at the informal definition of multiple attribute grammar inheritance and templates. 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 the programming language specification. Therefore, regular definitions, production rules, attributes, semantic rules and operations on semantic domains can be inherited, specialized or overridden from ancestor specifications. The language is specified in the following manner:

```
language L1 [extends L2, ..., LN] {
   lexicon {
      [[P] overrides | [P] extends] R regular expression
         ⋮
   }
   attributes type A1, ..., AM
      ⋮
   rule [[Y] extends | [Y] overrides] Z {
      X ::= X11 X12 ...  X1p compute {
            semantic functions }
         ⋮
      |
         Xr1 Xr2 ...  Xrt compute {
            semantic functions }
      ;
   }
      ⋮
   method [[N] overrides | [N] 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, rules which are generalized syntax rules that encapsulate semantic rules and methods on semantic domains. The benefits of multiple attribute grammar inheritance 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.

In our opinion the main weakness of multiple attribute grammar inheritance approach is that it does not help the designer in the case when languages have similar semantics and a totally different syntax. One possible solution to this problem is that of Composable Attribute Grammars (CAG) [11]. CAG consists of component attribute grammars and glue grammar. In component attribute grammar phrase structure and its semantics are expressed in terms of abstract, language independent context free grammar. The concrete syntactic structure is specified only in a glue grammar. In our approach templates are introduced with a similar goal. 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. Such patterns are described with templates. 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 [3], 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. Templates are also independent of a number of attribute occurrences which participate in semantic rules. For this purpose a variable list of arguments is proposed. As an example, a value distribution pattern is described as:

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

A template describing the value distribution pattern is:

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

The formal argument `X_in*` in the template `valueDistribution` is a variable list of arguments. Such arguments are denoted with an asterisk after the name. At template instantiation a part of semantic rules enclosed with braces is generated for each argument in the variable list. Together with a variable list of arguments some functions are defined which can be used for variable list manipulation (`first, last, succ, pred`). A successor for the last argument and a predecessor for the first argument do not exist. The usage of the above functions is presented in the next example. A pattern bucket brigade left is described as:

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

A template describing the pattern bucket brigade left is:

```
template <attributes Y_in, Y_out, X_in*, X_out*>
   compute bucketBrigadeLeft {
      if (empty(X_in*) && empty(X_out*))
         Y_out = Y_in;
      else
         first(X_in*) = Y_in;
         { X_in* = pred(X_out*);}
         Y_out = last(X_out*);
      endif
   }
```

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 compute {
   DECL.isGlobal = DECLS[0].isGlobal;
   DECL.inEnv = DECLS[0].inEnv;
   DECLS[1].isGlobal = DECLS[0].isGlobal;
   DECLS[0].outEnv = DECLS[1].outEnv;
   DECLS[1].inEnv = DECL.outEnv;
}
```

Specifications with templates are on higher abstraction level and hence more readable. In the semantic rules above, attribute computations are composed of bucked brigade and value distribution patterns.

```
DECLS ::= DECL \; DECLS compute {
   bucketBrigadeLeft<DECLS[0].inEnv, DECLS[0].outEnv,
      [DECL.inEnv, DECLS[1].inEnv], [DECL.outEnv, DECLS[1].outEnv]>
   valueDistribution<DECLS[0].isGlobal, [DECL.isGlobal, DECLS[1].isGlobal]>
}
```

The benefits of templates are:

- specifications are more readable and maintainable since templates are on higher abstraction level 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.

Let us look at the example of a simple language with assignment statements which may seem trivial, but a more concrete language would require several pages (for example COOL specifications are written on 25 pages). In the first attempt expressions have no side effects. The meaning of the program:

```
a := 5
b := a + 1 + a + a
```

is the following values: a=5, b=16. Let us develop the language without side effects in an incremental way. In each language increment only one semantic aspect are covered. In the first specification, only the rules for attribute `val` are given which reflect semantic aspect for value of an expression.

```
language Expr
   lexicon {
      Number   [0-9]+
      Operator \+
      ignore   [\0x09\0x0A\0x0D\ ]
   }
   attributes int *.val;
   rule Expression1 {
      EXPR ::= EXPR + TERM compute {
            EXPR[0].val = EXPR[1].val + TERM.val;
         };
```

```
    }
    rule Expression2 {
       EXPR ::= TERM compute {
             EXPR[0].val = TERM.val;
          };
    }
    rule Term1 {
       TERM ::= #Number compute {
             TERM.val = Integer.valueOf(#Number.value()).intValue();
          };
    }
 } //language Expr
```

The language ExprEnv is an extension of the language Expr where regular definitions Number, **ignore**, and attribute val are inherited and reused. The regular definition operator, and rules Expression1, Expression2, and Term1 are extended. Regular definition Identifier, and rules Start, Statements, Statement, and Term2 are added. In this language semantic aspect of symbol table management is covered.

```
 language ExprEnv extends Expr
    lexicon {
       Identifier          [a-z]+
       extends Operator    :=
    }
    attributes Hashtable *.inEnv, *.outEnv;
    rule Start {
       START ::= STMTS compute {
             STMTS.inEnv = new Hashtable();
             START.outEnv = STMTS.outEnv;
          };
    }
    rule Statements {
       STMTS ::= STMT STMTS compute {
             bucketBrigadeLeft<STMTS[0].inEnv,
                STMTS[0].outEnv, [STMT.inEnv, STMTS[1].inEnv],
                [STMT.outEnv, STMTS[1].outEnv]>
          }
       | compute {//epsilon
             bucketBrigadeLeft<STMTS.inEnv, STMTS.outEnv, [], []>
          };
    }
    rule Statement {
       STMT ::= #Identifier := EXPR compute {
             EXPR.inEnv = STMT.inEnv;
             STMT.outEnv = put(STMT.inEnv, #Identifier.value(), EXPR.val);
          };
    }
    rule extends Expression1 {
       EXPR ::= EXPR + TERM compute { // production can be omitted as in Expression2
             valueDistribution<EXPR[0].inEnv, [TERM.inEnv, EXPR[1].inEnv]>
          };
    }
```

```
    rule extends Expression2 {
       compute {
             valueDistribution<EXPR.inEnv, [TERM.inEnv]>
          }
    }
    rule Term2 {
       TERM ::= #Identifier compute {
             TERM.val = ((Integer)TERM.inEnv.get(
                #Identifier.value())).intValue();
          };
    }
    method Environment{
       import java.util.*;
       public Hashtable put(Hashtable env, String name, int val) {
             env = (Hashtable)env.clone();
             env.put(name, new Integer(val));
             return env;
          }
    }
 }// language ExprEnv
```

If later the designer needs expressions with side effects he/she must change only those parts which differ from ancestor specifications. In our example we have to use the bucket brigade left pattern instead of the value distribution pattern in rules: `Expression1`, `Expression2`, `Term1`, and `Term2`. Also, a new rule `Term3`, which produces a side effect with the following expression construct `[id := EXPR]` is introduced. The value of `id` is changed and propagated in further expressions. For example the next program:

```
a := 5
b := a + 1 + [a := 8] + a
```

produces the following values: a = 8 and b = 22. The language `ExprSideEffect` is an extension of the language `ExprEnv` where regular definitions `Number`, `Operator`, `Identifier` and **ignore**, attributes `inEnv`, `outEnv` and `val`, rules `Start`, `Statements` and method `Environment` are inherited and reused. The rules `Statement`, `Expression1`, `Expression2`, `Term1` and `Term2` are extended, and the regular definition `Separator` and the rule `Term3` are added.

```
 language ExprSideEffect extends ExprEnv
    lexicon {
       Separator      \[ | \]
    }
    rule extends Start {
       compute { }
    } // for starting production
    rule extends Statement {
       compute {
             bucketBrigadeLeft<STMT.inEnv, STMT.outEnv, [EXPR.inEnv],
                [put(EXPR.outEnv, #Identifier.value(), EXPR.val]))
          }
    }
    rule extends Expression1 {
       compute {
```

```
            bucketBrigadeLeft<EXPR[0].inEnv, EXPR[0].outEnv,
               [EXPR[1].inEnv, TERM.inEnv], [EXPR[1].outEnv, TERM.outEnv]>
         }
   }
   rule extends Expression2 {
      compute {
         bucketBrigadeLeft<EXPR.inEnv, EXPR.outEnv, [TERM.inEnv], [TERM.outEnv]>
      }
   }
   rule extends Term1 {
      compute {
            bucketBrigadeLeft<TERM.inEnv, TERM.outEnv, [], []>
         }
   }
   rule extends Term2 {
      compute {
            bucketBrigadeLeft<TERM.inEnv, TERM.outEnv, [], []>
         }
      }
   rule Term3 {
      TERM ::= [ #Identifier \:= EXPR ] compute {
            bucketBrigadeLeft<TERM.inEnv, TERM.outEnv, [EXPR.inEnv],
               [put(EXPR.outEnv, #Identifier.value(), EXPR.val)]>
            TERM.val = EXPR.val;
         }
   }
} // language ExprSideEffect
```

Let us look what semantic rules are generated from the template in rule `Term3`:

```
EXPR.inEnv = TERM.inEnv;
TERM.outEnv = put(EXPR.outEnv, #Identifier.value(), EXPR.val);
```

Language `ExprSideEffect` inherits properties from single parent. An example where language inherit properties from several parents can be found in [4, 21]. In [21] incremental development of PLM language is presented.

4. Formal Definition of Multiple Attribute Grammars Inheritance

Formally, inheritance can be characterized as $R = P \oplus \triangle R$ [8], where R denotes a newly defined object or class, P denotes the properties inherited from an existing object or class, $\triangle R$ denotes the incrementally added new properties that differentiate R from P, and $\oplus$ denotes an operation that combines $\triangle R$ with the properties of P. As a result of this combination, R will contain all the properties of P, except that the incremental modification part $\triangle R$ may introduce properties that overlap with those of P so as to redefine or cancel certain properties of P. Therefore, R may not always be fully compatible with P. The form of inheritance where properties are inherited from a single parent is known as single inheritance, as opposite to multiple inheritance where inheritance from several parents is allowed at the same time. Multiple inheritance can be formally characterized as $R = P_1 \oplus P_2 \oplus ... \oplus P_n \oplus \triangle R$. Before inheritance on regular definitions, context-free grammars and on attribute grammars are defined, let us look at the semantic domains used in formal definitions.

$ProdSem$ is a finite set of pairs (p, R_p), where p is a production and R_p is finite set of semantic rules associated with the production p.

$$\begin{aligned} ProdSem =& \{(p, R_p) | p \in P, p : X_0 \rightarrow X_1 X_2 ... X_n, \\ & R_p = \{X_i.a = f(X_{0.b}, \ldots, X_{j.c}) | X_i.a \in DefAttr(p)\}\} \end{aligned}$$

Properties in attribute grammars consist of lexical regular definitions, attribute definitions, rules which are generalized syntax rules that encapsulate semantic rules, and methods on semantic domains.

$$Property = RegdefName + AttributeName + RuleName + MethodName$$

For each language l, an $Ancestors(l)$ is a set of ancestors of the language l.

$$\begin{aligned} & Ancestors : Language \rightarrow \{Language\} \\ & Ancestors(l) = \{l_1, l_2, \ldots, l_n\} \end{aligned}$$

For each language l, a $LexSpec(l)$ is a set of mappings from regular definitions to regular expressions of the language l. A regular definition is a named regular expression.

$$\begin{aligned} & LexSpec : Language \rightarrow RegdefName \rightarrow RegExp \\ & LexSpec(l) = \{d_1 \mapsto rexp_1, \ldots, d_n \mapsto rexp_n\} \end{aligned}$$

For each language l, an $Attributes(l)$ is a set mappings from attributes to their types of the language l.

$$\begin{aligned} & Attributes : Language \rightarrow AttributeName \rightarrow Type \\ & Attributes(l) = \{a_1 \mapsto type_1, \ldots, a_n \mapsto type_n\} \end{aligned}$$

For each rule r in the language l, $Rules(l)(r)$ is a finite set of pairs (p, R_p), where p is a production and R_p is finite set of semantic rules associated with the production p.

$$\begin{aligned} & Rules : Language \rightarrow RuleName \rightarrow ProdSem \\ & Rules(l)(r) = \{(p, R_p) | p \in P, p : X_0 \rightarrow X_1 X_2 \ldots X_n, \\ & \qquad R_p = \{X_i.a = f(X_{0.b}, \ldots, X_{j.c} | X_i.a \in DefAttr(p))\}\} \end{aligned}$$

A set of properties of the language l_2, which are not accessible (and hence overridden) in the language l_1, is denoted with $OverriddenId(l_1, l_2)$.

$$\begin{aligned} & OverriddenId : (Language \times Language) \rightarrow \{Property\} \\ & OverriddenId(l_1, l_2) = \{pr_1, pr_2, \ldots, pr_n\} \end{aligned}$$

Rules inherited from ancestors must be merged with the rules in the specified language so that the underlying attribute grammar remains well defined. If production p exists in current and in inherited rules, then semantic rules must be merged $R_p = merge(R_{pC}, R_{pI})$. Otherwise rules are simply copied from inherited or current rules.

$$\begin{aligned} & Merge : ProdSem \times ProdSem \rightarrow ProdSem \\ & Merge(CurrentProd, InhProd) = \\ & \quad \{(p, R_p) | ((p, R_{pI}) \in InhProd \wedge (p, R_{pC}) \in CurrentProd \wedge R_p = merge(R_{pC}, R_{pI})) \vee \\ & \qquad ((p, R_p) \in InhProd \wedge (p, R_{pC}) \notin CurrentProd) \vee \\ & \qquad ((p, R_p) \in CurrentProd \wedge (p, R_{pI}) \notin InhProd))\} \end{aligned}$$

$merge(R_{pC}, R_{pI})$ is a set of semantic rules associated to production p where the semantic rule for the same attribute redefines the inherited ones.

$$\begin{aligned}&merge(R_{pC}, R_{pI}) =\\&\quad\{X_i.a = f(X_{0.b}, \ldots, X_{j.c}) | X_i.a \in DefAttr(p_C) \vee (X_i.a \in DefAttr(p_I) \wedge\\&\quad X_i.a \notin DefAttr(p_C))\}\end{aligned}$$

For the function $f : A \to B$, we let $f[a/b]$ be the function that acts just like f except that it maps specific value $a \in A$ to $b \in B$. That is:

$$\begin{aligned}&(f[a/b])(a) = b\\&(f[a/b])(a_0) = f(a_0); \forall a_0 \in A \wedge a_0 \neq a\end{aligned}$$

4.1. Regular definition inheritance

The input string can be recognized with different regular expressions even in monolitic lexical specifications. In such cases the first match rule is commonly used and the order of regular expressions becomes important. The concept of inheritance of regular definitions causes further problems as presented in the following example [4]:

```
AddSubCalc.digit      [0-9]
Dec.int               [0-9]+
```

For example, the input string '7' is recognized as `AddSubCalc.digit`. If reference to `Dec.int` was made in the syntax specifications, the error would be reported, despite the correctness of specifications. If the order of regular definitions were different, the same problem would appear with reference to `AddSubCalc.digit`. Our solution to this problem is to find all matching regular definitions for the input string. For example, the result of lexical analyses for the input string '7' would be the set `{AddSubCalc.digit, Dec.int}`. In that case reference to both regular definitions can be made and therefore the sequence of regular definitions becomes irrelevant. For these reasons the inheritance of regular definitions is defined in the following way:

Let $E_1, E_2, ..., E_m$ be sets of mappings from regular definitions to regular expressions of languages $l_1, l_2, \ldots l_m$ formally defined as

$$\begin{aligned}E_1 &= \{d_{11} \mapsto e_{11}, d_{12} \mapsto e_{12}, \ldots, d_{1k} \mapsto e_{1k}\}\\E_2 &= \{d_{21} \mapsto e_{21}, d_{22} \mapsto e_{22}, \ldots, d_{2l} \mapsto e_{2l}\}\\&\vdots\\E_m &= \{d_{m1} \mapsto e_{m1}, d_{m2} \mapsto e_{m2}, \ldots, d_{mn} \mapsto e_{mn}\}\end{aligned}$$

where d_{ij} is a regular definition and e_{ij} is a regular expression, then $E = E_2 \oplus \ldots \oplus E_m \oplus \triangle E_1$, where E_1, which inherits from $E_2, \ldots, E_m$, is defined as:

$$E = E_1 \cup \ldots \cup E_m.$$

Example:

$E_{Expr} = \{$`Expr.Number` $\mapsto$ `[0-9]+`, `Expr.Operator` $\mapsto$ `\+`,
`Expr.ignore` $\mapsto$ `[\0x09 \0x0A\0x0D\ ]`$\}$

$E_{Expr} \oplus \triangle E_{ExprEnv} = \{$`Expr.Number` $\mapsto$ `[0-9]+`, `Expr.Operator` $\mapsto$ `\+`,
`Expr.ignore` $\mapsto$ `[\0x09 \0x0A\0x0D\ ]`$\} \cup$

$\{$`ExprEnv.Identifier` $\mapsto$ `[a-z]+`, `ExprEnv.Operator` $\mapsto$ `:= | Expr.Operator`$\}$

$E_{Expr} \oplus \triangle E_{ExprEnv} \oplus \triangle E_{ExprSideEffect} =$ $\{$`Expr.Number` $\mapsto$ `[0-9]+`, `Expr.Operator` $\mapsto$ `\+`,
$\{$`Expr.ignore` $\mapsto$ `[\0x09 \0x0A\0x0D\ ]` $\} \cup$
`ExprEnv.Identifier` $\mapsto$ `[a-z]+`, `ExprEnv.Operator` $\mapsto$ `:= | Expr.Operator`$\} \cup$
$\{$`ExprSideEffect.Separator` $\mapsto$ `\[ | \]`$\}$

4.2. Context-free grammar inheritance

Let G_1, G_2, ..., G_m be context-free grammars, formally defined as

$$\begin{aligned} &G_1 = (T_1, N_1, S_1, P_1),\\ &G_2 = (T_2, N_2, S_2, P_2),\\ &\qquad\vdots\\ &G_m = (T_m, N_m, S_m, P_m), \text{ then} \end{aligned}$$

$G = G_2 \oplus \ldots \oplus G_m \oplus \triangle G_1$, where G_1, which inherits from G_2, ..., G_m, is defined as

$$\begin{aligned} &G = (T, N, S_1, P), \text{ where}\\ &T = T_1 \oslash \ldots \oslash T_m,\\ &N = N_1 \oslash \ldots \oslash N_m,\\ &P = P_1 \odot \ldots \odot P_m. \end{aligned}$$

Note that the start nonterminal symbol of context free grammar G is the start nonterminal of context-free grammar G_1. Since the incrementally added new productions P_1 may override some productions where terminal and nonterminal symbols are defined, the final set of terminal symbols T and the set of nonterminal symbols N are not simply a union of inherited terminal and nonterminal symbols. The operation $\oslash$ is defined as:

$$\begin{aligned} &V_1 \oslash V_2 \oslash \ldots \oslash V_m =\\ &\quad V_1 \cup (V_2 \setminus \{x | x \in OverriddenSym(l_1, l_2)\}) \cup \ldots \cup (V_m \setminus \{x | x \in OverriddenSym(l_1, l_m)\}). \end{aligned}$$

Where, $OverriddenSym(l_1, l_2)$ is a set of overridden symbols of the language l_2 which are not accessible from language l_1. Also, the set of productions P is not simply a union of inherited productions since some productions may be overridden or cause horizontal overlap [8]. The operation is defined as:

$$\begin{aligned} P = P_1 \odot \ldots \odot P_m = &P_1 \cup (P_2 \setminus \{p | p \in fst(Rules(l_2)(r)) \wedge r \in OverriddenId(l_1, l_2)\}) \cup \ldots \cup\\ &(P_m \setminus \{p | p \in fst(Rules(l_m)(r)) \wedge r \in OverriddenId(l_1, l_m)\}) \wedge\\ &dom(Rules(l_i)) \cap dom(Rules(l_j)) = \emptyset, i = 2..m, j = 2..m \wedge i \neq j. \end{aligned}$$

Example:

$OverriddenId($`ExprEnv`, `Expr`$) = \emptyset$
$OverriddenSym($`ExprEnv`, `Expr`$) = \emptyset$
$OverriddenId($`ExprSideEffect`, `ExprEnv`$) = \emptyset$
$OverriddenSym($`ExprSideEffect`, `ExprEnv`$) = \emptyset$

$G_{Expr} =$ ($\{$`#Number, +`$\}$, $\{$`EXPR, TERM`$\}$, `EXPR`,
$\{$`EXPR` $\rightarrow$ `EXPR + TERM`, `EXPR` $\rightarrow$ `TERM`, `TERM` $\rightarrow$ `#Number`$\}$)

$G_{Expr} \oplus \triangle G_{ExprEnv} =$ ($\{$`#Number, +, #Identifier, :=` $\}$,

$\{$`EXPR, TERM, START, STMTS, STMT`$\}$,
`START,` $\{$`EXPR` $\rightarrow$ `EXPR + TERM, EXPR` $\rightarrow$ `TERM,`
`TERM` $\rightarrow$ `#Number, TERM` $\rightarrow$ `#Identifier,`
`START` $\rightarrow$ `STMTS, STMTS` $\rightarrow$ `STMT STMTS,`
`STMT` $\rightarrow$ `#Identifier := EXPR`$\})$

$G_{Expr} \oplus \triangle G_{ExprEnv} \oplus \triangle G_{ExprSideEffect} = (\{$`#Number, +, #Identifier, :=, [, ]`$\}$,
$\{$`EXPR, TERM, START, STMTS, STMT`$\}$`, START,`
$\{$`EXPR` $\rightarrow$ `EXPR + TERM, EXPR` $\rightarrow$ `TERM,`
`TERM` $\rightarrow$ `#Number, TERM` $\rightarrow$ `[#Identifier := EXPR],`
`TERM` $\rightarrow$ `#Identifier, START` $\rightarrow$ `STMTS,`
`STMTS` $\rightarrow$ `STMT STMTS, STMT` $\rightarrow$ `#Identifier := EXPR`$\})$

4.3. Multiple attribute grammar inheritance

Let AG_1, AG_2, ..., AG_m be attribute grammars formally defined as:

$AG_1 = (G_1, A_1, R_1)$,
$AG_2 = (G_2, A_2, R_2)$,
$\vdots$
$AG_m = (G_m, A_m, R_m)$, then

$AG = AG_2 \oplus \ldots \oplus AG_m \oplus \triangle AG_1$, where AG_1, which inherits from $AG_2, \ldots, AG_m$, is defined as

$AG = (G, A, R)$, where
$G = G_2 \oplus \ldots \oplus G_m \oplus \triangle G_1$,
$A = A_1 \ominus \ldots \ominus A_m$,
$R = R_1 \otimes \ldots \otimes R_m$.

Since each attribute has a type, a set of attributes A_i is defined as:

$$A_i = \{a_{i1} \mapsto type_{i1}, \ldots, a_{in} \mapsto type_{in}\}.$$

Then, $A = A_1 \ominus \ldots \ominus A_m$ can not be defined simply as a union, since the same attribute can be of different type in a different set A_i. This situation denotes horizontal or vertical overlapping. Since unordered inheritance is used, horizontal overlapping is forbidden and vertical overlapping is resolved by asymmetric descendant-driven lookup [8]. Hence, $A = A_1 \ominus \ldots \ominus A_m$ is defined as:

$$\begin{aligned} A = & A_1 \cup (A_2 \setminus \{a_{1p} \mapsto type_{1p} | a_{1p} \in fst(A_1)\}) \cup \ldots \cup (A_m \setminus \{a_{1p} \mapsto type_{1p} | a_{1p} \in fst(A_1)\}) \wedge \\ & (\neg \exists a_{ji}, j = 2..m, i = 1..n, k \neq l : (a_{ji} \mapsto type_{jk}) \wedge (a_{ji} \mapsto type_{jl}) \wedge (type_{jk} \neq type_{jl})). \end{aligned}$$

The set of semantic rules R is not a simple union of inherited semantic rules, since some semantic rules may be overridden or may cause horizontal overlap. In any case, current semantic rules have to be merged with inherited semantic rules.

$$\begin{aligned} R = & R_1 \otimes \ldots \otimes R_m = \\ & R_1 \cup snd(Merge((P_1, R_1), (P_2, R_2 \setminus \{R_p | R_p \in snd(Rules(l_2)(r)) \wedge r \in OverriddenId(l_1, l_2)\}))) \\ & \cup \ldots \cup \\ & snd(Merge((P_1, R_1), (P_m, R_m \setminus \{R_p | R_p \in snd(Rules(l_m)(r)) \wedge r \in OverriddenId(l_1, l_m,)\}))) \\ & \wedge\ dom(Rules(l_i)) \cap dom(Rules(l_j)) = \emptyset, i = 2..m, j = 2..m, i \neq j. \end{aligned}$$

Example:

$AG_{Expr} = (G_{Expr}, A_{Expr}, R_{Expr})$
$A_{Expr} = \{$`EXPR.val` $\mapsto$ `int`, `TERM.val` $\mapsto$ `int`$\}$
$R_{Expr} = R_{Expression1} \cup R_{Expression2} \cup R_{Term1}$
$R_{Expression1} = R_{EXPR \rightarrow EXPR+TERM}$
$R_{Expression2} = R_{EXPR \rightarrow TERM}$
$R_{Term1} = R_{TERM \rightarrow \#Number}$
$R_{EXPR \rightarrow EXPR+TERM} = \{$`EXPR[0].val=EXPR[1].val + TERM[0].val`$\}$
$R_{EXPR \rightarrow TERM} = \{$`EXPR[0].val=TERM[0].val`$\}$
$R_{TERM \rightarrow \#Number} = \{$`TERM[0].val=Integer.valueOf(#Number[0].value()).intValue()`$\}$

$AG_{Expr} \oplus \triangle AG_{ExprEnv} = (G_{Expr} \oplus \triangle G_{ExprEnv}, A_{ExprEnv} \ominus A_{Expr}, R_{ExprEnv} \otimes R_{Expr})$
$A_{ExprEnv} \ominus A_{Expr} = \{$`START.outEnv` $\mapsto$ `Hashtable`, `STMTS.inEnv` $\mapsto$ `Hashtable`,
`STMTS.outEnv` $\mapsto$ `Hashtable`, `STMT.inEnv` $\mapsto$ `Hashtable`, `STMT.outEnv` $\mapsto$ `Hashtable`,
`EXPR.inEnv` $\mapsto$ `Hashtable`, `TERM.inEnv` $\mapsto$ `Hashtable` $\} \cup$
$\{$`EXPR.val` $\mapsto$ `int`, `TERM.val` $\mapsto$ `int`$\}$
$R_{ExprEnv} \otimes R_{Expr} = (R_{Start} \cup R_{Statements} \cup R_{Statement} \cup merge(R_{Expression1}, Expr.R_{Expression1})$
$\cup\ merge(R_{Expression2}, Expr.R_{Expression2}) \cup R_{Term2}) \cup Expr.R_{Term1}$
$R_{Start} = R_{START \rightarrow STMTS}$
$R_{Statements} = R_{STMTS \rightarrow STMT\ \ STMTS} \cup R_{STMTS \rightarrow \epsilon}$
$R_{Statement} = R_{STMT \rightarrow \#Identifier:=EXPR}$
$R_{Expression1} = R_{EXPR \rightarrow EXPR+TERM}$
$R_{Expression2} = R_{EXPR \rightarrow TERM}$
$R_{Term2} = R_{TERM \rightarrow \#Identifier}$
$R_{START \rightarrow STMTS} = \{$`STMTS[0].inEnv = new Hashtable()`,
`START[0].outEnv = STMTS[0].outEnv`$\}$
$R_{STMTS \rightarrow STMT\ \ STMTS} = \{$`STMT[0].inEnv = STMTS[0].inEnv`,
`STMTS[1].inEnv = STMT[0].outEnv`, `STMTS[0].outEnv=STMTS[1].outEnv`$\}$
$R_{STMTS \rightarrow \epsilon} = \{$`STMTS[0].outEnv = STMTS[0].inEnv`$\}$
$R_{STMT \rightarrow \#Identifier:=EXPR} = \{$`EXPR[0].inEnv=STMT[0].inEnv`,
`STMT[0].outEnv = put(STMT[0].inEnv,#Identifier.value(), EXPR[0].val)` $\}$
$R_{EXPR \rightarrow EXPR+TERM} = \{$`TERM[0].inEnv = EXPR[0].inEnv`, `EXPR[1].inEnv = EXPR[0].inEnv`$\}$
$R_{EXPR \rightarrow TERM} = \{$`TERM[0].inEnv=EXPR[0].inEnv`$\}$
$R_{TERM \rightarrow \#Identifier} =$
$\{$`TERM[0].val=((Integer)TERM[0].inEnv.get(#Identifier[0].value())).intValue()`$\}$
$merge(R_{Expression1}, Expr.R_{Expression1}) = \{$`EXPR[0].val=EXPR[1].val + TERM[0].val`,
`TERM[0].inEnv = EXPR[0].inEnv`, `EXPR[1].inEnv = EXPR[0].inEnv`$\}$
$merge(R_{Expression2}, Expr.R_{Expression2}) = \{$`EXPR[0].val = TERM[0].val`,
`TERM[0].inEnv=EXPR[0].inEnv`$\}$

$AG_{Expr} \oplus \triangle AG_{ExprEnv} \oplus \triangle AG_{ExprSideEffect} = (G_{Expr} \oplus \triangle G_{ExprEnv} \oplus \triangle G_{ExprSideEffect},$
$A_{ExprSideEffect} \ominus A_{ExprEnv} \ominus A_{Expr}, R_{ExprSideEffect} \otimes R_{ExprEnv} \otimes R_{Expr})$
$A_{ExprSideEffect} \ominus A_{ExprEnv} \ominus A_{Expr} = \{$`TERM.outEnv` $\mapsto$ `Hashtable`, `EXPR.outEnv`
$\mapsto$ `Hashtable`$\} \cup \{$`START.outEnv` $\mapsto$ `Hashtable`, `STMTS.inEnv` $\mapsto$ `Hashtable`,
`STMTS.outEnv` $\mapsto$ `Hashtable`,`STMT.inEnv` $\mapsto$ `Hashtable`, `STMT.outEnv` $\mapsto$ `Hashtable`,
`EXPR.inEnv` $\mapsto$ `Hashtable`, `TERM.inEnv` $\mapsto$ `Hashtable` $\} \cup$
$\{$`EXPR.val` $\mapsto$ `int`, `TERM.val` $\mapsto$ `int`$\}$
$R_{ExprSideEffect} \otimes R_{ExprEnv} \otimes R_{Expr} = merge(R_{Start}, ExprEnv.R_{Start}) \cup ExprEnv.R_{Statements} \cup$
$merge(R_{Statement}, ExprEnv.R_{Statement}) \cup merge(R_{Expression1}, ExprEnv.R_{Expression1})$
$\cup\ merge(R_{Expression2}, ExprEnv.R_{Expression2}) \cup merge(R_{Term1}, ExprEnv.R_{Term1}))\ \cup$
$merge(R_{Term2}, ExprEnv.R_{Term2}) \cup R_{Term3}$

$R_{Start} = R_{START \to STMTS}$
$R_{Statement} = R_{STMT \to \#Identifier:=EXPR}$
$R_{Expression1} = R_{EXPR \to EXPR+TERM}$
$R_{Expression2} = R_{EXPR \to TERM}$
$R_{Term1} = R_{TERM \to \#Number}$
$R_{Term2} = R_{TERM \to \#Identifier}$
$R_{Term3} = R_{TERM \to [\#Identifier:=EXPR]}$
$R_{START \to STMTS} = \{\ \}$
$R_{STMT \to \#Identifier:=EXPR} = \{$`EXPR[0].inEnv = STMT[0].inEnv,`
`STMT[0].outEnv = put(EXPR[0].outEnv, #Identifier.value(), EXPR[0].val)`$\}$
$R_{EXPR \to EXPR+TERM} = \{$`EXPR[1].inEnv = EXPR[0].inEnv,`
`TERM[0].inEnv = EXPR[1].outEnv, EXPR[0].outEnv = TERM[0].outEnv`$\}$
$R_{EXPR \to TERM} = \{$`TERM[0].inEnv=EXPR[0].inEnv, EXPR[0].outEnv = TERM[0].outEnv`$\}$
$R_{TERM \to \#Number} = \{$`TERM[0].outEnv = TERM[0].inEnv`$\}$
$R_{TERM \to \#Identifier} = \{$`TERM[0].outEnv = TERM[0].inEnv`$\}$
$R_{TERM \to [\#Identifier:=EXPR]} = \{$`EXPR[0].inEnv = TERM[0].inEnv,`
`TERM[0].outEnv = put(EXPR[0].outEnv, #Identifier.value(), EXPR[0].val,`
`TERM[0].val = EXPR[0].val)`$\}$
$merge(R_{Start}, ExprEnv.R_{Start}) = \{$`STMTS[0].inEnv = new Hashtable(),`
`START[0].outEnv = STMTS[0].outEnv`$\}$
$merge(R_{Statement}, ExprEnv.R_{Statement}) = \{$`EXPR[0].inEnv = STMT[0].inEnv,`
`STMT[0].outEnv = put(EXPR[0].outEnv, #Identifier.value(), EXPR[0].val)`$\}$
$merge(R_{Expression1}, ExprEnv.R_{Expression1}) = \{$`EXPR[0].val=EXPR[1].val + TERM[0].val,`
`EXPR[1].inEnv = EXPR[0].inEnv, TERM[0].inEnv = EXPR[1].outEnv,`
`EXPR[0].outEnv = TERM[0].outEnv`$\}$
$merge(R_{Expression2}, Expr.R_{Expression2}) = \{$`EXPR[0].val = TERM[0].val,`
`TERM[0].inEnv = EXPR[0].inEnv, EXPR[0].outEnv = TERM[0].outEnv` $\}$
$merge(R_{Term1}, ExprEnv.R_{Term1}) =$
$\{$`TERM[0].val=Integer.valueOf(#Number[0].value()).intValue(),`
`TERM[0].outEnv = TERM[0].inEnv` $\}$
$merge(R_{Term2}, ExprEnv.R_{Term2}) =$
$\{$`TERM[0].val=((Integer)TERM[0].inEnv.get(#Identifier[0].value())).intValue()),`
`TERM[0].outEnv = TERM[0].inEnv` $\}$

Let us look in more detail what is the result of operation $merge(R_{Expression1}, ExprEnv.R_{Expression1})$. Semantic rules asociated to production `EXPR → EXPR + TERM` in language `ExprEnv` are semantic rules obtained from operation $merge(R_{Expression1}, Expr.R_{Expression1})$.

$merge(R_{Expression1}, Expr.R_{Expression1}) = \{$`EXPR[0].val=EXPR[1].val + TERM[0].val,`
`TERM[0].inEnv = EXPR[0].inEnv, EXPR[1].inEnv = EXPR[0].inEnv`$\}$

Defined attributes in production `EXPR → EXPR + TERM` of the language `ExprEnv` are: `EXPR[0].val`, `TERM[0].inEnv`, `EXPR[1].inEnv`, and in production `EXPR → EXPR + TERM` of the language `ExprSideEffect` the defined attributes are `EXPR[1].inEnv`, `Term[0].inEnv`, `EXPR[0].outEnv`. Attributes `TERM[0].inEnv` and `EXPR[1].inEnv` are defined in both productions, however redefined semantic rules are used. Therefore the result of operation $merge(R_{Expression1}, ExprEnv.R_{Expression1})$ is

$\{$`EXPR[0].val=EXPR[1].val + TERM[0].val,`
`EXPR[1].inEnv = EXPR[0].inEnv, TERM[0].inEnv = EXPR[1].outEnv,`
`EXPR[0].outEnv = TERM[0].outEnv`$\}$

4.4. Operational semantics

In this section the operational semantics of the specification language for the compiler generator tool LISA, which supports multiple attribute grammar inheritance, is presented. Only the descriptions of the most interesting rules are given.

4.4.1. Abstract syntax

$$
\begin{array}{lll}
\mathit{LanguageSpec} & ::= & \textbf{language}\ l\ \text{"\{"}\ \mathit{Body}\ \text{"\}"} \\
 & \mid & \textbf{language}\ l_1\ \textbf{extends}\ l_2,\ l_3\ \text{"\{"}\ \mathit{Body}\ \text{"\}"} \\
\mathit{Rdef} & ::= & d\ \mathit{Rexp} \mid \textbf{extends}\ d\ \mathit{Rexp} \mid \textbf{extends}\ l.d\ \mathit{Rexp} \\
 & \mid & d_1\ \textbf{extends}\ d_2\ \mathit{Rexp} \\
 & \mid & d_1\ \textbf{extends}\ l.d_2\ \mathit{Rexp} \mid \textbf{overrides}\ d\ \mathit{Rexp} \mid \textbf{overrides}\ l.d\ \mathit{Rexp} \\
 & \mid & d_1\ \textbf{overrides}\ d_2\ \mathit{Rexp} \mid d_1\ \textbf{overrides}\ l.d_2\ \mathit{Rexp} \\
\mathit{Attr} & ::= & \mathit{Type}\ a; \\
\mathit{Rule} & ::= & \textbf{rule}\ r\ \text{"\{"}\ \mathit{Prods}\ \text{"\}"} \mid \textbf{rule extends}\ r\ \text{"\{"}\ \mathit{Prods}\ \text{"\}"} \\
 & \mid & \textbf{rule extends}\ l.r\ \text{"\{"}\ \mathit{Prods}\ \text{"\}"} \mid \textbf{rule}\ r_1\ \textbf{extends}\ r_2\ \text{"\{"}\ \mathit{Prods}\ \text{"\}"} \\
 & \mid & \textbf{rule}\ r_1\ \textbf{extends}\ l.r_2\ \text{"\{"}\ \mathit{Prods}\ \text{"\}"} \mid \textbf{rule overrides}\ r\ \text{"\{"}\ \mathit{Prods}\ \text{"\}"} \\
 & \mid & \textbf{rule overrides}\ l.r_2\ \text{"\{"}\ \mathit{Prods}\ \text{"\}"} \\
 & \mid & \textbf{rule}\ r_1\ \textbf{overrides}\ r_2\ \text{"\{"}\ \mathit{Prods}\ \text{"\}"} \\
 & \mid & \textbf{rule}\ r_1\ \textbf{overrides}\ l.r_2\ \text{"\{"}\ \mathit{Prods}\ \text{"\}"}
\end{array}
$$

4.4.2. Operational rules

The language l is evaluated in the context of domains *LexSpec*, *Attributes*, *Rules* and *Methods*. The result are lexical specifications, atrributes, rules and methods defined in the language l. In the first rule, the language l has no ancestors.

$$
\frac{\begin{array}{l} l, \mathit{LexSpec}, \mathit{Attributes}, \mathit{Rules}, \mathit{Methods} \vdash_{\mathit{Body}} \mathit{Body} : \\ \mathit{LexSpec}(l), \mathit{Attributes}(l), \mathit{Rules}(l), \mathit{Methods}(l) \end{array}}{\begin{array}{l} \mathit{LexSpec}, \mathit{Attributes}, \mathit{Rules}, \mathit{Methods} \vdash_{\mathit{LanguageSpec}} \\ \textbf{language}\ l\ \text{"\{"}\ \mathit{Body}\ \text{"\}"} : \mathit{LexSpec}(l), \mathit{Attributes}(l), \mathit{Rules}(l), \mathit{Methods}(l) \end{array}} \tag{1}
$$

In the case when language l_1 inherits the properties from languages l_2, l_3, then the regular definitions, attributes, rules, which contain production rules and semantic rules, and methods are evaluated according to the operations defined above.

$$
\frac{\begin{array}{l} \mathit{Ancestors} = \mathit{Ancestors}[l_1/\{l_2, l_3\}] \\ l_1, \mathit{LexSpec}, \mathit{Attributes}, \mathit{Rules}, \mathit{Methods} \vdash_{\mathit{Body}} \mathit{Body} : \\ \mathit{LexSpec}(l_1), \mathit{Attributes}(l_1), \mathit{Rules}(l_1), \mathit{Methods}(l_1) \end{array}}{\begin{array}{l} \mathit{LexSpec}, \mathit{Attributes}, \mathit{Rules}, \mathit{Methods} \vdash_{\mathit{LanguageSpec}} \\ \textbf{language}\ l_1\ \textbf{extends}\ l_2,\ l_3\ \text{"\{"}\ \mathit{Body}\ \text{"\}"} : \\ \quad \mathit{LexSpec}(l_1) \cup \mathit{LexSpec}(l_2) \cup \mathit{LexSpec}(l_3), \\ \quad \mathit{Attributes}(l_1) \ominus \mathit{Attributes}(l_2) \ominus \mathit{Attributes}(l_3), \\ \quad \mathit{Rules}(l_1) \oplus \mathit{Rules}(l_2) \oplus \mathit{Rules}(l_3), \\ \quad \mathit{Methods}(l_1) \oplus \mathit{Methods}(l_2) \oplus \mathit{Methods}(l_3) \end{array}} \tag{2}
$$

Since $\mathit{Rules}(l)$ can contain more than one rule, the operation $\mathit{Rules}(l_1) \oplus \ldots \oplus \mathit{Rules}(l_m)$ is defined as:

$$
\mathit{Rules}(l_1) \oplus \ldots \oplus \mathit{Rules}(l_m) = \bigcup_{i=2..m, r \in \mathit{RuleName}} \mathit{Merge}(\mathit{Rules}(l_1)(r), \mathit{Rules}(l_i)(r))
$$

The intuitive meaning of operation $\oplus$ is that it copies all inherited properties except those which are overridden providing that no horizontal overlap occurs. There is a semantic error in the case when a horizontal overlap occurs.

The regular definition consists of the name of the regular definition d and the regular expression $Rexp$. If the regular definition d does not yet exist in regular definitions of the current language, then a new mapping is added, where the regular definition d is mapped into regular expression $Rexp$.

$$\frac{\begin{array}{l} LexSpec(l)(d) = \perp \\ LexSpec(l)' = LexSpec(l)[d/Rexp] \end{array}}{l, LexSpec \vdash_{Rdef} d\ Rexp : LexSpec(l)'} \tag{3}$$

Regular definitions d_1 of the current language l_1 can also inherit the regular definition d_2 specified in the language l_2, provided that the regular definition d_1 is not yet defined in the current language l_1 and the regular definition d_2 is defined in the language l_2, and also that the regular definition d_2 is not already overridden (in other words, d_2 of the language l_2 is accessible from the language l_1). If this holds, the regular definition d_1 inherits the regular expression which is associated with the regular definition d_2. Hence, $d_1 = Rexp|InhRexp$.

$$\frac{\begin{array}{l} LexSpec(l_1)(d_1) = \perp \\ Super = \{l_2\}\ if LexSpec(l_2)(d_2) \neq \perp \wedge d_2 \notin OverriddenId(l_1, l_2) \\ InhRexp = LexSpec(L \in Super)(d_2) \\ LexSpec(l_1)' = LexSpec(l_1)[d_1/(Rexp|InhRexp)] \end{array}}{l_1, LexSpec \vdash_{Rdef} d_1\ \textbf{extends}\ l_2.d_2\ Rexp : LexSpec(l_1)'} \tag{4}$$

Other syntactic forms of extending lexical specifications have the following semantics, where $super(l)$ is ancestor of the language l where the regular definition d exists:

$$\begin{array}{l} l, LexSpec \vdash_{Rdef} \textbf{extends}\ d\ Rexp : LexSpec(l)' \equiv \\ \qquad l, LexSpec \vdash_{Rdef} d\ \textbf{extends}\ super(l).d\ Rexp : LexSpec(l)' \end{array} \tag{5}$$

$$\begin{array}{l} l_1, LexSpec \vdash_{Rdef} \textbf{extends}\ l_2.d\ Rexp : LexSpec(l)' \equiv \\ \qquad l_1, LexSpec \vdash_{Rdef} d\ \textbf{extends}\ l_2.d\ Rexp : LexSpec(l)' \end{array} \tag{6}$$

$$\begin{array}{l} l, LexSpec \vdash_{Rdef} d_1\ \textbf{extends}\ d_2\ Rexp : LexSpec(l)' \equiv \\ \qquad l, LexSpec \vdash_{Rdef} d_1\ \textbf{extends}\ super(l).d_2\ Rexp : LexSpec(l)' \end{array} \tag{7}$$

Regular definitions d_1 of the current language l_1 can also override the regular definition d_2 specified in the language l_2, provided that the regular definition d_1 is not yet defined in the current language l_1 and the regular definition d_2 is defined in the language l_2, and also that the regular definition d_2 is not already overridden. If this holds, the regular definition d_2 is now overridden and hence not accessible. In all other respects, this rule is equivalent to rule 3.

$$\frac{\begin{array}{l} LexSpec(l_1)(d_1) = \perp \\ Super = \{l_2\}\ if LexSpec(l_2)(d_2) \neq \perp \wedge d_2 \notin OverriddenId(l_1, l_2) \\ OverriddenId(l_1, L \in Super) = OverriddenId(l_1, L \in Super) \cup \{d_2\} \\ LexSpec(l_1)' = LexSpec(l_1)[d_1/Rexp] \end{array}}{l_1, LexSpec \vdash_{Rdef} d_1\ \textbf{overrides}\ l_2.d_2\ Rexp : LexSpec(l_1)'} \tag{8}$$

Other syntactic forms of overriding lexical specifications have the following semantics:

$$\begin{array}{l} l, LexSpec \vdash_{Rdef} \textbf{overrides}\ d\ Rexp\ :\ LexSpec(l)' \equiv \\ \qquad l, LexSpec \vdash_{Rdef} d\ \textbf{overrides}\ super(l).d\ Rexp\ :\ LexSpec(l)' \end{array} \tag{9}$$

$$\begin{array}{l} l_1, LexSpec \vdash_{Rdef} \textbf{overrides}\ l_2.d\ Rexp\ :\ LexSpec(l)' \equiv \\ \qquad l_1, LexSpec \vdash_{Rdef} d\ \textbf{overrides}\ l_2.d\ Rexp\ :\ LexSpec(l)' \end{array} \tag{10}$$

$$\begin{array}{l} l, LexSpec \vdash_{Rdef} d_1\ \textbf{overrides}\ d_2\ Rexp\ :\ LexSpec(l)' \equiv \\ \qquad l, LexSpec \vdash_{Rdef} d_1\ \textbf{overrides}\ super(l).d_2\ Rexp\ :\ LexSpec(l)' \end{array} \tag{11}$$

If attribute a is not yet defined in the current language l, then a new mapping $a \mapsto Type$ is added to Attributes.

$$\frac{\begin{array}{l} Attributes(l)(a) = \perp \\ Attributes(l)' = Attributes(l)[a/Type] \end{array}}{l, Attributes \vdash_{Attr} Type\ a\ :\ Attributes(l)'} \tag{12}$$

The next statement adds the new mapping $r \mapsto ProdSem$ to $Rules$, if rule r does not exist already in the language l. $ProdSem$ contains a set of all productions for rule r, and each production has an associated set of semantic rules.

$$\frac{\begin{array}{l} r \notin dom(Rules(l)) \\ l, r, Rules \vdash_{Prods} Prods : ProdSem \\ Rules(l)' = Rules(l)[r/ProdSem] \end{array}}{l, Rules \vdash_{Rule} \textbf{rule}\ r\ "\{"\ Prods\ "\}"\ : Rules(l)'} \tag{13}$$

The rule r_1 of the current language l_1 can also inherit the rule r_2 specified in the language l_2, provided that the rule r_1 is not yet defined in the current language l_1 and the rule r_2 is defined in the language l_2, and also the rule r_2 is not already overridden. If this holds, the rule r_1 inherits productions and semantic rules which are associated with rule r_2. Inherited semantic rules may overlap with current semantic rules, where current semantic rules have a higher priority (asymmetric descendant-driven lookup) than the inherited semantic rules (see operation $Merge$).

$$\frac{\begin{array}{l} r_1 \notin dom(Rules(l_1)) \\ Super = \{l_2\}\ if Rules(l_2)(r_2) \neq \perp \wedge r_2 \notin OverriddenId(l_1, l_2) \\ InhProd = \{(p, R_p) | (p, R_p) \in Rules(L \in Super)(r_2)\} \\ l_1, r_1, Rules \vdash_{Prods} Prods : ProdSem \\ Rules(l_1)' = Rules(l_1)[r_1/Merge(ProdSem, InhProd)] \end{array}}{l_1, Rules \vdash_{Rule} \textbf{rule}\ r_1\ \textbf{extends}\ l_2.r_2"\{"Prods"\}" : Rules(l_1)'} \tag{14}$$

Other syntactic forms of extending rule specifications have the following semantics:

$$\begin{array}{l} l, Rules \vdash_{Rule} \textbf{rule extends}\ r\ "\{"\ Prods\ "\}"\ : Rules(l)' \equiv \\ \qquad l, Rules \vdash_{Rule} \textbf{rule}\ r\ \textbf{extends}\ super(l).r\ "\{"\ Prods\ "\}"\ :\ Rules(l)' \end{array} \tag{15}$$

$$\begin{array}{l} l_1, Rules \vdash_{Rule} \textbf{rule extends}\ l_2.r\ "\{"\ Prods\ "\}"\ :\ Rules(l)' \equiv \\ \qquad l_1, Rules \vdash_{Rule} \textbf{rule}\ \mathrm{r}\ \textbf{extends}\ l_2.r\ "\{"\ Prods\ "\}"\ :\ Rules(l)' \end{array} \tag{16}$$

$$\begin{array}{l} l, Rules \vdash_{Rule} \textbf{rule}\ r_1\ \textbf{extends}\ r_2\ "\{"\ Prods\ "\}"\ :\ Rules(l)' \equiv \\ \qquad l, Rules \vdash_{Rule} \textbf{rule}\ r_1\ \textbf{extends}\ super(l).r_2\ "\{"\ Prods\ "\}"\ :\ Rules(l)' \end{array} \tag{17}$$

The rule r_1 of the current language l_1 can also override the rule r_2 specified in the language l_2, provided that the rule r_1 is not yet defined in the current language l_1 and the rule r_2 is defined in the language l_2, and also the rule r_2 is not already overridden. If this holds, the rule r_2 is now overridden and hence not accessible. In all other respects this rule is equivalent to rule 13.

$$\frac{\begin{array}{l} r_1 \notin dom(Rules(l_1)) \\ Super = \{l_2\} \; if Rules(l_2)(r_2) \neq \perp \wedge r_2 \notin OverriddenId(l_1, l_2) \\ OverriddenId(l_1, L \in Super) = OverriddenId(l_1, L \in Super) \cup \{r_2\} \\ l_1, r_1, Rules \vdash_{Prods} Prods : ProdSem \\ Rules(l)' = Rules(l)[r_1/ProdSem] \end{array}}{l_1, Rules \vdash_{Rule} \textbf{rule } r_1 \textbf{ overrides } l_2.r_2 \text{ "\{" } Prods \text{ "\}" } : Rules(l_1)'} \tag{18}$$

Other syntactic forms of overriding rule specifications have following semantics:

$$\begin{array}{l} l, Rules \vdash_{Rule} \textbf{rule overrides } r \text{ "\{" } Prods \text{ "\}" } : Rules(l)' \equiv \\ \qquad l, Rules \vdash_{Rule} \textbf{rule } r \textbf{ overrides } super(l).r \text{ "\{" } Prods \text{ "\}" } : Rules(l)' \end{array} \tag{19}$$

$$\begin{array}{l} l_1, Rules \vdash_{Rule} \textbf{rule overrides } l_2.r \text{ "\{" } Prods \text{ "\}" } : Rules(l)' \equiv \\ \qquad l_1, Rules \vdash_{Rule} \textbf{rule } r \textbf{ overrides } l_2.r \text{ "\{" } Prods \text{ "\}" } : Rules(l)' \end{array} \tag{20}$$

$$\begin{array}{l} l, Rules \vdash_{Rule} \textbf{rule } r_1 \textbf{ overrides } r_2 \text{"\{"} Prods \text{"\}"} : Rules(l)' \equiv \\ \qquad l, Rules \vdash_{Rule} \textbf{rule } r_1 \textbf{ overrides } super(l).r_2 \text{"\{"} Prods \text{"\}"} : Rules(l)' \end{array} \tag{21}$$

5. Related Work

There has been a lot of research on augmenting ordinary attribute grammars with extensions to overcome deficiencies of attribute grammars such as lack of modularity, extensibility and reusability [9, 10, 11, 12, 13, 14, 15].

Modular attribute grammars MAG [16] are proposed as a solution to attribute pragmatic problems. 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, MAG too specifies the semantic rules for sets of productions rather than for a particular production. 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. On the other hand, MAG has no counterpart to our multiple inheritance approach.

We borrowed the idea of grammar inheritance from [17] where the only property is a production rule, and extended it to multiple attribute grammar inheritance. The difference between the approaches is also in the granularity of modification. In the approach of [17] modification is possible only for the whole production rule, since the name of the property is left hand nonterminal.

In object-oriented attribute grammars [3, 18] 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. 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 the report [19], extensible attribute grammars are used to generate integrated programming systems in an incremental way. In order to perform incremental generation as quickly and as easily as possible, the restricted form of extension is used. For example, nonterminal symbols can not disappear on the right hand of productions upon extensions. At most they can be replaced by extended nonterminals which must contain all attributes of its respective base nonterminal. Therefore, extensible attribute grammars support some form of strict inheritance while our approach supports nonstrict inheritance.

In our opinion the only widely accepted approach with reusability of attribute grammars is the approach presented in [20] 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 [20] an attribute computation can be further independent from symbols used in particular language definitions.

6. Conclusion

When introducing a new concept, the designer has difficulties in integrating it into the language in an easy way. To enable incremental language design we introduce a new object oriented attribute grammar specification language based on the paradigm $\boxed{\text{Attribute Grammar} = \text{Class}}$. In multiple attribute grammar inheritance the properties which can be inherited or overridden are regular definitions, attributes, rules which encapsulate productions and semantic rules, and methods. Therefore, with multiple attribute grammar inheritance we can extend the lexical, syntax and semantic part of language definition. In the paper an example and the formal definition of multiple attribute grammar inheritance is given. The main advantages of the proposed approach are:

- simplicity and clearness of the approach,
- the object concept is simply transposed on the basic objects of attribute grammars at the specification level, and
- incremental language development is enabled.

We have incrementally designed various small programming languages, such as COOL and PLM with multiple attribute grammar inheritance. Our experience with these non-trivial examples shows that multiple inheritance in attribute grammars is useful in managing the complexity, reusability and extensibility of attribute grammars. The benefit of this approach is also that for each language increment a compiler can be generated and the language tested.

Bibliography

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

[2] 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

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

[4] Mernik M., Lenič M., Avdičaušević E., Žumer V. The Template and Multiple Inheritance Approach into Attribute Grammars. *International Conference on Computer Languages*, Chicago, pp. 102 – 110, 1998.

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

[6] Deransart P., Jourdan M. (Eds.) Attribute Grammars and their Applications. *1st Workshop WAGA, Lecture Notes in Computer Science 461*, Springer-Verlag, 1990.

[7] Alblas H., Melichar B. (Eds.) Attribute Grammars, Applications, and Systems. *Lecture Notes in Computer Science 545*, Springer-Verlag, 1991.

[8] Taivalsaari A. On the Notion of Inheritance. *ACM Computer Surveys*, Vol. 28, No. 3, pp. 438 – 479, 1996.

[9] Jourdan M., Parigot D., Julie C., Durin O., LeBellec C. Design, Implementation and Evaluation of the FNC-2 Attribute Grammar System. *ACM Sigplan Notices*, Vol. 25., No. 6., pp. 209 – 222, 1990.

[10] Swierstra S.D., Vogt H.H. Higher order attribute grammars: a merge between functional and object oriented programming. Report RUU-CS-90-12, Utrecht University, 1990.

[11] Farrow R., Marlowe T.J., Yellin D.M. Composable Attribute Grammars: Support for modularity in translator design and implementation. *19th Annual ACM Sigplan - Sigact Symposium on Principles of Programming Languages*, pp. 223 – 234, 1992.

[12] Boyland J., Graham S. Composing tree attributions. *21th Annual ACM Sigplan - Sigact Symposium on Principles of Programming Languages* , pp. 375 – 388. 1994.

[13] Efremidis S.G., Mughal K.A., Soraas L., Reppy J.H. AML: Attribute Grammars in ML. *Nordic Journal of Computing*, March 1997.

[14] Correnson L., Duris E., Parigot D., Roussel G. Generic programming by program composition. *In Workshop on Generic programming*, Marstrand, Sweden, June 1998.

[15] Lämmel R. Functional meta-programs towards reusability in the declarative paradigm. *Ph.D. thesis*, University of Rostock, Department of Computer Science, submitted, 1998.

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

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

[18] Hedin G. An overview of door attribute grammars. *5th International Conference on Compiler Construction (CC'94)*, Lecture Notes in Computer Science 786, Springer-Verlag, pp. 31 – 51, 1994.

[19] Marti R., Murer T. Extensible Attribute Grammars *Institut fur Technische Informatik und Kommunikationsnetze*, ETH Zurich, Report No. 92–6, 1992.

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

[21] Mernik M., Lenič M., Avdičaušević E., Žumer V. A reusable object-oriented approach to formal specifications of programming languages. *L'object*, Vol. 4, No. 3, pp. 273 – 306, 1998.