Automatic Generation of Language-based Tools

Pedro Rangel Henriques, Maria João Varanda Pereira

University of Minho — prh@di.uminho.pt
Polytechnic Institute of Bragança — mjoao@ipb.pt
Portugal

and

Marjan Mernik, Mitja Lenič, Enis Avdičaušević, Viljem Žumer

University of Maribor
Faculty of Electrical Engineering and Computer Science — Slovenia
{marjan.mernik, mitja.lenic, enis, zumer}@uni-mb.si

Abstract
Many tools can be automatically derived from formal language definitions, such as compilers/interpreters, editors, analyzers, visualizers/animators, etc. Some examples of language-based tools generated automatically by the LISA system are described in the paper. In addition the specification of an algorithm animator and program visualizer, Alma, generated from an extended LISA input-grammar is discussed; LISA principles and code are reused in Alma implementation.

1 Introduction
The advantages of formal specification of programming language semantics are well known. First, the meaning of a program is precisely and unambiguously defined; second, it offers a unique possibility for automatic generation of compilers or interpreters. Both these factors contribute to the improvement of programming language design and development. The programming languages that have been designed with one of the various formal methods have a better syntax and semantics,

1 The project was supported by Slovenian and Portugal governments under the contract SLO-P-11/01-04
2 The work of M. João is partially supported by the Portuguese program PRODEP, ação 5.2 da medida 5 – doutoramentos

©2002 Published by Elsevier Science B. V.
HENRIQUES et al

less exceptions and are easier to learn. Moreover, from formal language definitions many other language-based tools can be automatically generated, such as: pretty printers, syntax-directed editors, type checkers, dataflow analyzers, partial evaluators, debuggers, profilers, test case generators, visualizers, animators, documentation generators, etc. For a more complete list see [HK00]. In most of these cases the core language definitions have to be augmented with the tool-specific information. In other cases, just a part of formal language definitions is enough for automatic tool generation or implicit information must be extracted from the formal language definition in order to automatically generate a tool.

The goal of the paper is twofold. On one hand we discuss some of the tools in this last case, like editors to help in writing sentences of the language and various inspectors (such as automata visualizers, syntax tree visualizers, semantic evaluator animators) that are helpful for a better understanding of the language analysis process. Those examples have all been incorporated in the compiler generator system LISA [MLAŽ00]. On the other hand we present some extensions to the language definitions in a manner to make automatic generation of an algorithm animator and program visualizer possible.

Program visualizers/animators are very useful tools for deeper and clearer understanding of algorithms. As such, they are very valuable for programmers and students. Currently, algorithm animators and program visualizers are strongly language and algorithm-oriented, and they are not developed in a systematic or automatic way. In this paper we aim to show that animators could be also automatically generated from extended language definitions. We will briefly propose a specific solution for the development of such a tool, the Alma system, discussing its architecture and its implementation. The system has a front-end specific for each language and a generic back-end, and uses a decorated abstract syntax tree (DAST) as the intermediate representation between them. In the implementation of the Alma system the language development system LISA is used twice. It generates the front-end for each new language, and some parts of it (Java classes) are reused to build the back-end.

The organization of the paper is as follows. In section 2 related work is described. Language-based tools that are automatically generated by the LISA system are described in section 3. The design and implementation of the Alma system are described in section 4. A synthesis and concluding remarks are presented in section 5.

2 Related Work

The development of the first compilers in the late fifties without adequate tools was a very complicated and time consuming task. For instance, the implementation of the compiler for the programming language FORTRAN took about 18 human years. Later on, formal methods, such as operational semantics, attribute grammars, denotational semantics, action semantics, algebraic semantics, and abstract state machines, were developed. They made the implementation of programming
languages easier and finally contributed to the automatic generation of compilers/interpreters. Many tools have been built in the past years, based on the different formal methods and processing different parts of language specification, such as: scanner generators, parser generators and compiler generators. The automatic generation of a complete compiler was the primary goal of such systems. However, researchers soon recognized the possibility that many other language-based tools could be generated from formal language specifications. Therefore, many tools not only automatically generate a compiler but also complete language-based environments. Such automatically generated language-based environments include editors, type checkers, debuggers, various analyzers, etc. For example, the FNC-2 [JP97] is an attribute grammar system that generates a scanner/parser, an incremental attribute evaluator, a pretty printer, a dependency graph visualizer, etc. The CENTAUR system [BCD+89] is a generic interactive environment which produces a language specific environment from formal specifications written in Natural Semantics, a kind of operational semantics. The generated environment includes a scanner/parser, a pretty printer, a syntax-directed editor, a type checker, an interpreter and other graphic tools. The SmartTools system [ACD+01], a successor of the CENTAUR system, is a development environment generator that provides a compiler/interpreter, a structured editor and other XML related tools. The ASF+SDF environment [vdBvDH+01] generates a scanner/parser, a pretty printer, a syntax-directed editor, a type checker, an interpreter, a debugger, etc, from algebraic specifications. In the Gem-Mex system [AKP97] the formal language is specified with abstract state machines. The generated environment includes a scanner/parser, a type checker, an interpreter, a debugger, etc. The LRC system [SK98] from high-order attribute grammar specifications generates incremental scanner/parser and attribute evaluators, syntax-directed editor, multiple views of the abstract semantic tree (unparsing windows), windows-based interfaces, etc. From the above description of various well known compiler/interpreter generators can be noticed that editors, pretty printers, and type checkers are almost standard tools in such automatically generated environments. To our knowledge none of the existing compiler generators automatically generate visualizers and animators for programs written in a newly specified language.

Searching for tools that produce some kind of animation in order to explain the semantics underlying a given program (to help the programmers reasoning about it), we found several interesting animators and visualization systems—for instance: BALSA [BS84]; TANGO [Sta90]; JCAT [BNR97]; ZSTEP [LF95]; JELIOT [HPS+97]; PAVANE [RCWP92] or LENS [MS93]. However, most of the well known animation systems are not general purpose. They just animate a specific algorithm (allowing or not the choice of some configuration parameters) or, if they accept a larger set of algorithms, the programs must be written in a specific language. Usually in that case, the programmer shall use special data types or procedures (visual annotations) on the source code, which means that, the source program must be modified in order to be animated. The animators described in the literature are not constructed automatically from language specifications.
3 Tools from language definitions

LISA is a compiler-compiler, or a system that generates automatically a compiler/interpreter from attribute grammar based language specifications. The syntax and semantics of LISA specifications and its special features (“templates” and “multiple attribute grammar inheritance”) are described in more detail in [MLEŽ00]. The use of LISA in generating compilers for real programming languages (e.g. PLM, AspectCOOL and COOL, SODL) are reported in [MLAŽ98], [ALMŽ01], [MNA+01].

To illustrate LISA style, the specification of a toy language—Simple Expression Language with Assignments, SELA—is given below. From these descriptions LISA automatically generates a SELA compiler/interpreter.

```
language SELA {

  lexicon {
    Number [0-9]+ 
    Identifier [a-z]+ 
    Operator \+ | := 
    ignore \[0x09\0x0A\0x0D\ \]+ 
  }

  attributes Hashtable *.inEnv, *.outEnv;
  int *.val;

  rule Start {
    START ::= STMTS compute
    { START.inEnv = new Hashtable();
      START.outEnv = STMTS.outEnv; }
  }

  rule Statements {
    STMTS ::= STMT STMTS compute
    { STMT.inEnv = STMTS[0].inEnv;
      STMTS[1].inEnv = STMT.outEnv;
      STMTS[0].outEnv = STMTS[1].outEnv; }
    | STMT compute
    { STMT.inEnv = STMTS[0].inEnv;
      STMTS[0].outEnv = STMT.outEnv; }
  }

  rule Statement {
    STMT ::= #Identifier \:= EXPR compute
    { EXPR.inEnv = STMT.inEnv;
      STMT.outEnv = put(STMT.inEnv,
        #Identifier.value(), EXPR.val); }
  }

  rule Expression {
    EXPR ::= EXPR + EXPR compute
    { EXPR[2].inEnv = EXPR[0].inEnv;
      EXPR[1].inEnv = EXPR[0].inEnv;
      EXPR[0].val = EXPR[1].val + EXPR[2].val; }
  }

  rule Term1 {
    EXPR ::= #Number compute
    { EXPR.val = Integer.valueOf(
          #Number.value()).intValue(); }
  }

  rule Term2
}
```
HENRIQUES et al

Fixed and variable parts of LISA generated language-based tools

Besides that, LISA derives other tools. In the following subsections three families of such tools are described: editors to help the final users in the creation and maintenance of the sentences of the specified language, i.e., the source texts (or source programs) that he wants to process (compile/interpret); inspectors that are useful to understand the behaviour or debug the generated language processor itself (compiler/interpreter); and visualizers/animations, similar to inspectors, useful to understand the meaning of the source program that is being processed.

Automatic generation is possible whenever a tool can be built from a fixed part and a variable part; and also the variable part, language dependent, has to be systematically derivable from the language specifications. That part has a well defined internal representation that can be traversed by the algorithms of the fixed part. Table 1 summarizes those parts for some of the language-based tools generated by the LISA system. It is not the aim of this paper to describe all those algorithms, except the most interesting ones for program visualizations and animations. They are described in detail in section 4.

<table>
<thead>
<tr>
<th>Generated tool</th>
<th>Formal specifications</th>
<th>Fixed part</th>
<th>Variable part</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lexer</td>
<td>regular definitions</td>
<td>algorithm which interprets action table</td>
<td>action table: $S_1 \times \Sigma \rightarrow S_1 \times \Sigma$</td>
</tr>
<tr>
<td>Parser (LR)</td>
<td>BNF</td>
<td>algorithm which interprets action table and goto table</td>
<td>action table: $S_1 \times \Sigma \rightarrow A \times \Sigma \times \Gamma$goto table: $S_1 \times \Sigma \rightarrow S_1 \times \Sigma$</td>
</tr>
<tr>
<td>Evaluator</td>
<td>Attribute Grammars (AG)</td>
<td>tree walk algorithm</td>
<td>semantic functions</td>
</tr>
<tr>
<td>Language knowledgeable editor</td>
<td>regular definitions (extracted from AG)</td>
<td>matching algorithm</td>
<td>action table: $S_1 \times \Sigma \rightarrow S_1 \times \Sigma$</td>
</tr>
<tr>
<td>Structure editor</td>
<td>BNF (extracted from AG)</td>
<td>incremental parsing algorithm</td>
<td>same as parser</td>
</tr>
<tr>
<td>FSA visualization</td>
<td>regular definitions (extracted from AG)</td>
<td>FSA layout algorithm</td>
<td>same as lexer</td>
</tr>
<tr>
<td>Syntax tree visualization</td>
<td>BNF (extracted from AG)</td>
<td>syntax tree layout algorithm</td>
<td>syntax tree</td>
</tr>
<tr>
<td>Dependency graph visualization</td>
<td>extracted from AG</td>
<td>DG layout algorithm</td>
<td>dependency graph</td>
</tr>
<tr>
<td>Semantic evaluator animation</td>
<td>extracted from AG</td>
<td>semantic tree layout algorithm</td>
<td>decorated syntax tree &amp; semantic functions</td>
</tr>
<tr>
<td>Program visualization and animation (ALMA)</td>
<td>additional formal specifications</td>
<td>visual and rewrite rules &amp; visualization, rewriting and animation algorithm</td>
<td>decorated abstract tree (DAST)</td>
</tr>
</tbody>
</table>

Table 1

{  
EXPR ::= #Identifier  
{  
  EXPR.val = ((Integer)EXPR.inEnv.get(
    #Identifier.value());
  
  #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;
  }
}
3.1 Editors

Two different LISA generated language oriented editors, that is editors that are sensitive to the language syntax, are described in this section.

3.1.1 Language Knowledgeable Editors

LISA generates a language knowledgeable editor, which is a compromise between text editors and syntax-directed editors, from formal language specifications. The LISA generated language knowledgeable editor is aware of the regular definitions of the language lexicon (see table 1). Therefore, it can color the different parts of a program (comments, operators, reserved words, etc.) to enhance understandability and readability of programs. In Figure 1 operators in SELA programs are recognized while editing and displaying in a different colour.

3.1.2 Syntax-directed Editors

Syntax-directed editors are editors which are aware of the language syntax of edited programs. They help users to write syntactically correct programs before they are actually compiled, exhibiting that structure and/or inserting directly the keywords at the right places (the user only has to fulfill the variable parts of his text). A Structure Editor is a kind of syntax-directed editor, where the syntax structure of written programs are explicitly seen while editing the program (see Figures 2 and 3).

3.2 Inspectors for Language Processors

Three different LISA generated inspectors are introduced in this subsection.

3.2.1 Finite State Automaton Visualization

With the help of visual representation of directed graphs it becomes clear how complex automata can be specified with simple regular expressions and how some simple automata require a complex regular expression (like comments in C). It is also possible to determine the conflicts in specifications and resolve them. In Figure 4 a finite state automaton of the SELA language is presented.
Fig. 2. Structure editor

Fig. 3. Syntax tree view

Fig. 4. FSA view
3.2.2 Syntax Tree Visualization

The basic understanding of a compiler is based on the understanding of the parsing procedure. This tool is a graphical browser for the syntax tree built by the LISA generated compiler after parsing a given source program. With this kind of visualizations it is possible to measure the impact of different grammar specifications and parsing techniques on the shape and size of the tree, and assess their effect on compiler implementation; in that way it is possible to design a better syntax and thereby an easier and understandable semantics.

Figure 3 illustrates the output of this tool: a part of the syntax tree selected by the user while editing a program.

3.2.3 Dependency Graph Visualization

As attribute grammars are specified on the declarative level, the order of attribute evaluation is determined by the compiler construction tool. But that sequence is also important for humans to understand the actual evaluation order, once again to help language designers to improve their grammars.

Even more relevant is the detection of cycles on a grammar; if the attribute dependencies induce indirect cycles, they can be easily discovered with the aid of visual representations.

In Figure 5 an augmented dependency graph, drawn by the LISA generated tool for the 5th SELA production, is presented.

3.2.4 Semantic Evaluator Animation

In attribute grammars a set of attributes carrying semantic information is associated with each nonterminal. For example, attributes \texttt{inEnv} and \texttt{val} are associated with nonterminal \texttt{EXPR} (see SELA language specifications). In the evaluation process the value of these attributes has to be computed. The semantic analysis is better understood by animating the visits to the nodes of the semantic tree, and the evaluation of attributes in these nodes; Figure 6 shows a snap-shot of the animation process. Therefore, the animation of the evaluation process is also very helpful in
the debugging process. Users can also control the execution by single-stepping and setting the breakpoints.

3.3 Program Visualization and Animation

Another instance of tools that can be derived from formal language specification are program visualizers/animators. The purpose of such a family of tools is to help the programmer to inspect the data and control flow of a source program—static view of the algorithms realized by the program (visualization) —and to understand its behaviour—dynamic view of the algorithms’ execution (animation). Such a tool can be obtained by the specialization of a generic visualizer/animator (a language-independent back-end) providing an extension to the LISA attribute grammar that specifies the language to be analyzed. The AG extension just defines the way the input sentence should be converted into the animator’s internal representation (DAST)—see Figure 7.

Below is an example of such an extension for the SELA language, introduced in the previous subsections.

```plaintext
import "AlmaLib.lisa";
language AlmaSELA extends SELA, AlmaBase {
  rule extends Start {
    START ::= STMTS compute
    {
      ALMA_ROOT<START,STMTS>
      ALMA_TAB<START,STMTS>
    }
  }

  rule extends Statements {
    STMTS ::= STMT STMTS compute
    {
      ALMA_STATS<STMTS,STMT,STMTS[1]>
    }
  }
}```
The extension shown above illustrates the use of templates, and multiple attribute grammar inheritance that are both standard LISA features. It is used to specify the attribute evaluation related to the DAST construction; it assumes the syntax and semantics specified in SELA attribute grammar and adds new computing statements just to build the internal representation used by Alma. To write in a clear and concise way those statements, we use templates. Each template is specified as:

```plaintext
template<attributes X_in,Y_in>
compute ALMA_ROOT
{  X_in.dast=new Alma.CRoot(Y_in.tree);
}
```

The method CRoot is one of pre-defined methods inherited from AlmaBase and is used
to construct the DAST nodes. From this specification we generate a parser and a translator that converts each input text into an abstract representation used by the animator, common to all different source languages. That processor, we call it the animator’s front-end, is the language dependent component of the tool. In this case, its fixed part is more complex than in the cases studied in previous subsections 3.1 and 3.2: it is not just a standard algorithm (we use three language independent algorithms), but it requires also two standard data structures (a visual rule base, and a rewriting rule base). So, taking a source program in SELA:

\[
\begin{align*}
  a & := 2 + 2 + 5 \\
  b & := a + 3 + a \\
  c & := a + b + 6
\end{align*}
\]

these algorithms can generate a visualization like the one that can be seen in Figure 8. Take the picture as an example because the final layout (drawings used) can be modified by the Alma designer. Drawing procedures called by the visualizing rules can easily be changed. The system, to be discussed in the next section has a front-end specific for each language and a generic back-end, and uses a decorated abstract syntax tree (DAST) for the intermediate representation between them. A general overview of Alma’s specifications and structures, and its relation to a LISA generated compiler is provided in Figure 7.

4 Alma Implementation

The Alma system was designed to become a new generic tool for program visualization and animation based on the internal representation of the input program in order to avoid any kind of annotation of the source code (with visual types or statements), and to be able to cope with different programming languages.

4.1 Alma Architecture

To comply with the requirements above, we conceived the architecture shown in Figure 9. In Alma we also use a DAST as an internal representation for the meaning of the program we intend to visualize; in that way, we isolate all the source language dependencies in the front-end, while keeping the generic animation engine in the back-end. The DAST
is specified by an abstract grammar independent of the concrete source language. It is not possible to include here that grammar to formalize the type (structure, and attributes) of DAST nodes; for more details we suggest the reading of [VH01]. In some sense we can say that the abstract grammar models a virtual machine. So the DAST is intended to represent the program state in each moment, and not to reflect directly the source language syntax. In this way we rewrite the DAST to describe different program states, simulating its execution; notice that we deal with a semantic transformation process, not only a syntactic rewrite.

A Tree Walk Visualizer, traversing the tree, creates visual representations of nodes, gluing figures in order to get the program image on that moment. Then the DAST is rewritten (to obtain the next internal state), and redrawn, generating a set of images that will constitute the animation of the program.

4.1.1 Visualization in Alma

The visualization is achieved applying visualizing rules (VR) to DAST subtrees; those rules define a mapping between trees and figures. When the partial figures corresponding to the nodes of a given tree are assembled together, we obtain a visual representation for the respective program.

Visualizing Rules

The VRB (Visualizing Rule Base) is a mapping that associates with each attributed tree, defined by a grammar rule (or production), a set of pairs

\[
\text{VRB: DAST} \leftrightarrow \text{set (cond } \times \text{ dp)}
\]

where each pair has a matching condition, \(\text{cond}\), and a procedure, \(\text{dp}\), which defines the tree visual representation. Each \(\text{cond}\) is a predicate, over attribute values associated with tree nodes, that constrains the use of the drawing procedure (\(\text{dp}\)), i.e., \(\text{cond}\) restricts the visualizing rule applicability.

The written form of each visualizing rule is as follows:

\[
\text{vis\_rule(ProdId)} = \langle\text{tree-pattern}\rangle, \\
(\text{condition}), \\
(\text{drawing procedure})
\]
In this template, condition is a boolean expression (by default, evaluates to true) and drawing procedure is a sequence of one or more calls to elementary drawing procedures.

A visualizing rule can be applied to all the trees that are instances of the production ProdId. A tree-pattern is specified using variables to represent each node. At least, each node has the attributes value, name and type that will be used on the rule specification, either to formulate the condition, or to pass to the drawing procedures as parameters.

Notice that, although each VRB associates to a production a set of pairs, its written form, introduced above, only describes one pair, for the sake of simplicity; so it can happen to have more than one rule for the same production. To illustrate the idea suppose that in Alma’s abstract grammar a relational operation, rel_oper, is defined by the 13th production:

\[
p13: \text{rel}\_\text{oper} : \text{exp} \ \text{exp}
\]

where \(\text{exp}\) is defined as:

\[
p14: \text{exp} : \text{CONST}
p15: \text{exp} : \text{VAR}
p16: \text{exp} : \text{oper}
\]

To build a visual representation for that relational operation we need to distinguish two cases: the first occurs when the value of operand expressions is unknown (the value attributes are not yet instantiated); the second occurs when the value of the operands is known (that means that the expressions have been evaluated). So, different semantic cases of production \(p13\) will be represented by different figures, as shown in the example of figure 10 where we assume that the first expression (left operand) is a variable and the second expression (right operand) is a constant: the drawing on the top describes the first case, and the drawing below the second one.

The visualizing rules to specify that mapping are written below.

\[
\text{vis\_rule(p13)} = \langle\text{opr},a,c\rangle,
((a\.value=NULL) \text{ AND } (c\.value=NULL) \text{ AND } (a\.type=VAR) \text{ AND } (c\.type=CONST)),
\{\text{drawRect(a.name)},\text{drawRect()},\text{put(opr.name)},\text{put('?')})
\]

\[
\text{vis\_rule(p13)} = \langle\text{opr},a,c\rangle,
((a\.value\neq NULL) \text{ AND } (c\.value\neq NULL) \text{ AND } (a\.type=VAR) \text{ AND } (c\.type=CONST)),
\{\text{drawRect(a.name,a.value)},\text{drawRect(c.value)},\}
\]

Fig. 10. Visualization of a relational operation
HENRIQUES et al

\texttt{put(opr.name),put(’?’)}

**Visualization Algorithm**

The visualization algorithm traverses the tree applying the visualizing rules to the sub-trees rooted in each node according to a bottom-up approach (post-fix traversal). Using the production identifier of the root node, it obtains the set of possible representations; then a drawing procedure is selected depending on the constraint condition that is true.

The algorithm is presented below.

\begin{verbatim}
visualize(tree) {
    If not(empty(tree))
    then forall t in children(tree) do visualize(t);
    rules <- VRB[prodId(tree)]; found <- false;
    While (not(empty(rules)) and not(found))
        do r <- choice(rules);
        rules <- rules - r;
        found <- match(tree,r)
    If (found) then draw(tree,r); }
\end{verbatim}

4.1.2 Animation in Alma

Each rewriting rule (RR) specifies a state transition in the process of program execution; the results of applying the rule is a new DAST obtained by a semantic (may be also a syntactic) change of a sub-tree. This systematic rewriting of the original DAST is interleaved with a sequence of visualizations producing an animation. A main function synchronizes the rewriting process with the visualization in a parameterized way, allowing for different views of the same source program.

**Rewriting Rules**

The RRB (Rewriting Rule Base) is a mapping that associates a set of tuples with each tree.

\[ \text{RRB: } \text{DAST} \rightarrow \text{set}(\text{cond} \times \text{newtree} \times \text{atribsEval}) \]

where each tuple has a matching condition, cond, a tree, newtree, which defines syntactic transformations, and an attribute evaluation procedure, atribsEval, which defines the changes in the attribute values (semantic modifications).

The written form of each rewriting rule is as follows:

\begin{verbatim}
rule(ProdId)= \langle tree-pattern>,
      (condition),
      \langle NewProdId: newtree>,
      \{attribute evaluation\}
\end{verbatim}

\begin{verbatim}
<tree-pattern> = <root, child_1, ..., child_n>
<newtree> = <root, child_1, ..., child_n>
\end{verbatim}

In this template, condition is a boolean expression (by default, evaluates to true) and attribute evaluation is a set of statements that defines the new attribute values (by default, evaluates to skip).

A rewriting rule can be applied to all the trees that are instances of the production ProdId. A tree-pattern associates variables to nodes in order to be used in the other fields of the rule specification: the matching condition, the new tree and
the attribute evaluation. When a variable appears in both the tree-pattern (we
call the left side of the RR) and the newtree (so called right side of the RR), it
means that all the information contained in that node, including its attributes will
not be modified, i.e. the node is kept in the transformation as it is.
Notice that, although each RRB associates to a production a set of tuples, its writ-
ten form, introduced above, only describes one tuple. So, it can happen to have
more than one rule for the same production. For instance, consider the following
productions, belonging to Alma’s abstract grammar, to define a conditional state-
ment:

\[
\begin{align*}
p8: & \quad \text{IF} : \text{cond} \text{ actions actions} \\
p9: & \quad \mid \text{cond} \text{ actions}
\end{align*}
\]

The DAST will be modified using the following rules:

\[
\begin{align*}
\text{rule}(p8) &= <\text{if}, \text{op}, \text{a}, \text{b}>, \\
& \quad \{ \text{op.value=}\text{true} \}, \\
& \quad \{ \text{p9:if}, \text{op}, \text{a} >, \\
& \quad \{ \}
\end{align*}
\]

\[
\begin{align*}
\text{rule}(p8) &= <\text{if}, \text{op}, \text{a}, \text{b}>, \\
& \quad \{ \text{op.value=}\text{false} \}, \\
& \quad \{ \text{p9:if}, \text{op}, \text{b} >, \\
& \quad \{ \}
\end{align*}
\]

**Rewriting Algorithm**

The rewriting algorithm is also a tree-walker that traverses the tree until a rewrit-
ing rule can be applied, or no more rules match the tree nodes (in that case, the
transformation process stops). For each node, the algorithm determines the set of
possible RR using its production identifier (ProdId) and evaluating the contextual
condition associated with those rules. The DAST will be modified removing the
node that matches the left side of the selected RR and replacing it by the new tree
defined by the right side of that RR. This transformation can be just a semantic
modification (only attribute values change), but it can also be a syntactic modifica-
tion, (some nodes disappear or are replaced).

The rewriting algorithm follows:

\[
\text{DAST rewrite}(\text{tree})\
\begin{align*}
\text{If not}(\emptyset(\text{tree})) & \text{ then rules } \leftarrow \text{RRB[prodId(}\text{tree})\}; \\
\text{found } & \leftarrow \text{false}; \\
\text{While} \ (\text{not}(\emptyset(\text{rules})) \text{ and not}(\text{found})) \\
\text{do r } \leftarrow \text{choice}(\text{rules}); \\
\text{rules } & \leftarrow \text{rules } \text{ - } r; \\
\text{found } & \leftarrow \text{match}(\text{tree}, r) \\
\text{If} \ (\text{found}) \\
\text{then} \text{tree } \leftarrow \text{change}(\text{tree}, r) \\
\text{else} \ a & \leftarrow \text{nextchild}(\text{tree}) \\
\text{While} \ (\text{not}(\emptyset(\text{a})) \text{ and not}(\text{rewritten(a)})) \\
\text{do a } \leftarrow \text{nextchild}(\text{tree}) \\
\text{If not}(\emptyset(\text{a})) \text{ then} \text{tree } \leftarrow \text{rebuild}(\text{tree}, a, \text{rewrite}(a)) \\
\text{return}(\text{tree})
\end{align*}
\]

**Animation Algorithm**

The main function defines the animation process, calling the visualizing and the
rewriting processes repeatedly. The simplest way consists in redrawing the tree af-
After each rewriting, but the sequence of images obtained can be very long and may not be the most interesting. So the grain of the tree redrawing is controlled by a function, called below `shownow()`, that after each tree’s syntactic-semantic transformation decides if it is necessary to visualize it again; the decision is made taking into account the internal state of the animator (that reflects the state of program execution) and the value of user-defined parameters.

The animation algorithm, that is the core of Alma’s back-end, is as follows:

```java
animator(tree){
    visualize(tree);
    Do rewrite(tree);
    If shownow() then visualize(tree);
    until (tree==rewrite(tree))
}
```

When no more rules can be applied, the output and input of the rewrite function are the same.

### 4.2 Reusing LISA in Alma implementation

LISA itself, and the generated compilers, are implemented, in the programming language Java, following an object-oriented approach—see Figure 11 to get a general picture of LISA architecture integrated with all its generated tools. So it was very easy to identify and understand the data structures and functions used by LISA system and tools to process a given attribute grammar specification or a source programs—they are properly encapsulated in classes, as attributes and methods. So the coding of data structures and algorithms needed to implement Alma became straightforward, due to the reuse of some of the referred classes.

A global view of Alma implementation is provided in Figure 12. To build the DAST—that is the output of Alma’s front-end (generated by LISA), and the input of Alma’s back-end—the developer of the AG specification shall call some specific methods (provided in Alma’s standard library) to create a new tree node for each symbol of the DAST abstract language and to collect the trees associated with its children. To implement those methods, we just reused the Java classes `CTreeNode`, `CSyntaxTree`, `CParseSymbol`, used by LISA to create its internal tree representations.

As an immediate consequence, all the facilities provided in the LISA environment
to manipulate those trees became available to process the DAST.

Those classes were reused once again to build the maps VRB and RRB necessary for the implementation of the back-end (visualization and animation) algorithms, as described in subsections 4.1.1 and 4.1.2—remember that every (visualizing or rewriting) rule in both maps is defined in terms of tree-patterns.

The back-end itself is another Java class, specially developed for that purpose, that reuses the data structures and implements directly the algorithms shown in the previous subsections. To code that class, we kept the OO approach followed in LISA development.

To code the main class of Alma, it should contain the necessary methods to call and synchronize the animator’s front-end and back-end functions, we simply reused and adapted the standard Java class, Compile.java, made available in LISA library as the main class for its generated compilers.

5 Conclusion

Many applications today are written in well-understood domains. One trend in programming is to provide software tools designed specifically to handle the development of such domain-specific applications in order to greatly simplify their construction. These tools take a high-level description of the specific task and generate a complete application.

One such well established domain is compiler construction, because there is a long tradition of producing compilers by hand, and because the underlying theory (supporting all the analysis phases, and even code generation and optimization processes) is well understood. At present, there exist many generators which automatically produce compilers or interpreters from programming language specifications.

As shown in this paper, not just standard compilers/interpreters can be generated automatically. Formal language specifications contain a lot of information from which many language-based tools, such as editors, type checkers, debuggers, visu-
alizers, animators, can be generated. Sometimes the implicit information is enough, but in some other cases some extra data must be added. Concrete examples of both types, produced by the generator system LISA, were introduced and discussed along the article. We do not intend to discuss in this paper the actual performance of LISA, because our aim is to enhance the capabilities of the attribute grammar specifications; however our experiments allow us to say that execution time of the generated tools is completely acceptable when compared with similar programs.

The benefits of this approach to software development are many fold. On one hand, the developer just writes formal descriptions that are more concise and clear—they are faster to produce and easier to understand and maintain, because they are shorter; it is well known that thousands of lines of complex code are automatically obtained from a grammar definition written in just some dozens of lines. On the other hand, different tools can be obtained from the same specification, which is obviously a major advantage. Last but not least, a very good and optimal code (developed by experts in language processing methods and algorithms) is reused in the automatic generation process.

Discussing in detail the architecture and implementation of one particular tool, the program visualizer/animator Alma, we also proved that: (1) a grammatical approach to software engineering, supported by generators, is a nice way to develop applications (we systematically created a general animator instead of an algorithm or language dependent one); and (2) a modular, object-oriented, way of programming is valuable for effective reuse of the code (we used some LISA classes in Alma), saving in fact development time and effort.

Some front-end were implemented in order to tune the internal representation to be used by Alma. By now we are working on the back-end classes to implement the visualization and animation algorithms, and we expect some experimental results in the near future.

References


