Experiences in teaching principles of programming languages using a software tool

V. Žumer, M. Mernik, N. Korbar

University of Maribor, Faculty of Electrical Engineering and Computer Science, Smetanova 17, 62000 Maribor, Slovenia

Abstract

In the paper a software tool which supports teaching of the principles of programming languages and our experience with its usage in the educational process is described. Using the tool students have the possibility to experiment, estimate and test various lexical and syntax analysers, and attribute evaluation strategies. Preliminary results shows that students are more motivated and understand subject more deeply.

1 Introduction

The rapid progress in computer and information technology helps us to teach the subject more successfully using new methods and appropriate software tools. This fact was early recognised at our institution. Therefore there exists many software tools, like OOMetTool for program metrics [8], and SURFMOD for surface modelling [9]. All this tools are designed and developed in our laboratories to support educational processes. In the paper our experience with using one of such tool in teaching the principles of programming languages is described. In the third year of the undergraduate courses in computer science we are teaching the subject “Programming Language Principles”. The aim of the subject is to describe various programming language paradigms from the points of view of the programmer and the implementer and to demonstrate the most common methods used to analyse, translate, and execute programs on machines. An brief outline of the subjects that are included in a course of study are:
- Syntax : introduction to formal language theory, lexical analysis, syntax analysis, concrete and abstract syntax ;
Software Engineering in Higher Education

- Semantics: operational semantics, denotational semantics and attribute grammars;
- Concepts: values, types, expressions, commands, storage, bindings, abstractions, encapsulations, overloading, polymorphism, inheritance, exceptions and parallelism;
- Paradigms: imperative, functional, logic and object-oriented.

Syllabus shown above is interesting but for students little bit to extensive for one semester course. One of the possible approach is more motivating the students. Students are often more motivated when using various software tools which additionally explain and strengthen the discussion topic. For reasons stated above we have developed a tool LISA (Language Implementation System Based on Attribute Grammars) for automatic language implementation [7]. Attribute grammars are very suitable tool for compiler generation and several compiler generators have been built based on it, such as GAG, LINGUIST, TOOLS [5]. In LISA (Fig. 1) students have possibility to experiment, estimate and test various lexical and syntax analysers, and attribute evaluation strategies. Through real examples by implementing small languages from different programming paradigms also other discussion subjects are covered.

Fig. 1: Main window in LISA
2 The Tool LISA

LISA is designed and developed using object oriented technology and runs in Windows environment. User friendly multiwindow interface enable us to write formal language description in a quick and simple manner. All formal methods used in LISA are modular and merely independent from each other and therefore written in separate window. The purpose of LISA is to experiment with different formal definitions of programming languages using automatic compiler generation.

In lexical analysis or scanning the stream of characters, making up the source program, is read from left to right and grouped into tokens. Tokens are treated as terminal symbols in the grammar for the source language. The lexems matched by the pattern for the token represents strings of characters in the source program that can be treated together as lexical units. Regular expressions are most usually used formal method for specifying patterns and also LISA is using them. More precisely LISA uses regular definitions where each regular expression is associated with the name. In regular definitions we can use language symbols, previously defined regular expressions and any combination of them with the operation of union (ab), concatenation (ab), Kleene closure (a*), positive iteration (a+), option (a?) and character classes ![a-z]. If defined language contains any meta symbols, such as $!.*?[]()$, they ought to be enclosed by angle brackets. In the Fig.2 the regular definitions for arithmetic expressions is described.

```
OP_LOW <+> | <_.
OP_HIGH <*>|!%.
PARENTHESIS <(|>)>
DIGIT [0-9]
INTEGER DIGIT+
RDEF OP_LOW | OP_HIGH | PARENTHESIS | INTEGER.
```

Fig. 2: Regular definitions for arithmetic expressions

Regular definitions are then transformed into deterministic finite automata (DFA) by algorithm described in [1]. From DFA LISA generates lexical analyser or scanner in C++.

In syntax tokens of source program are grouped into grammatical phrases. The task of syntax analyser or parser is to determine if a string of tokens can be generated by a grammar. The syntax of programming language is usually described by well known BNF notation. In LISA we used standard BNF conventions. Nonterminal symbols are written in capital letters. Other symbols are terminals except meta symbols $![]()$. If defined language contains any meta symbols they ought to be enclosed by angle brackets. Terminal $\epsilon$ is represented with $<>$. Certain tokens can also be augmented by lexical value. In
BNF they are represented as names augmented by hash character '#'. In the Fig. 3 the BNF of the arithmetic expressions are presented.

\[
S ::= E. \\
E ::= T \cdot E. \\
E ::= + T \cdot E \mid T \cdot E \mid \cdot < >. \\
T ::= F \cdot T. \\
T ::= * F \cdot T \mid / F \cdot T \mid \% F \cdot T \mid \cdot < >. \\
F ::= ( E ) \mid \#\text{INTEGER.}
\]

Fig. 3: BNF for arithmetic expressions

When syntax of the sentences is correct we can compute the meaning of sentences or semantics. The meaning of programs in LISA is described with attribute grammars. A context-free grammar of a language defines parse tree for each syntactically correct program of the language. An attribute grammar is based on a context-free grammar and associates attributes with the nodes of a parse tree. Attribute evaluation rules are associated with the context-free productions of an attribute. Attributes in the node can be of two kinds: the inherited attributes, whose values are obtained from the siblings and parent of that node in the parse tree, and the synthesized attributes, whose values are obtained from the children of that node in the parse tree. Knuth proposed that the meaning of a program is defined with the values of the synthesized attributes at the root node of the parse tree. Semantic rules set up dependencies between attributes that will be represented by a graph. From the dependency graph, we derive an evaluation order for the semantic rules. LISA can generate two types of evaluators: Kennedy-Warren’s table driven evaluator [4] and Katayama’s evaluator [3]. Whole document containing semantic definitions in LISA is divided into three parts delimited by ‘%%’. In the first part of the document the #include directives and user functions can be given. Declarations of attributes are written in the second part of the document, followed by semantic functions. Semantic functions are written as assignment statements in C/C++. This semantic functions are directly copied into generated evaluator. Each semantic function must be followed by semicolon as in C syntax. Group of semantic functions for one grammar production is also marked by an extra semicolon. These groups must follow the order specified with the productions in grammar. In the Fig. 4 the semantic functions for arithmetic expressions are presented.

// user defined functions
#include "stdio.h"
int String2Int(CString str)
{
    int i;
    sscanf(str, "%d", &i);
}
return i;
}

// definition of attributes
int val; // synthesized attribute
int pod; // inhereted attribute

// semantic functions
S0.val = E0.val;

EE0.pod = T0.val;
E0.val = EE0.val;

EE1.pod = EE0.pod + T0.val;
EE0.val = EE1.val;

EE1.pod = EE0.pod - T0.val;
EE0.val = EE1.val;

EE0.val = EE0.pod;

TT0.pod = F0.val;
T0.val = TT0.val;

TT1.pod = TT0.pod * F0.val;
TT0.val = TT1.val;

TT1.pod = TT0.pod / F0.val;
TT0.val = TT1.val;

TT1.pod = TT0.pod % F0.val;
TT0.val = TT1.val;

TT0.val = TT0.pod;

F0.val = E0.val;

F0.val = String2Int(pParseTree->GetToken(l));;

Fig. 4: Semantic functions for arithmetic expressions

LISA generates from formal language definition an evaluator in C++ source code. It accepts input programs of the defined language and returns their meanings – values of synthesized attributes of the root node. Generated
evaluators have also the possibility to show values of all attributes, this becoming very important in debugging. In the Fig. 5 decorated parse tree for sample of arithmetic expression is presented. Synthesized attribute in the root node determines the meaning of expression.

![Fig. 5. Decorated parse tree for arithmetic expression 4-3-2](attachment:tree.png)

3 LISA’s impact on educational processes

LISA covered much of the subject “Principles of Programming Languages”. In the lexical part of the tool students learned about regular expressions, finite state automata and various possibility of their implementation like: table-driven scanners and scanner with goto’s. In the syntax part of the tool the Backus-Naur Form (BNF) and various LL(k) and LR(k) parsers are introduced. Finally, in the semantic part of the tool students learned about attribute grammars and various attribute evaluation strategies, like Kennedy-Warren’s treewalk evaluator and Katayama’s evaluator. With the usage of the tool students have implemented various small languages like LISP Kit [2], ASPLE [6], PLM [10], and C subset in a few weeks. In table 1 some characteristics of formal specification of implemented languages are presented.
Students are not just using the tool, but also learned some actual algorithms. Most prospective students are even involved in implementation of some new algorithms. LISA is still in development phase and this year we are using it for the first time. Therefore, students have also many suggestions how to improve the tool. This is very important feedback for ourselves. Preliminary results shows that students using LISA are more motivated and understand subject more deeply. An important fact is also recognised by students, that formal theory could be useful and practice nicely fit the theory. On the other hand the subject is also more interesting.

4 Conclusion

This paper the tool LISA, our experience using it in educational process, and its impact on this process is represented. From formal language definition LISA produces interpreter or compiler for defined language written in object oriented C++ language. Therefore, students noticed formal programming language definitions, and possible implementation in practice. Using the LISA students are able to implement various small programming languages in a few weeks. We strongly believe that without a software tool the discussion topics are much harder to understand and treat.

References


