An Educational Tool for Teaching Compiler Construction

Marjan Mernik, *Member, IEEE,* and Viljem Žumer, *Member, IEEE*

***Abstract*—Compiler construction is a well-developed discipline since there is a long tradition of producing compilers supported by practical underlying theory and a large selection of textbooks. In the compiler construction course, students learn how to write a compiler by hand and how to generate a compiler using tools like lex and yacc. However, these tools usually have little or no didactical value. In this paper, the software tool LISA is described. It facilitates learning and conceptual understanding of compiler construction in an efficient, direct, and long-lasting way. The authors' experience in using the tool shows the following didactical benefits: support for constructive learning, stimulation of exploratory and active learning, support for different learning styles and learning speed, increased motivation for learning, and better understanding of concepts.**

***Index Terms*—Compiler, compiler generator, computer-based educational environment, exploratory and active learning.**

I. Introduction

EDUCATORS are continuously challenged to teach students better and better. In recent years, education has moved from teacher-centered learning to student-centered learning [1], which can be characterized as a problem-oriented approach. The outcome of much research [2], [3] is that students learn better when they are engaged in activities to solve the problem. In this way, by active exploration and knowledge construction, students learn better than when reading textbooks and attending lectures. However, there is no best single approach to teaching and learning that can be applicable to a wide range of topics and different individuals. The need is to carefully combine traditional instruction learning approaches, such as lectures, textbooks, drill-and-practice, etc., with new learning approaches, such as constructivism [2], scaffolding [1], collaborative learning, just-in-time teaching, etc. On the other hand, the rapid progress in computer technology can help instructors to teach more successfully using new methods and appropriate software tools and environments. Computers, if used properly [4], are important in many educational approaches: computer scaffolding [5], CSCL (computer-supported collaborative learning) [6], CSILE (computer-supported intentional learning environments), CiC (computer-integrated classroom), computer-based educational environments, etc. Students need environments that facilitate learning and conceptual understanding of the underlying principles. The same applies to computer-based teaching of engineering disciplines, for example, computer science. Computer programs, like many other dynamic and abstract processes, are often best understood by observing graphical simulations of their behavior. The CoLoS project [7], [8] was the first to show that mathematical models describe only selected observational facts. The CoLoS project promotes simulation tools which support visualization and animation. The acquisition of knowledge through experimentation with simulated environments facilitates learning and conceptual understanding of the underlying principles in an efficient, direct, and long lasting-way. In this paper, the authors' experience with using one of the environments for teaching compiler construction is described. In the third year of undergraduate courses in computer science they are teaching the "Compiler Construction" course. The objective of the course is to find solutions to problems which are typically encountered when analyzing, translating, and executing programs on machines. After the course, students are able to develop compilers for small programming languages. The subjects included in the course of study are lexical analysis, syntax analysis, semantic analysis, intermediate code generation, optimization, and object code generation. Compiler construction is a well-developed discipline since there is a long tradition of producing compilers supported by underlying automata theories, especially finite state and pushdown automata, which all represent an important part of computer science. There is a large selection of good textbooks [9], [10] and course materials on the web [11]. The Compiler Construction course is often mentioned as one of the few courses where students can complete the whole project [12]. In this course, students have to learn how to write a compiler by hand and how to generate a compiler from high-level specifications, using tools such as lex and yacc [9]. Many tools have been built in the past years, such as scanner generators, parser generators, and compiler generators. However, these tools usually have little or no didactical value. They were not designed for educational purposes, but rather for experienced compiler writers where efficiency, space optimizations, modularity, and portability of generated evaluators were primary concerns. Moreover, none of the currently available tools support incremental language development [13]; therefore, the language designer has to design new languages from scratch or by scavenging old specifications. For reasons stated above, the authors developed the tool LISA (language implementation system based on attribute grammars). In LISA, students have the possibility to experiment, estimate, and test various lexical and syntax analyzers, and attribute evaluation strategies. LISA is an integrated development environment in which users can

Manuscript received October 22, 2001; revised February 18, 2002.

The authors are with the University of Maribor, Faculty of Electrical Engineering and Computer Science, Institute of Computer Science, Maribor 2000 Slovenia (e-mail: marjan.mernik@uni-mb.si; zumer@uni-mb.si).

Digital Object Identifier 10.1109/TE.2002.808277

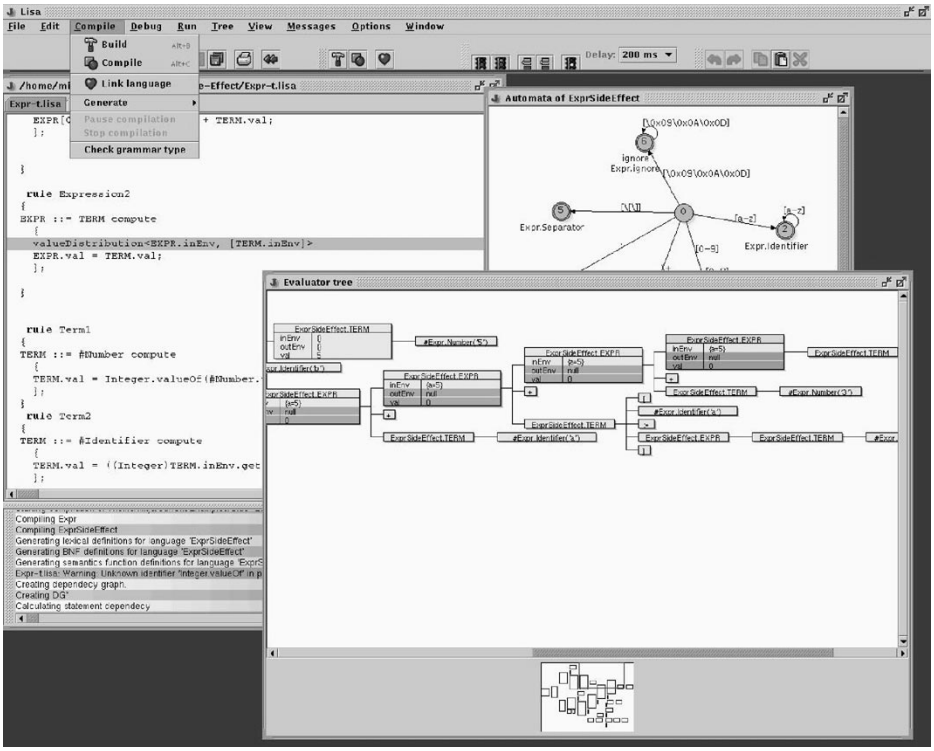

Fig. 1. LISA integrated development environment (IDE). In this figure, the semantic function currently under execution is highlighted in the specification source file. The following program $a := 5\ b := a + 3 + [a := 8] + a$ of the language ExprSideEffect [13] is evaluated.

specify, generate, compile-on-the-fly, and execute programs in a newly specified language (Fig. 1). The compiler/interpreter generated from LISA is visualized in a manner similar to [14], where an illustrated compiler for a simple block structured language PL/0 was implemented. The illustrated compiler in [14] was handwritten, and many graphical views were static; for example, the finite state machine and the syntax diagram did not change, since the user could not change the specification of the language, as is possible in this case.

II. The Tool LISA

Constructivists assert that learners construct knowledge by using mental models. Learning occurs in a context where students actively engage in designing experiments, making observations, and constructing, communicating, and debating explanations. One successful approach for creating such a context is using computer-based learning environments. The LISA tool is an environment that facilitates learning and conceptual understanding of compiler construction. LISA produces an interpreter or a compiler for a defined language written in the object-oriented Java language from a formal language specification. The structure of this specification is described in more detail in [13]. An example of LISA specification of a simple language with assignment statements is given in Fig. 2.

When observing LISA execution, the viewer can see the process of compilation and gain an intuitive understanding of compiler execution. For each phase (lexical, syntax, and semantic) appropriate animation is designed to enhance the viewer's cognitive model.

The LISA tool is freely available for educational institutions from http://marcel.uni-mb.si/lisa. It is run on different platforms and requires Java 2 SDK (software development kits & runtimes), version 1.2.2 or higher.

A. Lexical Analysis

In lexical analysis or scanning, the stream of characters representing the source program is read from left to right and grouped into tokens. The lexems matched by the pattern for the token represent strings of characters in the source program that can be treated together as a lexical unit. Regular expressions, which are the most frequently used formal method for specifying patterns are also used in LISA. More precisely, LISA uses regular definitions where each regular expression is associated with a name (see the "lexicon" part in the Fig. 2). Regular definitions are then transformed into deterministic finite state automata (DFA). LISA generates a lexical analyzer or a scanner in Java from DFA. The lexical analysis is best understood by animating deterministic finite-state automata (Fig. 3), where there is, at most, one transition from the state on the same input character.

B. Syntax Analysis

In syntax analysis, tokens of a source program are grouped into grammatical phrases. The task of the syntax analyzer or parser is to determine if a string of tokens can be generated by a grammar phrase. The syntax of the programming language is usually described by the well-known BNF (Backus–Naur Form) notation. In LISA, standard BNF conventions are used; context-free productions are specified in the rule part of language

```
language Expr {
   lexicon  {
              Number      [0-9]+
              Identifier  [a-z]+
              Operator    \= | :=
              ignore       [\0x09\0x0A\0x0D\ ]+
   }
   attributes Hashtable *.inEnv, *.outEnv; int *.val;
   rule Start {
       START ::= STMTS compute {
         STMTS.inEnv = new Hashtable(); // attributes inEnv and outEnv
         START.outEnv = STMTS.outEnv;   // represent the symbol table
       };
   }
   rule Statements {
       STMTS ::= STMT STMTS compute {
         STMT.inEnv = STMTS[0].inEnv;  // propagation of the symbol table
               STMTS[1].inEnv = STMT.outEnv; // through statements
         STMTS[0].outEnv = STMTS[1].outEnv;
       }
       | compute {  // STMTS ::= epsilon
         STMTS.outEnv = STMTS.inEnv;
       };
   }
   rule Statement {
       STMT ::= #Identifier \:= EXPR compute {
         EXPR.inEnv = STMT.inEnv;
         // put new pair into the symbol table
         STMT.outEnv = put(STMT.inEnv, #Identifier.value(), EXPR.val);
       };
   }
   rule Expression1 {
       EXPR ::= EXPR + TERM compute  {
         TERM.inEnv = EXPR[0].inEnv;     // distribution of the symbol table
         EXPR[1].inEnv = EXPR[0].inEnv;  // to sub-expressions
         EXPR[0].val = EXPR[1].val + TERM.val;  // computing the value of
expression
       };
   }
   rule Expression2 {
       EXPR ::= TERM compute {
         TERM.inEnv = EXPR.inEnv;
         EXPR.val = TERM.val;
       };
   }
   rule Term1 {
       TERM ::= #Number compute {
         TERM.val = Integer.valueOf(#Number.value()).intValue();
       };
   }
   rule Term2 {
       TERM ::= #Identifier compute  {
         // get a value of identifier from the symbol table
         TERM.val = ((Integer)TERM.inEnv.get(#Identifier.value())).intValue();
       };
   }
}
```

Fig. 2. The LISA specification of a simple language with assignment statements. An example of a program is $a := 5\ b := a + 3 + a$ with the meaning $\{(a,5),(b,13)\}$.

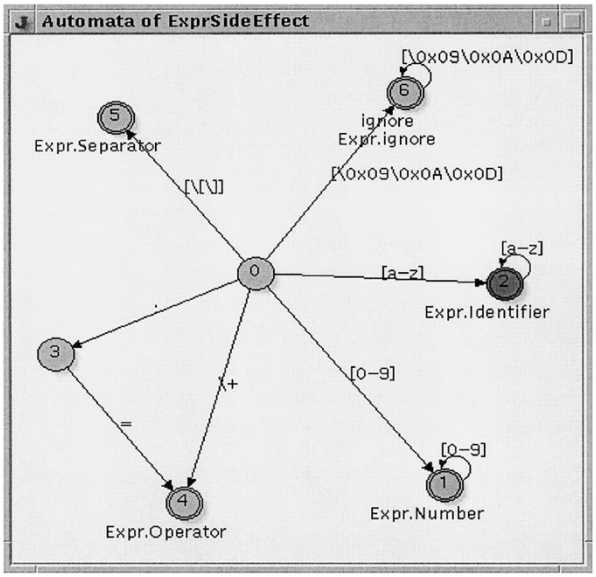

Fig. 3. Finite state automata animation displays the process of scanning. Currently, in the stream of characters, the token identifier is recognized. If there is no transition on the input character in the current state, a lexical error occurs, and an error message is written.

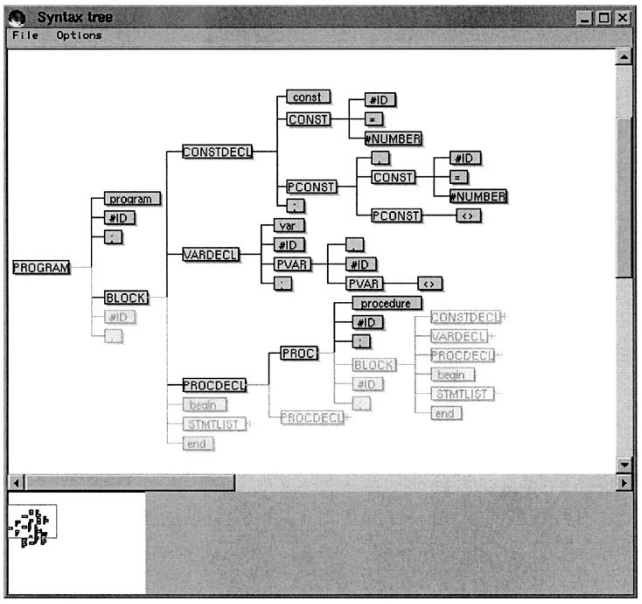

Fig. 4. The animation of the syntax analyzer shows the construction of the syntax tree. In this figure, the construction is done in a top-down manner.

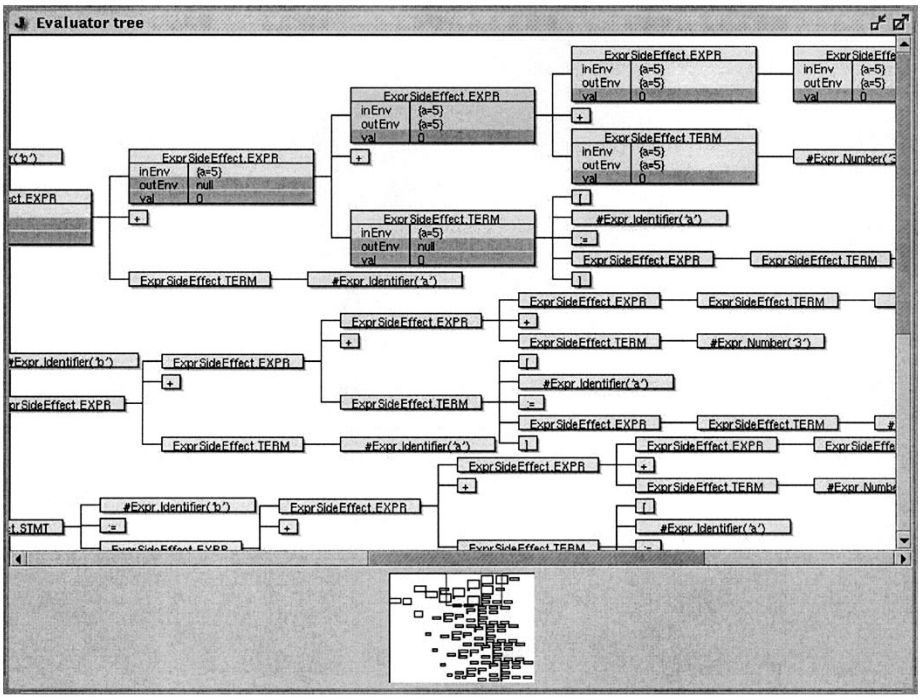

Fig. 5. In attribute grammars, a set of attributes carrying semantic information is associated with each nonterminal. For example, attributes inEnv, outEnv, and val are associated with nonterminal EXPR. In the evaluation process these attributes have to be evaluated. Therefore, the animation of the evaluation process is also very helpful in the debugging process. Users can also control the execution by single-stepping and setting the breakpoints.

definition (Fig. 2). A context-free grammar of a language defines the syntax tree for each syntactically correct program of the language. The syntax analysis is best understood by animating the construction of the syntax tree (Fig. 4).

C. Semantic Analysis

When the syntax of sentences is correct, the meaning of sentences or semantics can be computed. The meaning of programs in LISA is described with attribute grammars [15]–[17]. An attribute grammar is based on a context-free grammar and associates attributes with the nodes of a parse tree, thus obtaining an attributed or semantic tree. Attribute evaluation rules are associated with the context-free productions (see "compute" part in the Fig. 2). Attributes in the node can be of two kinds: the inherited attributes, whose values are obtained from the siblings and the 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. Semantic rules set up the dependencies between attributes that will be represented by a graph. The evaluation order for semantic rules is derived from the dependency graph. The semantic analysis is best understood by animating the node visits of the semantic tree and by the evaluation of attributes in the semantic tree (Fig. 5).

III. LISA's Impact on the Educational Processes

LISA covered much of the compiler construction course. In the lexical part of the tool, students learn about regular expressions, finite state automata, and various possibilities of their implementation. In the syntax part of the tool, the BNF and various LL(k) and LR(k) parsers are introduced. Finally, in the semantic part of the tool, students learn about attribute grammars and various attribute evaluation strategies. In this course, LISA was used in the following manner. In the beginning, the complete specification of a small language was given to the students. Students were asked to observe the animations of the finite state automata, parse and semantic tree, and to understand the semantic functions. In a later case, students had to slightly change the semantic functions and/or find small errors in specifications. In this manner, LISA provided a context in which students were actively engaged in designing experiments, making observations, constructing, communicating and debating explanations. In this step, it is important that students enhance their cognitive model of compiler workings. Later, students were asked to extend the specified language with new features. Finally, students were able to write the specification themselves for a small programming language. A very simple language for moving a robot can illustrate the approach. A robot can move in four directions. After moving, it is stopped in an unknown location, which the user wants to compute. The following specification without comments was given to our students (Fig. 6).

In the process of understanding the semantic functions, students were asked, for example, to change the initial position of the robot, and/or change the length of a particular movement. Moreover, errors were introduced into the specification, and students were asked to find them. An example of such erroneous specification is the following rule, where the coordinates are not properly propagated to the next move (Fig. 7).

Further, students were asked to extend the specified language with new features. For example, they were asked when the robot could reach the final position. One of the possible solutions is presented in Fig. 8.

Another example of adding a new language feature is a possibility that a robot can move with a different speed (Fig. 9).

```
language Robot {
    lexicon {
       ReservedWord left | right | up | down | begin | end
       ignore [\0x0D\0x0A\ ]
    }
    attributes int *.inx; int *.iny;
               int *.outx; int *.outy;
    rule start {
       START ::= begin COMMANDS end compute {
           START.outx = COMMANDS.outx;
           START.outy = COMMANDS.outy;
           COMMANDS.inx = 0;  // robot position in the beginning
           COMMANDS.iny = 0;
       };

    }
    rule commands {
       COMMANDS ::= COMMAND COMMANDS compute {
           COMMANDS.outx = COMMANDS[1].outx; // propagation of coordinates
           COMMANDS.outy = COMMANDS[1].outy; // to sub-commands
           COMMAND.inx = COMMANDS.inx;
           COMMAND.iny = COMMANDS.iny;
           COMMANDS[1].inx = COMMAND.outx;
           COMMANDS[1].iny = COMMAND.outy;
       }
       | epsilon compute {
           COMMANDS.outx = COMMANDS.inx;
           COMMANDS.outy = COMMANDS.iny;
       };
    }
    rule command {
       // each command changes one coordinate
       COMMAND ::= left compute {
           COMMAND.outx = COMMAND.inx-1;
           COMMAND.outy = COMMAND.iny;
       };
       COMMAND ::= right compute {
           COMMAND.outx = COMMAND.inx+1;
           COMMAND.outy = COMMAND.iny;
       };
       COMMAND ::= up compute {
           COMMAND.outx = COMMAND.inx;
           COMMAND.outy = COMMAND.iny+1;
       };
       COMMAND ::= down compute {
           COMMAND.outx = COMMAND.inx;
           COMMAND.outy = COMMAND.iny-1;
       };
    }
}
```

Fig. 6. The LISA specification of a simple language for moving a robot. An example of the program is $begin\ up\ right\ up\ right\ right\ down\ end$ with the meaning $\{outx = 3, outy = 1\}$. A robot, after executing the above program, stopped in the position (3,1).

```
rule commands {
   COMMANDS ::= COMMAND COMMANDS compute {
       COMMANDS.outx = COMMAND.outx;
       COMMANDS.outy = COMMAND.outy;
       COMMAND.inx = COMMANDS.inx;
       COMMAND.iny = COMMANDS.iny;
       COMMANDS[1].inx = COMMAND.outx;
       COMMANDS[1].iny = COMMAND.outy;
   }
   | epsilon compute {
       COMMANDS.outx = COMMANDS.inx;
       COMMANDS.outy = COMMANDS.iny;
   };
}
```

Fig. 7. An example of the erroneous specification. After executing, the program $begin\ up\ right\ up\ right\ right\ down\ end$, the calculated final position (0,1) is wrong.

The difference between the interpretation and compilation was also discussed. An example of expression interpretation is given in Fig. 2, while an example of the translation of expressions to reverse Polish notation, ready to be evaluated on a stack, is given in Fig. 10.

LISA was well accepted by our students. Students like the animated visualizations which provide an intuitive understanding of the compiler construction. Moreover, LISA helps students to understand the general organization of a compiler by showing its essential phases. After the course, the students were asked to fill out questionnaires. The results are presented in Table I.

One may argue that the rate (69%) which shows the importance of animation (question 5) is too low compared to [18] where 95% of the students agree that they understood the material better because of the use of interactive animation. Such comparison is very difficult because of the diversity of topics and individuals. However, from question 8 the authors can conclude that 10% of students still do not understand the working of compilers. Compiler construction was not an easy course (question 1) for most students (72%), and for some students (10%) the course is too difficult; even animation did not help them understand the working of compilers. On the other hand, better students can easily understand how compilers work. The remaining 21% of students may belong to this group of students. For them, animation was helpful but not important for understanding compiler workings (question 3). Therefore, it is better to compare the results of [18] with question 2, where very similar results were obtained.

Our experience using the tool LISA shows the following didactic benefits:

- simulations and animations keep students learning and active;

```
language RobotTime extends Robot {

    attributes double *.time;
    rule extends start {
       START ::= begin COMMANDS end compute {
            START.time = COMMANDS.time;
       };
    }
    rule extends commands {
       COMMANDS ::= COMMAND COMMANDS compute {
            // total time is sum of times spent in sub-commands
            COMMANDS[0].time = COMMAND.time + COMMANDS[1].time;
       }
       | epsilon compute {
           COMMANDS.time = 0;

       };
    }
    rule extends command { // each command spent 1 time step
       COMMAND ::= left compute {
           COMMAND.time = 1;
       };
       COMMAND ::= right compute {
           COMMAND.time = 1;
       };
       COMMAND ::= up compute {
           COMMAND.time = 1;
       };
       COMMAND ::= down compute {
           COMMAND.time = 1;
       };
    }
}
```

Fig. 8. LISA specifications support incremental language development by extending previous specifications, a feature which is called a "multiple attribute grammar inheritance" [13]. The meaning of the program $begin\ up\ right\ up\ right\ right\ down\ end$ is $\{outx = 3, outy = 1, time = 6.0\}$. Therefore, the robot stopped in the final position after 6 time steps.

```
language RobotSpeed extends RobotTime {

   lexicon {
       extends ReservedWord speed
       Number [0-9]+
    }

    attributes int *.in_speed, *.out_speed;

    rule extends start {
        compute {
            COMMANDS.in_speed = 1;  // beginning speed
            START.out_speed = COMMANDS.out_speed;
        }
    }

    rule extends commands {
       COMMANDS ::= COMMAND COMMANDS compute {
            COMMAND.in_speed = COMMANDS[0].in_speed;  // speed propagation
            COMMANDS[1].in_speed = COMMAND.out_speed; // to sub-commands
            COMMANDS[0].out_speed = COMMANDS[1].out_speed;
       }
       | epsilon compute {
           COMMANDS.out_speed = COMMANDS.in_speed;

       };
    }

     rule extends command {
       // these commands do not change speed
       COMMAND ::= left compute {
           COMMAND.time = 1.0/COMMAND.in_speed;
           COMMAND.out_speed = COMMAND.in_speed;
       };
       COMMAND ::= right compute {
           COMMAND.time = 1.0/COMMAND.in_speed;
           COMMAND.out_speed = COMMAND.in_speed;
       };
       COMMAND ::= up compute {
           COMMAND.time = 1.0/COMMAND.in_speed;
           COMMAND.out_speed = COMMAND.in_speed;
              };
       COMMAND ::= down compute {
           COMMAND.time = 1.0/COMMAND.in_speed;
           COMMAND.out_speed = COMMAND.in_speed;
       };
    }

    rule speed {
       COMMAND ::= speed #Number compute {
         COMMAND.time = 0; // no time is spent for this command
         COMMAND.out_speed = Integer.valueOf(#Number.value()).intValue();
         COMMAND.outx = COMMAND.inx;  // this command does not change the
coordinates
         COMMAND.outy = COMMAND.iny;
       };
    }
}
```

Fig. 9. The meaning of the program $begin\ up\ speed\ 2\ right\ up\ right\ right\ speed\ 1\ down\ end$ is $\{outx = 3, outy = 1, out\_speed = 1, time = 4.0\}$. As the result, the robot stopped in the final position sooner, after 4 time steps.

```
language ExpCode {
   lexicon {
       Number      [0-9]+
       Operator    \+ | \*
       Separator   \( | \)
       ignore       [\0x09\0x0A\0x0D\ ]+
   }
   attributes String *.code;

   rule Expression1 {
      EXPR ::= EXPR + TERM compute {
          EXPR[0].code = EXPR[1].code+ "\n" + TERM.code + "\n" + "ADD";
      };
   }
   rule Expression2 {
       EXPR ::= TERM compute {
          EXPR.code = TERM.code;
        };
   }
   rule Term1 {
      TERM ::= TERM * FACTOR compute {
          TERM[0].code = TERM[1].code + "\n" + FACTOR.code + "\n" + "MUL";
      };
   }
   rule Term2 {
      TERM ::= FACTOR compute {
         TERM.code = FACTOR.code;
      };
   }
   rule Factor1 {
       FACTOR ::= #Number compute {
          FACTOR.code = "PUSH " + #Number.value();
        };
   }
   rule Factor2 {
       FACTOR ::= ( EXPR ) compute {
          FACTOR.code = EXPR.code;
        };
   }
}
```

Fig. 10. The meaning of the program $10 + 2 * 3$ is $\{code = PUSH10\ PUSH2\ PUSH3\ MUL\ ADD\}$. Hence, LISA can also emit an assembly code, which can later be executed on abstract or real machines. In this case, operands are pushed onto the stack; and operators pop the required number of operands from the stack, do the operation, and push the result onto the stack.

TABLE I
QUESTIONNAIRE RESULTS

Question	Yes	No
1. Is it difficult to understand the inner working of a compiler?	72%	28%
2. Was the tool LISA of any help to a better understanding of compilers?	90%	10%
3. Was the tool LISA important for a better understanding of compilers?	76%	24%
4. Was the visual presentation important for a better understanding of compilers?	76%	24%
5. Was animation important for a better understanding of compilers?	69%	31%
6. Did the working of compilers interest you in the past?	69%	31%
7. Does the working of compilers interest you now?	83%	17%
8. Do you think that you understand compilers now?	90%	10%
9. Do you think that your knowledge will be long lasting?	62%	38%

- support for constructive learning develops students' mental models;
- immediate feedback to the user's actions stimulate exploratory and active learning (students have opportunity to change specifications – lexical, syntax and semantic – and observe the differences in execution/animation and outputs);
- individual learning and training sessions support different learning styles [19] and learning speed (students have opportunity to learn how to specify different parts of programming languages from the library of programming languages);
- increased and sustained motivation are provided for learning (students are much more motivated when using various software tools and environments which further explain and strengthen the discussed topic);
- better understanding of concepts is promoted (students learning compiler construction have difficulties in understanding the concepts and techniques when they are presented in a traditional way; when using LISA, students are able to implement various small programming languages – e.g., Wirth's PLM language [20] – in a few weeks).

It is difficult to provide some explicit measure of the above benefits and usefulness of the tool LISA in the "Compiler Construction" course. One evidence that students actually gain a deeper understanding of compiler construction is that the average grade of this course increased from 8.05, in the previous year's class where tool LISA was not used, to 8.6 in the class where the tool LISA was used. The grading scale used is the following: 6 – sufficient knowledge, 7 – satisfactory knowledge, 8 – good knowledge, 9 – very good knowledge, and 10 – excellent knowledge.

IV. CONCLUSION

In this paper, the tool LISA, experience with its usage in the educational process, and its impact on this process are presented. In LISA, students have the possibility to experiment, estimate, and test various lexical and syntax analyzers, and attribute evaluation strategies. When observing LISA execution, the viewer can see the process of compilation and gain an intuitive understanding of compiler execution. For each phase (lexical, syntax, and semantic) an appropriate animation is designed to enhance the viewer's cognitive model. The authors strongly believe that without a software tool the discussed topics are

much harder to understand and treat. On the other hand, the course is also more interesting. The important fact that formal theory can be useful and that practice nicely fits the theory was also recognized by our students.

ACKNOWLEDGMENT

The authors would like to thank our LISA project members, especially M. Lenič and E. Avdičaušević, for their efforts implementing the LISA system and for their useful discussions.

REFERENCES

[1] D. A. Norman and J. C. Spohrer, "Learner-centered education," *Commun. ACM*, vol. 39, no. 4, pp. 24–27, 1996.
[2] P. Makkonen, "Do WWW-based presentations support better (Constructivistic) learning in the basics of informatics?," in 33rd Hawaii Int. Conf. System Sciences, 2000.
[3] H. Eden, M. Eisenberg, G. Fischer, and A. Repenning, "Making learning a part of life," *Commun. ACM*, vol. 39, no. 4, pp. 40–42, 1996.
[4] N. W. Holmes, "The myth of the educational computer," *Computer*, vol. 32, no. 9, pp. 36–42, 1999.
[5] M. Guzdial, J. Kolodner, C. Hmelo, H. Narayanan, D. Carlson, N. Rappin, J. Hübscher, J. Turns, and W. Newstetter, "Computer support for learning through complex problem solving," *Commun. ACM*, vol. 39, no. 4, pp. 43–45, 1996.
[6] L. Harasim, "A framework for online learning: The virtual-U," *Computer*, vol. 32, no. 9, pp. 44–49, 1999.
[7] H. Hartel, *The Project CoLoS—Conceptual Learning of Science — Objectives, Background, and First Results* Kiel, 1993.
[8] F. Buret, D. Muller, and L. Nicolas, "Computer-aided education for magnetics," *IEEE Trans. Educ.*, vol. 42, pp. 45–49, Jan. 1999.
[9] A. V. Aho, R. Sethi, and J. Ullman, *Compilers, Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.
[10] C. N. Fisher and R. J. LeBlanc, *Crafting a Compiler*: Benjamin-Cummings, 1988.
[11] [Online]. Available: http://www.angelfire.com/ar/CompiladoresUCSE/COMPILERS.html
[12] H. Liu, "Software engineering practice in an undergraduate compiler course," *IEEE Trans. Educ.*, vol. 36, pp. 104–108, Jan. 1993.
[13] M. Mernik, M. Lenič, E. Avdičaušević, and V. Žumer, "Multiple attribute grammar inheritance," *Informatica*, vol. 24, no. 3, pp. 319–328, 2000.
[14] K. Andrews, R. Henry, and W. Yamamoto, "Design and implementation of the UW illustrated compiler," in *Proc. Sigplan'88 Conf. Programming Language Design and Implementation*, pp. 105–114.
[15] D. E. Knuth, "Semantics of contex-free languages," *Math. Syst. Theory*, vol. 2, no. 2, pp. 127–145, 1968.
[16] "Attribute grammars and their applications," in *Proc. 2nd Workshop on WAGA*, D. Parigot and M. Mernik, Eds., 1999.
[17] "Attribute grammars and their applications," in *Proc. 3rd Workshop on WAGA*, D. Parigot and M. Mernik, Eds., 2000.
[18] M. Budhu, "Interactive multimedia web-based courseware with virtual laboratories," in *Proc. IASTED Int. Conf. Computers and Advanced Technol. in Education*, 2000, pp. 19–25.
[19] C. A. Carver, R. A. Howard, and W. D. Lane, "Enhancing student learning through hypermedia courseware and incorporation of student learning styles," *IEEE Trans. Educ.*, vol. 42, pp. 33–38, Jan. 1999.
[20] N. Wirth, *Algorithms + DataStructures = Programs*. Englewood Cliffs: Prentice Hall, 1976.

Marjan Mernik (M'95) received the M.Sc. and Ph.D. degrees in computer science from the University of Maribor, Maribor, Slovenia, in 1994 and 1998, respectively.

He is currently an Assistant Professor at the University of Maribor, Faculty of Electrical Engineering and Computer Science. His research interests include principles, paradigms, design and implementation of programming languages, compilers, and formal methods for programming language description.

Dr. Mernik is a Member of the ACM and EAPLS.

Viljem Žumer (M'77) received the M.Sc. degree in computer science from the University of Ljubljana, Ljubljana, Slovenia, in 1977, and the Ph.D. degree in computer science from the University of Maribor, Maribor, Slovenia, in 1983.

He is currently a Professor at the University of Maribor, Faculty of Electrical Engineering and Computer Science. He is the Founder of the Computer Science Department at the University of Maribor and in charge of many projects. His research interests include computer architecture and programming languages.