Jaime Urquiza-Fuentes
Francisco Manso

A second evaluation of SOTA

Número 2011-03

Serie de Informes Técnicos DLSI1-URJC
ISSN 1988-8074
Departamento de Lenguajes y Sistemas Informáticos I
Universidad Rey Juan Carlos

Índice

A second evaluation of SOTA*

Jaime Urquiza-Fuentes and Francisco Manso

LITE - Laboratory of Information Technologies in Education
Rey Juan Carlos University, Madrid, Spain
jaime.urquiza@urjc.es, f.manso@alumnos.urjc.es

Abstract. This report presents the evaluation of an educational tool focused on the visualization of the symbol table in the context of a compiler course. In a first evaluation we used simulation exercises and tested basic concepts of symbol tables. We detected efficiency improvements, students who used the tool completed the exercises with the same grading and significantly faster than the students who did not use the tool. In addition students' opinion was positive. In a second evaluation we used more active tasks, and tested students' skills on writing parser specifications regarding symbol table management. We have detected significant improvements. Students who used the tool outperformed those who did not use the tool in a 22%.

Keywords: Evaluation, compiler visualization, symbol table

1 Introduction

Visualization has been widely used in computer science education. There are various surveys on using visualization as an aid for computer science education, e.g. [5]. At present there are tools which visualize program execution [10] or algorithms and data structures [2, 16].

There are also tools aimed at visualizing the relevant processes in compilers and language processors. It is a very common practice to divide a compiler construction course in two different parts. One of them involves the lexical and syntactical analysis of source code based on the formal languages theory techniques. Here many tools can be found in the literature as JFlap [13, 14] focused on authomata theory and formal languages, and VAST [1], CUPV [6], APA[1] [7] or GYacc [9] focused on parsing algorithms. The other part includes processes like syntax directed translation, with tools like as JACCIE[8], Lisa [11] or VCOCO [12], code generation and execution with tools like the PIPPIN Machine [3] and symbol table use or type checking for which we have not found any visualization tool aimed at education.

* This is the extended version of a paper presented at the ITiCSE 2011 Conference titled "Improving Compilers Education through Symbol Tables Animations"

[1] this tool has not name, APA is the title acronym of the paper where it is described

We have developed SOTA[2] [4] (SymbOl Table Animation), an educational tool aimed at visualizing, from a high level point of view, the working of a symbol table during the source code analysis. This paper reports on an experience about enhancing compilers teaching using the visualizations produced by this tool.

The rest of the paper is structured as follows. In the next section we describe the tool. In the section three we detail the experience used to evaluate the educational impact of the tool and its results. Finally, in the fourth section we address our conclusions and some future work.

2 Symbol Table Visualization with the Tool

In this section we describe our static and dynamic graphical representations of the symbol table concepts and the tool used to produce them. The main objective is to visualize the actual state of the symbol table, and the operations performed on it during the source code analysis. The tool has been designed for both, classroom and self study sessions.

The interface of the tool, see Fig. 1, is divided into three areas: the program area, the current state area and the messages area. The program area, on the left side of the interface, shows the source code of the program being analyzed. Currently, we work with a modification of the Pascal language called SimplePascal, its description is shown in the help of the tool. The current state area, on the upper right zone of the interface, shows the graphical representation of the current state of the symbol table, and the last operations (identifier insertion, scope creation and successful or failed searches) performed on it. Finally, the messages area, at the bottom right zone of the interface, shows brief textual descriptions of all the operations performed on the symbol table until the current state has been reached.

The students can edit their own programs or use a set of predefined demonstration programs –with a name and a textual description– available via web. Teachers can contribute to these demonstrations with their own programs. Next we describe how the symbol table concepts are visualized, from both points of view static and dynamic.

2.1 Static visualization

When the symbol table structure is made up of different scopes, the most suitable visualization is the tree structure. In this tree, the nodes represent the scopes, and the arcs define the parent-children scope relations. The tree will grow to the right for new procedures and functions, and to the bottom for anonymous scopes. Fig. 2 shows a source code and its corresponding structure: the root scope with three children scopes corresponding to subprograms –Fact, Add and Proc–, and finally an anonymous scope in the subprogram Proc.

In addition to the structure of the symbol table, the tool highlights the last operations performed on the symbol table. On the one hand, there are operations

[2] http://www.escet.urjc.es/~jurquiza/research-iticse.html#sota

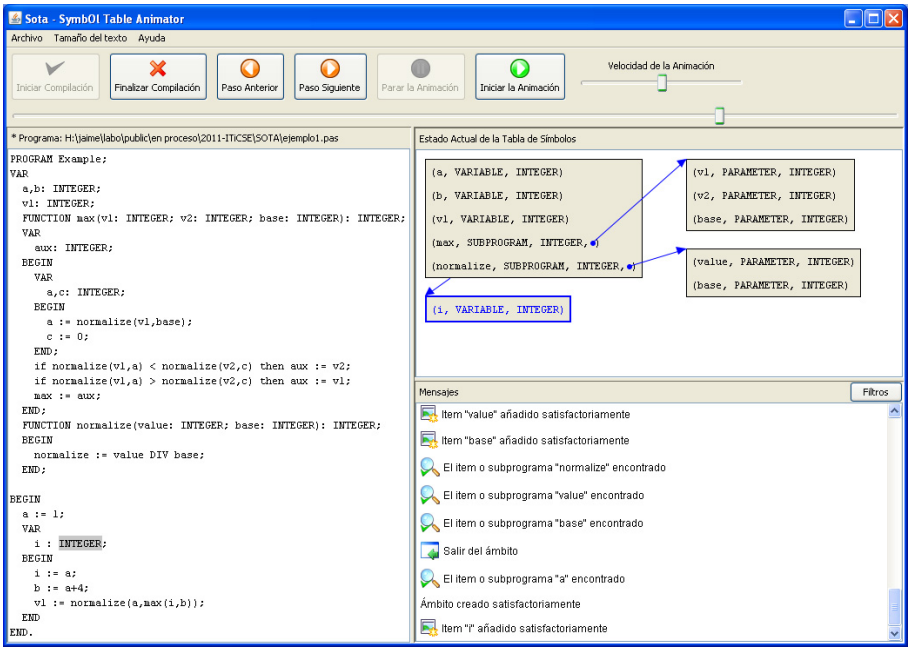


Fig. 1. Graphical user interface of the tool

```
PROGRAM Example2;
VAR
  a,b: INTEGER;
  FUNCTION Fact(n: INTEGER): INTEGER;
  BEGIN
        if(n > 1) then Fact := n*Fact(n-1)
        else Fact := 1;
  END;
  FUNCTION Add(n1: INTEGER; n2: INTEGER): INTEGER;
  BEGIN
        Add := n1 + n2;
  END;
  PROCEDURE Proc(y: INTEGER);
  VAR
    b1, b2, b3: INTEGER;
  BEGIN
        VAR
              b21, b22 : INTEGER;
        BEGIN
              aux := a;
        END;
  END;
BEGIN
  b := Fact(4);
END.
```

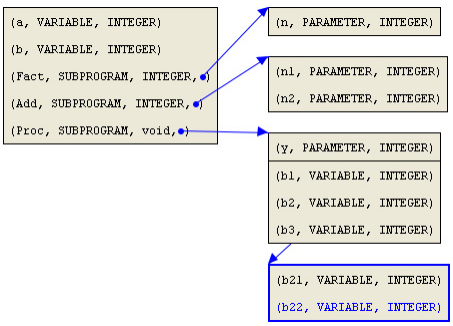


Fig. 2. Source code and the corresponding tree structure of the symbol table

that modify the symbol table structure, thus the last scope created and the last identifier inserted are highlighted. In Fig. 3, the last scope created corresponds to the procedure "Proc", and the last identifier inserted, corresponds to the variable "b3".

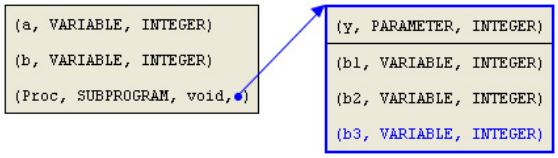


Fig. 3. The last scope created and the last identifier inserted are highlighted in blue color.

On the other hand, search operations and their results are highlighted: the current search scope, and the failed and successful search operations in the scopes. The scope where the compiler is searching the item is highlighted in red. If the search fails, the failed scope is marked with a diagonal line in red. If the search successes, the found item is highlighted in green. The following three figures show the three possibilities. In Fig. 4 the compiler is searching within the anonymous scope. In Fig. 5 a failed search in the anonymous scope is shown, but the search operation continues in the parent scope. Finally, in Fig. 6 is visualized a successful search, finishing in the root environment.

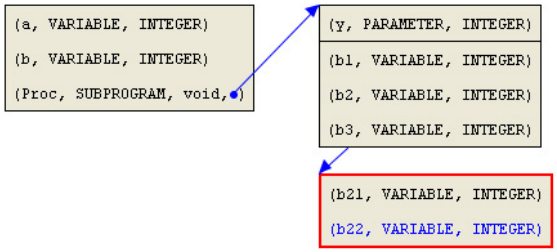


Fig. 4. Entry search in an anonymous environment

2.2 Symbol Table Animation

Animating the symbol table structure consists in a sequence of steps. Each step in the animation corresponds to an action performed on the symbol table during the program compilation. In the case of search operations, each step is mapped to the search operations performed on the different scopes during the search process. The process begin at the current scope and ends when the searched

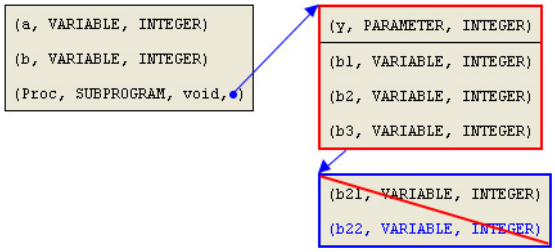


Fig. 5. Failed search in the anonymous scope, the new search scope is the parent scope

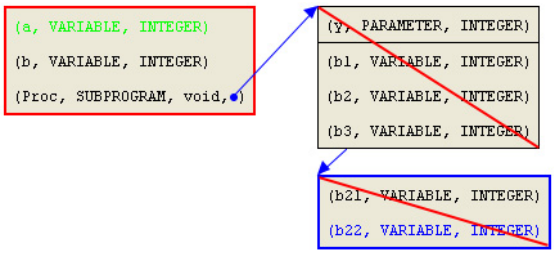


Fig. 6. A successful search in the root scope

item is found, or the root scope is reached. Animations can be controlled with typical VCR controls: begin, end, pause, play and speed selection. In addition, the tool allows to select the immediately previous or next animation step with the buttons "previous step" and "next step", or a concrete execution state, selected with the time bar.

To allow the user to have always accessible information about the performed actions at the moment, the tool provides a messages area. This area shows a message for each operation performed on the symbol table and its result: new scope creation, item insertion and search operations. In addition, the students can choose the kind of messages visible through a filter utility for the messages.

When a message is selected, the corresponding –the moment in which the action was performed– location in the source code is highlighted in yellow. E.g. in Fig. 7 it can be seen how when selecting a successful search message, the corresponding token in the source code is highlighted.

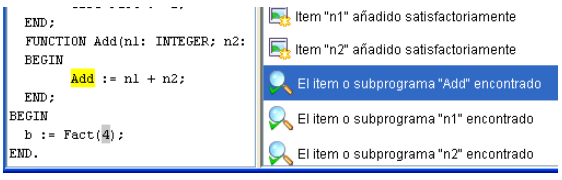


Fig. 7. Highlighting the token in the source code related to the messages

3 Educational evaluation

Visualizations can help students to understand concepts studied, but an evaluation is needed to know their actual educational impact. Next we describe the two evaluations performed with this tool.

3.1 First evaluation

This evaluation [15] was conducted as a controlled experiment with pre-post-test measurements. It was divided in two sessions, a theoretical session where concepts of the symbol table were explained and an exercises/laboratory session. The task performed by the treatment group consisted in mentally simulating how the symbol table structure would be built for a given source code, and assessing it with the tool.

We tested the effectiveness, efficiency and user's opinion about the tool. The effectiveness was measured with a knowledge test where, given a source code, the students had to draw the corresponding symbol table structure and answer questions about scopes and visibility errors. The efficiency was measured in terms of the time used to solve exercises during the exercises/laboratory session and to answer the knowledge test. Finally, students' opinion was collected with a questionnaire regarding ease of use, technical quality, usefulness and the support to the symbol table concepts.

The results of the experiment showed that there was no effectiveness improvement. But the treatment group performed significantly faster than the control group in completing the exercises (63,7%) and the test (32%), so there is efficiency improvement. Finally, the questionnaire about the tool showed that the students considered that the tool was easy to use, that its technical features had good quality, that its visualization features were very useful, and that the representation of the symbol table concepts was helpful. This acceptance of the tool by the students was also supported by actual use of the tool, 75% of the students used the tool to prepare the exam of the subject.

3.2 Second Evaluation

We have performed two changes with respect to the previous evaluation. On the one hand, the difference between student's opinion and pedagogical effectiveness of the previous evaluation leaded us to question the design of the knowledge tests. In the previous evaluation we focused on how the symbol table works, some of these concepts are close to visibility and scope, both seen in most of structured programming courses. In this evaluation we add an exercise about the parser specification dedicated to the symbol table management.

On the other hand, following the Hundhausen et al's [5] conclusions, we focus the tasks on what students do with the tool, rather than what the tool shows to students.

Subjects 57 students enrolled in the evaluation, the participation was voluntary. We divided participants in two groups, the control group (n=34, called CG) and the treatment group (n=23, called TG). Groups formation were independent from the experiment. Students in the CG followed a typical methodology in symbol table teaching, while TG followed a methodology adapted with the tool.

Variables of the evaluation We have used one independent variable, the use of the tool, and one dependent variable, pedagogical effectiveness. The measurement instrument is a knowledge test with questions regarding: the construction process of structure of the symbol table –given a source code the student has to draw the symbol table structure and answer questions about scopes, identifiers and errors, e.g. how many scopes (named and anonymous) have been created during the compiling process? – and the parser specification that builds the symbol table during compilation –given a grammar and the API specification for building the symbol table structure, the student has to insert semantic actions into the grammar using the API to produce the parser specification that builds the symbol table structure.

Tasks and protocol The protocol was divided in four steps, and lasted three weeks, see Fig. 8. At the beginning of the first week all the participants completed the pretest. During the second week each group attended to the theoretical (2 hours long) and lab (1 hours long) sessions. During the third week, again all the participants completed the post test.

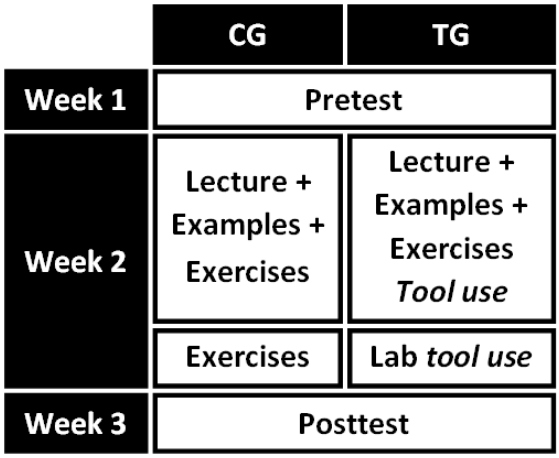


Fig. 8. Protocol of the second evaluation

The CG followed a typical teaching methodology without animations. The theoretical session consisted of teacher's explanations, examples and simple exer-

cises. The tasks proposed in the exercises session were simulation exercises about the construction of the structure of the symbol table of given source codes. Students completed up to four exercises in this session.

The TG followed a teaching methodology adapted to the use of the animations generated by the tool. The theoretical session consisted of teacher's explanations, examples and simple exercises supported by the tool. The tasks proposed in the lab session were two kinds of exercises: simulation and coding. First, the teacher gives a source code to the students. While they mentally simulate the construction process of the corresponding symbol table they use the tool to assess themselves. The second kind of exercise, coding, ask students to reverse their mental process. The teacher provides a schema of a symbol table structure. This schema specifies child and anonymous scopes, and visibility errors. Fig. 9 shows an example of a coding exercise. The students have to write the source code that produces such structure. Again they can use the tool to assess their solution. Students of the TG completed one simulation exercise and up to eight coding exercises.

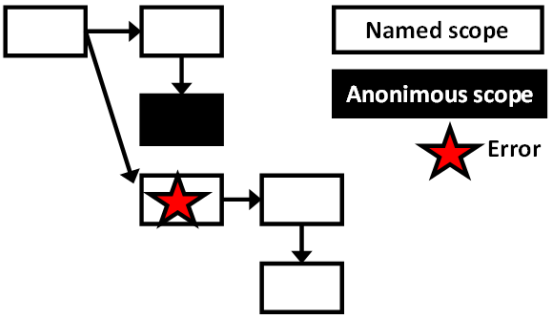


Fig. 9. An example of coding exercise of the second evaluation

Results We have studied students' scores in the post-test. Considering all the questions, we did not find post-test significant differences between both groups. Then we studied both kinds of questions separately. Regarding the construction process of the symbol table structure, we did not find post-test significant differences. But we found significant differences in the question regarding the parser specification ($t(39.84) = -2.8348$, $p < \mathbf{.01}$). Learning improvements of the CG were .1578, while those of the TG were .3817. Since the pre-test scores of both groups in this question were not significantly different ($t(50.588) = -1.8581, p > .05$), TG outperformed CG in more than 22% regarding parser specification for symbol table structure building.

4 Conclusions

We have evaluated the use of symbol table animations in a compiler course. Animations are generated with a specialized software visualization tool, this tool has been designed to help in teaching and learning concepts of the symbol table. In the first evaluation we did not detected effectiveness improvements, but students who used the tool were faster completing exercises and knowledge test, and had a positive opinion about the tool: they believe that the tool helps in classroom and self-study and actually use the tool to prepare the exam.

In the second evaluation we used more active tasks and tested how students designed parser specifications for symbol table management. The tool allowed us to practice coding exercises. Although they could be solved without the tool, actually the amount of exercises would be drastically low, taking into account: the time used by students to think about the problem, write the solution (pen and paper) and the time used by the teacher assessing the different solutions proposed by the students. The students who used the tool produced significantly better parser specifications, they outperformed the students who followed a typical teaching methodology in a 22%.

Note that writing parser specifications is one of the most important objectives of a compiler course. With the tool we have improved students' skills on writing these specifications regarding the symbol table management.

We have designed effective symbol table visualizations and their effective educational use. Our future work consists in the development of an API that generates these visualizations. Thus students can visualize the symbol table management in their own parser specifications.

5 Acknowledgments

This work was supported by project TIN2008-04103/TSI of the Spanish Ministry of Science and Innovation. Also, the authors thank to the former members of the research team, Micael Gallego and Francisco Gortázar, for the time and effort that they dedicated to this project.

A Knowledge Pre-test/Post-test

1. Given the following program written in *SimplePASCAL*:

```
 1: PROGRAM TEST1;
 2: VAR
 3:   var1, var2: INTEGER;
 4:   var3 : INTEGER;
 5:   PROCEDURE procedimiento1(arg1:INTEGER);
 6:   VAR
 7:     auxiliar1, auxiliar2:INTEGER;
 8:   BEGIN
 9:     auxiliar2:= 34*arg1;
10:     IF arg1<2 THEN auxiliar1:=1
11:       ELSE
12:         BEGIN
13:           auxiliar1:= 2;
14:           auxiliar2:= auxiliar2+5;
15:         END;
16:     VAR
17:       auxiliar1:INTEGER;
18:     BEGIN
19:       auxiliar2 := auxiliar1 * 3;
20:     END;
21:     auxiliar1 := auxiliar1 + auxiliar2 * arg1;
22:   END;
23:   FUNCTION funcion(arg1: INTEGER; arg2: INTEGER; arg3: INTEGER):INTEGER;
24:   VAR
25:     auxiliar1 : INTEGER;
26:     PROCEDURE proc1(arg1: INTEGER; arg2: INTEGER);
27:     BEGIN
28:       arg1 := 1;
29:       arg2 := 3;
30:       auxiliar1 := arg1 + arg2 * auxiliar2;
31:       procedimiento1(auxiliar1);
32:     END;
33:   BEGIN
34:     auxiliar1 := arg1 * arg2;
35:      proc1(auxiliar1,arg3);
36:     funcion := auxiliar1;
37:   END;
38: BEGIN
39:   var1 := 1;
40:   var2 := auxiliar2;
41:   var3 := proc1(var1+var2);
42:   var1 := funcion(var2+var3);
43:   procedimiento1(var1+var2-var3);
44: END.
```

Answer the follwing questions:

(a) How many scopes, where any entry has been inserted, have been created during the analysis once the 19th line has been completely processed:

a) 7	b) 5	**c) 3**	d) 1

(b) How many scopes, where any entry has been inserted, have been created during the analysis once the 40th line has been completely processed:

a) 7	**b) 5**	c) 3	d) 1

(c) How many entries will exist once the 35th line has been completely processed:

a) 13	b) 15	c) 17	d) 19

(d) How many symbol table access errors have been detected during the analysis of the program:

a) 0	b) 1	c) 2	**d) 3**

(e) If any error has been detected, detail: line number, identifier associated and the operation associated –insertion or search.

2. Given the following grammar which describes a language similar to *SimplePascal*:

```
programa ::= <PROGRAM> <ID>  ";" bloque "."
bloque ::= declaracionL <BEGIN> resto <END>
bloqueEjecutable ::= defVarL <BEGIN> resto <END>
resto ::= bloqueEjecutable resto | lambda

declaracionL ::= declaracion declaracionL | lambda
declaracion ::= defVar | defProcedimiento | defFuncion

defVar ::= <VAR> listaVarL | lambda
defProcedimiento ::= <PROCEDURE> <ID> parFormales ";" bloque ";"
defFuncion ::= <FUNCTION> tipo ":" <ID> parFormales ";" bloque ";"

parFormales ::= "(" listaParamL  ")" | lambda
listaParamL ::= listaParam ";" listaParamL | listaParam
listaParam ::= tipo ":" <ID> lid
lid ::= "," <ID> lid | lambda
listaVarL ::= listaVar ";" listaVarL | listaVar ";"
listaVar ::= tipo ":" <ID> lid

tipo ::= <INTEGER> | <REAL> | <BOOL>
```

It is asked to produce a traslator that generates the structure of the symbol table of a given program. Next the usable API is described:

InsertID(class , lexem , type) : Inserts an identifier into the currently active scope. The possible values of *class* are: proc, fun, param, vble. *Lexem* is the name. The possible values of *type* are: int, real, bool. Some examples are:

InsertID(proc , proc1 ,) : Inserts the entry corresponding to a procedure called "proc1". Note that no value is given to the type argument.

InsertID(fun , func1 , real) : Inserts the entry corresponding to a function called "func1" that returns a real value.

InsertID(param , arg1 , int) : Inserts the entry corresponding to a formal parameter called "arg1" of integer type. The paramenters will be inserted in the scope associated to the function or procedure that they belong to.

InsertID(vble , media , bool) : Inserts the entry corresponding to a variable called "media" of boolean type.

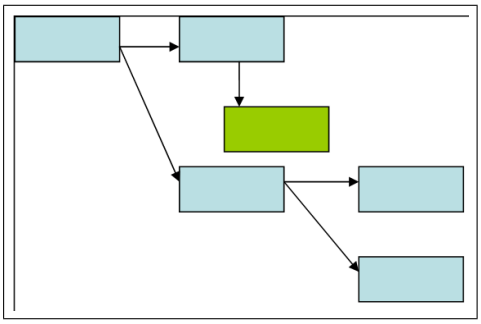
NewScope(name) : Creates a new scope associated to the entry of the identifier passed as the argument *name*, and also it activates the scope created. Note that the identifier entry must be previously created, except for the root scope whose identifier does not need to be created.

ExitScope() : Exits from the active scope and activates the parent scope.

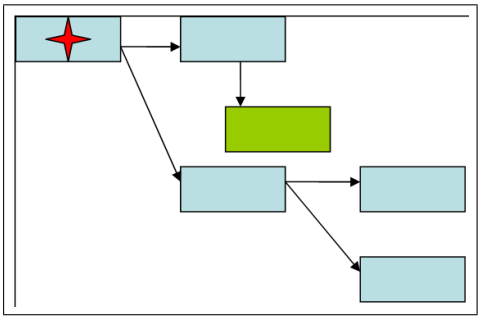
B Coding exercises

Next, eight exercises are given to work with the SOTA tool. The main task is write a program in SimplePascal that generates a symbol table whose structure follows the scheme given in the images. Blue rectangles represent to named scopes, while green rectangles represent to anonious scopes. Finally, red stars indicate symbol table errors to be detected.

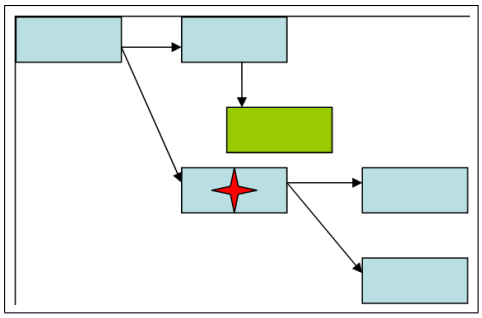
Exercise 1



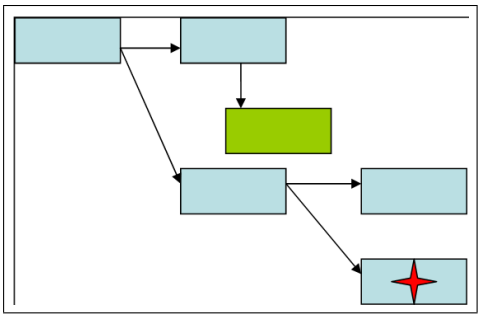
Exercise 2



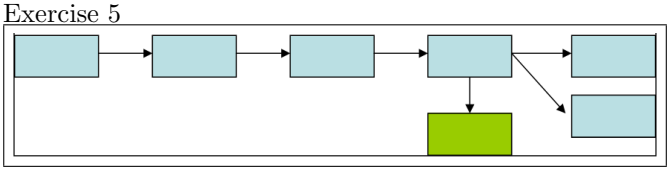
Exercise 3



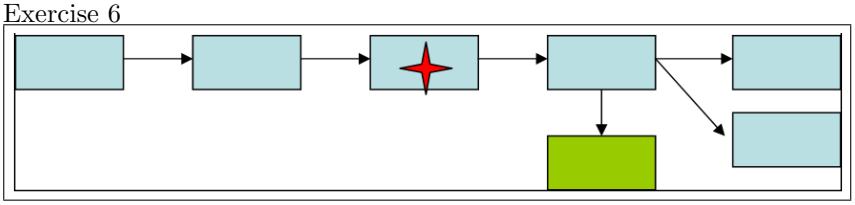
Exercise 4



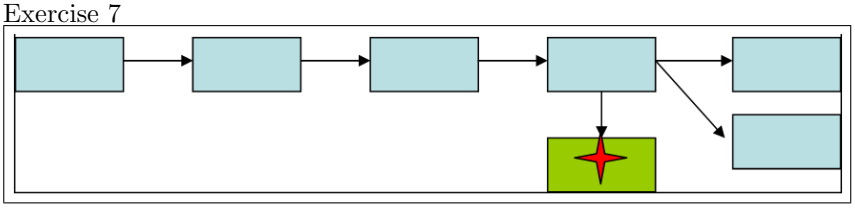
Exercise 5



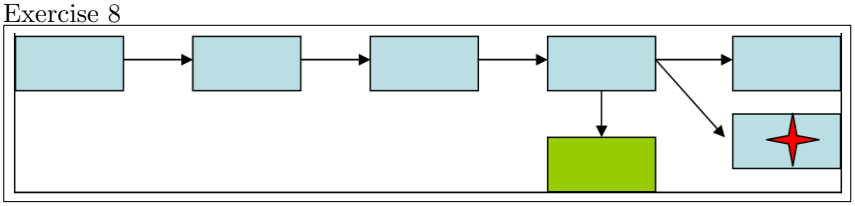
Exercise 6



Exercise 7



Exercise 8



References

1. F. Almeida-Martnez, J. Urquiza-Fuentes, and J. Velzquez-Iturbide. Visualization of syntax trees for language processing courses. *Journal of Universal Computer Science*, 15(7):1546–1561, 2009.

2. T. Chen and T. Sobh. A tool for data structure visualization and user-defined algorithm animation. In *Frontiers in Education Conference, 2001. 31st Annual*, volume 1, pages TID –2–7 vol.1, Los Alamitos, CA, USA, 2001. IEEE Computer Society Press.
3. R. Decker and S. Hirshfield. The pippin machine: simulations of language processing. *J. Educ. Resour. Comput.*, 1:4–17, December 2001.
4. M. Gallego-Carrillo, F. Gortázar-Bellas, J. Urquiza-Fuentes, and J. A. Velázquez-Iturbide. Sota: a visualization tool for symbol tables. *SIGCSE Bull.*, 37:385–385, June 2005.
5. C. Hundhausen, S. Douglas, and J. Stasko. A meta-study of algorithm visualization effectiveness. *J. Visual Lang. Comput.*, 13(3):259–290, 2002.
6. A. Kaplan and D. Shoup. Cupv – a visualization tool for generated parsers. *SIGCSE Bull.*, 32:11–15, March 2000.
7. S. Khuri and Y. Sugono. Animating parsing algorithms. *SIGCSE Bull.*, 30:232–236, March 1998.
8. N. Krebs and L. Schmitz. Visual syntax tools. http://www2.cs.unibw.de/Tools/Syntax/english /index.hmtl, 2004.
9. M. E. Lovato and M. F. Kleyn. Parser visualizations for developing grammars with yacc. *SIGCSE Bull.*, 27:345–349, March 1995.
10. A. Moreno, N. Myller, E. Sutinen, and M. Ben-Ari. Visualizing programs with jeliot 3. In *AVI '04: Proceedings of the working Conference on Advanced Visual Interfaces*, pages 373–376, New York, NY, USA, 2004. ACM Press.
11. M. Mwernik and V. Zumer. An educational tool for teaching compiler construction. *IEEE Transactions on Education*, 46, 2003.
12. R. D. Resler and D. M. Deaver. Vcoco: a visualisation tool for teaching compilers. In *Proceedings of the 6th annual conference on the teaching of computing and the 3rd annual conference on Integrating Technology into Computer Science Education: Changing the delivery of computer science education*, ITiCSE '98, pages 199–202, New York, NY, USA, 1998. ACM.
13. S. Rodger and T. Finley. *JFLAP - An Interactive Formal Languages and Automata Package*. Jones and Bartlett, Sudbury, MA, USA, 2006.
14. S. H. Rodger, E. Wiebe, K. M. Lee, C. Morgan, K. Omar, and J. Su. Increasing engagement in automata theory with jflap. *SIGCSE Bull.*, 41:403–407, March 2009.
15. J. Urquiza-Fuentes, J. Velázquez-Iturbide, M. Gallego-Carrillo, and F. Gortázar-Bellas. An evaluation of a symbol table visualization tool. In *Proc. of 8th International Symposium on Computers in Education (SIIE 2006)*, pages 198–205, 2006.
16. J. Velázquez-Iturbide, C. Pareja-Flores, and J. Urquiza-Fuentes. An approach to effortless construction of program animations. *Comput. Educ.*, 50(1):179–192, 2008.