Some modeling technologies in educating of young IT experts in the field of formal languages and their semantics

William Steingartner, Valerie Novitzká, Pavol Zorvan
Technical University of Košice
Faculty of Electrical Engineering and Informatics
Letná 9, 04200 Košice, Slovak Republic
{william.steingartner, valerie.novitzka}@tuke.sk, pavol.zorvan@student.tuke.sk

Abstract. *Visualization of calculations in the field of semantic methods significantly contributes to a better understanding of the individual steps of the methods used. The use of virtual machines to design and prove the language properties is an important element in teaching compiler design. In this paper, we present visualization software that for a given input source in the model (toy) language Jane generates bytecode for an existing Computron virtual machine platform and a sequence of instructions for an abstract machine for operational semantics.*

Keywords. abstract implementation, code translation, compiler, formal language, semantics, university didactics, virtual machine, visualization

1 Introduction

The training of future software experts requires a relatively broad knowledge base that the university can provide. Experience shows that teaching technology alone and programming principles are not enough due to rapid changes in information and communication technologies. In addition to advanced languages and technologies, curricula commonly include courses that provide theoretical knowledge of languages, correctness, and semantics, including the principles of language design. The curriculum for the study field of Informatics at the Faculty of Electrical Engineering and Informatics in Košice contains two important courses that provide a broad theoretical and practical framework in the field of formal languages and semantics: Formal languages (undergraduate course) and Semantics of programming languages (graduate course).

At present, when the contact of educators and students is limited mostly to online events, it is the interactive teaching software tools that play an irreplaceable *rôle* in the teaching process. The fact that theoretically oriented courses, such as Semantics of programming languages (and other courses focused on formal methods and theoretical foundations of computer science (Mihályi et al., 2017)) require special support for students, has forced a reassessment of the course teaching method and the design and development of new interactive teaching software tools. Similar software tools (even for other courses) were created to simplify the understanding process of the basics, for example, the course on Dynamic Geometry (Radaković and Herceg, 2018; Herceg et al., 2019), Formal Logic (Schreiner, 2019), Object-oriented Programming (Vaclavkova et al., 2019), Operating Systems (Genči et al., 2017) or general formal methods in software development (Korečko et al., 2014).

The paper is organized as follows: in Section 2, we present a pedagogical background and we explain our motivation for this research and development. Section 3 contains all necessary basic notions and the theory about the abstract machine and Computron Virtual Machine (VM). In Section 4 we present basic information about compilation and the main phases of the compiler concerning the software that is a subject of this article. Then, Section 5 describes the developed software and its main features and Section 6 presents on a specific example the use of our software and focuses on initial experience after software deployment. Finally, the Section 7 concludes our article.

2 Pedagogical background and motivation

For a better understanding of formal methods, we are currently focusing our research on the visualization of these methods. This research is covered by the KEGA project, which we refer to in the Acknowledgments section of this article. As a part of the project, we have already implemented some software tools that are used to visualize formal semantics for imperative languages. Some results are presented in the works (Steingartner, 2020, 2021). Because we want to continue this trend and prepare the basis for the visualization of semantic methods for domain-specific languages, we consider it necessary to expand the set of semantic methods in this field.

The course Formal languages is devoted to algorithms and techniques for processing text strings, es-

pecially parsing and processing sentences of formal languages. After completing the course, the students should be able to use regular expressions and understand the definition of formal languages using grammars. Students will be able to design grammar and implement a simple context-free language parser. In addition, they will know the basic algorithms for searching for strings and patterns in the text. For this course, a very useful tool has been developed – Computron VM. The software Computron VM is a virtual machine, which associates theoretical knowledge and practical compiler technology. For details, the readers are referred to (Kollár, 2012).

In the course of Semantics of programming languages, the students will get acquainted with different approaches to defining the semantics of programming languages. They will learn the most important methods of semantic description, such as operational, natural, denotational, algebraic, axiomatic, action and newly defined categorical semantics. They gain knowledge about the applications of various methods in the design, definition and implementation of programming languages. On the example of a simple procedural language, they will be able to define the semantics of standard and some extended language constructions.

As part of the mentioned KEGA project, we implemented some software tools enabling the visualization of selected semantic methods. We realize the visualization of semantic methods on the model abstract language *Jane*, which contains typical constructs (or patterns) known from imperative languages. Because a language based on the same principles also serves as the basis for constructing a compiler on the course of Formal languages, we are inspired to provide a tool to verify the translation from *Jane* to Computron VM code and for translating an input code into an abstract implementation for operational semantics. This will provide students with an interesting software tool that will be usable for both mentioned courses and can be successfully used in practice to verify the properties of compilation and some parts of the language.

3 Theoretical foundations

In this section, we present some necessary theoretical foundations and basic notions. In subsection 3.1, we present a language that is the subject of a study of both a Formal languages course and a Semantics course. Then in subsection 3.2, we present the Computron VM which serves as a main software tool for the course Formal languages. Subsection 3.3 briefly describes the main aspects of the abstract implementation of imperative languages and defines one kind of abstract machine.

3.1 Language *Jane*

For both mentioned courses, a simple abstract (toy) language for constructing a compiler into Computron VM code and for defining the semantic methods and proving their properties and equivalences is used. It is a non-real programming language grounded in imperative paradigm, epitomizing a tiny core fragment of conventional mainstream languages such as C or Java: standard imperative constructs as sequences of statements, selection (conditional), repetition (loops) and handling the values in memory (variables assignment). Moreover, statements for user input and output and nested blocks are defined. For the research and development, this language has been adopted by many authors and researchers. Moreover, there have been formulated also many approaches thanks to this abstract language. Some authors refer to this language as *IMP* (as simple imperative language) (Roşu and Şerbănuţă, 2010) or as *While* (defined for instance in (Nielson and Nielson, 2007)). We adopted the structure of this language as well, and we refer to this language as *Jane* (Steingartner et al., 2019).

Since the definition of this language is well-known (Nielson and Nielson, 2007), we do not repeat it and we present basic aspects only briefly. We build on the preliminaries that the definition of the language consists of (Dedera, 2014):

- syntax definition using EBNF, inductive definition or derivation rules,
- formal definition of the semantics of a particular language using an appropriate semantic method.

The abstract syntax of the language *Jane* is defined by the set of rules taking the syntactic elements from the following syntactic domains (or syntax categories):

$n \in \mathbf{Num}$	(strings of digits)
$x \in \mathbf{Var}$	(variables' names)
$e \in \mathbf{Expr}$	(arithmetic expressions)
$b \in \mathbf{Bexpr}$	(Boolean expressions)
$S \in \mathbf{Statm}$	(statements)
$D \in \mathbf{Decl}$	(declarations)

For each syntactic domain, we define exactly one production rule given in EBNF. Moreover, we consider for the language the declarations of variables.

- the elements in domains for numerals and variables' names have no internal structure from the semantic point of view, so we do not define rules for them;
- production rule for arithmetic expressions:

$$e ::= n \mid x \mid e + e \mid e - e \mid e * e \mid (e), \tag{1}$$

- production rule for Boolean expressions:

$$b ::= \mathtt{true} \mid \mathtt{false} \mid e = e \mid e \leq e \mid \neg b \mid b \wedge b \mid (b), \tag{2}$$

- production rule for the statements:

$$S ::= \quad x := e \mid \textbf{skip} \mid S; S \mid \textbf{if } b \textbf{ then } S \textbf{ else } S \mid$$
$$\mid \textbf{while } b \textbf{ do } S \mid \textbf{read } x \mid \textbf{print } e \mid \{S\} \tag{3}$$

- production rule for the declarations:

$$D ::= \quad \textbf{var } x; D \mid \varepsilon, \tag{4}$$

where ε stands for an empty declaration.

We note, that the sets of arithmetic and Boolean expressions are in general not limited to listed syntactic constructs and both rules can be extended by providing other correct syntactic constructs.

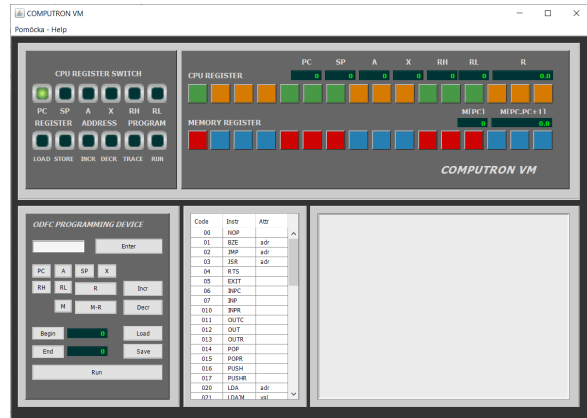

Figure 1: Computron User Interface

3.2 Computron VM architecture

Computron VM is a virtual machine based on a word-oriented architecture consisting of CPU and memory. The CPU has four single-word registers: A – accumulator, PC – program counter, SP – stack pointer and X – index register, and one double-word register R. Computron memory is organized as an array of capacity of 64K, i.e. 65536 single-word cells where indices of array M are considered as memory addresses. Computron Instruction Set consists of the following instruction categories:

1. No Operation Instruction
2. Control Flow Instructions
3. Input from keyboard Instructions
4. Output to screen Instructions
5. Stack `PUSH` and `POP` Instructions (single and double word)
6. Load accumulator A from memory
7. Load register R from memory
8. Store accumulator A and register R to memory
9. Load/Store index register X and stack pointer SP from/to memory
10. Boolean operations
11. Comparisons of words using register A and double words using register R
12. Arithmetic operations on integer and real numbers

As an example, we show the semantics of the instruction named `BZE` $addr$. This is a standard control-flow instruction: branch if zero, which makes a conditional jump to address $addr$:

$$\textbf{if } (A = 0) \textbf{ then } PC := addr \textbf{ else } PC := PC + 2;$$

For a complete description of the operational semantics of Computron VM, we refer the reader to the technical report (Kollár, 2011). The Computron User Interface is depicted in Figure 1.

In the course, students learn how to define the grammar and how to design and develop their compiler that transforms the input code written in *Jane* to the byte-code that is executable on Computron VM. We can simply say that the compiler generates the Computron data to the output binary file, where the binary file is a file of unsigned short values and it can be loaded by the ODFC Programming device of Computron VM.

3.3 Abstract implementation

A formal specification of the semantics of a programming language is useful when implementing it. In particular, it becomes possible to argue about the correctness of the implementation. This is usually realized as a translation of the higher-level language into a structured form of assembler code for a selected abstract machine. The next step is the proof of whether the translation is correct (Nielson and Nielson, 2007). An abstract machine is an intermediate language with a small-step operational semantics (Plotkin, 2004). Abstract machines provide an intermediate language stage for compilation. They bridge the gap between the high level of a programming language and the low level of a real machine. The instructions of an abstract machine are tailored to the particular operations required to implement operations of a specific source language or class of source languages (Diehl et al., 2000). First, the meaning of the abstract machine instructions is defined by operational semantics. Then translation functions that will map expressions and statements in the higher-level language into sequences of such instruction are defined. The correctness result then states that if a program is translated into code and the code is executed on the abstract machine then the same result must be provided as by semantic functions for structural operational semantics.

The description of particular computational steps of abstract machine is usually given by configurations of the form

$$\langle c, \sigma, s \rangle \,,$$

where

- c stands for a code – the sequence of instructions to be executed,
- σ is the evaluation stack, and

- s represent a storage.

Here, the evaluation stack is used to evaluate arithmetic and Boolean expressions. Formally, it is defined as a list of values that are elements of the semantic domain

$$\mathbf{Stack} = (\mathbf{Z} \cup \mathbf{B})^{*},$$

where $\mathbf{Z}$ stands for a set of integers and $\mathbf{B}$ is a set of semantic values of Boolean constants, $\mathbf{B} = \{\mathbf{ff}, \mathbf{tt}\}$. For the simplicity, the storage is assumed to be a memory state, an abstraction of memory. The complete definition can be found in (Nielson and Nielson, 2007).

The language of abstract machine is a structured assembler, which consists of instructions. These instructions are given by the following abstract syntax expressed by Backus-Naur form:

$$\begin{aligned} \mathit{instr} \quad ::= \quad & \texttt{PUSH}-n \mid \texttt{ADD} \mid \texttt{SUB} \mid \texttt{MULT} \mid \\ & \texttt{TRUE} \mid \texttt{FALSE} \mid \texttt{EQ} \mid \texttt{LE} \mid \texttt{AND} \mid \texttt{NEG} \mid \\ & \texttt{FETCH}-x \mid \texttt{STORE}-x \mid \\ & \texttt{EMPTYOP} \mid \texttt{BRANCH}(c, c) \mid \texttt{LOOP}(c, c) \\ c \quad ::= \quad & \varepsilon \mid \mathit{instr} : c \end{aligned}$$

A meta-variable c is ranging over a syntactic domain $\mathbf{Code}$ of sequences of instructions.

The semantics of the instructions of the abstract machine is given by operational semantics. The usual way of specifying the execution steps is by a transition system. The definition of a transition system for an abstract machine and the formal semantics of instructions can be found in (Nielson and Nielson, 2007).

A code for the abstract machine is generated by the translating functions. For each syntactic domain, one translating function is defined. Arithmetic and Boolean expressions are evaluated on the evaluation stack of the machine and the code to be generated must to effect this. This is accomplished by the following total functions:

$$\mathscr{TE} : \mathbf{Expr} \to \mathbf{Code}$$

and

$$\mathscr{TB} : \mathbf{Bexpr} \to \mathbf{Code}.$$

In both functions, the code generated for binary expressions consists of the code for the right argument followed by that for the left argument and finally the appropriate instruction for the operator. In this way, it is ensured that the arguments appear on the evaluation stack in the order required by the instructions.

The translation of statements into abstract machine code is given by the function

$$\mathscr{TS} : \mathbf{Statm} \to \mathbf{Code}.$$

Specifications of functions $\mathscr{TE}$, $\mathscr{TB}$ and $\mathscr{TS}$ are in the Table 1 and Table 2.

The specifications of translation functions listed in Tables 1 and 2 serve as the basis for the design of the compiler, which is the subject of this article.

Table 1: Translation of arithmetic and Boolean expressions

$$\begin{aligned} \mathscr{TE}\llbracket n \rrbracket &= \texttt{PUSH}-n \\ \mathscr{TE}\llbracket x \rrbracket &= \texttt{FETCH}-x \\ \mathscr{TE}\llbracket e_1 + e_2 \rrbracket &= \mathscr{TE}\llbracket e_2 \rrbracket : \mathscr{TE}\llbracket e_1 \rrbracket : \texttt{ADD} \\ \mathscr{TE}\llbracket e_1 - e_2 \rrbracket &= \mathscr{TE}\llbracket e_2 \rrbracket : \mathscr{TE}\llbracket e_1 \rrbracket : \texttt{SUB} \\ \mathscr{TE}\llbracket e_1 * e_2 \rrbracket &= \mathscr{TE}\llbracket e_2 \rrbracket : \mathscr{TE}\llbracket e_1 \rrbracket : \texttt{MULT} \\ \\ \mathscr{TB}\llbracket \texttt{true} \rrbracket &= \texttt{TRUE} \\ \mathscr{TB}\llbracket \texttt{false} \rrbracket &= \texttt{FALSE} \\ \mathscr{TB}\llbracket e_1 = e_2 \rrbracket &= \mathscr{TE}\llbracket e_2 \rrbracket : \mathscr{TE}\llbracket e_1 \rrbracket : \texttt{EQ} \\ \mathscr{TB}\llbracket e_1 \leq e_2 \rrbracket &= \mathscr{TE}\llbracket e_2 \rrbracket : \mathscr{TE}\llbracket e_1 \rrbracket : \texttt{LE} \\ \mathscr{TB}\llbracket \neg b \rrbracket &= \mathscr{TB}\llbracket b \rrbracket : \texttt{NEG} \\ \mathscr{TB}\llbracket b_1 \wedge b_2 \rrbracket &= \mathscr{TB}\llbracket b_2 \rrbracket : \mathscr{TB}\llbracket b_1 \rrbracket : \texttt{AND} \end{aligned}$$

Table 2: Translation of statements in *Jane*

$$\begin{aligned} & \mathscr{TS}\llbracket x := e \rrbracket = \mathscr{TE}\llbracket e \rrbracket : \texttt{STORE}-x \\ & \mathscr{TS}\llbracket \mathbf{skip} \rrbracket = \texttt{EMPTYOP} \\ & \mathscr{TS}\llbracket S_1; S_2 \rrbracket = \mathscr{TS}\llbracket S_1 \rrbracket : \mathscr{TS}\llbracket S_2 \rrbracket \\ & \mathscr{TS}\llbracket \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2 \rrbracket = \\ & = \mathscr{TB}\llbracket b \rrbracket : \texttt{BRANCH}(\mathscr{TS}\llbracket S_1 \rrbracket, \mathscr{TS}\llbracket S_2 \rrbracket) \\ & \mathscr{TS}\llbracket \mathbf{while}\ b\ \mathbf{do}\ S \rrbracket = \texttt{LOOP}(\mathscr{TB}\llbracket b \rrbracket, \mathscr{TS}\llbracket S \rrbracket) \end{aligned}$$

4 Code compilation

Programs are written in a specific programming language according to the target platform, on which they are executed. Usually, the code is written in higher-level languages. The next step is then a translation into the low-level language to be readable by a target platform (Kollár, 2012).

Programs used to perform a translation process of code from one form to another are commonly called compilers. In the case of translation from high-level to low-level language, they are also called compilers and in the case of source-to-source translation, they are referred to as *transpilers* (Herlihy et al., 2019).

At the end of the translation process is supposed to be the code in the target form, which should have exactly equal semantics as the original program. If we

want to perform translation of code, first we must be sure that the source code does not contain any errors. That is why every compiler have two parts: *analysis* and *synthesis* (Aho et al., 2006).

4.1 Analytic part of translation

The *analysis* divides the input source code into tokens and assigns them a grammatical structure. It creates a tree form of the source program that is passed to the *synthesis*.

If the analysis detects that the source program contains syntactic or semantic errors, instead of creating translation in the target language it provides a message with information about found errors, so the user can review the input code. The analysis consists of three standard phases:

- lexical analysis,
- syntax analysis,
- semantic analysis.

Lexical analysis, also known as scanning, reads the stream of characters and groups them into meaningful language symbols called *lexemes*. For each lexeme, it produces a token, which is usually pair of a token name and a specific attribute value. Lexical errors occur when the compiler is unable to recognize some specific sequences of characters as lexemes of the source language.

Syntax analysis uses a stream of tokens produced by lexical analysis to create a tree-like representation of the code that depicts the grammatical structure of the token stream. Typically, it uses a syntax tree in which each interior node represents an operation and leaves represent arguments of the parental operation. Syntax errors occur when some operators in interior nodes do not have the required number of arguments.

The semantic analysis uses a syntax tree to check the semantic consistency of the source program with the language definition. An important part of semantic analysis is type checking, where the compiler checks that each function has attributes of expected type (Aho et al., 2006).

4.2 Synthetic part of translation

The synthetic part creates translation from an intermediate representation of the source program, which is the result of the analysis part. If the analysis ends without errors, synthesis maps the syntax tree of the source program into the target code. An important part of code generation is also the assignment of registers to hold variables used in code. The result of synthesis should be semantically equivalent code executable on target platform (Aho et al., 2006).

5 Generator of byte-code

We have developed a software tool that reads input code written in an extended version of language *Jane* and translates it into several forms. The software is developed as a web application, so the user can access it directly through a web browser without having to install it directly. The user only needs to connect to the server on which the application is running (Zorvan, 2021).

The generator produces outputs into three target forms. The first one is the translation into instructions of virtual machine Computron, which is used in course Formal languages. The executable code for the Computron VM is stored in a binary file, where each instruction is stored in a one-word position (instruction without an argument) or a two-word position (instruction with an argument). Therefore, an output for the Computron is done in two different forms – one is binary which is executable for the virtual machine, the other form is user-readable (textual) and it provides the full instruction names.

The second is a translation into instructions of an abstract machine which is used as one of the methods to evaluate the semantics of programming languages.

The last form of output code is an XML structure, that can be used as a descriptive form for exporting analyzed language to another platform or for other extensions.

For realizing the compilation and construction of the *Jane* compiler, the ANTLR tool (Parr, 2013) has been used, which is one of the best known and most widely used language parsers used to implement compilers and language processors. The generated compiler performs lexical and syntax analysis based on the language definition and automatically creates a syntax tree of the program. Then it allows the compiler to walk through tree in node-by-node mode, calling specific translation functions for each type of node on entering and exit and build translated code by using them (Parr, 2013).

Semantic analysis is not performed automatically but is implemented directly in translation functions.

The application consists of these main components:

- editor used for writing or loading of input code,
- buttons to switch the display between editor and translations into various forms,
- the button that performs translation,
- panel with translation settings,
- buttons to load code from file and to save translated codes into the local device,
- output field with reported errors that were found during the analytic part of translation.

The Graphical user interface of our application is depicted in Figure 2.

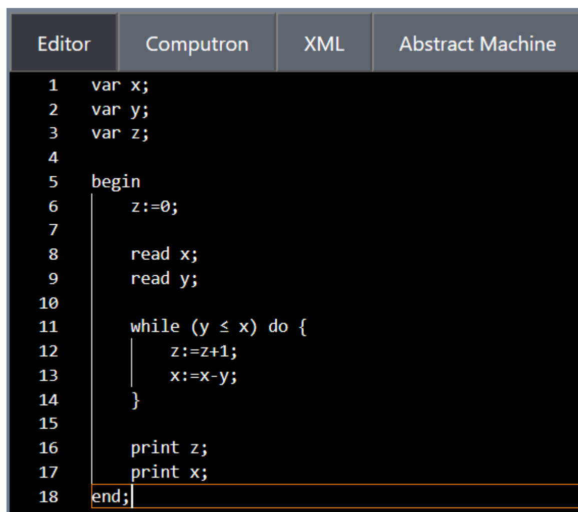

Figure 2: Byte-code Generator User Interface

5.1 Editor

To allow the user to easily enter the input source code, we added a code editor component into the application UI, that reads input source code from the user (manually or by uploading). For that purpose, we used Monaco editor[1] which also powers VS Code and has several useful utilities.

Monaco editor automatically provides line numbers at the beginning of each line. Monaco allows the programmers to define their language and then it offers by auto-complete function all available code snippets and commands to the user, which makes programming in an editor much faster and easier. It also replaces special keywords with symbols that are usually not on computer keyboards but are used in language *Jane*. For example, the editor replaces keyword **/neq** with the symbol $\neq$, or the keyword **/leq** with the symbol $\leq$.

There are four buttons in the upper right corner of the window above the textarea. They allow the user to switch between the editor and individual translations. Each button opens a specific tab below the editor, which displays the corresponding type of translation after a successful translation process. If an error occurs during the translation process, all translation windows remain blank.

If the source code was entered without errors, a translated form is provided when the compilation process is complete. However, if the specified source code contained errors, the program will provide an error listing.

5.2 Panel with settings

Panel with settings contains three translation specifications (Figure 3). First, it allows the user to choose, which translations should be performed – every translation is represented by its checkbox.

[1]https://microsoft.github.io/monaco-editor/

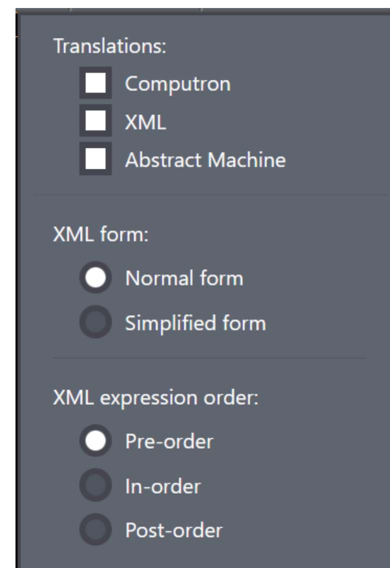

Figure 3: Compilation settings (white color represents checked option)

The second and the third setting applies to XML form only. User can choose normal XML form, in which expression annotations correspond to the *Jane* grammar in detail, or simplified form, that use the same wording for each arithmetic and logical expression. Lastly, the user can choose the order, in which are arithmetic and logical expressions written, specifically we can choose the position of operation signs – before operands, between them or at the end (prefix, infix or postfix form).

5.3 Loading and saving code

The load button allows the user to load some code from a file into the editor. The download button opens a dropdown menu with an option for every translation. After clicking on a specific translation, it opens *"Save as..."* pop-up window, which allows the user to set a name and select a location in the local device, where the user wants to save the output after the compilation. Every translation also uses its file format.

5.4 Compilation errors

If the compilation process returns errors instead of output code, a new panel will appear at the bottom of the screen. Error panel contains all errors, that have been found during the translation process. Each error consists of an info message that describes the error and the exact location in code given by line and character numbers.

6 Using a software tool

As we introduced in the previous sections, our software tool realizes a compilation of input source code written

```
PUSH-0 : STORE-z : READ-x : READ-y : LOOP( FETCH-x : FETCH-y :
LE, PUSH-1 : FETCH-z : ADD : STORE-z : FETCH-y : FETCH-x : SUB :
STORE-x ) : FETCH-z : PRINT : FETCH-x : PRINT
```

Figure 4: Compiled code for abstract machine

in the language *Jane* and based on selected preferences of the user, it provides three possible outputs: byte-code for the Computron VM, descriptive XML form and the sequence of instructions of the abstract machine for the structural operational semantics.

In Subsection 6.1, an example of using the software is presented. Subsection 6.2 focuses on initial experience after the software deployment.

6.1 Example of use

As an example, we show the particular visualization phases on a simple example of the integer quotient and remainder. Let the input program be as follows:

var x;
var y;
var z;

begin
  $z := 0$;
  read x;
  read y;

  while $(y \leq x)$ **do** {
    $z := z + 1$;
    $x := x - y$;
  }

  print z;
  print x;
end;

In this program, user is allowed to provide input values: dividend is stored in the variable x, and divisor in variable y. After the calculation, the result is stored in two variables: the quotient is in the variable z and remainder after the division is in x.

The result of compiling an input source code into the abstract machine code is in Figure 4.
The result after compilation into the Computron VM code is in Figure 5 (the input is arranged in three columns for better readability).
A fragment of the descriptive XML form is in Figure 6.
Finally, when running the compiled code on the Computron VM and giving as input values: $x = 17$ (dividend) and $y = 5$ (divisor), we get a result 3 (quotient, stored in z) and 2 (remainder, stored in x). This situation is listed in Figure 7. We note that in the current version of *Jane*, we do not recognize a visual offset in the form of *LF*. So we can see the result as one string 32.

```
0 LDS 4
2 JMP 8
4 81 //WORK MEMORY ADDRESS
5 NOP
6 NOP
7 NOP
8 LDAM 0
10 PUSH
11 POP
12 STA 7
14 INP
15 STA 5
17 INP
18 STA 6
20 LDA 6
22 PUSH
23 LDA 5
25 PUSH
26 POP
27 STA 4
29 POP
30 LE 4
32 PUSH
33 POP
34 BZE 70
36 LDA 7
38 PUSH
39 LDAM 1
41 PUSH
42 POP
43 STA 4
45 POP
46 ADD 4
48 PUSH
49 POP
50 STA 7
52 LDA 5
54 PUSH
55 LDA 6
57 PUSH
58 POP
59 STA 4
61 POP
62 SUB 4
64 PUSH
65 POP
66 STA 5
68 JMP 20
70 LDA 7
72 PUSH
73 POP
74 OUT
75 LDA 5
77 PUSH
78 POP
79 OUT
80 EXIT
```

Figure 5: Compiled code for Computron VM

```
<program>
   <declarations>
      <var>x</var>
      <var>y</var>
      <var>z</var>
   </declarations>
   <statements>
      <assign>
         <var>z</var>
         <value>
            <term>0</term>
         </value>
      </assign>
      <read>
         <var>x</var>
      </read>
      <read>
         <var>y</var>
      </read>
      <while>
         <condition>
            <expr>
               <operation>≤</operation>
               <var>y</var>
               <var>x</var>
```

Figure 6: Description of an input code in XML format

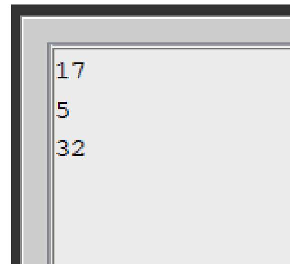

Figure 7: Computron screen with input and output (a fragment)

6.2 Using the tool in a real environment

Teaching tools often facilitate the teaching process. In the teaching of information technologies, such tools are various modeling and visualization tools that allow us to visualize the individual steps of various procedures and processes.

The teaching of the formal foundations of computer science is often associated with mathematical foundations, which are still taught in the classical form – using blackboard and chalk and accompanying interpretation. The social situation since 2020 has required the transfer of most of the teaching process to the online space, where classical methods were very limited or impossible to use. During the online teaching, short explanatory videos proved to be useful, but they did not allow interactive intervention into the content. Therefore, we focused on the main idea – to support the classical interpretation of the curriculum using visualization tools. For the courses Formal Languages and Semantics of Programming Languages, we have introduced a new tool that connects aspects from both courses. Its use is intended for teachers in the phase of curriculum interpretation (demonstrating and commenting on individual steps), for students (in the phase of study and experimentation as well as during laboratory work) and for experts from practice who need to formally verify language properties and so find out critical points in the programs.

Our experience so far shows that students and new educators who join the team positively welcome the individual software visualization tools, which equally contribute to the attractiveness and simplification of the teaching process. The use of the program in pedagogical practice is currently still in its beginning, so deeper feedback from students will be the subject of further research in the future.

7 Conclusion

In this article, we have presented the results achieved in the field of design and development of visualization software tools, which are part of the KEGA project cited in the Acknowledgments section. The inspiration for this research was for us also the current situation, when a large part of the educational process moved to the online space. We believe that software support aimed at visualizing computational steps will make a significant contribution to making the teaching of formal methods for software engineering more attractive. This will greatly facilitate the clarity of semantic methods as well as the design of one's language or verification of its properties.

In our future research, we want to focus on exploring other properties of imperative languages and extend existing methods in the field of semantic modeling or visualization of semantic methods. We want to extend this area to some domain-specific and concatenative languages, which are evolving rapidly, and we see their use in the field of software engineering as a benefit for the future. Secondly, we also want to focus on the active use of this tool in teaching, its full integration during the tuition of the mentioned courses and to deal with and explore the satisfaction of the users, learning performance and progress.

Acknowledgments

This work was supported by project KEGA 011TUKE-4/2020: "A development of the new semantic technologies in educating of young IT experts", granted by the Cultural and Education Grant Agency of the Slovak Ministry of Education.

References

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley.

Dedera, L. (2014). *Computer languages and their processing*. Armed Forces Academy of General Milan Rastislav Štefánik. (in Slovak).

Diehl, S., Hartel, P., and Sestoft, P. (2000). Abstract machines for programming language implementation. *Future Generation Computer Systems*, 16(7):739–751.

Genči, J., Bilanová, Z., Deák, A., and Vrábel, M. (2017). Project and team based teaching of system programming in the course of operating systems. In *2017 15th International Conference on Emerging eLearning Technologies and Applications (ICETA)*, pages 1–6.

Herceg, Đ., Radaković, D., Ivanović, M., and Herceg, D. (2019). Possible improvements of modern dynamic geometry software. *Computer Tools in Education*, (2):72–86.

Herlihy, A., Khineika, A., and Shestak, I. (2019). Transpiling between any programming languages (part 1). `https://engineering.mongodb.com/post/transpiling-between-any-programming-languages-part-1`. accessed: 2021-06-12.

Kollár, J. (2012). Computron VM: Identification of expert knowledge in virtual computer architecture development. In *CSE 2012 : International Scientific Conference on Computer Science and Engineering*, pages 87–94.

Kollár, J. (2011). Computron VM. `http://people.tuke.sk/jan.kollar/FJ/ComputronVM/help/CvmHelp.pdf`. accessed: 2021-05-28.

Korečko, Š., Sorád, J., Dudláková, Z., and Sobota, B. (2014). A toolset for support of teaching formal software development. In *International Conference on Software Engineering and Formal Methods*, pages 278–283. Springer.

Mihályi, D., Peniašková, M., Perháč, J., and Mihelič, J. (2017). Web-based questionnaires for type theory course. *Acta Electrotechnica et Informatica*, 17(4):35–42.

Nielson, H. and Nielson, F. (2007). *Semantics with Applications: An Appetizer (Undergraduate Topics in Computer Science)*. Springer-Verlag, Berlin, Heidelberg.

Parr, T. (2013). *The Definitive ANTLR 4 Reference*. Pragmatic Programmers, LLC, The, Raleigh.

Plotkin, G. (2004). A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139.

Radaković, D. and Herceg, Đ. (2018). Towards a completely extensible dynamic geometry software with metadata. *Computer Languages, Systems & Structures*, 52:1–20.

Roşu, G. and Şerbănută, T. F. (2010). An overview of the k semantic framework. *The Journal of Logic and Algebraic Programming*, 79(6):397–434. Membrane computing and programming.

Schreiner, W. (2019). Logic and Semantic Technologies for Computer Science Education. In Steingartner, W., Š. Korecko, and Szakál, A., editors, *Informatics'2019, 2019 IEEE 15th International Scientific Conference on Informatics, Poprad, Slovakia, November 20–22*, pages 415–420. IEEE. invited paper.

Steingartner, W. (2020). Support for online teaching of the semantics of programming languages course using interactive software tools. In *Proceedings of the International Conference ICETA 2020*.

Steingartner, W. (2021). On some innovations in teaching the formal semantics using software tools. *Open Computer Science*, 11(1):2–11.

Steingartner, W., Novitzká, V., and Schreiner, W. (2019). Coalgebraic operational semantics for an imperative language. *Computing and Informatics*, 38(5).

Vaclavkova, M., Kvet, M., and Sedlacek, P. (2019). Graphical development environment for object programming teaching support. In *INFORMATICS 2019 – IEEE 15th International Scientific Conference on Informatics, Proceedings*, pages 77–82. IEEE.

Zorvan, P. (2021). Byte-code generator for educational imperative language. Technical report, Technical University of Košice, Slovakia. (in Slovak).