Research Article

William Steingartner*

On some innovations in teaching the formal semantics using software tools

https://doi.org/10.1515/comp-2020-0130
Received Mar 12, 2020; accepted Apr 01, 2020

Abstract: In this work we discuss the motivation for innovations and need of a teaching tool for the visualization of the natural semantics method of imperative programming languages. We present the *rôle* of the teaching software, its design, development and use in the teaching process. Our software module is able to visualize the natural semantics evaluation of programs. It serves as a compiler with environment that can visually interpret simple programming language *Jane* statements and to depict them into a derivation tree that represents the semantic method of natural semantics. A formal definition of programming language *Jane* used in the teaching of formal semantics and production rules in natural semantics for that language are shown as well. We present, how the presented teaching tool can provide particular visual steps in the process of finding the meaning of well-structured input program and to depict complete natural-semantic representation of an input program.

Keywords: *Jane* language, innovation of teaching, language, natural semantics, parser, teaching software, university didactic, visualization

1 Introduction

The aims of an education system are that the learner must be able to understand the concepts and he must be able to apply these concepts in solving practical problems existing in society. Meeting the demands of current society is becoming complex due to rapid changes in technology. So, education techniques are required to be amended to meet the social and technological demands [1, 2]. In this context, the teaching of formal foundations of software engineering is nowadays a big challenge.

Formal methods are an integral part of the curriculum of computer science on many universities. In the context of computer science, formal methods refer to a variety of mathematical modeling techniques, which are used both to model the behavior of a computer system and to verify that the system satisfies design, safety and functional properties [3].

The course on Semantics of Programming Languages is taught as a graduate course in master study of Computer Science at the Faculty of Electrical Engineering and Informatics, Technical university of Košice, Slovakia. The course is a foundational one and obligatory in graduate study. There are many universities where the course with similar content is taught. Based on the bilateral international cooperation with Wolfgang Schreiner from the Research Institute for Symbolic computation, JKU Linz, Austria, we prepared under the project "Semantics technologies for computer science education" (Acronym: SemTech, Number of project: SK-AT-2017-0012) new teaching tools as a support for teaching the formal methods. The project was focused on the novel application of technologies which are based on the semantics of formal languages (programming languages, the language of predicate logic, etc.) to the education of university students in formal models of computer science.

One of the tools prepared under this cooperation was a module for visualizing the derivations in natural semantics of imperative programming languages. The idea was to bring an innovation of the existing course on Semantics – to extend and enrich the conventional teaching by using the interactive software which can help to understand better the principles of the mentioned semantic method.

It seems to be useful to show the future IT experts the advantages of formal methods and their fruitful usage in a process of practical software development [4–6]. One of the courses focused on formal methods in software engineering is a course on the Semantics of Programming Languages. In this course, mostly the methods focused on imperative (and often also functional or some domain-specific) languages are presented: operational semantics, denotational semantics and others.

***Corresponding Author: William Steingartner:** Technical University of Košice, Slovakia; Email: william.steingartner@tuke.sk

With some techniques, the teaching of formal methods can be more attractive and more understandable for students [7, 8]. For instance, formulating the modular structural operational semantics [9] where the descriptions of functional programs given in this method could be automatically translated into programs in the logic programming language Prolog, using the software provided to the students. Another approach was defining an action semantics [10] – a pure formal framework for describing the meaning of programs in textual phrases that are nearer to real programming languages and, moreover, quite easily understood by programmers that are not very familiar with mathematical methods.

A very fruitful method seems to be a visualization of processing the semantic method. We prepared more tools for visualization of some semantic methods, for instance, a tool for categorical semantics [11] (categorical denotational semantics we formulated in [12]), a tool for handling the mathematical expressions [13] or a tool for complex work with the abstract implementation of imperative language with an abstract machine for structural operational semantics [14]. Based on successful implementation in teaching process and the positive feedback from students, we have continued to prepare new tools.

The structure of a paper is as follows: in Section 2, we present some basic ideas and standard notions as a starting point. Furthermore, we show here the definition of a toy language *Jane* and we present how to define a natural semantics for this language. Section 3 presents the main motivation for implementing the innovations into a teaching process in the course on Semantics of Programming Languages and their outcomes. In Section 4, the structure of a teaching software is presented: we briefly show a methodological part of the design, grammar for parsing the input source codes. An example of using the program is also shown. More technical details about the implementation of the presented module can be found in [15]. The paper ends with a Conclusion section.

2 Theoretical background

In this paper, we are focused on building and presenting the teaching tool containing the parser of a simple toy programming language *Jane* and presenting on how a semantic tool for this language can be used in teaching process. The syntax of *Jane* is inspired and mostly adopted from the well-known toy language *While* [16], or sometimes referred also as *IMP* [17]. We start with the formal definition of language that serves as a background for preparing our software tool: for design and development and for teaching. Then we present abstract toy language *Jane* that is the subject of software tool processing. In the last part of this section, we briefly define natural semantics of imperative languages and we focus on semantics of *Jane* because the processing and visualizing of this method is a main subject of the presented software tool.

2.1 Language, syntax and semantics

Derivation of behavior of a command written in a computer language can be made with help of the exact definition of the language. Formal definition of a language consists of its syntax and semantics. The syntax of a language is considered as the set of rules that defines the combinations of symbols that are considered to be a correctly structured document or fragment in that language [18]. Syntax determines which character strings constitute well-formed programs. The programming language syntax determines the design and structure of programs written in some language [19–21]. Generally, the syntax is defined by

- a grammar – a set of rules that specify how input is structured;
- extended Backus-Naur form (BNF);
- an inductive definition.

Because syntax defines only the language structure, i.e. how it is allowed to classify individual constructions, it is necessary to use the second part of the language definition which is the specification of semantics. The semantics of a language describes the meaning (behavior) of a program in terms of the basic concepts of a language [22]. The only possibility of unambiguous writing of semantics is writing it using formal semantic methods.

The definition of formal semantics includes:

- semantic domains,
- specifications of semantic functions,
- semantic equations or deduction rules.

The semantic domain is a (mathematical) structure that contains the mathematical elements of a particular form representing meanings of elements from a given syntactic domain [21]. The sets serving as domains have a lattice-like structure [23]. For simplicity, we view these semantic domains as normal mathematical sets: basic sets (**Z** - sets of integers, **N** - natural numbers, **B** - Boolean values) and sets that arise by applying operations to these sets are very practical in the teaching process. The semantic function maps syntactic entities to the elements of semantic domain. Schematically, its specification is given as follows

[21]:

$$f : \textbf{Synt} \to \textbf{Sem}.$$

Semantic function is mostly given by:

- semantic equations (mainly for expressions), and
- production rules (for statements, see Section 2.3).

We determine in the production rules the meaning of each element (syntactic pattern/element) for a given syntactic domain [21]. For each syntactic domain, a unique semantic function is defined.

2.2 Formal definition of *Jane* programming language

In the previous section, we explained how a syntax and semantics of programming language are usually defined. In this section, we define the language that is used in teaching the course on the Semantics of Programming Languages. We present an abstract (non-real) programming language with language patterns belonging to an imperative paradigm. In the course on Semantics, we are focused on teaching the formal semantics of imperative languages. Therefore, abstract language with imperative patterns serves as a background. Its syntax is adopted from the well-known toy language *While* [16] and we refer to this language as *Jane*.

Now, we list the various syntactic categories (domains) and give a meta-variable that will be used to range over constructs of each category. For our language *Jane*, the meta-variables and categories are in the Table 1.

Table 1: Syntactic categories (domains) for the language *Jane*

Num	– for numeric strings
Var	– for variables
Expr	– for arithmetical expressions
Bexp	– for Boolean expressions
Statm	– for statements

The domains for numerals (**Num**) and variables (**Var**) have no internal structure from the semantic point of view. For the three remaining domains, particular production rules describing the syntax are defined.

For the arithmetic expressions, we formulate the following production rule:

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

where

- n denotes an integer numeral;
- x stands for a program variable;
- $e \bullet e$ represents an arithmetic operation that can be applied to arithmetic expressions (here in standard notation: addition (+), subtraction (–), multiplication (*));
- (e) represents an arithmetic expression enclosed into parenthesis.

The Boolean expressions of the language *Jane* are given by the following production rule. In the case of language *Jane*, their *rôle* is to provide a logical condition in a conditional or a loop statement.

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

where

- **false**, **true** represent syntactic forms of Boolean constants;
- $e = e$ represents an equality of arithmetic expressions;
- $e \leq e$ represents a relation "less then or equal" of arithmetic expressions;
- $\neg b$ stands for a negation of a Boolean expression;
- $b \wedge b$ is a conjunction of Boolean expressions;
- (b) represents a Boolean expression enclosed into parenthesis.

The language *Jane* contains five (standard) imperative constructs [21]:

- a variable assignment statement;
- an empty statement, used when there are no operations to perform in a context where a statement is required;
- a pattern of sequencing the statements – a statement list that consists of one or more statements written in sequence;
- a conditional statement with two mandatory ways of control-flow; and
- a loop statement that conditionally executes an embedded statement zero or more times.

Of course, for teaching purposes, the language *Jane* can be extended on syntactic level with new constructs and elements, like various types of loops, variables' declarations, or procedures, as well. In this approach, we work with standard D-constructs, without any extension.

The syntax of statements is given by the following production rule:

$$\begin{aligned} S ::= \quad & x := e \mid \textbf{skip} \mid S; S \\ & \mid \textbf{if } b \textbf{ then } S \textbf{ else } S \\ & \mid \textbf{while } b \textbf{ do } S. \end{aligned} \tag{3}$$

These five commands are considered as standard (basic) constructs of the imperative programming languages. They are also referred to as Dijkstra commands (or D-diagrams) [21]. Based on this abstract syntax, we can define standard imperative constructs in real imperative languages.

2.3 Natural semantics of imperative languages

We briefly express the basic properties of natural semantics of imperative languages: we list standard rules of this method and we show how this method is applied for the language *Jane*. We note that natural semantics [24] is considered as a hybrid of operational and denotational semantics that shows computation steps performed in a compositional manner. It is also referred as a "big-step semantics". This method has been proposed that is halfway between operational semantics and denotational semantics [25, 26]. Like structural operational semantics, natural semantics shows the context in which a computation step occurs, and like denotational semantics, natural semantics emphasizes that the computation of a phrase is built from the computations of its sub-phrases.

So, the main *rôle* of natural semantics is to define the relationship between the initial state before executing the language statement and the final state after executing this statement. The meaning of the statement (its semantics) is therefore considered as a change of memory state. Natural semantics does not follow the detailed execution of the statement; it is focused on changing the state that occurs by executing the entire statement, as it expresses the transition relation in natural semantics [21, 25]:

$$\langle S, s\rangle \to s'.$$

The general form of the production rule of natural semantics looks as follows [21]:

$$\frac{\langle S_0, s_0\rangle \to s_1, \ldots \langle S_n, s_n\rangle \to s}{\langle S, s_0\rangle \to s} \ (rule_{\text{ns}})$$

where

- S stands for a statement, possibly consisting of one or more statements written in sequence $S_1, \ldots, S_n$;
- s_0 is an initial state;
- s is a final state;
- $s_1, \ldots, s_{n-1}$ are states during the particular steps;
- in the notation $(rule_{\text{ns}})$ of a rule, *rule* represents name (or numeric notation) of the rule and an index "ns" indicates that it is a rule of natural semantics.

Notation of a general production rule $(rule_{\text{ns}})$ expresses the natural semantics of the statement S as a one-step change from the initial memory state s_0 to the final state s [21]. Now, we briefly introduce the rules of natural semantics for the statements of the language *Jane* listed in (3).

The assignment statement is defined by the axiom

$$\frac{}{\langle x := e, s\rangle \to s\,[x \mapsto \mathscr{E}[\![e]\!]s]} \ (1_{\text{ns}})$$

The rule (1_{ns}) above notation represents the meaning of the assignment command: the variable x is assigned to the value of an arithmetic expression e (implicitly typed as integer) calculated in the state s and a memory state is being actualized from s to s'.

An empty statement is defined by the axiom:

$$\frac{}{\langle \textbf{skip}, s\rangle \to s} \ (2_{\text{ns}})$$

Here, the state is not being changed after the execution of a statement.

The list of statements is defined by the following rule:

$$\frac{\langle S_1, s\rangle \to s'' \quad \langle S_2, s''\rangle \to s'}{\langle S_1; S_2, s\rangle \to s'} \ (3_{\text{ns}})$$

This rule expresses that statements are executed in particular steps with step-wise computing and passing the actual memory state.

The conditional statement is defined by the following two rules:

$$\frac{\langle S_1, s\rangle \to s' \quad \mathscr{B}[\![b]\!]s = \textbf{tru}}{\langle \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2, s\rangle \to s'} \ (4^{\textbf{tru}}_{\text{ns}})$$

$$\frac{\langle S_2, s\rangle \to s' \quad \mathscr{B}[\![b]\!]s = \textbf{fls}}{\langle \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2, s\rangle \to s'} \ (4^{\textbf{fls}}_{\text{ns}})$$

For the conditional statements, two rules based on the value of a Boolean conditions are defined. They are symmetric and their use is straightforward.

The loop statement is defined by the following two rules:

$$\frac{\langle S, s\rangle \to s'' \quad \langle \textbf{while } b \textbf{ do } S, s''\rangle \to s' \quad \mathscr{B}[\![b]\!]s = \textbf{tru}}{\langle \textbf{while } b \textbf{ do } S, s\rangle \to s'} \ (5^{\textbf{tru}}_{\text{ns}})$$

$$\frac{\mathscr{B}[\![b]\!]s = \textbf{fls}}{\langle \textbf{while } b \textbf{ do } S, s\rangle \to s} \ (5^{\textbf{fls}}_{\text{ns}})$$

A while loop allows code to be executed repeatedly based on a given Boolean condition zero or more times.

Generally, a program in *Jane* is considered as a sequence of statements, i.e. one compound statement. For

simplicity, we can denote the whole program by one statement (meta-)variable, e.g. P. Deriving the semantics of a program P we start in an initial state, e.g. s_0 and we construct a transition relation to a final state s. Such defined relation is a root of a derivation tree in natural semantics of a program P:

$$\langle P, s_0 \rangle \to s.$$

Then

- the transition of the program will form the root of the tree;
- by applying the rules of natural semantics, we create inner vertices;
- the leaves will form the axioms of natural semantics.

By step-wise applications of appropriate semantic rules, we construct a full derivation tree of a program in natural semantics.

3 Motivation for bringing innovations to teaching

The course on Semantics of Programming languages has been taught by conventional teaching methods which include lecturing and face to face interaction in a classroom. Lecturing method is based on a one sided input i.e. from teacher. The teacher delivers the content to learner and a learning level can be measured with the help of written examination.

We present a contribution to teaching modernization whose focus is the innovation of the existing course on Semantics of Programming Languages which is taught at the Faculty of Electrical Engineering and Informatics for graduate degree in the study program Computer Science. The main idea is to bring innovations based on presenting and visualizing the formal principles of programming languages and their semantics in more understandable and attractive way. For this purpose, we have started to prepare a project – software package consisting of modules for visualizing and simulating particular semantic methods (under the SemTech project).

The package is aimed at improving and making the teaching methods applied to the lecture and laboratory part of the course more attractive. Our goal is to apply modern software solutions in this course as a significant help in teaching the lectures and laboratory exercises and to support self-study and to apply the given software solution for the evaluation part of the course (tests, exams). Presented software module will allow an illustrative and understandable use of semantic method. In this way, we will achieve greater clarity in the applied procedures and principles in the course teaching.

Specifying the semantics of a programming language is a more difficult task than specifying its syntax. Formal, mathematical methods are precise, but they are also complex and abstract, and require study to understand. Different formal methods are available, with the choice of method depending on its intended use [27]. After some years of teaching the course on Semantics, we identified the points in which we can bring innovations and make this course more attractive for students and for young IT experts. We revised the content of the course to reflect the current state of the art in the world (mostly oriented to modern technologies). We designed and developed some new modules of teaching software [11, 13, 14]. Based on the extension and adaptation of the content of the course, a new textbook with supplementary materials was prepared. The using of this book assumes the coordination with particular modules of our software package. We note that standard teaching methods - explanation and model calculation using board where the *rôle* of the educator/teacher is indispensable cannot be so simply omitted. We think that modern visualizing methods can significantly help and we want to apply them but we do not want to avoid using standard methods as well [28].

The practical outcomes of using the software package will be used in teaching computer science courses in the field of software engineering focusing on the design and development of correct programs and systems, not only at the domestic university and at a cooperating university, Johannes Kepler University in Linz, Austria, or other universities where related and similar courses are a part of curriculum. One of the advantages can be also putting this software into practice for distance learning. Our software tool (or the complete software package) can serve as a modern interactive learning tool, as a support for new teachers of the course, as a tool for IT experts using formal methods or for other interested experts.

The proposed teaching software can help teachers and educators in providing a better and illustrative form for the students:

- during the lectures, the teacher can present examples directly and interactively; or at least use prepared examples depicted on screen-shots;
- at laboratory exercises and seminars, examples can be explained step-wise with possible change of input parameters to show the differences in programs' simulations.

The teaching software can be very useful for students especially in the following cases:

– during the laboratory work for simulation of program execution, when examining the conditions of how results will program produce based on input parameters;
– in the phase of self-study and self-preparation for testing or exams;
– when doing research or simulations when the visual output is needed.

Moreover, the program provides visual output that can be stored into graphic file. Another option is to export the LaTeX source of the produced visual output that can be used in other projects. We consider both output forms as very important for future work in studying, teaching and preparing output materials.

The theoretical outcomes of using the software package can find their application in the field of further research on the issue of interactive and experiential teaching of theoretical principles in computer science.

In addition, the most important contribution to social, industrial and economic practice, we consider an increasing of the professionalism of the young IT experts in the field of formal methods for software engineering and their potential and attractiveness in the labor market. Last but not least, we expect a raising of interest in formal methods and deepening skills of young IT experts.

4 Software tool for natural semantics

As a kind of modern support of teaching in the field of formal methods, we developed a software module for the process of visualization of natural semantics. The software is designed to help students understand better how to find the meaning of a code written in a simple toy programming language with the semantic method of natural semantics. Since the program reads an input source in *Jane*, it has been designed and developed simply as a compiler transforming the source code into semantic-driven visual output form.

4.1 Implementation of the visualization

The visualization has been implemented as a unit that takes an output provided by a compiler of a *Jane* code. We follow the basic idea that a compiler is a program that reads a program written in a source language and translates it in to an equivalent form in another language, called target language or target representation [29, 30]. In our case, a source program is written in a language *Jane*. As a target form, visual representation of program is provided: a semantic-driven simulation of program execution by step-wise application of concrete semantic rules.

The compiler is designed with its standard phases:

– the lexical analysis, which reads the input string from the code and produce tokens;
– syntax analysis – it analyzes the syntactical structure of the given input; and
– semantic analysis, which judges whether the syntax structure constructed in the source program derives any meaning or not.

A grammar that represents the language *Jane* is the following:

$$
\begin{aligned}
&commands \rightarrow sub\_commands\ ;\ \{sub\_commands\ ;\}\\
&sub\_commands \rightarrow\ assign \mid statement \mid cycle\\
&sub\_commands \rightarrow EPSILON\\
&assign \rightarrow VARIABLE\ ASSIGNMENT\ expr\\
&assign \rightarrow EPSILON\\
&statement \rightarrow IF\ (expr)\ then\ commands\\
&\qquad\qquad \{else\ commands\}\\
&statement \rightarrow EPSILON\\
&cycle \rightarrow WHILE\ (expr)\ do\ commands\\
&cycle \rightarrow EPSILON\\
&expr \rightarrow sig\_operator\ log\\
&log \rightarrow PM\ operator\ log\\
&log \rightarrow EPSILON\\
&sig\_operator \rightarrow PM\ operator\\
&sig\_operator \rightarrow operator\\
&operator \rightarrow sig\_term\ sum\\
&sum \rightarrow PM\ term\ sum\\
&sum \rightarrow EPSILON\\
&sig\_term \rightarrow PM\ term\\
&sig\_term \rightarrow term\\
&term \rightarrow factor\ product\\
&product \rightarrow MD\ sig\_factor\ product\\
&product \rightarrow EPSILON\\
&sig\_factor \rightarrow PM\ factor\\
&sig\_factor \rightarrow factor\\
&factor \rightarrow value\\
&factor \rightarrow NEG\ factor\\
&factor \rightarrow O\_BR\ expr\ C\_BR\\
&value \rightarrow NUM\\
&value \rightarrow VAR
\end{aligned}
$$

The compiler provides as its output the binary version of an input source – a byte-code. This form is used only in-

ternally, i.e. the compiler translates the source into the sequence of elements that are ready to provide a visual form – a tree in natural semantics. Visualizing mode is run after the compilation is successfully finished: it depicts the derivation tree of an input program in natural semantics by reading the provided compiled byte-code and interpreting it step-by-step. Before starting the visualization, the user is asked to put the values of input variables. After this step, the semantic implementation is ready to be visualized (depicted). Visualizing mode practically simulates particular steps of an input program with real (concrete) input values. When an input program was correct (at least without the logical errors), the program shows at the end of simulation a complete derivation tree in natural semantics with all memory states (the new variable values) that have occurred during the program simulation.

The program allows to input of the source code manually or by loading it from a file. For the *Jane* source codes, standard syntax according to the rules (1), (2) and (3) is used. The program recognizes the key words and supports the syntax-highlighting. We note, that for easier manipulation with the source and better readability:

- we enriched the blocks in a loop statement and a conditional statement with the keywords (textual "brackets") **begin** and **end**; and
- we allowed the use of C-like syntax for Boolean expressions (in contrast to rule (2)): '==' for equality of expressions, '<=' for less-or-equal symbol, '!' for negation and '&&' for logical conjunction.

The final step in implementation is to design a graphical interface. The interface was implemented using external *JavaFX*[1] which is a part of the Oracle JDK 9 implementation of Java, and *JLaTeXMath* libraries[2]. Using the first one, components were created to interact with the user, and the second allowed to render a graphical representation of the specified source code on the tree of natural semantics.

The resulting GUI look can be seen in Figure 1.

At the end of a simulation, the provided visual output can be stored into separate file as a picture (in one of `PNG` or `JPG` formats) for its future use. Moreover, the depicted derivation tree is displayed using the LaTeX style. The user can export and store the LaTeX source code of the displayed derivation tree into separate `TEX` file.

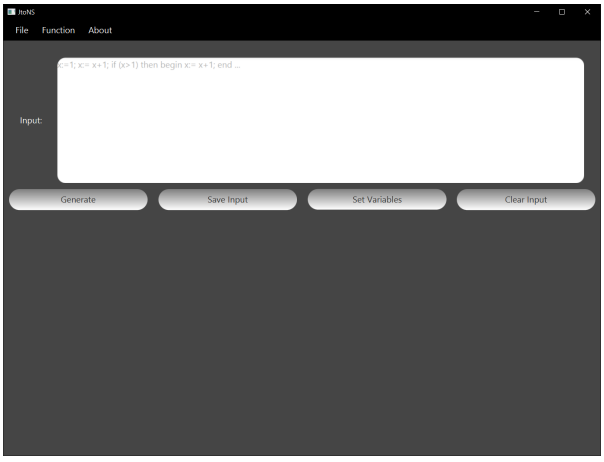


Figure 1: Graphical User Interface of visualizing tool

4.2 Graphic user interface of a program

In this section, we briefly present how this software can be used. When launched, the main application window appears, which contains the components to interact with. At the top of the window, we can see the menu bar where the user can work with all the features that the application offers. Right below this panel, there is a text area into which input code can be put. Below this text area is a set of buttons that perform the following tasks:

- *Generate* button – it calls the main program method and generates an application output.
- *Save Input* button – it is used to store an input to a user-selected location in the computer memory.
- *Set Variables* button – it allows user to set variables before the output is generated.
- *Clear Input* button – it serves for clearing the text area.

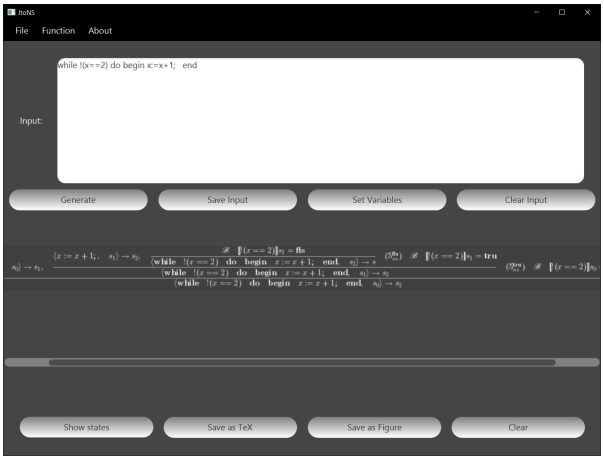


Figure 2: Application after output generation

1 https://docs.oracle.com/javafx/2/overview/jfxpub-overview.htm
2 https://github.com/opencollab/jlatexmath

After entering the input string in the form of a *Jane* code and possibly setting the variables and pressing the *Generate* button, a new component appears at the bottom of the screen. This component contains a canvas to render the resulting natural semantic tree and a set of buttons. The set consists of the following buttons:

- *Show States* button – it displays a pop-up window in which the table shows the states that the program has passed during execution.
- *Save as TeX* button – it saves the latex-code output string to a text file on a user-selected location in the computer memory.
- *Save as Figure* button – similar to the previous button, it saves the result to the computer's memory but in the image format of the user's choice. Supported formats are `JPG` or `PNG`.
- *Clear* button – it removes the entire component that was created after pressing the *Generate* button.

Accessing these functions is also possible from the upper menu bar.

4.3 Example of using the application

As an example of using this program, we show how our software tool produces a visual output for the Euclidean algorithm finding the greatest common divisor of two given numbers.

In the text area, we insert the program realizing the Euclidean algorithm in the form of *Jane* code. We write the program into the text area, then we set the initial values for variables x and y to **96** and **64**, resp.

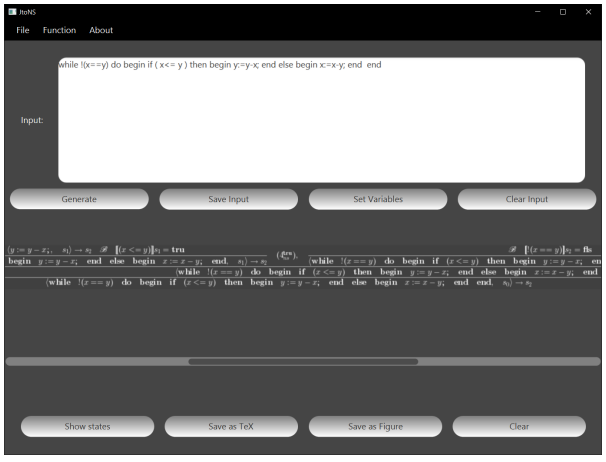
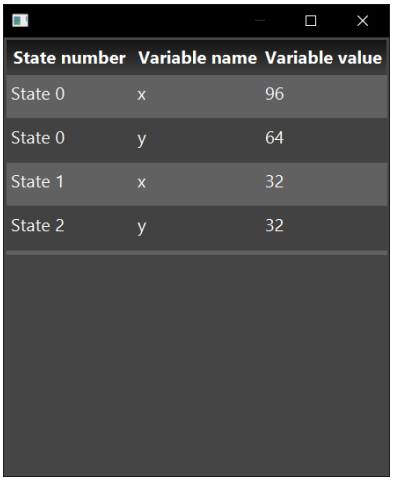


Figure 3: Example of using the tools with implementation of Euclidean algorithm

The following is an example of a program given as an input:

while !(x == y) **do begin**
if (x <= y) **then begin** y := y – x **end**
else begin x := x – y **end**
end

The result can be seen in Figure 3. Moreover, we can see the states that occurred during the program functioning in the table. When the user recalls the states (using the *Show States* button), the table appears with all the states (Figure 4).



State number	Variable name	Variable value
State 0	x	96
State 0	y	64
State 1	x	32
State 2	y	32

Figure 4: Memory states during the computation of Euclidean algorithm

5 Conclusion

In this paper, a software tool that allows visualizing the construction of trees in natural semantics of imperative programming languages was presented and discussed. Moreover, its need and *rôle* in teaching process was emphasized. The method of natural semantics is implemented on a toy (non-real) language *Jane* used for teaching the principles of languages and the semantics in many applications. A developed software allows the visualization of the semantic method and allows the quick understanding and the easy reading of its results.

Our tool for semantic-driven visualizations fully supports all standard imperative constructs - variable assignment, sequencing of statements, conditional and loop statement. As an added value, the program allows the users to store the state table into a text file, to store a source

code of an input program after modifications, to store the picture of a depicted tree and to produce a source code of a constructed tree for LaTeX. A presented software tool is planned to be integrated into a larger software package for teaching the semantic methods. The main motivation for creating this tool was mainly help for students and young IT experts to get more familiar with formal methods grounded in the field of semantics of programming languages.

Acknowledgement: This work has been supported in the frame of the initiative project "Semantic Modeling of Component-Based Program Systems" under the bilateral program "Aktion Österreich — Slowakei, Wissenschafts- und Erziehungskooperation" and by the project KEGA 011TUKE-4/2020: "A development of the new semantic technologies in educating of young IT experts".

References

[1] Dostál J., Wang X., Steingartner W., Naungchalerm P., Digital Intelligence - New Concept in Context of Future School of Education, Proceedings of International Conference of Education, Research and Innovation – ICERI 2017 Conference (16th–18th November 2017, Seville, Spain), available at SSRN: https://ssrn.com/abstract=3255366

[2] Jhanji K., Kumar A. R., Modernization in Teaching Learning Process, In Innovative Teaching Practices for 4G students, IOR International Press, 2019, 105–109

[3] Sotiriadou A., Kefalas P., Teaching Formal Methods in Computer Science Undergraduates, 2000, unpublished, available online

[4] Dostál J., Wang X., Naungchalerm P., Brosch A., Steingartner W., Researching computing teachers' attitudes towards changes in the curriculum content — an innovative approach or resistance? In: 2017 Second International Conference on Informatics and Computing – ICIC 2017 (2017, Jayapura, Indonesia), IEEE, New York, 2017, 1–6

[5] Novitzká V., Logical Reasoning about Programming of Mathematical Machines, Acta Electrotechnica et Informatica, 2005, 5(3), 50–55

[6] Steingartner W., Radaković D., Novitzká V., Eldojali M. A. M., An analysis of some aspects of component-based programming for selecting appropriate categorical structures as their models, Acta Electrotechnica et Informatica, 2017, 17(2), 3–10

[7] Bilanová Z., Perháč J., About possibilities of applying logical analysis of natural language in computer science, Proceedings of IEEE 13th International Symposium on Applied Computational Intelligence and Informatics – SACI 2019 (29th–31st May 2019, Timişoara, Romania), IEEE, Danvers, 2019, 256–260

[8] Mihályi D., Peniašková M., Perháč J., Mihelič J., WEB-Based Questionnaires For Type Theory Course, Acta Electrotechnica et Informatica, 2017, 17(4), 35–42

[9] Mosses P. D., Teaching Semantics of Programming Languages with Modular SOS, In: Proceedings of the 2006 Conference on Teaching Formal Methods: Practice and Experience, Series TFM'2006, BCS Learning & Development Ltd., Swindon, UK, 2006

[10] Mosses P. D., Theory and Practice of Action Semantics, BRICS Report Series RS9653, University of Aarhus, Aarhus, Denmark, 1996

[11] Steingartner W., Perháč J., Biliński A., A Visualizing Tool for Graduate Course: Semantics of Programming Languages, IPSI BgD Transactions on Internet Research, 2019, 15(2), 52–58

[12] Steingartner W., Novitzká V., Bačíková M., Korečko Š., New approach to categorical semantics for procedural languages, Computing and Informatics, 2017, 36(6), 1385–1414

[13] Steingartner W., Yar-Muhamedov I., Learning software for handling the mathematical expressions, Journal of Applied Mathematics and Computational Mechanics, 2018, 17(2), 77–91

[14] Kochaníková Ž., Steingartner W., Eldojali M. A. M., A code generator for an abstract implementation of imperative language, In: Electrical Engineering and Informatics VIII : Proceedings of the Faculty of Electrical Engineering and Informatics of the Technical University of Košice, 2017, 342–347

[15] Steingartner W., Haratim M., Dostál J., Software visualization of natural semantics of imperative languages – a teaching tool, In: Proceedings of the 2019 IEEE 15th International Scientific Conference on Informatics – Informatics 2019 (20th–22nd November 2019, Poprad, Slovakia) IEEE, Danvers, 387–392

[16] Nielson H. R., Nielson F., Semantics with Applications: An Appetizer (Undergraduate Topics in Computer Science) 2007th Edition, Springer, 2007

[17] Roşu G., K – A Semantic Framework for Programming Languages and Formal Analysis Tools, In: D. Peled and A. Pretschner (eds.), Dependable Software Systems Engineering, Series NATO Science for Peace and Security, IOS Press, 2017

[18] Perháč J., Mihályi D., Novitzká V., Between syntax and semantics of resource oriented logic for IDS behavior description, Journal of Applied Mathematics and Computational Mechanics, 2016, 15(2), 105–118

[19] Dedera L., Computer languages and their processing, Armed Forces Academy, Liptovský Mikuláš, Slovakia, 2014 (*in Slovak*)

[20] Gabbrielli M., Martini S., Programming languages: principles and paradigms, Springer-Verlag London, 2010

[21] Novitzká V., Steingartner W., Semantics of Programming Languages, Technical University of Košice, Košice, Slovakia, 2015 (*in Slovak*)

[22] Waite W. M., Goos G., Compiler Construction, Series: Texts and Monographs in Computer Science, Springer-Verlag, 1984, reprint 1996

[23] Slonneger K., Kurtz B. L., Formal Syntax and Semantics of Programming Langauges: A Laboratory-Based Approach, Addison-Wesley, Reading, Massachusetts, 1995

[24] Kahn G., Natural semantics, In: Brandenburg F.J., Vidal-Naquet G., Wirsing M. (eds) STACS 87. STACS 1987. Lecture Notes in Computer Science, vol. 247, Springer, Berlin, Heidelberg

[25] Benčík M., Dedera L., Natural Semantics of Battle Management Languages, In: Proceedings of the 2019 Communication and Information Technologies – KIT, (9th–11th October 2019, Vysoké Tatry, Slovakia), IEEE, 2019

[26] Schmidt A. D., Programming language semantics, In: Encyclopedia of Computer Science, John Wiley and Sons Ltd., Chich-

ester, UK, 2003, 1463–1466
[27] Louden K., Lambert K., Programming languages – Principles and Practice, Third edition, Cengage Learning, USA, 2011
[28] Teplická K., Steingartner W., Matvija M., Innovative didactic methods in the teaching process at universities. Technical University of Košice, 2020, (*in Slovak*).
[29] Aho A. V., Ravi S., Ulman J. D., Compilers, principles, techniques, and tools, Addison-Wesley Publishing Company, 1987
[30] Kollár J., Compilers, elfa s.r.o., Košice, Slovakia, 2010 (*in Slovak*)