Sveučilište u Rijeci
University of Rijeka
http://www.uniri.hr

Polytechnica: Journal of Technology Education, Volume 5, Number 2 (2021)
Politehnika: Časopis za tehnički odgoj i obrazovanje, Volumen 5, Broj 2 (2021)

Politehnika
Polytechnica
http://www.politehnika.uniri.hr
e-mail: cte@uniri.hr

DOI: https://doi.org/10.36978/cte.5.2.1

Izvorni znanstveni rad
Original scientific paper
UDK: 004.423
004.432.2

The visualization of a graph semantics of imperative languages

Erik Gajdoš, William Steingartner

Faculty of Electrical Engineering and Informatics

Technical University of Košice

Letná 9, 042 00 Košice, Slovakia

erik.gajdos.3@student.tuke.sk, william.steingartner@tuke.sk

Abstract

This work aims to present the software support for teaching in the field of formal semantics of imperative programming languages. The main part focuses on a software tool that provides a visual representation of the individual steps of the calculation in categorical semantics, which can also be referred to as graph semantics. The use of software tools in teaching to visually represent computational steps considerably facilitates understanding by students and can also serve as a good basis for supporting distance learning. Our program works in the standard form: after reading the correct user input, a visual representation of the meaning of the program is generated in the form of a category of states, which is displayed as an oriented graph. For better extensibility, the program is implemented as a web application.

Keywords: *categorical semantics, compiler, semantics of languages, university didactics, visualization, web application.*

1 Introduction

In the process of developing new applications and systems, it is necessary to know how the program is performed. To describe this aspect of a code and a whole formal description of programming languages, the methods grounded in the semantics of programming languages are very fruitful. There are several types of semantic approaches to programs (Nielson and Nielson, 2007), based on current requirements. One of the main roles of semantics is to predict the behavior and output of program execution.

The education of young IT experts must also follow current trends in computer science and information technologies (Herceg et al., 2019). Therefore, in our opinion, it is essential that the formal foundations, which make it possible to abstract and formally prove several procedures, be part of the curriculum for informatics (Reichl and Schreiner, 2020). Many of these formal methods are based on the formal semantics of programming languages. Therefore, we consider the development and use of visualization tools that enable static or dynamic visualization of semantic procedures as a helpful and innovative element in the modernization of education in the field of basics of software engineering (Steingartner, 2021).

In this paper, we focus on a software tool that enables static visualization of categorical denotational semantics on which we refer also to as graph semantics. Moreover, this software will be integrated into planned future software package that will enable to visualize several semantic methods and to help in education process. We present our motivation for the implementation and deployment of the software in the teaching process as well as the methodological

design of the application. Furthermore, we focus on the main role of the application, the description of the architecture, how the program was developed and the main user requirements for the target (intended) functionality of the application. We note that the purpose of this article is not to present instructions on how to work with the application and thus replace or extend the user manual.

The structure of the paper is as follows. In Sect. 2, the basic concepts for our approach and necessary preliminaries are introduced. Sect. 3 is focused on the final architecture. Sect. 4 describes the methods for processing and compiling the input code and its transformation to the output representation and Sect. 5 describes the main points about the technical realization and implementation. In Sect. 6, we briefly present the functionality and work of the application on a simple example. Finally, Sect. 7 then concludes our paper.

2 Preliminaries and basic concepts

In this section, we present the basic concepts and theoretical foundations necessary to introduce the content of the researched parts.

2.1 Categories and categorical semantics

Our approach to categorical denotational semantics was introduced and defined in the paper (Steingartner et al., 2017). This method is the new approach for describing the semantics of programming languages. Its foundations are based on standard denotational semantics. This type of semantics uses mainly mathematical elements. For describing the program, a category of states is constructed.

Mathematical category theory serves as the basis for defining semantics. Because the definition of categories is well known, we provide only the necessary basics in this text. For further details, we refer the reader to, for example, Category (Barr, Wells, 1990).

A category is a structure consisting of objects $(A, B, C, \ldots)$ and arrows between them $(f: A \rightarrow B, g: B \rightarrow C$, etc.). For each object, an identity morphism exists (e.g. $id_A: A \rightarrow A$); and the composition of morphisms must hold – for two arrows $f: A \rightarrow B$ and $g: B \rightarrow C$, there must be an arrow (their composition) that goes from A to C:

$$g \circ f: A \rightarrow C.$$

A morphism is visually understood as an oriented edge in the graph. This is an initial point in this research when we consider that an edge models the execution of a statement. Program statements that can be considered as morphisms are those statements that cause a change of program state. In contrary to the denotational semantics of functional languages which is a well-known method, the research in this area lacks a categorical definition of imperative and procedural languages. In such languages, a state is a foundational notion. For purposes of standard semantic methods, the state is defined as a function that assigns to a variable its value (Nielson and Nielson, 2007). Then the state is an element of a set of states, a semantic domain for the given semantic model mostly represented as a function space. An environment that expresses the context dependencies known from operational semantics (Plotkin, 2004) is now a part of category objects and is given by the level of nesting. Hence each state is represented as a function, that assigns to a variable on a given level of nesting its value (Steingartner et al., 2019). These states are objects in the category of states. We note that a particular graph that expresses a path in a program as its meaning represents a single program execution for specific (specific) input values.

2.2 Categories in the background

Categories have a great power to express dependencies and properties graphically and in a very easy and elegant way (Brandenburg, 2016; Perháč et al., 2017; Walters, 1992). This is the main motivation for graphical visualizing of the categorical semantics. During the program execution, each statement has access to actual values of variables, which are stored in the current state of the program. The state of a program is changed in case if an actual statement modifies the value of some variable. The semantics of this kind of statement is a function that provides new state *s'* based on actual state *s*. A variable assignment statement can be considered as this type of statement because it changes the value of a variable. The mathematical definition of this function is

$$[\![S]\!]: s \rightarrow s',$$

where *S* stands for statement and *s*, *s'* for states. The order of the execution of the statements is as they are written.

A sequence of the statements which modify the state of the program can be visualized separately or as one composite function

$$[\![S_1; S_2]\!] = [\![S_2]\!] \circ [\![S_1]\!],$$

where S_1 is the first statement and S_2 is the second statement. The execution of commands is sequential (similar to the composition of the corresponding functions) and each function continues the calculation

in the state that is the result of the previous function. In this situation, the statement S_1 in state s produces new state s' and then the second statement S_2 is executed in the state s' which produces the final state s'' of the sequence. Definition of the sequential execution in state s is:

$$[\![S_1; S_2]\!]s = [\![S_2]\!]([\![S_1]\!])s.$$

A graphical representation of the sequential execution is in depicted Figure 1. Generally, the path in a graph (a composite morphism in a category, the compositionality property in category) from the initial state to the final state represents the semantics of the program for which the categorical model is constructed.

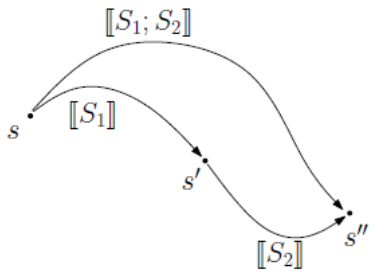

Figure 1. Sequential and chain execution of statements (Steingartner et al., 2017)

The category of states has the property that it also contains two special objects – initial and final object. Because each program execution must begin in some initial state, based on the properties of a category, the initial state is the initial object of a particular category of states. Similarly, the object representing the error and the immediate termination of the program execution is the final object of the category of states (there is exactly one unique morphism from each object to the end object).

2.3 Language Jane for defining the semantics

As a modeling language, we present a simple imperative language named Jane. The language Jane (Steingartner et al., 2019) is an abstract language embodying a tiny core fragment of conventional mainstream languages such as C and Java. We note that this concept of abstract imperative language is well-known and is also mentioned as language *While* or *IMP*, presented e.g., in (Nielson and Nielson, 2007; Roșu and Șerbănuță, 2010). We adopted the structure of this language and for pedagogical reasons, we refer to this language as Jane (it is an acronym for the Slovak name *JAzyk Na Edukáciu* – a *language for education*). In addition, we note that this abstract language is widely used in teaching the formal foundations of languages, syntax, and semantics, as well as in research in verifying and proving the various properties of imperative languages with a subsequent transfer to a particular (real) language.

This language embodies also standard arithmetic and Boolean expressions. We assume implicit typing for arithmetic expressions - all arithmetic expressions are of type integer. The syntax of the language Jane is given by the following rules in EBNF.

Syntax of arithmetic expressions:

$$e ::= n \mid x \mid e + e \mid e - e \mid e * e \mid (e),$$

where n stands for numeral and x for a variable. Syntax of Boolean expressions:

$$b ::= true|false|\neg b|b \wedge b|e = e|e \leq e|(b).$$

In language Jane, the Boolean expressions are evaluated in conditional and loop statements.

In the current version of the language, we work with expressions that are listed in the syntax. For example, we do not use integer division for arithmetic expressions, and we do not use some other relational operators or disjunction for Boolean expressions. However, if necessary, we can also express some logical connectors using existing ones.

Generally, the syntax of the expressions is not closed and can be extended as needed. Associated with this is the need to extend the language specification at the software level as well. So if we change the syntax of expressions, we need to extend the grammar of the language and add new rules to the compiler.

Finally, the syntax of the statements is given by the following rule:

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

In this work, we present software that can visualize the mentioned type of semantic with the usage of mentioned imperative language. This system is implemented as part of the research under the project cited in the Acknowledgment section and documented in (Gajdoš, 2021).

3 Processing and compiling of the input code

The first point is to figure out how the input code from the user is processed. For this purpose, we decided to create a compiler (Appel, 2002) of the input code. Input code is written in the Jane language which is translated to the JavaScript language. As an

output language was chosen JavaScript because this language does not need to be compiled anymore since it is interpreted language. Another aspect of choosing this solution is that we needed to simulate the execution of the input code for its next visualization and to see how the program works. Another reason for making the compiler is that Jane language is not a regular type of language, hence, using context-free grammar with the appropriate compiler was the best solution.

In general, the compiler of the programming languages consists of two parts (Aho, 2006; Dedera, 2014): analysis and synthesis.

(1) Analysis – this part analyzes the input code on the lexical, syntactic, and semantic sides. The lexical analysis creates a sequence of the tokens where each token has its meaning. Based on the output of lexical analysis syntactic analysis, or in another meaning parsing of the input code makes tree structure of the sequence of statements depends on the defined language grammar. This part also controls if were used only allowed data types. The last analysis of this part is the semantic analysis which controls the consistency of created tree by semantics if were used only defined variables.

(2) Synthesis – generates the output program.

The next step is to define language grammar for syntactic analysis. The grammar of the language is written in the Extended Backus-Naur form (EBNF). This grammar is presented in Table 1. Each row of this table describes one rule of the language grammar. All terminal symbols which represent characters are written in quotation marks, keywords, an indication of variables and values are written in italic with the noncapital first letter. Nonterminal tokens are written in italic with the capital first letter.

Program	→	*Stat_seq* ”EOF”
State_seq	→	*Stat* { *Stat* }
Stat	→	*var* ”:=” *Expr* ”;” | *if* ”(” *Log_Exp* ”)” *then Body* [*else Body*] | *while* ”(” *Log_Exp* ”)” *do Body*
Body	→	”{” *Stat_seq* ”}”
Log_Exp	→	*Comparison* { ”∧” *Comparison* }
Comparison	→	*Log_Term* (”=” | ”≤”) *Log_Term*
Log_term	→	*Expr* | [” ¬”] ”(” *Log_Exp* ”)”
Expr	→	*Mul* { (”+” | ”-”) *Mul* }
Mul	→	*Term* { ”*” *Term* }
Term	→	*var* | *val* | ”(” *Expr* ”)”

Table 1. Language grammar for the compiler in EBNF

Our implementation of the compiler is based on listed rules. As the first step, the lexical analysis which recognizes used variables and stores them for the next initialization is performed. The expected input for lexical analysis is an input code that produces an array of identified variables. Before performing the next step, syntax analysis, it is possible to define a default value for each variable. Syntax analysis works with the defined grammar and an array of variables with their default values. A standard error-recovery algorithm during the compilation is performed: if during this analysis some error occurs, is this error logged and compiling process continues with the next symbol. After this analysis, the output code is generated if the actual sequence of tokens is syntactically correct. Before the generating of the output code in the syntactic analysis is a written statement for initialization of the identified variables with their values. Furthermore, the visualization process is initialized. After these two steps of initialization, compiler starts to translate the sequence of the input statements. When the compiler identifies the assignment statement it writes this statement and function for updating the output object with information about the new state and the nesting level of this statement for better visualization information. Generating of the output code finishes when the generator reaches the EOF symbol. After the token generation process is complete, a semantic analysis is performed. If some exception occurred during the compilation, the access code is marked as invalid. This process of generating output code is presented in Figure 2.

4 Architecture of the application

We decided to design and develop this system as a web application. At the first, we needed to identify the requirements of the system. The first requirement is that the user should be able to provide an input code. Another requirement is that the user should be able to set the default values for used variables in the input code. It depends on these two requirements we needed to design and implement the compiler which will be able to compile input code and simulate the behavior. The next requirement is that after compiling and simulating the input code, a graph depicting the meaning of the input program shall be rendered and provided as an output. Additional requirements that we identified is that it user can save written code, list the existing codes and simulate its behavior. The last requirement which we identified is that the user should be able to download the rendered output graph.

Depending on the listed requirements and the decision that this system is implemented as a web application, we split the system into three separate levels. The design of the application as the conceptual model is shown in Figure 3. The first level of the application is the user interface or the visualization

level which is used for reading the input from the user and for rendering the output graph. The second level is the server or application level which handles all requests from the user and reacts to them.

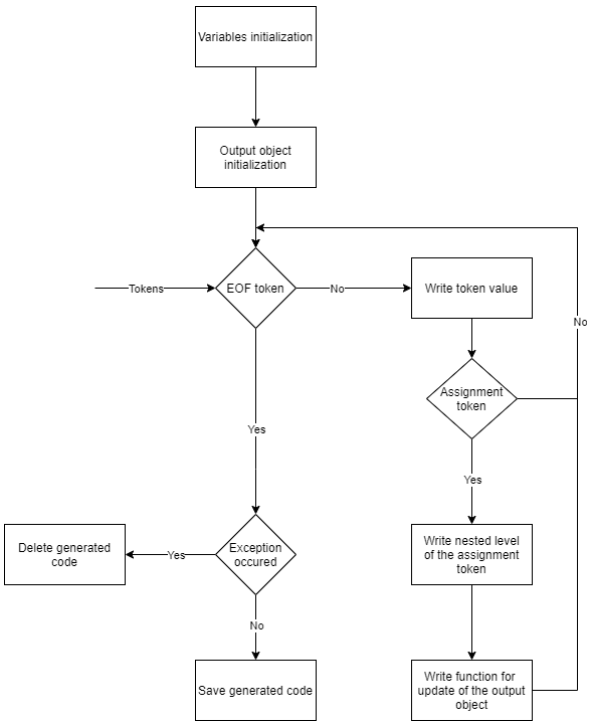

Figure 2. Process of generating output code

Communication between the first and second levels is implemented by REST API calls. This level also communicates with the third level on which a database is implemented. The database level stores all information about users and their saved codes.

The resulting multi-level architecture meets modern and current standards for web application development. It enables a higher level of modularization and containerization, which also facilitates the maintenance of the application and the possibilities for its future software extension.

As was mentioned, each level has its responsibility by user´s actions. The visualization module reads the input from the user as input code and sets the default values of variables. It also provides the forms for

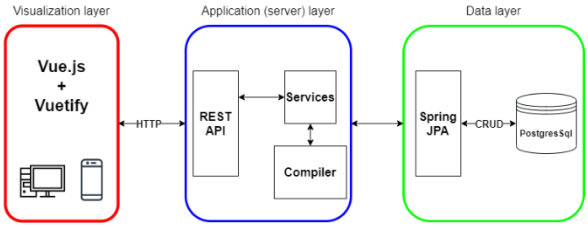

Figure 3. Conceptual model of the system

saving the code, register and login of a user. This level displays to the user a list of available stored codes, output graph of the actual visualization process and it provides the functionality to store this graph also locally (to download it). Each request received from the user is sent to the server level by REST API calls. If the data are to the server during those communication calls, they are encapsulated into JSON objects. The server part of the application handles all requests from the visualization level and sends back the requested data. At this level, a compiler of the input code is also implemented, together with the execution module for the output code which generates data for rendering the output graph. The security for managing the access to data and manipulating them is an integral part of the module. Security is based on the user´s credentials and the information about the owner of the stored codes. User´s verification is implemented by JWT token authentication. This part of the system communicates with the last part of the system which is the data layer. Communication between these two layers is by JPA interface. For the database layer, we had chosen the PostgreSQL database system. This more detailed description of the architecture is depicted in Figure 4.

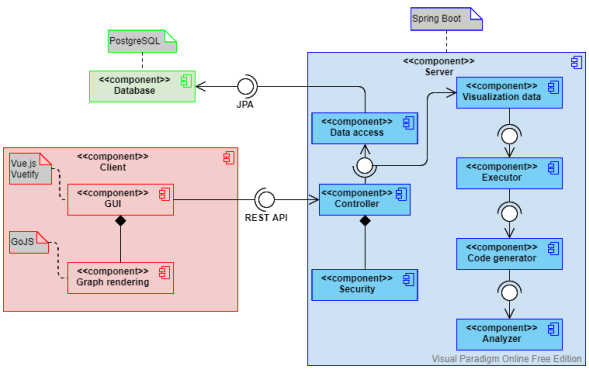

Figure 4. Component diagram describing system architecture

5 Implementation of the system

The system is implemented in two main modules. The first module is for the user interface and the second one serves as the server of the application.

This structure is implemented by the Maven tool which is independently on the developing platform. The main file of this structure is a special XML file that contains all information about the compilation of the final executable file. Each module contains this kind of file with information about compiling that module. In the root level is the parent XML file which has references to all modules with information about the order of the compilation and generating the output file.

The user interface is implemented by Vue.js framework with additional libraries. This framework is component-oriented which allows reusability of the created components. For the graphical aspect, we used the material library Vuetify which contains a lot of predefined components with the possibility to customize them. This material design library helps us also to achieve responsibility for the application. For handling and managing data in the front-end part, we used Vuex and for routing and serving the right components was used Vue Router. Depending on the architecture, we needed to figure out how we will make API calls and for this purpose we used the Axios library. The whole user interface is multilingual thanks to the Vue-i18n library. We have implemented three languages: English, German, and Slovak for now. For the main part of the application, visualizing the graph of the semantics we used the GoJS library.

The server part of the application is implemented in Java with the Spring Boot framework. This framework is component-oriented as well as a front-end framework. The reason for choosing this framework was for its easy implementation of the REST API interface and easy connection to the database. The server consists of the entities which specify database tables, API calls requests and responses. It also contains controllers which are handling API calls with associated services that perform requested actions and communicate with the database. We have implemented security for some of the API interfaces because we needed to ensure the integrity of the data stored in a database. This security is implemented by JWT token authentication and based on the user´s login credentials.

6 Example

Now we demonstrate usage of the system and its implementation on the following example which is code for finding the maximal value of the three variables (Figure 5).

```
if (x ≤ y) then {
        max:=y;
} else {
        max:=x;
}

if (max ≤ z) then {
        max:=z;
}
```

Figure 5. Source code of finding the maximum of three values

First, we need to open the application. The initial screen is shown in Figure 6. The navigation of the application is situated at the top. At the bottom of the application, we have a footer that contains contact to the developer and functionality to change the language.

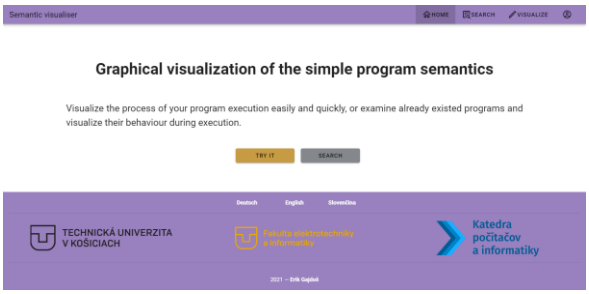

Figure 6. Home page of the application

At the start of the visualizing process, we choose from the navigation menu option Visualize which shows us the view where we can enter and visualize the code. This screen is shown in Figure 7. At the first, we have there only one option for entering the code for analysis. If we are logged in, we will have there also option for entering the name of the code and its description with the possibility to store it for later. After entering the code, we need to click on the button Lexical analysis, which sends the code to the server and then shows us the option for setting the default values of variables. This setting is not necessary but can affect the output of the semantic visualization.

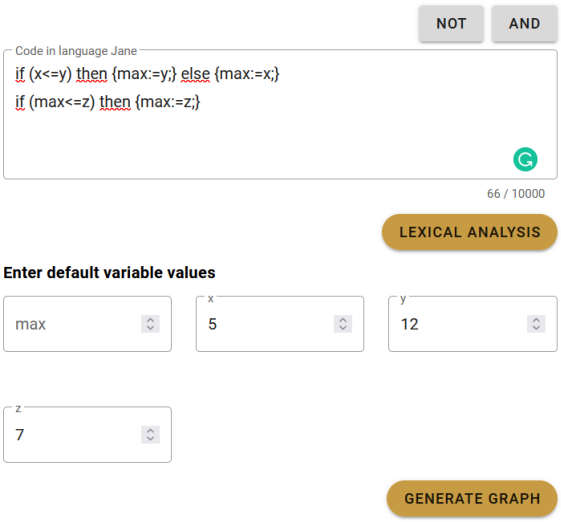

Figure 7. Visualization page

When we are done with the setting, we can continue with the button Generate graph. After clicking on this button, an input code and default values are sent to the server which compiles the code, generates the executable one and if there are no errors also executes it. After a successful execution process, the server sends back data describing the output graph. Depending on the received data, the user interface renders the output graph. An example of this graph is depicted in Figure 8 for input values $x = 5, y = 12, z = 7$. After rendering the graph, the option for downloading this graph is also shown.

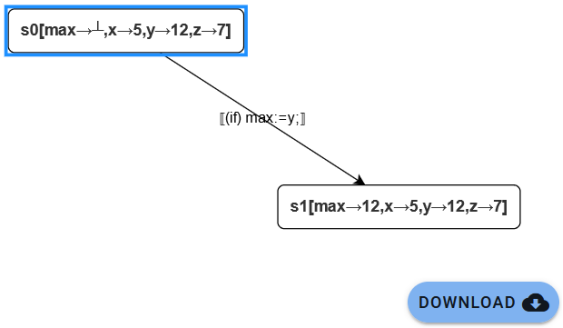

Figure 8. Example of the output graph

In case of errors during the compilation process, the code is not executed and a response from the server is sent with the information about occurred errors. For example, if we remove the semicolon from the variable assignment statement and a curly brace from the else branch, an error depicted in Figure 9 occurs.

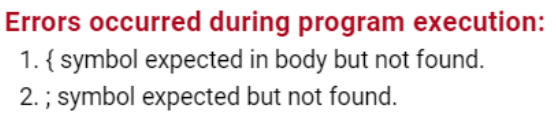

Figure 9. Example of the error output

7 Conclusion

We presented in this paper application for visualizing the semantics of the imperative languages. We described the theoretical background of this work and the reasons for its implementation. Our application allows the entering of custom input code in Jane language and it provides the syntactic analysis, identifying incorrectly entered statements and visualizing the semantics of the given code. Additional functions are storing (saving) the code for later work, listing the existing codes and evaluating their behavior and downloading the rendered graph as a PNG image. This application can be used for research work and mainly in the course Semantics of programming languages as a learning tool for students and teachers as a tool for better describing the semantics and behavior of the programs.

Acknowledgment

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.

Appel, A. W. (2002). *Modern Compiler Implementation in Java*. Cambridge: Cambridge University Press.

Barr, M., Wells, C. (1990). *Category Theory for Computing Science*. New York: Prentice Hall.

Brandenburg, M. (2016). Einführung in die Kategorientheorie. Springer Spektrum (in German)

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

Gajdoš, E. (2021). The visualization of a graph semantics of imperative languages. *Technical Report*. Technical University of Košice (in Slovak).

Herceg, D. et al. (2019). Possible improvements of modern dynamic geometry software. Computer Tools in Education, (2):72–86.

Nielson, H.R., Nielson, F. (2007). Semantics with Applications: An Appetizer. Undergraduate Topics in Computer Science. Springer-Verlag London.

Parr, T. (2013). The Definitive ANTLR 4 Reference. Pragmatic Bookshelf, Raleigh, NC, 2nd edition.

Perháč, J., Mihályi, D., Novitzká, V. (2017). Modeling Synchronization Problems: From Composed Petri Nets to Provable Linear Sequents. Acta Polytechnica Hungarica, 14(8), 165-182.

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

Roșu, G., Șerbănuță, T. F. (2010), An overview of the K semantic framework, The Journal of Logic and Algebraic Programming, 79(6), 397-434.

Reichl, F.X., Schreiner, W. (2020). Mathematical Model Checking Based on Semantics and SMT. IPSI Transactions on Internet Research, 16(2):4-13.

Steingartner, W. et al. (2017). New approach to categorical semantics for procedural languages. *Computing and Informatics*, 36(6), pp. 1385–1414, Slovakia. doi:10.4149/cai_2017_6_1385

Steingartner, W., Novitzká, V., Schreiner, W. (2019). Coalgebraic Operational Semantics for an Imperative Language. *Computing and Informatics*, 38(5), pp. 1181–1209, Bratislava, Slovakia. doi:10.31577/cai_2019_5_1181

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

Walters, R.F.C. (1992). Categories and Computer Science. Cambridge University Press