Journal of Applied Mathematics and Computational Mechanics 2018, 17(2), 77-91
www.amcm.pcz.pl p-ISSN 2299-9965
DOI: 10.17512/jamcm.2018.2.07 e-ISSN 2353-0588

LEARNING SOFTWARE FOR HANDLING THE MATHEMATICAL EXPRESSIONS

William Steingartner, Iskender Yar-Muhamedov

Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics
Technical University of Košice
Košice, Slovakia
william.steingartner@tuke.sk, iskender.eu@gmail.com

Received: 23 May 2018; Accepted: 28 June 2018

Abstract. Educating young software engineers and IT experts is a great challenge nowadays. Still new technologies are used in a practical approach and many of them come from formal methods. To help future software experts in the understanding of formal methods grounded in semantics, learning software that illustrates and visualizes important techniques seems to be very fruitful. In this paper, we present software, which handles the arithmetic and Boolean expressions, their analysis, evaluation, drawing the syntax tree and the other techniques with the expressions. This software is devoted as a teaching tool for teachers when explaining appropriate theory and for students for self-studying and making their own experiments. Furthermore, this software is an integral part of our software package for several semantic methods.

MSC 2010: *68N20, 68Q55, 97Q60*
Keywords: *abstract syntax tree, learning software, lexical analysis, mathematical method, semantics of expressions, shunting-yard algorithm, syntax analysis*

1. Introduction

At present, computer science makes use of various formal methods and selected methods in applied mathematics to help in the understanding of complex software systems and to reason about their behavior. Formal methods are very important for developing of correct software systems, in particular to verify the correctness of the systems or at least of some desired aspects of their behavior. In the last decades, the impact of formal methods increases more and more [1, 2]. In the educational process, we consider adequately educating and training them in the basics of formal logic and formal language semantics to be a very important role. In this manner, the tools support software development on the basis of formal methods [3, 4]. All of these tools and techniques are rooted in the formal semantics of the particular programming languages. Based on our experience, we extended the package of software support tools [5, 6] with a new module for handling the arithmetic and Boolean expressions.

During the execution of many programs, some expressions are evaluated and used for control flow or stored in memory as partial or final results of calculations [7, 8]. The language Jane works with both types of expressions. The arithmetic expressions are implicitly typed as integers. Their values can be evaluated as transient, used in Boolean expressions and forgotten; or stored in memory when standing on the right-hand side of an assignment statement. Boolean expressions are typed with standard Boolean type with two values (true and false). We note, that the Jane programming language does not contain short circuit operators. Then always the whole Boolean expression is being evaluated. The values of Boolean expressions are only transient because in our language we consider only storing of integers in memory and we do not consider to store Boolean values in memory. The Boolean expressions are evaluated and used in conditional and loop statements, and their values are never stored in memory for another use. When working with expressions, it is necessary to test them for the syntax errors. After an error occurs, the error recovery process starts as a next step [9]. On the other hand, when an expression is syntactically correct, a user needs to assign concrete values to variables in an expression and to evaluate the given expression for the final resulting value.

Conversion to postfix notation is necessary when constructing the syntax tree and when evaluating the expressions. Furthermore, a syntax tree of an expression is important for better understanding of the structure of an expression when evaluating it [9]. Taking into consideration all these requirements, we have prepared a new module for handling the expressions. The program:

- checks the input expression and recognizes the type (arithmetic or Boolean one);
- realizes the error recovery;
- allows a user to input the values for particular variables identified in the expression entered;
- evaluates the whole expression with input values (can be possibly changed interactively, in any time);
- produces the postfix form of an expression;
- draws the abstract syntax tree for an input expression in three versions (only variable names or only values or variable names with values);
- stores the output.

We present in the second section the obvious syntax of arithmetic and Boolean expressions of the Jane programming language. In the third section we define the semantic functions using standard mathematical methods for the semantic of expressions. The fourth section presents the learning software package for a course on Semantics of programming languages and in the fifth section, we present how our module for handling the expressions was developed and how it works.

2. Syntax of expressions

A course on Semantics of programming languages is oriented mostly with semantics of imperative languages. For purposes of the course, the main facts about the imperative paradigm are presented on the model programming language Jane.

Jane is a simple imperative language, which contains all traditional imperative constructs (the so called van Dijkstra's constructs or D-diagrams [10]) - assignment statement for storing the values in memory, sequencing of statements, conditional statement and prefix logical loop statement. Furthermore, it contains also an empty statement. The greatest advantage of an empty statement is that it contributes to writing clear and unambiguous programs. The syntax of statements in Jane programming language is as follows:

$$S ::= x := e \mid \texttt{skip} \mid S;S \mid \texttt{if}\ b\ \texttt{then}\ S\ \texttt{else}\ S \mid \texttt{while}\ b\ \texttt{do}\ S.$$

In this section we briefly present the syntax of arithmetic and Boolean expressions that are very widely used in computations. Our module allows one to work with standard and extended grammars of particular expressions, so we present in both cases the standard and the extended syntax of expressions.

2.1. Arithmetic expressions

Arithmetic expressions usually stand on the right-hand side of assignment statements. Their standard syntax is the following:

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

where:
- n stands for a string of numerals;
- x represents an atomic variable;
- $e+e$, $e-e$, $e*e$ are standard arithmetic operations; and
- any $e \in \mathbf{Expr}$ stands for well-formed syntactic element of syntactic domain for arithmetic expressions.

All expressions are implicitly typed as integers.

Moreover, if there is a situation where user also needs an integer division operation or unary operators, the program can work with extended grammar in this form:

$$e ::= n \mid x \mid e+e \mid e-e \mid e*e \mid e/e \mid -e \mid +e \mid (e). \tag{2}$$

An operation of integer division is in some languages denoted as $e\ \texttt{div}\ e$.

2.2. Boolean expressions

Boolean expressions occur in conditional and loop statements. Here, the values of Boolean expressions are never stored; they are only used in the flow of program. We say that these values are transient. Syntax of Boolean expressions is the following:

$$b ::= \texttt{false} \mid \texttt{true} \mid e = e \mid e \leq e \mid \neg b \mid b \wedge b \mid (b), \tag{3}$$

where:

- $\texttt{false}$ and $\texttt{true}$ are Boolean constants;
- $e = e$ and $e \leq e$ represent equality and less or equal relation, resp.;
- $\neg b$ stands for a negation of the Boolean expression;
- $b \wedge b$ represents conjunction of two Boolean expressions; and
- any $b \in \mathbf{BExpr}$ stands for a well-formed syntactic element of syntactic domain for Boolean expressions.

All Boolean expressions are typed as two-valued Booleans.

To make our application closer to the standard use of Boolean operations, it allows one to also use the operator for disjunction of two Boolean expressions,

$$b \vee b$$

according to the extended following grammar:

$$b ::= \texttt{false} \mid \texttt{true} \mid e = e \mid e \leq e \mid \neg b \mid b \wedge b \mid b \vee b \mid (b). \tag{4}$$

The remaining Boolean operations over two Boolean expressions and the remaining inequalities can be constructed by combining the basic relations and logical connections listed in rules (3) and (4).

3. Semantics of expressions

In this section, we show how the semantics of arithmetic and Boolean expressions is defined. To define semantics, we start with the notion of a memory state.

3.1. The notion of memory state

Execution of a program generally causes a change of some memory cells. Every snapshot of a memory during program execution can be abstracted as a state where program variables have assigned some values [10, 11]. Execution of a statement can modify some values of program variables, i.e. a state is changed. On the other hand, during the evaluation of expressions, the values are only taken from a memory without any change of particular memory cells.

States are enclosed in a semantic domain of states denoted **State**, which is a function space:

$$\mathbf{State} = \mathbf{Var} \to \mathbf{Z},$$

where **Var** stands for a semantic domain for variables' names. This domain consists only of well-formed variables' names and it has no internal structure for semantic purposes. Any state $s \in \mathbf{State}$ represents a function and its application on the name of a variable provides the value of the given variable:

$$s\,x \in \mathbf{Z}.$$

3.2. Semantics of arithmetic expressions

Semantics of arithmetic expressions is defined by the semantic function

$$\mathscr{E} : \mathbf{Expr} \to \mathbf{State} \to \mathbf{Z},$$

where **Z** stands for a semantic domain of integer numbers and semantic function is defined for particular syntactic forms of rules (1) and (2) as follows:

$$\begin{array}{ll}
\mathscr{E}[\![n]\!]s & = \mathscr{N}[\![n]\!], \\
\mathscr{E}[\![x]\!]s & = s\,x, \\
\mathscr{E}[\![e_1 + e_2]\!]s & = \mathscr{E}[\![e_1]\!]s \oplus \mathscr{E}[\![e_2]\!]s, \\
\mathscr{E}[\![e_1 - e_2]\!]s & = \mathscr{E}[\![e_1]\!]s \ominus \mathscr{E}[\![e_2]\!]s, \\
\mathscr{E}[\![e_1 * e_2]\!]s & = \mathscr{E}[\![e_1]\!]s \otimes \mathscr{E}[\![e_2]\!]s, \\
\mathscr{E}[\![e_1 / e_2]\!]s & = \mathscr{E}[\![e_1]\!]s \oslash \mathscr{E}[\![e_2]\!]s, \\
\mathscr{E}[\![-e]\!]s & = \mathscr{N}[\![0]\!] \ominus \mathscr{E}[\![e]\!]s, \\
\mathscr{E}[\![+e]\!]s & = \mathscr{E}[\![e]\!]s, \\
\mathscr{E}[\![(e)]\!]s & = (\mathscr{E}[\![e]\!]s)\,,
\end{array}$$

where the operators on the left-hand side represent the syntactic form of arithmetic operators, the operator on the right-hand side represent the real mathematic operation and the function $\mathscr{N}$ is a function that transforms syntactic numerals into integer values.

3.3. Semantics of Boolean expressions

Semantics of Boolean expressions is defined by the following semantic function

$$\mathscr{B} : \mathbf{BExpr} \to \mathbf{State} \to \mathbf{B},$$

where **B** stands for a semantic domain of Boolean values:

$$\mathbf{B} = \{\mathbf{false}, \mathbf{true}\}$$

and the semantic function is defined for particular syntactic forms of rules (3) and (4) as follows:

$$\begin{array}{ll}
\mathscr{B}[\![\texttt{true}]\!]\,s & = \textbf{true};\\
\mathscr{B}[\![\texttt{false}]\!]\,s & = \textbf{false};\\
\mathscr{B}[\![e_1 = e_2]\!]\,s & = \begin{cases} \textbf{true}, & \text{if } \mathscr{E}[\![e_1]\!]\,s = \mathscr{E}[\![e_2]\!]\,s;\\ \textbf{false}, & \text{if } \mathscr{E}[\![e_1]\!]\,s \neq \mathscr{E}[\![e_2]\!]\,s; \end{cases}\\
\mathscr{B}[\![e_1 \leq e_2]\!]\,s & = \begin{cases} \textbf{true}, & \text{if } \mathscr{E}[\![e_1]\!]\,s \leq \mathscr{E}[\![e_2]\!]\,s;\\ \textbf{false}, & \text{if } \mathscr{E}[\![e_1]\!]\,s > \mathscr{E}[\![e_2]\!]\,s; \end{cases}\\
\mathscr{B}[\![\neg b]\!]\,s & = \begin{cases} \textbf{true}, & \text{if } \mathscr{B}[\![b]\!]\,s = \textbf{false};\\ \textbf{false}, & \text{if } \mathscr{B}[\![b]\!]\,s = \textbf{true}; \end{cases}\\
\mathscr{B}[\![b_1 \wedge b_2]\!]\,s & = \begin{cases} \textbf{true}, & \text{if } \mathscr{B}[\![b_1]\!]\,s = \textbf{true} \text{ and } \mathscr{B}[\![b_2]\!]\,s = \textbf{true};\\ \textbf{false}, & \text{if } \mathscr{B}[\![b_1]\!]\,s = \textbf{false} \text{ or } \mathscr{B}[\![b_2]\!]\,s = \textbf{false}; \end{cases}\\
\mathscr{B}[\![b_1 \vee b_2]\!]\,s & = \begin{cases} \textbf{false}, & \text{if } \mathscr{B}[\![b_1]\!]\,s = \textbf{false} \text{ and } \mathscr{B}[\![b_2]\!]\,s = \textbf{false};\\ \textbf{true}, & \text{if } \mathscr{B}[\![b_1]\!]\,s = \textbf{true} \text{ or } \mathscr{B}[\![b_2]\!]\,s = \textbf{true}; \end{cases}\\
\mathscr{B}[\![(b)]\!]s & = (\mathscr{B}[\![b]\!]s)\,.
\end{array}$$

4. Software package for teaching

A teaching course on Semantics of programming language is now being supported by a software package consisting of several modules. Actually, the software package contains the following modules:

- compiler of source in Jane into Abstract machine code - a source-to-source compiler (Fig. 1 - case *a*);
- emulator of Abstract machine code (Fig. 1 - case *a*);
- decompiler of Abstract machine code into Jane source (Fig. 1 - case *b*);
- a tool for visualization of program execution based on categorical denotational semantics (Fig. 1 - case *c*);
- a tool for handling the expressions described in this paper (Fig. 1 - case *d*).

A general scheme of our software package is depicted in Figure 1. In this paper we introduce and present the last case - the software module for working with expressions.

The software package allows one to work with the Jane language - to compile it into Abstract machine code when working with structural operational semantics and then to execute the Abstract machine code, to find source in Jane from the input Abstract machine code and to visualize the running program in categorical denotational semantics. Furthermore, the arithmetic and Boolean expressions are very widely used in programs and especially in each of the listed modules, so the module for expressions is an important part of this package. The main motivation for this integrated software package was to help students understand better formal semantics, and to motivate them to make their own experiments. We follow the joint goal based on common research work [12]. In the next chapter we completely present the module in case *d* of our software package.

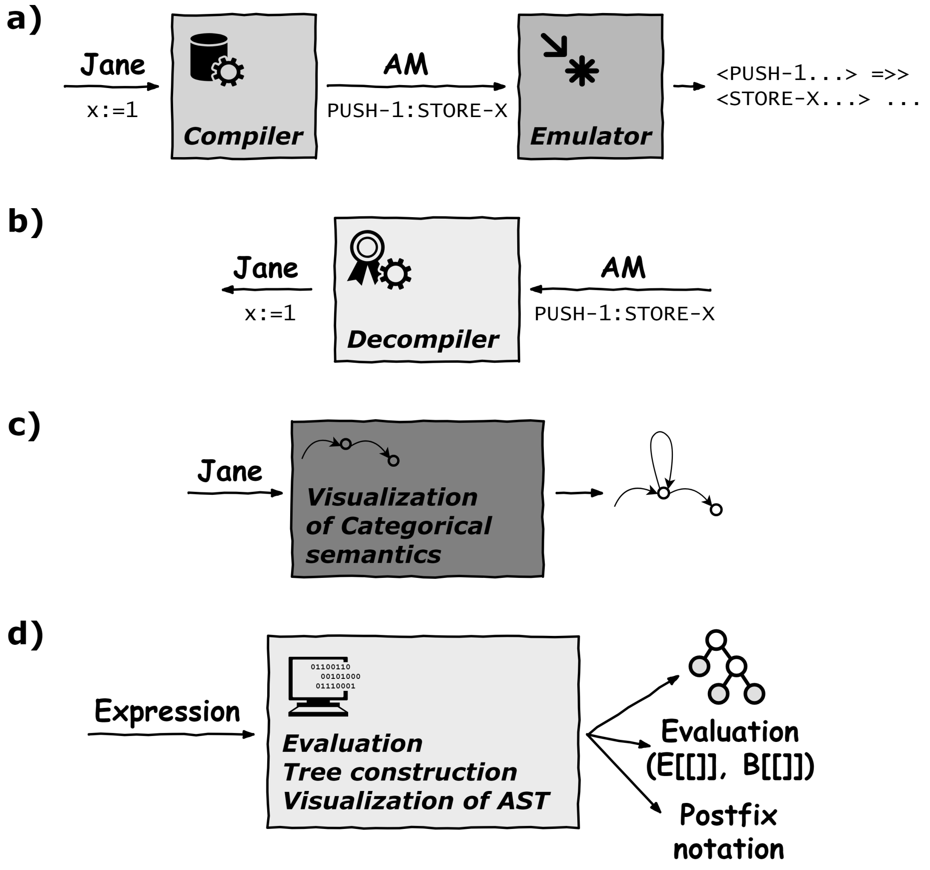

Fig. 1. General scheme of software package

5. Implementation of a module

The application we developed implements the primary functions that are necessary in the study of the course Semantics of programming languages. All of them are related to evaluating both arithmetical expressions and Boolean expressions. Another one of the primary function is to produce an Abstract Syntax Tree (AST) reflecting the structure of the input expression, if the input expression is a valid expression in the Jane programming language according to the defined grammar [10, 13]. Secondary functions represent input expression in the postfix form and produce an error message if an input expression is not valid [13].

The implementation of the program consists of several software modules:
- lexical analysis;
- transformation of input expressions given in infix form to postfix form;
- syntax analysis;
- assignment of values to variables;
- evaluation of the input expression;
- painting the Abstract Syntax Tree (AST).

An object-oriented design was used to develop the module. The general (simplified) UML scheme of the module is in Figure 2.

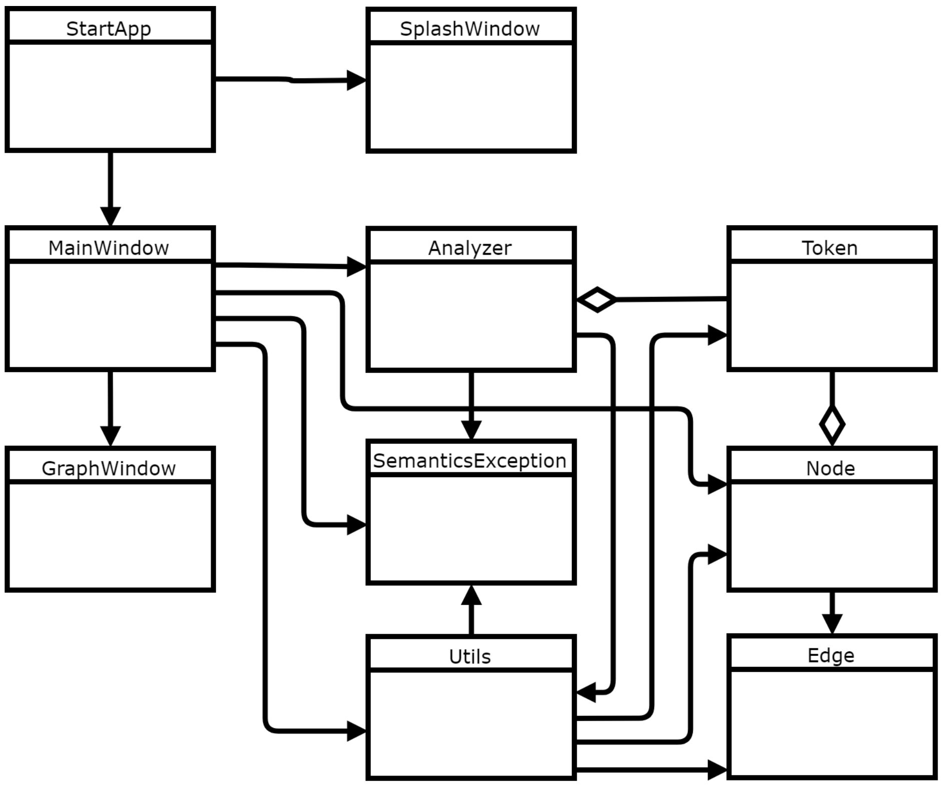

Fig. 2. Simplified UML diagram of the module

We used the Swing API for providing a graphical user interface (GUI) for our application. Swing was developed to provide a more sophisticated set of GUI components than the earlier Abstract Window Toolkit (AWT). The Java Swing API provides a pluggable look-and-feel (PLAF) capability, which allows Swing GUI widgets to change appearance based on the programmer's customized look-and-feel setting [15]. Swing API components are not implemented by a platform-specific code. Instead, they are written entirely in Java and therefore are platform-independent.

The first step in implementation of a software module is a lexical analyzer, where we at first created a *Token* class that represents entity of token.

An input expression is represented as a *String* data type, and the first step after input is to delete all whitespaces (typically spaces, line feeds "$\backslash n$", line breaks "$\backslash r$", tabs "$\backslash t$" etc.) in the input expression. After this, it is necessary to check whether the string characters belong to the defined grammar [13].

The next step of implementation of the Lexical analyzer is an implementation *parser*() method to parse the input string and to store tokens to the prepared data structure *LinkedList*$<$ *Token* $>$. Further action is transforming the input expression in the infix form to the postfix form. For this action we created the *transformToPostfix*() method which is based on the shunting-yard algorithm [14]. Here, we first defined the precedence of all existing operators in the Jane programming language. Each input expression in the Jane programming language will have a single AST based on the following precedence and associativity rules:

- parentheses have precedence over all operators;
- unary $+$, unary $-$ and $\neg$ have precedence over all binary operators;
- operators $*$ and $/$ have precedence over binary $+$, $-$, $\vee$, $\wedge$, $=$, $\leq$;
- operators $+$ and $-$ have precedence over $\vee$, $\wedge$, $=$, $\leq$.

The shunting-yard algorithm is a method for transforming arithmetic and Boolean expressions specified in infix notation to postfix notation, also known as Reverse Polish notation (RPN). This algorithm was invented by Edsger Wybe Dijkstra in 1960 [14], and an example of using this algorithm also implemented by our software is depicted in Figure 3. The idea of the shunting-yard algorithm is to keep operators in the first stack, noted as "Additional stack" and the postfix expression is formed in the second (output) stack, which we noted as "Output expression" [14]. The directional arrow into Garbage Collector in the Figure 3 represents deleted useless element from the stack. Generally the Garbage Collector in this algorithm is used for automatic memory management.

The algorithm works as follows: it sequentially reads the tokens from the input list of tokens and moves the read token onto one of the stacks, according to the rules listed below.

- If the token is a constant or a variable, it is moved onto the output stack.
- If the token is an operator, the algorithm moves it onto the additional stack, but before that it checks the precedence of the operator on a top of the additional stack. If there is an operator with a higher or equal precedence and the actual work-in-process operator is left associative or the actual work-in-process operator is right associative and at the top of the additional stack there is an operator with a higher precedence, then algorithm moves the operator from the top of the additional stack onto the output stack. This procedure repeats until the additional stack is empty or the condition for moving token from the top of the additional stack is not satisfied.
- If the token is a left parenthesis, it is moved onto the additional stack.
- If the token is a right parenthesis, the algorithm sequentially moves all the tokens from the additional stack onto the output stack until the token on the top of the additional stack is left parenthesis. After that, the algorithm moves the left parenthesis from the additional stack. If the algorithm does not find the left parenthesis,

then the input expression is not syntactically correct. In this case, the user will receive an error message.

In case of successful processing of the input expression by the shunting-yard algorithm we obtain an expression in the Reverse Polish Notation as an output argument. The method's input parameter is the *LinkedList* $<$ *Token* $>$ data structure which represents an input expression in infix form, and output parameter the *LinkedList* $<$ *Token* $>$ data structure which represent input expression transformed into the postfix form (RPN).

Moreover, the shunting-yard algorithm can be used to directly evaluate expressions [14].

The next step of development is implementation of the software module for syntax analysis. To achieve this goal, we created the *syntaxChecker*() method which handles the following input parameters:

- *LinkedList* $<$ *Token* $>$ data structure - represents input expression in postfix form, and
- the Boolean variable - defines working mode of the *syntaxChecker*().

The value **true** of the Boolean variable defines the extended grammar working mode of the *syntaxChecker*() and handling the expressions according the syntax in (4); and value **false** of the Boolean variable defines the base grammar working mode of the *syntaxChecker*() and handling the expressions according the syntax in (3). This method was implemented on the basis of the scheme of the algorithm, which is depicted in Figure 4.

As it can be seen in Figure 4, the *syntaxChecker*() method contains one initial state (Start) and several finite ones. This method finishes in the final state if an input expression is syntactically correct. Otherwise the method terminates in one of the *SemanticsException* states. The module of the *SemanticsException* state opens a dialog window to notify the user of the founded syntax error.

For implementing the module Assignment of values to the variables module, we created *getAllVariables*() method that finds all existing variables in the *LinkedList* $<$ *Token* $>$ data structure, and it creates a two-dimensional array. The first dimension saves the names of the variables and the second dimension stores the values of the variables. Based on this array, we created the *TableModel* class which extends the *AbstractTableModel* class and represents our table model. Thus, the model of our table consists of two columns. The first column is called the Name (a name of a variable), and the second one is called Value (the value of the variable). The number of rows in the table is equal to the number of names of non-repeating variables. In addition, we override *getColumnClass*() method in the *TableModel* class. For the first column, we set a value of *String.class* type and for the second column - *Integer.class* type. An overriding of this method will strictly determine the types of values that can be stored in the columns of the table. At the same time, the overriden method *getColumnClass*() does not allow one to assign the value of a variable of any other type except the *Integer* type.

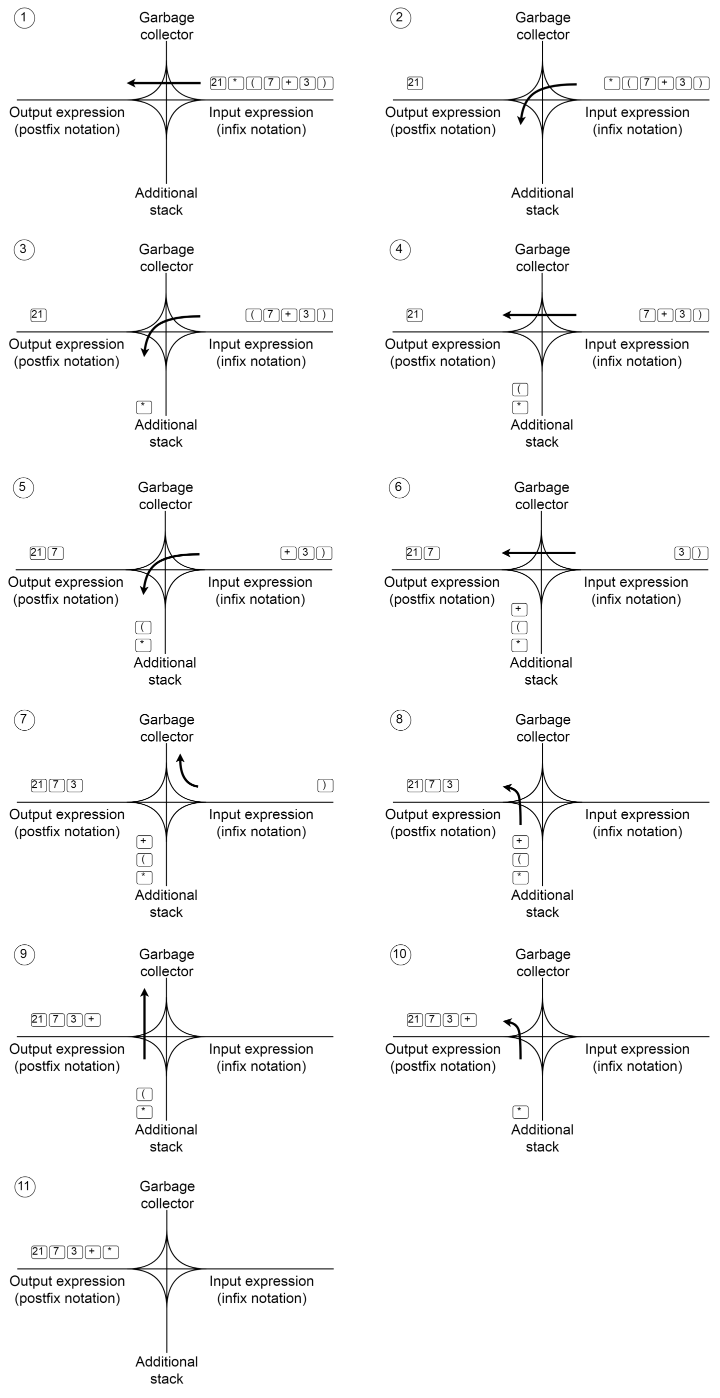

Fig. 3. A scheme of the shunting-yard algorithm

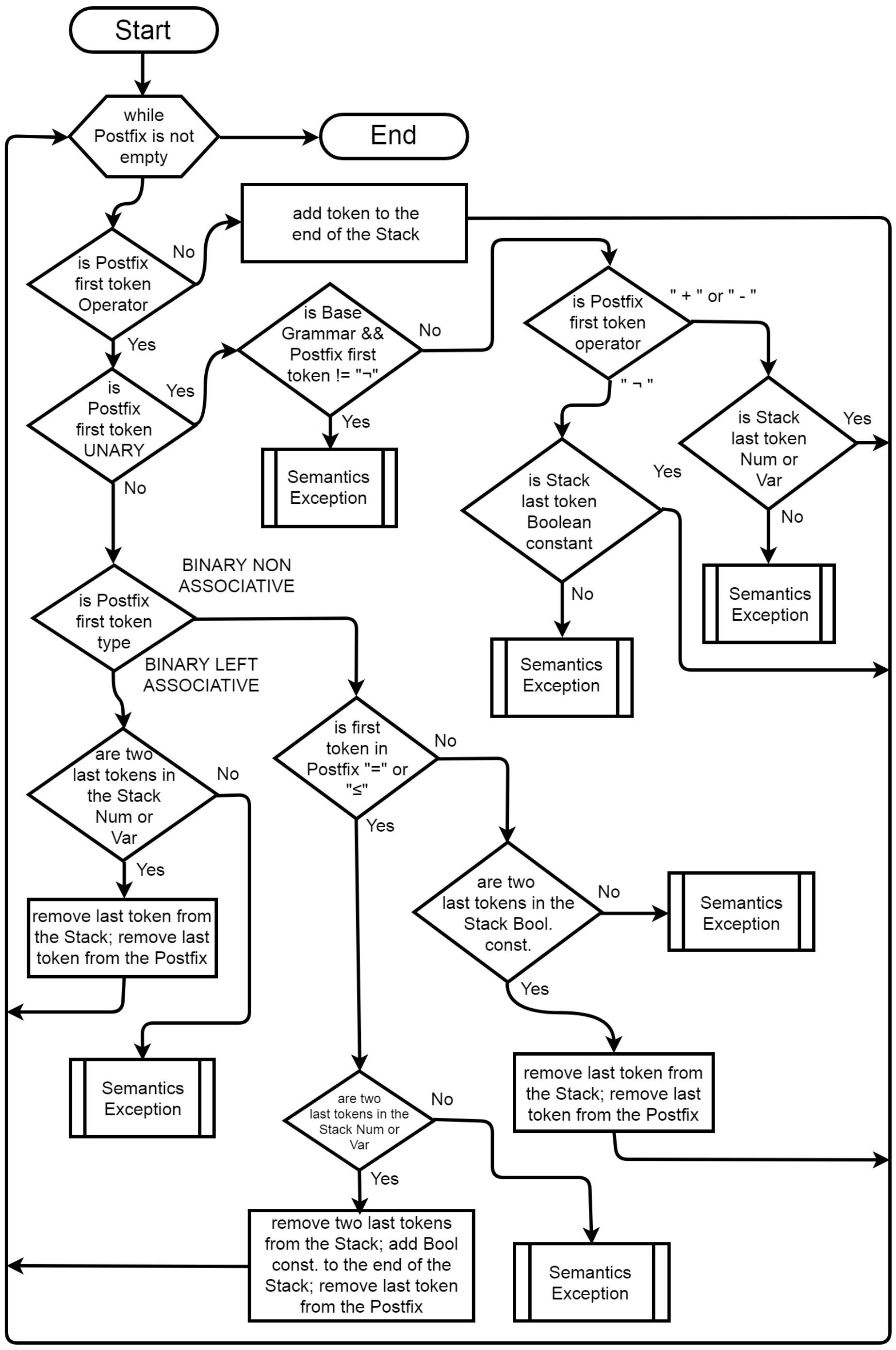
Start
while Postfix is not empty
End
is Postfix first token Operator
No
add token to the end of the Stack
Yes
is Postfix first token UNARY
Yes
is Base Grammar && Postfix first token != "¬"
No
Yes
Semantics Exception
is Postfix first token operator
" + " or " - "
" ¬ "
is Stack last token Num or Var
Yes
No
Semantics Exception
is Stack last token Boolean constant
Yes
No
Semantics Exception
No
is Postfix first token type
BINARY NON ASSOCIATIVE
BINARY LEFT ASSOCIATIVE
are two last tokens in the Stack Num or Var
No
Yes
remove last token from the Stack; remove last token from the Postfix
Semantics Exception
is first token in Postfix "=" or "≤"
No
Yes
are two last tokens in the Stack Bool. const.
No
Semantics Exception
Yes
remove last token from the Stack; remove last token from the Postfix
are two last tokens in the Stack Num or Var
No
Yes
remove two last tokens from the Stack; add Bool const. to the end of the Stack; remove last token from the Postfix
Semantics Exception

Fig. 4. The scheme of the *syntaxChecker*() algorithm

Evaluation of the input expression module is the primary functionality of the development software. For this module we created *calculate*() method. This method contains input parameter of the *LinkedList* $<$ *Token* $>$ type and output parameter of the *Object* type.

The algorithm works as follows: it repeatedly reads tokens from the input data structure *LinkedList* $<$ *Token* $>$ and processes it according to the rules listed below.

- If the token is a constant or a variable, the algorithm moves it onto the addition *LinkedList* $<$ *Token* $>$ data structure.
- If the token is an unary operator, then algorithm extracts the token from the end of the addition *LinkedList* $<$ *Token* $>$ data structure and applies this operator to this token (operand). Then it writes the result to the end of the addition *LinkedList* $<$ *Token* $>$ data structure.
- If the token is a binary operator, then the algorithm extracts two tokens from the end of the addition *LinkedList* $<$ *Token* $>$ data structure and applies this operator to these tokens (operands). Then it writes the result to the end of the addition *LinkedList* $<$ *Token* $>$ data structure.

We simply note, that the *Token* class represents token. It contains information about whether the token is an operator or an operand. If it is an operator, the *Token* class also contains information about whether this operator is binary or unary, its precedence etc.

After reading the whole input *LinkedList* $<$ *Token* $>$ data structure it stays empty and after the addition *LinkedList* $<$ *Token* $>$ data structure contains only one element. This element is a result after the evaluation of input expression.

For Painting the Abstract Syntax Tree module, we created *Node* class that represents recursive data structure of a tree. We also created the *getTree*() method, which transform *LinkedList* $<$ *Token* $>$ data structure to the recursive data structure of a type *Node*.

To draw and graphically visualize the AST tree, we use the ”JUNG” software library. ”JUNG” - the Java Universal Network/Graph Framework - is a software library that provides a common and extendible language for the modeling, analysis, and visualization of data that can be represented as a graph or network. It is written in Java, which allows JUNG-based applications to make use of the extensive built in capabilities of the Java API, as well as those of other existing third-party Java libraries [16]. The final step for the graphical visualization of the AST tree is the development of *createTree*() method, which creates recursive *Forest* $<$ *Node*, *Edge* $>$ data structure from the *Node* recursive data structure. In addition, in our application, we implemented the function of saving the generated AST onto PDF file based on the vector graphics. For this function, we decided to use the ”FreeHEP PDF Driver” software library [17].

6. Conclusions

We presented in this paper the application that can process arithmetic and Boolean expressions, their syntactic analysis, output in the form of a postfix notation and Abstract Syntax Tree, to identify incorrectly entered expressions (implementation of error recovery) and interactive user intervention during entering data with subsequent evaluation of the results. Also, the application contains a function for saving a generated Abstract Syntax Tree onto a PDF file based on the vector graphics. Moreover, this function allows the user to use the generated vector graphics images in future research. The application is going to be used as a learning tool for the students of the course Semantics of programming languages:

- for preparing the lecture materials and laboratory exercises;
- for individual study and the for the experimental approach to learning; and
- for preparing the testing and exam materials.

A learning tool for expressions has an irreplaceable *rôle* in our software package. We want to extend our learning software package with other modules based on operational semantics with categorical structures and coalgebras according to [18] and for the other important and relevant semantic methods.

Acknowledgment

This work has been supported by the Slovak Research and Development Agency under the contract No. SK-AT-2017-0012: Semantics technologies for computer science education.

References

[1] Novitzká, V. (2005). Logical reasoning about programming of mathematical machines. *Acta Electrotechnica et Informatica*, *5*, 3, 50-55.

[2] Siedlecka-Lamch, O., Kurkowski, M., & Piatkowski, J. (2016). Probabilistic model checking of security protocols without perfect cryptography assumption. In Proc.: 23rd International Conference, Computer Networks 2016, Brunow, Poland, June 14-17 2016, Communications in Computer and Information Science, 608, 107-117, Springer, 2016.

[3] Korečko, Š., & Sobota, B. (2017). Computer games as girtual environments for safety-critical software validation. *Journal of Information and Organizational Sciences*, *41*, 2, 197-212.

[4] Grzybowski, A.Z. (2010). Goal programming approach for deriving priority vectors - some new ideas. *Scientific Research of the Institute of Mathematics and Computer Science*, *9*, 1, 17-27.

[5] Steingartner, W., & Novitzká, V. (2017). *Learning tools in course on semantics of programming languages*. Conference on Mathematical Modeling in Physics and Engineering - MMFT 2017, Poraj, Poland, September 18th-21th 2017. Czestochowa; Czestochowa University of Technology.

[6] Steingartner, W., Eldojali, M., Radaković, D., & Dostál, J. (2017). *Software support for course in semantics of programming languages*. 2017 IEEE 14th International Scientific Conference on Informatics. Poprad, Slovakia, November 14th-16th 2017.

[7] Shkarupylo, V., Skrupsky, S., Oliinyk, A., & Kolpakova, T. (2017). Development of stratified approach to software defined networks simulation. *Eastern-European Journal of Enterprise Technologies*, *5*, 9(89) - Information and Controlling System, 67-73, doi: 10.15587/1729-4061.2017.110142.

[8] Oravec, J., Turán, J., Ovseník, L., Ivaniga, T., Solus, D., & Márton, M. (2018). Asymmetric image encryption approach with plaintext-related diffusion. *Radioengineering*, 27, 1.

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

[10] Steingartner, W., & Novitzká, V. (2015). *Semantics of Programming Languages*. 1st edition. Technical University of Košice (in Slovak).

[11] Nielson, H.R., & Nielson, F. (2007). *Semantics with Applications*. London; Springer-Verlag.

[12] Schreiner, W., & Steingartner, W. (2018). *Visualizing Execution Traces in RISCAL.* Technical Report, Research Institute for Symbolic Computation (RISC). Linz; Johannes Kepler University.

[13] Yar-Muhamedov, I. (2018). *Software tool for course semantics of programming languages*. Masters thesis. Technical University of Košice.

[14] Norvell, T. (1999). *Parsing Expressions by Recursive Descent*. Retrieved from http://www.engr.mun.ca/ theo/Misc/exp_parsing.htm.

[15] Portyankin, I. (2010) *Swing, Effective User Interfaces*, 2*nd* Edition, Moscow, Lory (in Russian).

[16] JUNG 2.0 Tutorial. (2009).

[17] Donszelmann, M. et al. (2007). *FreeHEP PDF Driver*. Retrieved from https://mvnrepository.com/artifact/org.freehep/freehep-graphicsio-pdf.

[18] Steingartner, W., Novitzká, V., Eldojali, M.A., & Radaković, D. (2016). *Some aspects about coalgebras as foundation for expressing the semantics of imperative languages*. Mathematics and Computer Science - MaCS 2016, Proceedings of the 11th Joint Conference on Mathematics and Computer Science. Eger, Hungary, May 20-22, 2016.