Journal of Investigations on Engineering & Technology
Volume 1, Issue 2, 14-18, 2018

Development of Programming Learning Environment with Compiler Techniques

Sefa ARAS*[1], Eyüp GEDİKLİ[1a]

[1] Karadeniz Teknik Üniversitesi, Teknoloji Fakültesi, Yazılım Mühendisliği Bölümü, 61830, Trabzon

(Alınış / Received: 14.12.2018, Kabul / Accepted: 31.12.2018, Online Yayınlanma / Published Online: 31.12.2018)

Keywords

Programming Learning Environment,
Compiler,
Lexical Analysis,
Syntax Analysis,
Finite State Machines.

Abstract: In this study, a task-based programming learning environment with simple syntax was developed. Existing environments are applications that are mostly programming with visual components, are difficult to develop and contribute to the transition to real programming languages. Within the developed environment, a new programming language has been defined, which is close to the actual programming languages. The programming learning environment has been developed as open source using computer science and engineering techniques, and is a framework for researchers seeking to develop such an environment. The language in the programming learning environment is verified by lexical and syntax analysis steps. Finite state machines control the success of the task. Regular expressions allow users to parse the code written by the user and make the necessary analysis on the code.

1. Introduction

Programming learning improves cognitive skills as well as teaching computer science concepts. Providing this improvement depends on the success of programming learning. While learning programming with real programming languages, individuals face syntax difficulties. Solving these difficulties takes longer than programming. This problem has revealed programming learning environments. In programming learning environments, programming is usually done using visual components [1]. However, programming with visual components leads to difficulties for individuals in transition to real programming languages. This leads to the need for a learning environment that resembles real programming languages.

In the Scratch environment where tasks are performed with visual components, puzzle components that can take values are used [2]. By assigning values to certain fields with the correct parts, assignments, operations, conditions and loops can be performed. In Vimap [3] consisting of multidimensional blocks in which programming instructions are represented by drag and drop, certain values are written into blocks. Greenfoot, which is Java-based and supports object-oriented programming, has been developed for young people [4]. RoBlock is a web based visual programming language consisting of eight modules. The user learns programming by performing tasks in each mode [5].

In this study, a programming learning environment which is simple syntax, task-based, motivates users with intelligent feedback, motivation with gamification is developed. Within the developed environment, a new programming language has been defined, which is close to the actual programming languages. Lexical and syntax analysis steps verify the language developed in the programming learning environment. Finite automats control the success status of task. Regular expressions allow users to parse the code written by the user and make the necessary analysis on the code.

In the second part of the work, the method and the used tools, in the third part system architecture, in the last part the results are given.

* Corresponding Author: sefaaras@ktu.edu.tr ORCID: 0000-0002-4043-3754
[a] ORCID: 0000-0002-7212-5457

2. Method

It is stated in the literature that programming learning environments developed with visual components improve cognitive skills and reduce syntax difficulty [6-8]. However, it is difficult for individuals who learn programming using visual components to switch to real programming languages [9, 10]. Therefore, the language used in the programming learning environment developed is very close to the real programming languages and the better learning process is aimed with the intelligent feedbacks.

2.1. The Tools Used

The programming learning environment has been developed on the web because it does not require installation according to the desktop and mobile applications. The JavaScript programming language, which can work without the need for plugins, is preferred for playing animations. Intellij as editor, Apache Server for presentation and Mozilla Firefox as browser. In the developed programming learning environment, compiler techniques have been realized by using regular expressions and finite state machines.

2.2. Compiler Techniques

Compilers for each programming language have been developed to make programs written in different programming languages workable on different machines. The compilers convert the source code to the target program [11]. The compilers perform this process in several stages. In theory, these stages are followed in sequence, but in practice this sequence is not always followed, and sometimes the stages can be combined [12]. The operation diagram of compiler is given in Figure 1.

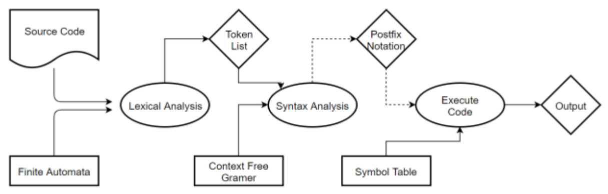


Figure 1. Compiler operation diagram

A basic compiler usually consists of four stages [13]:

Table 1. Token Table

Source Code	Token
text	ID
“text”	STR
number	INT
;	END
,	BR
+	ADD
-	SUB
*	MUL
/	DIV
(	LBT
)	RBT
{	LCB
}	RCB
[	LBB
]	RBB
text = expression	ID EQU EXP
text = expression + expression	ID EQU EXP ADD EXP
text = expression - expression	ID EQU INT SUB INT
text = expression * expression	ID EQU INT MUL INT
text = expression / expression	ID EQU INT DIV INT
expression (logical operators) expression	EXP LOG EXP
eger (logical expression) { code }	IF LBT LEXP RBT LCB CODE RCB
degilse eger (logical expression) { code }	ELIF LBT LEXP RBT LCB CODE RCB
degilse { code }	ELSE LCB CODE RCB
tekraret (number) { code }	FOR LBT INT RBT LCB CODE RCB
oldugu surece(logical expression) { code }	WHL LBT LEXP RBT LCB CODE RCB
[] text = number	LBB RBB ID EQU INT
text[number] = expression	ID LBB INT RBB EQU EXP
program { code }	MAIN LCB CODE RCB
text(expression) { code }	FUNC LBT EXP RBT LCB CODE RCB

1. Lexical Analysis: In the lexical analysis phase, first white characters (space character, tab, new line, etc.) are filtered out. Then the characters in the source code are separated by symbols called tokens. Lexical analysis is also called the stage of preparation for the phase of syntactic analysis [11-13].

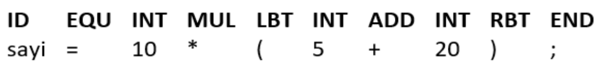
The tokens and their equivalents for the programming language developed within the programming learning environment are given in Table 1. The programming language is fully covered by creating tokens for each command and symbol. The expression in the source code; variable (ID), text (STR), or number (INT). Logical operators only generate conditional expressions. Logical expressions are formed by comparing two expressions with each other. The end of logical statements can be true or false.

$$a = 10 * (5 + 10) \tag{1}$$

An example calculation is given in (1). With the developed programming language, this calculation is performed as follows:

- sayi = 10 * (5 + 20);

In the calculation process of equation 1 is given in the state of Figure 2 was parsed into tokens by the compiler.



ID	EQU	INT	MUL	LBT	INT	ADD	INT	RBT	END
sayi	=	10	*	(	5	+	20	)	;

Figure 2. Token of equation 1

The main purpose of the lexical analysis phase is to facilitate work in the next phase of syntax analysis. Simple systems perform lexical analysis and syntax analysis together. However, the separation of these stages has advantages such as efficiency and modularity [12].

2. Syntax Analysis: The syntax analysis phase parse the token list created during the lexical analysis phase into a tree structure called a syntax tree. For this reason, this stage is also called parsing. If the code is not accepted at this stage, it is indicated as a syntax error together with the corresponding error message. [11-13].

The finite automata in Figure 3 control the tokens given in Figure 2 during the syntax analysis phase. The finite automata changes state according to the token list. The expected tokens(s) for each case vary. An unexpected token in the current situation causes a syntax error. The finite automata is that the encoder accepts the appropriately coded computation. Examining Figure 3, the finite state machine accepts infinite number of left parentheses in case number 2, infinite right parentheses in number 3 case. However, for each left bracket, a right bracket must appear. Parentheses are checked with regular expressions before the computation is verified by the finite automata. Controlling the parentheses first provides ease of operation. If there is an error in the parentheses, a parenthesis error is given without checking at the codes.

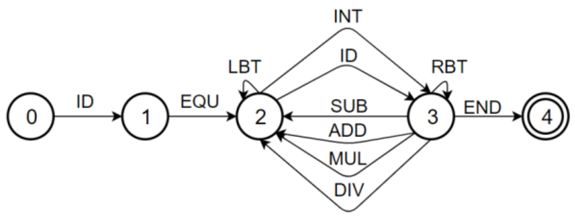


Figure 3. Proposed finite automata for the operations

3. Type Checking: An expression in the code block can be variable, text, or number. When assigning and operator operations are performed, the data type must be decided. Variables (ID) always begin with a character, followed by a number or character. Texts (STR) only accept characters, numbers (INT) only accept numbers [12].

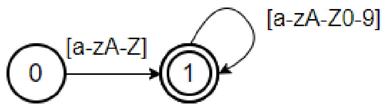


Figure 4. Variable name definition

The finite automata in Fig. 4 is used to verify the variable name. Lowercase letters are represented [a-z], uppercase letters are represented [A-Z] and numbers are represented by [0-9].

4. Code Generation: The ultimate goal of a compiler is to translate code written in a high-level programming language into programs that can run on a computer. The codes written in this context should be transformed into a programming language that can understand the environment in which the work should be performed [13]. This language can be scripting languages for the browser while being machine codes for the computer.

The developed programming learning environment is working on browsers. The code written in the environment needs to be converted into JavaScript, which is a scripting language. The code written in the learning environment is prepared to work in the browser because of the successful passage of the analysis steps. The written code is converted into JavaScript programming language using regular expressions and executed in the browser.

The insertion sorting algorithm function written in the developed programming language is given in Figure 5. This function is tokenize by parsing the lexical and syntax analysis steps into the final tokens. These tokens are converted into JavaScript programming language using regular expressions. The sorting algorithm in the JavaScript programming language is given in Figure 6.

```
sirala([] dizi, n)
{
  olduğu sürece(i < n)
  {
    anahtar= dizi[i];
    j= i - 1;
    olduğu sürece(j >= 0 && dizi[j] > anahtar)
    {
      dizi[j + 1]= dizi[j];
      j= j - 1;
    }
    dizi[j + 1]= anahtar;
  }
}
```

Figure 5. Insertion sort algorithm

```
function sirala(dizi, n)
{
  while(i < n)
  {
    anahtar= dizi[i];
    j= i - 1;
    while(j >= 0 && dizi[j] > anahtar)
    {
      dizi[j + 1]= dizi[j];
      j= j - 1;
    }
    dizi[j + 1]= anahtar;
  }
}
```

Figure 6. Insertion sort algorithm in Javascript

3. System Architecture

The general code notations of the developed environment are given in Table 1. The environment needs to verify these codes in the analysis phase. In the analysis phase, the code written before is parsed into symbols called tokens. The finite automata given in Figures 7-9 in terms of the writing rules verify obtain tokens.

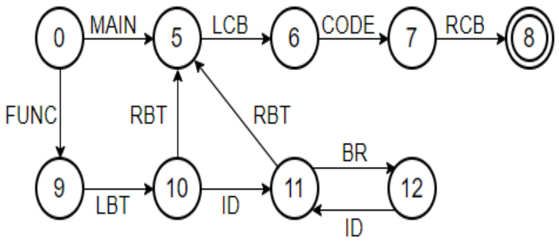


Figure 7. Proposed finite automata for the functions

Function and program (main) blocks are being verified by shown the automata in Figure 7. Functions can take variable(s) as parameters, the program block does not accept parameters. The automata given in Figure 3 verify once the function and program blocks are appropriately generated, the code blocks.

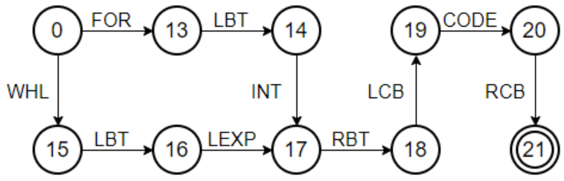


Figure 8. Proposed finite automata for the loops

The automata given in Fig. 8 confirms the loops. Loops; It consists of “tekraret” (for) and “oldugu surece” (while) codes. The for command executes the code in the blocks specified number times, the while command executes the code in the block as long as the given condition is correct.

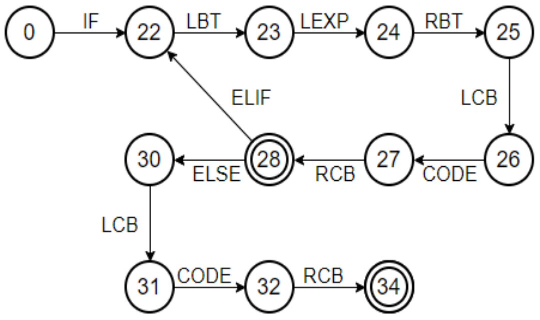


Figure 9. Proposed finite automata for the conditions

The automata given in Fig. 9 checks the conditions. Conditions; “eger” (if), “degilse eger” (else if) and “degilse” (else). Else if and else commands are not accepted by the automata without if command. The else command can come only once, and only at the end, while an unlimited number of else if command can come after if command arrives.

3.1. Task System

In the developed learning environment, the user is progressing successfully by fulfilling the tasks given in the predefined scenarios. In Figure 10, an example task is given.

```
[] dizi= 5;
sayaç= 1;
tekraret(5)
{
   dizi[sayaç]= sayaç;
   sayaç= sayaç + 1;
}
```

Figure 10. Example task code

The automata given in Figs. 11 and 12 control the example given in Fig. 10.

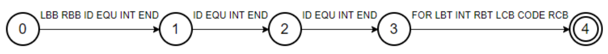


Figure 11. Task Controller

The automata given in Figure 11 control the general structure. The status does not change except for the codes given in the figure. Loop is being verified by shown the automata in Figure 12.

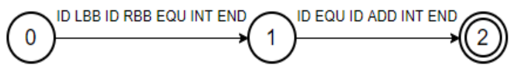


Figure 12. Loop Controller for Task

4. Conclusions

In this study, it was tried to develop an open source programming learning environment using compiler techniques. The developed environment provides a framework for future environments and expected to allow the user to proceed faster to use real programming languages. Through the developed framework, subsequent learning can determine the user's learning ability and deficiencies. Algorithms can provide information on how to develop logic skills. A comprehensive new language can be created with the existing infrastructure.

References

[1] Özyurt, Ö., Özyurt, H., Aras, S. (2016, May). Çocukların Kodlama Öğrenebilecekleri Ortamların İncelenmesi. In International Computer and Instructional Technologies Symposium (ICITS) on (pp. 399-400).

[2] Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Kafai, Y. (2009). Scratch: programming for all. Communications of the ACM, 52(11), 60-67.

[3] Sengupta, P., Farris, A. V., Wright, M. (2012). From agents to continuous change via aesthetics: learning mechanics with visual agent-based computational modeling. Technology, Knowledge and Learning, 17(1-2), 23-42.

[4] Kölling, M. (2010). The greenfoot programming environment. ACM Transactions on Computing Education (TOCE), 10(4), 14.

[5] García, P. G., De la Rosa, F. (2016). RoBlock-Web App for Programming Learning. International Journal of Emerging Technologies in Learning, 11(12).

[6] Bers, M. U., Flannery, L., Kazakoff, E. R., Sullivan, A. (2014). Computational thinking and tinkering: Exploration of an early childhood robotics curriculum. Computers and Education, 72, 145-157.

[7] Papadakis, S., Kalogiannakis, M., Zaranis, N. (2016). Developing fundamental programming concepts and computational thinking with ScratchJr in preschool education: a case study. International Journal of Mobile Learning and Organisation, 10(3), 187-202.

[8] Fessakis, G., Gouli, E., Mavroudi, E. (2013). Problem solving by 5–6 years old kindergarten children in a computer programming environment: A case study. Computers and Education, 63, 87-97.

[9] Armoni, M., Meerbaum-Salant, O., Ben-Ari, M. (2015). From scratch to “real” programming. ACM Transactions on Computing Education (TOCE), 14(4), 25.

[10] Koorsse, M., Cilliers, C., Calitz, A. (2015). Programming assistance tools to support the learning of IT programming in South African secondary schools. Computers and Education, 82, 162-178.

[11] Aho, A. V., Sethi, R., Ullman, J. D. (2007). Compilers: principles, techniques, and tools (Vol. 2). Reading: Addison-wesley.

[12] Mogensen, T. Æ. (2009). Basics of Compiler Design. Torben Ægidius Mogensen.

[13] De Graaf, D. (2017). Practical use of Automata and Formal Languages in the compiler field.