International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 08 Issue: 05 | May 2021 www.irjet.net p-ISSN: 2395-0072

Self-made Programming Language: Alfa

Akshay Kurhekar[1], Devansh Shrivastav[2], Kush Shah[3]

[1,2,3]U.G Students, of Computer Engineering, Sinhgad Institute of Technology, Maharashtra, India

---***---

Abstract - *This paper is primarily intended for the students in computer Engineering. This new language Alfa is a user friendly Object Oriented Programming Language whose syntaxes can be easily understood by the user so that she/he can write the code without any confusion. Specific details can be inherited from existing languages such as java, python, C, C++, etc. and specialized in defining the semantics of various programming languages, and the reusability and modifiability of many programming languages features can be manifested across language paradigms.*

***Key Words*: tokens, keywords, primitive data type, user defined, data type.**

1. INTRODUCTION

The programming language is a formal language comprising a set of instructions that produce various kinds of output. Programming languages are used in computer program to implement algorithms . Most of the programming languages are HLL (High Level Language) which is only understood by the programmers but machine understands 0s & 1s i.e., Binary Language or MLL (Machine Level Language).

Thousands of different programming languages have been created, and more are being created every year. Many programming languages are written in an imperative form while other languages use the declarative form.

Syntax

The syntax of a computer language is the set of rules that defines the combinations of symbols that are considered to be correctly structured statements or expressions in that language. This applies both to programming languages, where the document represents source code, and to markup languages, where the document represents data.

Semantics

The field concerned with the rigorous mathematical study of the meaning of programming languages. It does so by evaluating the meaning of syntactically valid strings defined by a specific

programming language, showing the computation involved. In such a case that the evaluation would be of syntactically invalid strings, the result would be non-computation. Semantics describes the processes a computer follows when executing a program in that specific language. This can be shown by describing the relationship between the input and output of a program, or an explanation of how the program will be executed on a certain platform, hence creating a model of computation.

Imperative programming Language

In computer science, imperative programming is a programming paradigm that uses statements that change a program's state. In much the same way that the imperative mood in natural languages expresses commands, an imperative program consists of commands for the computer to perform

Examples: FORTRAN, AGOL, PASCAL, C etc.

Object Oriented Programming Language

Object-oriented programming is a programming paradigm based on the concept of "objects", which can contain data and code: data in the form of fields, and code, in the form of procedures. A feature of objects is that an object's own procedures can access and often modify the data fields of itself.

Examples: C++, Java, Python, C#, Ruby, SCALA etc.

2. LITERATURE REVIEW

In *A Framework Based on Compiler Design Techniques for Programming Learning Environments* by Sefa Aras, Eyup Gedikli, Ozcan Ozyurt in 2018, we inferred that current programming learning environments are platforms that are done programming with visual components generally. In

such environments, it is not possible to make a syntax error due to the physical structure of the visual components. In these environments, users are aiming to be able to combine visual components rather than programming logic. Therefore, when learners of programming with visual components have difficulties by using real programming languages. To prevent such problems, a new language has been created in the developed learning environment, which has a simple syntax and similar to real programming languages. This programming language is intended for users to learn programming concepts and to be easy to adapt to real programming languages.

In *Programming Languages* by Niyazi ARI, Prof Dr. sch. techno ETH, and Nuraiym Mamatnazarova MsCS in 2014, basic classification of programming languages was explained like declarative and imperative programming languages. Secondly it gives a structure of C-program with a examples. And it covers the logical and functional programming like COlmnon LISP, Standard Meta Languages, and Functional programming with Haskell. Declarative programming languages is illustrated by key concept, and examples with any applications for easy understanding for students.

In *Qualitative Assessment of Compiled, Interpreted and Hybrid Programming Languages* by Ampomah Ernest Kwame, Ezekiel Mensah Martey, Abilimi Gilbert Chris in 2017, The study shown that both compiler and interpreter programming languages have varying advantages and disadvantages when used by programmers to write programs. The interpreter technique is slow and inefficient, since lines of code are repeated and translated while the program is running. However, due to the fact that anytime interpreted language program is run, the interpreter refers to the source code it is a relatively easy to modify and rerun a piece of code or to move the code to a computer environment different where it was developed and run. Interpreters are very good development tools since it can be easily edited, and are therefore ideal for beginners in programming and software development. However they are not good for professional developers due to the slow execution nature of the interpreted code. On the other hand the compiler technique translate the whole code into a single machine code program and run this machine code. Execution of code is very fast when using compiler technique, however, the code cannot be executed on any other platform apart from the one the code was developed on it. The hybrid language combines both techniques and minimizes the disadvantages associated with each of the two techniques while maintaining the advantages they have to some degree. Due to the high execution speed of both compiled and hybrid languages, they are good for professional software developers.

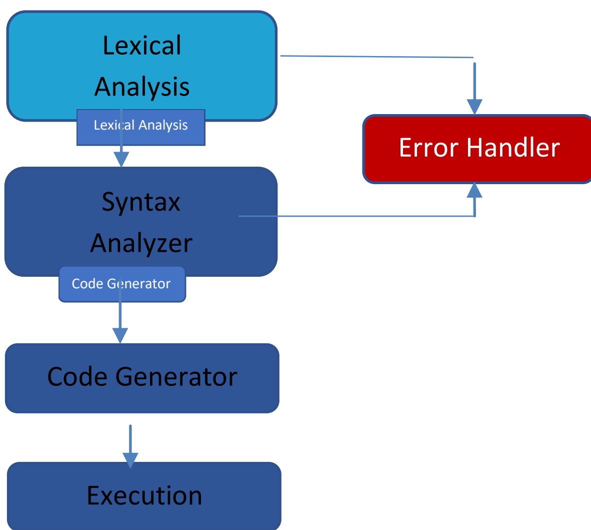
In *Design and Implementation of an Interpreter Using Software Engineering Concepts* by Fan Wu, Hira Narang, Miguel Cabral in 2014, the design of an interpreter for the SimpleC programming language in the context of a software engineering project has been presented. The paper also has demonstrated that some of the standard software engineering concepts such as object-oriented design, design patterns, UML diagrams, etc., can provide a useful track of the evolution of an interpreter, as well as enhancing confidence in its correctness. A similar project could be introduced at Tuskegee University to meet some requirements not satisfied by shorter projects. Some requirements include, but are not limited to, writing a complete project using challenging algorithms and data structures, use of different development tools, objectoriented design, and team management which is an important issue to consider given that only team work in software engineering and database courses.

In *Object Oriented Programming Vs Procedural Programming* by A. A. B. C. B. Adhikari in 2016, the author proposed about the differences between Object-Oriented Programming and Procedural Programming it is obvious that OOP is based on objects and classes while Procedural Programming is based on procedures. Using objects in OOP rather than procedures as in procedural programming allow the developers to reuse a single code anywhere as needed. Thus, allowing coding methods that are more complicated with ease and using less code. When we consider about the security of the data when using either of the programming paradigms, OOP provides more secu-rity as it has a more improved data concealing mechanism rather than procedural programming languages.

3. SYSTEM ARCHITECTURE

As shown in the figure 1 the first step in interpreting our language is Lexical analysis. In this, the interpreter reads the input of characters from the source code and convert it into a stream of tokens by removing any whitespace or comments in the source code. The part of the interpreter that does it is called a **lexical analyzer**, or **lexer** for short. They all mean the same: the part of interpreter that turns the input of characters of the input code into a stream of tokens. It works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer.

Table -1: fig 1



Before you can interpret an expression you first need to recognize what kind of phrase it is, whether it is addition or subtraction, for example. The lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions. Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, this phase uses context-free grammar (CFG), which is recognized by push-down automata. The process of finding the structure in the stream of tokens, or put differently, the process of recognizing a phrase in the stream of tokens is called **parsing**. The part of an interpreter or compiler that performs that job is called a **parser**. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is a **parse tree.**

A parser should be able to detect and report any error in the program based on the parse tree. It is expected that when an error is encountered, the parser should be able to handle it and carry on parsing the rest of the input. Mostly it is expected from the parser to check for errors but errors may be encountered at various stages of the interpretation process. A program may have the following kinds of errors at various stages:

- **Lexical Error**:

  It is encountered when name of some identifier is typed incorrectly etc.

- **Syntactical Error**:

  It is encountered when missing semicolon or unbalanced parenthesis etc.

- **Semantical Error**:

  It is encountered when there is an incompatible value assignment etc.

- **Logical Error**:

  It is encountered when the code not reachable, infinite loop or etc.

  When a parser encounters an error anywhere in the statement, it ignores the rest of the statement by not processing input from erroneous input to delimiter, such as semi-colon. This is the easiest way of error-recovery and also, it prevents the parser from developing infinite loops.

Code generation can be considered as the final phase of interpretation. Through post code generation, optimization process can be applied on the code, but that can be seen as a part of code generation phase itself.

Step 1 : The interpreter reads a source code or instruction. Then it verifies that the instruction is well formatted, i.e. it checks the syntax of each line. If it encounters any error, it immediately halts the translation and shows an error message.
Step 2 : If there is no error, i.e. if the code is well formatted then the interpreter translates it into its equivalent form in intermediate language called "Byte code". Thus, after successful execution of code, it is completely translated into Byte code.
Step 3: Byte code is sent to the Python Virtual Machine(PVM).Here again the byte code is executed on PVM. If an error occurs during this execution then the execution is halted with an error message.

4. COMPARISION

A. Variable Declaration:

1. In C++
In C++ the variable declaration requires some primitive data types such as int, float, char, bool etc.

Syntax: data_type identifier;
example: int a;

2. In python
In python the variable declaration doesn't need a data type before identifier.

Syntax: identifier = value
example: a = 0

3. In Alfa
In Alfa we have a generic data type for declaring variables i.e., **var** because the common mistake every programmer does is giving a wrong data type to the variable so, **var** then automatically assigns the desired data type to the identifier corresponding to the value assigned to it.

Syntax: var identifier = value; # here ' ; ' is optional
example:

```
var abc = 0;
['VAR', 'INTEGER', '=', '0']
```

Here **'abc'** is assigned an integer datatype by virtue of the value given.

B. Selection Statements syntax:

1. In C++

The basic selection statements used in C++ is **if, if ...else** etc.

Syntax:

if (*condition*){// *statements*}

if (*condition*){//*statements*}
else {// *statements*}

if (*condition*){//*statements*}
else if(*condition*){//*statements*}
else{//*statements*}

example:

```
if(2 == 2){cout<<"TRUE";}

if (2 > 3){cout<<"TRUE";}
else{cout<<"FALSE";}

if (2 > 3){cout<<"FALSE";}
else if( 2 == 3){cout<<"FALSE";}
else{cout<<"TRUE";}
```

2. In Python

The basic selection statements used in python is **if, if ...else** etc.

Syntax:

- **if** *condition:*
  # statements
- **if** *condition*:
  #statements

  else:
  #statements
- **if** *condition*:
  #statements
  elif *condition*:
  #statements
  else:
  #statements

example:

- **if** 2 == 2:
  print("TRUE")
- **if** 2 == 3:
  print("FALSE")
  else:
  print("TRUE")
- **if** 2 > 3:
  print("FALSE")
  elif 2 ==3:
  print("FALSE")
  else:
  print("TRUE")

3. In Alfa

The basic selection statements used in Alfa is **if, if ...else** etc. here the syntax used is inherited from C++ and Python for the ease of use of programmers who program in both the languages. As shown in the syntax below we have used curly brackets because it is used for implementing the concept of modularity.
Syntax:

- **if** *condition* {// *statements*}
- **if** *condition*{//*statements*}
  else {// *statements*}
- **if** *condition* {//*statements*}
  elif *condition* {//*statements*}
  else{//*statements*}

example:

- if 2 == 2 {print("TRUE");}
- if 2 > 3 { print("TRUE");}
  else{print("TRUE");}
- if 2 > 3 { print("TRUE");}
  elif 2 == 3 { print("TRUE");}
  else{print("TRUE");}

C. Iteration Statements syntax:

1. In C++

The iteration statements used in C++ are **for, while, do while.** These statements are often referred to as looping statements.

Syntax:

- **while**(*condition*){
  //statements
  }

- **do**{
  //statements
  }**while**(*condition*);

- **for** (*initialization* ; *condition* ; *updation*)
  {
  //statements
  }

2. In Python

The iteration statements used in python are **for, while.** These statements are often referred to as looping statements.

Syntax:

- **while** *condition*:
  //statements

- **for** *condition* :
  //statements

example:

- **while** i < 10:
  print(i)
  i = i + 1

- **for** i in range (1,10) :
  print(i)

3. In Alfa

The iteration statements used in Alfa are **for, while.** These statements are often referred to as looping statements.

Syntax:

- **while** *condition*{
  //statements
  }

- **for** *indemtifier* = *initial_ value* **to** *end_value* **step** *step_value* {
  //statements
  }
  Here **step** and step_value is optional

example:

- **while** i < 10{
  print(i)
  i = i + 1
  }
- **for** i = 0 to 10 **step** 2{
  print(i)
  }

D. Functions :

1. In C++

In C++ a function always returns a value of data type that is mentioned before the function name unless its return type is mentioned as **void**. The return type can be a primitive datatype or a user defined data type.

Syntax:

a) function definition:

return_type function_name (data_type parameter_name){
/ /statements
return *parameter_name;*
}

b) function calling:

function_name (*parameter_name*);

2. In Python

In python it is not necessary that function should always return a value. The return type is not mentioned for a function instead it uses a generic keyword **def.**

Syntax:

a) function definition:

def *function_name (parameters)*:
#statements
return *parameter_name*

b) function calling:
function_name (*parameter_name*)

3. In Alfa

In Alfa **fun** keyword is used to define a function because, in some existing languages the data type of value to be returned does not match the return type of function so in order to overcome this the **fun** keyword is used.

The datatype of formal parameters are not required to be mentioned.

a) function definition:

fun *function_name (parameters)*{
 //statements
 return *parameter_name;*
}

b) function calling:

function_name (*parameter_name*);

E. Error Handling

As we know that in any programming language various types of errors are encountered. Similarly in Alfa when an error occurs, the type of error is precisely mentioned e.g. runtime error, invalid syntax error etc. As well as the traceback call is also shown with line number and the position of error element.

```
Alfa > var a = 20
20
Alfa > var b = 0
0
Alfa > a/b
Traceback (most recent call last):
  File <stdin>, line 1, in <program>
Runtime Error: Division by zero

a/b
  ^
Alfa >
```

F. Execution Statement

1. In C++

compile: g++ *filename*.cpp

execution: ./a.out in Linux and ./a.exe in Windows

2. In Python

python3 *filename*.py

3. In Alfa

run(“*filename.txt*”)

G. Comments

1. In C++

Single line: // *statements*
Multi Line: /* *statements* */

2. In Python

Single line: # *statements*
Multi Line: Python does not really have a syntax for multi-line comments but you can use a multiline string.
“ “ “ *statements* ”””

3. In Alfa Single line : # *statements*

5. CONCLUSION

A new Object-Oriented programming language is developed along with its interpreter while maintaining and implementing the basic syntaxes and semantics. We have overcome some of the constraints of existing programming languages at a basic level to make the language user friendly. Also, we gained immense knowledge about the architecture and methodology of the designing of a compiler. We took references from various available resources that helped us for the successful execution of the project.

6. REFERENCES

[1] Programming Languages, 2014 11th International Conference on Electronics, Computer and Computation (ICECCO), Niyazi ARI, Prof Dr. sch. techno ETH, and Nuraiym Mamatnazarova MsCS

[2] Object-Oriented Software Specification in Programming Language Design and Implementation Proceedings. The Twenty-Second Annual International Computer Software and Applications Conference (Compsac '98) (Cat. No.98CB 36241) Year: 1998 Barrett R. Bryant and Viswanathan Vaidyanathan

[3] An Experience Report on Teaching Compiler Design Concepts using Case-Based and Project-Based Learning Approaches 2016 IEEE Eighth International Conference on Technology for Education (T4E) Divya Kundra and Ashish Sureka

[4] Qualitative Assessment of Compiled, Interpreted and Hybrid Programming Languages, Communications on Applied Electronics (CAE) – ISSN : 2394-4714 Foundation of Computer Science FCS, New York, USA, Ampomah Ernest Kwame, Ampomah Ernest Kwame, Abilimi Gilbert Chris

[5] A Framework Based on Compiler Design Techniques for Programming Learning Environments, 2018 International Conference on Artificial Intelligence and Data Processing (IDAP) Sefa Aras, Eyup Gedikli, Ozcan Ozyurt Software Engineering Department Karadeniz Technical University Trabzon, Turkeyq

[6] Design and Implementation of an Interpreter Using Software Engineering Concepts, (IJACSA) International Journal of Advanced Computer Science and Applications,Fan Wu, Hira Narang, Miguel Cabral