BUILDING A SMALL COMPILER

Student No: 2004595

Ogundiran Jamie

# ABSTRACT

In this paper, I represent a small compiler for a simple source language that translates the source code into Python, the compiler consists of several components, including a lexer, parser, code generator and semantic analyser.

The lexer reads the source code character one by one and generates tokens that represent the meaning of the code. The parser groups the tokens from the lexer into a syntax tree, which maps the structure of the program. The semantic analyser checks for any error and verifies that it is semantically valid. The code generator translates the source code into a python code using visitor, The generated python code is then compiled using a python interpreter.

The resulting compiler is efficient to use, allowing developers to quickly translate simple source code into python. The compiler is also extensible, allowing developers to build extra features on it.

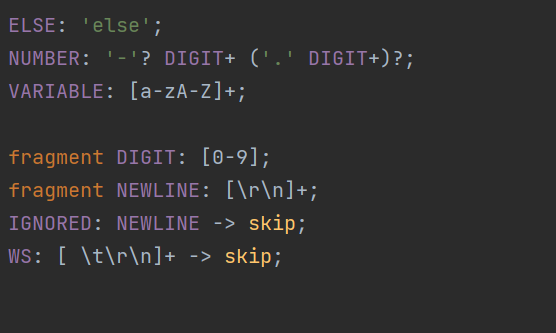
In general, our compiler is useful for any developer who need to translate simple code into python and build on it for more complex compilers.

# INTRODUCTION

In this paper, I will discuss a small compiler for a simple source language that translates code into Python. The compiler is compromised of 4 key components, including a lexer, parser, code generator and semantic analyser, all of which work together to ensure that the resulting Python code is accurate and semantically valid. The use of a visitor for code generation makes the compiler both efficient and extensible, allowing developers to quickly translate simple source code into Python and build on it for more complex compilers. This tool can be especially useful for developers who are new to Python and need to translate code from another language, as well as for those working on more complex projects that require more advanced compilers. In this essay we will, discuss the specification and the implementation of the compiler, and test it using a simple python code.

# SPECIFICATION

## REGULAR EXPRESSION & BNF



The code above displays the regular expression and BNF of my grammar file, it contains NUMBER, VARIABLE, DIGIT, NEWLINE and WS. The regular expressions represent:

Fragment DIGIT: fragment is a way to define a reusable sub-rule, in this case it’s defined as a DIGIT and contains integer [0 - 9].

Fragment NEWLINE: NEWLINE contains [\r\n] which points to next line.

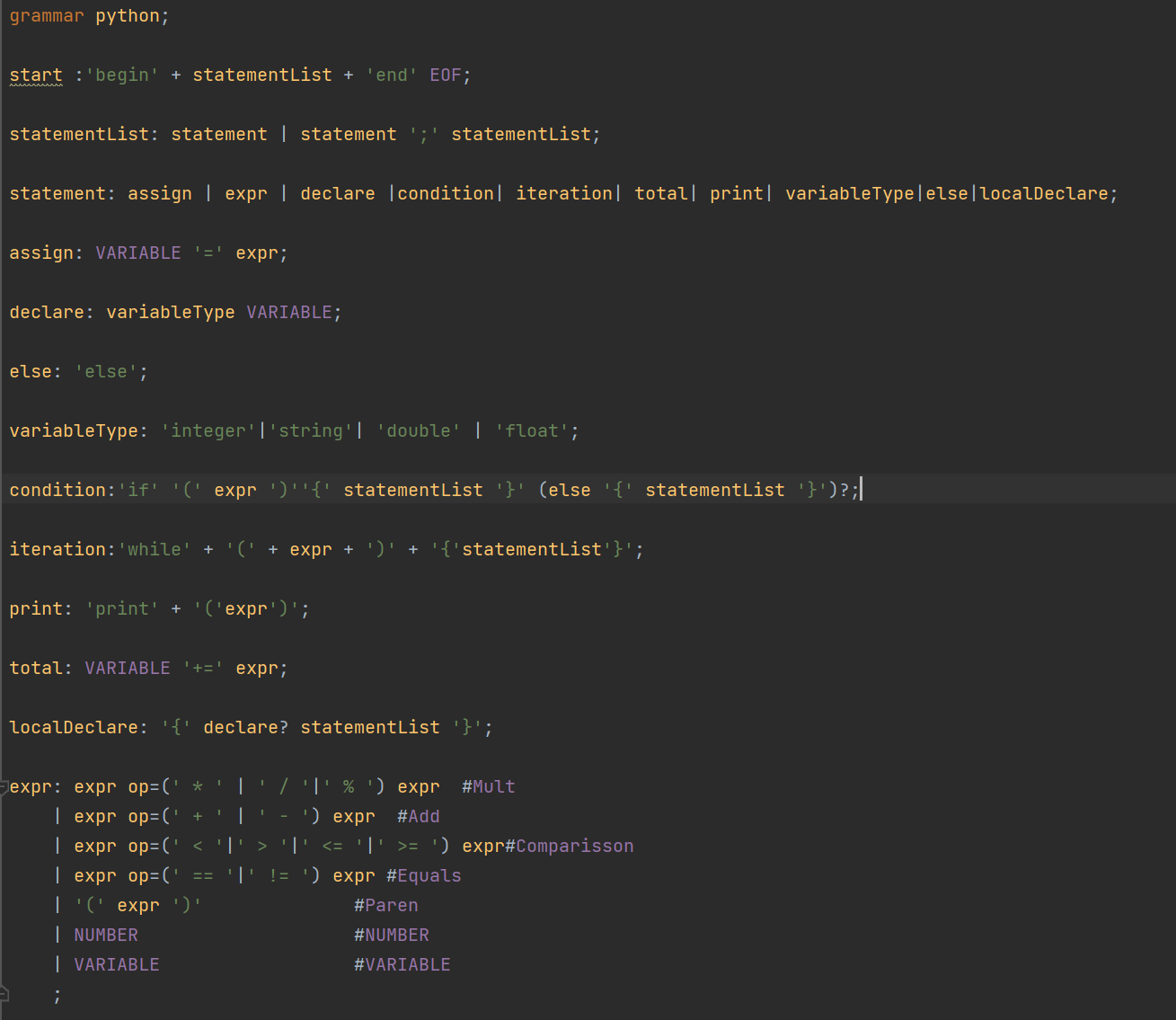
NUMBER: NUMBER allows DIGIT to be represented with a “.” or “-”, this way we can represent float number.

VARIABLE: VARIABLE contains any alphabet character from A – Z. (allows capital and small letters)

IGNORED: IGNORED skips NEWLINE

WS: WS skips any whitespaces

## SOURCE LANGUAGE



The code above displays the grammar expression of the source code, each source code means:

Start: The root of the source code, it contains statementList which we will discuss later, and a text “begin” at the start and “end” and EOF at the end of the rule. The statementList needs to be surrounded with “begin” and “end” for the grammar to work.

StatementList: the function of statementList is to allow sequencing of statements, the rule explains that the rule needs to be in the state of either statement or statement with semicolon at the end followed with statementList.

Statement: consists of rules such as assign, expr, declare, condition, iteration, total, print, variableType, else, localDecl and block. Each statement has its own function

Assign: the function of assign is to assign expression into a variable (Eg. A = x + 12). Assign needs to be written in a form VARIABLE = expr where expr is any variable that matches the rule expr.

variableType: contains a set of identifiers used for declaration. These identifiers being integer, string, double and float.

Declare: the function of declare is declaration of variable to an identifier (Eg. Int n, double n, float n). Declare needs to be written in a form variableType VARIABLE where variableType contains sets of indentifier.

Condition: the function of the condition statement is initiating a statement when the expression of the first statement is true, if not it initiates another statement. The condition rule needs to be in the format if (expr){statementList} else {statementList}. Moreover, the else statement is an option, where the condition rule will still be met with only if (expr){ staementList }.

Else: Else allows only the value ‘else’. It’s used when the if statement returns false.

Iteration: the function of iteration is looping through the statement. The iteration rule needs to be written in the form while(expr){staementList}. The while will loop through the staementList until the expression is met.

Print: the function of print is to print the expr to the terminal. The Print rule needs to be in the form Print (expression).

Total: The function of total is to add the total sum of an expression to a variable. The total rule needs to be in the form VARIABLE += expr.

Block: the function of block is to declare variable locally. The block rule needs to be written in the form {declare statementList}. (Eg. {int a a = 6 + 5})

Expr:

# IMPLEMNTATION

The implementation of the grammar file above is done by generating the antler recognizer on the IntelliJ IDE. After generating the antlr recognizer we get java files including pythonBaseListener, pythonBaseVisitor, pythonListener, pythonVisitor, python tokens file and python interpreter file. These files generate the parse tree for the user to modify and build on it. To create a program that translates the source code into a python code I extended the pythonBaseVisitor to a new java class called prettyPrint. Transversing through the parse tree using visitor I stored each statement component in a string builder. For indent I added it depending on the indent level of the statement. For example, if the statement is a condition statement, the indent level is added by 1 and a space of 1 indent will be added to the string builder. Furthermore, a symbol table which is a data structure used by compilers and interpreters to store information about the variables, was created in the visitor as well. I’ve added any statement used in the source code in a map and added details about it in the map.

For python interpreter that compiles the translated python code and outputs the results, I have used the Jython library in Java.

import org.antlr.v4.runtime.CharStream;  
import org.antlr.v4.runtime.CharStreams;  
import org.python.util.PythonInterpreter;

CharStream in = CharStreams.*fromFileName*(textFile);  
 try (  
 PythonInterpreter pyInterp = new PythonInterpreter()) {  
 pyInterp.exec(in.toString());  
 } catch (Exception e){  
 System.*out*.println("Wrong python code");  
 }  
}

Here is the code for interpreting the translated source code from java to python. CharStream reads the text file and convert it to a value you can manipulate with. After reading the text I initiated PythonInterpreter, then I used the function exec to execute the translated source code in python.

The try catch around PythonInterpreter catches for error and prints out a message wrong python code to the terminal. The error is caught when the inputted code does not match with the python language format.

# TESTING

## Source code translation

p1.  
begin  
n = 12  
previous = 0  
current = 0  
index = 1  
while(index <= n){temp = current;current = current + previous; previous = temp; index+= 1}  
end

To test the translation from the source code to python code, I have created a code which prints out the Fibonacci numbers to the terminal. Upon running the code, I got the result of:

n = 12  
previous = 0  
current = 1  
index = 1  
while index <= n:  
 print(current)  
 temp = current  
 current = current + previous  
 previous = temp  
 index+=1

The translated source code above is a python equivalent to the Fibonacci function.

## Symbol Table

Using the same test subject in source code translation, I displayed the symbol table of the function. Here is the result:

Symbol Table:   
  
n : Variable n is a integer  
while : While is a loop  
Print : Prints the expression or variable to the terminal  
+= : adds a value to an assigned expression as the sum total

The symbol table includes n, while, print and +=, which are the expressions that needs explanation other than the declaration and assignment of variable. The test is a success since it produces every symbol available in the function.

## Python Interpreter

Using the Fibonacci function that was translated from the source code, I tested it on the python interpreter to see if it outputs the correct result.

The expected results are these numbers vertically: 1 1 2 3 5 8 13 21 34 55 89 144

Result:

Python result:   
  
1  
1  
2  
3  
5  
8  
13  
21  
34  
55  
89  
144

As you can see the python interpreter outputted the number 1 1 2 3 5 8 13 21 34 55 89 144 vertically. Hence, we can say that the test was a success.

# CONCLUSION

In conclusion, the presented compiler is a powerful tool for developer who need to translate simple source code into Python. The source code includes the basic structure including variable declaration, variable assignment, iteration, and conditions. The source code could be used to build more complex language. Overall, this compiler hasthe potential to be an asset for the developers to build on it or to translate a simple source code into Python.