

University of Essex

Year 3

Small Compiler – Assignment 2

Nathan Jordan

1905341

CE305 Languages and Compilers

Somdip Dey

Date: 11/03/2023

Contents

1. Abstract.....	5
2. Introduction.....	5
3. Methodology	5
3.1. Language Design and Specification	5
Tokens.....	6
Grammar rules.....	6
Jypa syntax	8
3.2. Python compiler	8
Visual rendering of the AST.....	9
Error handling and type checking	9
Functions.....	10
4. Experimentation & Result.....	10
5. Discussion.....	12
6. Conclusion.....	13
References.....	14
Appendices	15

List of Figures

Figure 1: Snippet of Jypa code for a factorial function.	8
Figure 2: Parse tree displayed.....	9
Figure 3: Error of returned value from function mismatch.....	10
Figure 4: All 1,097 tests passing.	10
Figure 5: Test suite contents.....	10
Figure 6: Reserved word renaming (abs) and wrapping of non-string in string concatenation.....	11
Figure 7: For loop converted into while and increment conversion.	11
Figure 8: Three sample python codes correct output.	12

Abbreviations and Definitions

ANTLR4 – (ANOther Tool for Language Recognition) a parser generator for reading, processing, executing, or translating structured text or binary files.

Java – Programming language

AST – Abstract Syntax Tree.

Regular expression – a language used for pattern matching in text.

EOF – End of File.

Python – Programming language.

Junit – a unit testing library in Java.

1. Abstract

This report is the development of a small compiler which should translate from the new programming language, Jypa, into Python using ANTLR4 and Java. The new language must include the basic notations of variable declarations and assignments, expressions, and sequencing of statements; also, the control flow structure should support conditional statements, unbounded iteration, and statement blocks. Additionally, extended features are included, such as in-depth error handling and type checking, which consists of creating a symbol table to report errors, allowing local variables and declaration, and using doubles, integers, booleans and strings; Jypa also has support for functions with return values and parameters.

For the demonstration that the compiler operates successfully, three pieces of code have been developed and concatenated into one input passed to the compiler, saving the compiled python code into a file and executing it in the python environment.

The findings of this project demonstrate that the compiler is a reliable tool which successfully converts source language code into Python with many tests to verify the language's specification.

2. Introduction

A compiler is software that transforms source code into a different language, which can be compiled into machine-level or high-level languages. First, it uses a set of rules to partition the source code into tokens through lexical analysis or tokenisation. Next, the compiler performs syntax analysis or parsing to check that the grammar/syntax is correct. Once the parsing is complete, the semantic analysis process starts, which involves checking that the source code is semantically correct, and finally, it will translate the input source code into the appropriate target language.

It is well known that Python is one of the first programming languages people learn and is very popular amongst beginners and widely used [1] [2]. However, because it is easy to learn due to being dynamically typed, developers who only know how to program in Python may need to learn what a statically typed language is and its benefits. To combat this and help Python developers branch out their programming knowledge, the target language has been specifically designed to be static so that it can act as an intermediate language when migrating to more popular statically typed languages such as Java and C++.

This project aims to define a new statically typed programming language with the basic features of a modern programming language that is easy to learn and create a compiler that can compile into the more well-known language Python.

3. Methodology

This section will describe the language design, specification, and grammar file of Jypa and the compiler, producing the AST, error handling, type checking and function definitions.

3.1. Language Design and Specification

The following section will describe the rules of the grammar file "Expr.g4" and reference the rules by name (in italics); BNF notation and the grammar file can be found in (Appendix A) [3]. The regular expressions of the grammar file can be found in (Appendix B). The reader is assumed to have a basic understanding of regular expressions (regex).

Tokens

A complete list of the tokens for this language can be found in (Appendix C).

Jypa contains reserved words that cannot be used for variable or function names: null, if, elseif, else, while, for, function, true, false, return, Integer, Double, String and Boolean. Jypa also includes various operators for performing arithmetic, relational, and logical operations. These operators include the arithmetic operators '-', '+', '/', '*', '^', and '%', as well as relational operators like '==', '!=', '<', '<=', '>', and '>=', and logical operators such as '&&' and '||'. Additionally, the '=' symbol is used as an assignment operator and can be combined with any arithmetic operator to form an augmented assignment operator; increment and decrement operators are also supported '++', '--'.

To group expressions and statements and to define the control flow of expressions, the language uses grouping tokens such as '(', ')', '{', and '}'.

Grammar rules

Jypa is broken up into sections called blocks, the rule that controls the flow of the language, as the main scope is a block, each function is a block, and each control flow is a block. In ANTLR4, the grammar file allows for a recursion-type rule setting; for example, an expression can call to itself, meaning it can handle multi-term expressions such as '1 + 2 + 3'.

The grammar file is split into terminals (block capitals) and non-terminal rules.

Non-terminal rules:

Non-terminal rules are rules that can be expanded into other rules however, splitting the regular expressions up into multiple non-terminal rules makes it easier to read, design and implement the grammar.

(Surrounded by ' ' means a string literal)

- *start* – must contain a *block* and an EOF token.
- *Statement* – must be one of the following rules: *ifStatement*, *whileStatement*, *forStatement*, *varAssignment* ';', *functionDef*, and *functionCall* ';'.
- *ifStatement* – should have 'if' with an *expression* surrounded by brackets followed by a *block* surrounded by curly brackets an optional number of 'elseif' which follow a similar pattern and followed by an optional 'else' with a *block*.
- *functionDef* – must have 'function' with a *name* and brackets with *parameterDeclaration* inside, followed by an open curly bracket, followed by a *block* with an optional return statement denoted by 'return' *DATATYPE* and an *expression*.
- *parameterDeclaration* – can contain zero or more *functionParameterAssignment* separated with a comma.
- *functionParameterAssignment* – *DATATYPE* followed by a *name*.
- *functionCall* – *name* followed by zero or more *expressions* to form arguments for the call surrounded with brackets.

- *whileStatement* – must contain 'while' with an *expression* in brackets and a *block* in curly brackets.
- *forStatement* – should have 'for' '(' a *varAssignment*, *expression* and another *varAssignment* separated by semi-colons ')' ' '.
- *block* – can contain any number of *NEWLINES* before or after a *statement* multiple times.
- *varAssignment* –
 - *VarDeclaration* – *DATATYPE ID* with an optional equals *expression*.
 - *Assignment* – *ID* followed by an equal and an *expression*.
 - *AssignmentOperators* – *ID* followed by an augmented assignment operator followed by an *expression*.
 - *AssignmentIncDec* – *ID* followed by either '++' or '--'.
- *expression* –
 - *ExpUnary* – *expression* with a '-' or '+' in front of it.
 - *ExpMult* – either multiplication or exponentiation on two *expressions*.
 - *ExpDivision* – either division or modulus on two *expressions*.
 - *ExpAdd* – either addition or subtraction on two *expressions*.
 - *ExpAndOr* – compares two *expressions* with either 'and' or 'or'.
 - *ExpComparison* – compares two *expressions* with '==', '!=', '<', '<=', '>', or '>='.
 - *ExpNegate* – applies not '!' to an *expression*.
 - *ExpBrackets* – an *expression* surrounded by brackets.
 - *ExpFunctionCall* – calls *functionCall*.
 - *ExpID* – *ID* which is a name for a variable.
 - *ExpNumber* – a number as defined by *NUMBER*.
 - *ExpString* – a string as defined by *STRING*.
 - *ExpNull* – a value of 'null'.
- *name* – an *ID*.

Terminal rules:

Most terminal rules reference string literals, which do not require explanation.

- *ASSIGNOPERATOR* – one of *ADD*, *SUB*, *MUL*, *DIV*, or *MOD* followed by '='.
- *DECREMENT* - '--'
- *INCREMENT* - '++'
- *BOOLEAN* – either 'true' or 'false'.
- *DATATYPE* – one of *Integer*, *Double*, *String*, *Boolean*.
- *ID* – a letter followed by zero or more letters, numbers or underscores.
- *NUMBER* - starts with an optional -/+ prefix, followed by a decimal point with one or more *DIGITS*, one or more *DIGITS* followed by a decimal point or one or more *DIGITS*. This is followed by zero or more *DIGITS* and an optional E notation. E.g.
 - 42, -3.14, 6.4593, 6.01E5, -9.81e-3, .23, -0.000123.
- *DIGITS* – a single digit 0-9.
- *STRING* – ''' zero or more *CHARACTERS* until the end ''' is reached.
- *COMMENT* - '/' skips all characters until the end of the line.
- *WS* – skips the whitespace characters space or tab.

- *NEWLINES* – one or more newline, carriage or *WS* characters
- *CHARACTER* – a single character from the range [a-zA-Z0-9 \!"£\$%^&*()_+={}[\]@~<:?:>,;/.-].

Jypa syntax

Jypa's syntax is relatively synonymous with Java in terms of the prefix of the type in front of each declared variable, curly brackets usage for scope blocks, and semi-colons for lines breaks; however, the inspiration was also drawn from Python when it comes to the simplicity of functions/method as they can be overwhelming in Java, this creates a good learning curve for Jypa. A snippet of Jypa code can be seen in figure 1.

```
function factorial(Integer n) {
    print("Factorial number of " + n + ":");

    Integer val = 1;
    Integer counter = 1;

    while (counter <= n) {
        val*= counter;
        counter++;
    }

    return String val + "\n";
}
```

Figure 1: Snippet of Jypa code for a factorial function.

A difference between Jypa and Python, and Java is that a return type from a function is required at the end of the function, compared to Java, which is contained within the function signature, and Python, which does not require a type. This was designed to have a better flow when reading through code as the return type is alongside the return value, as opposed to Java, and it does not clutter the function signature.

Although Jypa is a statically typed language, it does perform type conversions automatically for some variable/operator configurations. For example, a string being added with any type except null will be concatenated; arithmetic operations between an integer and a double are also allowed; however, the resulting type will be double.

3.2. Python compiler

Before compiling the source code into Python, the source code is tokenised and parsed for grammar errors, after passing the `SyntaxSemanticsVisitor` class visits all the tokens to check for any syntax or semantic errors. Finally, the compiler will call the class `TranslatorVisitor` that performs the conversion into Python. In Python, the indents of each code block are essential, so the indent level is stored and is incremented/decremented whenever the compiler enters or exits a block. Besides that, the compiler appends to a `StringBuilder` until the entire tree has been traversed, returning the string value to the main class.

Visual rendering of the AST

Visualising the AST proved too challenging to implement when using a library to visualise a tree and attempting to create an AST. After six-seven hours of trying, a less correct version of the AST is displayed (parse tree), as seen in figure 2.

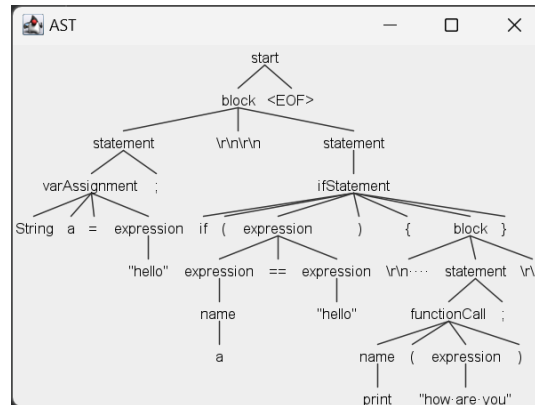


Figure 2: Parse tree displayed.

Error handling and type checking

As mentioned previously, error checking is performed when parsing and when checking for syntax and semantic errors, which gives the error checking a thorough inspection of the source code. In addition, a symbol table is utilised to check the type and scope of variables and the return type and parameters of functions; there is a symbol table for the main body of the source code and a table for every function that gets defined, which means that there can be a local declaration of variables and not just global ones. Finally, the scope level is an instance variable that holds the current scope level of the source code, which is used to clean up variables when the compilation exits that scope level; each time a scope changes, either by entering or exiting a block, the scope and scope level is set to the relevant values.

However, the symbol table is one of many ways to detect type errors. Instead of attempting to detect whether an operation between two variables is correct and then checking if the return type matches with the assignee, a map is populated (typeByString) with all possible variable/operator combinations and the type that will be returned. This way, the compiler can check if the entry is in the map; if it is not, then it is not supported, and receive the value to check with the assignee at the same time. Other maps are included; however, these are mainly for error message outputs.

Functions

Jypa supports user-defined functions with return statements and parameters but also includes several built-in functions such as `print`, `random`, `absInteger`, `absDouble`, and `round`. All function definitions are stored in the main symbol table to be accessed in any scope. Just as with variables, functions must be defined before being referenced.

When calling functions with parameters, the parameter counts and types are compared, ensuring that they both match, throwing an error if not, and the return value is checked if that value is to be assigned or operated on see figure 3.

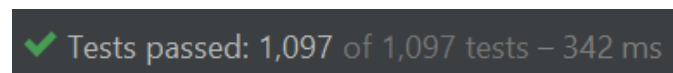
```
function intS() {  
    return Integer 10;  
}  
String testS = intS();  
  
- Checking syntax/semantics  
Syntax/semantic error  
Exception in thread "main" SyntaxSemanticException:  
    Type mismatch: Cannot assign an Integer to the String variable "testS": (StringtestS=intS())  
    Error occurred on line 53: String testS = intS();  
    at input file.(input.txt:53)  
  
    at SyntaxSemanticsVisitor.visitVarDeclaration(SyntaxSemanticsVisitor.java:385)
```

Figure 3: Error of returned value from function mismatch.

4. Experimentation & Result

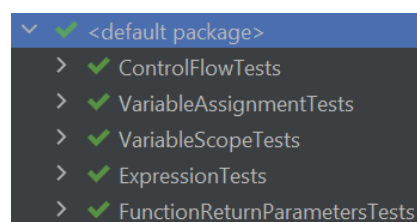
From the result of assignment one, there was a clear indication that there needed to be more testing information. To ensure comprehensive testing coverage, the compiler has an extensive JUnit test suite with 1,097 individual tests, although most test for combinations of type/operator applications. See both figures 4 and 5 for the tests.

There are 1,025 tests in `ExpressionTests` which, for the most part, were created by generating each test as a string, printing all of them out, and copy-pasting them; this was achieved by looping through a combination of types, operators and values.



✓ Tests passed: 1,097 of 1,097 tests – 342 ms

Figure 4: All 1,097 tests passing.



```
✓ <default package>  
  > ✓ ControlFlowTests  
  > ✓ VariableAssignmentTests  
  > ✓ VariableScopeTests  
  > ✓ ExpressionTests  
  > ✓ FunctionReturnParametersTests
```

Figure 5: Test suite contents.

An idea that, if time permit, would have been implemented was unit tests for the compiled Python output, as those tests were carried out by hand. For example, if a variable or function name that is a reserved word in Python must be changed, see figure 6. Another test is for string concatenation; Jypa allows all concatenation with strings; however, Python does not and non-string variables/values must be wrapped with `str()` figure 6.

```
1 String abs = 1 + "str" + true;

- Translation into Python:
abs1 = str(1) + "str" + str(True)
```

Figure 6: Reserved word renaming (*abs*) and wrapping of non-string in string concatenation.

Two more differences between Jypa and Python are that Python does not support increment or decrement operators and does not have a 'for' style loop, so both need to be converted (see figure 7).

```
1 for (Integer i = 0; i < 10; i++) {
2     print(i);
3 }

- Translation into Python:
i = 0
while i < 10:
    print(i)
    i = i + 1
```

Figure 7: For loop converted into while and increment conversion.

To ascertain that the compiler and source code work as expected, three pieces of code (Fibonacci numbers, factorial and odd, even and sum) were concatenated and saved into input.txt, running the compiler compiles the Jypa code into Python, saves it to script.py and runs it in the python environment, which gives the correct output, see figure 8.

```
- Running in python:
Fibonacci numbers:
1
1
2
3
5
8
13
21
34
55
89
144

Factorial number of 8:
40320

Odd, even and sum:
2500
2550
5050
```

Figure 8: Three sample python codes correct output.

5. Discussion

The results from this report show that both the Jypa language, and the compiler are extremely versatile and cover a significant portion of a modern-day programming language and compiler with great accuracy. Using ANTLR4 provides a seamless way of interacting with the parse tree, which is also created by ANTLR; the ANTLR preview is very helpful when adding new functionality to see the actual hierarchy of the source code which is useful when compiling the Jypa code. Similarly, use of Java and IntelliJ has been very helpful, because all the tools that are needed for development are all in one place.

An added feature of the compiler is that it attempts to keep the spacing of the source code when translated into python, which was added before it was realised that a compiler is only interested in the instructions of the code and not the structure. Many improvements can be made to the compiler as some sections of the code are not as organised as others; also, with Jypa, the print function must have a parameter; even if nothing needs to be printed, an empty string is a current solution to this problem, however, to resolve this, the entire symbol table would need to be refactored.

6. Conclusion

In summary, this project was a success that resulted in a hopefully valuable contribution to developing a new statically typed programming language alongside its compiler, which will aid in the educational aspect of broadening software developers' skillset. This new language supports the primary variable assignment and expressions, function declarations, error and type checking, and control flow with code blocks. Paired with this is a compiler that can translate Jypa source code into the prevalent and well-known programming language, Python, which makes it easy to integrate and use the language with existing Python projects and libraries. Finally, the development of Jypa has demonstrated the feasibility and potential of using ANTLR4 for developing compilers and new programming languages.

References

- [1] "The State of Developer Ecosystem 2022," JetBrains, [Online]. Available:
] <https://www.jetbrains.com/lp/devecosystem-2022/>.
- [2] "Stack Overflow Developer Survey," Stack Overflow, [Online]. Available:
] <https://insights.stackoverflow.com/survey/2021#technology-most-popular-technologies>.
- [3] "WebFOCUS Technical Library," [Online]. Available:
] https://ecl.informationbuilders.com/wf/index.jsp?topic=%2Fshell_7703%2Fref%2Fsql%2F03sqlnew2.htm.

Appendices

Appendix A

BNF notation for Expr.g4 grammar file

```

start ::= <block>EOF
statement ::= <ifStatement> | <whileStatement> | <forStatement> | <varAssignment> <SCOLON> | <functionDef> | <functionCall> <SCOLON>
ifStatement ::= <IF> <LBRACKET> <expression> <RBRACKET> <LCBRACKET> <block> <RCBRACKET> [<NEWLINES>] [<ELSEIF> <LBRACKET> <expression> <RBRACKET> <LCBRACKET> <block> <RCBRACKET>]* [<NEWLINES>] [<ELSE> <LCBRACKET> <block> <RCBRACKET>]
functionDef ::= <FUNCTION> <name> <LBRACKET> <parameterDeclatration> <RBRACKET> <LCBRACKET> <block> [<RETURN> <DATATYPE> <expression> <SCOLON>] [<NEWLINES>]* <RCBRACKET>
parameterDeclatration ::= [<functionParameterAssignment>] [<COMMA> <functionParameterAssignment>]*
functionParameterAssignment ::= <DATATYPE> <name>
functionCall ::= <name> <LBRACKET> [<expression>] [<COMMA> <expression>]* <RBRACKET>
whileStatement ::= <WHILE> <LBRACKET> <expression> <RBRACKET> <LCBRACKET> <block> <RCBRACKET>
forStatement ::= <FOR> <LBRACKET> <varAssignment> <SCOLON> <expression> <SCOLON> <varAssignment> <RBRACKET> <LCBRACKET> <block> <RCBRACKET>
block ::= [<NEWLINES>]* [ [<NEWLINES>]* <statement> [<NEWLINES>]*]*
varAssignment ::= <DATATYPE> <ID> [<EQUALS> <expression>] | <ID> <EQUALS> <expression> | <ID> <ASSIGNOPERATOR> <expression> | <ID> [<INCREMENT> | <DECREMENT>]
expression ::= [<ADD> | <SUB>] <expression> | <expression> [<MULTIPLY> | <POW> | <DIVI> | <MOD>] <expression> | <ADD> | <SUB> | <AND> | <OR> | <EQUALTO> | <NOTEQUAL> | <GREATERTHAN> | <GREATEREQUAL> | <LESSTHAN> | <LESSEQUAL>] <expression> | <NOT> <expression> | <functionCall> | <BOOLEAN> | <STRING> | <NULL> | <name> | <NUMBER> | <LBRACKET> <expression> <RBRACKET>
name ::= <ID>

ASSIGNOPERATOR ::= {ADD | SUB | MUL | DIV | MOD} EQUALS
DECREMENT ::= <SUB> <SUB>
INCREMENT ::= <ADD> <ADD>
NULL ::= 'null'
SCOLON ::= ';'
IF ::= 'if'
ELSEIF ::= 'elseif'
ELSE ::= 'else'
WHILE ::= 'while'
FOR ::= 'for'
LBRACKET ::= '('
RBRACKET ::= ')'
LCBRACKET ::= '{'
RCBRACKET ::= '}'
EQUALS ::= '='
EQUALTO ::= '=='
NOTEQUAL ::= '!='
GREATERTHAN ::= '>'
GREATEREQUAL ::= '>='
LESSTHAN ::= '<'
LESSEQUAL ::= '<='
NOT ::= '!'
AND ::= '&&'
OR ::= '||'
MUL ::= '*'
POW ::= '^'
DIV ::= '/'
MOD ::= '%'
ADD ::= '+'
SUB ::= '-'
FUNCTION ::= 'function'
BOOLEAN ::= {'true' | 'false'}
COMMA ::= ','
RETURN ::= 'return'
DATATYPE ::= {'Integer' | 'Double' | 'String' | 'Boolean'}
ID ::= [a-zA-Z][a-zA-Z0-9]*
NUMBER ::= ['-'|'+'] {'<DIGITS>+ | <DIGITS>+ '.' | <DIGITS>+ [<DIGITS>]* ['e'|'E'] ['-'|'+'] <DIGITS>+}
DIGITS ::= [0-9]
STRING ::= '"' [[CHARACTER]*] '"'
COMMENT ::= ('/'/' ~[\r\n]*) -> skip
WS ::= [ \t]+ -> skip
NEWLINE ::= ('\n'|'\r')+
CHARACTER ::= [a-zA-Z0-9 \!\"£$%^&*() +={}|\@~<.:;./-]

```

Appendix B

Regular expression from Expr.g4 grammar file

```
grammar Expr;

start: block EOF;

statement: ifStatement
        | whileStatement
        | forStatement
        | varAssignment SCOLON
        | functionDef
        | functionCall SCOLON
        ;

ifStatement: IF LBRACKET expression RBRACKET LCBRACKET block RCBRACKET NEWLINES?
            (ELSEIF LBRACKET expression RBRACKET LCBRACKET block RCBRACKET)* NEWLINES?
            (ELSE LCBRACKET elseBlock=block RCBRACKET)? ;

functionDef: FUNCTION identifier=name LBRACKET parameterDeclatration RBRACKET LCBRACKET block
            (RETURN DATATYPE returnValue=expression SCOLON)? NEWLINES* RCBRACKET ;

parameterDeclatration: functionParameterAssignment? (COMMA functionParameterAssignment)* ;

functionParameterAssignment: DATATYPE identifier=name ;

functionCall: identifier=name LBRACKET expression? (COMMA expression)* RBRACKET ;

whileStatement: WHILE LBRACKET condition=expression RBRACKET LCBRACKET block RCBRACKET ;

forStatement: FOR LBRACKET varInitialise=varAssignment SCOLON condition=expression SCOLON
            step=varAssignment RBRACKET LCBRACKET block RCBRACKET ;

block: NEWLINES* | (NEWLINES* statement NEWLINES*)* ;

varAssignment:
    DATATYPE identifier=ID (EQUALS value=expression)? # VarDeclaration
    | identifier=ID EQUALS value=expression # Assignment
    | identifier=ID op=ASSIGNOPERATOR value=expression # AssignmentOperators
    | identifier=ID op=(INCREMENT | DECREMENT) # AssignmentIncDec
    ;

expression:
    op=(ADD | SUB) expression # ExpUnary
    | left=expression op=(MUL | POW) right=expression # ExpMult
    | left=expression op=(DIV | MOD) right=expression # ExpDivision
    | left=expression op=(ADD | SUB) right=expression # ExpAdd
    | left=expression op=(AND | OR) right=expression # ExpAndOr
    | left=expression op=(EQUALTO | NOTEQUAL | GREATERTHAN | GREATEREQUAL | LESSTHAN |
    LESSEQUAL) right=expression # ExpComparison
    | negate=NOT expression # ExpNegate
    | LBRACKET expression RBRACKET # ExpBrackets
    | functionCall # ExpFunctionCall
    | BOOLEAN # ExpBoolean
    | name # ExpId
    | NUMBER # ExpNumber
    | STRING # ExpString
    | NULL # ExpNull
    ;

name: ID ; //Something to do with not printing out in an error message, since ID is a
terminal

ASSIGNOPERATOR: (ADD | SUB | MUL | DIV | MOD) EQUALS ;
DECREMENT: SUB SUB ;
INCREMENT: ADD ADD ;
NULL: 'null' ;
COLON: ':' ;
SCOLON: ';' ;
IF: 'if' ;
ELSEIF: 'elseIf' ;
```



```

ELSE: 'else' ;
WHILE: 'while' ;
FOR: 'for' ;
LBRACKET: '(' ;
RBRACKET: ')' ;
LCBRACKET: '{' ;
RCBRACKET: '}' ;
EQUALS: '=' ;
EQUALTO: '==' ;
NOTEQUAL: '!=' ;
GREATERTHAN: '>' ;
GREATEREQUAL: '>=' ;
LESSTHAN: '<' ;
LESSEQUAL: '<=' ;
NOT: '!' ;
AND: '&&' ;
OR: '||' ;
MUL: '*' ;
POW: '^' ;
DIV: '/' ;
MOD: '%' ;
ADD: '+' ;
SUB: '-' ;
FUNCTION: 'function' ;
BOOLEAN: 'true' | 'false' ;
COMMA: ',' ;
RETURN: 'return' ;

DATATYPE: ('Integer' | 'Double' | 'String' | 'Boolean') ;
ID: [a-zA-Z][a-zA-Z0-9_]* ; //Identifier names
NUMBER: ('.' DIGITS+ | DIGITS+ '.' | DIGITS+) DIGITS* ([eE][+]? DIGITS+)? ;
DIGITS: [0-9] ;
STRING: '"' CHARACTER*? '"' ;

COMMENT : ('/' ~[\r\n]*) -> skip;
WS: [ \t]+ -> skip; //Whitespace characters
NEWLINES: ('\n' | '\r' | WS)+ ;
CHARACTER: [a-zA-Z0-9 _\!"#$%^&*()_+={}[\]@~<:;>,;/.-] ;

```

Appendix C

Tokens of the language

Token	Value	Integer value
ASSIGNOPERATOR	1	1
DECREMENT	2	2
INCREMENT	3	3
NULL	'null'	4
COLON	':'	5
SCOLON	','	6
IF	'if'	7
ELSEIF	'elseif'	8
ELSE	'else'	9
WHILE	'while'	10
FOR	'for'	11
LBRACKET	'('	12
RBRACKET	')'	13
LCBRACKET	'{'	14
RCBRACKET	'}'	15
EQUALS	'='	16
EQUALTO	'=='	17
NOTEQUAL	'!='	18
GREATERTHAN	'>'	19
GREATEREQUAL	'>='	20
LESSTHAN	'<'	21
LESSEQUAL	'<='	22
NOT	'!'	23
AND	'&&'	24
OR	' '	25
MUL	'*'	26
POW	'^'	27
DIV	'/'	28
MOD	'%'	29
ADD	'+'	30
SUB	'-'	31
FUNCTION	'function'	32
BOOLEAN		33
COMMA	','	34
RETURN	'return'	35
DATATYPE	36	36
ID	37	37
NUMBER	38	38
DIGITS	39	39
STRING	40	40
COMMENT	41	41
WS	42	42
NEWLINES	43	43
CHARACTER	44	44