INTERPRETATION OF SIMPOL CODE USING C#

Ermitaño, Jeano Frederick U.

Student at the University of the Philippines Open University (PHILIPPINES)

Abstract

This research project will detail the successful SIMPOL code interpretation, developed with Windows Forms and powered by Microsoft C#. The output of the application will be three things: the symbol table, the token table, and the result of the operations within the SIMPOL code, if any. The paper will highlight the various compilation phases, mainly lexical, syntax, and semantic analysis, and the methodologies used to implement them. Conditional statements were used for the lexer, and a recursive descent parser or RDP was used for the parser. Data tables were used for token and symbol management. Code snippets and sample results will be provided to aid the reader, as well as test cases for validation. A state transition diagram will also be given to supplement the understanding of the lexical analysis phase. Lastly, instructions to run the program will be detailed thoroughly.

1. INTRODUCTION

Interpretation is one of two ways to implement programming languages. It takes a completely different language and translates it to the target language line by line, which is then used to execute the commands written in the interpreted language. In our case, SIMPOL is our interpreted language and C# is our target language. Interpretation is generally slower than compilation, but it is easier to write, thus we chose to follow interpretation. C# is the chosen target language because I am comfortable in the Visual Studio environment, having built several applications in it.

1. SIMPOL Specification

The language requires two code blocks: the *variable* block and the *code* block. The *variable* block holds variable declarations only. Variable initialization is not supported, and assignment is done inside the *code* block. The *code* block contains various operations supported by SIMPOL such as asking for input and printing values. Arithmetic operations, numeric predicates, logical operations, and assignment statements are also supported. In Fig. 1, we can see the structure of a SIMPOL code.

```
variable {
INT a
}
code {
ASK a
PRT SUB 100 a
}
```

Fig. 1: Sample SIMPOL code

1. Variable Block

All variable declarations should be in this code block. The syntax for the block is: variable<space><left_curly_brace><newline><variable_declarations><newline><right_curly_brace>. The specific format for variable declaration is <datatype><space><variable_name>. SIMPOL supports only three datatypes: integers, strings, and Boolean. An example integer variable would be: INT aNumber. A variable of type string would be: STG someString. A Boolean variable would look like: BLN imABoolean. Variable names can be of any character length as long as it is in the English alphabet. Lastly, variable names cannot start with a number or any symbol other than a letter.

2. Code Block

The code block houses the core of the language; it is where operations are written. Inside this block, only user input/output operations are allowed, as well as variable assignment. Similar to the variable block, the code block should follow this syntax: code<space><left_curly_brace><newline><operations><right_curly_brace>. An input statement follows this format: ASK<space><variable_name>. An input statement in code would look like: ASK myVariable. Meanwhile, an output statement follows this format: PRT <operations>. We will discuss more about the print statement in the next sub-subsection. An output statement would look like: PRT SUB 5 3, that would print the result of 5 - 3 which is 2. A variable assignment would follow this format: PUT<space><expression or value><space>IN<space><variable_name>. A sample variable assignment would look like: PUT 21 IN someNumberVar. Assignment operations will be discussed in the succeeding sub-subsections.

3. PRT Statement

The PRT or output statement holds all arithmetic and logical operations, as well as numeric predicates. Apart from that, the PRT function can simply accept numbers, strings, and Boolean values which is then printed for the user. It should follow this format: PRT<space><operations>. It can be as simple as printing the value of a variable, PRT numberVariable, or complex by using nested operations: PRT AND GRE 5 2 AND true EQL ADD 20 1 SUB 22 1, which would print "true." We will discuss this specific example in the succeeding sub-subsections

4. Assignment Statement

Assignment statements take an operation or a series of them and assign the result to the provided variable name. Alternatively, this statement also accepts integer, Boolean, and string values. PUT 500 IN a and PUT NON false IN b are valid assignment expressions, as well as PUT OHR NON true LET 98 102 IN c, which is a logical operation comparing another logical operation and a numeric predicate.

5. Arithmetic Expression

Addition, subtraction, multiplication, division, and modulo operations are supported by SIMPOL. In our language, we do not denote arithmetic operations with the usual plus (+), minus (-), asterisk (*), forward slash (/), and percent (%) signs. Instead, SIMPOL uses the keywords ADD, SUB, MUL, DIV, and MOD. Furthermore, ordinary arithmetic operations will generally look like 5 + 7 which uses infix notation; SIMPOL utilizes the prefix or Polish notation. The Polish notation or PN attaches the operators to the left of their operands. In the example earlier: 5 + 7, in Polish notation it will look like: + 5 7. In SIMPOL, it will look like: ADD 5 7. This operation accepts integers and variables of type integer, and will always return an integer. Therefore, this operation be used in conjunction with other operations that can handle integers.

6. Numeric Predicate

Numeric predicates take two expressions that can be an arithmetic operation, an integer variable, or an integer value. Similar to arithmetic operations, numeric predicates also work in the Polish notation, and it also substitutes the regular greater than (>), greater than or equal to (>=, ≥), less than (<), less than or equal to (<=, ≤), and equal (=) signs for GRT, GRE, LET, LEE, and EQL symbols respectively. A sample numeric predicate will look like: LET 64 32, which returns false. As mentioned in the previous sub-subsection, an arithmetic operation can be placed as one of the operands. Thus, a numeric predicate with an arithmetic operation can look like: GRT numberTen ADD 2 7, which returns true if the variable numberTen is equal to 10 and above. This operation will always return a Boolean value and can be used together with operations that handle Boolean values.

7. Logical Expression

Two Boolean values are compared in this operation. Here, numeric predicates, Boolean variables, and true/false keywords are accepted operands. Much like the previous operations, logical operations are also in Polish notation and swaps the usual double ampersand (&&), double pipe (||), and exclamation mark (!) for AND, OHR, and NON symbols. AND and OHR symbols take two Boolean values, whereas the NON keyword takes only one Boolean value. An example logical operation would be: AND false true, which return false. A NON statement would be NON true, that returns the opposite of true which is false. Numeric predicates are valid operands in this operation, and a logical operation with a numeric predicate will look like: OHR LEE 15 14 NON true, which compared the numeric predicate

with another logical operation that is NON true. The example will return false. This operation will always return a Boolean value.

8. Nested Operations

Nested operations are supported by SIMPOL, as long as the expected input datatypes are valid. In the last example in the PRT Statement sub-subsection, we see this code: PRT AND GRE 5 2 AND true EQL ADD 20 1 SUB 22 1. The main operation is a print statement, which prints the result of the logical operation AND. The value to be printed will be the result of the AND expression, that takes a GRE numeric predicate and another logical expression. The GRE numeric predicate will take the integers 5 and 2, which equals to true. The AND expression after the GRE numeric predicate takes the keyword true and another numeric predicate that is EQL. The EQL numeric predicate takes two arithmetic expressions, ADD and SUB. The ADD expression will take integers 20 and 1, which equals to 21. The SUB expression will take integers 22 and 1, which also equals to 21. The EQL numeric predicate will then compare the results of the ADD and SUB: 21 and 21, and will return true. The AND expression that holds the keyword true and the EQL predicate will then compare true and the result of the EQL predicate that is also true, thus the AND expression will return true. Finally, the first AND expression will compare the result of the GRE predicate, which was true, with the result of the second AND expression, which is also true, and the whole expression will return true. Thus, true will be printed.

9. Integer, String, and Boolean Values

SIMPOL will only recognize positive integers, that is 0 and upwards. For a string to be valid, it must be enclosed in dollar signs ($). Inside the dollar signs can be any Unicode character. With that said, the string $This is my string$ is valid, as well as $Hello World!$ and $p3n_p1n34ppl3_4ppl3_p3n$. Lastly, Boolean values will be the keywords true and false.

10. File Format

SIMPOL can be written in any text editor including Notepad. Just remember to save the file with a ".SIM" extension, otherwise the interpreter will not recognize the file. The interpreter is designed to accept only files ending with .SIM.

2. SIMPOL Grammar

Now that we know the specifications of the language, it's capabilities and limitations; we need to define a grammar for our parser to use. I have transcribed the specifications into a complete grammar in Extended Backus-Naur Form[1] or EBNF that you can see in Figure 2:

Program ::= variable { (<VariableDeclaration>)* } code { (Print | Ask | AssignmentStatement)* }

<declaration> ::= (INT | STG | BLN) <ID>

<ID> ::= <Alphabet>(<Alphanumeric>)*

<Alphabet> ::= 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z'

<Alphanumeric> ::= <Alphabet> | <Number>

<Number> ::= ('0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9')+

<Ask> ::= ASK <ID>

<Print> ::= PRT (<ArithmeticExpression> | <NumericPredicate> | <LogicalOperation> | <String> | <ID>)

<AssignmentStatement> ::= PUT (<ArithmeticExpression> | <NumericPredicate> | <LogicalOperation> | <String> | "true" | "false") IN <ID>

<ArithmeticExpression> ::= ((ADD | SUB) <Term> <Term>)* | <Term>

<Term> ::= ((MUL | DIV | MOD) <Factor> <Factor>)* | <Factor>

<Factor> ::= <Number> | <ArithmeticExpression> | <ID>

<String> ::= $<Unicode>$

<Unicode> ::= any unicode character

<NumericPredicate> ::= (GRT | GRE | LET | LEE | EQL) <ArithmeticExpression> <ArithmeticExpression>

<LogicalOperation> ::= (AND | OHR) <ParseLogical> <ParseLogical> | NON <ParseLogical>

<ParseLogical> ::= <NumericPredicate> | <LogicalOperation> | "true" | "false" | <ID>

Fig. 2: Complete grammar for the SIMPOL language in EBNF notation

Kindly refer back to the specifications to help you understand the grammar. For a quick review on EBNF notation, pipes or '|' characters mean alternation. Strings enclosed in angle brackets are non-terminals, and anything else that are not enclosed in the brackets are terminal symbols. Grouping is done by placing the elements inside parentheses. Asterisks symbolize repetition of a terminal, non-terminal, or a group of terminals and/or non-terminals of zero or more times. Plus or '+' signs mean repetition of one or more times. Furthermore, <Alphanumeric>, <Number>, <Alphabet>, and <Unicode> will only serve as an aid for grammar readability. These will be replaced with built-in C# functions such as char.IsDigit(), char.IsLetterOrDigit(), char.IsLetter(), etc.

2. METHODOLOGY

1. Integrated Development Environment (IDE)

For my IDE, I used Visual Studio 2015 running on a Windows 10 machine. I developed the application using Windows Forms in C# to provide a graphical user interface or (GUI) to our interpreter. This is done so that it is easier for the user to select the .SIM file and to visualize the token and symbol tables as actual tables represented inside data grids. I find file browsing inside a console application to be cumbersome.

2. Running The Interpreter

After writing your SIMPOL code and it is time to interpret it, run the SIMPOL.exe application. Click on the "Open File" button and navigate to the folder containing your .SIM file. Then, click on the "Interpret" button and it will start the interpretation process. Please be reminded that interpreters will translate the code line by line, therefore any errors on the current line will be alerted to the user via an error prompt. In some cases, errors might cause the application to crash. When this incident happens, it is best to contact me using this email: jeano.ermitano@live.com. Please include the error message as well as the .SIM file so I may troubleshoot and fix this issue. Nonetheless, when the interpreter reads an ASK statement, the application will show a dialog box prompting the user to type in a value for the current variable. Keep in mind the rules for integer, string and Boolean values. When the interpretation is finished, the textbox on the left will show the contents of the .SIM file as well as the result of the operations. Any printed values will be shown here. To the right of this textbox will be two data grids stacked on top of each other. The data grid on top contains any symbols or variables that were or were not used during interpretation. The bottom data grid contains the tokens found by the lexical analyzer.

3. The Lexer

Our lexer class has 6 methods. We will discuss each method and the class construction in the next sub-subsections. For a visual representation, here in Figure 3 is a state transition diagram for our lexer:

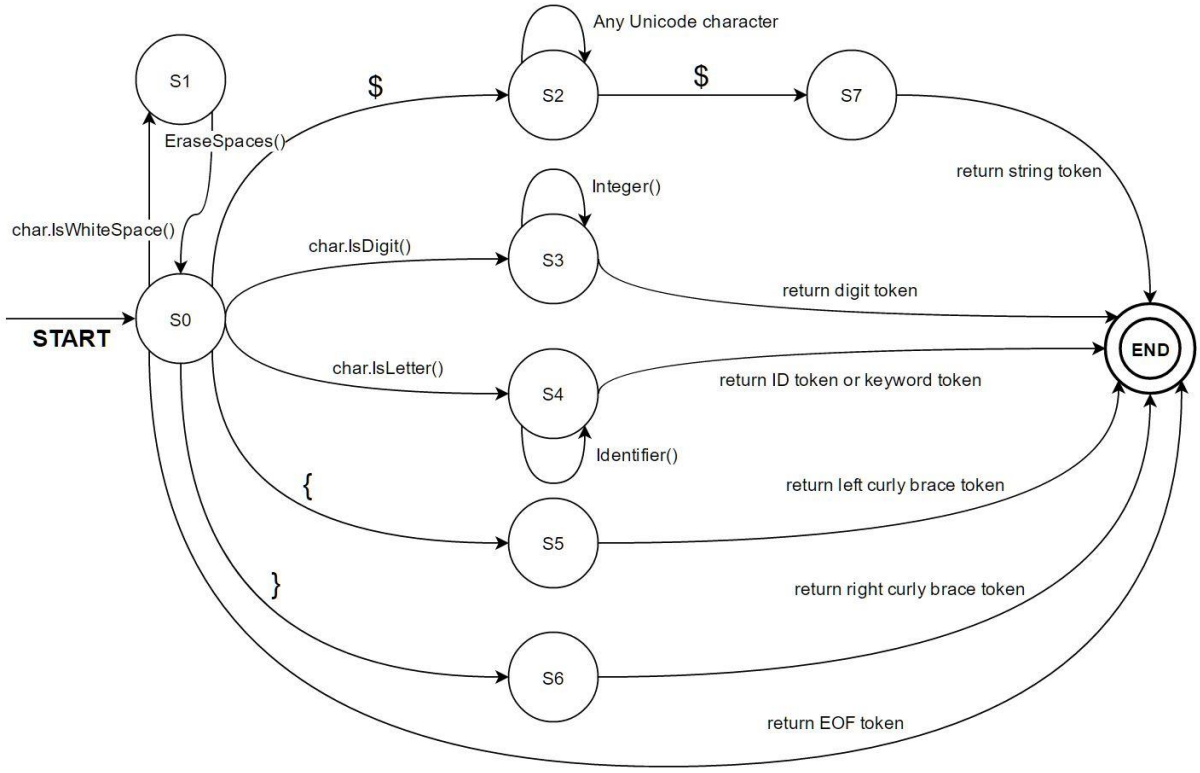

Fig. 3: State transition diagram of lexical analyzer

1. Lexer()

This is the class's constructor with one parameter. It accepts a string, which will be the content of the .SIM file. This will only be used once when the user clicks the "Interpret" button. It will also set the instance variables text to inputText, position to 0, and currentChar to text[position] which is the first character in the inputText. These instance variables will be used by the methods

2. GetNextToken()

This method will do most of the token processing. Given the instance variable currentChar, it will identify if the character is a whitespace character, a digit, a letter, a dollar sign '$', a left curly brace '{', or a right curly brace '}' while it is not null. To save us the trouble of using regular expressions to classify the characters, I used C#'s built-in char methods: char.IsWhiteSpace(), char.IsDigit(), char.IsLetterOrDigit(), char.IsLetter(). An object of type Token will be returned by this method and will subsequently add the token to the token table.

3. EraseSpaces()

This method will erase any whitespace between two characters. When it encounters a newline character, the loop will break, signalling the end of the current line being processed.

4. Advance()

This method will increment the position counter as well as selecting the next character in the text string variable.

5. Identifier()

This method will try and build a word by examining the current character if it is a letter or a number, add it to the local string variable result if it is, and use the Advance() method to get the next character until it hits a whitespace character. After it has built the word, it will then check if it exists inside the ReservedKeywords dictionary, returning the keyword token if it is, otherwise it will return an ID token, signifying that the word is an ID or a variable name.

6. Integer()

Similar to the Identifier() method, this method will try and build a multi-digit number. This method will return the resulting multi-digit number as a string.

7. ParseString()

This method acts much like the Identifier() method. This method will try and build a word until it hits a newline character or a dollar sign '$'. This will return a string token and will add the token to the token table.

4. The Parser

We will be using a top-down parser, specifically a Recursive Descent Parser or RDP. An RDP works it way from the highest level of the parse tree to the lowest level. This will make use of recursive procedures, hence the name. This kind of parser is easy to implement once you complete the grammar for the language because each production in the grammar will be a procedure in the parser. Although tools are available such as ANTLR, JavaCC, etc. which will write the RDP for you, it is better to create your own so that you can essentially grasp the logic behind recursive descent parsers. I chose to handwrite the parser, with the help of Ruslan Spivak's "Let's Build a Simple Interpreter"[2] guide. However, I had to manually translate the Python codes in the guide into C#, which is slightly difficult for someone who hasn't touched Python before. I managed to follow through his guide until part 7[3], after which he modifies the code to support Abstract Syntax Trees or ASTs to parse complex languages such as Pascal. Details about ASTs will be discussed in the Semantic Analysis subsection. Getting back to our parser, it will use 12 methods. Let us define our class's constructor and instance variables.

1. Interpreter()

This is our constructor and it will get passed a lexer object, which will be assigned to the lexer instance variable. Also, the current token will be assigned the returned token by GetNextToken() method, which is the first token in the .SIM text file.

2. Eat()

This method will set the current token to the next token by comparing the current token's type to the parameter token type, and gets the next token if it matches. Remember that current token is an instance variable, and is assigned the first token in the SIMPOL code. The very first initialization of this method will compare the first token's type (which might be "variable", according to our grammar) with the passed token type, and it will update the instance variable with the next token. Eat will only be used to consume terminal symbols.

3. Program()

This method is equal to the start symbol of our grammar which is "Program." Do have a copy of the grammar I gave in the previous section, as it can help in comprehending this method and the succeeding methods. Recall that the "Program" start symbol in the grammar produces "variable { (<VariableDeclaration>)* } code { (Print | Ask | AssignmentStatement)* }." Inside the Program() method, we can find the Eat(Token.TokenVariable) code, which means it will consume the current token, add it to the token table, and set the current token to the next token. Token.TokenVariable is simply a string constant with the value "variable." As I stated in the Eat() method, the very first token should be "variable," hence the first use of the method should have the parameter Token.TokenVariable, or simply "variable". This will check the SIMPOL code in the text file if it complies with the grammar. Our grammar states that the word "variable" is the first word and token. If our parser encounters a word that isn't "variable," it will pop up an error. Moreover, you might notice the while loop after consuming the first left curly brace. Revert back to our grammar and you will see: (<VariableDeclaration>)*. As I said before, asterisks denote repetition by zero or more times. While our parsers does not encounter the right curly brace character, it will proceed to the VariableDeclaration() method. The same goes for the Print(), Ask(), and AssignmentStatement() methods inside the code block.

4. VariableDeclaration()

This method will compare the current token's type if the variable being declared is of type "INT", "BLN" or "STG" and adds a symbol to the symbol table with the datatype and the variable name, leaving the value column null because we are just declaring variable names and their datatypes.

5. Print()

This method will check if the current token's type corresponds to the given keywords and calls the appropriate method to complete the task. The result of the called method will be added to the output local variable and is then printed to the user. It can also print strings, but will trim the starting and ending dollar signs; and it can also print a variables value.

6. Ask()

This method will ask for a user's input. This is achieved by displaying a dialog box prompting the user to enter a value for the current variable. After entering a value, the symbol is edited to add the new value.

7. AssignmentStatement()

This method will assign a value to a variable by checking the current token's type if it matches with a keyword, after which the corresponding method will be called to handle the operation. This behaves similarly to the Print() method, but this method won't accept a variable, therefore assigning a variable's value to another variable is not possible. The result of any operations will be assigned to the variableValue local variable. The symbol's value will then be updated using this variable.

8. ArithmeticExpression()

This method will only handle addition and subtraction operations, otherwise, the Term() method will be called. This is done so that correct operator precedence is followed. The result instance variable will be updated with the result of the operation, or the result of the Term() method, which is the return value.

9. Term()

This method will only handle multiplication, division and modulo operations, otherwise, the Factor() method will be called. The result instance variable will be updated with the result of the operation, or the result of the Factor() method, which is the return value.

10. Factor()

This method will handle the atomic integers, as well as any symbols that are passed as operands. Furthermore, when any operator is encountered by this method, it will call the ArithmeticExpression() method to start parsing again.

11. NumericPredicate()

This method will accept two arithmetic expressions as operators, and compares them for equalities or inequalities. This will return a Boolean value.

12. LogicalOperation()

This method will handle Boolean algebra by comparing two Boolean values, which are results of the ParseLogical() helper method. This will return a Boolean value.

13. ParseLogical()

This is a helper method used by the LogicalOperation() method. This method checks the current token's type and calls the appropriate the method to handle the Boolean operation. Also, Boolean variables are also accepted, as well as atomic strings "true" and "false."

5. Semantic Analyzer

For our semantic analyzer, I used a datatable to keep track of variables and perform type checking. SIMPOL is not a complex language, and therefore does not warrant use of an Intermediate

Representation or IR. Type mismatch is the only thing our semantic analyzer needs to look out for. For example, assigning a letter to an integer variable is incorrect.

6. Error Handler

The Error class contains a ShowError() method that accepts two parameters of type string, which will be used as message text and caption respectively. It will display a preconfigured dialog box that has an error icon and an "OK" button.

3. RESULTS

In this section is where you can test the interpreter using the sample test cases. Please see Section 2, Subsection 2: "Running The Interpreter" for instructions on how to run the interpreter. Feel free to copy and paste the codes into your .SIM file.

1. Hello World!

```
variable {

}

code {

PRT $Hello World!$

}
```

Expected Result: *Hello World!*

2. Addition of Two Numbers

```
variable {

}

code {

PRT ADD 21 4

}
```

Expected Result: *25*

3. Multiplication of Two Variables

```
variable {

INT a

INT b

}

code {

PUT 5 IN a

PUT 3 IN b

PRT MUL a b

}
```

Expected Result: *15*

4. Subtraction of A Number and An Input Number

```
variable {

INT a
```

}

code {

ASK a

PRT SUB 100 a

}

Expected Result: *depends on input*

5. Nested Arithmetic Expressions

variable {

}

code {

PRT MUL 2 SUB DIV 25 5 MOD 18 ADD 3 5

}

Expected Result: *6*

6. Arithmetic Expression Inside A Numeric Predicate Inside a Logical Operation

variable {

}

code {

PRT AND GRE 10 DIV 20 2 true

}

Expected Result: *true*

7. Sir Reinald's Sample Test Case

variable {

STG str

STG name

INT num1

INT num2

INT num3

BLN bol1

BLN bol2

}

code {

PUT $The result is: $ IN str

ASK name

PUT true IN bol1

PUT false IN bol2

PUT ADD 1 2 IN num1

PUT 100 IN num2

PRT $Your name is $

PRT name

PRT OHR true AND bol1 bol2

PUT MUL 10 ADD num1 num2 IN num3

PRT num3

PRT DIV MUL 10 ADD num1 num2 MUL 10 ADD num1 num2

PRT $Goodbye!$

}

Expected Result:

Your name is

<user input>

true

1030

1

Goodbye!

4. CONCLUSION

As we can see from the Results section, my interpreter can pretty much parse any SIMPOL code, granted it follows the SIMPOL specification. Writing an interpreter for an unusual language might seem like a hard task, which it is. But when you lay out the grammar correctly, writing the RDP will be a lot less difficult. Having tested the application against a battery of complex operations. I therefore conclude this research paper by saying that a SIMPOL interpreter can be created using C#.

REFERENCES

1. http://condor.depaul.edu/ichu/csc447/notes/wk3/BNF.pdf
2. https://ruslanspivak.com/lsbasi-part1/
3. https://ruslanspivak.com/lsbasi-part7/