An Analysis of the SIMPLE Interpreter

Shounak Ghosh, December 20th, 2019

Introduction:

In this document, we outline the process undertaken to modify a given context-free grammar (the SIMPLE language), and then create an interpreter that can parse in an input, and print the resulting output to the console . The SIMPLE code is to be executed in a line-by-line fashion, by the definition of an interpreter. We begin with an introduction to the grammar, and then an explanation of the modifications necessary in order to make it suitable for parsing. We will then determine a weak equivalence between the original and modified grammar because it is impossible to rigorously prove that they generate the same language. We then move on to the challenges associated with the structure of the grammar, and the steps taken to deal with the semantics and limitations present. Finally, we have a description of the object hierarchy used to build the SIMPLE interpreter, and conclude with some sample SIMPLE test cases, along with a discussion of their expected and actual outputs.

The Grammar:

The context-free grammar shown below consists of terminals and nonterminals, which come together to define the syntax followed by the SIMPLE language. Terminals consist of the actual components of the SIMPLE language, namely integers, arithmetic operators, relational operators, and the bolded words in the grammar below. Nonterminals appear on the left-hand side of the grammar below and can be thought of as syntactic variables that are associated with a specific set or sets of terminals and nonterminals, via production rules. Multiple sets of terminals and nonterminals can be denoted using the "|" symbol, which directly translates to "or". An empty set of terminals and nonterminals is represented by epsilon, or "Ɛ".

Program → Statement P
P → Program | Ɛ
Statement → **display** Expression St1
| **assign id** = Expression
| **while** Expression **do** Program **end**
| **if** Expression **then** Program St2
St1 → **read id** | Ɛ
St2 → **end** | **else** Program **end**
Expression → Expression **relop** AddExpr
| AddExpr
AddExpr → AddExpr + MultExpr
| AddExpr - MultExpr
| MultExpr
MultExpr → MultExpr * NegExpr
| MultExpr / NegExpr
| NegExpr
NegExpr → -Value | Value
Value → **id** | **number** | **(**Expression**)**

Grammar Modifications:

Because the context-free grammar is essential in defining the structure of the SIMPLE language, we must make sure that the grammar is correctly modified so that it is suitable for a top-down recursive descent parser. We can do so by making sure that the grammar had been left factored and is right recursive, which are the two requirements for our LL(1) parser. We will begin by removing all instances of left recursion in the grammar, which occurs within the Expression hierarchy. Using left recursion removal techniques, we have the modified grammar below, on the left.

Program → Statement P
P → Program | Ɛ
Statement → **display** Expression St1
| **assign id** = Expression
| **while** Expression **do** Program **end**
| **if** Expression **then** Program St2
St1 → **read id** | Ɛ
St2 → **end** | **else** Program **end**
Expression → AddExpr Expression'
Expression' → **relop** AddExpr Expression' | Ɛ
AddExpr → MultExpr AddExpr'
AddExpr' → + MultExpr AddExpr'
| - MultExpr AddExpr' | Ɛ
MultExpr → NegExpr MultExpr'
MultExpr' → * NegExpr MultExpr'
| / NegExpr MultExpr' | Ɛ
NegExpr → -Value | Value
Value → **id** | **number** | **(**Expression**)**

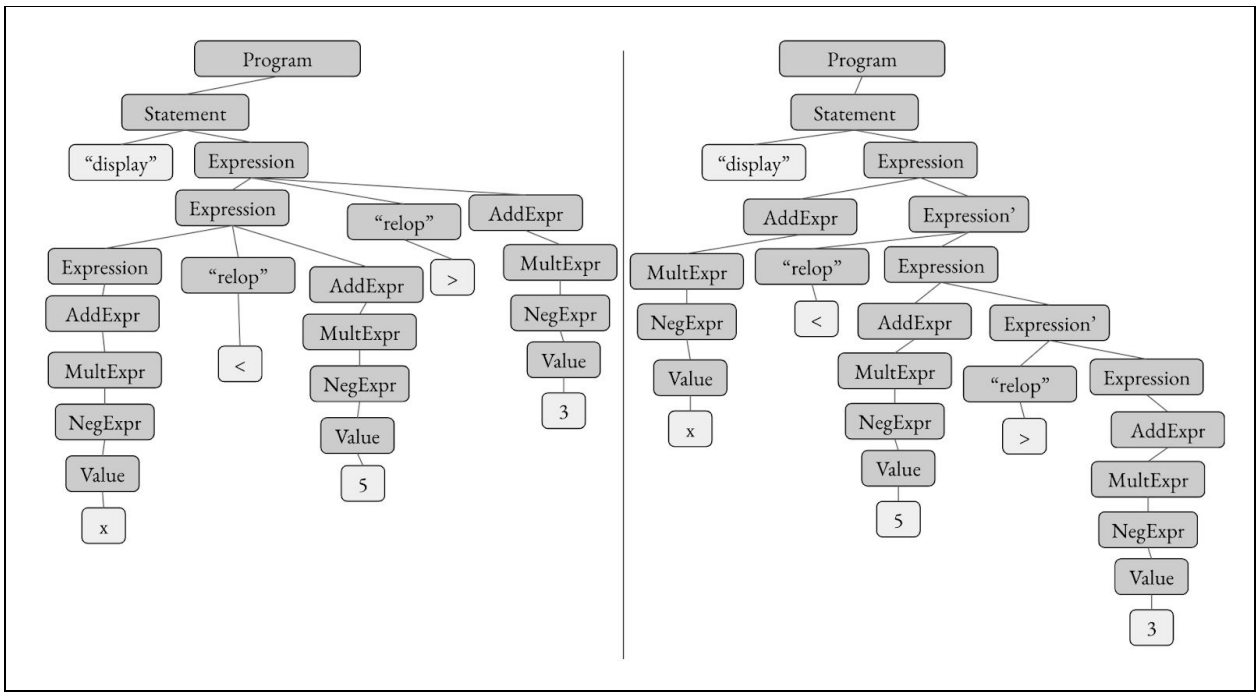
Program → Statement P
P → Program | Ɛ
Statement → **display** Expression St1
| **assign id** = Expression
| **while** Expression **do** Program **end**
| **if** Expression **then** Program St2
St1 → **read id** | Ɛ
St2 → **end** | **else** Program **end**
Expression → AddExpr Expression'
Expression' → **relop** Expression | Ɛ
AddExpr → MultExpr AddExpr'
AddExpr' → + AddExpr | - AddExpr | Ɛ
MultExpr → NegExpr MultExpr'
MultExpr' → * MultExpr| / MultExpr | Ɛ
NegExpr → -Value | Value
Value → **id** | **number** | **(**Expression**)**

We can further simplify this grammar by substituting nonterminal productions already defined leading to the top-down recursive descent parser compatible grammar shown above, to the right.

Proof of Equivalence:

We would like to show that our modifications to the grammar to make it suitable for parsing is still equivalent to our original language, meaning that both grammars generate the same exact language. However, it is impossible to rigorously show that the grammars are equivalent to one another, for we would have to test every single possible input and check that the grammars either both can or cannot create the input. Since the input pool is infinite, it is unfeasible to completely show that the grammars are equivalent, but we can show that they both can generate a sample input, which will create a weak equivalence between the grammars. The modifications we made only affected the Expression hierarchy, it is unnecessary to test the acceptance of all statement types. Thus, we take the SIMPLE

code fragment “display x < 5 > 3”, which gives us the two parse trees below, with the original grammar’s tree on the left, and the modified grammar tree on the right.

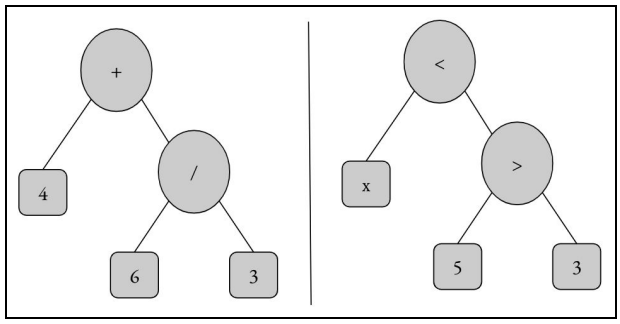


As shown above, both grammars were able to produce the sample code, as expected, hence creating a weak equivalence between the grammars. More code fragments can be used to more definitively prove that the grammars generate the same language, but the results will be similar to the case shown above. However, it is important to note that the same parse tree is not generated by the original and modified grammar, as the original grammar had a left-associative structure, where the modified grammar is right-associative. This would not affect the evaluation of expressions such as “4 + 6/3” but will play a key role in defining how mathematically invalid expressions such as “x < 5 > 3” will be handled.

Semantics and Limitations:

The main complication that occurs within the context-free grammar is the combination of relational and arithmetic operators within the Expression hierarchy, which leads to mathematically invalid statements, such as chains of relational operators and the evaluation of numerical values to true or false. We can tackle these challenges by creating two distinct types: Numericals and Booleans, which will both extend the Value. Numerical and Boolean objects are essentially wrapper classes for integers and booleans respectively, and they both have the methods getNumericalValue and getBooleanValue. These methods convert the objects to the primitive integer and boolean types, with zero being evaluated to false, and all other numbers being evaluated to true. This deals with the question of evaluating integers to true or false, and we can extend this conversion between booleans and integers to evaluate chains of relational operators. The expression tree for “4 + 6/3 = 6” is shown on the left below, ane is right associative so 6/3 = 2 is evaluated first, and then 4 and 2 are added for a final sum of

6. Likewise, the expression tree constructed for " x < 5 > 3" has the same structure, and is evaluated in the same fashion. "5 > 3" is evaluated to true, which is then converted to 1 by the getNumericalValue method, so the final expression tree will be evaluated to true if x is less than 1, or false otherwise.



Object Descriptions:

The interpreter is divided up into four main components, namely the AST, Environment, Parser, and Scanner packages. The AST (abstract syntax tree) consists of a hierarchy of classes that each individually handles specific statement executions or expression evaluation, and come together as a whole to evaluate the input SIMPLE code. The Environment package has one class (the Environment class) that stores all variable declarations and provides the classes in the AST with variable values as needed. The Parser package holds the Parser class, which generates the abstract syntax tree, and the Tester class which serves as a driver used to read in the input SIMPLE text file and print the output to the console. The final package, the Scanner, contains the Scanner class which performs a character-by-character lexical analysis of the input code in order to generate tokens, which are then processed by the parser.

AST Package

- AddExpr: Evaluates an addition or subtraction operation given an operand, MultExpr, AddExpr and outputs a Numeral object.
- Assign: Extends Statement, declares or reassigns a variable to a Value within the Environment.
- Boolean: Wrapper class for booleans, extends Value.
- Display: Extends Statement, evaluates an Expression and prints the output onto the console. Reads input if the Display statement is followed by a Read statement.
- Expression: Extends Value, uses comparison operators on two given Expressions, outputs a Boolean object.
- If: Extends Statement, executes a Program if a given Expression evaluates to true, executes a separate Program if the initial condition is false.
- MultExpr: Evaluates a multiplication or division operation given an operand, NegExpr, MultExpr and outputs a Numeral object.
- NegExpr: Negates a given Value if it is preceded by a "-", returns the Value as is otherwise.

- Numeral: Wrapper class for integers, extends Value.
- Program: Executes a statement, and then a child program if one exists.
- Read: Extends Statement, reads in a Numeral or a Boolean and stores it in a variable.
- Statement: Abstract; subclasses must override an “exec” method.
- Value: Abstract; subclasses must override an “eval” method.
- Variable: Extends Value, places a new String-Value pair within the Environment.
- While: Extends Statement, executes a given program while a given expression evaluates to true.

Environment Package

- Environment: Stores the state of the variables declared.

Parser Package

- Parser: Creates the abstract syntax tree by using a recursive descent model.
    - parseProgram: Calls parseStatement, and recurses if end-of-file has not been reached.
    - parseChildProgram: Calls parseStatement, recurses if the next token is the beginning of a Statement.
    - parseStatement: Reads in the different types of statements and outputs AST objects accordingly.
    - parseExpression: Calls parseAddExpr, reads in a relational operator, and recurses. Returns a new Expression object.
    - parseAddExpr: Calls parseMultExpr, reads in either an addition or subtraction operator, and recurses. Returns a new AddExpr object.
    - parseMultExpr: Calls parseNegExpr, reads in either a multiplication or division operator, and recurses. Returns a new MultExpr object.
    - parseNegExpr: Checks for a “-” character. Calls parseValue and returns an NegExpr object.
    - parseValue: Reads in either an Expression, a Numeral or a Boolean. Returns a relevant Value object.
    - parseVariable: Reads in the currentToken from the Scanner, returns a Variable object.
    - parseNumeral: Converts the currentToken to an Integer, returns a Numeral Object.
- Tester: Driver program that takes in the SIMPLE code, and prints the output to the console.

Scanner Package

- Scanner: Uses character-by-character lexical analysis to group characters into tokens, while also determining whether they are digits or identifiers.

Test Code:

SimpleTest1.txt

```
display 3
assign x = 1
display x read x
while x < 10 do
   display x
   assign x = x+1
   display x < 5 > 3
end
if x = 9 then
    display x
    assign x = 25
    display x
else if x = 10 then
    display x
    assign x = 35
    display x
else
    display x
    assign x = 45
    display x
end
end
display x + 4
end
```

SimpleTest2.txt

```
display 4/3 read limit
assign x = 1
assign count = 0
while count < limit do
        display x
        assign count = count+1
        assign x = x*(x+1)
end
if count = limit
   then
        display count=limit
if (x < 3)
   then
        display -600
else
        display (x+1)*3
end
end
if (limit+5)
   then
        display limit
end
```

Test Results:

SimpleTest1.txt

```
3
1
4
4
false
5
false
6
false
7
false
8
false
9
false
10
35
39
```

SimpleTest2.txt

```
1
4
1
2
6
42
true
5421
4
```

Test Case Conclusions:

SimpleTest1.txt

We can work our way through the SIMPLE code using a line by line analysis. The program begins by outputting 3 to the console and assigning 1 to the variable “x”. The value of “x” is then displayed, and the user is asked for an input (4 in this case), which is then stored in “x”. Next, a loop body, which

displays “x”, increments its value, and displays the output of the relational operator chain “x < 5 > 3” is executed while “x” remains less than 10. Upon exiting the while loop, the code branches into an if-else if-else structure. Since the value of “x” is 10, the code displays “x”, assigns the value 35 to the variable, displays “x” once again, and then displays “x + 4” after exiting the if statement. The test results match, and “false” is printed each time because “x” is always greater than 1, which “5>3” is evaluated too, as discussed in a previous section above.

SimpleTest2.txt

Once again we can assume an interpretive mindset and use a line by line analysis to determine the output of the SIMPLE code. First, 4/3 is evaluated to 1, which is outputted to the console, and the user is prompted for an input, which is stored into the variable “limit” (once again 4 in our case). Variables “x” and “count” are declared and assigned values of 1 and 0, respectively. A loop body that displays “x”, increments count, and assigns the product of “x” and “ x + 1” to itself executes while “count” is less than “limit”. The program then branches into a series of if statements. The first outputs “count=limit” if “count=limit” is equal to true. The else body of the next statement is evaluated because “ x < 3” is false, so the value of “(x + 1) * 3” is displayed on the screen. Finally “limit +5” evaluates to true so the value of limit is printed to the console, and the program ends. The test results agree, and any nonzero integer value is evaluated to false, as mentioned above.

General Conclusion

The test cases above examine all the statement types supported by the grammar, along with combinations of these statements to create program flow execution and the evaluation of mathematically invalid expressions, so further test cases would be superfluous. We can be sure that our final product is an interpreter based on the structure of the Program class, because a single Statement is first executed, and then a child Program, if one exists, is also executed. Thus, the SIMPLE code is executed in a line-by-line fashion, so we have indeed created an interpreter for the SIMPLE language.

References

- https://www.cs.utexas.edu/users/novak/cs375vocab.html
- https://link.springer.com/chapter/10.1007%2F978-3-642-28830-2_18
- https://www.cs.wcupa.edu/rkline/fcs/parse-trees.html
- https://www.cs.rochester.edu/~nelson/courses/csc_173/grammars/cfg.html
- https://youtu.be/3_VCoBfrt9c