Sahil Jain

Ms. Datar

Honors Advanced Topics Computer Science: Compilers and Interpreters

30 May 2018

SIMPLE Compiler/Interpreter Research Paper

Purpose

This document is intended to describe the complete design, functionality, and testing of the interpreter made for the SIMPLE language. The design section will lay out the overall structure of the interpreter as well as any methods or any services that each class provides. It will also detail any changes to the grammar that is laid out in the document entitled "Compilers and Interpreters Take Home Test" and justify them as necessary. The justifications will include any current semantic issues, limitations, or general problems present in the grammar. The next section will include a testing plan and strategy followed by the outputs of the provisional tests. The results section will thoroughly explain the outputs of the final tests and provide some analysis. Finally, the conclusions section will outline how the results were successful, and possible fixes and improvements for the future. Source code will be included as a zip file attachment.

Design

<u>Limitations and Semantic Issues</u>

The main semantic issue present in the current grammar is the combination of a Condition as an Expression. Even though a condition for an if statement or while loop normally returns a boolean, either true or false, the condition is now required to give a numerical value, as shown by

the grammar. However, this problem was resolved by returning a 1 for true and a 0 for false in the Condition class. All while and if statements recognize that if the value of the condition is 0, the program associated with the condition should not run, and vice versa for the case that the condition is 1.

As for limitations, having a recursive descent parser requires a really specific type of grammar. Since each step in the grammar is determined by the next token, there can not be any steps in the grammar at which there are duplicate paths upon viewing a single token. This means that left factoring is a necessary step in order to allow for accurate parsing. A change as the following is ideal:

A --> xyz | xyd should be transformed to:

A-xyA'

A' --> z | d

In the first case, the parser would not be able to identify which step to take by simply looking at an x in the token stream. n the second case, the parser is able to distinguish which step it wants to take depending on whether the current token is either a z or d since there is an extra terminal created.

In addition, all left recursion must be removed from the grammar before a recursive descent parser can work correctly. As an example, a grammar with the format A --> A+B | B should be changed to:

A--> BA'

A'--> + BA' | B

Since this transformation has left removed the recursion of A calling itself , the change is successful, making this grammar perfect for recursive descent parsing.

Grammar Transformations and Analysis

n to transform this grammar are left factoring and the removal of left recursion. Since this grammar has already been left factored, the main transformation for the grammar is to remove left recursion. The transformed grammar is below:

Program -> Statement P

P -> Program | e

Statement -> display Expression St1

| assign id = Expression

| while Expression do Program end

| if Expression then Program St2

St1 -> read id | e

St2 -> end | else Program end

Expression -> AddExpr Expression'

Expression' -> relop AddExpr Expression' | e

AddExpr -> MultExpr AddExpr'

AddExpr' -> + MultExpr AddExpr' | - MultExpr AddExpr' | e

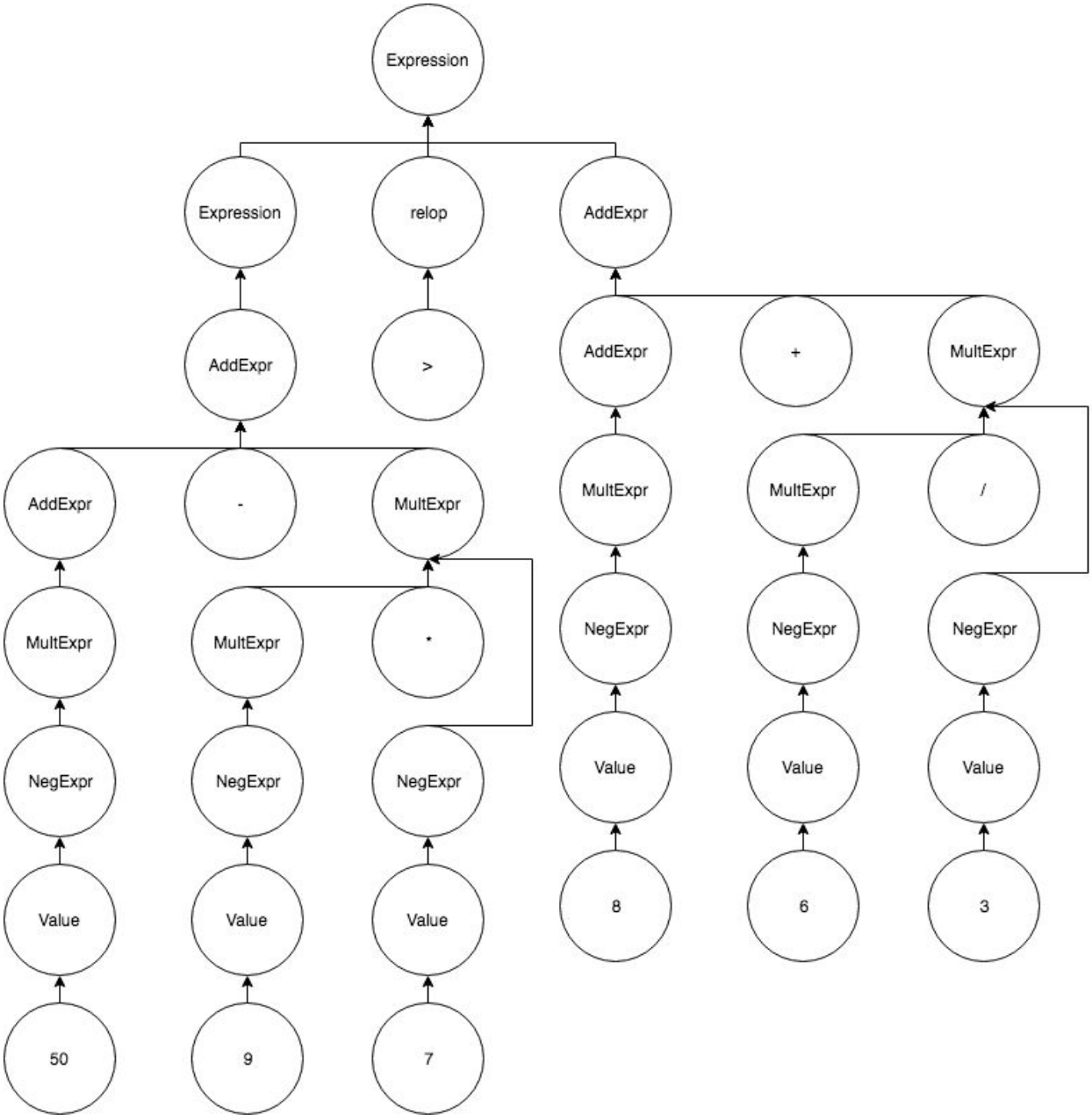
MultExpr -> NegExpr MultExpr'

MultExpr' -> * NegExpr MultExpr' | / NegExpr MultExpr' | e

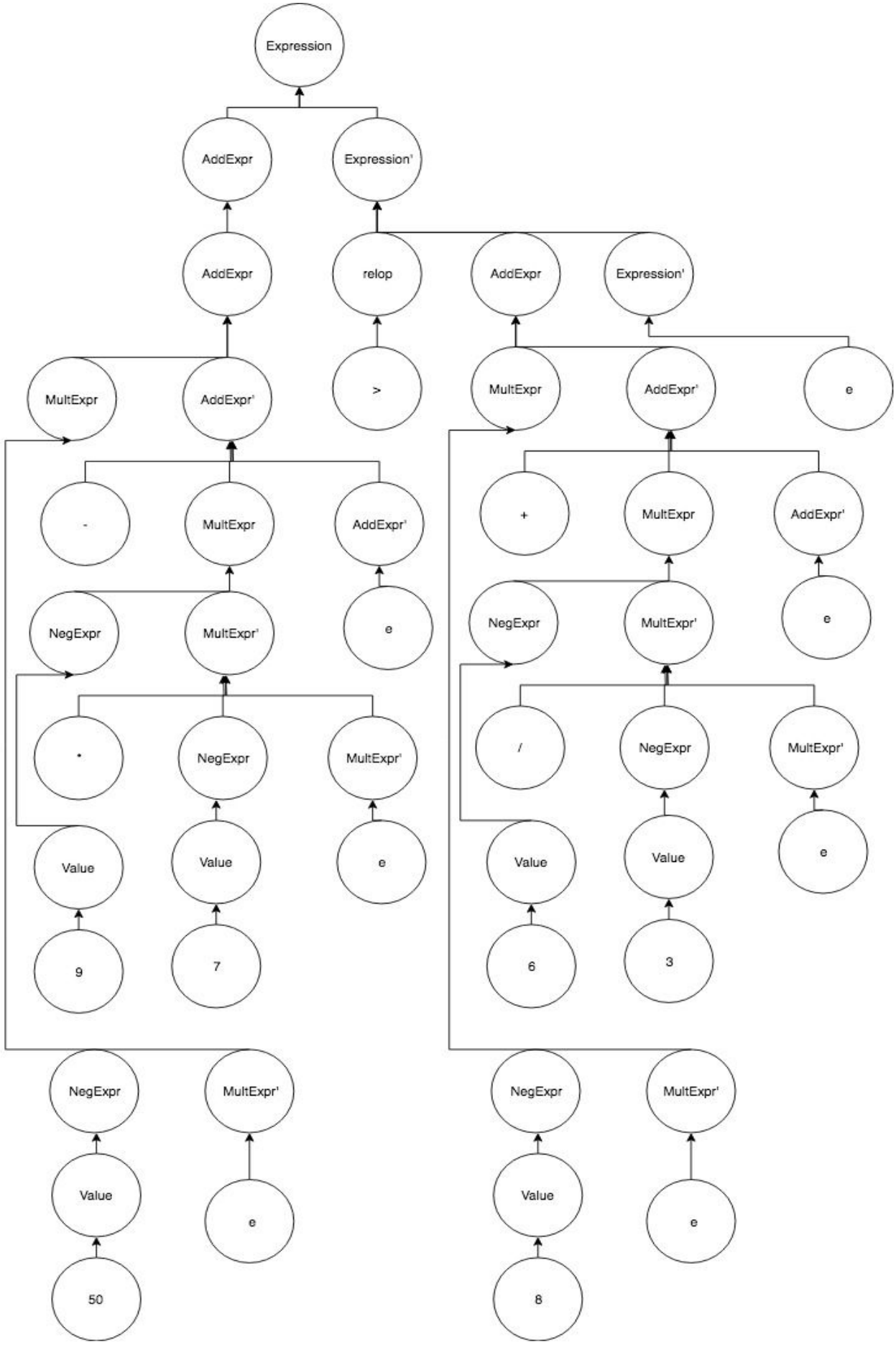
NegExpr -> -Value | Value

Value -> id | number | (Expression)

Since only the portions of Expression, AddExpr, and Expr have changed, if the parse trees are the same for expressions in both grammars, both grammars must be equal. As an example, the grammar trees of the expression 50 - 9 * 7 > 8 + 6 / 3 are going to be compared for similarity. For the first grammar, the tree is the following:



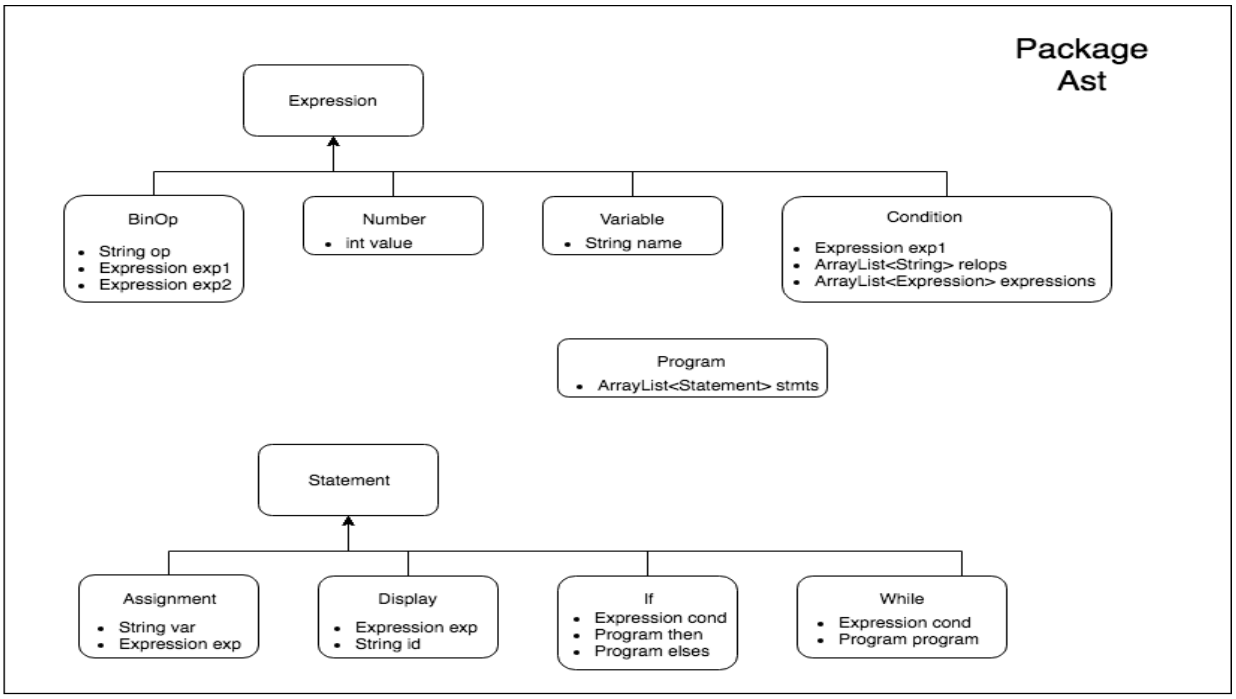
The tree for the second tree is as follows:

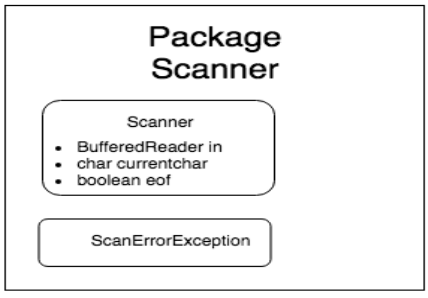

Expression
AddExpr
Expression'
AddExpr
relop
AddExpr
Expression'
MultExpr
AddExpr'
>
MultExpr
AddExpr'
e
-
MultExpr
AddExpr'
+
MultExpr
AddExpr'
NegExpr
MultExpr'
e
NegExpr
MultExpr'
e
*
NegExpr
MultExpr'
/
NegExpr
MultExpr'
Value
Value
e
Value
Value
e
9
7
6
3
NegExpr
MultExpr'
NegExpr
MultExpr'
Value
e
Value
e
50
8

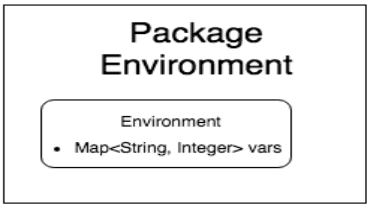
Since both of these grammars generate the same trees, they must be equal. Therefore, for the sake of recursive descent parsing, it is justified to transform the grammar in this way since there are no differences in the language that it produces.

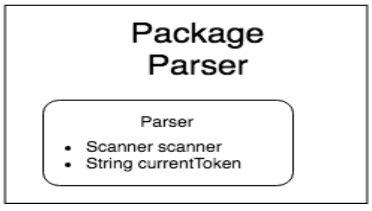
Object Structure

The structure of the various packages, classes, and objects is shown below. Full descriptions and details of the services each provides is located in the next section.









Description of Each Object

1. Parser

Summary: Parser is a class that parses a stream of tokens returned by the Scanner as described by the above grammar. The constructor takes in a scanner as input for the so that it can obtain the tokens. The class contains many methods (which will be described in more detail below) that are denoted by parseX, where x is the object that the parser is currently parsing. As an example, the parseStatement method parses a Statement as denoted by the Statement section in the grammar above and returns a Statement with all of the necessary instance variables. Please note that, while most of the parseX methods are recursive, some utilize the while loop to repeatedly parse X.

Services: The services that Parser provides are the following:

Program parseProgram() - The parseProgram method parses a program according to the grammar above. It uses a while loop to repeatedly parse a Statement while the currentToken is either "display", "assign", "while," or "if," which are all of the possible starting tokens for a statement. After each iteration of the while loop, the Statement is added to the ArrayList of Statements. At the end of the method, a Program is returned with its argument being the ArrayList of Statements.

Statement parseStatement() - The parseStatement method parses a Statement according to the grammar above. It uses multiple if statements to check what type of statement the current token denotes. (Note that if the current token does not denote any type of statement in the grammar, a ScanErrorException is thrown detailing the current token and what it should have been.) If the

current token is "display," the expression to be displayed is parsed and added as an argument for the "Display" object. Then, if the current token is "read," an additional String representing the id of the variable whose value is to be changed is parsed and added as a parameter to the "Display" object. For an if statement, an Expression, representing the condition, and a program, representing the statements that should be executed if the expression is evaluation to be true, are parsed and added as arguments for the "If" object. If an else token is encountered, an extra program is parsed and added as an argument for the object. A while statement requires both an Expression, representing the condition required to be met, and a Program. Both of these components are parsed and added as arguments for a "While" object should the first token be "while." Finally, for the statement denoted by an "assign" token, an "Assignment" object is returned with a String representing the id for the variable and an Expression representing the value of the variable.

Expression parseExpression() - parseExpression parses a condition for an If or While statement or an AddExpr in the case that no boolean operators are present, returning an Expression. At the very beginning, it parses an AddExpr. Then, for each additional boolean operator(denoted by "relop" in the grammar) in the current Expression, the operator and an Expression are parsed and added to corresponding ArrayLists. If there are no boolean operators, the AddExpr object is just returned. However, if there are boolean operators, a "Condition" object with the AddExpr and two ArrayLists passed as parameters is returned.

Expression parseAddExpr() - parseAddExpr parses an AddExpr as shown in the above grammar, handling expressions containing plus and minus signs. It first parses a MultExpr and sets it to a temporary Expression variable called num. Then, while the current token is either a plus or

minus, num is set to a new BinOp object with arguments of the corresponding operator, num, and a newly parsed MultExpr. This num object is then returned at the end of the method.

Expression parseMultExpr() - parseMultExpr parses a MultExpr as shown in the above grammar, and handles multiplication and division signs in expressions. First, a NegExpr is parsed and is set to a temporary Expression variable. While the current token is equal to either the multiplication sign or the division sign, this variable is set to a new BinOp object with the operator, the varaible, and a newly parsed NegExpr as its arguments. This variable is then returned at the end of the method.

Expression parseNegExpr() - parseNegExpr handles for minus signs in Expressions, shown by the grammar above. If the current token is a minus sign, a new BinOp object is returned with a multiplication operator, a new number with an argument of -1, and a newly parsed Value. However, it there is no minus sign, a newly parsed value is returned.

Expression parseValue() - As shown in the grammar above, there are three things that the parseValue method parses: an id for a variable, a Number, or an expression surrounded by parentheses. For the first case, a Variable object is returned with its argument as the id String. For the second case a newly parsed number is returned; finally, the latter case simply involves removing the parenthesis from the input stream and returning a newly parsed Expression.

Expression parseNumber() - The parseNumber parses a number as shown in the above grammar. It first converts the current token to an integer. Then, it returns a Number, with this integer as its parameter.

2. Assignment

Summary: The assignment class represents an assignment in the ast parser, as denoted by the "assign" statement in the grammar. It extends the Statement class, and therefore implements the exec method, which assigns the value of the variable to be the expression given. In addition, it initializes the instance variables of a String variable name and an expression as the value of the variable inside its constructor.

Services: The assignment class provides the following services:

void exec(Environment env) - The exec method executes the assignment by evaluating the expression and declaring a new variable in the environment with the name of the variable and the value.

3. BinOp

Summary: The BinOp class represents an operation carried out in the ast parser. It extends the Expression class and therefore has the eval method, which returns an integer representing the result of the operation expression. It takes in three arguments: an operator, representing the operation to be carried out, and two expressions, which represent the two values that are to be either multiplied, divided, added, or subtracted.

Services: The BinOp class provides the following services:

int eval(Environment env) - The eval method evaluates the BinOp by first evaluating each of the two expressions and then performing the corresponding operation to them, either multiplication, division, addition, or subtraction. The result of this operation is returned as an integer.

4. Condition

Summary: The Condition class represents an Expression shown in the above grammar and is used to evaluate the validity of statements for if and while loops. It extends Expression

class and therefore has the eval method, which returns 0 if the expression is false or 1 if the expression is true. The constructor for the Condition class takes in an expression, representing the value that all of the other expressions will be compared to, a list of Strings that contains a list of all boolean operators to be used, and a list of Expressions that contains all of the other expressions that will be compared to the first one.

Services: The Condition class provides the following services:

int eval(Environment env) - The eval method uses a for loop to parse through the ArrayList of boolean operators. For each boolean operator, the expression instance variable is compared to the expression in the Expression ArrayList corresponding to the index of the current boolean operator. If at any point the comparison turns out to be false, 0 is returned by the method. However, if all comparisons are evaluated to be true, 1 is returned at the end of the method.

5. Display

Summary: The Display class represents a display statement in the ast parter, as denoted by the "display" statement in the grammar. Its arguments include an expression, whose value is to be printed out, and a String identification if there is a "read" segment in the display statement. The value that the user inputs is to be stored in the variable that has the name given by identification. Since this class extends the Statement class, it implements the exec method, which prints out the value of the expression and stores the user input in the variable of the given name.

Services: The Display class provides the following services:

void exec(Environment env) - The exec method of the Display object first evaluates the Expression in the environment and prints out the value. Then, if there is a read portion of the

statement, an integer value is read. This value and the name of the variable are used to create a new variable.

6. Expression

Summary: The abstract class Expression models an Expression in the AST Parser. Note that the "Expression" in the grammar above is represented by the Condition in the ast package. This Expression class is a generic superclass for all types of expressions that can be evaluated, namely Condition, BinOp, Number, and Variable. It has the abstract method eval that is required in all classes that extend Expression and is intended to return an integer representing the value of the expression.

Services: The Expression class provides the following services:
abstract int eval(Environment env) - eval is a generic method that must be implemented in all subclasses of Expression. It returns an integer representing the value of the expression. Note that Environment is always one of the parameters as it contains a Map with a key value pair for each variable, which is useful for evaluating expressions with variables.

7. If

Summary: The If class represents an if statement in an ast parser, as denoted by the "if" statement in the grammar above. It has three arguments: an Expression, or the condition, and two Programs, one which represents the list of statements to be executed in the case that the condition is true, and one which represents the list of statements to be executed in the case that the condition is false. Note that one of the constructors also handles for the case that there is no "else" section in the statement and therefore excludes the second Program as an argument. Since

the If class extends Statement, it implements exec, which executes specific Programs depending on whether the condition is true or false.

Services: The If class provides the following services:

void exec(Environment env) - exec executes the If statement by first evaluating the condition. If it is true, it executes the first Program. Then, if there is an "else" section in the If statement, it executes the latter Program if the condition is false.

8. Number

Summary: The Number class represents a number in an ast parser, as denoted by the Number section in the above grammar. It takes in a single argument, which is an integer, and sets it to be its instance variable. Since Number extends Expression, it implements the eval method, which simply returns the integer value.

Services: The Number class provides the following services:

int eval(Environment env) - The eval method simply returns the integer value of this Number.

9. Program

Summary: The Program class represents a Program, as denoted by the Program section in the above grammar. Its argument is a list of statements, which is set to be the ArrayList instance variable. In the exec method, each statement is executed one by one.

Services: The Program class provides the following services:

void exec(Environment env) - The exec method of the Program simply executes each statement in the ArrayList one by one.

10. Statement

Summary: The abstract class Statement models a Statement in the ast Parser. It is a superclass for all types of Statements in the language, particularly the Display, If, Assignment, and the While. It contains an abstract method exec that all of the subclasses are required to implement to be able to successfully execute at run time.

Services: The Statement class provides the following services:

void exec(Environment env) - exec is a generic method that is to be implemented in all subclasses of Statement so that each type of statement can execute during run time. It takes in an Environment because the environment contains the list of variables and their values in a Map, and is therefore useful for any statement involving a variable.

11. Variable

Summary: The Variable class models a variable expression in the ast Parser for the SIMPLE language. It takes in a single argument, which is the name of variable. Since this class extends Expression, it implements the eval method, which returns the value of the variable associated with the name.

Services: The Variable class provides the following services:

int eval(Environment env) - The eval method returns the value of the variable associated with the given name, utilizing the getVariable method of the Environment class.

12. While

Summary: The While class models a while statement in the SIMPLE language, as denoted by the "while" statement in the grammar above. Its arguments include a condition, which is to be evaluated, and a program, which contains a list of statements associated with the

while loop. Since While extends Statement, it implements the exec method, which executes the Program while the condition is true.

Services: The While class provides the following services:

void exec(Environment env) - The exec method uses a while loop to continuously execute the Program while the condition is true. The condition is considered to be true when the eval method of the Condition returns 1.

13. Environment

Summary: The Environment class is most useful to store and access variables in the SIMPLE language, and is therefore used by every Statement or Expression that contains a variable or assignment. It contains a Map which has a key of a String and a value of an integer; this class contains a complete list of the variables in the program. It contains two main methods: getVariable, which returns the integer value of the variable associated with the given name, and declareVariable, which adds a new variable with the given name and value.

Services: The Environment class provides the following services:

int getVariable(String variable) - The getVariable method returns the integer value of the variable in the Map associated with the given name.

void setVariable(String variable, int num) - Adds a new variable entry to the Map with the key being the given String name and the value being the given integer.

14. Scanner

Summary: The Scanner class is used to output a stream of tokens for SIMPLE code so that it can be parsed and executed. There are three instance variables in this class: a BufferedReader in order to read the file, a char representing the current character, and a boolean

variable called eof which helps to determine whether the Scanner has reached the end of the file. This class takes in a String representing the location of the file where the code must be read. In the constructor, the Scanner first appends a "." to the file, as a "." is the token signals the Scanner that the file has ended. Then, all of the instance variables are initialized and the first character is scanned.

The Scanner parses three main types of tokens: identifiers, which are defined as a letter followed by any combination of letters and digits, operators, and numbers, which are defined as a string of digits. Methods denoted by scanX scan the different types of tokens, while methods denoted by isX are used to determine the type of the token based on the current char. All of these methods are utilized in the nextToken() method, which scans the next token by first determining the type of token and then calling the respective scanX method. Note that all white spaces, defined as spaces, tabs, new lines, and returns, are ignored.

Services: The Scanner class provides the following services:

boolean hasNext() - Using the instance variable eof, hasNext returns true if the end of file has not been reached, or false otherwise. It is used in the nextToken() method to make sure it is possible to scan another token.

boolean isOperator() - isOperator returns true if the current character is an operator.

boolean isDigit() - isDigit returns true if the current character represents the start of a number, which is any digit from 0 to 9.

boolean isLetter() - isLetter returns true if the current character represents the start of an identifier, which is any lowercase or uppercase letter.

boolean isWhiteSpace() - isWhiteSpace returns true if the current character represents blank of white space, which is either a space, tab, new line, or return.

String nextToken() - nextToken scans the next token by first determining the type of the current token using the current character and the above boolean methods. Depending on the the type of token, the corresponding scanX method is called to scan that type of token.

15. ScanErrorException

Summary: The ScanErrorException is used to through error messages whenever there is an error while parsing or scanning. It most cases, it is used when there are illegal characters or when the expected character or token does not match the actual character/token. It has two constructors, one which creates a new Exception with no arguments, and another that creates a new Exception with a given error message String. This class does not have any services.

Test Plan and Strategy

Testing for the SIMPLE interpreter is carried out in steps. First, after the completion of the Scanner class code,

a main method is created in the Scanner class with the name of a file with SIMPLE code as its parameter. This main method contains instructions to print out all of the tokens in the file while there is a next token. The result on the console is then compared to the expected stream of tokens.

Next, each type of expression and statement is tested as their respective classes are written. For example, in order to test for the Display statement, a sample statement such as "display 3 read x"

is written in the input file. Appropriate corrections to code are made after comparison of the console output to the expected output.

Finally, the two given test files are set as input for the Parser, and the output is recorded and checked off by the instructor.

Test Output

The test for the Scanner as described above was carried out on the first simpleTest file, and outputted the following:

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 END

This test was considered to be successful since all of the tokens were present without issue. In addition, all white space was removed since there are no newline or space characters. Finally, the END token was printed out and recognized correctly, meaning the "." was appended to the file.

Below is one of the tests carried out after the implementation of Program, Variable, Assignment, Display, and Number:

display 3

assign x = 1

display x read x

```
display x
```

After 5 was entered in when prompted for input, the following output was printed:

```
3
1
5
```

The expected output is 3 (for the first statement), 1 (for display x, since x is assigned to be 1), and 5, since 5 is assigned to x via user input. Since the expected output matches the actual output, these classes seemed to be in working order.

Another test tried out after the one above was for if statements, and it was carried out after If and Condition were implemented:

```
assign x = 1
if x > 3 then
display x + 2
else
display x + 1
end
end
```

The output was 2. The expected output would also be two since x less than three and 1+1 is displayed. Therefore, the If statement also worked accurately.

Finally, to test for the while loop, a simple test was conducted which was to print out the numbers 1 to 10:

```
assign x = 1
while x<11 do
display x
assign x = x+1
end
```

The output was indeed the numbers 1 to 10, meaning that the while statement, along with the other expressions and statements, worked. Therefore, testing on the two simpleTest files was started.

Results

The output of the first test file (when 2 is inputted for x) is printed below:

3 1 2 0 3 1 4 0 5 0 6 0 7 0 8 0 9 0 10 35 39

The expected output for this test file should be the numbers 3 and 1 followed by the numbers 2 through 9, each with a 0 following it to represent the falsity of the expression "x<5>3." However, after the number 3, the following number should be a 1 since x is four at the point when that

expression is evaluated. Finally, 10, 35, and 39 should be printed out since x is equal to 10 at the end of the while loop.

Since the expected output matches the console output, the test for file 1 was successful. Other tests for varying x values were also successful.

The output for test file 2 (when 3 is inputted for limit) is printed below:

1 1 2 6 1 129

The expected output for this test file should first be 1 (for 4/3). Since count is initially 0 and the limit is 3, the while loop is run three times, with x being set to x*(x+1) each time, the next three numbers should be 1, 2, and 6. The end of the while loop should lead to a value of three for count and 42 for x. Then, since count is equal to limit, 1 should be outputted. Since x is not less than 3, the next if statements should lead to a print out of (x+1)*3, or 129. Finally, since the expression limit +5 should not evaluate to true, nothing is outputted for the last if statement.

Since the expected output matches the console output, the test for file 2 was successful. Note that tests for varying limit values were conducted, also with expected outputs.

Conclusion

Since the interpreter was able to produce accurate results for numerous test cases and files, the design and building of this interpreter was rather successful. However, as with any compiler or interpreter, there is always room for improvement in the future. For future implementations, it would be ideal to print out the Strings "true" or "false" to the console for expressions with

boolean operators to allow for better readability for the user. In addition, more test cases and files should be utilized to further test this interpreter to find out potential bugs and improvements for it. Overall, even though the SIMPLE language is only able to perform minimal tasks and calculations, its implementation allowed for the enriching experience of making a compiler from scratch given a simple grammar and a few instructions.