Jackelyn Shen

11/21/14

P.7

SIMPLE Interpreter Design Document

**Introduction**

This document will explain the design of my SIMPLE Interpreter outlined in the Compilers and Interpreters Take Home Test. In this document, I will describe the provided grammar, its possible problems and limitations, and any alterations I made to make it suitable for recursive descent parsing. I will outline each object I created for the interpreter, its purpose, and its relationship to other objects included in the interpreter. I will also detail my various testing strategies and the resulting printout for each test. A copy of the entire source code and test code is included. Finally, I will outline my conclusions and thoughts about the grammar, tests, comparisons to the PascalParser lab, and the overall interpreter design I have written, adding comments for possible future improvements.

**Design**

Grammar

Here is the original grammar given on the test, already left factored.

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 -> Expression **relop** AddExpr | AddExpr

AddExpr -> AddExpr + MultExpr | AddExpr – MultExpr | MultExpr

MultExpr -> MultExpr \* NegExpr | MultExpr / NegExpr | NegExpr

NegExpr -> -Value | Value

Value -> **id** | **number** | (Expression)

This grammar seems fine except for Expression, AddExpr and MultExpr. The non-terminals Expression, AddExpr and MultExpr are currently ambiguous, and attempting to parse these would result in an infinite loop. So I introduced a whileTerm to remove the ambiguity and replaced the first component with MultExpr. The new grammar retains the logic of the original grammar but also makes the grammar suitable for recursive descent.

For example, I changed AddExpr to:

AddExpr -> MultExpr whileAdd

whileAdd -> + MultExpr whileAdd | - MultExpr whileAdd | e

The resulting output would still be a series of MultExprs separated by either a – or a +, or a single MultExpr.

Using the same logic, the MultExpr grammar would be:

MultExpr -> NegExpr whileMult

whileMult -> \* NegExpr whileMult | / NegExpr whileMult | e

The resulting output would still be a series of NegExprs separated by either a \* or a /, or a single NegExpr.

I also added a whileTerm to the Expression grammar below.

Expression -> AddExpr whileExp

whileExp -> relop AddExpr whileExp | e

Thus, the logic is the same: and Expression is defined by a series of AddExprs separated by a relop, or a single AddExpr.

My final altered grammar is listed 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 whileExp

whileExp -> **relop** AddExpr whileExp | e

AddExpr -> MultExpr whileAdd

whileAdd -> + MultExpr whileAdd | - MultExpr whileAdd | e

MultExpr -> NegExpr whileMult

whileMult -> \* NegExpr whileMult | / NegExpr whileMult | e

NegExpr -> -Value | Value

Value -> **id** | **number** | (Expression)

However, there is a slight problem with the grammar that involves the definition of an Expression. This grammar does not differentiate between Expressions with relop and Expressions without relop, which causes problems when an Expression is used in a conditional Statement such as while or if. For example, we would only except Expressions containing relops to be located in between the tokens “while” and “do” but following this grammar, the line “while 3 do [Program] end” would semantically make sense to this interpreter but would not make sense logically. Additionally, only an Expression without a relop would make sense in the display and assign grammars, but “display 1<2” and “assign x = 1<2” would still conform to this grammar. Also, the Expression grammar creates the complicated problem of having multiple consecutive relop Expressions, such as “1<2<3,” which cannot be handled.

Objects

The objects that I created for the SIMPLE Interpreter were extremely similar to the ones I used for PascalParser.

Here is a basic diagram detailing each package and their classes:

**package ast**

Number

Identifier

Expression BinOp

ExpOp

Display

Statement Assign

If

While

**package environment**

Environment

**package interpreter**

Interpreter

InterpreterTester

I kept the basic abstract objects Expression and Statement, which provides a superclass for more defined objects and initializes the eval (returns an Integer for Expressions) and exec (returns void for Statements) methods for evaluating or executing each component of the abstract syntax tree.

The Number, Identifier, BinOp, and ExpOp classes are all subclasses of Expression.

A Number object stores and takes in an Integer value, and overrides the Expression’s eval method to return that integer.

An Identifier object stores and takes in a String value, and overrides the eval method to return the Integer value linked to that String located in the Environment, which contains the symbol table.

A BinOp object organizes and evaluates operations by storing 3 instance variables: a lefthand Expression, a righthand Expression, and a String operator. The eval method for BinOp returns an integer representing the operation between the two evaluated expressions (operators include +, -, \*, and /).

An ExpOp object organizes and evaluates condition expressions and stores 3 instance variables: a lefthand Expression, a righthand Expression, and a String relop. In its eval method, the ExpOp object evaluates whether or not the condition is valid based on the String relop between the two evaluated Expressions (relops include =, <>, <, >, <=, and >=). The eval method needs to return an integer, so if the condition is valid, 1 is returned; otherwise 0 is returned.

The Display, Assign, If, While and Program classes are all subclasses of Statement.

A Display object stores and executes SIMPLE display statements. It has 2 instance variables: an Expression to be printed out and a Statement representing the identifier to be set. The Display exec method overrides that of the Statement object by printing out the display Expression, and if read is valid, executing the read assignment.

An Assign object organizes and executes SIMPLE assignments according to the grammar, setting 2 instance variables: a String identifier and Expression to be assigned to the variable. In the exec method, the Assign object adds the identifier and its Expression to the symbol table stored in the Environment.

An If object contains and executes SIMPLE if statements. It contains 3 instance variables representing the if Expression, then Statement, and else Statement. In its exec method, an If object checks to see if the if Expression is valid, and if it is, it executes the then Statement; otherwise, the If object executes the else Statement if it is valid.

A While object contains and executes SIMPLE while statements. It contains 2 instance variables representing the while Expression and the do Statement. In its exec method, a While object checks to see if the while Expression is valid, and it it is, it executes the do Statement and loops back to checking the while Expression again, repeating the process.

A Program object organizes SIMPLE programs and statements, becoming the root of the abstract syntax tree outputted by the interpreter. It takes in two Statement instance variables, one of which is the actual program statement and the other as another Program. In the exec method, a Program object executes the program statement and the other Program if it is valid.

I also recreated the Environment class to store the symbol table, which is represented by a HashMap that stores and links String variables to their Integer values. I used the exact same methods as the Environment in the PascalParser lab to get and set variables.

I created an Interpreter object to hold all the Interpreter code, and it parses SIMPLE programs that conform to the altered grammar detailed above. I used the same recursive descent approach as the one used in the PascalParser lab to create methods for the SIMPLE Interpreter.

I started off with the parseProgram method, which returns a Statement. It first parses a Statement and then checks to see if the currentToken is not either end, else, or an empty String. For currentToken, an “end” token would represent the end of the do Program in the While Statement and the then Program in the If Statement. An “else” token represents the end of the else Program in the If Statement, and the empty string represents the end of the file. If the current token is not any of these, the Interpreter returns a new Program containing the parsed Statement and parses another Program to insert as the second parameter. Otherwise, the Interpreter returns a new Program with the parsed Statement as the first parameter and null for the Program parameter. As the grammar allows for a Program to be parsed after the program Statement, it was necessary to attempt to parse the next program within the current program, and thus I created a Program design with a Statement and another Program as instance variables.

In the parseStatement method, the Interpreter checks to see if the currentToken is either display, assign, if, or while, according to the grammar. If it is “display,” the Interpreter then parses an Expression and checks to see if read is the next token. If so, the Interpreter stores the id token and reads in the user input, creating a new Assign object. The Interpreter then returns a new Display object containing the parsed Expression and the read Statement. If the current token is “assign,” the Interpreter stores the next id token and parses an Expression, returning a new Assign object. If the token is “if,” the Interpreter consecutively parses an if Expression, then Statement, and else Statement, returning a new If object. If the token is “while,” the Interpreter parses a while Expression and then a do Statement, returning a new While object. If the current token is neither of these, null is returned.

In parseExp, by following the grammar, the Interpreter first parses an AddExpr “exp,” and while the next token is a relop, it sets exp as a new ExpOp object with exp, the relop, and another parsed AddExpr as parameters. Returns exp.

In parseAddExpr, the Interpreter first parses a MultExpr “addExpr,” and while the next token is either a + or -, it sets addExpr as a new BinOp object with addExpr, the operator, and another parsed MultExpr as parameters. Returns addExpr.

In parseMultExpr, the Interpreter first parses a NegExpr “multExpr,” and while the next token is either a \* or /, it sets addExpr as new BinOp object with multExpr, the operator, and another parsed NegExpr as parameters. Returns multExpr.

In parseNegExpr, if the current token is -, the Interpreter returns a new BinOp object with 0, -, and a parsed Value as parameters; otherwise, returns a parsed Value.

In parseValue, if the current token is (, the Interpreter returns another parsed Expression inside of the parenthesis. If the token is an identifier, it returns a new Identifier object with the token identifier as a parameter. Otherwise, the Interpreter returns a parsed Number.

In parseNumber, the Interpreter converts the String token to an integer and sets it as a parameter for a new returned Number.

Similar to the PascalParser lab, the method eat uses the Scanner to take in a String representing the current token.

Finally, I created an InterpreterTester object to test my SIMPLE Interpreter. The design for this tester class is extremely similar to ParserTester used in the PascalParser lab. The InterpreterTester encapsulates a single main class, which creates a Scanner object and inserts it into the parameter of a new Interpreter object. The tester creates a new Environment object and uses it to execute the program outputted by a call to the Interpreter’s method parseProgram. Thus, each component of the abstract syntax tree created by the Interpreter is executed.

Test Strategy

1. Consecutive programs can be parsed until the end of the file.
2. A display statement (without read) will display the following expression.
3. A display statement (with read) will display the following expression, store the token after read as an identifier, and ask for user input to set the variable.
4. An assign statement will set the identifier to the following Expression.
5. In a while statement, until the Expression becomes false, the do Program will execute.
6. A do Program is all programs between the “do” token and the “end” token.
7. In an if statement, if the Expression is valid, the then Program will execute. If not, the statement will end or the else statement will execute.
8. A then Program is all programs between the “then” token and “end”/”else” token, and an else Program is all programs between the “else” and the “end” tokens.
9. An Expression can be a single AddExpr or multiple AddExpr connected by relops.
10. AddExpr, MultExpr, NegExpr, and Value are executed correctly and in the proper order.
11. The Environment gets and sets Strings and Integers correctly.

**Test Plan**

Test 1

Test to see if display and read statements work correctly.

See if consecutive programs are all parsed.

Test 2

Test to see if assign statements work.

Test to see if negative numbers are stored correctly.

Test 3

Test to see if various relop conditions/ExpOp expressions are handled correctly.

Test to see whether if statements work correctly, with and without else statements.

Test to see if the then program includes all programs in between tokens “then” and “else”/“end”.

Test to see whether BinOp expressions calculate correctly.

Test to see whether (Expression) is handled.

Test 4:

Test to see whether while statements are executed correctly.

Test to see if the do program includes all programs in between tokens “do” and “end”.

Test to see if order of operations are done correctly.

Test 5:

Tests to see if an if statement works inside of a while statement.

Tests to see if a while statement works inside of an if statement.

**Source Code**

Attached is the source code for all of the objects I created for the SIMPLE interpreter.

**Test Code**

Test 1:

display 1 read x //set user input to 2

display x

Test 2:

assign x = -1

display x

Test 3:

if 1>3 then display 1-3 else display 3-1 end

if 3>=1 then display 2\*(3\*-3+5) display 3/1 else display 3 end

if 1=3 then display 1 end

Test 4:

assign x = 1

while x<>3 do display x-4\*2 assign x = x+1 end

if x=3 then display x\*4+2 end

Test 5:

assign x = 0

while x<=5 do if x<3 then display x else display -99 end assign x=x+1 end

if x=6 then while x<10 do display x assign x=x+2 end else display 0 end

**Results**

Test 1:

1

2

Test 2:

-1

Test 3:

2

-8

3

Test 4:

-7

-6

14

Test 5:

0

1

2

-99

-99

-99

6

8

**Conclusions**

The Interpreter correctly executed all five tests catered to my testing strategy and plan. As we are not interested in correcting the various semantic difficulties of this specific grammar, I explicitly did not test SIMPLE code that would not make sense logically, mentioned in the main body of this document.

The SIMPLE Interpreter was extremely similar to the PascalParser lab. Much of the grammar was quite similar to that of PascalParser, so my Interpreter’s recursive descent methods paralleled that of the previous lab. Additionally, many of the abstract syntax tree classes are exactly the same, such as Expression, Statement, Assign, BinOp, and Number. The Environment class was also used in the same way. An addition of the class ExpOp helped differentiate the relop Expressions with the non relop Expressions. However, this grammar had no Condition nonterminal, which allowed for illogical use of an Expression in some Statements. A helpful addition in the SIMPLE Interpreter that did not appear in the PascalParser lab was that multiple consecutive programs could be parsed inside of the grammar, whereas the PascalParser could only parse programs until the end of the file in its tester. I believe that an addition of an end of file marker, such as the period for the PascalParser lab, would help the Interpreter determine the end of a program. I found the grammar for the display statement quite interesting, as it allowed for an identifier to be set by user input. To make the Interpreter more user-friendly, I printed out the line “Set [id] to: ” to the parseStatement method for parsing display statements, to indicate when the user should input a number. However, I think the grammar could be improved by separating the display statement and the read statement into two different Statements, as the display statement prints out a separate Expression than the read user input. Overall, the SIMPLE Interpreter, although it had its limitations, effectively encouraged me to alter the grammar so that it could be properly parsed, decide on appropriate abstract syntax tree objects for the new grammar, create and connect recursive descent methods, and figure out testing strategies for the Interpreter to pass.