Nathan Dalal
Mr. Page
Compilers and Interpreters
15 April 2015

Designing a SIMPLE Interpreter

Introduction

The SIMPLE language offers a simple yet comprehensive introduction to recursive descent parsing and the formation of abstract syntax trees. The goal of this exercise is to design and implement a recursive descent parser that scans and parses the SIMPLE grammar into an abstract syntax tree.

In our interpreter, a scanner will accept tokens according to a regular language. A recursive descent parser will parse the tokens and assemble the abstract syntax tree. Once the assembly of a parser is completed, testing and results will be shown. The parser will parse SIMPLE code and interpret the results, and answers will reflect the grammar used in design.

Implementation and testing will be done in Java, and code design of each portion of our interpreter will reflect the use of Java.

The SIMPLE Regular Language

The SIMPLE language scanner accepts tokens according to the following regular language:

keyword -> [“display”, “read”, “assign”, “while”, “if”, “then”, “else”, “do”, “end”]
id -> letter (letter | digit)*
number -> digit digit*
relop -> {<, >, >=, <=, <>, =}
binop -> {+, -, *, /}

letter -> [A...Z a...z]
digit -> [0...9]
whitespace -> [' ' \t' '\r' '\n']
parentheses -> [‘(‘ ‘)’]

The bolded regular definitions in the regular language of SIMPLE are the token types returned by our scanner. The letter and digit definitions are used in our token definitions, while lexemes that match the whitespace definition is ignored and skipped in our scanner. Our scanner will give precedence to the token that appears first in the SIMPLE regular language; for example, if a lexeme matches the keyword and id regular definitions, the returned token will be of a keyword token type.

The SIMPLE Grammar

The SIMPLE language has a specified grammar. An equivalent grammar to the SIMPLE language is shown here (note that bolded terms symbolize terminals in our grammar):

program -> statement whileprogram
whileprogram -> program | ε
statement -> **display** expression readoption
| **assign id =** expression
| **while** expression **do** program **end**
| **if** expression **then** program elseoption **end**
readoption -> **read id** | ε
elseoption -> **else** program | ε

expression -> addExp whileExpression
whileExpression -> **relop** addExp | ε
addExp -> multExp whileAddExp
whileAddExp -> + multExp | - multExp | ε
multExp -> negExp whileMultExp
whileMultExp -> * negExp | / negExp | ε
negExp -> -value | value
value -> **id | number |** (expression)

Minor semantic changes have been taken to facilitate the implementation of this grammar in the code in comparison to the supplied grammar. The original terminals P, St1, and St2 have been changed to whileprogram, readoption, and elseoption respectively. The new names for these nonterminals are self-documenting and provide context for their use in the grammar.

The expression, addExpr, multExpr, and negExpr terminals have been left-factored to allow for recursive descent parsing. Consider the previous definition for the expression terminal:

expression -> expression **relop** addExpr | addExpr

At the beginning of a method that parses an expression, it would immediately call itself and lead to stack overflow, never processing the relational operator or the next expressions. Therefore, left-factoring allows our new grammar to be immediately implemented using recursive descent parsing.

SIMPLE Interpreter Design in Java

The code design to implement the SIMPLE interpreter is outlined in this section. The source code structure we will use to implement the interpreter in Java is shown below:

Main.java
scanner
    Scanner.java
    ScanErrorException.java
    Token.java
environment
    Environment.java
parser
    Parser.java
ast
    Statement.java
    Display.java
    ReadOption.java
    Assignment.java
    While.java
    If.java
    Expression.java
    BinRelOp.java
    Number.java
    Variable.java

In the assembly of a scanner that reflects our regular language, we will build a Scanner that uses a one-character look ahead to return tokens of our regular language. This scanner will return tokens until the end of file is reached. A ScanErrorException class extending the Exception class is created to indicate tokens not part of our regular language. A Token class manages a lexeme and a token type. Given an identifier token that matches a keyword, the created token is changed to reflect the keyword token type.

We will then create an Environment class. Environment will use a Map<String, Integer> object to store and get the values of variables. One global environment will be created and passed around as a parameter in execution of each of the elements of the abstract syntax tree (within the ast package).

We will create a Parser class that has a parse method for every element of the abstract syntax tree. The Parser class will have a static interpret method that will initialize a Parser and execute a parsed program. The Main class, responsible for comparing expected and observed outputs, will call this method in testing.

In building a recursive descent parser, we will parse a full Program instance that contains the abstract syntax tree of an input SIMPLE program (a Program is the root of our abstract syntax tree). Elements of the abstract syntax tree are extracted into the grammar and built into classes. Notice how through the grammar above, a program consists of a list of statements; therefore, a Program class will hold a List<Statement> object as its instance variable. A sample of SIMPLE code is parsed using the parseProgram method and is executed using an execution method that passes around an instance of the Environment class. We will parse all Expression objects along the way, regardless of their type (more on that soon); we will deal with type mismatch errors in execution.

We will create an abstract Expression class that will be overridden by expression elements from our grammar; these classes are Number, Variable, and BinRelOp. Notice how, through the construction of our grammar, an expression can evaluate to boolean or integer type. Therefore, we will create two eval methods with return types of integer or boolean, which also pass around the instance of the Environment class from Program. Number will convert an incoming String lexeme in a number token type to an integer and return it in the Integer eval method. It will throw a RuntimeException in the boolean eval method as Number will only evaluate to an integer type. Variable mimics Number in every way, except for two key differences: it will take a String lexeme in an identifier token type, and in its integer eval method, Variable will return the identifier's value from the environment. BinOp will evaluate in type integer and wants an arithmetic operand, while RelOp will evaluate in type boolean and wants a relational operand; both will throw exceptions for the evaluation of the other type. They are very similar, though, as they both will store a left and a right expression and evaluate them to integers; they will then return an integer or boolean evaluation respectively.

We will create an abstract Statement class that will be overridden by statement elements from our grammar; these classes are Display, Assignment, If, and While. The Statement class will have one abstract exec method to execute the statement, passing around the instance of the Environment class from Program. The Display class will take an Expression of any type, evaluate it (try-catch block evaluating integer first, then boolean as fallback), and print it out to the screen; it will also take a ReadOption instance and execute a reading and assignment if the passed instance is not null. To clarify, ReadOption is a utility class and will not extend Statement. The Assignment statement will take a String variable name and an integer Expression and associate the evaluation of the expression with the variable name in the passed Environment. The If class will take a boolean Expression and two Program instances (one for the "then" Program and one for the "else" Program); it will execute the "then" Program if the passed Expression evaluates to true and the "else" Program if it evaluates to false. Of course, the "else" block is not required as seen in our grammar, and the If class will skip over the "else" Program if the passed "else" Program is null. The While class will take a boolean Expression and a Program and will continue to execute the Program if the Expression keeps evaluating to true, exactly like a typical while loop control structure.

The Parser class is at the heart of the program. Using tokens supplied by the Scanner, it will have parse methods for every part of the abstract syntax tree. By executing the parsed Program with one global Environment instance, the inputted SIMPLE code will be interpreted.

Testing the SIMPLE Interpreter

We want to test the ability of our SIMPLE interpreter to parse and interpret the language. We also want to test how the interpreter handles errors in SIMPLE code. In the first test, consider the following SIMPLE code segment and our interpreter output:

```
1   display 3 >= 6 read x                false
2   display x                            -2  (user input)
3   assign y = 5                         -2
4   display 3+x = 6+y                    true
5   read y                               3  (user input)
6   display y                            3
```

The parser will see the first display string and evaluate the Expression 3 >= 6. It sees a boolean output and prints out false. Then through the read option after a display statement, we can ask for the value of x from the user. We display the user inputted x variable by getting its value from the Environment. We then assign 5 to y. Then we display the boolean Expression 3 + x = 6 + y. Due to the grammar of our parser, it will find the RelOp first, and make 3 + x and 6 + y the two expressions and evaluate them. Both of those BinOp expressions are evaluated to integers and compared, resulting in a true evaluation. We then see that we can overwrite the value of y through the correct display of 3 instead of the previously assigned 5. Through this test, we have verified the strength of our parser for two reasons: our parser displayed the correct answer and identified the expression as a boolean expression in line 5, and x was evaluated to -2 in line 2, confirming that negative numbers work as well. Additionally, we have also confirmed the functionality of our Scanner, ScanErrorException, Token, Environment, Program, Statement, Display, ReadOption, Number, Variable, BinOp, RelOp, Expression, and Assignment classes.

Consider the following code segment that tests the If and While classes:

```
1   display 4/3 read limit
2   assign x = 1
3   assign count = 0

4   while count < limit do
5     display x
6     assign count = count+1
7     assign x = x*(x+1)
8   end

9   if count = limit
10    then
11      display count=limit
12      if (x < 3)
13      then display -600
14      else display (x+1)*3
15    end
16  end

17  if (limit+5)
18    then display limit end
```

```
1
5 (user input)
1
2
6
24
120
true
2163
Exception in thread "main"
java.lang.RuntimeException:
RelOp can only be executed as
boolean. Found op: +
```

Analyzing the while loop starting on line 4, we see that the while loop will compute the factorial from numbers 1 to limit. After getting 5 as the limit, it displays the factorial, increments the count, and changes the value of x to the next answer. The boolean expression of the while loop as continually re-evaluated until false. Looking at the displayed console output on the right, we see that the while loop behaves as expected. The count variable equals the limit variable, making the next if statement true, and this is shown in the console output of true. Then, (x+1)*3, the expression in the else block, is evaluated and displayed due to x<3 evaluating to false. Notice how parentheses were used successfully in evaluation. Also notice how the outside if block has no else block, showing that the else block is optional. Finally, because the Expression class has both boolean and integer values, we must check if the If class accepts an integer. In parsing, the expression is accepted; however, during evaluation, a RuntimeException is thrown, showing that If and While can handle illegal Expression instances.

Through these two tests, we can see our code can successfully interpret all SIMPLE code.

Conclusions

Challenges included being able to use two read statements in the same program. The problem lied in the way that Java handles System.in. The Scanner class necessary to read from System.in closes System.in when its close method is called. Declaring a public, static, and final Scanner in ReadOption and finally closing it in the Main method solved the problem.

In the end, we have built a SIMPLE interpreter capable of parsing powerful programs. The SIMPLE language lacks strings to notify the user of what is going on, but even the demonstration of factorial in the second test example shows that the SIMPLE language is primitive yet capable.

Source Code

The source code and compiled classes used to build the SIMPLE interpreter is included in the src folder within this zip package.