CS280
Programming Assignment 3
CS280 Fall 2018

Using the token definitions from programming assignment 2, we can construct a language with the following grammar rules:

```
Prog := Slist
Slist := Stmt SC { Slist }
Stmt := IfStmt | PrintStmt | Expr
IfStmt := IF Expr THEN Stmt
PrintStmt := PRINT Expr
Expr := LogicExpr { ASSIGN LogicExpr }
LogicExpr := CompareExpr { (LOGICAND | LOGICOR) CompareExpr }
CompareExpr := AddExpr { (EQ | NEQ | GT | GEQ | LT | LEQ) AddExpr }
AddExpr := MulExpr { (PLUS | MINUS) MulExpr }
MulExpr := Factor { (STAR | SLASH) Factor }
Factor := MINUS Primary | Primary
Primary := IDENT | ICONST | SCONST | TRUE | FALSE | LPAREN Expr
RPAREN
```

This language will be used for the remainder of the semester.

The following items describe the language.

1. The language contains three types: integer, string, and boolean.
2. All operators are left associative except for ASSIGN and unary MINUS, which are both right associative.
3. An IfStmt evaluates the Expr. The expr must evaluate to a boolean. If the boolean is true, then the Stmt is executed.
4. A PrintStmt evaluates the Expr and prints its value. A newline is not appended.
5. In an Expr, the ASSIGN operator copies the value from the operand on the right side of the operator to the operand on the left side of the operator. The left side of the operator must be an IDENT. If the IDENT does not exist, it is created. If the ident already exists, its value is replaced. The value of the Expr is the value of the identifier, and the type of the Expr is the type associated with the value.
6. The type of an IDENT is the type of the value assigned to it.
7. The LogicExpr performs short-circuited evaluation of logical and and or.
8. The CompareExpr performs a comparison between its operands.
9. The PLUS and MINUS operators in AddExpr represent addition and subtraction.
10. The STAR and SLASH operators in MulExpr represents multiplication and division.
11. The MINUS operator in Factor is a unary minus.
12. The type of TRUE is boolean true. The type of FALSE is boolean false.

13. It is an error if a variable is used before a value is assigned to it.
14. Addition is defined between two integers (the result being the sum) or two strings (the result being the concatenation).
15. Subtraction is defined between two integers (the result being the difference).
16. Multiplication is defined between two integers (the result being the product) or for an integer and a string (the result being the string repeated integer times).
17. Division is defined between two integers
18. Logical and and logical or are defined between two booleans. The result is the boolean value of the logical operation.
19. Comparisons for equality and inequality is defined between pairs of integers, pairs of strings, and pairs of booleans. The result is the boolean result of the test.
20. Comparisons for other than equality and inequality are defined between pairs of integers and pairs of strings.
21. Unary minus is defined for an integer (which is defined as -1 * the integer) and for boolean (which is defined as a logical not).
22. Performing an operation with incorrect types or type combinations is an error.
23. Multiplying a string by a negative integer is an error.
24. Assigning to anything other than an identifier is an error.
25. An IF statement whose Expr is not boolean typed is an error.

Note that by the time the semester is over, you will need to handle all of these items that describe the language. However, you will not need to handle most of them in assignment 3.

For Programming Assignment 3, you MUST implement a recursive descent parser. You may use the lexical analyzer you wrote for Assignment 2, OR you may use a solution provided by the professor.

A skeleton for the solution, with some of the rules implemented, is provided as starter code. You may use this to begin the assignment if you like.

You must create a test program for your parser. The program takes zero or one command line argument. If zero command line arguments are specified, the program should take input from the standard input. If one command line argument is specified, the program should use the argument as a file name and take input from that file. If the file cannot be opened, the program should print COULD NOT OPEN followed by the name of the file, and should then stop. If more than one command line argument is specified, the program should print TOO MANY FILENAMES, and should then stop.

The result of an unsuccessful parse is a set of error messages printed by the parse functions. If the parse fails, the program should stop after the parse function returns.

The assignment does not specify the exact error messages that should be printed out by the parse; however the format of the message should be the line number, followed by a colon and a

space, followed by some descriptive text. Suggested messages might include “No statements in program”, “Missing semicolon”, “Missing identifier after set”, etc.

The result of a successful parse is a parse tree.

Assignment 3 is meant to test that you can properly detect poorly formed programs, and that you can traverse the parse tree for well formed programs.

If a parse tree has been generated, it should be traversed to perform various tests.

The following information must be printed out for the parse tree:

1. Number of leaves (Format is LEAF COUNT: N, where N is the number of leaves)
2. Number of string constants (Format is STRING COUNT: N)
3. Number of identifiers (Format is IDENT COUNT: N, where N is the number of identifiers)
4. Comma separated list of the most frequently used identifiers

If there are no identifiers, steps 3 and 4 can be skipped

NOTE that your program might be graded using different input file names and error cases. SOLVE THE GENERAL PROBLEM and DO NOT HARDCODE output for test cases.

PART 1:

- Compiles
- Argument error cases
- Files that cannot be opened
- Too many filenames

PART 2:

- Statement lists, statements, and all primaries

PART 3

- Everything else