Project 2

CMSC 330 6381

Professor Ellis, Brenda

Wanner HernandezR

March 9, 2021

**Assignment Details**

The second project involves completing and extending the C++ program that evaluates statements of an expression language contained in the module 3 case study.

The statements of that expression language consist of an arithmetic expression followed by a list of assignments. Assignments are separated from the expression and each other by commas. A semicolon terminates the expression. The arithmetic expressions are fully parenthesized infix expressions containing integer literals and variables. The valid arithmetic operators are +, –, \*, /. Tokens can be separated by any number of spaces. Variable names begin with an alphabetic character, followed by any number of alphanumeric characters. Variable names are case sensitive. This syntax is described by BNF and regular expressions in the case study.

The program reads in the arithmetic expression and encodes the expression as a binary tree. After the expression has been read in, the variable assignments are read in and the variables and their values of the variables are placed into the symbol table. Finally the expression is evaluated recursively.

Your first task is to complete the program provided by providing the three missing classes, Minus, Times and Divide.

Next, you should extend the program so that it supports relational, logical and conditional expression operators as defined by the following extension to the grammar:

<exp> -> '(' <operand> <op> <operand> ')' |  
'(' <operand> ':' <operand> '?' <operand> ')' | '(' <operand> '!' ')'

<op> -> '+' | '-' | '\*' | '/' | '>' | '<' | '=' | '&' | '|'

Note that there are a few differences in the use of these operators compared to their customary use in the C family of languages. Their differences are:

* •  In the conditional expression operator, the symbols are reversed and the third operand represents the condition. The first operand is the value when true and the second the value when false
* •  The logical operators use single symbols not double, for example the *and* operator is & not &&
* •  The negation operator ! is a postfix operator, not a prefix one
* •  There are only three relational operators not the usual six and the operator for equality

is = not ==

Like C and C++, any arithmetic expression can be interpreted as a logical value, taking 0 as false and anything else as true

Your final task is to make the following two modifications to the program:

• The program should accept input from a file, allowing for multiple expressions arranged one per line.

1

• All results should be changed from double to int. In particular the evaluate function should return an int.

You may assume that all input to the program is syntactically correct.

## **Process and Lessons Learned**

This code was really changeling and hard at some point, but it thought me so many skills and the ability to improve my coding skills. This Lessons taught me to Implement an expression Interpreter and taught me new kills that can be used to create and read operations like Arithmetic, Logic and etc. The code is far from perfect, but it has the ability to solve simple problems which it is really cool and excited. Overall, I enjoy doing this assignment and I learn so much in doing so.

Below I have explained in depth on the step I took to creating this project.

Emitter –

* I implemented utput of the recursive descent parser. In this implementation, Emitter will produce a Swing JFrame which is the result of the parsing activity over a provided source document.
* Emit stores and acts on the activity of the parser. This method will initialize and use two structures, a symbolTable and a parsingStack. The parsingStack is loaded with postfix expressions. Effectively it contains a linear representation of a parse tree. The symbolTable is loaded with values only when variables are declared with values.
* Evaluate the contents of the parsingStack. This method will consume the parsing stack in reverse order and evaluate the contents. Whenever a variable is encountered it is replaced with the value in the symbol stack. We implement a stack machine for the evaluation of expressions. As a result we create a stack called args. Non operators go on this stack. Operators cause data to be removed from the stack to be replaced by results of the operation.

LexicalScanner –

* The LexicalScanner generates a stream of tokens. Based on the specified grammar and defined lexemes, the lexical scanner generates a stream of tokens.
* Construct a lexical scanner to scan the contents of a reader stream. The stream is assumed to be unbuffered and should consist of the program to be parsed. This class does not close the stream.
* Advance to the next token. This method is the heavy lifter of the lexical scanner. It keeps track of line number and character offsets of tokens and consumes the reader. It will either return a token or throw a syntax error or I/O exception if a fault is detected. It will NOT close the reader at end of file detection. Instead, it will continuously return an ENDOFINPUT token.

Main –

* The Main class provides a command line launch of the parser. This class consists of the public static void main entry point.

ParseException –

* ParseException is used to identify problems in recursive descent parsing. The recursive descent parser can find defects in the code it's parsing these are reported via this exception.

RecursiveDescentParser –

* This class implements a Recursive Descent Parser. It parses the grammar as specified in the Project 2 homework assignment. This grammar is intended to represent a domain specific language designed to provide an expression interpreter. This code will act like an interpreter in that it will evaluate the expressions.
* Parse a program from a Reader producing a evaluation result. This method will parse the contents of a Reader and generate an expression evaluation result.
* Parse a program from a String producing a evaluation result. This method will parse the contents of a Reader and generate an expression evaluation result.
* Parse a program from a File producing a evaluation result. This method will parse the contents of a Reader and generate an expression evaluation result.
* Match the expected type(s) and consume the next token. This method will match the token type to the current token and then consume the next token from the stream. A null <code>types</code> always matches.
* Match the expected type(s) and consume the next token. This method will match the token type(s) to the current token and then consume the next token from the stream if there was a match. A null <code>types</code> always matches.

SyntaxException –

* SyntaxException is used to identify lexical parsing problems. Whenever there is a problem identifying a token, this exception is raised.

Tokern –

* Token is the representation of the basic lexeme of the language. Tokens embody the type content and location in the source file where they were found.
* Tokens are immutable. Once constructed they cannot be changed.

Type –

* The Type enumeration captures the definition of each token type in the language. Each element of the enumeration has a compiled regular expression (except synthetic types) used to parse the input readers. Each regular expression consumes input. RE's that reduce to a\* or a? are not permitted based on the implementation of LexicalScanner not supporting multiple concurrent matching tokens while scanning. White space is not considered as a token type.
* A matcher for the given Type. Return a matcher that can identify the compiled regular expression.

**Compiled:**

**Graphical user interface, text, application

Description automatically generated**

**Test case Input/ Images:**

**Test case Error 1:**

Compiled error

Image:

Graphical user interface, text, application

Description automatically generated

**Test case Error 2:**

(x + (y / 12) – 3), x = 2, y = 6;

PARSING ERROR: Unexpected content [ - ] – expecting [)] at [ 1:14]

Image:

Graphical user interface, text, application

Description automatically generated

**Test case Error 3:**

(x + (y - 3)), x = 2, y = - 6;

PARSING ERROR: Unexpected content [ - ] – expecting [ENDOFINPUT token] at [ 3:27]

Image:

Graphical user interface, text, application

Description automatically generated

**Test cases**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No. | Input | Operator Type | Output | Pass/Fail |
| 1 | (x + y), x = 2, y = 3; | Arithmetic | Value = 5 | Pass |
| 2 | (x & y), x = 1, y = 0; | Logical | Value = 0 | Pass |
| 3 | (x | y), x = 1, y = 0; | Logical | Value = 1 | Pass |
| 4 | ((x + 3) > y), x = 2, y = 4; | Arithmetic  Relational | Value = 1 | Pass |
| 5 | (x < (y - 2)), x = 3, y = 3; | Arithmetic  Relational | Value = 0 | Pass |
| 6 | ((x & (4 > y))!), x = 1, y = 3; | Logical  Relational | Value = 0 | Pass |
| 7 | (x : y ? (z = 1)), x = 0, y = 1, z = 2; | Ternary Conditional  Relational | Value = 1 | Pass |
| 8 | (x : y ? (z = 1)), x = 0, y = 2, z = 1; | Ternary Conditional  Relational | Value = 0 | Pass |
| 9 | ((x + 1) : y ? (z > 1)), x = 0, y = 2, z = 3; | Arithmetic  Ternary Conditional  Relational | Value = 1 | Pass |
| 10 | (x : (y + 1) ? (z < 1)), x = 0, y = 2, z = 3; | Ternary Conditional  Relational | Value = 3 | Pass |

**Test case 1:**

Graphical user interface, text, application

Description automatically generated

**Test case 2:**

**Graphical user interface, text, application

Description automatically generated**

**Test case 3:**

**Graphical user interface, text, application

Description automatically generated**

**Test case 4:**

**Graphical user interface, text, application, email

Description automatically generated**

**Test case 5:**

**Graphical user interface, text, application, email

Description automatically generated**

**Test case 6:**

**Graphical user interface, text, application, email

Description automatically generated**

**Test case 7:**

**Graphical user interface, text, application, email

Description automatically generated**

**Test case 8:**

**Graphical user interface, text, application

Description automatically generated**

**Test case 9:**

**Graphical user interface, text, application

Description automatically generated**

**Test case 10:**

**Graphical user interface, text, application

Description automatically generated**