Project Three: An Interpreter for Arithmetic Expressions That Uses A Recursive Descent Parser

1. Introduction

Recursive descent parsing can be viewed as an attempt to find the leftmost derivation for an input string. It can also be viewed as a method for attempting to create a parse tree for the input string starting at the root and creating the nodes of parse tree in order.

This form of recursive descent parsing is a top down parser (therefore left recursion is **not** allowed) and the parser may have to perform backtracking. Recursive descent parsers like this one are not very common because backtracking is rarely needed to parse programming languages and backtracking is not very efficient for parsing natural languages.

However, recursive descent parsers are relatively easy to construct and convey the general idea of what a parser does.

2. Objective

For this project you will write an interpreter in C for arithmetic expressions as defined by a particular grammar that is implemented using a recursive descent parser. Your parser is a language recognizer in that it will determine whether each expression that is input is syntactically correct (you will also catch any lexical errors). Your parser will also be an interpreter in that expressions that are syntactically correct it will evaluate and return the resulting
value.

3. Grammar

Your parser will use the following grammar:

<bexpr> → <expr>;
<expr> → <term> <ttail>
<ttail> → <add_sub_tok> <term> <ttail> | epsilon
<term> → <stmt> <stail>
<stail> → <mult_div_tok> <stmt> <stail> | epsilon
<stmt> → <factor> <ftail>
<ftail> → <compare_tok> <factor> <ftail> | epsilon
<factor> → <expp> ^ <factor> | <expp>
<expp> → (<expr>) | <num>
<add_sub_tok> → + | -
<mul_div_tok> → * | /
<compare_tok> → < | > | <= | >= | != | ==
<num> → {0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 }+

The name ttail is short for term_tail. The name ftail is short for factor_tail. The name stail is short for statement_tail. The words factor and term are commonly used when describing arithmetic expressions.

Recursive descent parsers are a type of LL parser as described in Subsection 4.3.2 and Section 4.4. This requires that the grammar not have left recursion (see pages 191-192) and so the above grammar is not left recursive. Recall from Subsection 3.3.1.9 that left recursion in a grammar for expressions implies left associativity and that right recursion in a grammar for expressions implies right associativity.

The above grammar does have right recursion. For example, the non-terminal **ttail** is on the right end of the first rule for the **ttail** non-terminal. So, if you build the parse tree for an expression that has two operators of equal precedence such as

6 - 7 + 8

you will see that the plus operator is lower in the parse tree (further from the root) than the minus operator. As we discussed in Chapter 3 (see page 126) then means that the plus operator will be evaluated first with the parse tree specifying the evaluation order.

However, the semantic routines of the compiler can override the evaluation order implied by the parse tree to ensure a different associativity (in this case, left associativity instead of right associativity). See the sentence ``Fortunately, left associativity can be enforced by the compiler, even though the grammar does not dictate it" (page 130) and ``However, it is relatively easy to design the code generation based on this grammar so that the addition and multiplication operators will have left associativity" (page 192).

Among operators of equal precedence according to the grammar your parser/interpreter must use left associativity for the evaluation of all of addition, subtraction, multiplication, division, and the relational operators (<, <=, >, >=, ==, and !=). For exponentiation it must use right associativity.

The relational operators evaluate to 0 for false and 1 for true. Thus, for example

1 + 7 == 9 < 2;

first evaluates 7 == 9 which is false which is 0 so it becomes

1 + 0 < 2;

which then evaluates 0 < 2 which is true which is 1 so it becomes

1 + 1;

which then evaluates to be 2.

Your program must handle syntax errors. Also, your program must handle the lexical error of a character that is not a lexeme in the language. For example, if the input is

2 + @ < 2;

Your output should be the following because @ is not a lexeme in the language.

===> `@`
Lexical Error: not a lexeme

There can be whitespace inside or before or after an expression. Whitespace can be blank characters, tab characters, or newline characters. For example, 2 + 5 ; is a valid expression that has some whitespace inside it. But you can assume an expression is all on one line.

4. Required Functionality and Design

4.1. Basic Design Features

Your program will then read expressions from a file. Whenever it encounters an expression it will

- evaluate the expression for syntactical correctness and lexical correctness as well as
- write to the output file the expression and the proper result for the value of that expression (see the examples for the format)

If an expression has a syntax error or a lexical error, you should report in your output. You do not have to produce an expression evaluation value for syntactically or lexically invalid expressions.

You will need to write a function for each non-terminal. Your functions **must** have the same name as the non-terminal they model. The body of each function will model the Right Hand Side(RHS) of the production for that non-terminal.

For example: You will write a function named **expr** that will call a function named **term** then call a function named **ttail**.

Each function that models a non-terminal should do the following:

- Accept a character string as a parameter that holds the current token from the input stream.
- If the function expects a terminal character, check and make sure that the **expected** terminal and the **actual** terminal character match. If they do match, get the next token from the input stream. If they do **not** match then print an error condition and return an error value.
- Call the appropriate function(s) based upon the non-terminals listed in the RHS of the production.
- If the current token is consumed during a function, then call your get_token function to get the next token from the input stream.

NOTE: You may use a modified version of the get_token function you wrote for your last project to get the next token from the input stream in this program. You will most likely have to make some modifications to get_token() so it will work as needed for this program.

4.2. Further Constraints

You will use the following conventions:

- All valid expression must end with a ;
- A value of true will be represented by the number 1.
- A value of false will be represented by the number 0.
- The ^ symbol represents exponentiation.
- The + symbol represents addition.

- The - symbol represents subtraction.
- The * symbol represents multiplication.
- The / symbol represents division.
- The <, >, <=, >=, !=, and == represent less than, greater than, less than or equal, greater than or equal, not equal, and equals respectively.
- Use the value -999999 to represent an error value. If you want to use different values for the two cases of a lexical error and a syntax error that is fine.
- When an error is encountered, stop parsing the current line and pass the error value back up to main. Then go on to processing the next statement as shown in the example output file.
- Your main function must be in a file called **interpreter.c** or **main.c**
- Your tokenizer must be in a file called **tokenizer.c**
- Your functions needed for parsing must be in a file called **parser.c**
- Create header files as needed to resolve conflicts. You may not copy and paste prototypes needed into C files themselves, you must instead use **#include** to accomplish this.
- You can only #include header files (that is, .h files), not .c files. To #include a .c file into another .c file defeats one of the main reasons to use multiple source files in C.
- Use constants instead of magic numbers where possible.
- The atoi() function might be helpful for converting a string to an integer. The function name stands for Ascii To Integer and ASCII is an old name for the standard character-encoding (it is the predecessor to Unicode).
- If you want to access in main.c the variable named line which is a global variable in the file tokenizer.c, then in main.c near the top have

  extern char * line;

  This does not create a new variable named line. It just says that a variable named line exists as a global variable in another file and can be used in this file.
- If you find it helpful, you can add one or two more global variables to any of the three source files you made do so. Remember to use an extern statement in the other files if you want to access the global variables in those other files.
- You must use file-based input and output. You execute your program by the command

./interpreter input_file_name.txt output_file_name.txt

or

./main input_file_name.txt output_file_name.txt

4.3. Compiling with Multiple Source Files

As mentioned above you will have three .c files: interpreter.c, tokenizer.c, and parser.c. You will also have several header files (that is, .h files). In C you use #include preprocessor directives to include header files in .c files. The header files are where you put your #defines and your function prototypes. You do not use header files to hold regular C source code such as variable declarations and function definitions.

You do not use #include directives to include other .c files in a .c file. To #include a .c file in another .c file just means that after the preprocessor has run you have just put all your source code in one giant file and then have to compile all the source code together at the same time. This cancels one of the

main reasons to have multiple .c files. With multiple .c files you can compile individual .c files into .o files (that is, object modules) and then just link the object modules together. For example, suppose you are not currently debugging tokenizer.c and interpreter.c. In this case, you can just one time do

gcc -c tokenizer.c

which creates an object file named tokenizer.o and

gcc -c interpreter.c

which creates an object file named interpreter.o. You can then get into a debug cycle with your parser.c by doing

gcc parser.c tokenizer.o interpreter.o

which compiles your parser.c into a parser.o and then links parser.o with tokenizer.o and interpreter.o to create your a.out executable. Your debug cycle will be faster because compilation will be faster because tokenizer.o and interpreter.o have already been compiled; you are only compiling parser.c. This won't work if you are #including one .c file into another.

5. Provided Code and Example Input and Output

5.1 Provided Source Code

I have provided a **parser.h** file that specifies the function prototypes you will need for all the non-terminals. I have provided a **parser.c** file that has the grammar as a comment and functions for two of the non-terminals: **expr** and **ttail**. The functions for **expr** and **ttail** illustrate how you to implement the rest of the non-terminals.

Recall that your program is both a parser (that is, a language recognizer determining whether a sentence is syntactically correct) and an interpreter (that is, evaluating the value of the expression that is syntactically correct). The interpreter functionality which I refer to as the semantic routines is not actually separate routines in this program. Instead in the two example functions, that the functionality is represented by the variable **subtotal** and operations on it. For example, in the function **ttail** the code fragment

subtotal + term_value

is performing the expression evaluation in the case of a plus operator token.

5.2 Example Input File and Output File

Tthe example input file of expressions is on the class web page as the file input_left_associative.txt. The example output and output file of expressions and results for are listed on the class web page as output_left_associative.txt. The output assumes your semantic routines implement left associativity for all operators except for exponentiation. Your program's output must be the same as in this sample.

5.3 Debugging Hints

If you add printf() statements to create a trace of your program as you are debugging your program, remember to end each printf() statement with "\n". A printf() statement does not automatically immediately cause its argument to be displayed on the console. All the printf() statement does, in general, is add its output string to the output buffer which at some time in the future is emptied. The "n" forces the content of the output buffer to be emptied at that point in time. This is important since otherwise, a segmentation fault might not be occurring where you think it is since the intermediate printf() output strings are not getting printed.

Executing your program within the gnu debugger, gdb, can be helpful. It allows breakpoints and single-stepping. It also shows a complete stack trace upon a segmentation fault. The handout for Project 1 has a section with more information about gdb.

6. Programming Style and Documentation

Your program must follow good programming style as reflected in the programs in the McDowell textbook. The programming style should also be consistent with the Sun's Java Coding Standard (see http://www.oracle.com/technetwork/java/codeconv-138413.html) to the extent possible given that your program is in C instead of Java.

Your program must be fully commented which means

- On agora create a project3 directory that has in it a README text file and the files for your project source code. The README file lists the author names, date, that this is Project 3, explains how to compile your program (including the exact command to enter) and how to run your program (including showing the exact command to be entered), a description of the purpose of the program, and any aspects of the program that do not work. You will lose fewer points for parts that do not work that you tell me about.
- every file must have a start-of-file comment at the top with your name, date, and the name of the project and a description of what the code in that file does. The provided parser.c file has a start of file comment already. You need to modify that start of file comment since you modified that source file. So you need to add your name, date, and description of the code in the file after you have made your changes.
- every function must have a comment above the header which is like a javadoc method header comment (including describing every parameter and return value). Comment other lines of code when you think it will help the readability of your program

7. Team Assignments, Grading and Turn-In

If you did Project 2 by yourself you can do Project 3 with one other person in the class. In that case you can start with either of your Project 2 solutions. If you did Project 2 with someone else, you need to do Project 3 with that same person.

If your program does not compile, the score is zero. If your program is composed of multiple files tar everything up into one file for submission. The project is due at 5 pm on Monday, the 6th of April. Submit your files as a tar file via **handin** on agora. If your tar file is called **project3.tar.gz**, then the command will be:

handin.352.1 3 project3.tar.gz

If you are in the directory containing the project3 directory as a subdirectory, then the command

tar -cvzf project3.tar.gz project3

will create the project3.tar.gz file in the current directo**ry.**

Projects can be turned in late with my normal late policy. That policy is that every day late is three points off (so 5:01 pm is one day late) not including weekends and holidays up to a maximum of thirty late points. The last day to turn in late projects is the Friday of the next to last week of classes which is Friday, the 24th of April.