CS 352 Project 3
Spring 2014 Andrew Scott

An Interpreter for Arithmetic Expressions That Uses A Recursive Descent Parser

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.

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. Your parser will also be an interpreter in that expressions that are syntactically correct it will evaluate and return the resulting value.

Grammar

Your parser will use the following grammar:

h*bexpr*i → h*expr*i ;

$$
\begin{array}{lcl}
\langle expr \rangle & \rightarrow & \langle term \rangle\ \langle ttail \rangle \\
\langle ttail \rangle & \rightarrow & \langle add\_sub\_tok \rangle\ \langle term \rangle\ \langle ttail \rangle \mid \varepsilon \\
\langle term \rangle & \rightarrow & \langle stmt \rangle\ \langle stail \rangle \\
\langle stail \rangle & \rightarrow & \langle mult\_div\_tok \rangle\ \langle stmt \rangle\ \langle stail \rangle \mid \varepsilon \\
\langle stmt \rangle & \rightarrow & \langle logic \rangle\ \langle ltail \rangle \\
\langle ltail \rangle & \rightarrow & \langle log\_tok \rangle\ \langle logic \rangle\ \langle ltail \rangle \mid \varepsilon \\
\langle logic \rangle & \rightarrow & \langle expp \rangle\ \hat{}\ \langle logic \rangle \mid \langle expp \rangle \\
\langle expp \rangle & \rightarrow & (\ \langle expr \rangle\ ) \mid \langle num \rangle \\
\langle add\_sub\_tok \rangle & \rightarrow & + \mid - \\
\langle mul\_div\_tok \rangle & \rightarrow & * \mid / \\
\langle log\_tok \rangle & \rightarrow & < \mid > \mid <= \mid >= \mid != \mid == \\
\langle num \rangle & \rightarrow & \{0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9\}+
\end{array}
$$

Recursive descent parsers are a type of LL parser as described in Subsection 4.3.2 and Section 4.4. This requires that the grammer 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 grammer does have right recursion. For example, the non-terminal ttail is on the right end of the first rule for the ttail nonterminal. 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).

Your parser/interpreter can use either left associativity for the evaluation of all of addition, subtraction, multiplication, division, and the relational operators (<, <=, >, >=, ==, and ! =) or it can use right associativity for the evaluation of all of them. You just need to be consistent. For exponentiation it should 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.

Required Functionality and Design

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:

1. Accept a character string as a parameter that holds the current token from the input stream.
2. 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.
3. 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 gettoken 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.

When you execute your program, it should prompt the user to input an expression, that it will

1. evaluate for syntactical correctness as well as
2. produce the proper result for the value of that expression.

After every expression is evaluated, the program will prompt the user for another expression, until they enter a 'q' to quit the program.

If an expression has a syntax error, you should report this to the user, and ask for another expression to evaluate. You do not have to produce a value for syntactically invalid expressions.

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.
- When an error is encountered, stop parsing the current line and pass the error value back up to main. Then ask the user for another expression.
- Your main function must be in a file called 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.
- Use constants instead of magic numbers where possible.

Provided Code and Example Output

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

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.

Two versions of the output of a sample test run of the program are listed on the class web page. One version assumes your semantic routines implement left associativity for all operators except for exponentiation. One version assumes your semantic routines implement right associativity for all operators.

Alternative Programming Languages for Implementation

Instead of C you can use Javascript or Python to implement Project Three.

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

- every function must have a comment above the header which is like a javadoc method header comment (including describing every parameter and return value)
- every file must have a comment at the top with your name, date, and the name of the project
- in the file containing your main function in the comment at the top with your name there must be a description of the project that explains the purpose of the project
- comment other lines of code when it will help the readability of your program

Grading and Turn-In

The program will be graded using the following rubric.

- 10% Correct comments and style
- 10% Compile correctly (no errors or warnings using -Wall) • 5% Program is composed of multiple files, properly linked together.
- 5% Proper use of constants and formatted output.
- 40% Correctly determines legal/illegal syntax
- 30% Correctly evaluates valid expressions and prints results.

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 on Blackboard. The project is due at 1:25pm on Friday, the 18th of April. Submit your files as a tar file via handin on polaris. If your tar file is called project3.tar, then the command will be: handin.352.1 3 project3.tar