Comp 80 – Programming Languages – Spring 2003

Programming Project 1

Due date: Thu 2/6 11:00 pm

Introduction

In this project you will write a recursive descent parser for the simple expression language given in class. The grammar is as follows where the start symbol is S and # stands for the empty string (normally $): 

\[
\begin{align*}
S & \rightarrow E \text{ endexp} \\
E & \rightarrow T T2 \\
T2 & \rightarrow * T T2 \mid \# \\
T & \rightarrow F F2 \\
F2 & \rightarrow \* F F2 \mid \# \\
F & \rightarrow \text{id} \mid \text{num} \mid (E)
\end{align*}
\]

In this grammar endexp, *, *, id, num, (, ) are terminals, and S, E, T, T2, F, F2 are non-terminals. In our language id stands for an identifier (letter followed by combination of letters and digits except for the “reserved word” endexp) and num stands for an integer (sequence of digits).

We are providing you with a lexer for this language. Each call to the lexer returns the next token from the input. Each token is described by a type which corresponds to the terminal of the grammar and a string giving the text of the token. In addition to the terminals, ENDINP signifies the end of the input. The relevant code in the file lexer.h is as follows:

```c
#define PLUS 0
#define MULT 1
#define NUM 2
#define ID 3
#define OPEN 4
#define CLOSE 5
#define ENDEXP 6
#define ENDINP 7

struct token {
  int type;
  char* value;
};
token lexer();
```

The lexer was automatically produced by the lex program (and actually includes C code). You do not need to read its code - it should be sufficient to use the interface above. The lexer is provided as

```
/comp/80/files/pp1/lexer.h
/comp/80/files/pp1/lexer.cpp
```

You should copy lexer.h into your directory and use #include "lexer.h" in your program. Assuming that your code is in expressions.cpp, on andante you can compile using

```
g++ /comp/80/files/pp1/lexer.o expressions.cpp
```

Or you can copy and compile the lexer.cpp file directly.
Your Task

You need to write a recursive descent parser for the our expression language. Your main program should simply call the parser to parse one expression. While parsing you should print messages as follows. When you start parsing non-terminal symbol X you should print “Starting X”. When this non-terminal has been successfully parsed, i.e. just before exiting the procedure for this non-terminal you should print “Exit X”. When you match any terminal expected by the grammar to the input you should also print a statement. For each of +, *, (, ), endexp you should print e.g. “Matched +” replacing the “+” with the relevant symbol and ENDEXP for the last one. For id, num you should print the value as well as in “Matched NUM 3” or “Matched ID x2”. If at any point during parsing, your parser expects one kind of terminal (e.g. a +) and finds another (e.g. a NUM) the parser should print an error message and exit. The error message should indicate the non-terminal currently being parsed as in “ERROR when parsing F2”.

Note that since a trace of a run of a recursive descent parser gives a postorder traversal of the parse tree one can reconstruct the parse tree from your printout. You should find this useful when looking at the examples we provide as well as the output from your program.

The following two examples illustrate the requirements: on input “x2 +3 endexp” the output should be:

Starting S
Starting E
Starting T
Starting F
Matched ID x2
Exit F
Starting F2
Exit F2
Exit T
Starting T2
Matched +
Starting T
Starting F
Matched NUM 3
Exit F
Starting F2
Exit F2
Exit T
Starting T2
Exit T2
Exit T2
Exit E
Matched ENDEXP
Exit S

On input “x2 3 +3 endexp” the output should be:

Starting S
Starting E
Starting T
Starting F
Matched ID x2
Exit F
Starting F2
ERROR when parsing F2

Several larger examples are provided in /comp/80/files/pp1/
**Submitting your program**

Make sure that your code is clear and readable and well documented. The grade will be based both on correctness and on readability.

Assuming your code is in expressions.cpp in the current directory on andante, you should submit by typing

```
provide comp80 ppl expressions.cpp
```

**For Extra Credit**

This part is optional and you can earn for it up to 20 extra points (for a max of 120 points on the project). Extend your code to parenthesize the expression you are parsing correctly. For example on input “2+3*5 endexp” you should output “(2+(3*5))” and on input “2+3*5+X27*7*(2+3*7) endexp” you should output “((2+(3*(5+6)))+((X27*7)*(2+(3*7))))”. Notice that there are no extra parenthesis or missing parenthesis Your program should print nothing else - i.e. your program should print just one line on such an expression. If the parse cannot be completed you should print an error message as in the previous part.

In order to implement this you will need to compute a value for each node of the parse tree (you do it on the fly since we do not actually produce a tree). You’ll probably want to look at Chapter 4 of the text which explains how to handle values by attribute grammars in predictive parsers.

Assuming your code for this part is in expressionsextra.cpp in the current directory on andante, you should submit by typing

```
provide comp80 pplb expressionsextra.cpp
```