CSC521 Project 2

Table-Driven LL(1) Parser for the Calculator Language with Comments Using C Programming

In class we have discussed a simple "calculator" language. Its LL(1) grammar is Figure 2.16 on page 74 of the textbook. The calculator allows values to be read into (numeric) variables, which may then be used in expressions. The values of the expressions, in turn, can be written to the output. It is a simple language because the control flow is strictly linear (no loops, if statements, or other jumps).

A parser takes the output of a scanner as input. Recall that a scanner groups a sequence of characters into tokens, and reports any lexical errors. The basic implementation of the scanner for the calculator language written in C, called "scanner.c", is given.

There are two kinds of implementation of LL top-down parsing: **recursive descent and table-driven**.

1. Recursive Descent Parser (textbook 2.3.1)

A recursive descent parser (RDP) is a parser that constructs a parse tree from the root down, predicting at each step which production will be used to expand the current node, based on the next available token of input. It also reports any syntactical errors if any. In RDP, each subroutine corresponds, one to one, to a nonterminal of the grammar. Example 2.24 on page 75 illustrates how RDP parses a "sum and average" program in simple calculator language.

Figure 2.17 shows the pseudo-code of RDP for calculator language. It verifies that a program is syntactically correct (i.e., that none of the "otherwise parse_error" clauses in the case statements are executed and that match() always sees what it expects to see). The basic implementation of the parser for the calculator language, called "parser.c", is given.

Corresponding to Figure 2.17, there are ten functions in parser.c as follows:

```
void program();
void stmt_list();
void stmt();
void expr();
void term_tail();
void term();
void factor_tail();
void factor();
void add_op();
void mult_op();
```

2. Table Driven Parser (textbook 2.3.3)

In table driven top-down parser, it requires a stack and parse table to do the same job as RDP. The parser iterates around a loop in which it pops the top symbol off the stack and performs the following actions. If the popped symbol is a terminal, the parser attempts to match it against an incoming token from the scanner. If the match fails, the parser announces a syntax error. If the popped symbol is a nonterminal, the parser uses that nonterminal together with the next available input token to index into the two-dimensional table that tells it which production to predict (or whether to announce a syntax error).

The "scanner.c" is given and is partially working. First, it only checks whether the first token in an input string of tokens is valid or not. Second, it does not check for comments. That is, it should check if the comments have the correct forms (either /* ….*/ or //) following the regular expression we discussed in class and in Project 1.

3. Task Description

Your task is to implement a parser using **table-driven parsing**, using C programming language, called "parser.c", that works correctly with "scanner.c".

When executing your programs, it can take a calculator language program (with comments) and find if it has lexical errors and syntax errors or not.

Please compile both programs by typing

```
gcc parser.c scanner.c -o parser
```

To run the program, the user will type at command line:

```
./parser prog1
```

, where `prog1` is the name of the calculator program to be checked input by the user.

Next, your program should output error messages indicating any lexical or syntax error in the calculator program; otherwise, output a "No lexical and syntax error." message.

For example, below calculator program below, called "prog1", is both lexically and syntactically correct:

```
/* prog1 */
read a
read b
area := a * b
perimeter := 2 * (a + b)
write area
write perimeter
```

Below program, called “prog2” is lexically incorrect.

```
/* prog2
read a
read b
area := a * b
perimeter := 2 * (a + b)
write area
write perimeter
```

The below program, called “prog3,” is lexically correct but syntactically incorrect.

```
/* prog3 */ */
read a
read b
area := a * b
perimeter := 2 * (a + b)
write area
write perimeter
```

The below programs, called “prog4” and “prog5”, respectively, are lexically correct but syntactically incorrect.

```
// prog4
read a b
area := a * b
perimeter := 2 * (a + b)
write area
write perimeter

and

// prog5
read a
read b
area := a * b
perimeter := 2 * (a + b
write area
write perimeter
```

If you have more test programs, please name them “prog6”, “prog7”, etc..

Please make sure that the error messages (both lexical and syntactical) are detailed and informative by showing which lines of the program has errors.

Submission: Please submit a .zip file named LastName1_LastName2_LastName3_521_proj2.zip, which includes the two C programs (scanner.c and parser.c), and a readme. In the readme, please indicate clearly a) if your programs run correctly, according to the five test programs, b) and if so, at least five screenshots to support your claim, c) any other issues you want the instructor to know, d) contributions of the three team members.