CECS 342

Lab assignment 2

Due date: Wednesday, September 27

20 points

Introduction to Parsing

• Syntax analysis is often referred to as **parsing.**

• Responsibilities of a syntax analyzer, or parser:

Determine whether the input program is syntactically correct.

Produce a parse tree. In some cases, the parse tree is only implicitly constructed.

• When an error is found, a parser must produce a diagnostic message and recover.

Recovery is required so that the compiler finds as many errors as possible.

• Parsers are categorized according to the direction in which they build parse trees:

Top-down parsers build the tree from the root downward to the leaves.

Bottom-up parsers build the tree from the leaves upward to the root.

Top-Down Parsers

• A top-down parser traces or builds the parse tree in preorder: each node is visited before its branches are followed.

• The actions taken by a top-down parser correspond to a leftmost derivation.

Bottom-Up Parsers

• A bottom-up parser constructs a parse tree by beginning at the leaves and progressing

toward the root. This parse order corresponds to the reverse of a rightmost derivation.

Problem

The Recursive-Descent Parsing Process

• A recursive-descent parser consists of a collection of subprograms, many of which are recursive; it produces a parse tree in top-down order.

• A recursive-descent parser has one subprogram for each nonterminal in the grammar.

• EBNF is ideally suited for recursive-descent parsers.

- An EBNF description of simple arithmetic expressions:

<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → id | int constant | (<expr>)

- These rules can be used to construct a recursive-descent function named `expr` that parses arithmetic expressions.
- The lexical analyzer is assumed to be a function named `lex`. It reads a lexeme and puts its token code in the global variable `nextToken`. Token codes are defined as named constants.

Writing a recursive-descent subprogram for a rule with a single RHS is relatively simple.

For each terminal symbol in the RHS, that terminal symbol is compared with `nextToken`. If they do not match, it is a syntax error. If they match, the lexical analyzer is called to get to the next input token.

For each nonterminal, the parsing subprogram for that nonterminal is called.

A recursive-descent subprogram for <expr>, written in C:

```
/* expr
Parses strings in the language generated by the rule:
<expr> -> <term> {(+ | -) <term>}
*/
void expr(void) {
printf("Enter <expr>\n");
/* Parse the first term */
term();
/* As long as the next token is + or -, get
the next token and parse the next term */
//YOUR CODE

//Display exit message
printf("Exit <expr>\n");
}
```

- Each recursive-descent subprogram, including `expr`, leaves the next input token in `nextToken`.
- `expr` does not include any code for syntax error detection or recovery, because there are no detectable errors associated with the rule for <expr>.

The subprogram for <term> is similar to that for <expr>:

```
/* term
Parses strings in the language generated by the rule:
<term> -> <factor> {(* | /) <factor>}
*/
void term(void) {
printf("Enter <term>\n");
/* Parse the first factor */
//YOUR CODE

/* As long as the next token is * or /, get the
next token and parse the next factor */
//YOUR CODE

//Display exit message
printf("Exit <term>\n");
}
```

A recursive-descent parsing subprogram for a nonterminal whose rule has more than one RHS must examine the value of `nextToken` to determine which RHS is to be parsed.

The recursive-descent subprogram for <factor> must choose between two RHSs:

```
/* factor
Parses strings in the language generated by the rule:
<factor> -> id | int_constant | ( <expr> )
*/
void factor(void) {
printf("Enter <factor>\n");
/* Determine which RHS: variable or constant*/
//YOUR CODE
     /* Get the next token */
     //YOUR CODE
/* If the RHS is ( <expr> ), call lex to pass over the
left parenthesis, call expr, and check for the right
parenthesis (No right parenthesis → syntax error */

       //YOUR CODE

/* It was not an id, an integer literal, or a left
Parenthesis → syntax error */
     //YOUR CODE
//Display exit message
   printf("Exit <factor>\n");
}//close the function
```

- The `error` function is called when a syntax error is detected. A real parser would produce a diagnostic message and attempt to recover from the error.

Grading and submission

Create the following three test files to test the program:

Test1.txt

(sum + 47) / total

Test2.txt

50 47 / x

Test3.txt

 (sum + 47 / total

Output for the file test1.txt

```
Enter <factor>
Next token is: 11, Next lexeme is sum
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 21, Next lexeme is +
Exit <factor>
Exit <term>
Next token is: 10, Next lexeme is 47
Enter <term>
Enter <factor>
Next token is: 26, Next lexeme is )
Exit <factor>
Exit <term>
Exit <expr>
Next token is: 24, Next lexeme is /
Exit <factor>
Next token is: 11, Next lexeme is total
Enter <factor>
Next token is: -1, Next lexeme is EOF
Exit <factor>
Exit <term>
Exit <expr>
```

Output for test2.txt

```
Next token is: 10, Next lexeme is 50
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 10, Next lexeme is 47
Exit <factor>
Exit <term>
Exit <expr>
Next token is: 24, Next lexeme is /
Enter <expr>
Enter <term>
Enter <factor>
SYNTAX ERROR
Exit <factor>
Next token is: 11, Next lexeme is x
Enter <factor>
Next token is: -1, Next lexeme is EOF
Exit <factor>
Exit <term>
Exit <expr>
```

Output for test3.txt

```
Next token is: 25, Next lexeme is (
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 11, Next lexeme is sum
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 21, Next lexeme is +
Exit <factor>
Exit <term>
Next token is: 10, Next lexeme is 47
Enter <term>
Enter <factor>
Next token is: 24, Next lexeme is /
Exit <factor>
Next token is: 11, Next lexeme is total
Enter <factor>
```

```
Next token is: -1, Next lexeme is EOF
Exit <factor>
Exit <term>
Exit <expr>
SYNTAX ERROR
Exit <factor>
Exit <term>
Exit <expr>
```

Grading;

1. All members submit the following files to Canvas:
    a. lab3.c file -20 points)
    b. A pdf file which has a copy of lab3.c and runtime output.

2. Write on the comment the team group number and the completion of the lab assignment for each team member. All the team members must agree with the completion of each team member.

    The program run successfully with correct output (100% completion)
    The program run successfully with incorrect output (60% completion)
    The program has syntax errors or is incomplete (40% completion)
    The program has few coding and syntax errors (30% completion)
    There is no submission (0%)