Assignment

Copy *YPL1Scanner.cpp* into ***YPL1Parser.cpp***, the next step in developing your *YPL* compiler. Study the *SPL1Parser.cpp* program source code and Section 4.4 of our textbook entitled **"Recursive-Descent Parsing"** to learn how to use recursive descent to **parse** source programs. **Syntax analysis** requires the *SPL1* parser to use *GetNextToken()* to **scan** the source file token-by-token, from top-to-bottom, left-to-right. The non-terminal that the entire *SPL1* parser is designed to **recognize** is the *SPL1* **goal symbol <SPLProgram>**.

Take notice of the following observations as you study *SPL1Parser.cpp*

- Every other terminal and non-terminal subsequently recognized by the parser derives, either directly or indirectly, from the *SPL1* goal symbol. That is, by definition, the goal symbol of *SPL1* is the **root** of every *SPL1* **parse tree**.
- The **right-hand-side** of the <SPLProgram> **production rule** contains the non-terminal <PROGRAMDefinition> which implies that the algorithm used to parse <SPLProgram> must include references to the parsing function Dr. Hanna named *ParsePROGRAMDefinition()*.
- Most parsing functions make one or more references to *GetNextToken()* to scan for, to recognize, the next token in the source file. Sometimes it's handy to be able to "look-ahead" a token or two beyond the token currently under consideration. Why? To facilitate correct parsing decisions, of c☺urse!

Now—and this is the most difficult part of the assignment—incorporate the parser design ideas you learned while studying *SPL1Parser.cpp* into *YPL1Parser.cpp*.

To test-and-debug *YPL1Parser.cpp*, design a series of *YPL1* source files that, like Dr. the Hanna source files *P1.spl* through *P3.spl*, fully "white-box" test the logic your parser uses to recognize **grammatically-correct** *YPL1* programs.

Send an email to AHanna@StMaryTX.edu with a single Word document attachment that contains (1) your version of the *YPL1Parser.cpp*; and (2) the list files that your compiler produces when it "compiles" your versions of *P1.spl*, *P2.spl*, and *P3.spl*.

Overview of Recursive-Descent Parsing

A **recursive-descent parser** is a top-down parser built from a set of **mutually-recursive** procedures (or non-recursive equivalents that explicitly use **stacks** to simulate recursion) where each such procedure parses (that is, recognizes) one of the "conceptually-big" non-terminal symbols of the grammar. **Thus the structure of the resulting program closely mirrors the structure of the grammar that it recognizes.**

A **predictive parser** is a recursive-descent parser that does not require backtracking. The *SPL* parser is a predictive parser. Predictive parsing is

possible only for the class of **`LL(k)`** grammars; that is, context-free grammars for which there exists some positive integer `k` that allows a recursive-descent parser to decide which **production (grammar rule)** to use by examining only the next `k` tokens of input. (Note A scanner's "look-ahead" makes the "next `k` tokens of input" available to parsing logic if and only if enough "look-ahead" is provided by the scanner logic.) The `LL(k)` grammars therefore exclude all ambiguous grammars, as well as all grammars that contain left recursion. (Question Why don't we have to worry about left-recursive rules or right-recursive rules?! Hint What does the **metasymbology `{...}*`** mean?) Any context-free grammar can be transformed into an equivalent grammar that has no left recursion, but removal of left recursion does not always yield an `LL(k)` grammar. A predictive parser runs in **linear time**, that is, with a time complexity of **`Θ(n)`**, where `n` is the number of tokens contained in the source file being parsed.

Predictive parsers can be depicted using **state-transition diagrams** for each non-terminal symbol in which the edges between the initial and the final states are labelled with the symbols—both terminals and non-terminals—on the right side of the production rule.

Note An **`LL`** parser is a top-down parser for a subset of context-free languages. It parses the input from Left-to-right, performing **Leftmost derivation** of the **sentence**. An `LL` parser is called a **`LL(k)`** parser when it uses `k` tokens of look-ahead to parse sentences. (See https://en.wikipedia.org/wiki/Recursive_descent_parser on September 15, 2015 for more detail.)

SPL Parsing Functions

The violate-on-pain-of-death rule for designing recursive-decent parser logic is

> **On exit, every parsing function must always ensure `tokens[0]` (the "next token") contains the first token to be considered by the next call to a parsing function.**

As an example of "obedience" to the preceding rule, consider the *SPL1* parsing function *ParsePRINTStatement()* that "recognizes" the *SPL1* **`***SIMPLIFIED***`** `PRINT` statement syntax repeated below for your convenience

```
<PRINTStatement>      ::= PRINT (( <string> | ENDL ))                || ***SIMPLIFIED***
                            { , (( <string> | ENDL )) }* .

//-------------------------------------------------------------
void ParsePRINTStatement(TOKEN tokens[])
//-------------------------------------------------------------
{
   void GetNextToken(TOKEN tokens[]);
```

```
   EnterModule("PRINTStatement");

   do
   {
      GetNextToken(tokens);

      switch ( tokens[0].type )
      {
         case STRING:
            GetNextToken(tokens);
            break;
         case ENDL:
            GetNextToken(tokens);
            break;
         default:
            ProcessCompilerError(tokens[0].sourceLineNumber,
                                 tokens[0].sourceLineIndex,
                                 "Expecting string or ENDL");
      }
   } while ( tokens[0].type == COMMA );

   if ( tokens[0].type != PERIOD )
      ProcessCompilerError(tokens[0].sourceLineNumber,
                           tokens[0].sourceLineIndex,
                           "Expecting '.'");

   GetNextToken(tokens);

   ExitModule("PRINTStatement");
}
```

To obey the inviolate parsing rule, the highlighted reference to *GetNextToken()* must be included to scan the token in the source file that comes after the `'.'` that the *SPL* grammar rule for `<PRINTStatement>` requires at the end of every `PRINT` statement. Note By design, a `'.'` is required at the end of almost every *SPL* executable statement!

Computational and Critical Thinking Questions (25 points)

1. The **conditionally-compiled** global variable **`level`** is used to provide a visual indication of the depth-of-nesting that the call to the parsing function **`module`** is at in a series calls required to parse (recognize) the grammar goal symbol. The tracing output is intended to allow the user to "traverse" the parse tree of the program being compiled. What is the `level` of the goal symbol, that is, what value of the global variable `level` do both "tracing" functions ***EnterModule()*** and ***ExitModule()*** (repeated below for your convenience) use when outputting tracing information after being called from *ParseSPLProgram()*? Hint You can "compute" the value of `level` by finding the initialization of `level` in *main()* and

by studying the logic of the tracing function *EnterModule()*. `level` = 0

```
//-----------------------------------------------
// Global variables
//-----------------------------------------------
READER<CALLBACKSUSED> reader(SOURCELINELENGTH,LOOKAHEAD);
LISTER lister(LINESPERPAGE);

#ifdef TRACEPARSER
int level;
#endif

//-------------------------------------------------------------
void EnterModule(const char module[])
//-------------------------------------------------------------
{
#ifdef TRACEPARSER
   char information[SOURCELINELENGTH+1];

   level++;
   sprintf(information,"   %*s>%s",level*2," ",module);
   lister.ListInformationLine(information);
#endif
}

//-------------------------------------------------------------
void ExitModule(const char module[])
//-------------------------------------------------------------
{
#ifdef TRACEPARSER
   char information[SOURCELINELENGTH+1];

   sprintf(information,"   %*s<%s",level*2," ",module);
   lister.ListInformationLine(information);
   level--;
#endif
}
```

2. (Continuing 1) Which *sprintf()* parameter corresponds to the `*` in `"%*s"` format specifier?
A: `level*2` B: `" "` C: `module` A

3. Briefly explain how to add the reserved word `INPUT` to *SPL1Parser.cpp*.

it would need to be enumerated and entered into the tokentable record as a reserved word. Then it would need to be added as a case under statements

4. Assume that the *.spl* source program file being parsed by *SPL1Parser.cpp* is syntactically valid (that is, contains no context-free syntax errors). When the *ParseSPLProgram()* parsing function is called by *main()* T or F? `tokens[0].type` must be the `TOKENTYPE` literal `PROGRAM` and `tokens[0].lexeme` must contain one of the correct 128 mixed-case spellings of the reserved word `"PROGRAM"`. T

5. T or F? A syntax error diagnosed by *SPL1Parser.cpp* causes an error message to be output to the *.list* file and to the standard output device. T

6. Briefly explain how the effect of the *ProcessCompilerError()* comment `// Use "panic mode" error recovery technique: report error message and terminate compilation!` changes the *SPL1Parser.cpp* flow-of-control.
It aborts the function and flow of control is sent to main to handle the exception

7. The *SPL1* BNF production rule shown below requires that the last token in a syntactically-valid *.spl* source file be the `EOPTOKEN`.

```
<SPLProgram>          ::= <PROGRAMDefinition> EOPC                    || ***SIMPLIFIED***
```
T

T or F? *SPL1Parser.cpp* enforces this rule with the following logic found in *ParseSPLProgram()*

```
if ( tokens[0].type != EOPTOKEN )
   ProcessCompilerError(tokens[0].sourceLineNumber,tokens[0].sourceLineIndex,
                        "Expecting end-of-program");
```

Consider the following parsing function *ParsePROGRAMDefinition()* code for Questions 8 through 11

```
GetNextToken(tokens);

while ( tokens[0].type != END )
   ParseStatement(tokens);

GetNextToken(tokens);
```

8. T or F? The first call to *GetNextToken()* in the parsing function *ParsePROGRAMDefinition()* code segment shown above scans for the token-and-lexeme that comes immediately after the `PROGRAM` reserved word. T

9. T or F? A `<PROGRAMDefinition>` with no statements in its body is legal *SPL1* syntax.

```
<PROGRAMDefinition>   ::= PROGRAM                                     || ***SIMPLIFIED***
                             { <statement> }*
                          END
```
T

10. (Continuing 9) T or F? The pre-test loop `while ( tokens[0].type != END )` in the code segment shown above enforces this syntax rule for `<PROGRAMDefinition>`.
is this why i am getting errors that there is no statement?
T

11. Why is the second call to *GetNextToken()* in the code segment shown above absolutely necessary?!
so that parse spl program is able to check if the next token is a EOPC

Consider the following parsing function *ParseStatement()* code for Question 12

```
switch ( tokens[0].type )
{
   case PRINT:
      ParsePRINTStatement(tokens);
      break;
   default:
      ProcessCompilerError(tokens[0].sourceLineNumber,tokens[0].sourceLineIndex,
                           "Expecting beginning-of-statement");
      break;
}
```

12. Assume that the *.spl* source program file being parsed by *SPL1Parser.cpp* is syntactically-valid (that is, contains no context-free syntax errors). When the parsing function *ParseStatement()* is called by *ParsePROGRAMDefinition()* T or F? `tokens[0].type` must be the `TOKENTYPE` literal `PRINT` and `tokens[0].lexeme` must contain one of the correct 32 mixed-case spellings of the reserved word `"PRINT"`. Hint Consider your answer to Question 11.
F

FYI #1 Imagine that the `<INPUTStatement>` syntax shown below is added to *SPL*. Question How will the code segment shown above have to change?! Answer A `case INPUT` section will be added. More generally, how will a *SPL* statement whose syntax does not begin with a reserved word like `PRINT` or `INPUT` be able to be recognized by the `switch`-statement in the code segment shown above?!

```
<INPUTStatement>      ::= INPUT [ <string> ] <variable> .
```

13. (Continuing ***FYI #1***) T or F? The parsing logic in *ParseINPUTStatement()* must require a loop.
T

14. (Continuing 9) Why does the parsing logic in *ParsePROGRAMDefiniton()* use a `while`-loop but the parsing logic in *ParsePRINTStatement()* use a `do/while`-loop?

```
<PRINTStatement>      ::= PRINT (( <string> | ENDL ))                || ***SIMPLIFIED***
                            { , (( <string> | ENDL )) }* .
```

parse print uses a do while loop to test if there is a comma after the token to print is read while parse program tests beforehand if each token signifies the end of the program

Consider the following parsing function *ParsePRINTStatement()* code for Questions 15 to 18

```
do
{
   GetNextToken(tokens);

   switch ( tokens[0].type )
   {
      case STRING:
         GetNextToken(tokens);
         break;
      case ENDL:
         GetNextToken(tokens);
         break;
      default:
         ProcessCompilerError(tokens[0].sourceLineNumber,tokens[0].sourceLineIndex,
                              "Expecting string or ENDL");
   }
} while ( tokens[0].type == COMMA );

if ( tokens[0].type != PERIOD )
   ProcessCompilerError(tokens[0].sourceLineNumber,tokens[0].sourceLineIndex,
                        "Expecting '.'");

GetNextToken(tokens);
```

15. Complete the following sentence, "The parsing logic changes that must be made to the parsing function *ParsePRINTStatement()* to accommodate the following `<PRINTStatement>` syntax change are…" Hint `<expression>` is a conceptually-big *SPL* non-terminal symbol that "deserves" its own parsing function, *ParseExpression()*. *ParseExpression()* will be introduced in *SPL2Compiler.cpp*.

```
<PRINTStatement>      ::= PRINT   (( <string> | <expression> | ENDL ))
                              { , (( <string> | <expression> | ENDL )) }* .
```

to check if the token type is an expression, and to parse the expression before checking the nest token

16. T or F? The highlighted call to *GetNextToken()* in the code segment shown above is necessary to (1) discard the reserved word `PRINT` token-and-lexeme; or (2) to discard the `COMMA` token-and-lexeme. T

17. The parsing logic of the parsing function *ParsePRINTStatement()* enforces the rule that every `<PRINTStatement>` must end with the `PERIOD` token-and-lexeme. A or D? The `PERIOD` token-and-lexeme isn't absolutely necessary to the *SPL1Parser.cpp* parsing logic because it is not used as the sentinel token-and-lexeme to terminate the `do/while`-loop. In that sense, then, the `PERIOD` token-and-lexeme is an example of **"syntactic sugar"** meaning that it could be eliminated from *SPL* syntax rules.

A

18. Give an example of a legal *SPL1* token-and-lexeme that would cause the "`Expecting string or ENDL`" syntax error to be diagnosed by the following *ParsePRINTStatement()* `switch`-statement `default`-clause. Hint A "legal *SPL1* token-and-lexeme" is any token that the *SPL1Parser.cpp* scanner *GetNextToken()* recognizes, except for `UNKTOKEN`. This "legal *SPL1* token-and-lexeme" causes the syntax error to be diagnosed because it wasn't one of the two acceptable token-and-lexemes, namely, `STRING` or `ENDL`.

```
default:
   ProcessCompilerError(tokens[0].sourceLineNumber,tokens[0].sourceLineIndex,
                        "Expecting string or ENDL");
```

comma

FYI #2 The use of the **metasymbols** `{...}*` in the right-hand-side of grammar rule `<LHS>` typically implies a loop in the portion of the parser that parses (recognizes) `<LHS>` and the use of the metasymbols `[...]` in the right-hand-side of grammar rule `<LHS>` typically implies a selection in the portion of the parser that parses `<LHS>`.

19. (Continuing ***FYI #2***) T or F? The use of the metasymbols `((...))` in the right-hand-side of grammar rule `<LHS>` typically implies a conjunction and a selection in the portion of the parser that parses `<LHS>`.

F

20. Briefly explain why `tokens[]` is a parsing-function formal parameter.

so that the current token in the first element of the array can be analyzed by the parsing function being called

21. (Continuing 20) What type of parsing-function parameter must `tokens[]` be based on the logic of the parsing functions you've studied so far? A: `IN` B: `OUT` C: `IO`

C

22. (Continuing 20) T or F? It is the responsibility of a parsing function to ensure that, upon exit as a post-condition, the token-and-lexeme array element `tokens[0]` contain the first token beyond the syntax that the parsing function is designed to recognize.

T

23. The algorithmic design of a scanner is determined, in large part, by the grammar of the language being recognized and T or F? the algorithmic design of a parser is determined, in large part, by the grammar of the language being recognized.

T

24. When a syntax error is diagnosed during parsing of a *SPL* program, the error is reported, then the parser always terminates. Explain why the recursive descent parsing algorithm that *SPL1Parser.cpp* uses can't just "discard" the tokens remaining in the flawed statement and then resume the parse with first token in the next statement.

It would take an extremely large amount of work to have the CALLEE function discard tokens of the current statement until the next statement, and then send that information to its CALLER before continuing on.

25. C/C++ macro definitions are processed by the pre-processor. A macro reference (that is, the name of the macro) is replaced with its text. For

example, given the following macro definition, each and every occurrence of the macro name `PI` is replaced during pre-processing with its text `3.14159`. This pre-processor replacement of a macro name with its text is called **text substitution**.

```
#define PI 3.14159
```

What is the text for the *SPL1Parser.cpp* macro shown here?

`#define TRACEPARSER` the text for this macro is defined throughout the program with varius #ifdef TRACEPARSER blocks of code

FYI #3 The `12` in the definition of the `description[]` array field in the following `const`-ant `TOKENTABLERECORD` `struct`-ure is the upper-bound of the length of the strings used to initialize each `description[]` array. Question What should you change the `12` to if you needed to add the description `"END_OF_PROGRAM_MODULE"`? Answer 21, `strlen("END_OF_PROGRAM_MODULE").`

```
//-------------------------------------------------------------
struct TOKENTABLERECORD
//-------------------------------------------------------------
{
   TOKENTYPE type;
   char description[12+1];
   bool isReservedWord;
};
```

```
;-------------------------------------------------------------
; Dr. Art Hanna
; Program #1
; P1.spl
;-------------------------------------------------------------
PROGRAM
END

;-------------------------------------------------------------
; Dr. Art Hanna
; Program #2
; P2.spl
/*-------------------------------------------------------------*/

PROGRAM
   PRINT "\"Howdy, world!\"\n".
   PRINT "\"Howdy, world!\"",ENDL.
END

;-------------------------------------------------------------
; Dr. Art Hanna
; Program #3
; P3.spl
;-------------------------------------------------------------

PROGRAM
   PRINT "Howdy".
   PRINT ","," ",world","!",ENDL.
END
```

SPL1 **parser list file for *P2.spl* (with macros `TRACESCANNER` and `TRACEPARSER` `#define`-d)**

```
"P2.spl" Page    1
Line Source Line
---- ------------------------------------------------------------------------------
   1 ;-------------------------------------------------------------
At (   1:  0) begin line comment
   2 ; Dr. Art Hanna
At (   2:  0) begin line comment
   3 ; Program #2
At (   3:  0) begin line comment
   4 ; P2.spl
At (   4:  0) begin line comment
   5 /*-------------------------------------------------------------*/
```

```
At (   5:  0) begin block comment depth = 1
   6
At (   5: 59)   end block comment depth = 1
   7 PROGRAM
   8    PRINT "\"Howdy, world!\"\n".
At (   7:  0) token =      PROGRAM lexeme = |PROGRAM|
At (   8:  3) token =        PRINT lexeme = |PRINT|
   9    PRINT "\"Howdy, world!\"",ENDL.
At (   8:  9) token =       STRING lexeme = |\"Howdy, world!\"\n|
     >SPLProgram
       >PROGRAMDefinition
At (   8: 30) token =       PERIOD lexeme = |.|
         >Statement
           >PRINTStatement
At (   9:  3) token =        PRINT lexeme = |PRINT|
At (   9:  9) token =       STRING lexeme = |\"Howdy, world!\"|
At (   9: 28) token =        COMMA lexeme = |,|
           <PRINTStatement
         <Statement
         >Statement
           >PRINTStatement
  10 END
At (   9: 29) token =         ENDL lexeme = |ENDL|
At (   9: 33) token =       PERIOD lexeme = |.|
  11
At (  10:  0) token =          END lexeme = |END|
At (  11:  0) token =     EOPTOKEN lexeme = ||
At (  11:  0) token =     EOPTOKEN lexeme = ||
           <PRINTStatement
         <Statement
At (  11:  0) token =     EOPTOKEN lexeme = ||
       <PROGRAMDefinition
     <SPLProgram
******* SPL parser ending
```

SPL1 **parser list file for *P2.spl* (with only macro `TRACEPARSER` `#define`-d)**

```
"P2.spl" Page    1
Line Source Line
---- ---------------------------------------------------------------------------
   1 ;-------------------------------------------------------------
   2 ; Dr. Art Hanna
   3 ; Program #2
   4 ; P2.spl
```

```
 83      5 /*-------------------------------------------------------*/
 84      6
 85      7 PROGRAM
 86      8    PRINT "\"Howdy, world!\"\n".
 87      9    PRINT "\"Howdy, world!\"",ENDL.
 88        >SPLProgram
 89          >PROGRAMDefinition
 90            >Statement
 91              >PRINTStatement
 92              <PRINTStatement
 93            <Statement
 94            >Statement
 95              >PRINTStatement
 96     10 END
 97     11
 98              <PRINTStatement
 99            <Statement
100          <PROGRAMDefinition
101        <SPLProgram
102   ******* SPL parser ending
```