Project Details

This is the first part of your project 2. In this part, you will use C++ to implement an LL(1) parser generator and driver, as sketched in Figures 2.24 and 2.19 in the textbook.

Your parser generator should accept as input any LL(1) grammar conforming to the format described below. It should output initialized C++ data structures which, if linked to your driver and an appropriate scanner, will produce a working parser for strings in the language. Your parser, in turn, should accept any string in the language defined by the CFG given to the parser generator. To demonstrate that it works correctly, it should print a trace of its predictions and matches.

If the grammar given to the parser generator is malformed, or not LL(1), the parser generator should print a helpful error message and quit. If the string given to the parser contains syntax errors, the parser should recover gracefully and keep on parsing (more on this below).

To simplify your task, a basic scanner is provided; it accepts a variety of common tokens, with which you can construct a variety of sample grammars.

Grammar Format

Input to your parser generator should consist of

1. A list of token names and numbers, one per line. Token names are strings of printable, non-white-space characters. Token numbers are small non-negative integers, not necessarily contiguous or in any particular order. (This will make it easier for you to use the same scanner for several different grammars. You may assume that all token numbers are less than 128, that the value 0 is not used, and that the value 1 is reserved for end-of-file.)

2. A blank line.

3. A list of productions, one per line. Each production should consist of a symbol name, the meta-symbol "→", and a sequence of zero or more symbol names. I will test your code with only pure BNF: no alternation, no Kleene closure. You may assume that productions with the same left-hand side will be consecutive, and that the start symbol is the left-hand side of the first production.

Example

The simple calculator grammar of Figure 2.16 might be input to your parser generator as follows. The token numbers are taken from the given scanner.

```
tok_eof     1
ident       2
rw_read    13
rw_write   18
lit_int    19
becomes    21
op_add     22
op_sub     23
op_mul     24
op_div     25
lparen     26
rparen     27
```

```
    program    ->  stmt_list tok_eof
    stmt_list  ->  stmt stmt_list
    stmt_list  ->
    stmt       ->  ident becomes expr
    stmt       ->  rw_read ident
    stmt       ->  rw_write expr
    expr       ->  term term_tail
    term_tail  ->  add_op term term_tail
    term_tail  ->
    term       ->  factor fact_tail
    fact_tail  ->  mult_op factor fact_tail
    fact_tail  ->
    factor     ->  lparen expr rparen
    factor     ->  ident
    factor     ->  lit_int
    add_op     ->  op_add
    add_op     ->  op_sub
    mult_op    ->  op_mul
    mult_op    ->  op_div
```

When you run this through your parser generator, it should produce a C++ version of the table in Figure 2.19. In addition, it should produce tables giving the FIRST and FOLLOW sets of every nonterminal, and an indication of which of these can generate epsilon; you'll need these tables for error recovery. The exact format of the tables is up to you. One possible format might look something like the following. It uses row-pointer layout for right-hand sides and FOLLOW sets, and contiguous two-dimensional layout for the main parse table.

```
static const int max_terminal = 27;
static const int num_nonterminals = 10;
static const int num_productions = 19;
char *terminal_names[] = {
    "",
    "tok_eof",      // 1
    "ident",        // 2
    "", "", "", "", "", "", "", "", "", "",
    "rw_read",      // 13
    "", "", "", "",
    "rw_write",     // 18
    "lit_int",      // 19
    "",
    "becomes",      // 21
    "op_add",       // 22
    "op_sub",       // 23
    "op_mul",       // 24
    "op_div",       // 25
    "lparen",       // 26
    "rparen",       // 27
};
char *non_terminal_names[] = {
    "undef",        // 0 -- not used
    "program",      // 1
    "stmt_list",    // 2
    "stmt",         // 3
    "expr",         // 4
    "term_tail",    // 5
    "term",         // 6
```

```
    "fact_tail",    // 7
    "factor",       // 8
    "add_op",       // 9
    "mult_op"       // 10
};

// Right-hand sides, in reverse order.  Negative numbers indicate tokens.
int rhs1[] = {-1, 2, 0};           // eof, stmt_list
int rhs2[] = {2, 3, 0};            // stmt_list, stmt
int rhs3[] = {0};                  // epsilon
int rhs4[] = {4, -21, -2, 0};      // expr, becomes, ident
int rhs5[] = {-2, -13, 0};         // ident, read
int rhs6[] = {4, -18, 0};          // expr, write
int rhs7[] = {5, 6, 0};            // term_tail, term
int rhs8[] = {5, 6, 9, 0};         // term_tail, term, add_op
int rhs9[] = {0};                  // epsilon
int rhs10[] = {7, 8, 0};           // factor_tail, factor
int rhs11[] = {7, 8, 10, 0};       // factor_tail, factor, mult_op
int rhs12[] = {0};                 // epsilon
int rhs13[] = {-27, 4, -26, 0};    // rparen, expr, lparen
int rhs14[] = {-2, 0};             // ident
int rhs15[] = {-19, 0};            // lit_int
int rhs16[] = {-22, 0};            // op_add
int rhs17[] = {-23, 0};            // op_sub
int rhs18[] = {-24, 0};            // op_mul
int rhs19[] = {-25, 0};            // op_div

int* right_hand_sides[] = {0,
    rhs1, rhs2, rhs3, rhs4, rhs5, rhs6, rhs7, rhs8, rhs9, rhs10,
    rhs11, rhs12, rhs13, rhs14, rhs15, rhs16, rhs17, rhs18, rhs19};

int parse_tab[][max_terminal] = {
// See Figure 2.19 in the text, but note that tokens in this example are ordered differently, and have gaps.
// Index table as parse_tab[top-of-stack_nonterminal-1, input_token-1];
   1,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  0,  0,  0,  0,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,
   3,  2,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  2,  0,  0,  0,  0,  2,  0,  0,  0,  0,  0,  0,  0,  0,  0,
   0,  4,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  5,  0,  0,  0,  0,  6,  0,  0,  0,  0,  0,  0,  0,  0,  0,
   0,  7,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  7,  0,  0,  0,  0,  0,  0,  7,  0,
   9,  9,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  9,  0,  0,  0,  0,  9,  0,  0,  0,  8,  8,  0,  0,  0,  9,
   0, 10,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 10,  0,  0,  0,  0,  0,  0, 10,  0,
  12, 12,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 12,  0,  0,  0,  0, 12,  0,  0,  0, 12, 12, 11, 11,  0, 12,
   0, 14,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 15,  0,  0,  0,  0,  0,  0, 13,  0,
   0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 16, 17,  0,  0,  0,  0,
   0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 18, 19,  0,  0
};

bool generates_epsilon[] = {false, true, false, false, true, false, true, false, false, false};

int first1[]  /* program */    = {2, 13, 18, 1, 0};  // ident, rw_read, rw_write, tok_eof
int first2[]  /* stmt_list */  = {2, 13, 18, 0};     // ident, rw_read, rw_write
int first3[]  /* stmt */       = {2, 13, 18, 0};     // ident, rw_read, rw_write
int first4[]  /* expr */       = {26, 2, 19, 0};     // lparen, ident, lit_int
int first5[]  /* term_tail */  = {22, 23, 0};        // op_add, op_sub
int first6[]  /* term */       = {26, 2, 19, 0};     // lparen, ident, lit_int
int first7[]  /* fact_tail */  = {24, 25, 0};        // op_mul, op_div
int first8[]  /* factor */     = {26, 2, 19, 0};     // lparen, ident, lit_int
int first9[]  /* add_op */     = {22, 23, 0};        // op_add, op_sub
int first10[] /* mult_op */    = {24, 25, 0};        // op_mul, op_div

int* first_sets[] = {first1, first2, first3, first4, first5, first6, first7, first8, first9, first10};

int follow1[]  /* program */    = {0};                                  // empty
int follow2[]  /* stmt_list */  = {1, 0};                               // tok_eof
int follow3[]  /* stmt */       = {2, 13, 18, 1, 0};                    // ident, rw_read, rw_write, tok_eof
int follow4[]  /* expr */       = {27, 2, 13, 18, 1, 0};                // rparen, ident, rw_read, rw_write, tok_eof
int follow5[]  /* term_tail */  = {27, 2, 13, 18, 1, 0};                // rparen, ident, rw_read, rw_write, tok_eof
int follow6[]  /* term */       = {22, 23, 27, 2, 13, 18, 1, 0};        // op_add, op_sub, rparen, ident, rw_read, rw_write, tok_
int follow7[]  /* fact_tail */  = {22, 23, 27, 2, 13, 18, 1, 0};        // op_add, op_sub, rparen, ident, rw_read, rw_write, tok_
```

```
int follow8[]  /* factor */     = {22, 23, 24, 25, 27, 2, 13, 18, 1, 0};  // op_add, op_sub, op_mul, op_div, rparen, ident, rw_read
int follow9[]  /* add_op */     = {26, 2, 19, 0};                         // lparen, ident, lit_int
int follow10[] /* mult_op */    = {26, 2, 19, 0};                         // lparen, ident, lit_int

int* follow_sets[] = {follow1, follow2, follow3, follow4, follow5, follow6, follow7, follow8, follow9, follow10};
```

Given the input

```
read A
read B
sum := A + B
write sum
write sum / 2
```

your driver should print the right-hand column of Figure 2.21.

Syntax error recovery

Your parser must implement phrase-level recovery from syntax errors. This should allow it to continue to parse a program (and find more syntax errors) after it encounters an instance of invalid syntax. Specifically,

1. If the input token is tok_error or some other token not used in the given grammar, you should print an error message and consume the token before inspecting the top-of-stack symbol.

2. If you have a terminal at the top of the parse stack and the input token doesn't match it, you should pop the expected token, print an error message, and leave the current input token unconsumed.

3. If you have a nonterminal N at the top of the parse stack for which there is no prediction (zero in the parse table), you should consume input tokens until you find a token T in FIRST(N) or FOLLOW(N). If T is in FOLLOW(N), you should pop N from the stack and continue; otherwise you should continue with N still in place.

Division of labor and writeup

For this project, you have been assigned to a group. This means that you and your group member should sit together with one computer. One of you writes code and the other reviews each line of code as it is typed in. The roles are switched frequently.

The requirement for your README file is same as in Project 1.

Grading scale

The project will be graded based on your source code and the README file:

- Code (70%):
  - Correctness, completeness and efficiency (60%): Your code should implement everything that was required in the assignment. It should produce the right output given normal, expected inputs, and some sort of reasonable response to unexpected inputs. Unless otherwise instructed, and as long as it does not severely compromise programming style, you are expected to use the most efficient data structures and algorithms.

  - Programming style (including internal documentation and program organization) (10%): Your code should have appropriate abstractions, data structures, algorithms, and variable names; declarations for all constants; and a judicious number of helpful comments. You should think of programming as explaining to the readers of your programs what you want the computer to do.

- README file (30%)
  - Completeness (20%): Your write-up should include all the items discussed in the README files.
  - Readability (10%): Your write-up should be well organized, clear, and concisely presented.

C++ Resources

- `accu.org/index.php/weblinks/c48/`
  Resources page of the Association of C and C++ Users, based in Great Britain. Incredibly rich set of links to resources of every kind.
- `www.sgi.com/tech/stl/`
  Official on-line reference for the C++ Standard Template Library (STL).
- `www.research.att.com/~bs/C++.html` Bjarne Stroustrup's C++ home page.

Try use the STL. Dont write your own code for lists, sets, mappings, etc.!

What to turn in:

Zip your source files and README files (one for each solution of these four languages) and upload your zip file to Canvas under category Project2-part1 by the deadline.