Project 2
Implementation of an LALR(1) Parser
Due Friday, December 11, 2020

1. Problem

In this assignment you are requested to use the tool Bison to write an LALR(1) parser for the simple language Lotus. The grammar for Lotus is given as follows:

program → **Identifier ()** *function_body*
function_body → **{** *variable_declarations statements* **}**
variable_declarations → **empty** | *variable_declarations variable_declaration*
variable_declaration → **int Identifier ;**
statements → **empty** | *statements statement*
statement → *assignment_statement* | *compound_statement*
| *if_statement* | *while_statement* | *exit_statement*
| *read_statement* | *write_statement*
assignment_statement → **Identifier =** *arith_expression* **;**
compound_statement → **{** *statements* **}**
if_statement → **if (** *bool_expression* **)** *statement*
| **if (** *bool_expression* **)** *statement* **else** *statement*
while_statement → **while (** *bool_expression* **)** *statement*
exit_statement → **exit ;**
read_statement → **read Identifier ;**
write_statement → **write** *arith_expression* **;**
bool_expression → *bool_term* | *bool_expression* **||** *bool_term*
bool_term → *bool_factor* | *bool_term* **&&** *bool_factor*
bool_factor → *bool_primary* | **!** *bool_primary*
bool_primary → *arith_expression* **==** *arith_expression*
| *arith_expression* **!=** *arith_expression*
| *arith_expression* **>** *arith_expression*
| *arith_expression* **>=** *arith_expression*
| *arith_expression* **<** *arith_expression*
| *arith_expression* **<=** *arith_expression*
arith_expression → *arith_term* | *arith_expression* **+** *arith_term*
| *arith_expression* **-** *arith_term*
arith_term → *arith_factor*
| *arith_term* ***** *arith_factor*
| *arith_term* **/** *arith_factor*
| *arith_term* **%** *arith_factor*

arith_factor → *arith_primary* | **-** *arith_primary*

arith_primary → **Integer** | **Identifier** | **(** *arith_expression* **)**

The parser needs to parse a program written in Lotus defined above. The parser should be able to trace the parsing process by using the option "-p" to print each production reduced by the parser. No syntax error recovery is needed. Each syntax error message should include the line number where the error is detected. The format of the error message is as follows:

Syntax error: line xx

The parser reads input from stdin, writes output to stdout, and writes errors to stderr.

Consider the following Lotus program:

```
// A program to sum 1 to n
sum( )
{
   int  n;
   int  s;

   read n;
   if (n < 0) {
      write -1;
      exit;
   } else {
      s = 0;
      while (n > 0) {
         s = s + n;
         n = n - 1;
      }
   }
   write s;
}
```

The output for using the option –p is as follows:

```
variable_declarations -> empty
variable_declaration -> int Identifier ;
variable_declarations -> variable_declarations variable_declaration
variable_declaration -> int Identifier ;
variable_declarations -> variable_declarations variable_declaration
```

statements -> empty
read_statement -> read Identifier ;
statement -> read_statement
statements -> statements statement
arith_primary -> Identifier
arith_factor -> arith_primary
arith_term -> arith_factor
arith_expression -> arith_term
arith_primary -> Integer
arith_factor -> arith_primary
arith_term -> arith_factor
arith_expression -> arith_term
bool_primary -> arith_expression < arith_expression
bool_factor -> bool_primary
bool_term -> bool_factor
bool_expression -> bool_term
statements -> empty
arith_primary -> Integer
arith_factor -> - arith_primary
arith_term -> arith_factor
arith_expression -> arith_term
write_statement -> write arith_expression ;
statement -> write_statement
statements -> statements statement
exit_statement -> exit ;
statement -> exit_statement
statements -> statements statement
compound_statement -> { statements }
statement -> compound_statement
statements -> empty
arith_primary -> Integer
arith_factor -> arith_primary
arith_term -> arith_factor
arith_expression -> arith_term
assignment_statement -> Identifier = arith_expression ;
statement -> assignment_statement
statements -> statements statement
arith_primary -> Identifier
arith_factor -> arith_primary
arith_term -> arith_factor

arith_expression -> arith_term
arith_primary -> Integer
arith_factor -> arith_primary
arith_term -> arith_factor
arith_expression -> arith_term
bool_primary -> arith_expression > arith_expression
bool_factor -> bool_primary
bool_term -> bool_factor
bool_expression -> bool_term
statements -> empty
arith_primary -> Identifier
arith_factor -> arith_primary
arith_term -> arith_factor
arith_expression -> arith_term
arith_primary -> Identifier
arith_factor -> arith_primary
arith_term -> arith_factor
arith_expression -> arith_expression + arith_term
assignment_statement -> Identifier = arith_expression ;
statement -> assignment_statement
statements -> statements statement
arith_primary -> Identifier
arith_factor -> arith_primary
arith_term -> arith_factor
arith_expression -> arith_term
arith_primary -> Integer
arith_factor -> arith_primary
arith_term -> arith_factor
arith_expression -> arith_expression - arith_term
assignment_statement -> Identifier = arith_expression ;
statement -> assignment_statement
statements -> statements statement
compound_statement -> { statements }
statement -> compound_statement
while_statement -> while (bool_expression) statement
statement -> while_statement
statements -> statements statement
compound_statement -> { statements }
statement -> compound_statement
if_statement -> if (bool_expression) statement else statement

statement -> if_statement
statements -> statements statement
arith_primary -> Identifier
arith_factor -> arith_primary
arith_term -> arith_factor
arith_expression -> arith_term
write_statement -> write arith_expression ;
statement -> write_statement
statements -> statements statement
function_body -> { variable_declarations statements }
program -> Identifier ( ) function_body

2. Handing in your program

You should provide a make file named “makefile” to build this assignment. See online manual make(1) for more information. The executable file should be named “parser” namely,

```
gcc -o parser $(OBJ) -lfl
```

To turn in the assignment, upload a compressed file containing makefile, *.l, *.y *.h, and *.c to eCourse2 site.

3. Grading

The grading is based on the correctness of your program and the programming style of your program. The correctness will be tested by a number of test cases.

To facilitate the grading of teaching assistants, you should test your program on the machine csie1.

It is best to incrementally build your program so that you always have a partially-correct working program. It is also best to construct a shell script to systematically test your program.