The objective of this MP is to develop a recursive descent parser for a subset of C. The grammar is attached. The subset includes functions, assignments, loops, conditional, switch, if, and goto statements. Possible types (constant and variables) are integer, float and string. Float constants could be written in scientific notation (e.g. 3.45e+2) or be just a sequence of digits with a decimal point. String constants will be a sequence of characters enclosed in double quotes ("). The escape sequences for new line and double quote of C are accepted. There are scalar variables and arrays, but there are no pointer variables. Also, there is no char data type and strings are not arrays of chars. Expressions could involve arithmetic operations, comparisons and logical operations. The semantics of expressions, including implicit arithmetic conversions are the same of C.

The lexical scanner could be done using a scanner generator or by hand. I recommend that it be done by hand, but no points will be deducted if the lexical scanner is done using an automatic generator.

The parser could be written in any language that you desire, but if you use an esoteric language, you may have to train the grader. Also, you should take into account that the code generation part (to be assigned on September 28) should be built on top of this parser, unless you choose to use a parser written in java that we will provide.

The output of the parser should be the sequence of productions that form the derivation sequence and an abstract syntax tree. Please see your TA or instructor if you don’t know how to design an abstract syntax tree.
Simplified C Grammar

function_list ::= 
    function_definition function_list 
    | function_definition

function_definition ::= 
    type_specifier IDENTIFIER ( parameter_list ) 
    compound_statement

compound_statement ::= 
    { declaration_statement_list  statement_list } 
    | { statement_list } 
    | { declaration_statement_list } 
    | { }

declaration_statement_list ::= 
    declaration_statement declaration_statement_list 
    | declaration_statement

declaration_statement ::= 
    real_type declarator_list ;

declarator_list ::= 
    declarator 
    | declarator , declarator_list

parameter_list ::= 
    parameter , parameter_list 
    | parameter 
    | /* empty */
parameter ::= 
    real_type declarator

type_specifier ::= 
    real_type
    | VOID

real_type ::= 
    INT
    | FLOAT
    | STRING

declarator ::= 
    | declarator [ CONSTANT ]
    | IDENTIFIER

statement_list ::= 
    statement statement_list
    | statement

statement ::= 
    compound_statement
    | assignment_statement
    | if_statement
    | switch_statement
    | while_loop
    | do_while_loop
    | for_loop
    | case_statement
    | default_statement
    | jump_statement

assignment_statement ::= 
    variable = expression ;

if_statement ::= 
    IF ( expression ) statement
switch_statement ::= 
    SWITCH ( expression ) statement

while_loop ::= 
    WHILE ( expression ) statement

do_while_loop ::= 
    DO statement WHILE ( expression ) ;

for_loop ::= 
    FOR ( assignment_statement ; expression ; assignment_statement ) statement

case_statement ::= 
    CASE expression : statement

default_statement ::= 
    DEFAULT : statement

jump_statement ::= 
    CONTINUE ;
    | BREAK ;
    | RETURN ;
    | RETURN expression ;

expression ::= 
    and_expression 
    | expression || and_expression

expression_list ::= 
    expression , expression_list 
    | expression

and_expression ::= 
    equality_expression 
    | and_expression && equality_expression
equality_expression ::= 
    inequality_expression 
    | equality_expression == inequality_expression 
    | equality_expression != inequality_expression 

inequality_expression ::= 
    additive_expression 
    | inequality_expression < additive_expression 
    | inequality_expression > additive_expression 
    | inequality_expression <= additive_expression 
    | inequality_expression >= additive_expression 

additive_expression ::= 
    multiplicative_expression 
    | additive_expression + multiplicative_expression 
    | additive_expression - multiplicative_expression 

multiplicative_expression ::= 
    unary_expression 
    | multiplicative_expression * unary_expression 
    | multiplicative_expression / unary_expression 

unary_expression ::= 
    factor 
    | ! factor 
    | - factor 

factor ::= 
    variable 
    | IDENTIFIER ( expression_list ) 
    | CONSTANT 
    | ( expression ) 

variable ::= 
    IDENTIFIER 
    | variable [ expression ]