DEPARTMENT OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY
Spring 2024

Theory Of Programming Language
(CT-367)

Batch 2022
Section B

Hafsa Imtiaz (CT-22060)
Zainab Furqan Ahmed (CT-22067)
Sheeza Aslam (CT-22064)

DEPARTMENT OF COMPUTER SCIENCE & INFORMATION TECHNOLOGY
BACHELORS OF SCIENCE IN COMPUTER SCIENCE

Complex Computing Problem Assessment Rubrics

Course Code: CT-367			**Course Title: Theory of Programming Language**
Criteria and Scales			
Excellent (3)	**Good (2)**	**Average (1)**	**Poor (0)**
Criterion 1: Understanding the Problem: How well the problem statement is understood by the student			
Understands the problem clearly and clearly identifies the underlying issues and functionalities.	Adequately understands the problem and identifies the underlying issues and functionalities.	Inadequately defines the problem and identifies the underlying issues and functionalities.	Fails to define the problem adequately and does not identify the underlying issues and functionalities.
Criterion 2: Research: The amount of research that is used in solving the problem			
Contains all the information needed for solving the problem	Good research, leading to a successful solution	Mediocre research which may or may not lead to an adequate solution	No apparent research
Criterion 3: Code: How complete the code is along with the assumptions and selected functionalities			
Complete Code according to the according to the selected functionalities of the given case with clear assumptions	Incomplete Code according to the selected functionalities of the given case with clear assumptions	Incomplete Code according to the selected functionalities of the given case with unclear assumptions	Wrong code and naming conventions
Criterion 4: Report: How thorough and well organized is the solution			
All the necessary information clearly organized for easy use in solving the problem	Good information organized well that could lead to a good solution	Mediocre information which may or may not lead to a solution	No report provided

Total Marks:_____

Teacher's Signature:_____

Lexical and Syntax Analyzer for C Language

Description of Selected Language

The C programming language is a procedural programming language developed in the early 1970s by Dennis Ritchie at Bell Labs. It is a structured, statically typed, middle-level language known for its efficiency, portability, and power. C provides low-level access to memory while maintaining high-level language features, making it suitable for both system programming and application development. The language is characterized by its minimal keywords, rich set of operators, and a core of essential functions that can be extended through libraries. C has significantly influenced many modern programming languages and remains widely used for operating systems, embedded systems, and performance-critical applications.

Selected Functionalities

1. **Lexical Analysis**
    - Tokenization of keywords, identifiers, constants, operators, and punctuators
    - Recognition of comments (both single-line and multi-line)
    - Handling of preprocessor directives
    - Line number tracking for error reporting
    - String and character literal processing
2. **Syntax Analysis**
    - Parsing of declarations and definitions
    - Handling of expressions and statements
    - Recognition of control structures (if-else, loops, switch)
    - Support for function definitions and calls
    - Parsing of complex data types (structs, enums)
    - Error reporting with line numbers

Regular Expressions/Rules

Keywords

- Rule: "auto"|"break"|"case"|"char"|"const"|"continue"|...
- Action: Return the corresponding token type

Identifiers

- Rule: {LETTER}({LETTER}|{DIGIT})* where {LETTER} is [a-zA-Z_] and {DIGIT} is [0-9]
- Action: Store the identifier name and return the IDENTIFIER token

Constants

- Rule for integers: {DIGIT}+
- Rule for floating-point: {DIGIT}+(\.{DIGIT}+)?([eE][+-]?{DIGIT}+)?
- Rule for character constants: \'([^\'\n]|\\.)*\'
- Action: Store value and return CONSTANT token

String Literals

- Rule: \"([^\"\n]|\\\")*\"
- Action: Store string and return STRING_LITERAL token

Operators

- Rule for arithmetic operators: "+"|"-"|"*"|"/"|"%"
- Rule for relational operators: "<"|">"|"<="|">="|"=="|"!="
- Rule for logical operators: "&&"|"||"|"!"
- Rule for assignment operators: "="|"+="|"-="|"*="|"/="|"%="|...
- Rule for increment/decrement: "++"|"--"
- Action: Return corresponding token type

Comments

- Rule for multi-line comments: "/*" starts a comment state, "*/" ends it
- Rule for single-line comments: "//" starts a line comment state, newline ends it
- Action: Ignore comment text but track line numbers

Preprocessor Directives

- Rule: ^"#" at start of line enters preprocessor state, newline exits
- Action: Report preprocessor directive and ignore content

Whitespace

- Rule: [\t\r\f\v]+
- Action: Ignore

Line Counting

- Rule: \n
- Action: Increment line counter

Source Code:

c_lexer.l (Lexical Analyzer)

```
%{
  #include <stdio.h>
  #include <stdlib.h>
  #include <string.h>
  #include "c_parser.tab.h" // This will be generated by Bison

  // For tracking line numbers
  int line_num = 1;
  %}

  /* States */
  %x COMMENT
  %x LINE_COMMENT
  %x PREPROCESSOR

  /* Regular expressions for various token patterns */
  DIGIT      [0-9]
  LETTER     [a-zA-Z_]
  ID         {LETTER}({LETTER}|{DIGIT})*
  NUMBER     {DIGIT}+(\.{DIGIT}+)?([eE][+-]?{DIGIT}+)?
  WHITESPACE  [ \t\r\f\v]+
  STRING     \"([^\"\n]|\\\")*\"
  CHAR       \'([^\'\n]|\\.)*\'

  %%

  "/*"               { BEGIN(COMMENT); }
  <COMMENT>"*/"          { BEGIN(INITIAL); }
  <COMMENT>\n            { line_num++; }
  <COMMENT>.             { /* ignore other characters in    comment */ }

  "//"               { BEGIN(LINE_COMMENT); }
  <LINE_COMMENT>\n       { line_num++; BEGIN(INITIAL); }
  <LINE_COMMENT>.        { /* ignore other characters in line comment */ }

  ^"#"               { BEGIN(PREPROCESSOR); printf("Preprocessor directive at line %d\n",
line_num); }
```

```
<PREPROCESSOR>\n        { line_num++; BEGIN(INITIAL); }
<PREPROCESSOR>.         { /* ignore characters in preprocessor directive */ }

\n                      { line_num++; }
{WHITESPACE}            { /* ignore whitespace */ }

"auto"      { return AUTO; }
"break"     { return BREAK; }
"case"      { return CASE; }
"char"      { return CHAR; }
"const"     { return CONST; }
"continue"  { return CONTINUE; }
"default"   { return DEFAULT; }
"do"        { return DO; }
"double"    { return DOUBLE; }
"else"      { return ELSE; }
"enum"      { return ENUM; }
"extern"    { return EXTERN; }
"float"     { return FLOAT; }
"for"       { return FOR; }
"goto"      { return GOTO; }
"if"        { return IF; }
"int"       { return INT; }
"long"      { return LONG; }
"register"  { return REGISTER; }
"return"    { return RETURN; }
"short"     { return SHORT; }
"signed"    { return SIGNED; }
"sizeof"    { return SIZEOF; }
"static"    { return STATIC; }
"struct"    { return STRUCT; }
"switch"    { return SWITCH; }
"typedef"   { return TYPEDEF; }
"union"     { return UNION; }
"unsigned"  { return UNSIGNED; }
"void"      { return VOID; }
"volatile"  { return VOLATILE; }
"while"     { return WHILE; }

"+"         { return '+'; }
```

```
"-"         { return '-'; }
"*"         { return '*'; }
"/"         { return '/'; }
"%"         { return '%'; }
"="         { return '='; }
"<"         { return '<'; }
">"         { return '>'; }
"!"         { return '!'; }
"&"         { return '&'; }
"|"         { return '|'; }
"^"         { return '^'; }
"~"         { return '~'; }
"?"         { return '?'; }
":"         { return ':'; }
";"         { return ';'; }
","         { return ','; }
"."         { return '.'; }
"("         { return '('; }
")"         { return ')'; }
"["         { return '['; }
"]"         { return ']'; }
"{"         { return '{'; }
"}"         { return '}'; }

"++"        { return INC_OP; }
"--"        { return DEC_OP; }
"=="        { return EQ_OP; }
"!="        { return NE_OP; }
"<="        { return LE_OP; }
">="        { return GE_OP; }
"&&"        { return AND_OP; }
"||"        { return OR_OP; }
"<<"        { return LEFT_OP; }
">>"        { return RIGHT_OP; }
"->"        { return PTR_OP; }
"+="        { return ADD_ASSIGN; }
"-="        { return SUB_ASSIGN; }
"*="        { return MUL_ASSIGN; }
"/="        { return DIV_ASSIGN; }
"%="        { return MOD_ASSIGN; }
```

```
"&="        { return AND_ASSIGN; }
"^="        { return XOR_ASSIGN; }
"|="        { return OR_ASSIGN; }
"<<="       { return LEFT_ASSIGN; }
">>="       { return RIGHT_ASSIGN; }

{ID}        {
            yylval.str_val = strdup(yytext);
            return IDENTIFIER;
            }

{NUMBER}    {
            yylval.num_val = atof(yytext);
            return CONSTANT;
            }

{STRING}    {
            yylval.str_val = strdup(yytext);
            return STRING_LITERAL;
            }

{CHAR}      {
            yylval.str_val = strdup(yytext);
            return CONSTANT;
            }

.           { printf("Unrecognized character: %s at line %d\n", yytext, line_num); }
%%

int yywrap() {
    return 1;
}
```

c_parser.y (Syntax Analyzer)

```
%{
  #include <stdio.h>
  #include <stdlib.h>
  #include <string.h>

  extern int yylex();
  extern int line_num;
  extern char* yytext;
  extern FILE* yyin;

  void yyerror(const char* s);

  // Uncomment for debugging
  // #define YYDEBUG 1
  %}

  %union {
     int int_val;
     double num_val;
     char* str_val;
  }

  /* Tokens from the lexical analyzer */
  %token AUTO BREAK CASE CHAR CONST CONTINUE DEFAULT DO DOUBLE
  %token ELSE ENUM EXTERN FLOAT FOR GOTO IF INT LONG
  %token REGISTER RETURN SHORT SIGNED SIZEOF STATIC STRUCT SWITCH
  %token TYPEDEF UNION UNSIGNED VOID VOLATILE WHILE

  %token <str_val> IDENTIFIER
  %token <num_val> CONSTANT
  %token <str_val> STRING_LITERAL

  %token INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
  %token AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
  %token SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
  %token XOR_ASSIGN OR_ASSIGN PTR_OP

  %start translation_unit
```

```
/* Resolve dangling else conflict */
%nonassoc IFX
%nonassoc ELSE

%%

primary_expression
    : IDENTIFIER        { printf("Identifier: %s\n", $1); free($1); }
    | CONSTANT          { printf("Constant: %f\n", $1); }
    | STRING_LITERAL    { printf("String: %s\n", $1); free($1); }
    | '(' expression ')'
    ;

postfix_expression
    : primary_expression
    | postfix_expression '[' expression ']'
    | postfix_expression '(' ')'
    | postfix_expression '(' argument_expression_list ')'
    | postfix_expression '.' IDENTIFIER
    | postfix_expression PTR_OP IDENTIFIER
    | postfix_expression INC_OP
    | postfix_expression DEC_OP
    ;

argument_expression_list
    : assignment_expression
    | argument_expression_list ',' assignment_expression
    ;

unary_expression
    : postfix_expression
    | INC_OP unary_expression
    | DEC_OP unary_expression
    | unary_operator cast_expression
    | SIZEOF unary_expression
    | SIZEOF '(' type_name ')'
    ;

unary_operator
    : '&'
```

```
    | '*'
    | '+'
    | '-'
    | '~'
    | '!'
    ;

cast_expression
    : unary_expression
    | '(' type_name ')' cast_expression
    ;

multiplicative_expression
    : cast_expression
    | multiplicative_expression '*' cast_expression
    | multiplicative_expression '/' cast_expression
    | multiplicative_expression '%' cast_expression
    ;

additive_expression
    : multiplicative_expression
    | additive_expression '+' multiplicative_expression
    | additive_expression '-' multiplicative_expression
    ;

shift_expression
    : additive_expression
    | shift_expression LEFT_OP additive_expression
    | shift_expression RIGHT_OP additive_expression
    ;

relational_expression
    : shift_expression
    | relational_expression '<' shift_expression
    | relational_expression '>' shift_expression
    | relational_expression LE_OP shift_expression
    | relational_expression GE_OP shift_expression
    ;

equality_expression
```

```
    : relational_expression
    | equality_expression EQ_OP relational_expression
    | equality_expression NE_OP relational_expression
    ;

and_expression
    : equality_expression
    | and_expression '&' equality_expression
    ;

exclusive_or_expression
    : and_expression
    | exclusive_or_expression '^' and_expression
    ;

inclusive_or_expression
    : exclusive_or_expression
    | inclusive_or_expression '|' exclusive_or_expression
    ;

logical_and_expression
    : inclusive_or_expression
    | logical_and_expression AND_OP inclusive_or_expression
    ;

logical_or_expression
    : logical_and_expression
    | logical_or_expression OR_OP logical_and_expression
    ;

conditional_expression
    : logical_or_expression
    | logical_or_expression '?' expression ':' conditional_expression
    ;

assignment_expression
    : conditional_expression
    | unary_expression assignment_operator assignment_expression
    ;
```

```
assignment_operator
    : '='
    | MUL_ASSIGN
    | DIV_ASSIGN
    | MOD_ASSIGN
    | ADD_ASSIGN
    | SUB_ASSIGN
    | LEFT_ASSIGN
    | RIGHT_ASSIGN
    | AND_ASSIGN
    | XOR_ASSIGN
    | OR_ASSIGN
    ;

expression
    : assignment_expression
    | expression ',' assignment_expression
    ;

constant_expression
    : conditional_expression
    ;

declaration
    : declaration_specifiers ';'
    | declaration_specifiers init_declarator_list ';'
    ;

declaration_specifiers
    : storage_class_specifier
    | storage_class_specifier declaration_specifiers
    | type_specifier
    | type_specifier declaration_specifiers
    | type_qualifier
    | type_qualifier declaration_specifiers
    ;

init_declarator_list
    : init_declarator
    | init_declarator_list ',' init_declarator
```

```
    ;

init_declarator
    : declarator
    | declarator '=' initializer
    ;

storage_class_specifier
    : AUTO
    | REGISTER
    | STATIC
    | EXTERN
    | TYPEDEF
    ;

type_specifier
    : VOID
    | CHAR
    | SHORT
    | INT
    | LONG
    | FLOAT
    | DOUBLE
    | SIGNED
    | UNSIGNED
    | struct_or_union_specifier
    | enum_specifier
    ;

struct_or_union_specifier
    : struct_or_union IDENTIFIER '{' struct_declaration_list '}'
    | struct_or_union '{' struct_declaration_list '}'
    | struct_or_union IDENTIFIER
    ;

struct_or_union
    : STRUCT
    | UNION
    ;
```

```
struct_declaration_list
    : struct_declaration
    | struct_declaration_list struct_declaration
    ;

struct_declaration
    : specifier_qualifier_list struct_declarator_list ';'
    ;

specifier_qualifier_list
    : type_specifier specifier_qualifier_list
    | type_specifier
    | type_qualifier specifier_qualifier_list
    | type_qualifier
    ;

struct_declarator_list
    : struct_declarator
    | struct_declarator_list ',' struct_declarator
    ;

struct_declarator
    : declarator
    | ':' constant_expression
    | declarator ':' constant_expression
    ;

enum_specifier
    : ENUM '{' enumerator_list '}'
    | ENUM IDENTIFIER '{' enumerator_list '}'
    | ENUM IDENTIFIER
    ;

enumerator_list
    : enumerator
    | enumerator_list ',' enumerator
    ;

enumerator
    : IDENTIFIER
```

```
    | IDENTIFIER '=' constant_expression
    ;

type_qualifier
    : CONST
    | VOLATILE
    ;

declarator
    : pointer direct_declarator
    | direct_declarator
    ;

direct_declarator
    : IDENTIFIER
    | '(' declarator ')'
    | direct_declarator '[' constant_expression ']'
    | direct_declarator '[' ']'
    | direct_declarator '(' parameter_type_list ')'
    | direct_declarator '(' identifier_list ')'
    | direct_declarator '(' ')'
    ;

pointer
    : '*'
    | '*' type_qualifier_list
    | '*' pointer
    | '*' type_qualifier_list pointer
    ;

type_qualifier_list
    : type_qualifier
    | type_qualifier_list type_qualifier
    ;

parameter_type_list
    : parameter_list
    | parameter_list ',' '.'
    ;
```

```
parameter_list
    : parameter_declaration
    | parameter_list ',' parameter_declaration
    ;

parameter_declaration
    : declaration_specifiers declarator
    | declaration_specifiers abstract_declarator
    | declaration_specifiers
    ;

identifier_list
    : IDENTIFIER
    | identifier_list ',' IDENTIFIER
    ;

type_name
    : specifier_qualifier_list
    | specifier_qualifier_list abstract_declarator
    ;

abstract_declarator
    : pointer
    | direct_abstract_declarator
    | pointer direct_abstract_declarator
    ;

direct_abstract_declarator
    : '(' abstract_declarator ')'
    | '[' ']'
    | '[' constant_expression ']'
    | direct_abstract_declarator '[' ']'
    | direct_abstract_declarator '[' constant_expression ']'
    | '(' ')'
    | '(' parameter_type_list ')'
    | direct_abstract_declarator '(' ')'
    | direct_abstract_declarator '(' parameter_type_list ')'
    ;

initializer
```

```
    : assignment_expression
    | '{' initializer_list '}'
    | '{' initializer_list ',' '}'
    ;

initializer_list
    : initializer
    | initializer_list ',' initializer
    ;

statement
    : labeled_statement
    | compound_statement
    | expression_statement
    | selection_statement
    | iteration_statement
    | jump_statement
    ;

labeled_statement
    : IDENTIFIER ':' statement
    | CASE constant_expression ':' statement
    | DEFAULT ':' statement
    ;

compound_statement
    : '{' '}'
    | '{' statement_list '}'
    | '{' declaration_list '}'
    | '{' declaration_list statement_list '}'
    ;

declaration_list
    : declaration
    | declaration_list declaration
    ;

statement_list
    : statement
    | statement_list statement
```

```
    ;

expression_statement
    : ';'
    | expression ';'
    ;

selection_statement
    : IF '(' expression ')' statement %prec IFX
    | IF '(' expression ')' statement ELSE statement
    | SWITCH '(' expression ')' statement
    ;

iteration_statement
    : WHILE '(' expression ')' statement
    | DO statement WHILE '(' expression ')' ';'
    | FOR '(' expression_statement expression_statement ')' statement
    | FOR '(' expression_statement expression_statement expression ')' statement
    ;

jump_statement
    : GOTO IDENTIFIER ';'
    | CONTINUE ';'
    | BREAK ';'
    | RETURN ';'
    | RETURN expression ';'
    ;

translation_unit
    : external_declaration
    | translation_unit external_declaration
    ;

external_declaration
    : function_definition
    | declaration
    ;

function_definition
    : declaration_specifiers declarator declaration_list compound_statement
```

```
    | declaration_specifiers declarator compound_statement
    | declarator declaration_list compound_statement
    | declarator compound_statement
    ;

%%

void yyerror(const char* s) {
    fprintf(stderr, "Error: %s at line %d\n", s, line_num);
}
int main(int argc, char* argv[]) {
    // Uncomment for debugging
    // yydebug = 1;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <input_file>\n", argv[0]);
        return 1;
    }

    FILE* input_file = fopen(argv[1], "r");
    if (!input_file) {
        fprintf(stderr, "Cannot open file %s\n", argv[1]);
        return 1;
    }
    yyin = input_file;
    printf("Parsing file: %s\n", argv[1]);

    int parse_result = yyparse();

    fclose(input_file);

    if (parse_result == 0) {
        printf("Parsing completed successfully.\n");
        return 0;
    } else {
        printf("Parsing failed.\n");
        return 1;
    }
}
```

Test File (test.c)

```c
/* This is a test C file for our lexical and syntax analyzer */
#include <stdio.h>
// Function to calculate factorial
int factorial(int n) {
    if (n <= 1) {
        return 1;
    }
    return n * factorial(n - 1);
}
// Main function
int main() {
    int num = 5;
    int result = 0;
    int a = 10;
    int b = 5;
    int i=0;

    // Calculate factorial
    result = factorial(num);

    // Print result
    printf("Factorial of %d is %d\n", num, result);
    printf("a + b = %d\n", a + b);
    printf("a - b = %d\n", a - b);
    printf("a * b = %d\n", a * b);
    printf("a / b = %d\n", a / b);
    // Test control structures
    for (i = 0; i < 5; i++) {
        if (i % 2 == 0) {
            printf("%d is even\n", i);
        } else {
            printf("%d is odd\n", i);
        }
    }
    return 0;
}
```

Output:

```
C:\Users\hafsa\Desktop\TPL\claud ai>c_compiler.exe test.c
Parsing file: test.c
Preprocessor directive at line 3
Identifier: n
Constant: 1.000000
Constant: 1.000000
Identifier: n
Identifier: factorial
Identifier: n
Constant: 1.000000
Constant: 5.000000
Constant: 0.000000
Constant: 10.000000
Constant: 5.000000
Constant: 0.000000
Identifier: result
Identifier: factorial
Identifier: num
Identifier: printf
String: "Factorial of %d is %d\n"
Identifier: num
Identifier: result
Identifier: printf
String: "a + b = %d\n"
Identifier: a
Identifier: b
Identifier: printf
String: "a - b = %d\n"
Identifier: a
Identifier: b
Identifier: printf
String: "a * b = %d\n"
Identifier: a
Identifier: b
Identifier: printf
String: "a / b = %d\n"
Identifier: a
Identifier: b
Identifier: i
Constant: 0.000000
Identifier: i
Constant: 5.000000
Identifier: i
Identifier: i
Constant: 2.000000
Constant: 0.000000
Identifier: printf
String: "%d is even\n"
Identifier: i
Identifier: printf
String: "%d is odd\n"
Identifier: i
Constant: 0.000000
Parsing completed successfully.
```

Limitations

1. **Variable Declaration Restrictions**:

   All variable declarations must be at the start of a code block. Mid-block declarations are not supported by the analyzer.

2. **Platform-Specific Issues**:

   When compiling on Windows systems, there may be warnings related to library handling since Windows handles these libraries differently than Unix-based systems. The warnings about unused functions can be ignored - they're normal when using Flex.

3. **Multiple Variable Declarations**:

   The current implementation has difficulty parsing multiple variable declarations in a single statement (e.g., int a = 10, b = 5;).

4. **Advanced C Features**:

   The analyzer doesn't support some C99/C11 features like:

   - Variable-length arrays
   - Inline functions
   - Compound literals
   - Designated initializers
   - _Bool, _Complex, and _Imaginary types

5. **Error Recovery**:

   The parser stops at the first syntax error without attempting to recover and continue parsing.

6. **Library Functions**: Standard library functions are treated as any other function without special handling.

Future Improvements

1. Implement a proper symbol table for tracking identifiers
2. Add semantic analysis for type checking
3. Improve error recovery mechanisms
4. Support for C99/C11 language features
5. Handle complex declarations and initializations more robustly