Principle of Compiler Construction

(COCSC14)

Lab File

Submitted by:
Ashish Kumar
2019UCO1518

To:
Ms. Rahila Khan

Index:

1. Develop a lexical analyser for “C” using LEX tool.

input.c

```
#include <stdio.h>
void main(int a)
{
    /* HELLO This is a comment*/
    int a, b, c;
    a = 1;
    b = 2;
    if (a > b)
        c = 0;
    else
        c = -1;

    printf("The value of c: %d", c);
    for (int i = 0; i < 5; i++)
        i++;
    return 0;
}

```

c_lex_analyser.l

```
%{
int COMMENT=0;
%}

identifier [a-zA-Z][a-zA-Z0-9]*

%%
#.*\n {printf("%sThis is a PREPROCESSOR DIRECTIVE\n",yytext);}

auto|break|case|char|const|continue|default|do|double|else|enum|extern|float|for|goto|if|int|long|register|return|short|signed|sizeof|static|struct|switch|typedef|union|unsigned|void|volatile|while {printf("\n%s is a KEYWORD",yytext);}

"/*" {COMMENT = 1;}
"*/" {COMMENT = 0;}

{identifier}\( {if(!COMMENT)printf("\nFUNCTION: \n%s",yytext);}

{identifier}(\[[0-9]*\])? {if(!COMMENT) printf("\n%s is an IDENTIFIER",yytext);}
```

```
\".*\" {if(!COMMENT)printf("\n%s is a STRING",yytext);}

[0-9]+ {if(!COMMENT) printf("\n%s is a NUMBER ",yytext);}

\{ {if(!COMMENT) printf("\nBLOCK BEGINS");}
\} {if(!COMMENT) printf("\nBLOCK ENDS");}

\) {if(!COMMENT);printf("\n)");}
= {if(!COMMENT) printf("\n%s is an ASSIGNMENT OPERATOR",yytext);}

\<= |
\>= |
\< |
\== |
\!= |
\> {if(!COMMENT) printf("\n%s is a RELATIONAL OPERATOR",yytext);}

\, |
\;  {if(!COMMENT) printf("\n%s is a SEPERATOR",yytext);}

%%
int main(int argc, char **argv)
{
    FILE *file;
    file=fopen("input.c","r");
    if(!file)
    {
        printf("could not open the file");
        exit(0);
    }
    yyin=file;
    yylex();
    printf("\n");
    return(0);
}

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

Running the code :

```
Microsoft Windows [Version 10.0.22000.318]
(c) Microsoft Corporation. All rights reserved.

C:\Users\dangi\OneDrive\Desktop\github\academic_nsut\sem 5\principles of
compiler construction\practicals\prac2>flex C_lex_analyzer.l

C:\Users\dangi\OneDrive\Desktop\github\academic_nsut\sem 5\principles of
compiler construction\practicals\prac2>gcc lex.yy.c

C:\Users\dangi\OneDrive\Desktop\github\academic_nsut\sem 5\principles of
compiler construction\practicals\prac2>a.exe
#include <stdio.h>
This is a PREPROCESSOR DIRECTIVE

void is a KEYWORD
FUNCTION:
main(
int is a KEYWORD
a is an IDENTIFIER
)

BLOCK BEGINS


int is a KEYWORD
a is an IDENTIFIER
, is a SEPERATOR
b is an IDENTIFIER
, is a SEPERATOR
c is an IDENTIFIER
; is a SEPERATOR

a is an IDENTIFIER
= is an ASSIGNMENT OPERATOR
1 is a NUMBER
; is a SEPERATOR

b is an IDENTIFIER
= is an ASSIGNMENT OPERATOR
2 is a NUMBER
; is a SEPERATOR

if is a KEYWORD (
a is an IDENTIFIER
> is a RELATIONAL OPERATOR
```

```
b is an IDENTIFIER
)

c is an IDENTIFIER
= is an ASSIGNMENT OPERATOR
0 is a NUMBER
; is a SEPERATOR

else is a KEYWORD

c is an IDENTIFIER
= is an ASSIGNMENT OPERATOR -
1 is a NUMBER
; is a SEPERATOR


FUNCTION:
printf(
"The value of c: %d" is a STRING
, is a SEPERATOR
c is an IDENTIFIER
)
; is a SEPERATOR

for is a KEYWORD (
int is a KEYWORD
i is an IDENTIFIER
= is an ASSIGNMENT OPERATOR
0 is a NUMBER
; is a SEPERATOR
i is an IDENTIFIER
< is a RELATIONAL OPERATOR
5 is a NUMBER
; is a SEPERATOR
i is an IDENTIFIER++
)

i is an IDENTIFIER++
; is a SEPERATOR

return is a KEYWORD
0 is a NUMBER
; is a SEPERATOR

BLOCK ENDS
```

2.Represent “C” language using Context Free Grammar.

```
The Context Free Grammar for C language can be given by G = (V, T, S, P):
where:

V = set of non-terminals
  = {program_unit, translation_unit, external_decl, function_definition, decl,
decl_list, decl_specs, storage_class_spec, type_spec, type_qualifier,
struct_or_union_spec, struct_or_union, struct_decl_list, init_declarator_list,
init_declarator, struct_decl, spec_qualifier_list, struct_declarator_list,
struct_declarator_list, struct_declarator, enum_spec, enumerator_list,
enumerator, declarator, direct_declarator, pointer, type_qualifier_list,
param_list, param_decl, id_list, initializer, initializer_list, type_name,
abstract_declarator, direct_abstract_declarator, stat, labeled_stat,
exp_stat,compound_stat, stat_list, selection_stat, iteration_stat, jump_stat,
exp assignment_exp, assignment_operator, conditional_exp, logical_or_exp,
logical_and_exp, inclusive_or_exp, exclusive_or_exp, and_exp, equality_exp,
relational_exp, shift_expression, additive_exp, mult_exp, cast_exp, unary_exp,
unary_operator, postfix_exp, primary_exp, argument_exp_list, consts,
int_const, char_const, float_const, id, string, enumeration_const,
storage_const, type_const, qual_const, struct_const, enum_const, DEFINE, IF,
ELSE, FOR, DO, WHILE, BREAK, SWITCH, CONTINUE, RETURN, CASE, DEFAULT, GOTO,
SIZEOF, PUNC, or_const, and_const, eq_const, shift_const, rel_const,
inc_const, point_const, HEADER}

T = set of terminals
  = {All ASCII characters}

S = start symbol = program_unit

P = set of productions
program_unit                -> HEADER
program_unit
                            | DEFINE primary_exp
program_unit
                            |
translation_unit

translation_unit            ->
external_decl
                            | translation_unit
external_decl

external_decl               -> function_definition
                            | decl
```

```
function_definition          -> decl_specs declarator decl_list
compound_stat
                             | declarator decl_list compound_stat
                             | decl_specs declarator
compound_stat
                             | declarator compound_stat

decl                         -> decl_specs init_declarator_list
';'
                             | decl_specs ';'

decl_list                    -> decl
                             | decl_list decl

decl_specs                   -> storage_class_spec decl_specs
                             | storage_class_spec
                             | type_spec
decl_specs
                             |
type_spec
                             | type_qualifier decl_specs
                             | type_qualifier

storage_class_spec           -> storage_const

type_spec                    ->
type_const
                             | struct_or_union_spec
                             | enum_spec

type_qualifier               -> qual_const

struct_or_union_spec         -> struct_or_union id '{' struct_decl_list '}' ';'
                             | struct_or_union id

struct_or_union              -> struct_const

struct_decl_list             -> struct_decl
                             | struct_decl_list struct_decl

init_declarator_list         -> init_declarator
                             | init_declarator_list ',' init_declarator

init_declarator              -> declarator
                             | declarator '=' initializer

struct_decl                  -> spec_qualifier_list struct_declarator_list ';'
```

```
spec_qualifier_list          -> type_spec spec_qualifier_list
                             | type_spec
                             | type_qualifier spec_qualifier_list
                             | type_qualifier

struct_declarator_list       -> struct_declarator
                             | struct_declarator_list ',' struct_declarator

struct_declarator            -> declarator
                             | declarator ':' conditional_exp
                             | ':' conditional_exp

enum_spec                    -> enum_const id '{' enumerator_list '}'
                             | enum_const '{' enumerator_list '}'
                             | enum_const id

enumerator_list              -> enumerator
                             | enumerator_list ',' enumerator

enumerator                   -> id
                             | id '=' conditional_exp

declarator                   -> pointer direct_declarator
                             | direct_declarator

direct_declarator            ->
id
                             | '(' declarator
')'
                             | direct_declarator '[' conditional_exp
']'
                             | direct_declarator '[' ']'
                             | direct_declarator '(' param_list ')'
                             | direct_declarator '(' id_list
')'
                             | direct_declarator '('
')'

pointer                      -> '*' type_qualifier_list
                             | '*'
                             | '*' type_qualifier_list pointer
                             | '*' pointer

type_qualifier_list          -> type_qualifier
                             | type_qualifier_list type_qualifier
```

```
param_list                  -> param_decl
                            | param_list ',' param_decl

param_decl                  -> decl_specs declarator
                            | decl_specs abstract_declarator
                            | decl_specs

id_list                     -> id
                            | id_list ',' id

initializer                 -> assignment_exp
                            | '{' initializer_list '}'
                            | '{' initializer_list ',' '}'

initializer_list            -> initializer
                            | initializer_list ',' initializer

type_name                   -> spec_qualifier_list abstract_declarator
                            | spec_qualifier_list

abstract_declarator         -> pointer
                            | pointer direct_abstract_declarator
                            |   direct_abstract_declarator

direct_abstract_declarator  -> '(' abstract_declarator ')'
                            | direct_abstract_declarator '[' conditional_exp
']'
                            | '[' conditional_exp ']'
                            | direct_abstract_declarator '[' ']'
                            | '[' ']'
                            | direct_abstract_declarator '(' param_list ')'
                            | '(' param_list ')'
                            | direct_abstract_declarator '(' ')'
                            | '(' ')'

stat                        ->
labeled_stat
                            |
exp_stat
                            |
compound_stat
                            |
selection_stat
                            | iteration_stat
                            | jump_stat

labeled_stat                -> id ':' stat
```

```
                          | CASE int_const ':' stat
                          | DEFAULT ':' stat

exp_stat                  -> exp ';'
                          | ';'

compound_stat             -> '{' decl_list stat_list
'}'
                          | '{' stat_list
'}'
                          | '{' decl_list
'}'
                          | '{'
'}'

stat_list                 ->
stat
                          | stat_list
stat

selection_stat            -> IF '(' exp ')'
stat                                %prec "then"
                          | IF '(' exp ')' stat ELSE stat
                          | SWITCH '(' exp ')' stat

iteration_stat            -> WHILE '(' exp ')' stat
                          | DO stat WHILE '(' exp ')' ';'
                          | FOR '(' exp ';' exp ';' exp ')' stat
                          | FOR '(' exp ';' exp ';'   ')' stat
                          | FOR '(' exp ';' ';' exp ')' stat
                          | FOR '(' exp ';' ';' ')' stat
                          | FOR '(' ';' exp ';' exp ')' stat
                          | FOR '(' ';' exp ';' ')' stat
                          | FOR '(' ';' ';' exp ')' stat
                          | FOR '(' ';' ';' ')' stat

jump_stat                 -> GOTO id ';'
                          | CONTINUE ';'
                          | BREAK ';'
                          | RETURN exp ';'
                          | RETURN ';'

exp                       -> assignment_exp
                          | exp ',' assignment_exp

assignment_exp            -> conditional_exp
```

```
                            | unary_exp assignment_operator
assignment_exp

assignment_operator         -> PUNC
                            | '='

conditional_exp             -> logical_or_exp
                            | logical_or_exp '?' exp ':' conditional_exp

logical_or_exp              -> logical_and_exp
                            | logical_or_exp or_const logical_and_exp

logical_and_exp             -> inclusive_or_exp
                            | logical_and_exp and_const inclusive_or_exp

inclusive_or_exp            -> exclusive_or_exp
                            | inclusive_or_exp '|' exclusive_or_exp

exclusive_or_exp            -> and_exp
                            | exclusive_or_exp '^' and_exp

and_exp                     -> equality_exp
                            | and_exp '&' equality_exp

equality_exp                -> relational_exp
                            | equality_exp eq_const relational_exp

relational_exp              -> shift_expression
                            | relational_exp '<' shift_expression
                            | relational_exp '>' shift_expression
                            | relational_exp rel_const shift_expression

shift_expression            -> additive_exp
                            | shift_expression shift_const additive_exp

additive_exp                -> mult_exp
                            | additive_exp '+' mult_exp
                            | additive_exp '-' mult_exp

mult_exp                    -> cast_exp
                            | mult_exp '*' cast_exp
                            | mult_exp '/' cast_exp
                            | mult_exp '%' cast_exp

cast_exp                    -> unary_exp
                            | '(' type_name ')' cast_exp
```

```
unary_exp                   -> postfix_exp
                            | inc_const unary_exp
                            | unary_operator cast_exp
                            | SIZEOF unary_exp
                            | SIZEOF '(' type_name ')'

unary_operator              -> '&' | '*' | '+' | '-' | '~' |
'!'

postfix_exp                 ->
primary_exp
                            | postfix_exp '[' exp ']'
                            | postfix_exp '(' argument_exp_list ')'
                            | postfix_exp '(' ')'
                            | postfix_exp '.' id
                            | postfix_exp point_const id
                            | postfix_exp inc_const

primary_exp                 ->
id
                            |
consts
                            |
string
                            | '(' exp ')'

argument_exp_list           -> assignment_exp
                            | argument_exp_list ',' assignment_exp

consts                      ->
int_const
                            | char_const
                            | float_const
                            | enumeration_const

int_const                   -> [0-9]+

char_const                  -> "'"."'"

float_const                 -> [0-9]+"."[0-9]+

id                          -> [a-zA-z_][a-zA-z_0-9]*

string                      -> \".*\"

enum_const                  -> "enum"
```

```
storage_const               -> "auto"
                            | "register"
                            | "static"
                            | "extern"
                            | "typedef"

type_const                  -> "void"
                            |  "char"
                            |  "short"
                            |  "int"
                            |  "long"
                            |  "float"
                            |  "double"
                            |  "signed"
                            |  "unsigned"

qual_const                  -> "const"
                            |  "volatile"

struct_const                -> "struct"
                            |  "union"

DEFINE                      -> "#define"[ ]+[a-zA-z_][a-zA-z_0-9]*

IF                          -> "if"

ELSE                        -> "else"

FOR                         -> "for"

DO                          -> "do"

WHILE                       -> "while"

BREAK                       -> "break"

SWITCH                      -> "switch"

CONTINUE                    -> "continue"

RETURN                      -> "return"

CASE                        -> "case"

DEFAULT                     -> "default"

GOTO                        -> "goto"
```

```
SIZEOF                      -> "sizeof"

PUNC                        -> "*="
                            |  "/="
                            |  "+="
                            |  "%="
                            |  ">>="
                            |  "-="
                            |  "<<="
                            |  "&="
                            |  "^="
                            |  "|="

or_const                    -> "||"

and_const                   -> "&&"

eq_const                    -> "=="
                            |  "!="

shift_const                 -> ">>"
                            |  "<<"

rel_const                   -> "<="
                            |  ">="

inc_const                   -> "++"
                            |  "--"

point_const                 -> "->"

HEADER                      -> "#include"[ ]+<[a-zA-z_][a-zA-z_0-9.]*>
```

3.Develop a simple calculator using LEX and YACC .

cal.l

```
%{
/* Definition section */
#include<stdio.h>
#include "cl.tab.h"
extern int yylval;
%}

/* Rule Section */
%%
[0-9]+ {
        yylval=atoi(yytext);
        return NUMBER;

    }
[\t] ;

[\n] return 0;

. return yytext[0];

%%

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

cl.y

```
%{
/* Definition section */
#include<stdio.h>
int flag=0;
%}

%token NUMBER

%left '+' '-'

%left '*' '/' '%'

%left '(' ')'
```

```
/* Rule Section */
%%

ArithmeticExpression: E{

        printf("\nResult=%d\n", $$);

        return 0;

        };
E:E'+'E {$$=$1+$3;}

|E'-'E {$$=$1-$3;}

|E'*'E {$$=$1*$3;}

|E'/'E {$$=$1/$3;}

|E'%'E {$$=$1%$3;}

|'('E')' {$$=$2;}

| NUMBER {$$=$1;}

;

%%

//driver code
void main()
{
printf("\nEnter Any Arithmetic Expression :\n");

yyparse();
if(flag==0)
printf("\nEntered arithmetic expression is Valid\n\n");
}

void yyerror(char *a)
{
printf("\nEntered arithmetic expression is Invalid\n\n");
flag=1;
}
```

Running the code:

```
Microsoft Windows [Version 10.0.22000.318]
(c) Microsoft Corporation. All rights reserved.

C:\Users\dangi\OneDrive\Desktop\github\academic_nsut\sem 5\principles of
compiler construction\practicals\prac4>bison -d cl.y

C:\Users\dangi\OneDrive\Desktop\github\academic_nsut\sem 5\principles of
compiler construction\practicals\prac4>flex cal.l

C:\Users\dangi\OneDrive\Desktop\github\academic_nsut\sem 5\principles of
compiler construction\practicals\prac4>gcc lex.yy.c cl.tab.c -w

C:\Users\dangi\OneDrive\Desktop\github\academic_nsut\sem 5\principles of
compiler construction\practicals\prac4>a.exe

Enter Any Arithmetic Expression :
9+3*(4*7/3)+1

Result=37

Entered arithmetic expression is Valid


C:\Users\dangi\OneDrive\Desktop\github\academic_nsut\sem 5\principles of
compiler construction\practicals\prac4>
```

4. Develop a parser for “C” language using LEX and YACC

input.c

```c
#include <stdio.h>
#define PI 3.14

struct inp
{
    int a;
};

int check(int a, int b)
{
    return (a > b);
}

/* Sample code */
int main()
{
    struct inp ab;
    int r = 5;
    printf("abc");
    return 0;
}
```

parser.l

```
%option yylineno

%{
    #include<stdio.h>
    #include"parser.tab.h"
%}


%%
"#include"[ ]+<[a-zA-z_][a-zA-z_0-9.]*> {return HEADER;}
"#define"[ ]+[a-zA-z_][a-zA-z_0-9]* {return DEFINE;}
"auto"|"register"|"static"|"extern"|"typedef" {return storage_const;}
"void"|"char"|"short"|"int"|"long"|"float"|"double"|"signed"|"unsigned"
{return type_const;}
"const"|"volatile" {return qual_const;}
"enum" {return enum_const;}
"struct"|"union" {return struct_const;}
"case" {return CASE;}
```

```
"default" {return DEFAULT;}
"if" {return IF;}
"switch" {return SWITCH;}
"else" {return ELSE;}
"for" {return FOR;}
"do" {return DO;}
"while" {return WHILE;}
"goto" {return GOTO;}
"continue" {return CONTINUE;}
"break" {return BREAK;}
"return" {return RETURN;}
"sizeof" {return SIZEOF;}
"||" {return or_const;}
"&&" {return and_const;}
"=="|"!=" {return eq_const;}
"<="|">=" {return rel_const;}
">>"|"<<" {return shift_const;}
"++"|"--" {return inc_const;}
"->" {return point_const;}
"*="|"/="|"+="|"%="|">>="|"-="|"<<="|"&="|"^="|"|=" {return PUNC;}
[0-9]+ {return int_const;}
[0-9]+"."[0-9]+ {return float_const;}
"'"."'" {return char_const;}
[a-zA-z_][a-zA-z_0-9]* {return id;}
\".*\" {return string;}
"//"(\\.|[^\n])*[\n]
                                ;
[/][*]([^*]|[*]*[^*/])*[*]+[/]
                                ;
[
\t\n]
                                ;
";"|"="|","|"{"|"}"|"("|")"|"["|"]"|"*"|"+"|"-
"|"/"|"?"|":"|"&"|"|"|"^"|"!"|"~"|"%"|"<"|">"                  {return
yytext[0];}
%%

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

parser.y

```
%{
    #include<stdio.h>
    int yylex(void);
    int yyerror(const char *s);
    int success = 1;
%}

%token int_const char_const float_const id string storage_const type_const
qual_const struct_const enum_const DEFINE
%token IF FOR DO WHILE BREAK SWITCH CONTINUE RETURN CASE DEFAULT GOTO SIZEOF
PUNC or_const and_const eq_const shift_const rel_const inc_const
%token point_const ELSE HEADER
%left '+' '-'
%left '*' '/'
%right UMINUS
%nonassoc "then"
%nonassoc ELSE

%start program_unit
%%
program_unit                : HEADER
program_unit
                            | DEFINE primary_exp
program_unit
                            |
translation_unit
                            ;
translation_unit            :
external_decl
                            | translation_unit
external_decl
                            ;
external_decl               : function_definition
                            | decl
                            ;
function_definition         : decl_specs declarator decl_list
compound_stat
                            | declarator decl_list compound_stat
                            | decl_specs declarator
compound_stat
                            | declarator compound_stat
                            ;
decl                        : decl_specs init_declarator_list
';'
                            | decl_specs ';'
                            ;
```

```
decl_list                   : decl
                            | decl_list decl
                            ;
decl_specs                  : storage_class_spec decl_specs
                            | storage_class_spec
                            | type_spec
decl_specs
                            |
type_spec
                            | type_qualifier decl_specs
                            | type_qualifier
                            ;
storage_class_spec          : storage_const
                            ;
type_spec                   :
type_const
                            | struct_or_union_spec
                            | enum_spec
                            ;
type_qualifier              : qual_const
                            ;
struct_or_union_spec        : struct_or_union id '{' struct_decl_list '}' ';'
                            | struct_or_union id
                            ;
struct_or_union             : struct_const
                            ;
struct_decl_list            : struct_decl
                            | struct_decl_list struct_decl
                            ;
init_declarator_list        : init_declarator
                            | init_declarator_list ',' init_declarator
                            ;
init_declarator             : declarator
                            | declarator '=' initializer
                            ;
struct_decl                 : spec_qualifier_list struct_declarator_list ';'
                            ;
spec_qualifier_list         : type_spec spec_qualifier_list
                            | type_spec
                            | type_qualifier spec_qualifier_list
                            | type_qualifier
                            ;
struct_declarator_list      : struct_declarator
                            | struct_declarator_list ',' struct_declarator
                            ;
struct_declarator           : declarator
                            | declarator ':' conditional_exp
```

```
                            | ':' conditional_exp
                            ;
enum_spec                   : enum_const id '{' enumerator_list '}'
                            | enum_const '{' enumerator_list '}'
                            | enum_const id
                            ;
enumerator_list             : enumerator
                            | enumerator_list ',' enumerator
                            ;
enumerator                  : id
                            | id '=' conditional_exp
                            ;
declarator                  : pointer direct_declarator
                            | direct_declarator
                            ;
direct_declarator           :
id
                            | '(' declarator
')'
                            | direct_declarator '[' conditional_exp
']'
                            | direct_declarator '[' ']'
                            | direct_declarator '(' param_list ')'
                            | direct_declarator '(' id_list
')'
                            | direct_declarator '('
')'
                            ;
pointer                     : '*' type_qualifier_list
                            | '*'
                            | '*' type_qualifier_list pointer
                            | '*' pointer
                            ;
type_qualifier_list         : type_qualifier
                            | type_qualifier_list type_qualifier
                            ;
param_list                  : param_decl
                            | param_list ',' param_decl
                            ;
param_decl                  : decl_specs declarator
                            | decl_specs abstract_declarator
                            | decl_specs
                            ;
id_list                     : id
                            | id_list ',' id
                            ;
initializer                 : assignment_exp
```

```
                            | '{' initializer_list '}'
                            | '{' initializer_list ',' '}'
                            ;
initializer_list            : initializer
                            | initializer_list ',' initializer
                            ;
type_name                   : spec_qualifier_list abstract_declarator
                            | spec_qualifier_list
                            ;
abstract_declarator         : pointer
                            | pointer direct_abstract_declarator
                            |   direct_abstract_declarator
                            ;
direct_abstract_declarator  : '(' abstract_declarator ')'
                            | direct_abstract_declarator '[' conditional_exp
']'
                            | '[' conditional_exp ']'
                            | direct_abstract_declarator '[' ']'
                            | '[' ']'
                            | direct_abstract_declarator '(' param_list ')'
                            | '(' param_list ')'
                            | direct_abstract_declarator '(' ')'
                            | '(' ')'
                            ;
stat                        :
labeled_stat
                            |
exp_stat
                            |
compound_stat
                            |
selection_stat
                            | iteration_stat
                            | jump_stat
                            ;
labeled_stat                : id ':' stat
                            | CASE int_const ':' stat
                            | DEFAULT ':' stat
                            ;
exp_stat                    : exp ';'
                            | ';'
                            ;
compound_stat               : '{' decl_list stat_list
'}'
                            | '{' stat_list
'}'
```

```
                            | '{' decl_list
'}'
                            | '{'
'}'
                            ;
stat_list                   :
stat
                            | stat_list
stat
                            ;
selection_stat              : IF '(' exp ')'
stat                                  %prec "then"
                            | IF '(' exp ')' stat ELSE stat
                            | SWITCH '(' exp ')' stat
                            ;
iteration_stat              : WHILE '(' exp ')' stat
                            | DO stat WHILE '(' exp ')' ';'
                            | FOR '(' exp ';' exp ';' exp ')' stat
                            | FOR '(' exp ';' exp ';'   ')' stat
                            | FOR '(' exp ';' ';' exp ')' stat
                            | FOR '(' exp ';' ';' ')' stat
                            | FOR '(' ';' exp ';' exp ')' stat
                            | FOR '(' ';' exp ';' ')' stat
                            | FOR '(' ';' ';' exp ')' stat
                            | FOR '(' ';' ';' ')' stat
                            ;
jump_stat                   : GOTO id ';'
                            | CONTINUE ';'
                            | BREAK ';'
                            | RETURN exp ';'
                            | RETURN ';'
                            ;
exp                         : assignment_exp
                            | exp ',' assignment_exp
                            ;
assignment_exp              : conditional_exp
                            | unary_exp assignment_operator
assignment_exp
                            ;
assignment_operator         : PUNC
                            | '='
                            ;
conditional_exp             : logical_or_exp
                            | logical_or_exp '?' exp ':' conditional_exp
                            ;
logical_or_exp              : logical_and_exp
                            | logical_or_exp or_const logical_and_exp
```

```
                              ;
logical_and_exp               : inclusive_or_exp
                              | logical_and_exp and_const inclusive_or_exp
                              ;
inclusive_or_exp              : exclusive_or_exp
                              | inclusive_or_exp '|' exclusive_or_exp
                              ;
exclusive_or_exp              : and_exp
                              | exclusive_or_exp '^' and_exp
                              ;
and_exp                       : equality_exp
                              | and_exp '&' equality_exp
                              ;
equality_exp                  : relational_exp
                              | equality_exp eq_const relational_exp
                              ;
relational_exp                : shift_expression
                              | relational_exp '<' shift_expression
                              | relational_exp '>' shift_expression
                              | relational_exp rel_const shift_expression
                              ;
shift_expression              : additive_exp
                              | shift_expression shift_const additive_exp
                              ;
additive_exp                  : mult_exp
                              | additive_exp '+' mult_exp
                              | additive_exp '-' mult_exp
                              ;
mult_exp                      : cast_exp
                              | mult_exp '*' cast_exp
                              | mult_exp '/' cast_exp
                              | mult_exp '%' cast_exp
                              ;
cast_exp                      : unary_exp
                              | '(' type_name ')' cast_exp
                              ;
unary_exp                     : postfix_exp
                              | inc_const unary_exp
                              | unary_operator cast_exp
                              | SIZEOF unary_exp
                              | SIZEOF '(' type_name ')'
                              ;
unary_operator                : '&' | '*' | '+' | '-' | '~' |
'!'
                              ;
postfix_exp                   :
primary_exp
```

```
                            | postfix_exp '[' exp ']'
                            | postfix_exp '(' argument_exp_list ')'
                            | postfix_exp '(' ')'
                            | postfix_exp '.' id
                            | postfix_exp point_const id
                            | postfix_exp inc_const
                            ;
primary_exp                 :
id
                            |
consts
                            |
string
                            | '(' exp ')'
                            ;
argument_exp_list           : assignment_exp
                            | argument_exp_list ',' assignment_exp
                            ;
consts                      :
int_const
                            | char_const
                            | float_const
                            | enum_const
                            ;
%%

int main()
{
    yyparse();
    if(success)
        printf("Parsing Successful\n");
    return 0;
}

int yyerror(const char *msg)
{
    extern int yylineno;
    printf("Parsing Failed\nLine Number: %d %s\n",yylineno,msg);
    success = 0;
    return 0;
}
```

Running the code :

```
C:\Users\dangi\OneDrive\Desktop\github\academic_nsut\sem 5\principles of
compiler construction\practicals\prac5>bison -d parser.y

C:\Users\dangi\OneDrive\Desktop\github\academic_nsut\sem 5\principles of
compiler construction\practicals\prac5>flex parser.l

C:\Users\dangi\OneDrive\Desktop\github\academic_nsut\sem 5\principles of
compiler construction\practicals\prac5>gcc lex.yy.c parser.tab.c -w

C:\Users\dangi\OneDrive\Desktop\github\academic_nsut\sem 5\principles of
compiler construction\practicals\prac5>a.exe < input.c
Parsing Successful

C:\Users\dangi\OneDrive\Desktop\github\academic_nsut\sem 5\principles of
compiler construction\practicals\prac5>
```