COMPILER DESIGN

ASSIGNMENT-2

Name:Arjun N R

SRN:PES2UG22CS910

Task-1 Abstract Syntax Tree generation

Lexer.l

```
%{
   #include <stdio.h>
   #include "abstract_syntax_tree.h"  // Include types first
   #include "parser.tab.h"           // Then include parser definitions
   extern void yyerror(char* s);
%}

/* Regular definitions */
digit [0-9]
letter      [a-zA-Z]
id     {letter}({letter}|{digit})*
digits      {digit}+
opFraction      (\.{digits})?
opExponent    ([Ee][+-]?{digits})?
```

```
number   {digits}{opFraction}{opExponent}
%option yylineno
%option noyywrap

%%
\/\/(.*) ; // ignore comments
[ \t\r\n]+ ; // ignore whitespaces - handle multiple
whitespace chars

"do"       { yylval.text = strdup(yytext); return T_DO; }
"while"     { yylval.text = strdup(yytext); return T_WHILE; }
"if"       { yylval.text = strdup(yytext); return T_IF; }
"else"      { yylval.text = strdup(yytext); return T_ELSE; }

"<="       { yylval.text = strdup(yytext); return T_LE; }
">="       { yylval.text = strdup(yytext); return T_GE; }
"=="       { yylval.text = strdup(yytext); return T_EQ; }
"!="       { yylval.text = strdup(yytext); return T_NE; }
"<"        { yylval.text = strdup(yytext); return T_LT; }
">"        { yylval.text = strdup(yytext); return T_GT; }

"("        { return *yytext; }
")"        { return *yytext; }
"{"        { return *yytext; }
"}"        { return *yytext; }
"."        { return *yytext; }
","        { return *yytext; }
"*"        { return *yytext; }
"+"        { return *yytext; }
";"        { return *yytext; }
"-"        { return *yytext; }
"/"        { return *yytext; }
"="        { return *yytext; }

{number}    {
            yylval.text = strdup(yytext);
            return T_NUM;
           }
{id}        {
            yylval.text = strdup(yytext);
            return T_ID;
           }
.          { printf("Unrecognized character: %s\n", yytext); } //
Print unrecognized chars for debugging
```

```
%%
```

Parser.y

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

	// Forward declarations for the types used in %union
	struct expression_node;
	struct statement_node;
	typedef struct expression_node expression_node;
	typedef struct statement_node statement_node;

	#include "abstract_syntax_tree.h"

	void yyerror(char* s);
	int yylex();
	extern int yylineno;

	// For debugging
	extern FILE* yyin;
```

```
        extern char* yytext;
%}

%union
{
        char* text;
        expression_node* exp_node;
        statement_node* stmt_node;
}

%token <text> T_ID T_NUM
%token <text> T_DO T_WHILE T_IF T_ELSE
%token <text> T_LT T_GT T_LE T_GE T_EQ T_NE

%type <exp_node> E T F ASSGN
%type <text> REL
%type <exp_node> CONDITION
%type <stmt_node> START STMT STMT_LIST

/* specify start symbol */
%start START
```

```
%%
START : STMT_LIST  {
        $$ = $1;
        flat_display_statement_tree($$, 1);
        printf("\n\nValid syntax\n");
        YYACCEPT;
    }
    | ASSGN {
        // Original AST display for backward compatibility with old test files
        $$ = init_assignment_node($1, NULL);
        printf("%s\n", $1->val);
        display_exp_tree($1->left);
        display_exp_tree($1->right);
        printf("\nValid syntax\n");
        YYACCEPT;
    }
    ;

STMT_LIST : STMT STMT_LIST {
        if ($1 != NULL) {
            statement_node* temp = $1;
```

```
                        while (temp->next != NULL) {
                                temp = $1;
                        }
                        temp->next = $2;
                        $$ = $1;
                } else {
                        $$ = $2;
                }
        }
        | /* empty */ { $$ = NULL; }
        ;

STMT : T_DO '{' STMT_LIST '}' T_WHILE '(' CONDITION ')' ';' {
                $$ = init_do_while_node($7, $3, NULL);
        }
        | T_IF '(' CONDITION ')' '{' STMT_LIST '}' T_ELSE '{'
STMT_LIST '}' {
                $$ = init_if_else_node($3, $6, $10, NULL);
        }
        | T_IF '(' CONDITION ')' '{' STMT_LIST '}' {
                $$ = init_if_else_node($3, $6, NULL, NULL);
        }
        | ASSGN ';' {
                $$ = init_assignment_node($1, NULL);
        }
        ;

CONDITION : T_ID REL T_ID {
                $$ = init_exp_node($2, init_exp_node($1, NULL,
NULL), init_exp_node($3, NULL, NULL));
        }
        | T_ID REL T_NUM {
                $$ = init_exp_node($2, init_exp_node($1, NULL,
NULL), init_exp_node($3, NULL, NULL));
        }
        | T_NUM REL T_ID {
                $$ = init_exp_node($2, init_exp_node($1, NULL,
NULL), init_exp_node($3, NULL, NULL));
        }
        ;

REL : T_LT { $$ = strdup("<"); }
        | T_GT { $$ = strdup(">"); }
        | T_LE { $$ = strdup("<="); }
        | T_GE { $$ = strdup(">="); }
        | T_EQ { $$ = strdup("=="); }
        | T_NE { $$ = strdup("!="); }
        ;

ASSGN : T_ID '=' E   { $$ = init_exp_node("=",
init_exp_node($1, NULL, NULL), $3); }
        ;

/* Expression Grammar */
E : E '+' T  { $$ = init_exp_node("+", $1, $3); }
  | E '-' T   { $$ = init_exp_node("-", $1, $3); }
  | T         { $$ = $1; }
  ;

T : T '*' F  { $$ = init_exp_node("*", $1, $3); }
  | T '/' F   { $$ = init_exp_node("/", $1, $3); }
  | F         { $$ = $1; }
  ;

F : '(' E ')'  { $$ = $2; }
  | T_ID     { $$ = init_exp_node($1, NULL, NULL); }
  | T_NUM        { $$ = init_exp_node($1, NULL, NULL); }
  ;

%%

/* error handling function */
void yyerror(char* s)
{
	printf("Error :%s at %d (near '%s')\n", s, yylineno, yytext);
}


/* main function - calls the yyparse() function which will in turn drive yylex() as well */
int main(int argc, char* argv[])
{
	if (argc < 2) {
		printf("Usage: %s <input_file>\n", argv[0]);
		return 1;
	}

	// Open the input file
}

    FILE* input_file = fopen(argv[1], "r");
    if (!input_file) {
        printf("Error: Could not open input file %s\n", argv[1]);
        return 1;
    }

    // Set yyin to use the input file
    yyin = input_file;

    // Print parsing of the file
    printf("Parsing file: %s\n", argv[1]);

    // Parse the input
    yyparse();

    // Clean up
    fclose(input_file);

    return 0;
}
```

Abstract_syntax_tree.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "abstract_syntax_tree.h"

expression_node* init_exp_node(char* val,
expression_node* left, expression_node* right)
{
    expression_node* node =
(expression_node*)malloc(sizeof(expression_node));
    node->val = strdup(val);
    node->left = left;
    node->right = right;
    return node;
}

statement_node* init_if_else_node(expression_node*
condition, statement_node* if_body, statement_node*
else_body, statement_node* next)
{
    statement_node* node =
(statement_node*)malloc(sizeof(statement_node));
```

```
    node->type = strdup("if-else");
    node->condition = condition;
    node->if_body = if_body;
    node->else_body = else_body;
    node->next = next;
    return node;
}

statement_node* init_do_while_node(expression_node* condition, statement_node* body, statement_node* next)
{
    statement_node* node = (statement_node*)malloc(sizeof(statement_node));
    node->type = strdup("do-while");
    node->condition = condition;
    node->if_body = body;  // We'll use if_body to store the do-while body
    node->else_body = NULL;
    node->next = next;
    return node;
}
```

```
statement_node* init_assignment_node(expression_node*
assign_exp, statement_node* next)
{
    statement_node* node =
(statement_node*)malloc(sizeof(statement_node));
    node->type = strdup("assignment");
    node->condition = assign_exp;
    node->if_body = NULL;
    node->else_body = NULL;
    node->next = next;
    return node;
}

void display_exp_tree(expression_node* exp_node)
{
    if (exp_node == NULL)
        return;

    printf("%s\n", exp_node->val);
    display_exp_tree(exp_node->left);
    display_exp_tree(exp_node->right);
}
```

```
// Helper function to print indentation
void print_indent(int level) {
    for (int i = 0; i < level; i++) {
        printf("  ");
    }
}

void display_statement_tree(statement_node* stmt_node,
int indent_level)
{
    if (stmt_node == NULL)
        return;

    if (strcmp(stmt_node->type, "assignment") == 0) {
        print_indent(indent_level);
        printf("Assignment:\n");
        print_indent(indent_level + 1);
        printf("Expression:\n");
        display_exp_tree(stmt_node->condition);
    }
    else if (strcmp(stmt_node->type, "if-else") == 0) {
        print_indent(indent_level);
        if (stmt_node->else_body != NULL) {
            printf("If-Else Statement:\n");
        } else {
            printf("If Statement:\n");
        }
        print_indent(indent_level + 1);
        printf("Condition:\n");
        display_exp_tree(stmt_node->condition);
        print_indent(indent_level + 1);
        printf("If Body:\n");
        display_statement_tree(stmt_node->if_body,
indent_level + 2);
        if (stmt_node->else_body) {
            print_indent(indent_level + 1);
            printf("Else Body:\n");
            display_statement_tree(stmt_node->else_body,
indent_level + 2);
        }
    }
    else if (strcmp(stmt_node->type, "do-while") == 0) {
        print_indent(indent_level);
        printf("Do-While Statement:\n");
        print_indent(indent_level + 1);
        printf("Body:\n");
        display_statement_tree(stmt_node->if_body,
indent_level + 2);
        print_indent(indent_level + 1);
        printf("Condition:\n");
        display_exp_tree(stmt_node->condition);
    }

    // Display the next statement
    display_statement_tree(stmt_node->next, indent_level);
}

// Helper function to print an expression node in flat format
void flat_display_exp_node(expression_node* exp_node) {
    if (exp_node == NULL)
        return;

    printf("%s", exp_node->val);

    if (exp_node->left != NULL) {
        printf(", ");
        flat_display_exp_node(exp_node->left);
    }

    if (exp_node->right != NULL) {
        printf(", ");
        flat_display_exp_node(exp_node->right);
    }
}

// Display statement tree in flat format with comma separation
void flat_display_statement_tree(statement_node* stmt_node, int is_first) {
    if (stmt_node == NULL)
        return;

    if (!is_first) {
        printf(", ");
    }

    if (strcmp(stmt_node->type, "assignment") == 0) {
        flat_display_exp_node(stmt_node->condition);
    }
    else if (strcmp(stmt_node->type, "if-else") == 0) {
        if (stmt_node->else_body == NULL) {
            printf("if");
            printf(", ");
            flat_display_exp_node(stmt_node->condition);

            if (stmt_node->if_body != NULL) {
                printf(", seq");
                flat_display_statement_tree(stmt_node->if_body,
0);
            }
        } else {
            printf("if-else");
            printf(", ");
            flat_display_exp_node(stmt_node->condition);

            if (stmt_node->if_body != NULL) {
                printf(", seq");
                flat_display_statement_tree(stmt_node->if_body,
0);
```

```
            }

            if (stmt_node->else_body != NULL) {
                printf(", seq");
                flat_display_statement_tree(stmt_node->else_body,
0);
            }
        }
    }
    else if (strcmp(stmt_node->type, "do-while") == 0) {
        printf("do-while");

        if (stmt_node->if_body != NULL) {
            printf(", seq");
            flat_display_statement_tree(stmt_node->if_body, 0);
        }

        printf(", ");
        flat_display_exp_node(stmt_node->condition);
    }

    // Display the next statement
    if (stmt_node->next != NULL) {
        flat_display_statement_tree(stmt_node->next, 0);
    }
}
```

Abstract_syntax_tree.h

```
typedef struct expression_node
{
    char* val;
    struct expression_node* left;
    struct expression_node* right;
}expression_node;

// Statement node with condition and body parts
typedef struct statement_node
{
    char* type;                 // "if-else" or "do-while"
    struct expression_node* condition;
    struct statement_node* if_body;
    struct statement_node* else_body;
    struct statement_node* next;  // For statement sequences
}statement_node;
```

```c
expression_node* init_exp_node(char* val, expression_node* left, expression_node* right);

statement_node* init_if_else_node(expression_node* condition, statement_node* if_body, statement_node* else_body, statement_node* next);

statement_node* init_do_while_node(expression_node* condition, statement_node* body, statement_node* next);

statement_node* init_assignment_node(expression_node* assign_exp, statement_node* next);


void display_exp_tree(expression_node* exp_node);

void display_statement_tree(statement_node* stmt_node, int indent_level);

void flat_display_statement_tree(statement_node* stmt_node, int is_first);

void flat_display_exp_node(expression_node* exp_node);
```

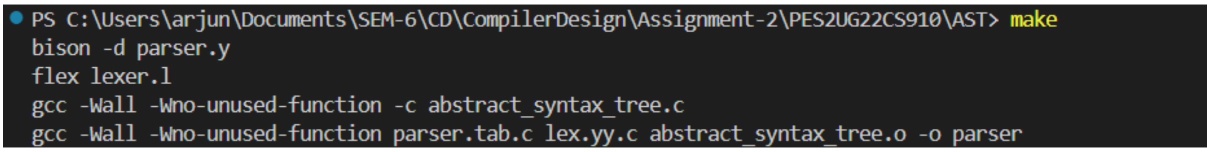
Makefile

```makefile
CC = gcc

YACC = bison -d

LEX = flex

CFLAGS = -Wall -Wno-unused-function
```

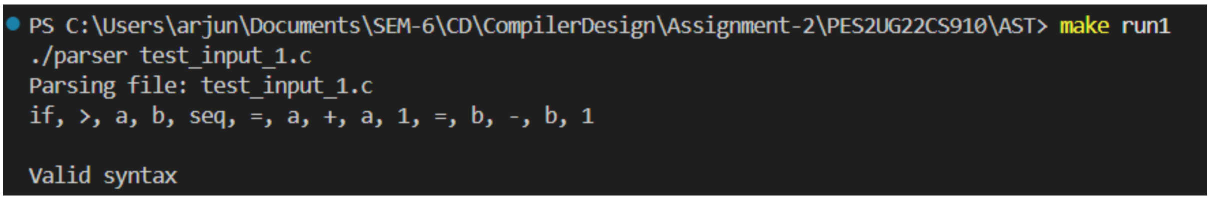
```
EXE = parser
OBJS = parser.tab.c lex.yy.c abstract_syntax_tree.o

all: $(EXE)

$(EXE): $(OBJS)
   $(CC) $(CFLAGS) $(OBJS) -o $(EXE)

abstract_syntax_tree.o: abstract_syntax_tree.c abstract_syntax_tree.h
   $(CC) $(CFLAGS) -c abstract_syntax_tree.c

parser.tab.c parser.tab.h: parser.y
   $(YACC) parser.y

lex.yy.c: lexer.l parser.tab.h
   $(LEX) lexer.l

clean:
   rm -f $(EXE) parser.tab.c parser.tab.h lex.yy.c *.o

# Sample run commands - use these to run the parser
```

```
run1: $(EXE)
    ./$(EXE) test_input_1.c

run2: $(EXE)
    ./$(EXE) test_input_2.c

run3: $(EXE)
    ./$(EXE) test_input_3.c

run4: $(EXE)
    ./$(EXE) test_input_4.c

# Help text explaining usage
help:
    @echo "Usage:"
    @echo "  make run1 - Parse test_input_1.c"
    @echo "  make run2 - Parse test_input_2.c"
    @echo "  make run3 - Parse test_input_3.c"
    @echo ""
    @echo "Or directly: ./$(EXE) <input_file>"
```

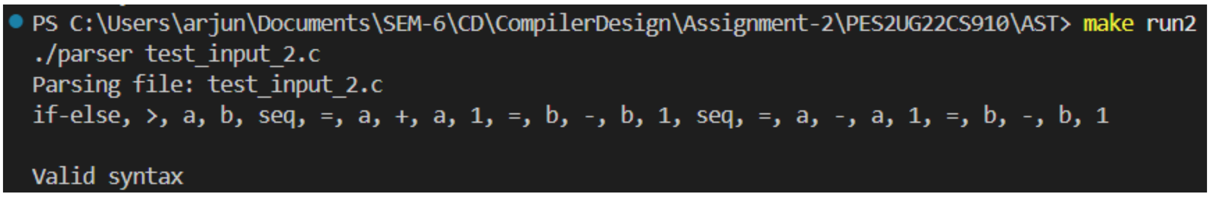
Outputs:



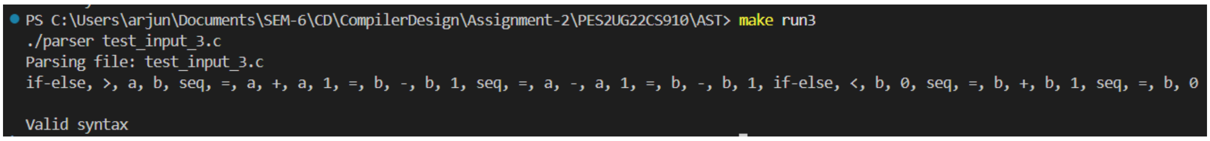
```
PS C:\Users\arjun\Documents\SEM-6\CD\CompilerDesign\Assignment-2\PES2UG22CS910\AST> make
bison -d parser.y
flex lexer.l
gcc -Wall -Wno-unused-function -c abstract_syntax_tree.c
gcc -Wall -Wno-unused-function parser.tab.c lex.yy.c abstract_syntax_tree.o -o parser
```



```
PS C:\Users\arjun\Documents\SEM-6\CD\CompilerDesign\Assignment-2\PES2UG22CS910\AST> make run1
./parser test_input_1.c
Parsing file: test_input_1.c
if, >, a, b, seq, =, a, +, a, 1, =, b, -, b, 1

Valid syntax
```



```
PS C:\Users\arjun\Documents\SEM-6\CD\CompilerDesign\Assignment-2\PES2UG22CS910\AST> make run2
./parser test_input_2.c
Parsing file: test_input_2.c
if-else, >, a, b, seq, =, a, +, a, 1, =, b, -, b, 1, seq, =, a, -, a, 1, =, b, -, b, 1

Valid syntax
```



```
PS C:\Users\arjun\Documents\SEM-6\CD\CompilerDesign\Assignment-2\PES2UG22CS910\AST> make run3
./parser test_input_3.c
Parsing file: test_input_3.c
if-else, >, a, b, seq, =, a, +, a, 1, =, b, -, b, 1, seq, =, a, -, a, 1, =, b, -, b, 1, if-else, <, b, 0, seq, =, b, +, b, 1, seq, =, b, 0

Valid syntax
```

Task-2: Intermediate Code Generation

Lexer.l

```
%{
   #define YYSTYPE char*
   #include <unistd.h>
   #include "parser.tab.h"
   #include <stdio.h>
   extern void yyerror(char* s);
%}

/* Regular definitions */
```

```
digit [0-9]
letter      [a-zA-Z]
id    {letter}({letter}|{digit})*
digits      {digit}+
opFraction      (\.{digits})?
opExponent    ([Ee][+-]?{digits})?
number   {digits}{opFraction}{opExponent}
%option yylineno
%option noyywrap

%%
\/\/(.*) ; // ignore comments
[\t\n ] ; // ignore whitespaces (added space to whitespace list)
"if"       { return T_IF; }
"else"     { return T_ELSE; }
"do"       { return T_DO; }
"while"    { return T_WHILE; }
"<="       { return T_LE; }
">="       { return T_GE; }
"=="       { return T_EQ; }
"!="       { return T_NE; }
```

```
"("             { return *yytext; }
")"             { return *yytext; }
"{"        { return *yytext; }
"}"        { return *yytext; }
"."        { return *yytext; }
","        { return *yytext; }
"*"        { return *yytext; }
"+"        { return *yytext; }
";"        { return *yytext; }
"-"        { return *yytext; }
"/"        { return *yytext; }
"="        { return *yytext; }
">"        { return *yytext; }
"<"        { return *yytext; }
{number}{
                   yylval = strdup(yytext);
                   return T_NUM;
             }
{id}       {
                   yylval = strdup(yytext);
                   return T_ID;
             }
```

```
.              {} // anything else => ignore
%%
```

Parser.y

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

        #define YYSTYPE char*

        void yyerror(char* s);
        int yylex();
        extern int yylineno;

        FILE* icg_quad_file;
        int temp_no = 1;

        // For storing label information
        char* if_true_label;
        char* if_false_label;
        char* if_end_label;
        char* loop_start_label;
%}

%token T_ID T_NUM T_IF T_ELSE T_DO T_WHILE
%token T_LE T_GE T_EQ T_NE

/* Specify precedence to resolve the dangling else problem
*/
%nonassoc IFX
%nonassoc T_ELSE

/* specify start symbol */
%start START

%%
START : STMT_LIST  {
                                        printf("\n");
                                        print_three_address_code();
                                        printf("\n");
                                        print_quad_table();
                                        printf("Valid syntax\n");
```

```
                                                YYACCEPT;
                                        }
        ;

STMT_LIST : STMT STMT_LIST { }
        | /* epsilon */ { }
        ;

STMT : IF_STMT
    | DO_WHILE_STMT
    | ASSGN ';' { }
    ;

IF_STMT : T_IF '(' CONDITION ')' '{' {
                        // Generate labels
                        if_true_label = new_label();
                        if_false_label = new_label();

                        // Jump to true part if condition is true
                        add_quad("If", $3, "", if_true_label);
                        // Otherwise go to false part
                        add_quad("goto", "", "", if_false_label);
```

```
            // Mark the beginning of true part
            add_quad("Label", "", "", if_true_label);
        }
        STMT_LIST '}' ELSE_PART
  ;

ELSE_PART : T_ELSE '{' {
            // Generate end label for if-else
            if_end_label = new_label();
            // After true block, go to end
            add_quad("goto", "", "", if_end_label);
            // Mark the beginning of false part
            add_quad("Label", "", "", if_false_label);
        }
        STMT_LIST '}' {
            // End of if-else statement
            add_quad("Label", "", "", if_end_label);
            }
  | %prec IFX {
            // End of if statement without else
            add_quad("Label", "", "", if_false_label);
        }
  ;

DO_WHILE_STMT : T_DO '{' {
            // Generate label for loop start
            loop_start_label = new_label();
            // Mark the beginning of loop
            add_quad("Label", "", "", loop_start_label);
        }
        STMT_LIST '}' T_WHILE '(' CONDITION ')' ';' {
            // If condition is true, jump back to loop start
            add_quad("If", $7, "", loop_start_label);
            }
  ;

CONDITION : T_ID REL T_ID {
                char* temp = new_temp();
                add_quad($2, $1, $3, temp);
                $$ = temp;
            }
    | T_ID REL T_NUM {
                char* temp = new_temp();
                add_quad($2, $1, $3, temp);
```

```
                        $$ = temp;
                    }
        | T_NUM REL T_ID {
                        char* temp = new_temp();
                        add_quad($2, $1, $3, temp);
                        $$ = temp;
                    }
        ;

REL : '<' { $$ = "<"; }
        | '>' { $$ = ">"; }
        | T_LE { $$ = "<="; }
        | T_GE { $$ = ">="; }
        | T_EQ { $$ = "=="; }
        | T_NE { $$ = "!="; }
        ;

/* Grammar for assignment */
ASSGN : T_ID '=' E    { quad_code_gen($1, $3, "=", ""); }
        ;

/* Expression Grammar */
```

```
E : E '+' T  {
                                        char* temp = new_temp();
                                        quad_code_gen(temp, $1, "+", $3);
                                        $$ = temp;
                                }
        | E '-' T     {
                                        char* temp = new_temp();
                                        quad_code_gen(temp, $1, "-", $3);
                                        $$ = temp;
                                }
        | T           { $$ = $1; }
        ;

T : T '*' F  {
                                        char* temp = new_temp();
                                        quad_code_gen(temp, $1, "*", $3);
                                        $$ = temp;
                                }
        | T '/' F     {
                                        char* temp = new_temp();
                                        quad_code_gen(temp, $1, "/", $3);
                                        $$ = temp;
```

```
                        }
        | F         { $$ = $1; }
        ;

F : '(' E ')'  { $$ = $2; }
        | T_ID              { $$ = $1; }
        | T_NUM         { $$ = $1; }
        ;

%%

/* error handling function */
void yyerror(char* s)
{
        printf("Error: %s at line %d\n", s, yylineno);
}

/* main function - calls the yyparse() function which will in
turn drive yylex() as well */
int main(int argc, char* argv[])
{
        if (argc < 2) {
```

```
        printf("Usage: %s <input_file>\n", argv[0]);
        return 1;
    }

    // Open the input file
    FILE* input_file = fopen(argv[1], "r");
    if (!input_file) {
        printf("Error: Could not open input file %s\n",
argv[1]);
        return 1;
    }

    // Set yyin to use the input file
    extern FILE* yyin;
    yyin = input_file;

    // Parse the input
    yyparse();

    // Clean up
    fclose(input_file);
```

```
        return 0;
}
```

Quad_generation.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "quad_generation.h"

// Structure to store quadruples
typedef struct quadruple {
    char op[10];
    char arg1[20];
    char arg2[20];
    char result[20];
    struct quadruple* next;
} quadruple;

// Global variables
int label_no = 1;
quadruple* quad_list = NULL;
quadruple* quad_tail = NULL;
```

```
// Function to add a new quadruple to the list
void add_quad(char* op, char* arg1, char* arg2, char* result)
{
    quadruple* new_quad =
(quadruple*)malloc(sizeof(quadruple));
    strcpy(new_quad->op, op);
    strcpy(new_quad->arg1, arg1 ? arg1 : "");
    strcpy(new_quad->arg2, arg2 ? arg2 : "");
    strcpy(new_quad->result, result ? result : "");
    new_quad->next = NULL;

    if (quad_list == NULL) {
        quad_list = new_quad;
        quad_tail = new_quad;
    } else {
        quad_tail->next = new_quad;
        quad_tail = new_quad;
    }
}

void quad_code_gen(char* a, char* b, char* op, char* c) {
    add_quad(op, b, c, a);
}

char* new_temp() {
    char* temp = (char*)malloc(sizeof(char)*10);
    sprintf(temp, "t%d", temp_no);
    ++temp_no;
    return temp;
}

char* new_label() {
    char* label = (char*)malloc(sizeof(char)*10);
    sprintf(label, "L%d", label_no);
    ++label_no;
    return label;
}

// Function to generate three-address code
void emit_3ac(char* op, char* arg1, char* arg2, char* result)
{
    if (strcmp(op, "Label") == 0) {
        printf("%s :\t ", result);
    } else if (strcmp(op, "goto") == 0) {
        printf("%s \t\t\t %s\n", op, result);
    } else if (strcmp(op, "If") == 0) {
        printf("if %s goto %s\n", arg1, result);
    } else if (strcmp(op, "=") == 0) {
        printf("%s = %s\n", result, arg1);
    } else {
        printf("%s = %s %s %s\n", result, arg1, op, arg2);
    }
}

// Check if a label is referenced by any quad
int is_label_referenced(char* label) {
    quadruple* q = quad_list;
    while (q != NULL) {
        if ((strcmp(q->op, "goto") == 0 || strcmp(q->op, "If") ==
0) &&
            strcmp(q->result, label) == 0) {
            return 1;
        }
        q = q->next;
    }
    return 0;
}

// Function to print the 3AC from quadruple list
void print_three_address_code() {
    printf("Three address code:\n");
    quadruple* q = quad_list;
    while (q != NULL) {
        if (strcmp(q->op, "Label") == 0) {
            // Only print the label if it's referenced somewhere
            // or if it's not the last quad
            if (is_label_referenced(q->result) || q->next != NULL) {
                printf("%s :\t ", q->result);
            }
        } else if (strcmp(q->op, "goto") == 0) {
            printf("goto %s\n", q->result);
        } else if (strcmp(q->op, "If") == 0) {
            printf("if %s goto %s\n", q->arg1, q->result);
        } else if (strcmp(q->op, "=") == 0) {
            printf("%s = %s\n", q->result, q->arg1);
        } else {
            printf("%s = %s %s %s\n", q->result, q->arg1, q->op, q->arg2);
        }
        q = q->next;
    }
    printf("\n");
}

// Function to print the quadruple table
void print_quad_table() {
    printf("op\targ1\targ2\tresult\n");
    quadruple* q = quad_list;
    while (q != NULL) {
        printf("%s\t%s\t%s\t%s\n", q->op, q->arg1, q->arg2, q->result);
        q = q->next;
    }
}
```

Quad_generation.h

```
extern FILE* icg_quad_file;     //pointer to the output file
extern int temp_no;             //variable to keep track of current temporary count
extern int label_no;          //variable to keep track of current label count


void quad_code_gen(char* a, char* b, char* op, char* c);
char* new_temp();
char* new_label();
void emit_3ac(char* op, char* arg1, char* arg2, char* result);
void add_quad(char* op, char* arg1, char* arg2, char* result);
void print_three_address_code();
void print_quad_table();
```

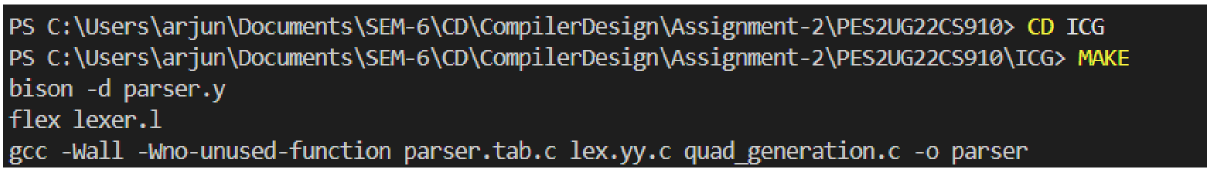
Makefile

```
CC = gcc
YACC = bison -d
LEX = flex
CFLAGS = -Wall -Wno-unused-function

EXE = parser
OBJS = parser.tab.c lex.yy.c quad_generation.c

all: $(EXE)
```

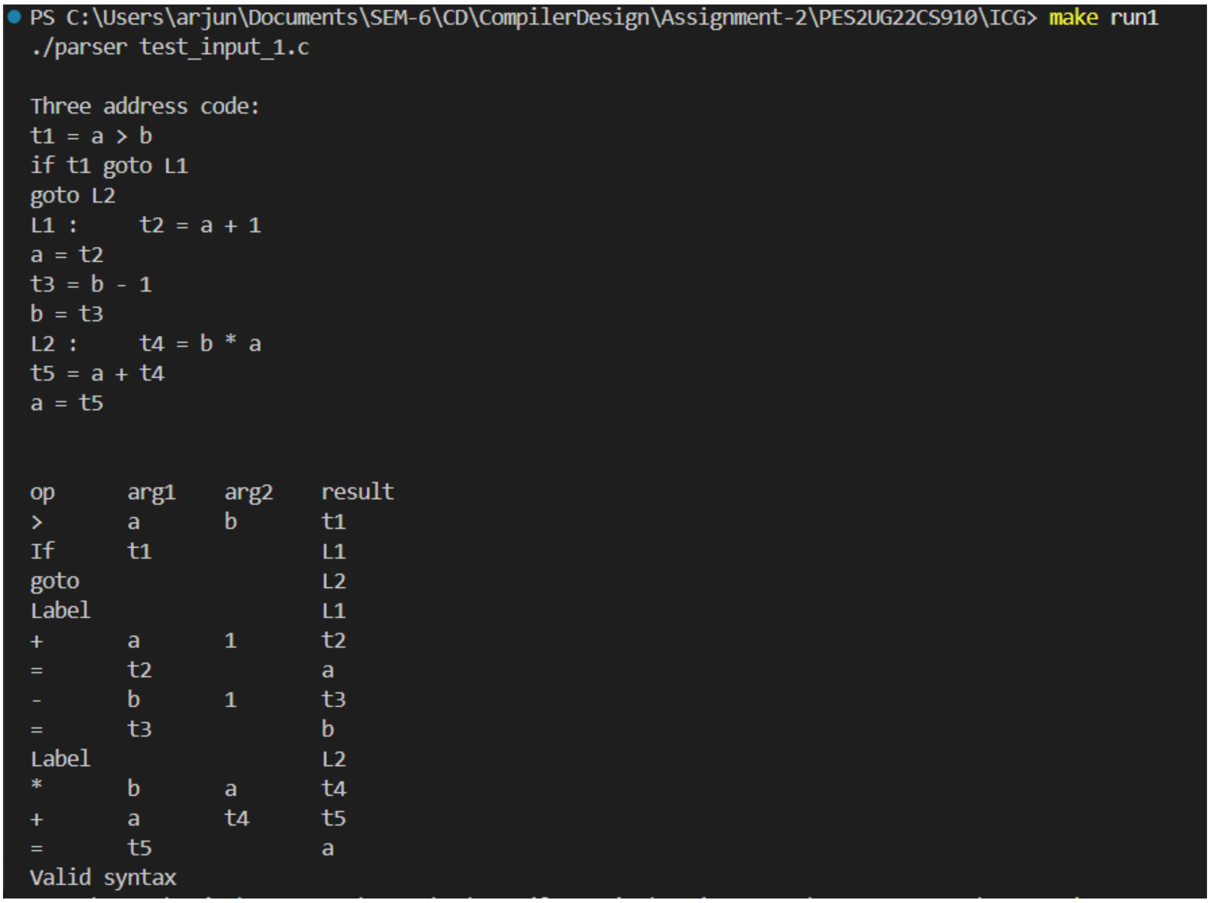
```
$(EXE): $(OBJS)
    $(CC) $(CFLAGS) $(OBJS) -o $(EXE)

quad_generation.o: quad_generation.c quad_generation.h
    $(CC) $(CFLAGS) -c quad_generation.c

parser.tab.c parser.tab.h: parser.y
    $(YACC) parser.y

lex.yy.c: lexer.l parser.tab.h
    $(LEX) lexer.l

clean:
    rm -f $(EXE) parser.tab.c parser.tab.h lex.yy.c *.o

# Sample run commands
run1:
    ./$(EXE) test_input_1.c

run2:
    ./$(EXE) test_input_2.c
```

```
# Help text explaining usage

help:

    @echo "Usage:"

    @echo "  make run1 - Process test_input_1.c"

    @echo "  make run2 - Process test_input_2.c"

    @echo ""

    @echo "Or directly: ./$(EXE) <input_file>"
```

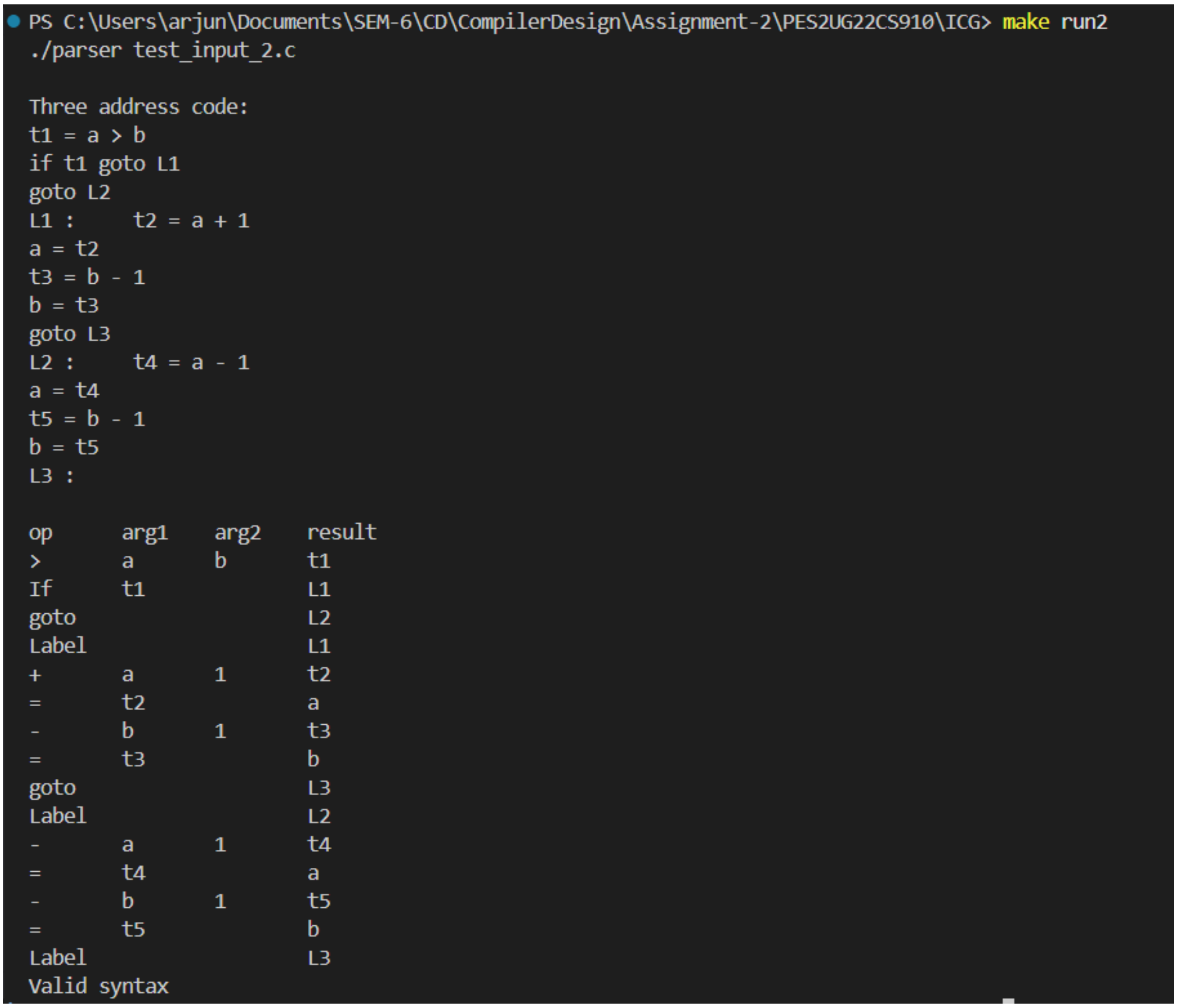
Outputs



```
PS C:\Users\arjun\Documents\SEM-6\CD\CompilerDesign\Assignment-2\PES2UG22CS910> CD ICG
PS C:\Users\arjun\Documents\SEM-6\CD\CompilerDesign\Assignment-2\PES2UG22CS910\ICG> MAKE
bison -d parser.y
flex lexer.l
gcc -Wall -Wno-unused-function parser.tab.c lex.yy.c quad_generation.c -o parser
```



```
PS C:\Users\arjun\Documents\SEM-6\CD\CompilerDesign\Assignment-2\PES2UG22CS910\ICG> make run1
./parser test_input_1.c

Three address code:
t1 = a > b
if t1 goto L1
goto L2
L1 :     t2 = a + 1
a = t2
t3 = b - 1
b = t3
L2 :     t4 = b * a
t5 = a + t4
a = t5


op      arg1    arg2    result
>       a       b       t1
If      t1              L1
goto                    L2
Label                   L1
+       a       1       t2
=       t2              a
-       b       1       t3
=       t3              b
Label                   L2
*       b       a       t4
+       a       t4      t5
=       t5              a
Valid syntax
```



```
PS C:\Users\arjun\Documents\SEM-6\CD\CompilerDesign\Assignment-2\PES2UG22CS910\ICG> make run2
./parser test_input_2.c

Three address code:
t1 = a > b
if t1 goto L1
goto L2
L1 :     t2 = a + 1
a = t2
t3 = b - 1
b = t3
goto L3
L2 :     t4 = a - 1
a = t4
t5 = b - 1
b = t5
L3 :

op      arg1    arg2    result
>       a       b       t1
If      t1              L1
goto                    L2
Label                   L1
+       a       1       t2
=       t2              a
-       b       1       t3
=       t3              b
goto                    L3
Label                   L2
-       a       1       t4
=       t4              a
-       b       1       t5
=       t5              b
Label                   L3
Valid syntax
```