Ex-no: 9 M. Rohith
07-11-2024 3122 21 5001 085

SSN COLLEGE OF ENGINEERING, KALAVAKKAM
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
UCS 2702 - Compiler Lab
Programming Assignment-9 Implementation of mini compiler

Combine all the phases and make it as a single code to run all the phases.

Input: source code in C language

Output: Assembly language code

Program code:

lexer.l

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

#define MAX_TOKENS 100

typedef struct {
    char symbol[32];
    char type[10];
} Token;

Token token_array[MAX_TOKENS];
int token_count = 0;

void yyerror(char *s);
void add_token(char *symbol, char *type);
void print_tokens();
%}


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

";"              {add_token(";","Semicolon"); return ';';}
":="        { add_token(yytext, "Operator"); return ASSIGN; }
"+"         { add_token("+", "Operator"); return PLUS; }
"-"        { add_token("-", "Operator"); return MINUS; }
"*"         { add_token("*", "Operator"); return MUL; }
"/"        { add_token("/", "Operator"); return DIV; }
"and"       { add_token("and", "AND"); return AND; }
"or"        { add_token("or", "OR"); return OR; }
"if"       { add_token("if", "Keyword"); return IF; }
"else"      { add_token("else", "Keyword"); return ELSE; }
```

```
"while"      { add_token("while", "Keyword"); return WHILE; }
"<"          { add_token("<", "relop"); return LT; }
">"          { add_token(">", "relop"); return GT; }
"("          { add_token("(", "LPAREN"); return '('; }
")"          { add_token(")", "RPAREN"); return ')'; }
"{"          { add_token("{", "LBRACE"); return '{'; }
"}"          { add_token("}", "RBRACE"); return '}'; }
{identifier} {
    yylval.str = strdup(yytext);
    add_token(yytext, "ID");

    return ID;
}
[0-9]+ {
    add_token(yytext, "Number");
    yylval.num = atoi(yytext);
    return NUM;
}
[ \t\n] ;
. {
    printf("Unexpected character: %s\n", yytext);
}

%%

void add_token(char *symbol, char *type) {
    if (token_count < MAX_TOKENS) {
        strncpy(token_array[token_count].symbol, symbol,32);
        strncpy(token_array[token_count].type,type,10);
        token_count++;
    } else {
        printf("Token limit reached. Cannot add more tokens.\n");
    }
}

void print_tokens() {
    printf("Tokens:\n");
    for (int i = 0; i < token_count; i++) {
        printf("|%-14s|%-15s|\n", token_array[i].symbol, token_array[i].type);
    }
}

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

parser.y

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

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

int temp_count = 1;
int label_count = 100;
int print_mode=1;
int ind=0;
char print_stmts[1000];


typedef struct{
            char target[10];
            char op[10];
            char arg1[10];
            char arg2[10];
            char other[100];
}Instruction;
Instruction instruction_set[100];

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

int new_label() {
    return label_count++;
}




%}

%union {
    char* str;
    int num;
}

%token <str> ID
%token <num> NUM
%token ASSIGN PLUS MINUS MUL DIV AND OR LT GT
%token IF ELSE WHILE
%type <str> expr term factor bool_expr rel_expr comparison condition else_stmt
%type <str> if_stmt stmt while_stmt

%%
start: stmt_list
```

```
    ;

stmt_list: stmt
    | stmt_list stmt
    ;

stmt: ID ASSIGN expr ';' {
            if(print_mode==1)
            {

                        printf("%s = %s\n", $1, $3);
                        fflush(stdout);
                        free($3);
                        free($1);
            }
            else
            {
                        char *tac=(char*) malloc(50);
                        sprintf(tac,"%s = %s\n",$1,$3);
                        strcat(print_stmts,tac);
                        free(tac);

            }
    }
    | ID ASSIGN bool_expr ';'{
        if(print_mode==1)
            {

                        printf("%s = %s\n", $1, $3);
                        fflush(stdout);
                        free($3);
                        free($1);
            }
            else
            {
                        char *tac=(char*) malloc(50);
                        sprintf(tac,"%s = %s\n",$1,$3);
                        strcat(print_stmts,tac);
                        free(tac);
            }
    }
    | if_stmt
    | while_stmt
    ;

if_stmt: IF '(' condition ')' '{' stmt_list '}' {
        printf("if %s goto E.true\n", $3);
        printf("goto E.false\n");
        printf("E.true:\n");
            printf("%s",print_stmts);
            strcpy(print_stmts,"\0");
        printf("goto E.end\n");
        printf("E.false:\n");
        printf("E.end:\n");
        print_mode=1;
  }
  |IF '(' condition ')' '{' stmt_list '}' else_stmt  '{' stmt_list '}'{

            printf("if %s goto E.true\n", $3);
        printf("goto E.false\n");
        printf("E.true:\n");
            printf("%s",$8);
        printf("goto E.end\n");
        printf("E.false:\n");
        printf("%s",print_stmts);
        printf("goto E.end\n");
        printf("E.end:\n");
            print_mode=0;
  }

    ;
else_stmt: ELSE
{

            char *temp=(char *) malloc(50);
            sprintf(temp,"%s",print_stmts);
            $$=temp;
        strcpy(print_stmts,"\0");




}




while_stmt: WHILE '(' condition ')' '{' stmt '}'
{
            printf("s.begin:\n");
            printf("if %s goto E.true\n", $3);
        printf("goto E.false\n");
        printf("E.true:\n");
            printf("%s\n",print_stmts);
            strcpy(print_stmts,"");
            printf("goto s.begin\n");
        printf("E.false: goto s.next\n");
        printf("s.next:");
        print_mode=1;
     }
     ;

expr: expr PLUS term {
            if(print_mode==1)
            {
                        char* temp = new_temp();
```

```
                printf("%s = %s + %s\n", temp, $1, $3);
                fflush(stdout);
                $$ = temp;
                free($1);
                free($3);
        }
        else
        {
                char *tac=(char*)malloc(50);
                char* temp = new_temp();
                sprintf(tac,"%s = %s + %s\n", temp, $1, $3);
                $$=temp;
                strcat(print_stmts,tac);

        }
   }
   | expr MINUS term {
        if(print_mode==1)
        {
                char* temp = new_temp();
                printf("%s = %s - %s\n", temp, $1, $3);
                fflush(stdout);
                $$ = temp;
                free($1);
                free($3);
        }
      else
        {
                char *tac=(char*)malloc(50);
                char* temp = new_temp();
                sprintf(tac,"%s = %s + %s\n", temp, $1, $3);
                strcat(print_stmts,tac);
        }
   }
   | term {
      $$ = $1;
   }
   ;

term: term MUL factor {
        if(print_mode==1)
        {

                char* temp = new_temp();
                printf("%s = %s * %s\n", temp, $1, $3);
                fflush(stdout);
                $$ = temp;
                free($1);
                free($3);
        }
        else
        {
                char *tac=(char*)malloc(50);
                char* temp = new_temp();
                sprintf(tac,"%s = %s * %s\n", temp, $1, $3);
                strcat(print_stmts,tac);

        }
  }
  | term DIV factor {
        if(print_mode==1)
        {
                char* temp = new_temp();
                printf("%s = %s / %s\n", temp, $1, $3);
                fflush(stdout);
                $$ = temp;
                free($1);
                free($3);
        }
        else
        {
                char *tac=(char*)malloc(50);
                char* temp = new_temp();
                sprintf(tac,"%s = %s / %s\n", temp, $1, $3);
                strcat(print_stmts,tac);
        }
  }
  | factor {
      $$ = $1;
  }
  ;

factor: ID {
      $$ = strdup($1);
  }
  | NUM {
      char* temp = (char*)malloc(10);
      sprintf(temp, "%d", $1);
      $$ = temp;
  }
  | MINUS factor {
        if(print_mode==1)
        {
                char* temp = new_temp();
                printf("%s = -%s\n", temp, $2);
                fflush(stdout);
                $$ = temp;
                free($2);
        }
        else
        {
                char* temp = new_temp();
                char*tac= (char*) malloc(50);
        sprintf(tac,"%s = -%s\n", temp, $2);
```

```
            $$ = temp;

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

bool_expr: bool_expr OR rel_expr {
        char* temp = new_temp();
        printf("%s = %s or %s\n", temp, $1, $3);
        fflush(stdout);
        $$ = temp;
        free($1);
        free($3);
    }
    | rel_expr {
        $$ = $1;
    }
    ;

rel_expr: rel_expr AND comparison {
        char* temp = new_temp();
        printf("%s = %s and %s\n", temp, $1, $3);
        fflush(stdout);
        $$ = temp;
        free($1);
        free($3);
    }
    | comparison {
        $$ = $1;
    }
    ;

comparison: ID LT ID {
        int l1 = label_count++;
        int l2 = label_count++;
        int l3 = label_count++;
        char* temp = new_temp();
        printf("%d: if %s < %s goto %d\n", l1, $1, $3, l2);
        printf("%s := 0\n", temp);
        printf("goto %d\n", l3);
        printf("%d: %s := 1\n", l2, temp);
        printf("%d:\n", l3);
        fflush(stdout);
        $$ = temp;
    }
    | ID GT ID {
        int l1 = label_count++;
        int l2 = label_count++;
        int l3 = label_count++;
        char* temp = new_temp();
        printf("%d: if %s > %s goto %d\n", l1, $1, $3, l2);
        printf("%s := 0\n", temp);
        printf("goto %d\n", l3);
        printf("%d: %s := 1\n", l2, temp);
        printf("%d:\n", l3);
        fflush(stdout);
        $$ = temp;
    }
    ;
condition: ID LT ID{
            char* temp=new_temp();
            sprintf(temp, "%s < %s", $1, $3);
            $$ = temp;
            print_mode=0;}
       | ID GT ID{char* temp=new_temp();
            sprintf(temp, "%s > %s", $1, $3);
            $$ = temp;
            print_mode=0;}


%%

void yyerror(char *s) {
    fprintf(stderr, "Error: %s\n", s);
}

void load_instruction() {
    FILE *fp = fopen("output.txt", "r");
    if (fp == NULL) {
        perror("Error opening file");
        return;
    }

    char line[100];
    while (fgets(line, sizeof(line), fp) != NULL) {
        int count = sscanf(line, "%s = %s %s %s",
                 instruction_set[ind].target,
                 instruction_set[ind].arg1,
                 instruction_set[ind].op,
                 instruction_set[ind].arg2);

        if (count >=2) {

          ind++;
        }
        else
        {

          strncpy(instruction_set[ind].other, line, sizeof(instruction_set[ind].other) - 1);
          instruction_set[ind].other[sizeof(instruction_set[ind].other) - 1] = '\0';
          ind++;
      }

   }

   fclose(fp);


}
void constant_folding()
{
        for(int i=0;i!=ind;i++)
        {
                if(instruction_set[i].other[0]=='\0'){
                        if(isdigit(instruction_set[i].arg1[0]) && isdigit(instruction_set[i].arg2[0]))
                        {

                                if(strcmp(instruction_set[i].op,"+")==0)
                                {
                                        int val1=atoi(instruction_set[i].arg1);
                                        int val2=atoi(instruction_set[i].arg2);
                                        int result=val1+val2;
                                        strcpy(instruction_set[i].arg2,"");
                                        strcpy(instruction_set[i].op,"");
                                        sprintf(instruction_set[i].arg1,"%d",result);
                                }
                                if(strcmp(instruction_set[i].op,"*")==0)
                                {
                                        int val1=atoi(instruction_set[i].arg1);
                                        int val2=atoi(instruction_set[i].arg2);
                                        int result=val1*val2;
                                        strcpy(instruction_set[i].arg2,"");
                                        strcpy(instruction_set[i].op,"");
                                        sprintf(instruction_set[i].arg1,"%d",result);
                                }

                        }
                }
        }
}
void algebric_identites()
{
        for(int i=0;i!=ind;i++)
        {
                if(instruction_set[i].other[0]=='\0')
                {
                        if(strcmp(instruction_set[i].op,"+")==0 &&
strcmp(instruction_set[i].arg2,"0")==0)
                        {
                                strcpy(instruction_set[i].op,"");
                                strcpy(instruction_set[i].arg2,"");
                        }
```

```
            if(strcmp(instruction_set[i].op,"*")==0 &&
strcmp(instruction_set[i].arg2,"1")==0)
            {
                strcpy(instruction_set[i].op,"");
                strcpy(instruction_set[i].arg2,"");
            }
        }
    }
}
void strength_reduction()
{
    for(int i=0;i!=ind;i++)
    {
        if(instruction_set[i].other[0]=='\0')
        {
            if(strcmp(instruction_set[i].op,"**")==0)
            {
                strcpy(instruction_set[i].op,"*");
                strcpy(instruction_set[i].arg2,instruction_set[i].arg1);
            }
        }
    }
}
void dead_code_elimination()
{
    int count=0;
    for(int i=0;i!=ind;i++)
    {
        if(instruction_set[i].other[0]=='\0')
        {
            for(int j=i+1;j!=ind;j++)
            {
                if(instruction_set[j].other[0]=='\0')
                {
                    if(strcmp(instruction_set[i].op,instruction_set[j].op)==0 &&
                    strcmp(instruction_set[i].arg1,instruction_set[j].arg1)==0 &&
                    strcmp(instruction_set[i].arg2,instruction_set[j].arg2)==0)
                    {


                        strcpy(instruction_set[j].op,"\0");
                        strcpy(instruction_set[j].arg1,"\0");
                        strcpy(instruction_set[j].arg2,"\0");
                        strcpy(instruction_set[j].target,"\0");
                        count=count+2;
                    }
                }
            }
        }
    }
    ind=ind-count;
}
```

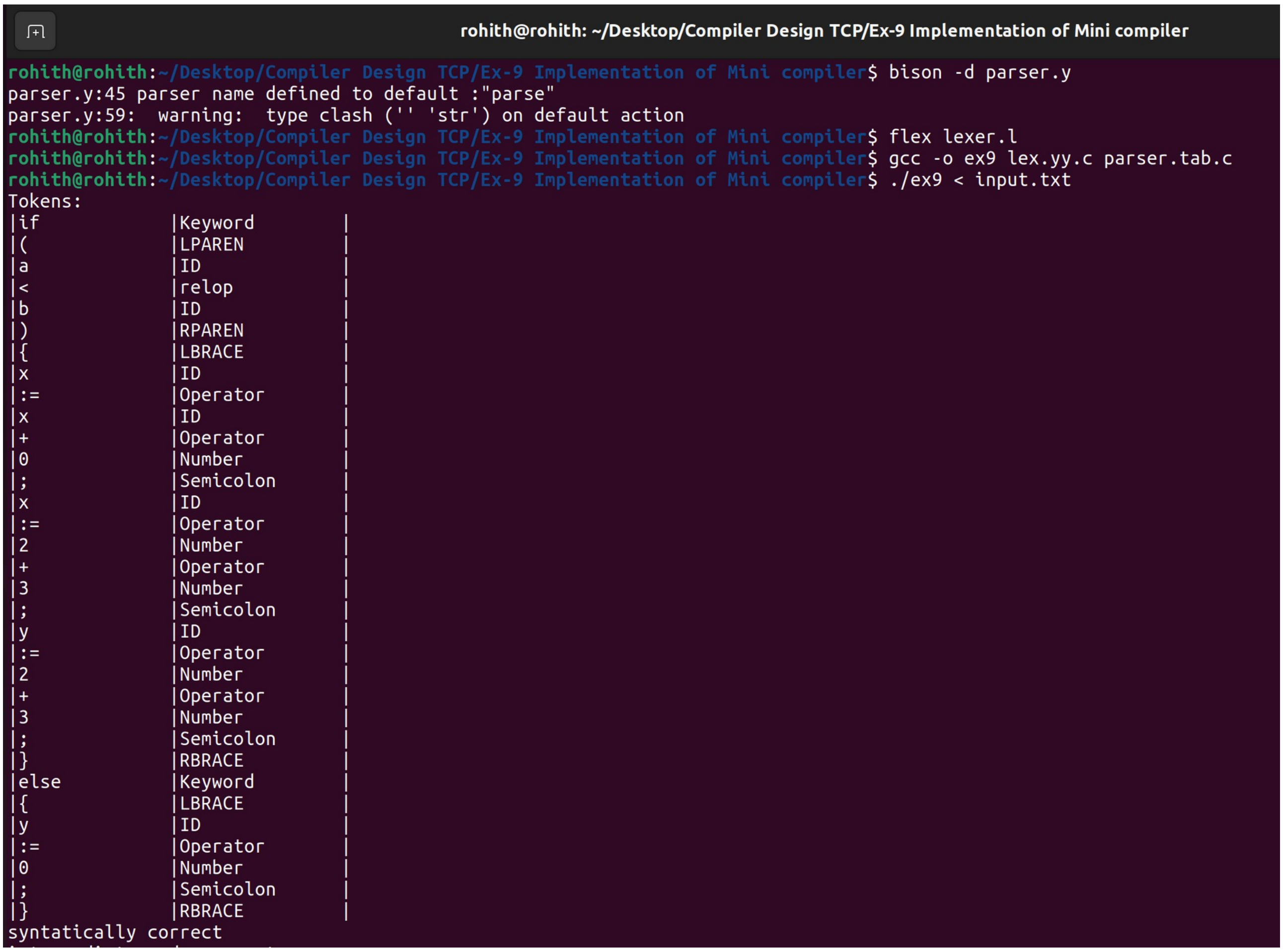
```
//code generation
void code_generation()
{
    FILE *fp=fopen("assembly_code.txt","w");
    int register_count=0;
    for(int i=0;i!=ind;i++)
    {
        if(instruction_set[i].other[0]=='\0'){
            if(strcmp(instruction_set[i].op,"\0")==0)
            {
                fprintf(fp,"MOV %s,%s\
n",instruction_set[i].target,instruction_set[i].arg1);
                printf("MOV %s,%s\
n",instruction_set[i].target,instruction_set[i].arg1);
            }
            else if(strcmp(instruction_set[i].op,"+")==0)
            {
                fprintf(fp,"MOV R%d,%s\n",register_count,instruction_set[i].arg1);
                printf("MOV R%d,%s\n",register_count,instruction_set[i].arg1);
                fprintf(fp,"ADD %s,R%d\n",instruction_set[i].arg2,register_count);
                fprintf(fp,"MOV %s,R%d\n",instruction_set[i].target,register_count);
                printf("ADD %s,R%d\n",instruction_set[i].arg2,register_count);
                printf("MOV %s,R%d\n",instruction_set[i].target,register_count);
                register_count++;
            }
            else if(strcmp(instruction_set[i].op,"*")==0)
            {
                fprintf(fp,"MOV R%d,%s\n",register_count,instruction_set[i].arg1);
                printf("MOV R%d,%s\n",register_count,instruction_set[i].arg1);
                fprintf(fp,"MUL %s,R%d\n",instruction_set[i].arg2,register_count);
                fprintf(fp,"MOV %s,R%d\n",instruction_set[i].target,register_count);
                printf("MUL %s,R%d\n",instruction_set[i].arg2,register_count);
                printf("MOV %s,R%d\n",instruction_set[i].target,register_count);
                register_count++;
            }
            else if(strcmp(instruction_set[i].op,"-")==0)
            {
                fprintf(fp,"MOV R%d,%s\n",register_count,instruction_set[i].arg1);
                printf("MOV R%d,%s\n",register_count,instruction_set[i].arg1);
                fprintf(fp,"SUB %s,R%d\n",instruction_set[i].arg2,register_count);
                fprintf(fp,"MOV %s,R%d\n",instruction_set[i].target,register_count);
                printf("SUB %s,R%d\n",instruction_set[i].arg2,register_count);
                printf("MOV %s,R%d\n",instruction_set[i].target,register_count);
                register_count++;
            }
            else if(strcmp(instruction_set[i].op,"/")==0)
            {
                fprintf(fp,"MOV R%d,%s\n",register_count,instruction_set[i].arg1);
                printf("MOV R%d,%s\n",register_count,instruction_set[i].arg1);
                fprintf(fp,"DIV %s,R%d\n",instruction_set[i].arg2,register_count);
                fprintf(fp,"MOV %s,R%d\n",instruction_set[i].target,register_count);
                printf("DIV %s,R%d\n",instruction_set[i].arg2,register_count);
                printf("MOV %s,R%d\n",instruction_set[i].target,register_count);
                register_count++;
            }
        }
        else
        {
            if(instruction_set[i].other[strlen(instruction_set[i].other)-2]==':' &&
instruction_set[i].other[0]!='i' && instruction_set[i].other[0]!='w')
            {
                printf("%s",instruction_set[i].other);
                fprintf(fp,"%s",instruction_set[i].other);
            }
            if(instruction_set[i].other[0]=='g' && instruction_set[i].other[1]=='o'
            && instruction_set[i].other[2]=='t' && instruction_set[i].other[3]=='o' )
            {
                char *temp = instruction_set[i].other + 5;
                printf("JMP %s",temp);
                fprintf(fp,"JMP %s",temp);
            }

            if(instruction_set[i].other[0]=='i' && instruction_set[i].other[1]=='f')
            {
                printf("CMP %c,%c\
n",instruction_set[i].other[3],instruction_set[i].other[7]);
                fprintf(fp,"CMP %c,%c\
n",instruction_set[i].other[3],instruction_set[i].other[7]);
                if(instruction_set[i].other[5]=='<')
                {
                    printf("JL E.true\n");
                    fprintf(fp,"JL E.true\n");
                }
                if(instruction_set[i].other[5]=='>')
                {
                    printf("JG E.true\n");
                    fprintf(fp,"JG E.true\n");
                }
            }
        }

    }
    fclose(fp);
}




int main() {
   char ch;
   int i=0;
   int j=0;
   int flag=0;
```

```
freopen("output.txt","w",stdout);
if (yyparse()==0)
{
        flag=1;
}
else
{
        flag=0;
}
fflush(stdout);
fclose(stdout);
freopen("/dev/tty", "a", stdout);
print_tokens();
if(flag==1)
{
        printf("syntatically correct\n");
}
else
{
        printf("syntatically not correct\n");
}
load_instruction();
printf("intermediate code generator\n");
for (int i = 0; i < ind; i++) {
    if (instruction_set[i].other[0] != '\0') {  // If 'other' field is not empty
        printf("%s", instruction_set[i].other);
    } else {
        // Print the parsed instruction
        printf("%s = %s %s %s\n",
               instruction_set[i].target,
               instruction_set[i].arg1,
               instruction_set[i].op,
               instruction_set[i].arg2);
    }
}

constant_folding();
algebric_identites();
strength_reduction();
//dead_code_elimination();
printf("optimized code\n");
for (int i = 0; i < ind; i++) {
    if (instruction_set[i].other[0] != '\0') {  // If 'other' field is not empty
        printf("%s", instruction_set[i].other);
    } else {
        // Print the parsed instruction
        printf("%s = %s %s %s\n",
               instruction_set[i].target,
               instruction_set[i].arg1,
               instruction_set[i].op,
               instruction_set[i].arg2);
    }
    }
    printf("Assembly code\n");
    code_generation();



    return 0;
   }
```

input.txt

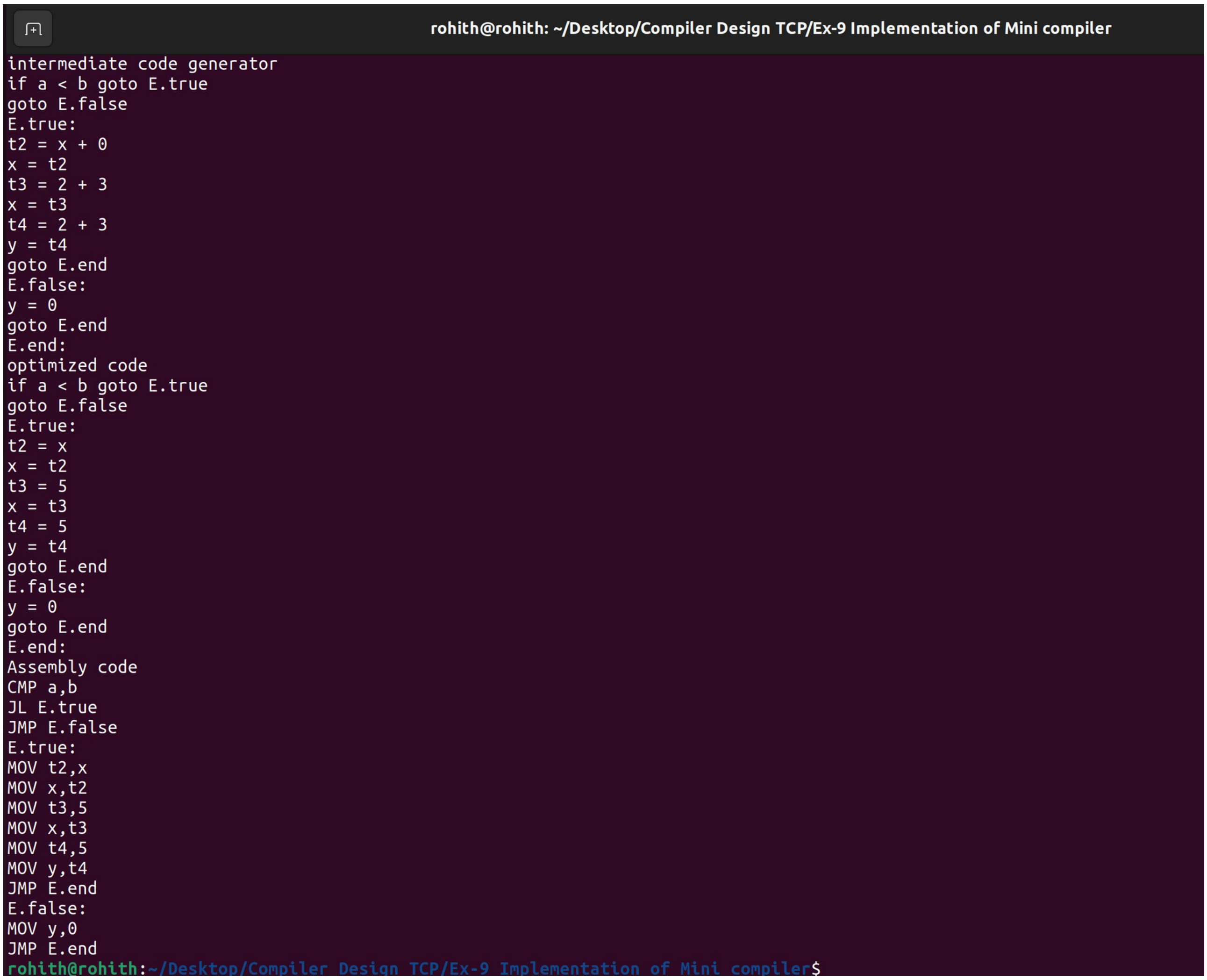
```
if(a<b)
{
x:=x+0;
x:=2+3;
y:=2+3;
}
else
{
y:=0;
}
```

Output



rohith@rohith: ~/Desktop/Compiler Design TCP/Ex-9 Implementation of Mini compiler

```
rohith@rohith:~/Desktop/Compiler Design TCP/Ex-9 Implementation of Mini compiler$ bison -d parser.y
parser.y:45 parser name defined to default :"parse"
parser.y:59:  warning:  type clash ('' 'str') on default action
rohith@rohith:~/Desktop/Compiler Design TCP/Ex-9 Implementation of Mini compiler$ flex lexer.l
rohith@rohith:~/Desktop/Compiler Design TCP/Ex-9 Implementation of Mini compiler$ gcc -o ex9 lex.yy.c parser.tab.c
rohith@rohith:~/Desktop/Compiler Design TCP/Ex-9 Implementation of Mini compiler$ ./ex9 < input.txt
Tokens:
|if            |Keyword        |
|(             |LPAREN         |
|a             |ID             |
|<             |relop          |
|b             |ID             |
|)             |RPAREN         |
|{             |LBRACE         |
|x             |ID             |
|:=            |Operator       |
|x             |ID             |
|+             |Operator       |
|0             |Number         |
|;             |Semicolon      |
|x             |ID             |
|:=            |Operator       |
|2             |Number         |
|+             |Operator       |
|3             |Number         |
|;             |Semicolon      |
|y             |ID             |
|:=            |Operator       |
|2             |Number         |
|+             |Operator       |
|3             |Number         |
|;             |Semicolon      |
|}             |RBRACE         |
|else          |Keyword        |
|{             |LBRACE         |
|y             |ID             |
|:=            |Operator       |
|0             |Number         |
|;             |Semicolon      |
|}             |RBRACE         |
syntatically correct
```



rohith@rohith: ~/Desktop/Compiler Design TCP/Ex-9 Implementation of Mini compiler

```
intermediate code generator
if a < b goto E.true
goto E.false
E.true:
t2 = x + 0
x = t2
t3 = 2 + 3
x = t3
t4 = 2 + 3
y = t4
goto E.end
E.false:
y = 0
goto E.end
E.end:
optimized code
if a < b goto E.true
goto E.false
E.true:
t2 = x
x = t2
t3 = 5
x = t3
t4 = 5
y = t4
goto E.end
E.false:
y = 0
goto E.end
E.end:
Assembly code
CMP a,b
JL E.true
JMP E.false
E.true:
MOV t2,x
MOV x,t2
MOV t3,5
MOV x,t3
MOV t4,5
MOV y,t4
JMP E.end
E.false:
MOV y,0
JMP E.end
rohith@rohith:~/Desktop/Compiler Design TCP/Ex-9 Implementation of Mini compiler$
```

assembly_code.txt

```
CMP a,b
JL E.true
JMP E.false
E.true:
MOV t2,x
MOV x,t2
MOV t3,5
MOV x,t3
MOV t4,5
MOV y,t4
JMP E.end
E.false:
MOV y,0
JMP E.end
E.end:
```

output.txt

```
if a < b goto E.true
goto E.false
E.true:
t2 = x + 0
x = t2
t3 = 2 + 3
x = t3
t4 = 2 + 3
y = t4
goto E.end
E.false:
y = 0
goto E.end
E.end:
```

Learning Outcomes:

● Understanding the key phases of a compiler (lexical analysis, syntax analysis, semantic analysis,
optimization, and code generation)
● Learn how to integrate tools like Flex for lexical analysis and Bison/Yacc for parsing to build the
components of a compiler