29/01/2021

A1: Implementation of Lexical Analyzer for the patterns - (identifier, comments, operators, constants)

185001188
Vanathi G
CSE-C

Aim -

Develop a Lexical analyzer using C to recognize the patterns namely - identifiers, constants, comments and operators - using the given regular expressions.

Program -

```
/* PROGRAM : Implementation of Lexical Analyzer for the patterns (identifier, comments,
operators, constants) */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

#define DELIMITER (c == ' ' || c == '\n' || c == ';')
#define MAX 32

typedef struct
{
  char name[MAX];
  int n;

}identifier;

int isKeyword(identifier id);

void main()
{
  char c, prev;
  int err_flag = 0, digit_seen = 0, char_seen = 0;

  // 0 for start state; revert to this state after encountering the delimiter
  int state = 0;

  identifier id;
  strcpy(id.name, "");
  id.n = 0;

  // Keep reading characters until EOF
  while((c = getchar()) != EOF)
  {
   // For every state, define what state to move to based on read character
   switch(state)
   {
    case 0:
```

```
{
  if(c == '<' || c == '>')
    state = 1;
  else if(c == '=')
    state = 2;
  else if(c == '!')
    state = 4;
  else if(c == '+' || c == '-' || c == '*' || c == '%')
    state = 5;
  else if(isdigit(c))
    state = 6;
  else if(c == '\'')
    state = 10;
  else if(c == '"')
    state = 11;
  else if(isalpha(c) || c == '_')
  {
    id.name[id.n++] = c;
    state = 14;
  }
  else if(c == '/')
    state = 15;
  else if(c == '\n')
    c = '\n';
  else
    err_flag = 1;
  break;
}
case 1:
{
  if DELIMITER
  {
    printf("RELOP ");
    state = 0;
  }
  else if(c == '=')
    state = 3;
  else
    err_flag = 1;
  break;
}
case 2:
{
  if DELIMITER
  {
    printf("ASSIGN ");
    state = 0;
  }
  else if(c == '=')
    state = 3;
  else
```

```
    err_flag = 1;
  break;
}
case 3:
{
  if DELIMITER
  {
    printf("RELOP ");
    state = 0;
  }
  else
    err_flag = 1;
  break;
}
case 4:
{
  if DELIMITER
  {
    printf("LOGOP ");
    state = 0;
  }
  else if(c == '=')
    state = 3;
  else
    err_flag = 1;
  break;
}
case 5:
{
  if DELIMITER
  {
    printf("ARITHOP ");
    state = 0;
  }
  else
    err_flag = 1;
  break;
}
case 6:
{
  if DELIMITER
  {
    printf("NUMCONST ");
    state = 0;
  }
  else if(isdigit(c))
    state = 6;
  else if(c == '.')
    state = 7;
  else if(c == 'E' || c == 'e')
    state = 8;
```

```
  else
    err_flag = 1;
  break;
}
case 7:
{
  if(DELIMITER && digit_seen == 1)
  {
    printf("NUMCONST ");
    state = 0;
    digit_seen = 0;
  }
  else if(isdigit(c))
  {
    digit_seen = 1;
    state = 7;
  }
  else
    err_flag = 1;
  break;
}
case 8:
{
  if(DELIMITER && digit_seen == 1)
  {
    printf("NUMCONST ");
    state = 0;
    digit_seen = 0;
  }
  else if(c == '+' || c == '-')
    state = 9;
  else if(isdigit(c))
  {
    digit_seen = 1;
    state = 8;
  }
  else
    err_flag = 1;
  break;
}
case 9:
{
  if(DELIMITER && digit_seen == 1)
  {
    printf("NUMCONST ");
    state = 0;
    digit_seen = 0;
  }
  else if(isdigit(c))
  {
    digit_seen = 1;
```

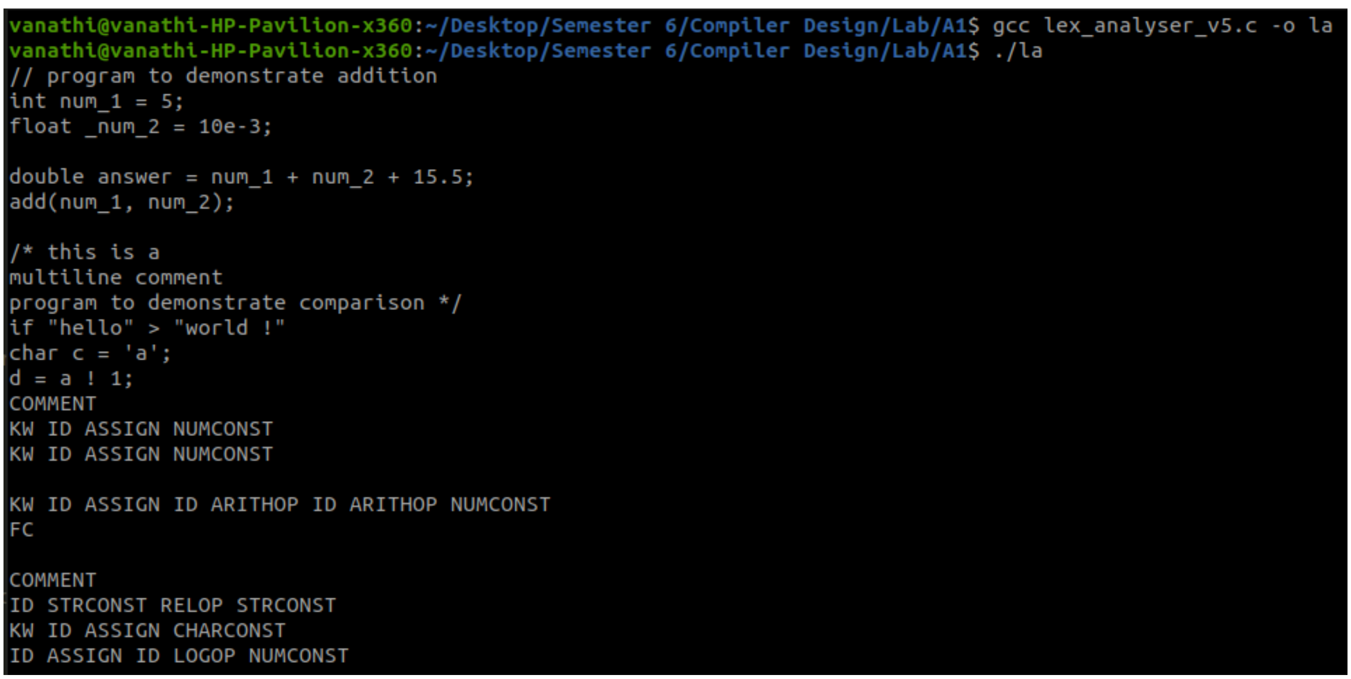
```
    state = 9;
  }
  else
    err_flag = 1;
  break;

}
case 10:
{
  if(c == '\'')
  {
    state = 12;
    char_seen = 0;
  }
  else if(c!='\n' && char_seen == 0)
  {
    char_seen = 1;
    state = 10;
  }
  else
    err_flag = 1;
  break;
}
case 11:
{
  if(c == '"')
    state = 13;
  else if(c != '\n')
    state = 11;
  else
    err_flag = 1;
  break;
}
case 12:
{
  if DELIMITER
  {
    printf("CHARCONST ");
    state = 0;
  }
  else
    err_flag = 1;
  break;
}
case 13:
{
  if DELIMITER
  {
    printf("STRCONST ");
    state = 0;
  }
```

```
  else
    err_flag = 1;
  break;
}
case 14:
{
  if DELIMITER
  {
    id.name[id.n]='\0';
    if(isKeyword(id))
      printf("KW ");
    else
      printf("ID ");
    state = 0;
    strcpy(id.name, "");
    id.n = 0;
  }
  else if(isalnum(c) || c == '_')
  {
    id.name[id.n++] = c;
    state = 14;
  }
  else if(c == '(')
    state = 19;
  else
    err_flag = 1;
  break;
}
case 15:
{
  if DELIMITER
  {
    printf("ARITHOP ");
    state = 0;
  }
  else if(c == '/')
    state = 16;
  else if(c == '*')
    state = 17;
  else
    err_flag = 1;
  break;
}
case 16:
{
  if(c == '\n')
  {
    printf("COMMENT ");
    state = 0;
  }
  else
```

```
      state = 16;
    break;
  }
  case 17:
  {
    if(c == '*')
      state = 18;
    else
      state = 17;
    break;
  }
  case 18:
  {
    if(c == '/')
    {
      printf("COMMENT ");
      state = 0;
    }
    else
      state = 17;
    break;
  }
  case 19:
  {
    if(c == ')')
      state = 20;
    else if(isalnum(c) || c == ' ' || c == '_' || c == ',')
      state = 19;
    else
      err_flag = 1;
    break;
  }
  case 20:
  {
    if DELIMITER
    {
      printf("FC ");
      state = 0;
    }
    else
      err_flag = 1;
    break;
  }

}
if(err_flag == 1)
{
  printf("Invalid token!\n");
  break;
}
if(state != 17 && c == '\n')
```

```
    printf("\n");
  }
  printf("\n");

}

// Function to check whether it is an ID or keyword
int isKeyword(identifier id)
{
  char words[32][15] = {"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"};
  for(int i=0; i<32; i++)
  {
    if(strcmp(id.name, words[i]) == 0)
      return 1;
  }
  return 0;
}
```

I/O Snapshots -



```
vanathi@vanathi-HP-Pavilion-x360:~/Desktop/Semester 6/Compiler Design/Lab/A1$ gcc lex_analyser_v5.c -o la
vanathi@vanathi-HP-Pavilion-x360:~/Desktop/Semester 6/Compiler Design/Lab/A1$ ./la
// program to demonstrate addition
int num_1 = 5;
float _num_2 = 10e-3;

double answer = num_1 + num_2 + 15.5;
add(num_1, num_2);

/* this is a
multiline comment
program to demonstrate comparison */
if "hello" > "world !"
char c = 'a';
d = a ! 1;
COMMENT
KW ID ASSIGN NUMCONST
KW ID ASSIGN NUMCONST

KW ID ASSIGN ID ARITHOP ID ARITHOP NUMCONST
FC

COMMENT
ID STRCONST RELOP STRCONST
KW ID ASSIGN CHARCONST
ID ASSIGN ID LOGOP NUMCONST
```

```
vanathi@vanathi-HP-Pavilion-x360:~/Desktop/Semester 6/Compiler Design/Lab/A1$ ./la
_hello hello hello_world
23 4.5 4e+10
'a' '$' '9'
"hello world" "???" "a string :)"
// single line comment
/* multi-line
comment */
function_call() fc_2(int a, int b)
ID ID ID
NUMCONST NUMCONST NUMCONST
CHARCONST CHARCONST CHARCONST
STRCONST STRCONST STRCONST
COMMENT
COMMENT
FC FC
```

```
vanathi@vanathi-HP-Pavilion-x360:~/Desktop/Semester 6/Compiler Design/Lab/A1$ ./la
> < >= <=
RELOP RELOP RELOP RELOP
=
ASSIGN
!
LOGOP
+ - * / %
ARITHOP ARITHOP ARITHOP ARITHOP ARITHOP
==
RELOP
```

Learning Outcomes -

1. I successfully developed a basic lexical analyser that recognizes various patterns such as constants, variables, operators, comments and function calls.
2. I learnt what a token is and why we need to convert the high-level language words to tokens before syntax analysis.
3. I understood the regular expressions used to represent these tokens and was also able to use the transition diagram to help implement the lexical analyser.
4. I found it challenging to implement the recognition of multi-line comments but I was successfully able to do it after constructing the required transitions in the transition diagram.
5. I also understood how a lexical analyser works and its functions in the compiler.

12/02/2021

A2: Implementation of Lexical Analyzer for the patterns using Lex (identifier, comments, operators, constants)

185001188
Vanathi G
CSE C

Aim :

To implement a lexical analyzer using lex for identifiers, comments, operators and constants.

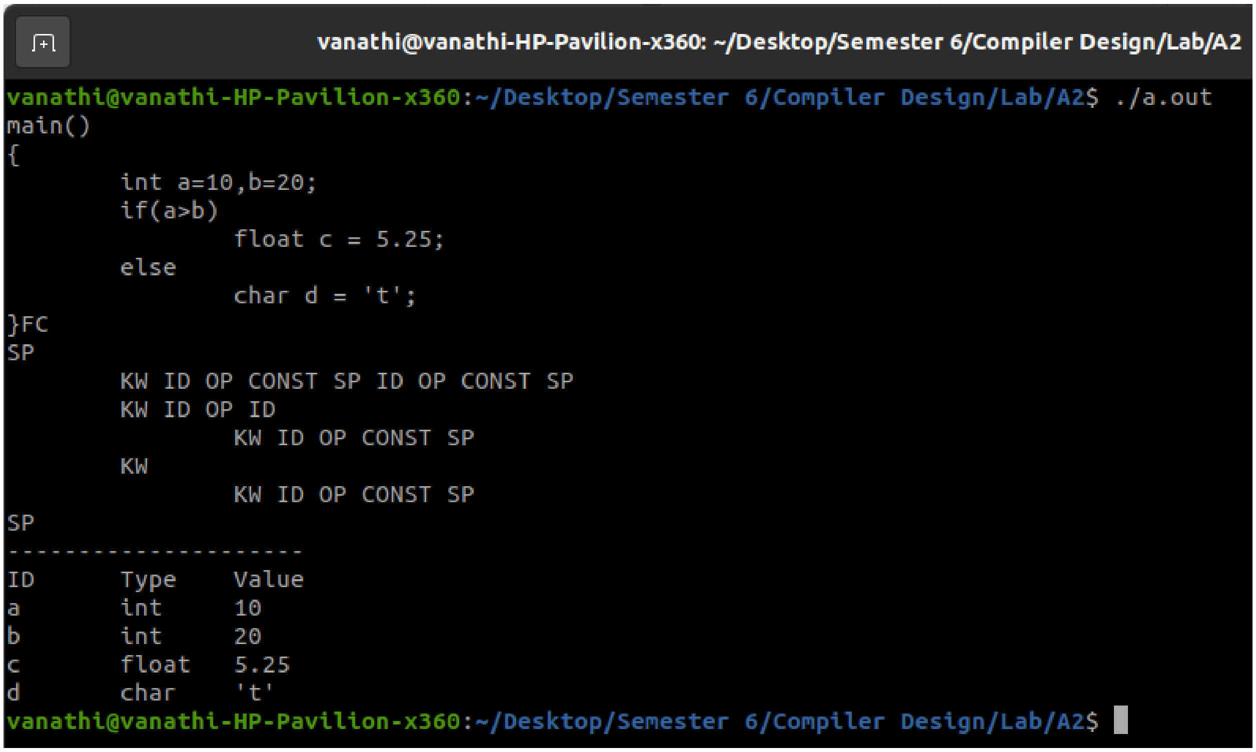
Program :

```
/*lex program to count number of words*/
%{
	#include<stdio.h>
	#include<string.h>

	typedef struct {
		char type[10];
		char varname[32];
		char init_value[32];

		}symbolTable;

	char curr_type[10];

	symbolTable st[10];
	int ptr = -1, exists = 0, i, assign_expected = 0;

%}

digit [0-9]
letter [a-zA-Z]

digits {digit}+
optFrac \.{digits}
optExp E("+"|"-")?{digits}
numberconst {digits}({optFrac})?({optExp})?

charconst \'{letter}\'
stringconst \"({letter}|" "|{digit})*\"
constant {numberconst}|{charconst}|{stringconst}

id {letter}({letter}|{digit})*(\[{digit}*\])?

start \/\/
single {start}({letter}|{digit}|" ")*

start1 \/\*
end1 \*\/
```

```
multi {start1}({letter}|{digit}|"\n"|" ")*{end1}

relop "<"|"<="|"=="|"!="|">"|">="
arithop "+"|"-"|"*"|"/"|"%"
logicalop "&&"|"||"|"!"
assignop "="

sp ","|";"|"{"|"}"

operator {relop}|{arithop}|{logicalop}|{assignop}

keyword
("auto"|"break"|"case"|"char"|"const"|"continue"|"default"|"do"|"double"|"else"|"enum"|"extern"|"flo
at"|"for"|"goto"|"if"|"int"|"long"|"register"|"return"|"short"|"signed"|"sizeof"|"static"|"struct"|"switch"|"t
ypedef"|"union"|"unsigned"|"void"|"volatile"|"while")

function ("printf"|"main")


/* Rules Section*/
%%
{single} {printf("SINGLE-LINE COMMENT ");}
{multi} {printf("MULTI-COMMENT ");}

{constant} {
	printf("CONST ");
	if(assign_expected == 1)
	{
		strcpy(st[ptr].init_value, yytext);
		assign_expected = 0;
	}
}

{keyword} {
	printf("KW ");
	if(strcmp(yytext, "int") == 0 || strcmp(yytext, "float") == 0 || strcmp(yytext, "double") == 0 ||
	strcmp(yytext, "char") == 0)
		strcpy(curr_type, yytext);
}
{function} {printf("FC ");}
{id} {
	printf("ID ");
	exists = 0;
	for(i=0; i<=ptr; i++)
	{
		if(strcmp(st[i].varname, yytext) == 0)
			exists = 1;
	}
	if(exists == 0)
	{
		ptr++;
```

```
                    strcpy(st[ptr].type, curr_type);
                    strcpy(st[ptr].varname, yytext);
                    strcpy(st[ptr].init_value, "");
          }

}

{operator} {
          printf("OP ");
          if(yytext[0] == '=')
                    assign_expected = 1;

}
{sp} {printf("SP ");}

["\n"] {printf("\n");}

[" "|"("|")"] {};
%%

int yywrap(void){return 1;}

int main()
{
   // The function that starts the analysis
   yylex();

   printf("\n--------------------\nID\tType\tValue\n");
   for(int i=0; i<=ptr; i++)
   {
         printf("%s\t%s\t%s\n", st[i].varname, st[i].type, st[i].init_value);

   }

   return 0;
}
```

I/O Snapshot -



Learning Outcomes:

1. I successfully developed a basic lexical analyser using lex that recognizes various patterns such as identifiers, comments, operators and constants.
2. I understood the regular expressions used to represent these tokens and how to denote them in the lex tool.
3. The most challenging part of this assignment was identifying the difference between keywords and functions.
4. I also understood how a lexical analyser works and its functions in the compiler.

19/02/2021

A3: Elimination of Immediate Left Recursion using C

185001188
Vanathi G
CSE C

Aim -

To write a C program that eliminates the left recursion in a given input grammar.

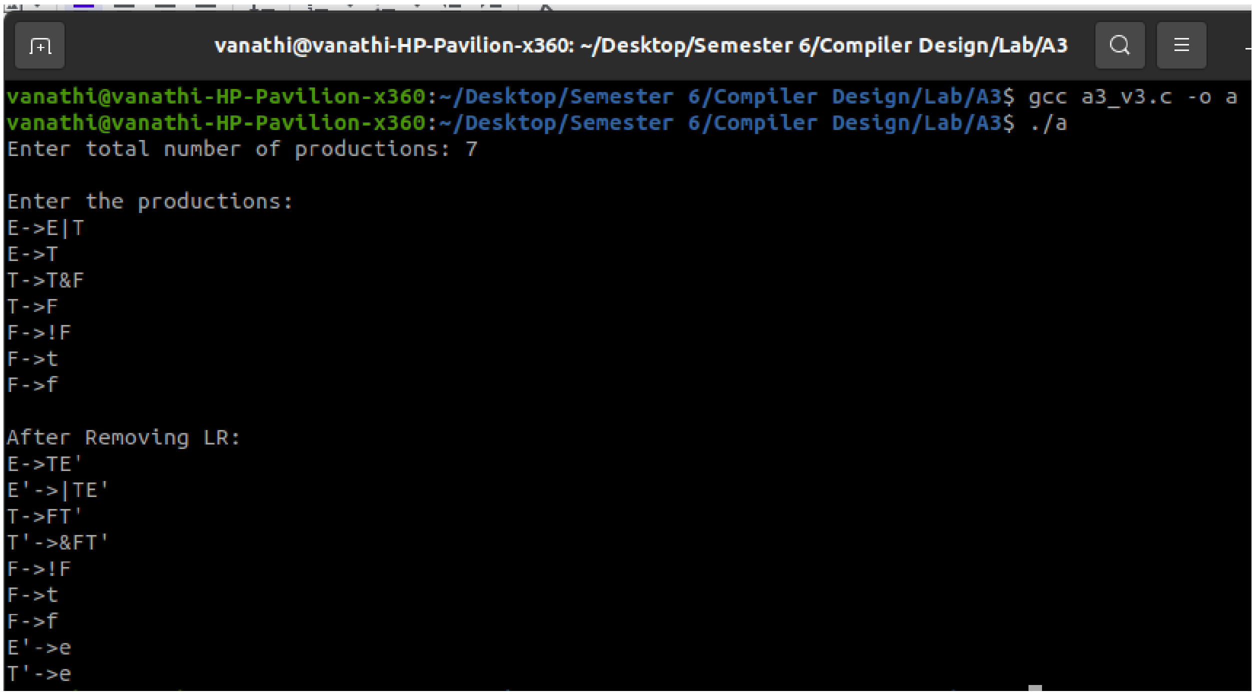
Program -

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

#define MAXPROD 50
#define MAXLEN 20

void main()
{
	char grammar[MAXPROD][MAXLEN], prod[MAXLEN], temp[3];
	char symbol;
	int n, i, j, l, ptr, posn;
	int rec[MAXPROD], rec_c=0;

	// Input and Output of productions of the grammar

	printf("Enter total number of productions: ");
	scanf("%d", &n);
	printf("\nEnter the productions:\n");

	for(i=0; i<n; i++)
		scanf("%s", grammar[i]);

	// Checking whether the grammar is left recursive

	for(i=0; i<n; i++)
	{
		l = strlen(grammar[i]);
		if(grammar[i][0] == grammar[i][3])
			rec[rec_c++] = i;
	}

	/* Removing left recursion:
		1. A -> beta is already a production, we just have to append A' to the end of it
		2. Modifying A -> A(alpha) to A'->(alpha)A' can  be done in-place
		3. Add new production A' -> e after this
	*/
```

```
	char epsilon[] = "->e";

	for(i=0; i<rec_c; i++)
	{
		posn = rec[i];
		strcpy(prod, grammar[posn]);
		l = strlen(prod);
		symbol = prod[0];

		for(j=0; j<n; j++)
		{
			if(j!=posn && grammar[j][0] == symbol)
				break;
		}

		// new symbol
		temp[0] = symbol;
		temp[1] = '\'';
		temp[2] = '\0';

		strcat(grammar[j], temp);
		strcpy(grammar[posn], grammar[j]);

		grammar[j][1] = '\'';
		grammar[j][2] = '-';
		grammar[j][3] = '>';

		for(ptr=4; ptr<l; ptr++)
			grammar[j][ptr] = prod[ptr];
		grammar[j][ptr] = '\0';
		strcat(grammar[j], temp);

		grammar[n][0] = '\0';
		strcpy(grammar[n], temp);
		strcat(grammar[n], epsilon);
		n++;
	}
	printf("\nAfter Removing LR:\n");
	for(i=0; i<n; i++)
		printf("%s\n", grammar[i]);

}

/* productions :
E->E|T
E->T
T->T&F
T->F
F->!F
F->t
```

F->f
*/

I/O Snapshot -



Learning Outcomes -

1. I successfully developed a C program that eliminates left recursion in the given grammar (input as a set of productions).
2. I used the conversion method we learnt in the theory classes to implement this.I
3. I also understood the need for left recursion elimination in a given grammar.

26/02/2021

A4: Recursive Descent Parser using C

185001188
Vanathi G
CSE-C

AIM -

To write a C program that implements a recursive descent parser for the given grammars:

1. G: E -> E+T|T
   T -> T*F | F
   F -> i
2. G: E -> E+T|E-T|T
   T -> T*F | T/F|F
   F -> (E)|i

PROGRAMS -

```
/* G:   E -> E+T|T
      T -> T*F | F
      F -> i
*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAXLEN 40

typedef struct
{
	int ptr; // points to "lookahead" basically
	char string[MAXLEN];
}buffer;

void E(buffer *inp); // need pointers as inp because we will have to modify lookahead ptr
void T(buffer *inp);
void EPrime(buffer *inp);
void TPrime(buffer *inp);
void F(buffer *inp);

void main()
{
	buffer *inp;
	inp = malloc(sizeof(buffer));
	inp->ptr = 0;
	scanf("%s", inp->string);
	strcat(inp->string, "$");

	E(inp);
	if(inp->string[inp->ptr] == '$')
		printf("Success\n");
```

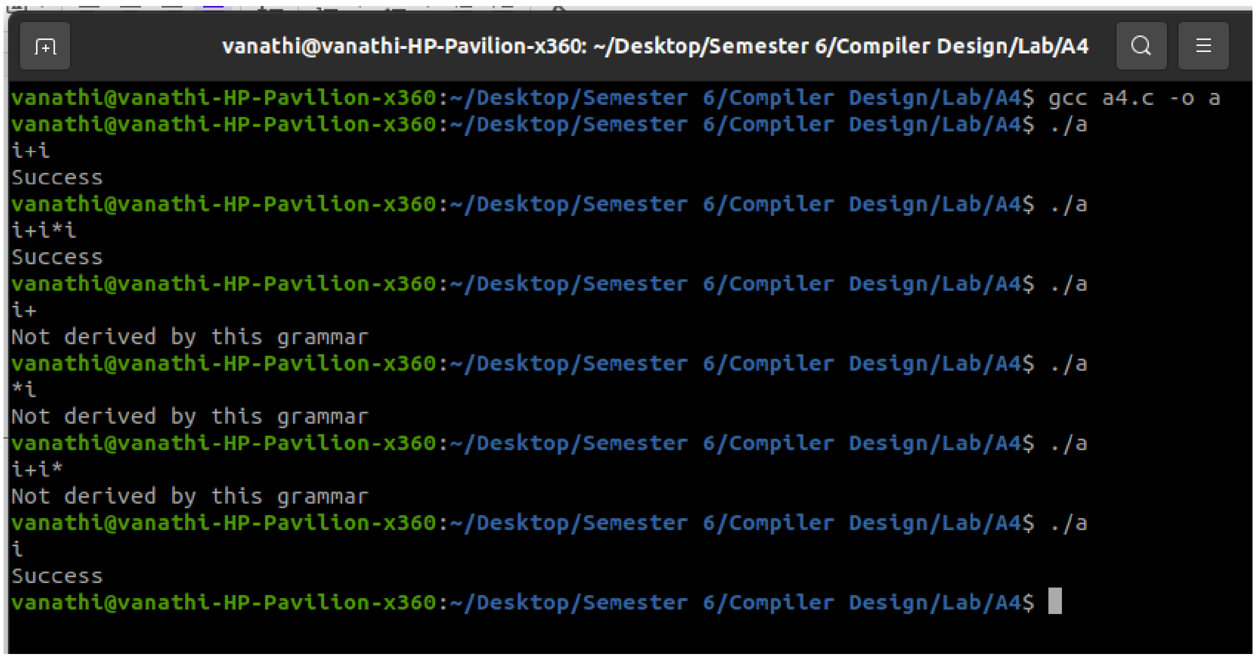
```
	else
		printf("Not derived by this grammar\n");
}

// E -> TE'
void E(buffer *inp)
{
	T(inp);
	EPrime(inp);
}

//T -> FT'
void T(buffer *inp)
{
	F(inp);
	TPrime(inp);
}

//E' -> +TE' | epsilon
void EPrime(buffer *inp)
{
	// if the current symbol is + we need to call T and E'
	if(inp->string[inp->ptr] == '+')
	{
		inp->ptr++;
		T(inp);
		EPrime(inp);
	}
	// otherwise for epsilon we just return
	return;
}

//T' -> *FT' | epsilon
void TPrime(buffer *inp)
{
	if(inp->string[inp->ptr] == '*')
	{
		inp->ptr++;
		F(inp);
		TPrime(inp);
	}
	return;
}

//F -> i
void F(buffer *inp)
{
	if(inp->string[inp->ptr] == 'i')
		inp->ptr++;
	else
	{
```

```
                printf("Not derived by this grammar\n");
                exit(0);
        }
        return;
}

/* G:   E -> E+T|E-T|T
     T -> T*F | T/F|F
     F -> (E)|i
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAXLEN 40

typedef struct
{
        int ptr; // points to "lookahead" basically
        char string[MAXLEN];
}buffer;

void E(buffer *inp); // need pointers as inp because we will have to modify lookahead ptr
void T(buffer *inp);
void EPrime(buffer *inp);
void TPrime(buffer *inp);
void F(buffer *inp);

void main()
{
        buffer *inp;
        inp = malloc(sizeof(buffer));
        inp->ptr = 0;
        scanf("%s", inp->string);
        strcat(inp->string, "$");

        E(inp);
        if(inp->string[inp->ptr] == '$')
                printf("Success\n");
        else
                printf("Not derived by this grammar\n");
}

// E -> TE'
void E(buffer *inp)
{
        T(inp);
        EPrime(inp);
}

//T -> FT'
```

```
void T(buffer *inp)
{
	F(inp);
	TPrime(inp);
}

//E' -> +TE' | epsilon
void EPrime(buffer *inp)
{
	// if the current symbol is + we need to call T and E'
	if(inp->string[inp->ptr] == '+' || inp->string[inp->ptr] == '-')
	{
		inp->ptr++;
		T(inp);
		EPrime(inp);
	}
	// otherwise for epsilon we just return
	return;
}

//T' -> *FT' | epsilon
void TPrime(buffer *inp)
{
	if(inp->string[inp->ptr] == '*' || inp->string[inp->ptr] == '/')
	{
		inp->ptr++;
		F(inp);
		TPrime(inp);
	}
	return;
}

//F -> i
void F(buffer *inp)
{
	if(inp->string[inp->ptr] == 'i')
		inp->ptr++;
	else if(inp->string[inp->ptr] == '(')
	{
		inp->ptr++;
		E(inp);
		if(inp->string[inp->ptr] == ')')
			inp->ptr++;
		else
		{
			printf("Not derived by this grammar\n");
			exit(0);
		}

	}
	else
```

```
	{
		printf("Not derived by this grammar\n");
		exit(0);
	}
	return;
}
```

I/O SNAPSHOTS -



Learning Outcomes -

1. I successfully developed C programs that implement recursive descent parsers for the given grammars.
2. I used the method we learnt in the theory classes to implement this i.e by creating functions for each production and keeping track of the lookahead token.
3. Given an input string, the parser will be able to check whether the given grammar derives the string or not by parsing it.
4. I found the implementation of the parenthesis production in the second grammar quite challenging.
5. I learnt how recursive descent parsers work while doing this assignment.

11/03/2021

A5: Implementation of Desk Calculator using YACC Tool

185001188
Vanathi G
CSE-C

Aim -

To implement a desk calculator that performs basic arithmetic operations using the YACC tool.

Code -

a5.l :

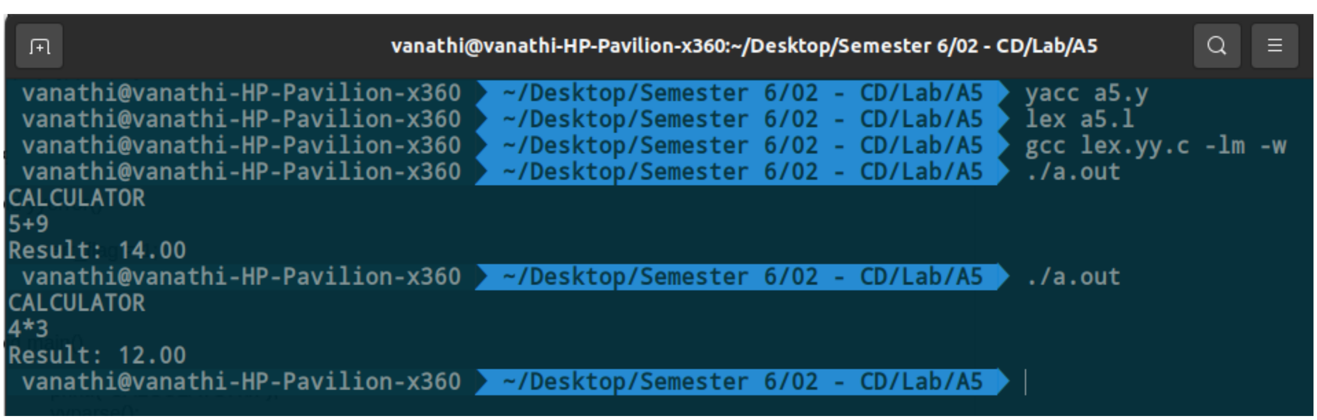
```
%{
	#include <stdio.h>
	#include "y.tab.c"
	extern YYSTYPE yylval;
%}

%%

[0-9]+ {yylval = atoi(yytext); return NUM;}
[\t] ;
[\n] return 0;
. return yytext[0];

%%

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

a5.y :

```
%{
	#include <stdio.h>
	#include <math.h>
	#define YYSTYPE double
	void yyerror();

	int err_flag = 0;

%}
%token NUM
/* for prec., first declared = lowest prec and left, right used to specify associativity */
%left '|'
%left '&'
%left '+' '-'
%left '*' '/'
```

```
%right '^'
%right '!'
%left '(' ')'
%%

S : E {printf("Result: %.2f\n",$$);}
E : E'+'E {$$ = $1 + $3;}
  | E'-'E {$$ = $1 - $3;}
  | E'*'E {$$ = $1 * $3;}
  | E'/'E {$$ = $1 / $3;}
  | E'^'E {$$ = pow($1, $3);}
  | E'&'E {$$ = $1 && $3;}
  | E'|'E {$$ = $1 || $3;}
  | '!'E {$$ = !$2;}
  | '('E')' {$$ = $2;}
  | NUM {$$ = $1;}

%%

void yyerror()
{
	err_flag = 1;
	return;
}

void main()
{
	printf("CALCULATOR\n");
	yyparse();
	if(err_flag)
	{
		printf("Enter numbers and operators only!\n");
	}

}
```

Output Screenshot -



Learning Outcomes -

1. I successfully developed a yacc program that implements a desk calculator for performing basic arithmetic operations.
2. I understood how yacc parses the input and it’s stack i.e. how to access the stack, perform operations and store the result in the stack.
3. I learnt how to associate rules with each production in the grammar in YACC.

02/04/2021

A6: Implementation of Syntax Checker Using Yacc Tool

185001188
Vanathi G
CSE-C

Aim -

To implement a syntax checker that checks whether the given input code is syntactically correct with respect to the grammar.

Programs -

a6_v3.y:

```
%{
	#include <stdio.h>
	#include <math.h>
	#define YYSTYPE double
	void yyerror();

	int err_flag = 0;

%}

%token ID
%token IF
%token NUM
%token ELSE
%token RELOP
%token FOR
%token UNOP

%%
P : S {printf("Syntactically Correct!\n");}
S : D
  | '{'B'}'
  | F
  | I
B : B S
  | S
F : FOR'('A';'C';'O')'S
O : A
  | U
I : IF'('C')'S ELSE S
C : ID RELOP ID
  | ID RELOP NUM
  | NUM RELOP ID
  | NUM RELOP NUM
A : ID'='E
D : ID'='E';'
```

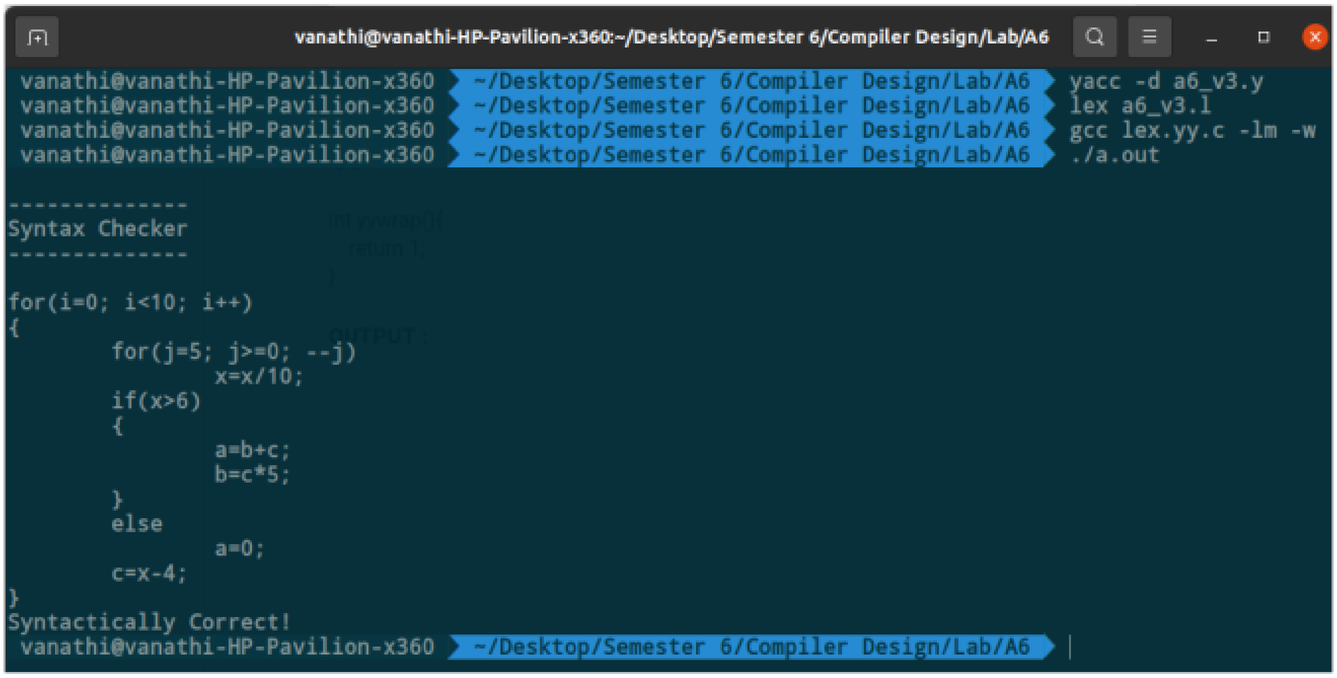
```
E : E'+'T
  | E'-'T
  | T
T : T'*'F
  | T'/'F
  | F
F : ID
  | NUM
U : ID UNOP
  | UNOP ID
  ;

%%
void yyerror()
{
	//err_flag = 1;
	return;
}

void main()
{
	printf("\n--------------\nSyntax Checker\n--------------\n\n");
	yyparse();

}
```

<u>a6_v3.l:</u>

```
%{
	#include <stdio.h>
	#include "y.tab.c"
	extern YYSTYPE yylval;
%}

relop "<"|"<="|"=="|"!="|">"|">="
unop "++"|"--"

%%
"if" {return IF;}
"else" {return ELSE;}
"for" {return FOR;}
{relop} {return RELOP;}
{unop} {return UNOP;}
([a-zA-Z])([a-zA-Z]|[0-9])* {return ID;}
[0-9]+ {return NUM;}
[\n] { }
[\t] { }
[' '] { }
. return yytext[0];

%%
```

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

Output :



```
vanathi@vanathi-HP-Pavilion-x360  ~/Desktop/Semester 6/Compiler Design/Lab/A6  yacc -d a6_v3.y
vanathi@vanathi-HP-Pavilion-x360  ~/Desktop/Semester 6/Compiler Design/Lab/A6  lex a6_v3.l
vanathi@vanathi-HP-Pavilion-x360  ~/Desktop/Semester 6/Compiler Design/Lab/A6  gcc lex.yy.c -lm -w
vanathi@vanathi-HP-Pavilion-x360  ~/Desktop/Semester 6/Compiler Design/Lab/A6  ./a.out

--------------
Syntax Checker
--------------

for(i=0; i<10; i++)
{
	for(j=5; j>=0; --j)
		x=x/10;
	if(x>6)
	{
		a=b+c;
		b=c*5;
	}
	else
		a=0;
	c=x-4;
}
Syntactically Correct!
vanathi@vanathi-HP-Pavilion-x360  ~/Desktop/Semester 6/Compiler Design/Lab/A6 
```

Learning Outcomes -

1. I successfully developed a yacc program that implements a syntax checker that checks whether the given input code is syntactically correct.
2. I understood what the syntax analysis phase of the compiler is and the need for it.
3. I found the integration of the syntax checking for each type of statement into one grammar quite challenging to implement.

12/04/2021

A7: GENERATION OF INTERMEDIATE CODE USING LEX AND YACC

Vanathi G
185001188
CSE C

AIM :

To generate intermediate code (or three address code - TAC) for a given code segment using lex and yacc programs.

CODE :

a7_v5.y:

```
%{
   #include <stdio.h>
   #include <stdlib.h>
   #include <math.h>
   void yyerror();

   struct info
   {
          char var[10];
          char code[100];
          char true[10];
          char false[10];
          char out[10];

   };

   void newTemp(int count, char *var)
   {
          char str_count[3];
          char varname[] = "t";
          sprintf(str_count, "%d", count);
          strcat(varname, str_count);
          strcpy(var, varname);
   }

   void newLabel(int count, char *label)
   {
          char str_count[3];
          char labelname[] = "L";
          sprintf(str_count, "%d", count);
          strcat(labelname, str_count);
          strcpy(label, labelname);
   }

   struct info* makeNode(int count, char type)
   {
```

```
		struct info *temp;
		temp = malloc(sizeof(struct info));

		if(type == 't')
		{
			newTemp(count, temp->var);
		}
		else
		{
			newLabel(count, temp->true);
			if(type == 'i')
				newLabel(count+1, temp->out);
			else {
				newLabel(count+1, temp->false);
				newLabel(count+2, temp->out);
			}

		}

		strcpy(temp->code, "");

		return temp;
	}

	int err_flag = 0;
	int tempvar_count=0;
	int label_count=0;

%}

%union {
	struct info *node;
	char name[50];
	char keyword;
}

%token <keyword> IF THEN ELSE ENDIF BEG END TYPE
%token <name> ID RELOP CONST
%type <node> AS I T F E

%%
START : DECL PROGRAM
	| DECL
	| PROGRAM

DECL : DECL D
	| D

D : ID':'TYPE';'
  | ID':'TYPE'='CONST';'
```

```
PROGRAM : BEG B END

B : B S
  | S

S : AS {printf("%s", $1->code);}
  | I {printf("%s", $1->code);}

I : IF'('ID RELOP ID')' THEN AS ELSE AS ENDIF{
                        $$ = makeNode(label_count, 'e');
                        label_count += 3;

                        char if_code[30];
                        sprintf(if_code, "\tif %s %s %s goto %s\n\tgoto %s\n", $3, $4, $5, $$->true,
$$->false);

                        sprintf($$->code, "%s%s:%s\tgoto %s\n%s:%s%s:", if_code, $$->true,
$8->code, $$->out, $$->false, $10->code, $$->out);
            }
  | IF'('ID RELOP ID')' THEN AS ENDIF{
                        $$ = makeNode(label_count, 'i');
                        label_count += 2;

                        char if_code[30];
                        sprintf(if_code, "\tif %s %s %s goto %s\n\tgoto %s\n", $3, $4, $5, $$->true,
$$->out);

                        sprintf($$->code, "%s%s:%s%s:", if_code, $$->true, $8->code, $$->out);
            }

AS : ID'='E';'{
                        $$ = makeNode(0, 't');
                        sprintf($$->code, "%s\t%s = %s\n", $3->code, $1, $3->var);
            }

E : T'*'E{
                        $$ = makeNode(tempvar_count, 't');
                        tempvar_count++;
                        sprintf($$->code, "%s%s\t%s = %s * %s\n", $1->code, $3->code, $$->var,
$1->var, $3->var);
            }

  | T'/'E{
                        $$ = makeNode(tempvar_count, 't');
                        tempvar_count++;
                        sprintf($$->code, "%s%s\t%s = %s / %s\n", $1->code, $3->code, $$->var,
$1->var, $3->var);
            }

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

```
T : T'+'F{
                    $$ = makeNode(tempvar_count, 't');
                    tempvar_count++;
                    sprintf($$->code, "%s%s\t%s = %s + %s\n", $3->code, $1->code, $$->var,
$1->var, $3->var);
          }

  | T'-'F{
                    $$ = makeNode(tempvar_count, 't');
                    tempvar_count++;
                    sprintf($$->code, "%s%s\t%s = %s - %s\n", $3->code, $1->code, $$->var,
$1->var, $3->var);
                    if(strlen($3->code) > 0)
                              strcat($$->code, $3->code);
          }

  | F {$$ = $1;}

F : ID {$$ = makeNode(0, 't'); strcpy($$->var, $1);}
  ;

%%

void yyerror()
{
   return;
}

int main()
{
   printf("----------------------------\nINTERMEDIATE CODE
GENERATION\n----------------------------\n");

   FILE *fp = fopen("input.txt", "r");
   char c = fgetc(fp);
         while (c != EOF)
         {
         printf ("%c", c);
         c = fgetc(fp);
         }
         fclose(fp);

   printf("\n----------------------------\nGENERATED CODE\n----------------------------\n");

   yyparse();
   printf("\n");
   return 0;
}
```

<u>a7_v5.l:</u>

```
%{
```

```
    #include <stdio.h>
    #include "y.tab.c"
%}

letter [a-zA-Z]
digit [0-9]
relop "<"|"<="|"=="|"!="|">"|">="
type "integer"|"real"|"char"

digits {digit}+
optFrac \.{digits}
optExp E("+"|"-")?{digits}
numberconst {digits}({optFrac})?({optExp})?

charconst \'{letter}\'

constant {numberconst}|{charconst}

%%

"if" {return IF;}
"then" {return THEN;}
"else" {return ELSE;}
"endif" {return ENDIF;}
"begin" {return BEG;}
"end" {return END;}
{type} {yylval.keyword = yytext[0]; return TYPE;}

{constant} {strcpy(yylval.name, yytext); return CONST;}
{relop} {strcpy(yylval.name, yytext); return RELOP;}
{letter}({letter}|{digit})* {strcpy(yylval.name, yytext); return ID;}
[' '] { };
['\t'] { };
['\n'] { };
. return yytext[0];

%%

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

OUTPUT :

```
-----------------------------
GENERATED CODE
-----------------------------
        if p < q goto L0
        goto L1
L0:     t0 = b + c
        t1 = p * t0
        a = t1
        goto L2
L1:     t2 = b - c
        t3 = q / t2
        a = t3
L2:     if ch == op goto L3
        goto L4
L3:     t4 = b * c
        t5 = a * t4
        p = t5
L4:     t6 = a + b
        t7 = t6 + c
        q = t7

 vanathi@vanathi-HP-Pavilion-x360  ~/Desktop/Semester 6/Compiler Design/Lab/A7
```

```
vanathi@vanathi-HP-Pavilion-x360:~/Desktop/Semester 6/Compiler Design/Lab/A7
 vanathi@vanathi-HP-Pavilion-x360  ~/Desktop/Semester 6/Compiler Design/Lab/A7  yacc a7_v5.y
 vanathi@vanathi-HP-Pavilion-x360  ~/Desktop/Semester 6/Compiler Design/Lab/A7  lex a7_v5.l
 vanathi@vanathi-HP-Pavilion-x360  ~/Desktop/Semester 6/Compiler Design/Lab/A7  gcc lex.yy.c -lm -w
 vanathi@vanathi-HP-Pavilion-x360  ~/Desktop/Semester 6/Compiler Design/Lab/A7  ./a.out < input.txt
---------------------------
INTERMEDIATE CODE GENERATION
---------------------------
a:integer;
b:real=5.253;
c:real=15;
ch:char='y';
op:char='5';
p:integer=10;
q:integer=11;

begin
if(p < q) then
        a = p * b + c;
else
        a = q / b - c;
endif
if(ch == op) then
        p = a * b * c;
endif
q = a + b + c;
end
```

Learning Outcomes -

1. I successfully wrote lex and yacc programs that generate intermediate code for the given code (written using the syntax of the newly created language PASCAL 2021).
2. I learnt what TAC / intermediate code is and why it is useful to generate it during this phase of the compiler.
3. I understood the concept of SDTs properly and used the rules used with each production to generate intermediate code.
4. I found it challenging to implement rules for different types of if statements (for example, if without the else part

23/04/2021

A8: Code Optimization Using C

Vanathi G
185001188
CSE C

Aim -

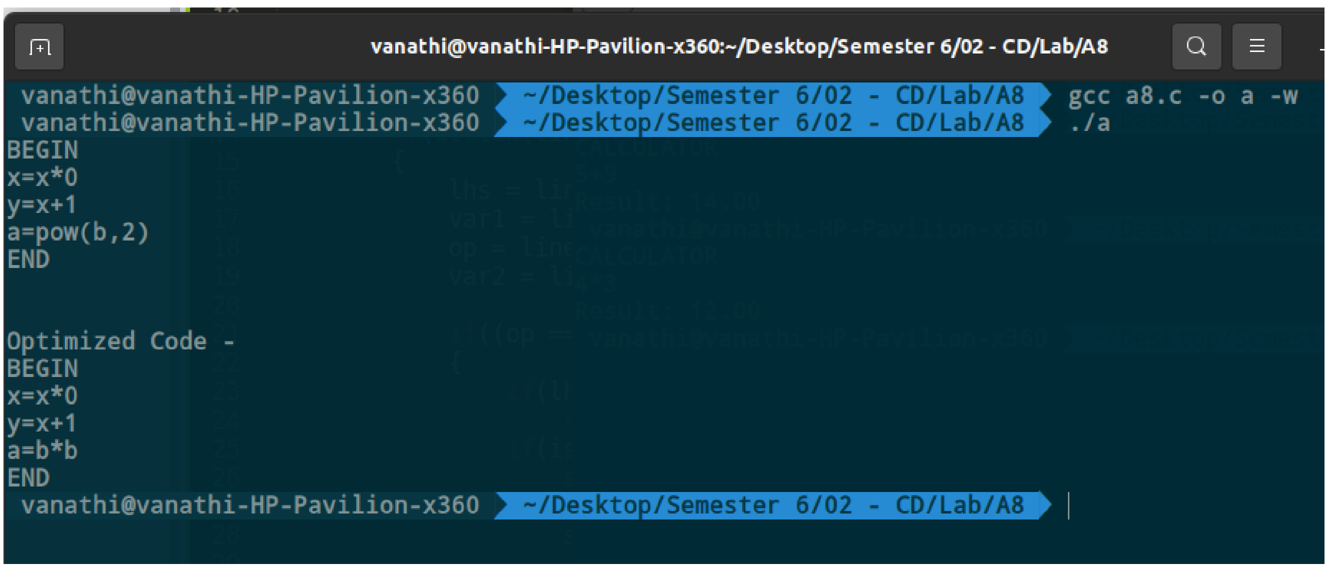
To implement a C program that performs algebraic and strength reduction types of code optimization for the given input code snippet.

Program -

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

void main(){

    char optimized[1024], line[10], opt_line[10], lhs, var1, var2, op;
    strcpy(optimized, "");

    do{
        scanf(" %s[^\n]", line);
        //for algebraic identities: x=0+x, x=1*x
        if(strlen(line) == 5)
        {
            lhs = line[0];
            var1 = line[2];
            op = line[3];
            var2 = line[4];

            if((op == '+' && (var1 == '0' || var2 == '0')) || (op == '*' && (var1 == '1' || var2 == '1')))
            {
                if(lhs == var1 || lhs == var2)
                    continue;
                if(isdigit(var1))
                    sprintf(opt_line, "%c=%c\n", lhs, var2);
                else
                    sprintf(opt_line, "%c=%c\n", lhs, var1);

            }
            else
                sprintf(opt_line, "%s\n", line);
        }
        else if (line[3] == '*' && line[4] == '2'){
            sprintf(opt_line, "%c=%c+%c\n", line[0], line[2], line[2]);
        }
        else if (line[3] == '*' && line[2] == '2'){
```

```
            sprintf(opt_line, "%c=%c+%c\n", line[0], line[4], line[4]);
        }
        else if (line[2] == 'p' && line[3] == 'o' && line[4] == 'w' && line[5] == '(' && line[8] == '2'){
            sprintf(opt_line, "%c=%c*%c\n", line[0], line[6], line[6]);
        }
        else{
            sprintf(opt_line, "%s\n", line);
        }
        strcat(optimized, opt_line);
    }while (strcmp(line, "END") != 0);

    printf("\n\nOptimized Code -\n");
    printf("%s", optimized);
}
```

Output -



Learning Outcomes -

1. I successfully performed algebraic code optimization and strength reduction for the given code snippet using a C program.
2. I learnt why it is useful to perform code optimization during this phase of the compiler.
3. I found it challenging to handle the different orders in which variables and algebraic identities could occur in the input during the optimization process.