**UCS 1602 - Compiler Design**

**Record**

Sharvan Ram Kumaran

CSE-C

185001143

**Exercise 1**

**Aim**:

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

**Code**:

// Lexical analyser - scans code and recognizes tokens

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#include <fcntl.h>

#include <stdbool.h>

int isOperator(char ch){

if (ch == '+' || ch == '-' || ch == '\*' || ch == '/' || ch == '%'){

return 1;

}

else if (ch == '>' || ch == '<'){

return 2;

}

else if(ch == '|' || ch == '&'){

return 3;

}

else if(ch == '='){

return 4;

}

return 0;

}

bool isKeyword(char \*str){

if(!strcmp(str, "if") || !strcmp(str, "else") || !strcmp(str, "while") ||

!strcmp(str, "for") || !strcmp(str, "do") || !strcmp(str, "break") ||

!strcmp(str, "switch") || !strcmp(str, "continue") || !strcmp(str, "return") ||

!strcmp(str, "case") || !strcmp(str, "default") || !strcmp(str, "void") ||

!strcmp(str, "int") || !strcmp(str, "char") || !strcmp(str, "bool") ||

!strcmp(str, "struct") || !strcmp(str, "goto") || !strcmp(str, "typedef") ||

!strcmp(str, "unsigned") || !strcmp(str, "long") || !strcmp(str, "short") ||

!strcmp(str, "float") || !strcmp(str, "double") || !strcmp(str, "sizeof")){

return true;

}

return false;

}

bool isSeparator(char ch){

if(ch=='{' || ch=='}' || ch==';' || ch=='(' || ch==')' || ch==','){

return true;

}

return false;

}

bool isFunc(char \*str){

if(strcmp(str,"main")==0 || strcmp(str,"printf")==0 || strcmp(str,"scanf")==0)

{

return true;

}

return false;

}

void lexanalyse(char \*input){

int i=0,j=0;

char ch,str[100];

for(i=0;i<strlen(input);i++){

ch = input[i];

if(ch=='#'){

printf("PDIR ");

while(input[i]!='\n'){

i++;

}

}

if(ch=='/'){

if(input[i+1]=='/'){

printf("SNGLINE ");

i+=2;

while(input[i]!='\n'){

i++;

}

}

else if(input[i+1]=='\*'){

i+=2;

printf("MLTLINE ");

while(input[i]!='\*' && input[i+1]!='/'){

i++;

}

}

}

int op = isOperator(ch);

if(op==4){

ch = input[++i];

if(ch=='=' || ch=='!'){

printf("RELOP ");

}

else if(ch==' '){

printf("ASSIGN ");

}

}

else if(op==2){

ch = input[++i];

if(ch=='=' || ch == ' ' || ch == '!'){

printf("RELOP ");

}

}

else if(op==3){

if(ch == input[i+1]){

printf("LOGICALOP ");

}

}

else if(op==1){

ch = input[++i];

if(ch=='=' || ch == '!'){

printf("ASSIGN ");

}

else if(ch==' '){

printf("ARITHOP ");

}

}

if(isSeparator(ch)){

printf("SP ");

}

if(isalnum(ch)){

if(isalpha(ch)){

while(isalnum(ch)){

str[j++]=ch;

ch=input[++i];

}

str[j]='\0';

if(isFunc(str)){

printf("FC ");

while(input[i]!=')'){

i++;

}

}

else if(isKeyword(str)){

printf("KW ");

}

else{

printf("ID ");

}

}

else{

printf("NUMCONST ");

}

}

if(ch==' '){

printf(" ");

}

}

}

int main(){

FILE \*fp;

char input[100];

fp = fopen("sample.c","r");

while(fgets(input,100,fp)){

lexanalyse(input);

printf("\n");

}

fclose(fp);

}

Sample File:

#include<stdio.h>

#include<stdlib.h>

int main(){

int a, b, c;

a = 50;

b = 30;

c = a + b;

if(a > c){

printf("Got it!");

}

return 0;

}

**Output**:Text

Description automatically generated

**Learning outcome**:

* Learned about the role of Lexical Analyser
* Learned about working of lexical analyser
* Learned to simulate tokenizing functions in C
* Learned the components of a program

**Result:**

Lexical Analyser was simulated using C

**Exercise 2**

**Aim**:

Develop a Lexical analyzer to recognize the patterns namely, identifiers, constants, comments and operators using the following regular expressions. Construct symbol table for the identifiers with the following information.

**Code:**

%{

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

int i = 0;

int j = 0;

int flag = 0;

int flag1 = 0;

int addr = 1000;

char buf[100];

typedef struct

{

char type[100];

char name[100];

char value[100];

}table;

table st[100];

void func(char str[],char type[]){

if(type=="KW"){

if(strcmp(str,"int")==0||strcmp(str,"float")==0||strcmp(str,"char")==0){

strcpy(buf,str);

flag1 = 1;

}

}

else if(flag1 == 0)return;

else if(type == "ID"){

if(flag==1){

i++;

strcpy(st[i].value,"--");

}

flag = 1;

strcpy(st[i].name ,str);

strcpy(st[i].type ,buf);

}

else if(type=="VAL"){

strcpy(st[i].value,str);

i++;

flag = 0;

strcpy(st[i].value,"--");

}

else if(strcmp(str,";")==0){

if(flag==1){

i++;

strcpy(st[i].value,"--");

}

flag1 = 0;

printf("HI\n");

}

}

%}

assign\_ops ("="|"+="|"-="|"%="|"/="|"\*=")

rel\_ops ("<"|">"|"<="|">="|"=="|"!=")

arith\_ops ("+"|"-"|"%"|"/"|"\*")

spl\_chars [{}(),;\[\]]

keyword (int|float|char|unsigned|typedef|struct|return|continue|break|if|else|for|while|do|extern|auto|case|switch|enum|goto|long|double|sizeof|void|default|register)

int\_const [0-9]+

float\_const [0-9]+.[0-9]+

char\_const [\'].[\']

string\_const [\"].\*[\"]

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

function [a-zA-Z\_][a-zA-Z0-9]\*[(].\*[)]

single\_cmt [/][/].\*

/\*Rules\*/

%%

{keyword} {

printf("KW ");

func(yytext,"KW");

}

{function} printf("FUNCT ");

{identifier} {

printf("ID ");

func(yytext,"ID");

}

{single\_cmt} printf("SCMT ");

{int\_const} {

printf("INT\_CONST ");

func(yytext,"VAL");

}

{float\_const} {

printf("FLT\_CONST ");

func(yytext,"VAL");

}

{char\_const} {

printf("CHAR\_CONST ");

func(yytext,"VAL");

}

{string\_const} {

printf("STR\_CONST ");

func(yytext,"VAL");

}

{rel\_ops} printf("REL\_OP ");

{arith\_ops} printf("ARITH\_OP ");

{assign\_ops} printf("ASSIGN\_OP ");

{spl\_chars} {

printf("SP");

func(yytext,"SP");

}

\n {

printf("\n");

}

[ \t] { }

%%

int yywrap(void){

return 1;

}

int main(int argc, char \*argv[]){

strcpy(st[0].value,"undefined");

yyin = fopen(argv[1], "r");

yylex();

printf("TYPE\tNAME\tVALUE\tSIZE\tADDR\n");

for(;j<i;j++){

printf("%s\t%s\t%s\t",st[j].type,st[j].name,st[j].value);

if(strcmp(st[j].type,"int")==0){printf("2\t");addr+=2;}

else if(strcmp(st[j].type,"float")==0){printf("4\t");addr+=4;}

else {printf("1\t");addr++;}

printf("%d\n",addr);

}

return 0;

}

**Output:**

**Text

Description automatically generated**

**Learning Outcome:**

* Learned how analyser splits the program statements to understand the purpose of the code
* Learned how to recognize patterns such as identifiers and regular expressions
* Learned how to use Lex tool
* Understood Lex methods like yywrap()

**Result:**

Lexical analyser was simulated using Lex tool

**Exercise 3:**

**Aim**:

Eliminate Left Recursion in given grammar

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

int main() {

char\* name = (char\*)calloc(100,sizeof(char));

printf("Enter File name: ");

scanf("%[^\n]",name);

FILE \*fp;

fp = fopen(name, "r");

int i = 0;

char str[100];

char ans[100][100];

int arow = 0,acol = 0;

printf("\n\t Eliminate left recursion\n");

int flag = 0;

while(fgets(str, 100, fp)) {

str[strlen(str)] = '\0';

char ch = str[0];

int start = 3;

int p1 = 0;

int p2 = 0;

char beta[100][100];

char alpha[100][100];

char A = str[0];

for(int i=3;i<=strlen(str);i++){

if(str[i] == '|'||str[i]=='\0'){

int h = 0;

if(str[start]==A){

for(int j=start+1;j<i;j++){

alpha[p1][h] = str[j];

h++;

}

alpha[p1][h] = '\0';

p1++;

}

else{

for(int j=start;j<i;j++){

if(str[i]=='\0'&&j==i-1)

break;

beta[p2][h] = str[j];

h++;

}

beta[p2][h] = '\0';

p2++;

}

start = i+1;

}

}

if(p1==0){

strcpy(ans[arow],str);

arow++;

continue;

}

flag = 1;

ans[arow][0] = A;

ans[arow][1] = '-';

ans[arow][2] = '>';

acol = 3;

for(int i=0;i<p2;i++){

for(int j=0;beta[i][j]!='\0';j++){

ans[arow][acol] = beta[i][j];

acol++;

}

ans[arow][acol] = A;

acol++;

ans[arow][acol] = '\'';

acol++;

if(i!=p2-1){

ans[arow][acol] = '|';

acol++;

}

}

ans[arow][acol] = '\0';

arow++;

acol = 6;

ans[arow][0] = A;

ans[arow][1] = '\'';

ans[arow][2] = '-';

ans[arow][3] = '>';

ans[arow][4] = 'e';

ans[arow][5] = '|';

for(int i=0;i<p1;i++){

for(int j=0;alpha[i][j]!='\0';j++){

ans[arow][acol] = alpha[i][j];

acol++;

}

ans[arow][acol] = A;

acol++;

ans[arow][acol] = '\'';

acol++;

if(i!=p1-1){

ans[arow][acol] = '|';

acol++;

}

}

ans[arow][acol] = '\0';

arow++;

}

if(!flag){

printf("No left recursion\n");

}

printf("New grammar:\n");

for(int i=0;i<arow;i++){

printf("%s\n",ans[i]);

}

}

**Input:**

A->A+T|T

T->T\*F|F

B->B+T|B-T|a|b

F->id

**Output:**

Text

Description automatically generated

**Learning Objective:**

* Learned how to eliminate left recursion
* Learned why left recursion is problematic
* Learned new format to store productions

**Result:**

Left Recursion has been eliminated from a set of productions

**Exercise 4:**

**Aim**:

Write a program in C to construct Recursive Descent Parser for the following grammar which is for arithmetic expression involving + and \*. Check the Grammar for left recursion and convert into suitable for this parser. Write recursive functions for every non-terminal. Call the function for start symbol of the Grammar in main().

**Code:**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

int space = 0;

void indent(){

for(int i = 0;i<space;i++)printf(" ");

}

void F(char\* inp, int\* cur);

void read(char\* inp) {

printf("Enter code(Expression):");

scanf(" %[^\n]", inp);

int n = strlen(inp);

inp[n] = '$';

inp[n + 1] = '\0';

}

void advance(int\* cur){

(\*cur) = (\*cur) + 1;

}

void err(char \*inp, int \*cur){

printf("\tError:Excpected CONST token, found: %c, at column %d\n", inp[\*cur], \*cur);

exit(0);

}

void Tprime(char\* inp, int\* cur){

space++;

indent();

printf("Entered T'()\n");

if(inp[\*cur] == '\*' || inp[\*cur] == '/') {

advance(cur);

F(inp, cur);

Tprime(inp, cur);

}else{

indent();

space--;

printf("Exiting T'()\n");

return;

}

}

void T(char\* inp, int\* cur){

space++;

indent();

printf("Entered T()\n");

F(inp, cur);

Tprime(inp, cur);

indent();

space--;

printf("Exiting T()\n");

}

void Eprime(char\* inp, int\* cur){

space++;

indent();

printf("Entered E'()\n");

if(inp[\*cur] == '+' || inp[\*cur] == '-') {

advance(cur);

T(inp, cur);

Eprime(inp, cur);

}else{

indent();

space--;

printf("Exiting E'()\n");

return;

}

}

void E(char\* inp, int\* cur){

space++;

indent();

printf("Entered E()\n");

T(inp, cur);

Eprime(inp, cur);

indent();

space--;

printf("Exiting E()\n");

}

void F(char\* inp, int\* cur){

space++;

indent();

printf("Entered F()\n");

if(inp[\*cur] == '('){

advance(cur);

E(inp, cur);

if(inp[\*cur] == ')'){

advance(cur);

}else{

err(inp, cur);

}

}

else if(inp[\*cur] == 'i'){

advance(cur);

}else{

err(inp, cur);

}

indent();

space--;

printf("Exiting F()\n");

}

void solve(char\* inp, int\* cur) {

printf("\n\tResuilt:\t");

E(inp, cur);

if (inp[\*cur] == '$') {

printf("\tSuccess\n");

}else{

printf("\tFailure\n");

}

}

void main() {

char\* inp = (char \*)malloc(sizeof(char) \* 108);

int v= 0;

int\* cur = &v;

read(inp);

solve(inp, cur);

}

**Output:** Text

Description automatically generated

Text

Description automatically generated

**Learning Outcome:**

* Learned about Recursive Descent Parser and it’s working

**Result:**

A recursive descent parser was built using appropriate functions.

**Exercise 5**

**Aim**:

Write Lex program to recognize relevant tokens required for the Yacc parser to implement desk calculator. Write the Grammar for the expression involving the operators namely, + , - ,\* , / , ^ , ( , ).Precedence and associativity has to be preserved. Yacc is available as a command in linux. The grammar should have non terminals E, Op and a terminal id.

**Code:**

Calc.l

%{

#include<stdio.h>

#include<stdlib.h>

#include "y.tab.c"

void yyerror(char \*);

int yylex(void);

int yylval;

%}

GT ">="

LT "<="

AND "&&"

OR "||"

EQ "=="

NEQ"!="

LS "<<"

RS ">>"

%%

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

['+'|'-'|'\*'|'/'|'%'|'^'] {return (\*yytext);}

[\t];

[\n] return 0;

. return yytext[0];

{GT} {return GT;}

{LT} {return LT;}

{AND} {return AND;}

{OR} {return OR;}

{NEQ} {return NEQ;}

{EQ} {return EQ;}

{LS} {return LS;}

{RS} {return RS;}

%%

int yywrap(void)

{

return 1;

}

Calc.y

%{

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

int yylex(void);

void yyerror(char \*);

int flag=0;

%}

%token NUM GT LT RS LS AND EQ NEQ OR

%left OR

%left AND

%left '|'

%left '&'

%left EQ NEQ

%left '>' GT

%left '<' LT

%left LS RS

%left '+' '-'

%left '\*' '/' '%'

%left '^'

%left '(' ')'

%%

Result: E {printf("Calculated Result=%d",$$);};

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

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

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

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

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

| E '^' E {$$=pow($1,$3);}

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

| E GT E {$$ = $1>= $3; }

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

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

| E LT E {$$ = $1<= $3; }

| E RS E {$$ = $1>>$3; }

| E EQ E {$$ = $1==$3;}

| E NEQ E {$$ = $1!=$3;}

| E LS E {$$ = $1<< $3; }

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

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

| E AND E {$$ = $1 && $3; }

| E OR E {$$ = $1 || $3; }

| NUM {$$ = $1;}

;

%%

int main()

{

printf("\nEnter expression: ");

yyparse();

if(flag==0)

printf("\nEntered expression is valid\n\n");

return 0;

}

void yyerror(char \* str)

{

printf("\nEntered expression is invalid\n\n");

flag=1;

}

**Output:**

Text

Description automatically generated

Text

Description automatically generated

**Learning Outcome:**

* Learned about Yacc tool and it’s working
* Learning about using Lex and Yacc tools together
* Learned about establishing precedence for operators

**Result:**

Calculator was implemented using Yacc and Lex

**Exercise 6:**

**Aim:**

Develop a Syntax checker to recognize the tokens necessary for the following statements by writing suitable grammars

Assignment statement

Conditional statement

Looping statement

**CHECK.L**

%{

#include<stdio.h>

#include<stdlib.h>

#include "y.tab.h"

void yyerror(char \*);

int yylex(void);

int yylval;

%}

assign ("=")

relop ("=="|"!="|">="|"<="|"<"|">")

arithop ("+"|"-"|"/"|"%"|"\*")

inde ("++"|"--")

logical ("||"|"&&")

id [a-zA-Z\_][a-zA-Z0-9\_]\*

%%

[0-9]+ {return NUM;}

{assign} {return ASSIGN;}

{relop} {return RELOP;}

{logical} {return LOGIC;}

{arithop} {return ARITH;}

{inde} {return INDE;}

"if" {return IF;}

"else if" {return ELSEIF;}

"else" {return ELSE;}

"for" {return FOR;}

"while" {return WHILE;}

{id} {return ID;}

[ \t] {;}

[\n] {;}

. {return \*yytext;}

%%

int yywrap(void)

{

return 1;

}

**CHECK.Y**

%{

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

int yylex(void);

int yyerror(char \*);

int flag=0;

%}

%token NUM ASSIGN ID

%token IF ELSEIF ELSE

%token FOR WHILE

%token RELOP LOGIC ARITH INDE

%%

stmts : bl stmts

| bl

;

bl : Loop '{' bl

| condstmt '{' bl

| expression ';'

| '}'

;

Loop : FOR '(' expression ';' condition ';' expression ')'

| FOR '(' ';' condition ';' ')'

| WHILE '(' condition ')'

;

condstmt : IF '(' condition ')'

| ELSEIF '(' condition ')'

| ELSE

;

condition : condn LOGIC condition

| condn

;

condn : ID RELOP ID

| ID RELOP NUM

| ID

;

expression : init

| ID ASSIGN ID ARITH ID

| ID ASSIGN ID ARITH NUM

| ID ASSIGN NUM ARITH NUM

| ID INDE

| INDE ID

;

init : ID ASSIGN init

| ID ASSIGN ID

| ID ASSIGN NUM

;

%%

int yyerror(char \*s)

{

flag = 1;

return 1;

}

int main(void)

{

printf("\n\nSYNTAX CHECKER USING YACC\n");

printf("\nCODE\n\n");

FILE \*fp = fopen("file.txt", "r+");

char c = fgetc(fp);

while (c != EOF)

{

printf ("%c", c);

c = fgetc(fp);

}

fclose(fp);

printf("\n\n");

FILE \*fps = fopen("file.txt", "r+");

yyparse();

fclose(fps);

if(flag==1)

{

printf("\nSyntactically Incorrect.\n");

}

else

{

printf("\nSyntactically Correct.\n");

}

return 0;

}

**FILE.TXT**

a = 3;b=10;

for(a = 0; a < b; a++){

if(a >= b){

a = a + 1;

}

else{

a = a - 1;

}

while(b < 10){

b++;

}

}

**OUTPUT:**

Text

Description automatically generated

Text

Description automatically generated

**Learning Objective:**

* Learned about syntax checking algorithms
* Learned about using file system with Yacc
* Learned about using multiple productions in Yacc

**Result:**

Syntax Checker was implemented using yacc.

**Exercise 7:**

**Aim:**

Generate Intermediate code in the form of Three Address Code sequence for the sample input program written using declaration, conditional and assignment statements in new language **Pascal-2021**, Following is the sample input

**Code:**

Tac.l

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "y.tab.h"

%}

term ([a-zA-Z\\_][a-zA-Z\\_0-9]\*)

num ([0-9]+)

real {num}\.{num}

relop ("<"|"<="|">"|">="|"=="|"!=")

addop ("+"|"-")

mulop ("\*"|"/"|"%")

spl (";"|","|"{"|"}"|"("|")"|"="|"&"|"|"|"!"|":")

%%

"begin" {return BGN;}

"end" {return END;}

"if" {return IF;}

"then" {return THEN;}

"else" {return ELSE;}

"integer" {return INT;}

"char" {return CHAR;}

"real" {return REALVAR;}

['].['] {yylval.ch = yytext[1]; return CHCONST;}

{term} {yylval.str = strdup(yytext); return VAR;}

{real} {yylval.floatval = atof(yytext); return REAL;}

{num} {yylval.intval = atoi(yytext); return NUM;}

{relop} {yylval.str = strdup(yytext); return RELOP;}

{mulop} {yylval.str = strdup(yytext); return MULOP;}

{addop} {yylval.str = strdup(yytext); return ADDOP;}

{spl} {return \*yytext;}

[ \t\n]+ {;}

. {char errmsg[100];

strcpy(errmsg, "Invalid Character: ");

strcat(errmsg, yytext);

strcat(errmsg, "\n");

yyerror(errmsg);}

%%

Tac.y

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <math.h>

int yylex(void);

int yyerror(char \*);

int yywrap();

int vars = 0, labels = 0;

struct info{

char \*var;

char \*code;

int intval;

float floatval;

char charval;

};

typedef struct info node;

node \*makeNode(){

//creating a new node to store intermediate code

node \*n = (node \*)malloc(sizeof(node));

n->intval = 0;

n->floatval = 0;

n->charval = 0;

n->var = (char \*)malloc(50 \* sizeof(char));

n->code = (char \*)malloc(5000 \* sizeof(char));

return n;

}

%}

/\*Declaration of tokens and precedence\*/

%token BGN END IF THEN ELSE INT CHAR REALVAR

%token REAL CHCONST VAR NUM RELOP ADDOP MULOP

/\*Increasing precedence\*/

%right MULOP

%left ADDOP

/\*Declaration of the types that YYSTYPE can take with the union\*/

%union{

int intval;

float floatval;

char ch;

char \*str;

struct info \*Node;

}

/\*Declaring types for the tokens\*/

%type<str> VAR RELOP ADDOP MULOP

%type<intval> NUM

%type<floatval> REAL

%type<ch> CHCONST

%type<Node> Program Structure Declarations Statements

%type<Node> Declaration Type Value Statement

%type<Node> Assignment Conditional Condition Expr

%type<Node> E T F

%%

Program : Structure{

printf("\nL%-5d - |\n%s", 0, $$->code);

}

;

Structure : Declarations BGN Statements END{

sprintf($$->code, "%s%10s\n%s", $1->code, "|", $3->code);

}

;

Declarations : Declaration Declarations{

$$ = makeNode();

sprintf($$->code, "%s%s", $1->code, $2->code);

}

| Declaration{

$$ = $1;

}

;

Declaration : VAR ':' Type ';' {

$$ = makeNode();

sprintf($$->code, "%10s %-5s := %s\n", "|", $1, $3->var);

}

| VAR ':' Type '=' Value ';'{

$$ = makeNode();

sprintf($$->code, "%10s %-5s := %s\n", "|", $1, $5->var);

}

;

Type : INT{

$$ = makeNode();

$$->intval = 0;

sprintf($$->var, "%d", 0);

sprintf($$->code, "");

}

| REALVAR{

$$ = makeNode();

$$->floatval = 0.0;

sprintf($$->var, "%.2f", 0.0);

sprintf($$->code, "");

}

| CHAR{

$$ = makeNode();

$$->charval = 0;

sprintf($$->var, "%s", "NULL");

sprintf($$->code, "");

}

;

Value : NUM{

$$ = makeNode();

$$->intval = $1;

sprintf($$->var, "%d", $1);

sprintf($$->code, "");

}

| REAL{

$$ = makeNode();

$$->floatval = $1;

sprintf($$->var, "%.2f", $1);

sprintf($$->code, "");

}

| CHCONST{

$$ = makeNode();

$$->charval = $1;

sprintf($$->var, "%c", $1);

sprintf($$->code, "");

}

;

Statements : Statement Statements{

$$ = makeNode();

sprintf($$->code, "%s%s", $1->code, $2->code);

}

| Statement{

$$ = $1;

}

;

Statement : Assignment {

$$ = $1;

}

| Conditional{

$$ = $1;

}

;

Assignment : VAR '=' Expr ';'{

$$ = makeNode();

char tac[100];

sprintf($$->var, "%s", $1);

sprintf(tac, "%10s %-5s := %s\n", "|", $$->var, $3->var);

sprintf($$->code, "%s%s", $3->code, tac);

}

;

Expr : E{

$$ = $1;

}

;

E : T MULOP E{

$$ = makeNode();

char tac[100];

sprintf($$->var, "x%d", ++vars);

sprintf(tac, "%10s %-5s := %s %s %s\n", "|", $$->var, $1->var, $2, $3->var);

sprintf($$->code, "%s%s%s", $1->code, $3->code, tac);

}

| T{

$$ = $1;

}

| F{

$$ = $1;

}

;

T : T ADDOP F{

$$ = makeNode();

char tac[100];

sprintf($$->var, "x%d", ++vars);

sprintf(tac, "%10s %-5s := %s %s %s\n", "|", $$->var, $1->var, $2, $3->var);

sprintf($$->code, "%s%s%s", $1->code, $3->code, tac);

}

| F{

$$ = $1;

}

;

F : VAR{

$$ = makeNode();

sprintf($$->var, "%s", $1);

sprintf($$->code, "");

}

| NUM{

$$ = makeNode();

$$->intval = $1;

sprintf($$->var, "%d", $1);

sprintf($$->code, "");

}

| REAL{

$$ = makeNode();

$$->floatval = $1;

sprintf($$->var, "%.2f", $1);

sprintf($$->code, "");

}

| CHCONST{

$$ = makeNode();

$$->charval = $1;

sprintf($$->var, "'%c'", $1);

sprintf($$->code, "");

}

;

Conditional : IF '(' Condition ')' THEN Statements ELSE Statements END IF{

$$ = makeNode();

int condnBlock = ++labels;

int endBlock = ++labels;

sprintf($$->code, "%s%10s if %s then goto L%d\n%s%10s goto L%d\n%10s\nL%-5d - |\n%s%10s\nL%-5d - |\n", $3->code, "|", $3->var, condnBlock, $8->code, "|", endBlock, "|", condnBlock, $6->code, "|", endBlock);

}

;

Condition : Expr RELOP Expr{

$$ = makeNode();

char tac[100];

sprintf($$->var, "%s%s%s", $1->var, $2, $3->var);

sprintf($$->code, "%s%s", $1->code, $3->code);

}

;

%%

int yyerror(char\* str){

printf("\n%s", str);

return 0;

}

int yywrap(){

return 1;

}

int main(){

printf("\n\t\tIntermediate Code Generation\n");

printf("\nYour Code:\n\n");

system("cat input.txt");

printf("\n\nThree Address Code:\n");

yyparse();

return 0;

}

Input.txt

i: integer=5;

a: real=4.2;

b: char='c';

c: integer=7;

x: integer;

begin

if (i>2) then

x=a-b/c;

else

x=a+b\*c;

end if

end

**Output:**

**Text

Description automatically generated**

**Learning Objective:**

* Learned about Three Address Code and it’s generation
* Learned about union of datatypes

**Result:**

Intermediate code generation using Lex and Yacc has been implemented

**Exercise 8:**

**Aim:**

Implement Code optimization using a simple C program

**Code:**

Opt.c

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

int main(int argc, char \*argv[]){

stdin = fopen(argv[1], "r");

char line[100];

printf("\nOptimized Code\n\n");

scanf(" %s[^\n]", line);

while (strcmp(line, "END") != 0){

if ((line[3] == '+' && line[4] == '0') || (line[3] == '\*' && line[4] == '1')){

if (line[0] == line[2]){

scanf(" %s[^\n]", line);

continue;

}

printf("%c=%c\n", line[0], line[2]);

}

else if ((line[3] == '-' && line[4] == '0') || (line[3] == '/' && line[4] == '1')){

if (line[0] == line[2]){

scanf(" %s[^\n]", line);

continue;

}

printf("%c=%c\n", line[0], line[2]);

}

else if ((line[3] == '+' && line[2] == '0') || (line[3] == '\*' && line[2] == '1')){

if (line[0] == line[4]){

scanf(" %s[^\n]", line);

continue;

}

printf("%c=%c\n", line[0], line[4]);

}

else if (line[3] == '\*' && line[4] == '2'){

printf("%c=%c+%c\n", line[0], line[2], line[2]);

}

//FOR: x=2\*y

else if (line[3] == '\*' && line[2] == '2'){

printf("%c=%c+%c\n", line[0], line[4], line[4]);

}

//FOR: x=pow(y,2);

else if (line[2] == 'p' && line[3] == 'o' && line[4] == 'w' && line[5] == '(' && line[8] == '2'){

printf("%c=%c\*%c\n", line[0], line[6], line[6]);

}

else{

printf("%s\n", line);

}

scanf(" %s[^\n]", line); //next line

}

return 0;

}

Input.txt

a=a+2

a=b+0

a=b-0

a=a+0

a=b\*1

a=c/1

a=1\*b

a=1\*a

a=2\*b

a=b\*2

a=pow(c,2)

c=pow(c,2)

END

**Output:**

**Text

Description automatically generated**

**Learning Outcome:**

* Learned about Code Optimized
* Learned about the problems of unoptimized code
* Learned about redundant statements

**Result:**

Code Optimizer was implemented in C