**CD LAB PROJECT**

**ANUBHAV TIKKU**

**DIVYANSH VERMA**

**SAURAV PANDA**

**OBJECTIVE**: Construction of parser for Decaf programming language

using Bison

**GRAMMAR:**

program:- class program '{' field\_decl\_multiple method\_decl\_multiple '}'

field\_decl\_multiple:- epsilon | field\_decl\_multiple field\_decl\_single ‘;'

field\_decl\_single:- type idList

idList:- identifier id\_single | identifier '[' int\_literal ']' id\_single

id\_single:- epsilon |',' identifier id\_single | ',' identifier '[' int\_literal ']' id\_single

method\_decl\_multiple:- epsilon | method\_decl\_single method\_decl\_multiple

method\_decl\_single:- type identifier argumentList block

| void identifier argumentList block

argumentList:- '(' ')' |'(' type identifier arg ')'

arg:- epsilon |',' type identifier arg

block:- '{' var\_decl\_multiple statement\_multiple ‘}'

var\_decl\_multiple:- epsilon | var\_decl\_single ';' var\_decl\_multiple

var\_decl\_single:- type identifier variableList

variableList:- epsilon|',' identifier variableList

statement\_multiple:- epsilon | statement\_multiple statement\_single

statement\_single:- location assignment\_opertor expr ';'

| method\_call ';'

| if condition block else\_block

| for identifier assignment\_opertor expr ',' expr block

| return return\_expr ';'

| break ';'

| continue ';'

| block

else\_block:- epsilon | else block

condition:- '(' expr ')'

return\_expr:- epsilon | expr

expr:- location | method\_call | literal | arith\_expr | rel\_expr | equal\_expr

| condition\_expr |'-' expr | '!' expr | '(' expr ')'

location:- identifier | identifier '[' expr ‘]'

method\_call:- method\_name '('parameterList')'

| callout '(' string\_literal callout\_arg ‘)’

method\_name:- identifier

parameterList:- epsilon | expr parameter

parameter:- epsilon | ',' expr parameter

literal:- int\_literal | char\_literal | bool\_literal

int\_literal:- decimal\_literal | hex\_literal

arith\_expr:- expr '\*' expr | expr '/' expr | expr '%' expr | expr '+' expr | expr '-' expr

rel\_expr:- expr '<' expr | expr '>' expr | expr less\_equal expr

| expr greater\_equal expr

equal\_expr:- expr equal\_equal expr | expr not\_equal expr

condition\_expr:- expr and expr | expr or expr

callout\_arg:- epsilon | callout\_arg ',' expr | callout\_arg ',' string\_literal

**LANGUAGES USED:** C, Flex ,Bison

**TYPE OF PARSER :** Bottom up parser

**METHODOLOGY USED:**

Decaf is a subset of Java containing the essential features of classes and

objects but without many of the more complex features such as threads and exception handling.

The bison program checks the grammar and syntax of the programming language. If the given input file is correct,the program can successfully parse through it , else an error is displayed. The flex program generates and returns tokens to the bison program.

**USER DOCUMENTATION:**

https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-035-computer-language-engineering-spring-2010/projects/MIT6\_035S10\_decaf.pdf

**CODE:**

**BISON PROGRAM**

%{

#include<stdio.h>

#include<stdlib.h>

extern FILE \*yyin;

int yyerror();

int yylex();

%}

%token CLASS PROGRAM FOR IF ELSE CONTINUE BREAK RETURN CALLOUT ASSIGNMENT\_OPERATOR IDENTIFIER VOID TYPE HEX\_LITERAL DECIMAL\_LITERAL

%token CHAR\_LITERAL BOOL\_LITERAL STRING\_LITERAL PRINT

%left TYPE

%left OR

%left AND

%left EQUAL\_EQUAL NOT\_EQUAL

%left GREATER\_EQUAL LESS\_EQUAL '<' '>'

%left '+' '-'

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

%nonassoc '!'

%%

program: CLASS PROGRAM '{' field\_decl\_multiple method\_decl\_multiple '}'

;

field\_decl\_multiple: |field\_decl\_multiple field\_decl\_single ';'

;

field\_decl\_single: TYPE idList

;

idList: IDENTIFIER id\_single |IDENTIFIER '[' int\_literal ']' id\_single

;

id\_single: |',' IDENTIFIER id\_single |',' IDENTIFIER '[' int\_literal ']' id\_single

;

method\_decl\_multiple: |method\_decl\_single method\_decl\_multiple

;

method\_decl\_single: TYPE IDENTIFIER argumentList block |VOID IDENTIFIER argumentList block

;

argumentList: '(' ')' |'(' TYPE IDENTIFIER arg ')'

;

arg: | ',' TYPE IDENTIFIER arg

;

block: '{' var\_decl\_multiple statement\_multiple '}'

;

var\_decl\_multiple: |var\_decl\_single ';' var\_decl\_multiple

;

var\_decl\_single: TYPE IDENTIFIER variableList

;

variableList: |',' IDENTIFIER variableList

;

statement\_multiple: |statement\_multiple statement\_single

;

statement\_single: location ASSIGNMENT\_OPERATOR expr ';'

|method\_call ';'

|IF condition block else\_block

|FOR IDENTIFIER ASSIGNMENT\_OPERATOR expr ',' expr block

|RETURN return\_expr ';'

|BREAK ';'

|CONTINUE ';'

|block

|printstmt

;

printstmt: PRINT expr ';'

;

else\_block: |ELSE block

;

condition: '(' expr ')'

;

return\_expr: |expr

;

expr: location |method\_call |literal |arith\_expr |rel\_expr |equal\_expr |condition\_expr |'-' expr |'!' expr

| '(' expr ')'

;

location: IDENTIFIER |IDENTIFIER '[' expr ']'

;

method\_call: method\_name '('parameterList')' |CALLOUT '(' STRING\_LITERAL callout\_arg ')'

;

method\_name:IDENTIFIER

;

parameterList: | expr parameter

;

parameter: |',' expr parameter

;

literal: int\_literal |CHAR\_LITERAL |BOOL\_LITERAL

;

int\_literal:DECIMAL\_LITERAL |HEX\_LITERAL

;

arith\_expr:expr '\*' expr |expr '/' expr |expr '%' expr |expr '+' expr |expr '-' expr

;

rel\_expr: expr '<' expr |expr '>' expr |expr LESS\_EQUAL expr|expr GREATER\_EQUAL expr

;

equal\_expr:expr EQUAL\_EQUAL expr |expr NOT\_EQUAL expr

;

condition\_expr: expr AND expr|expr OR expr

;

callout\_arg: |callout\_arg ',' expr |callout\_arg ',' STRING\_LITERAL

;

%%

int yyerror(char \*msg)

{

printf("\nError : Decaf program compilation failed\n");

exit(0);

}

int main()

{

#ifdef YYDEBUG

yydebug=1;

#endif

yyin = fopen("a1.txt", "r");

yyparse();

printf("\nDecaf Program succesfully compiled");

}

**FLEX PROGRAM**

%{

#include "decaf.tab.h"

%}

%%

"class" {return CLASS;}

“Print” {return PRINT;}

"Program" {return PROGRAM;}

"int"|"boolean" {return TYPE;}

"void" {return VOID;}

"true"|"false" {return BOOL\_LITERAL;}

"for" {return FOR;}

"if" {return IF;}

"else" {return ELSE;}

"break" {return BREAK;}

"continue" {return CONTINUE;}

"return" {return RETURN;}

"callout" {return CALLOUT;}

[\_a-zA-Z]+[\_a-zA-Z0-9]\* {return IDENTIFIER;}

[0-9]+ {return DECIMAL\_LITERAL;}

0x[0-9a-fA-F]+ {return HEX\_LITERAL;}

"="|"+="|"-=" {return ASSIGNMENT\_OPERATOR;}

'(\\.|[^\\'])+' {return CHAR\_LITERAL;}

"==" {return EQUAL\_EQUAL;}

"<=" {return LESS\_EQUAL;}

">=" {return GREATER\_EQUAL;}

"!=" {return NOT\_EQUAL;}

"||" {return OR;}

"&&" {return AND;}

\/\/.\*$ {;}

[ \t\n]\* {;}

\"(\\.|[^\\"])\*\" {return STRING\_LITERAL;}

. {return \*yytext;}

%%

int yywrap()

{return 1;

}

**a.txt File**

//example program

class Program{

int x,y[100];

boolean z,t[20];

boolean func1(int a, int b){

int z;

}

int func2(int c, boolean d){

int t;

}

boolean func(int a, int b){

for i=0, 99{

if(a[i]==5)

{

p = 10;

}

}

}

}

**SYMBOL TABLE sd.l**

%{

#include <stdio.h>

#include <stdlib.h>

#include "symbol1.h"

int scopeCount=0;

struct token \*tk;

%}

%%

{

("int"|"float"|"double"|"if"|"for"|"else"|"char"|"return"|"void"|"break"|"continue"|"callout"|"boolean") {

struct keywordsPos ele;

ele.keyword = malloc(100);

strcpy(ele.keyword, yytext);

ele.row = row;

arr[cnt++] = ele;

if(strcmp(yytext, "char")==0) {

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

}

col+=yyleng;

}

"{" {

scopeCount+=1;

col++;

}

"}" {

scopeCount-=1;

col++;

}

"class" {

col+=yyleng;

}

"if(".\*")"

{col+=yyleng;

}

"Program" {

col+=yyleng;

}

"main(".\*")" {

col+=yyleng;

}

"Print(".\*")" {

col+=yyleng;

}

"scanf(".\*")" {

col+=yyleng;

}

"#".\* {

row++;

col=1;

}

"//".\* {

row++;

col=1;

}

[/][\*][^\*]\*[\*]+([^\*/][^\*]\*[\*]+)\*[/] {

row++;

col=1;

}

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

insert(yytext, FUNCTION, row, col, scopeCount);

col+=yyleng;

}

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

insert(yytext, IDENTIFIER, row, col, scopeCount);

col+=yyleng;

}

[a-zA-Z\_][a-zA-Z0-9]\*"["[0-9]+"]" {

insert(yytext, IDENTIFIER, row, col, scopeCount);

col+=yyleng;

}

\n {

row++;

col=1;

}

. {

col+=yyleng;

}

}

%%

int main() {

yyin = fopen("a1.txt", "r");

yylex();

display();

}

int yywrap() {

return 1;

}

**SYMBOL TABLE symbol.h**

#include <stdio.h>

#define MAXLEN 10

int id=0;

enum TYPE {IDENTIFIER, FUNCTION};

char \*TYPES[] = {"IDENTIFIER", "FUNCTION"};

struct keywordsPos {

char \*keyword;

int row;

};

struct keywordsPos arr[100];

int cnt = 0;

char scope = 'G';

struct token {

char \*lexeme, \*varType, \*returnType, \*args;

char scope;

unsigned int row, col, index, argc, size;

enum TYPE type;

};

struct ListElement {

struct token tok;

struct ListElement \*next;

};

struct ListElement \*TABLE[MAXLEN] = {NULL};

unsigned int row=1, col=1;

void functionArgs(char \*str, char \*args) {

while(\*str!='(') {

str++;

}

char argv[100] = "<";

char \*temp = str;

while((temp = strstr(temp, "char"))!=NULL) {

strcat(argv, "char");

strcat(argv, ",");

temp++;

}

temp = str;

while((temp = strstr(temp, "int"))!=NULL) {

strcat(argv, "int");

strcat(argv, ",");

temp++;

}

argv[strlen(argv)-1] = '\0';

strcat(argv,">");

strcpy(args, argv);

}

unsigned long hash(char \*str){

unsigned long hash = 5381;

int c;

while (c = \*str++)

hash = ((hash << 5) + hash) + c; /\* hash \* 33 + c \*/

return hash;

}

int Search(char \*str){

unsigned long hashvalue;

hashvalue = hash(str);

hashvalue %= MAXLEN;

if(TABLE[hashvalue]==NULL)

return 0;

struct ListElement \*cur = TABLE[hashvalue];

while(cur!=NULL){

if(strcmp(str,cur->tok.lexeme)==0)

return 1;

cur=cur->next;

}

return 0;

}

void insert(char \*str, enum TYPE type, unsigned int row, unsigned int col, int scopeCount) {

if(Search(str)) {

return;

}

struct ListElement \*ele = malloc(sizeof(struct ListElement));

struct token tok;

tok.lexeme = malloc(100);

tok.varType = malloc(100);

tok.returnType = malloc(100);

tok.args = malloc(100);

strcpy(tok.args, "- ");

if(scopeCount>1) {

tok.scope = 'L';

} else {

tok.scope = 'G';

}

strcpy(tok.returnType, "-");

tok.type = type;

if(type == FUNCTION) {

strcpy(tok.varType, "func");

int lbpos = 0, commaCnt = 0;

char temp[100];

strcpy(temp, str);

for(int i=0;i<strlen(str);i++) {

if(str[i]=='(') {

lbpos = i;

} else if(str[i]==',') {

commaCnt += 1;

}

}

functionArgs(temp, tok.args);

str[lbpos] = '\0';

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

if(arr[i].row == row) {

strcpy(tok.returnType, arr[i].keyword);

break;

}

}

tok.argc = commaCnt+1;

tok.size = 0;

} else {

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

if(arr[i].row == row) {

strcpy(tok.varType, arr[i].keyword);

break;

}

}

int sbPos = -1;

for(int i=0;i<strlen(str);i++) {

if(str[i]=='[') {

sbPos = i;

}

}

int size = 0;

if(sbPos!=-1) {

for(int i=sbPos+1;str[i]!=']';i++) {

size \*= 10;

size += (str[i]-'0');

}

str[sbPos] = '\0';

}

if(size==0) {

size=1;

}

if(strcmp(tok.varType,"int")==0) {

size \*= 4;

} else if(strcmp(tok.varType, "float")==0) {

size \*= 4;

} else if(strcmp(tok.varType, "double")==0) {

size \*= 16;

}

tok.argc = -1;

tok.size = size;

}

tok.index = ++id;

tok.row = row;

strcpy(tok.lexeme, str);

ele->tok = tok;

ele->next = NULL;

int val;

val = hash(tok.lexeme);

val %= MAXLEN;

if(TABLE[val]==NULL)

TABLE[val]=ele;

else{

struct ListElement \*ptr = TABLE[val];

while(ptr->next!=NULL)

ptr=ptr->next;

ptr->next=ele;

}

}

void display() {

//sort();

printf("No:\tName\tType\tScope\tArgument size\tArguments\tSize\tRetType\n\n");

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

struct ListElement \*ptr = TABLE[i];

while(ptr!=NULL) {

printf("%d\t%s\t%s\t%c\t%d\t\t%s\t\t%d\t%s\n", ptr->tok.index, ptr->tok.lexeme,

ptr->tok.varType, ptr->tok.scope,

ptr->tok.argc, ptr->tok.args,

ptr->tok.size, ptr->tok.returnType);

ptr = ptr->next;

}

}

}





