

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Date : 03/02/2022 | | | | | | | |
|  | CSPC62 : COMPILER DESIGN  **LAB-1** | | | | | |  |
|  |  | | | | |  | |
|  | | |  |  | | | |
|  | | | Roll no. : 106119100Name : Rajneesh PandeySection : CSE-B |  | | | |
|  | |  | | |  | | |

1. Design a lexical analyzer that could ignore redundant spaces, tabs, new lines, and comments in a source program (C language).

**Code:**

%{

#include<stdio.h>

%}

%%

\/\/(.\*)\n ;

\/\\**[^\*/]*\*\\*\/\n ;

(*[\t]*|" ")\*\n+ {**fprintf**(yyout,"\n");}

\t ;

" "+ {**fprintf**(yyout," ");}

%%

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

int **main**(int k,char \*\*argcv)

{

  yyin=**fopen**(argcv[1],"r");

  yyout=**fopen**("out1.c","w");

**yylex**();

  return 0;

}

Text

Description automatically generated

Input:

Text

Description automatically generated

Output:

Text

Description automatically generated

1. Read an input C file. Design a lexical analyzer that could recognize keywords, identifiers and numeric data which is valid in C language. You may restrict the length of identifiers to some reasonable value (like 32). Display appropriate message if identifiers are not valid or it is too lengthy. List out the token names along with the recognized lexemes. Construct a symbol table which holds information (name, datatype, offset, size, scope) on valid identifiers.

Code

#include <stdbool.h>

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

char  keywords[22][10] = {"if","else","while","do","break","continue","int","double","float","return","char", "case", "sizeof", "long", "short","typedef","switch","unsigned","void", "static", "struct", "goto"};

// Parsing the input STRING.

void **parse**(char\* str)

{

    int left = 0, right = 0;

    int len = **strlen**(str);

    //delimitors

    char delimiter[] = {" +-\*/,;><=()[]{}\n\""};

    char operators[] = {"+-\*/><=&"};

    int operlen = **strlen**(operators);

    int dellen = **strlen**(delimiter);

    while (right <= len && left <= right) {

        if(str[left] == '/' && str[left + 1] == '/'){

          break;

        }

        // check for string right to be delimiter

**bool** isEndDel = **false**;

        for(int i=0;i<dellen;i++){

            if(delimiter[i] == str[right]){

                isEndDel = **true**;

                break;

            }

        }

        if(isEndDel == **false**)

            right++;

        isEndDel = **false**;

        for(int i=0;i<dellen;i++){

            if(delimiter[i] == str[right]){

                isEndDel = **true**;

                break;

            }

        }

        if (str[left] == '\"') {

          if(left == right)

          right++;

          while (str[right] != '\"') {

            right++;

          }

          right++;

          char\* word = (char \*) **malloc**((right - left + 2)\*sizeof(char));

**strncpy**(word, str + left , right - left);

**printf**("'*%s*' IS A STRING literal\n", word);

          left = right;

          continue;

        }

        if (isEndDel && (left == right)) {

            // check for operators

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

                if(str[right] == operators[i]){

**printf**("'*%c*' IS AN OPERATOR\n", str[right]);

                    break;

                }

            }

            right++;

            left = right;

        } else if ((isEndDel == **true** && left != right) || (right == len && left != right)) {

            char\* word = (char \*) **malloc**((right - left + 2)\*sizeof(char));

**strncpy**(word, str + left, right - left);

            word[right-left ] = '\0';

            int lengt = right-left;

**bool** isDone = **false**;

            for(int i=0;i<22;i++){

                if(!**strcmp**(keywords[i], word) && isDone == **false**){

**printf**("'*%s*' IS A KEYWORD\n", word);

                    isDone = **true**;

                }

            }

            if(isDone == **false**) {

                int i = 0;

                if (word[0] == '-') {

                    i = 1;

                }

**bool** isint = **true**;

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

                    if (word[i] < '0' || word[i] > '9') {

                        isint = **false**;

                        break;

                    }

                }

                if (isint) {

**printf**("'*%s*' IS AN INTEGER\n", word);

                    isDone = **true**;

                }

            }

            if(isDone == **false**) {

                int i = 0;

**bool** isfloat = **false**;

                if (word[0] == '-')

                    i++;

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

                    if ((word[i] < '0' || word[i] > '9') && word[i] != '.')

                        break;

                    if (word[i] == '.')

                        isfloat = **true**;

                }

                if (isfloat) {

**printf**("'*%s*' IS A REAL NUMBER\n", word);

                    isDone = **true**;

                }

            }

            if(isDone == **false**){

**bool** isDelEnd = **false**;

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

                    if(delimiter[i] == str[right - 1]){

                        isDelEnd = **true**;

                        break;

                    }

                }

**bool** isIdenti = **true**;

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

                    if(delimiter[i] == word[0] || (word[i] >= '0' && word[i] <= '9')){

                        isIdenti = **false**;

                        break;

                    }

                }

                if (isIdenti && !isDelEnd  && **strlen**(word) != 0)

**printf**("'*%s*' IS A VALID IDENTIFIER\n", word);

                else if (isIdenti == **false** && !isDelEnd)

**printf**("'*%s*' IS NOT A VALID IDENTIFIER\n", word);

            }

            left = right;

        }

    }

}

// DRIVER FUNCTION

int **main**(int argc, char\*\* argv)

{

  char fle[100];

**strcpy**(fle,argv[1]);

  char str[10000];

  FILE \*fp;

  fp =**fopen**(fle,"r");

  while (**fgets**(str,1000,fp)!=**NULL**){

**parse**(str);

  }

  return 0;

}

Input:

Text

Description automatically generated

Output1:

Text

Description automatically generated

Text

Description automatically generated with medium confidence

Code1.y

%{

    #include<stdio.h>

    #include<string.h>

    #include<stdlib.h>

    #include<ctype.h>

    #include"lex.yy.c"

    void **yyerror**(const char \*s);

    int **yylex**();

    int **yywrap**();

    void **add**(char);

    void **insert\_type**();

    int **search**(char \*);

    void **insert\_type**();

    struct dataType {

        char \* id\_name;

        char \* data\_type;

        char \* type;

        int line\_no;

    } symbol\_table[40];

    int count=0;

    int q;

    char type[10];

    extern int countn;

%}

%token VOID CHARACTER PRINTFF SCANFF INT FLOAT CHAR FOR IF ELSE TRUE FALSE NUMBER FLOAT\_NUM ID LE GE EQ NE GT LT AND OR STR ADD MULTIPLY DIVIDE SUBTRACT UNARY INCLUDE RETURN

%%

program: headers main '(' ')' '{' body return '}'

;

headers: headers headers

| INCLUDE { **add**('H'); }

;

main: datatype ID { **add**('F'); }

;

datatype: INT { **insert\_type**(); }

| FLOAT { **insert\_type**(); }

| CHAR { **insert\_type**(); }

| VOID { **insert\_type**(); }

;

body: FOR { **add**('K'); } '(' statement ';' condition ';' statement ')' '{' body '}'

| IF { **add**('K'); } '(' condition ')' '{' body '}' else

| statement ';'

| body body

| PRINTFF { **add**('K'); } '(' STR ')' ';'

| SCANFF { **add**('K'); } '(' STR ',' '&' ID ')' ';'

;

else: ELSE { **add**('K'); } '{' body '}'

|

;

condition: value relop value

| TRUE { **add**('K'); }

| FALSE { **add**('K'); }

|

;

statement: datatype ID { **add**('V'); } init

| ID '=' expression

| ID relop expression

| ID UNARY

| UNARY ID

;

init: '=' value

|

;

expression: expression arithmetic expression

| value

;

arithmetic: ADD

| SUBTRACT

| MULTIPLY

| DIVIDE

;

relop: LT

| GT

| LE

| GE

| EQ

| NE

;

value: NUMBER { **add**('C'); }

| FLOAT\_NUM { **add**('C'); }

| CHARACTER { **add**('C'); }

| ID

;

return: RETURN { **add**('K'); } value ';'

|

;

%%

int **main**() {

  yyparse();

  printf("\n\n");

    printf("\t\t\t\t\t\t\t\t PHASE 1: LEXICAL ANALYSIS \n\n");

    printf("\nSYMBOL   DATATYPE   TYPE   LINE NUMBER \n");

    printf("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n\n");

    int i=0;

    for(i=0; i<count; i++) {

        printf("%s\t%s\t%s\t%d\t\n", symbol\_table[i].id\_name, symbol\_table[i].data\_type, symbol\_table[i].type, symbol\_table[i].line\_no);

    }

    for(i=0;i<count;i++) {

        free(symbol\_table[i].id\_name);

        free(symbol\_table[i].type);

    }

    printf("\n\n");

}

int search(char \*type) {

    int i;

    for(i=count-1; i>=0; i--) {

        if(strcmp(symbol\_table[i].id\_name, type)==0) {

            return -1;

            break;

        }

    }

    return 0;

}

void add(char c) {

  q=search(yytext);

  if(!q) {

    if(c == 'H') {

            symbol\_table[count].id\_name=strdup(yytext);

            symbol\_table[count].data\_type=strdup(type);

            symbol\_table[count].line\_no=countn;

            symbol\_table[count].type=strdup("Header");

            count++;

        }

        else if(c == 'K') {

            symbol\_table[count].id\_name=strdup(yytext);

            symbol\_table[count].data\_type=strdup("N/A");

            symbol\_table[count].line\_no=countn;

            symbol\_table[count].type=strdup("Keyword\t");

            count++;

        }

        else if(c == 'V') {

            symbol\_table[count].id\_name=strdup(yytext);

            symbol\_table[count].data\_type=strdup(type);

            symbol\_table[count].line\_no=countn;

            symbol\_table[count].type=strdup("Variable");

            count++;

        }

        else if(c == 'C') {

            symbol\_table[count].id\_name=strdup(yytext);

            symbol\_table[count].data\_type=strdup("CONST");

            symbol\_table[count].line\_no=countn;

            symbol\_table[count].type=strdup("Constant");

            count++;

        }

        else if(c == 'F') {

            symbol\_table[count].id\_name=strdup(yytext);

            symbol\_table[count].data\_type=strdup(type);

            symbol\_table[count].line\_no=countn;

            symbol\_table[count].type=strdup("Function");

            count++;

        }

    }

}

void insert\_type() {

    strcpy(type, yytext);

}

void yyerror(const char\* msg) {

  fprintf(stderr, "%s\n", msg);

}

Code2.l

%{

    #include "code2.tab.h"

    int countn=0;

%}

%option yylineno

**alpha** [a-zA-Z]

**digit** [0-9]

**unary** "++"|"--"

%%

"printf"                    { return PRINTFF; }

"scanf"                     { return SCANFF; }

"int"                       { return INT; }

"float"                     { return FLOAT; }

"char"                      { return CHAR; }

"void"                      { return VOID; }

"return"                    { return RETURN; }

"for"                       { return FOR; }

"if"                        { return IF; }

"else"                      { return ELSE; }

^"#include"*[ ]*\*<.+\.h>      { return INCLUDE; }

"true"                      { return TRUE; }

"false"                     { return FALSE; }

*[-]*?{digit}+                { return NUMBER; }

*[-]*?{digit}+\.{digit}{1,6}  { return FLOAT\_NUM; }

{alpha}({alpha}|{digit})\*   { return ID; }

{unary}                     { return UNARY; }

"<="                        { return LE; }

">="                        { return GE; }

"=="                        { return EQ; }

"!="                        { return NE; }

">"                     { return GT; }

"<"                     { return LT; }

"&&"                        { return AND; }

"||"                        { return OR; }

"+"                         { return ADD; }

"-"                         { return SUBTRACT; }

"/"                         { return DIVIDE; }

"\*"                         { return MULTIPLY; }

\/\/.\*                      { ; }

\/\\*(.\*\n)\*.\*\\*\/           { ; }

*[ \t]*\*                      { ; }

*[\n]*                        { countn++; }

.                       { return \*yytext; }

*["]*.\**["]*                    { return STR; }

*[']*.*[']*                     { return CHARACTER; }

%%

int **yywrap**() {

    return 1;

}

Output21:

Text

Description automatically generated

1. Design a lexical analyzer that could recognize the operators in C language. Display the name of the operation along with the recognized operator symbol.

%{

#include<stdio.h>

%}

%%

\#.\*\n ;

"++"|"--" **printf**("Unary and Postfix ");ECHO;**printf**("\n");

"+"|"-"    **printf**("Unary and Additive ");ECHO;**printf**("\n");

"!"|"~"|"sizeof()" **printf**("Unary ");ECHO;**printf**("\n");

"\*"|"/"|"%"  **printf**("Multiplicative ");ECHO;**printf**("\n");

"<<"|">>" **printf**("Shift ");ECHO;**printf**("\n");

"<"|"<="|">"|">=" **printf**("Relational ");ECHO;**printf**("\n");

"=="|"!=" **printf**("Equality ");ECHO;**printf**("\n");

"&" **printf**("Bitwise AND ");ECHO;**printf**("\n");

"|" **printf**("Bitwise OR ");ECHO;**printf**("\n");

"^" **printf**("Bitwise XOR ");ECHO;**printf**("\n");

"&&" **printf**("Logical AND ");ECHO;**printf**("\n");

"||" **printf**("Logical OR ");ECHO;**printf**("\n");

"?:" **printf**("Conditional ");ECHO;**printf**("\n");

"="|"+="|"-="|"\*="|"/="|"%="|">>="|"<<="|"&="|"^="|"|=" **printf**("Assignment ");ECHO;**printf**("\n");

"," **printf**("Comma ");ECHO;**printf**("\n");

\".\*\" ;

\n ;

. ;

%%

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

int **main**()

{

   yyin = **fopen**("input.c", "r");

**yylex**();

return 0;

}

Input:

Text

Description automatically generated

Output:

Text

Description automatically generated

1. Write a Lex program that accepts all strings of a's and b's that do not contain the subsequence abb.

**Code:**

%{

#include<stdio.h>

%}

%%

b\*aa\*ba\*b(a|b)\* **printf**("Not Accepted") ;

*[\^b\*aa\*ba\*b(a|b)\*]*+   **printf**("Accepted");

%%

int **yywrap**()

{

return 1;

}

int **main**()

{

**printf**("Enter String\n");

   // called yylex

**yylex**();

return 0;

}

**Input/Output:**

Text

Description automatically generated

1. Write a Lex program that copies a C program, replacing each instance of the keyword float by double.

**Code:**

%{

#include<stdio.h>

//#include<string.h>

%}

%%

"float" **fprintf**(yyout,"double");

.|\n **fprintf**(yyout,"%s",yytext);

%%

int **yywrap**()

{

    return 1;

}

int **main**()

{

  yyin = **fopen**("input5.c", "r");

    yyout = **fopen**("output5.c", "w");

**yylex**();

    return 0;

}

Text

Description automatically generated

Input:

Text

Description automatically generated

Output:

Text

Description automatically generated