LANGAGE C-PASCAL

Analyse Lexicale

Réalisé par :

* Abchir Ouijdane
* Amanzou Anas
* Bahbah Ibrahim
* Lahnech Hamza

Encadré par :

* Pr ELGHAZI Souhail

1. Analyse Lexicale
2. L’alphabet du langage

∑ = {A , B , C , D , E , F , G , H , I , J , K , L , M , N ,O , P , Q , R , S , T , U , V ,

W , X , Y , Z, a , b , c , d , e , f , g , h , i , j , k , l , m , n , o , p , q , r , s , t , u , v ,

w , x , y , z, 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9, +, /, \*, -, >, <, |, &, =, !, ,, ), (, [, ], }, {, ;, #, . }

1. La liste des unités syntaxiques

enum TUnite{

    BIN\_OP,

PT,

BIN\_COMP,

NON,

COMMENT,

    AND,

    OR,

    CROCH\_OUV,

    CROCH\_FER,

    PAR\_OUV,

    PAR\_FER,

    VERG,

    ACC\_OUV,

    ACC\_FER,

    PT\_VER,

    MOT\_CLE,

MOT\_RESERVE,

    NOMBRE,

    IDENT,

COMMENT,

ERROR,

    AUTRE;

}

1. La structure de données

struct TUniteLexical{

    string UL;

    int attribut;

}

string getTUnite(){

        return UL;

    }

1. La grammaire améliorée du langage C-Pascal est la suivante :

<Programme> : : = < liste-declarations>< liste-fonctions> **main** ( ) <bloc>

< liste-declarations> : : = < liste-declarations><declaration> | **epsilon**

<declaration> : : = <declaration>, < declarateur-cp> | < declarateur-cp>;

<declaration-sp> : : = <declaration-sp>,< declarateur-sp>|< declarateur-sp>;

<declarateur-sp> : : = **entier** <variable> | **entier** <variable> [<nombre]

<declarateur-cp> : : = **entier** <variable> | **entier** <variable> [<nombre>] **| struct** <variablestruct>**{**<declaration-sp>}

< liste-fonctions> : : = < liste-fonctions> <fonction> | **epsilon**

<fonction> : : = <identificateur> ( <liste-parms> ) < liste-declarations><bloc>

<liste-parms> : : = <liste-parms> , <parm> | <parm>

<parm> : : = **entier** <variable> | **entier** <variable> [<nombre>]

<bloc> : : = { <liste-instructions> }

<liste-instructions> : : = <liste-instructions>< instruction > | **epsilon**

<instruction> : : = < iteration> I <selection> | <saut> |

<affectation><bloc> I <appel>| **epsilon** ;

<iteration> : : = **tanque** <condition> **faire** <bloc>

<selection> : : = **si** <condition> **alors** <bloc> | **si** <condition> **alors** <bloc> **sinon** <bloc>

<saut> : : = **retour** <expression>

<affectation> : : = <variable> = <expression> | <variablestruct>.

<variable>= <expression> | <variable> = **lire** ( )

<appel> : : = <identificateur> ( <liste-expressions>) | **ecrire** (<

expression>)

<liste-expressions> : : = <liste-expressions>,<expression> | <expression>

<condition> : : = !(<expression>) | <expression><binary-rel><expression> | <expression><binary-comp><expression>

<expression> : : = <terme> | <terme><binary-op><terme >

<terme>  : : = <variable> | <nombre> | <expression>

<variable>  : : = <variable>|<lettre>

<variablestruct>  : : = <variablestruct>|<lettre>

<nombre>  : : = <nombre><constante> | <constante>

<identificateur>  : : = <identificateur><lettre> | <lettre>

<lettre>  : : = A|B|C|...|Z|a|b|c|...­|z

<constante>  : : = 0|1|…­|9

<binary-op> : : = +|-|\*|/

<binary-rel> : : = **&** | |

<binary-comp> : : = <|>|>=|<=|==|!=

<caractere> : : = <constante>|<lettre>|<binary-op>|<binary-rel>|<binary-comp>|<autre>

<ligne> : : = <ligne><caractere >

<commentaire> : : = #<ligne>

1. Liste des unités lexicales

|  |  |  |  |
| --- | --- | --- | --- |
| Unité lexicale | Lexème | Attribut | Modèle |
| COMMENT | ‘’# ‘’ | X | # |
| PT | ‘’.’’ | X | . |
| BIN\_OP | Ex: “+”,”/”…. | Indice dans la table des opérateurs binaires | +|/|\*|- |
| BIN\_COMP | Ex:“<”,”>”,== | Indice dans la table des opérateurs binaires de comparaisons | <|>|>=|<=|==|!= |
| AND | “&” | X | & |
| OR | “|” | X | | |
| CROCH\_OUV | “[“ | X | [ |
| CROCH\_FER | “]” | X | ] |
| PAR\_OUV | “(“ | X | ( |
| PAR\_FER | ‘’)’’ | X | ) |
| COMMENT | ‘’#’’ | X | # |
| NON | ‘’ !’’ | X | ! |
| VERG | *‘’,’’* | X | , |
| AFF | *‘’=’’* | X | = |
| ACC\_OUV | *‘’{‘’* | X | { |
| ACC\_FER | *‘’}’’* | X | } |
| PT\_VER | *‘’;‘’* | X | ; |
| MOT\_CLE | *Ex :retour*  *Ex : entier* | Indice dans la table des mots clés | sinon|alors|si|tanque| entier|retour|faire |
| NOMBRE | ex : 15 | La valeur de l’entier | (0)|(1|2….|9)\*(0|1|…..9)\* |
| MOT\_RESERVE | *Ex :main* | Indice dans la table des mots réservés | lire|ecrire|main |
| IDENT | EX : MACHAN | L’indice dans la table des identificateurs | Lettre(lettre|chiffre)\* |

1. Les tables composant des unités lexicales
   1. La table des mots clés :

|  |  |  |
| --- | --- | --- |
| élément | Lexème | Indice |
| SINON | ‘’sinon’’ | 0 |
| ALORS | ‘’alors’’ | 1 |
| SI | ‘’si’’ | 2 |
| TANQUE | ‘’tanque’’ | 3 |
| FAIRE | ‘’faire’’ | 4 |
| ENTIER | ‘’entier’’ | 5 |
| RETOUR | ‘’retour’’ | 6 |
| STRUCT | “struct” | 7 |

1. La table des mots réservés :

|  |  |  |
| --- | --- | --- |
| élément | Lexème | Indice |
| LIRE | ‘’lire” | 0 |
| ECRIRE | “ecrire“ | 1 |
| MAIN | “main” | 2 |

1. La table des opérateurs binaires :

|  |  |  |
| --- | --- | --- |
| élément | Lexème | Indice |
| PLUS | ‘’+” | 0 |
| MOINS | “-“ | 1 |
| MULTIP | “\*” | 2 |
| DIVISE | “/” | 3 |

1. La table des opérateurs de comparaison :

|  |  |  |
| --- | --- | --- |
| élement | Lexème | Indice |
| INF | “<” | 0 |
| SUP | “>” | 1 |
| INF\_EQUAL | “<=” | 2 |
| SUP\_EQUAL | “>=” | 3 |
| EQUAL | “==” | 4 |
| DIFFERT | “!=” | 5 |

\*\*Diagramme de transition\*\*

Main.cpp

**#include <iostream>**

**#include "Lexical.h"**

**#include <string>**

**#include <fstream>**

**using namespace std;**

**int main()**

**{**

**string str="entier e;main(){e = 153;}";**

**Lexical LexicalTest(str);LexicalTest.readfichier();LexicalTest.uniteSuivante();**

**return 0;**

}

Lexical.h

**#ifndef LEXICAL\_H**

**#define LEXICAL\_H**

**#include <string>**

**#include <vector>**

**using namespace std;**

**enum TUnite{**

**BIN\_OP,**

**PT.**

**BIN\_COMP,**

**NON,**

**COMMENT,**

**AND,**

**OR,**

**AFF,**

**CROCH\_OUV,**

**CROCH\_FER,**

**PAR\_OUV,**

**PAR\_FER,**

**VERG,**

**ACC\_OUV,**

**ACC\_FER,**

**PT\_VER,**

**MOT\_CLE,**

**MOT\_RESERVE,**

**NOMBRE,**

**IDENT,**

**AUTRE,ERROR**

**};**

**typedef struct TUniteLexical{**

**string UL;**

**int attribut;**

**}TUniteLexical;**

**class Lexical**

**{**

**public:**

**Lexical(string s);**

**void readfichier();**

**void fin();**

**void affiche(TUniteLexical ul1);**

**void ULIdentif(string lexeme);**

**void uniteSuivante();**

**bool estBlanc(char c);**

**int getIndexIdentif(string stringToFind, vector<string> stringArray);**

**int getIndex(string stringToFind, vector<string> stringArray);**

**int estMotcle(string lexeme);**

**int estMotreserve(string lexeme);**

**char lc();**

**protected:**

**private:**

**char c=' ';string Str;int test=0;**

**//table des mot-clés**

**vector<string> motsCles={"sinon","alors","si","tantque","faire","entier","retour","structure"};**

**//table des mot-résérvés**

**vector<string> motsReserves={"lire","ecrire","main"};**

**//table des operateurs binaires**

**vector<string> binOp={"+","-","\*","/"};**

**//table des comparateurs binaires**

**vector<string> binComp={"<",">","<=",">=","==","!="};**

**// liste des identifiants**

**vector<string> identifiants;**

**};**

#endif // LEXICAL\_H

Lexical.cpp

**#include <iostream>**

**#include "Lexical.h"**

**#include <string>**

**#include <vector>**

**#include <fstream>**

**using namespace std;**

**Lexical::Lexical(string s)**

**{**

**Str=s;test=0;**

**//ctor**

**}**

**void Lexical::readfichier()**

**{**

**const string NomFichier("ProgrammeTest.txt");**

**ofstream monFlux(NomFichier.c\_str());**

**if(monFlux)**

**{**

**monFlux <<Str<< endl;**

**}**

**else**

**{**

**cout << "ERREUR: Impossible d'ouvrir le fichier." << endl;**

**}**

**ifstream fichier("ProgrammeTest.txt");**

**if(!fichier)**

**{**

**cout << "ERREUR: Impossible d'ouvrir le fichier en lecture." << endl;**

**}**

**else {**

**string ligne; //Une variable pour stocker les lignes lues**

**while(getline(fichier, ligne)) //Tant qu'on n'est pas à la fin, on lit**

**{**

**cout << ligne << endl;**

**}**

**}**

**}**

**char Lexical::lc()**

**{**

**TUniteLexical ul;**

**char car; //Une variable pour stocker les caractères du fichier**

**if(Str[test]!='\0') //Tant qu'on n'est pas à la fin, on lit**

**{**

**car=Str[test];c=car;return car;**

**}**

**else{**

**car='\0';**

**return car;**

**}**

**}**

**void Lexical::fin()**

**{**

**TUniteLexical ul;**

**ul.UL="FIN";ul.attribut=-1;cout<<"EOF"<<"\t";**

**affiche(ul);**

**cout<<"Tableau de motsCles : {";**

**for(int i=0;i<motsCles.size()-1;i++)cout<<motsCles[i]<<", ";**

**cout<<motsCles[motsCles.size()-1]<<"}";**

**cout<<endl;**

**cout<<"Tableau de motsReserves : {";**

**for(int i=0;i<motsReserves.size()-1;i++)cout<<motsReserves[i]<<", ";**

**cout<<motsReserves[motsReserves.size()-1]<<"}";**

**cout<<endl;**

**cout<<"Tableau des identifiants : {";**

**for(int i=0;i<identifiants.size()-1;i++)cout<<identifiants[i]<<", ";**

**cout<<identifiants[identifiants.size()-1]<<"}";**

**cout<<endl;**

**}**

**bool Lexical::estBlanc(char cc)**

**{**

**return (cc==' ' || cc=='\t' || cc=='\n' || cc=='\r');**

**}**

**int Lexical::getIndex(string stringToFind, vector<string> stringArray)**

**{**

**int index = -1;**

**for (int i=0;i<stringArray.size();i++) {**

**if (stringArray[i]==stringToFind) {**

**index = i;**

**break;**

**}**

**}**

**return index;**

**}**

**int Lexical::getIndexIdentif(string stringToFind, vector<string> stringArray)**

**{**

**int index = -1;**

**for (int i=0;i<stringArray.size();i++) {**

**if (stringArray[i]==stringToFind) {**

**index = i;**

**break;**

**}**

**}**

**return index;**

**}**

**int Lexical::estMotcle(string lexeme)**

**{**

**return getIndex(lexeme,motsCles);**

**}**

**int Lexical::estMotreserve(string lexeme)**

**{**

**return getIndex(lexeme,motsReserves);**

**}**

**void Lexical::ULIdentif(string lexeme)**

**{**

**TUniteLexical ul;**

**int pos = getIndexIdentif(lexeme,identifiants);**

**if(pos==-1){**

**identifiants.push\_back(lexeme);**

**}**

**ul.UL="IDENT";ul.attribut=getIndexIdentif(lexeme,identifiants);cout<<lexeme<<"\t";**

**affiche(ul);**

**}**

**void Lexical::affiche(TUniteLexical ul1)**

**{**

**char car;**

**for(int i=0;i<ul1.UL.size();i++){car=toupper(ul1.UL[i]);cout<<car;}**

**cout<<"\t\t"<<ul1.attribut<<endl;**

**}**

**void Lexical::uniteSuivante()**

**{**

**string lexeme="";TUniteLexical ul;**

**//if(c=='/0')**

**while(estBlanc(c)) {c=lc();test++;}**

**switch(c){**

**case '+': {**

**c=lc();test++;**

**ul.UL="BIN\_OP";ul.attribut=getIndex("+",binOp);cout<<"+"<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case '-':{**

**c=lc();test++;**

**ul.UL="BIN\_OP";ul.attribut=getIndex("-",binOp);cout<<"-"<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case '<':{**

**c=lc();test++;**

**if(c=='='){**

**c=lc();test++;**

**ul.UL="BIN\_COMP";ul.attribut=getIndex("<=",binComp);cout<<"<="<<"\t";**

**}**

**else{**

**ul.UL="BIN\_COMP";ul.attribut=getIndex("<",binComp);cout<<"<"<<"\t";**

**}**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case '>':{**

**c=lc();test++;**

**if(c=='='){**

**c=lc();test++;**

**ul.UL="BIN\_COMP";ul.attribut=getIndex(">=",binComp);cout<<">="<<"\t";**

**}**

**else{**

**ul.UL="BIN\_COMP";ul.attribut=getIndex(">",binComp);cout<<">"<<"\t";**

**}**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case '\*': {**

**c=lc();test++;**

**ul.UL="BIN\_OP";ul.attribut=getIndex("\*",binOp);cout<<"\*"<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case '&': {**

**c=lc();test++;**

**ul.UL="AND";ul.attribut=-1;cout<<"&"<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case '|': {**

**c=lc();test++;**

**ul.UL="OR";ul.attribut=-1;cout<<"|"<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case '/': {**

**c=lc();test++;**

**ul.UL="BIN\_OP";ul.attribut=getIndex("/",binOp);cout<<"/"<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case '=':{**

**c=lc();test++;**

**if(c=='='){**

**c=lc();test++;**

**ul.UL="BIN\_COMP";ul.attribut=getIndex("==",binComp);cout<<"=="<<"\t";**

**}**

**else{**

**ul.UL="AFF";ul.attribut=-1;cout<<"="<<"\t";**

**}**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case '!':{**

**c=lc();test++;**

**if(c=='='){**

**c=lc();test++;**

**ul.UL="BIN\_COMP";ul.attribut=getIndex("=!",binComp);cout<<"!="<<"\t";**

**}**

**else{**

**ul.UL="NON";ul.attribut=-1;cout<<"!"<<"\t";**

**}**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case '[':{**

**c=lc();test++;**

**ul.UL="CROCH\_OUV";ul.attribut=-1;cout<<"["<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case ']':{**

**c=lc();test++;**

**ul.UL="CROCH\_FER";ul.attribut=-1;cout<<"]"<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case '(':{**

**c=lc();test++;**

**ul.UL="PAR\_OUV";ul.attribut=-1;cout<<"("<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case ')':{**

**c=lc();test++;**

**ul.UL="PAR\_FER";ul.attribut=-1;cout<<")"<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case ',':{**

**c=lc();test++;**

**ul.UL="VERG";ul.attribut=-1;cout<<","<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case '{':{**

**c=lc();test++;**

**ul.UL="ACC\_OUV";ul.attribut=-1;cout<<"{"<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case '}':{**

**c=lc();test++;**

**ul.UL="ACC\_FER";ul.attribut=-1;cout<<"}"<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case ';':{**

**c=lc();test++;**

**ul.UL="PT\_VER";ul.attribut=-1;cout<<";"<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case '#':{**

**c=lc();test++;**

**ul.UL="COMMENT";ul.attribut=-1;cout<<'#'<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**default:**

**{**

**if(isdigit(c)){**

**while(isdigit(c)){**

**lexeme += c; //Aj**

**c=lc();test++;**

**}**

**ul.UL="NOMBRE";ul.attribut=stoi(lexeme);cout<<lexeme<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**else if(isalpha(c)){**

**while(isalpha(c)||isdigit(c)){**

**lexeme += c; //Aj**

**c=lc();test++;**

**}**

**int pos = estMotcle(lexeme);**

**int pos1 = estMotreserve(lexeme);**

**if(pos!=-1){**

**ul.UL=motsCles[pos];ul.attribut=pos;cout<<lexeme<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**if(pos1!=-1){**

**ul.UL=motsReserves[pos1];ul.attribut=pos1;cout<<lexeme<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**ULIdentif(lexeme);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**else{**

**c=lc();test++;**

**ul.UL="ERROR";ul.attribut=2;cout<<lexeme<<"\t";**

**affiche(ul);if(c=='\0')exit(0);uniteSuivante();**

**}**

**}}**

**}**

**programme en C-Pascal**

entier e;

main()

{

e = 153;

}

**Output de l’analyseur lexical**

**Lexème lu Unité Lexicale(Code) Attribut(valeur du lexème)**

entier ENTIER 5

e IDENT 0

; POINT\_VIRGULE -1

main MAIN -1

( PARENTHESE\_OUVRANTE 2

) PARENTHESE\_FERMANTE -1

{ ACCOLADE\_OUVRANTE -1

e IDENT 0

= EGAL -1

153 NOMBRE 153

; POINT\_VIRGULE -1

} ACCOLADE\_FERMANTE -1

EOF FIN -1