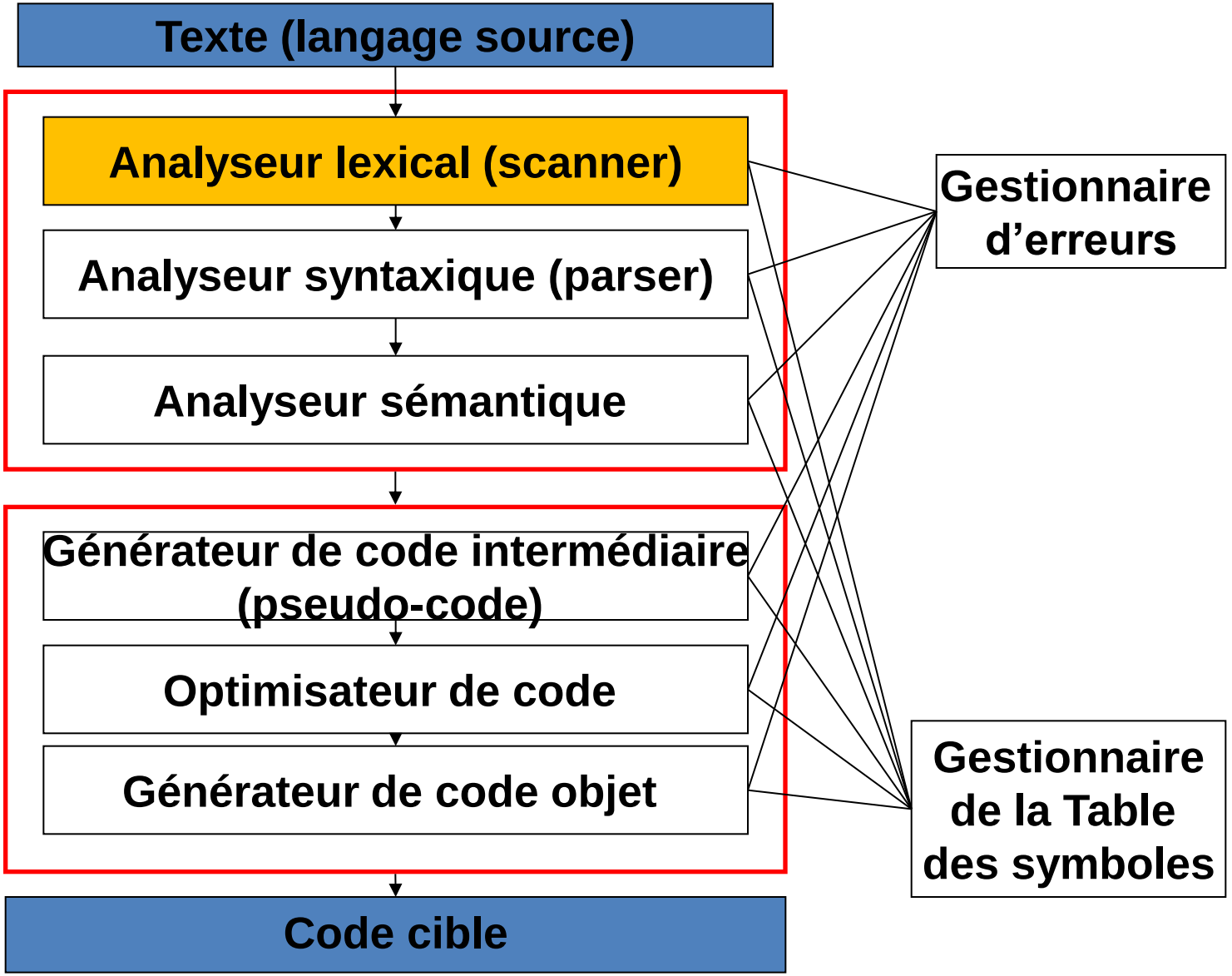
ECRITURE D'UN MINI COMPILATEUR

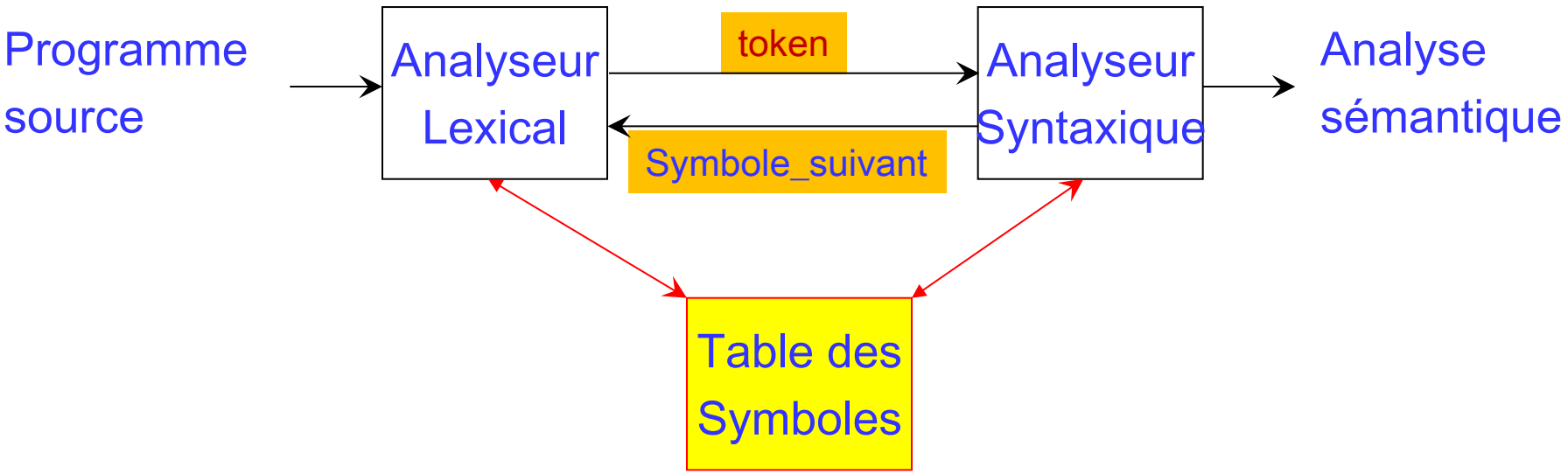
RAPPEL



Architecture générale d'un compilateur

L'analyseur lexical (ou scanner) fusionne les caractères lus du code source en groupes de mots qui forment logiquement des **unités lexicales (**ou **tokens) du** langage

Symboles : identificateurs, chaînes, constantes numériques,
Mots clefs : while, if, then
Opérateurs (ou symboles spéciaux) : <=, :=, =



Que doit retourner l'analyseur lexical à l'analyseur syntaxique ???

Soient les 2 exemples suivants:

A + 15 * B

Toto +45879*tata

Y a-t-il une différence au niveau syntaxique entre les deux phrases????

Identificateur + constante * identificateur

PASSER LES SEPARATEURS: espace, \t, \n et commentaire

SYMBOLE

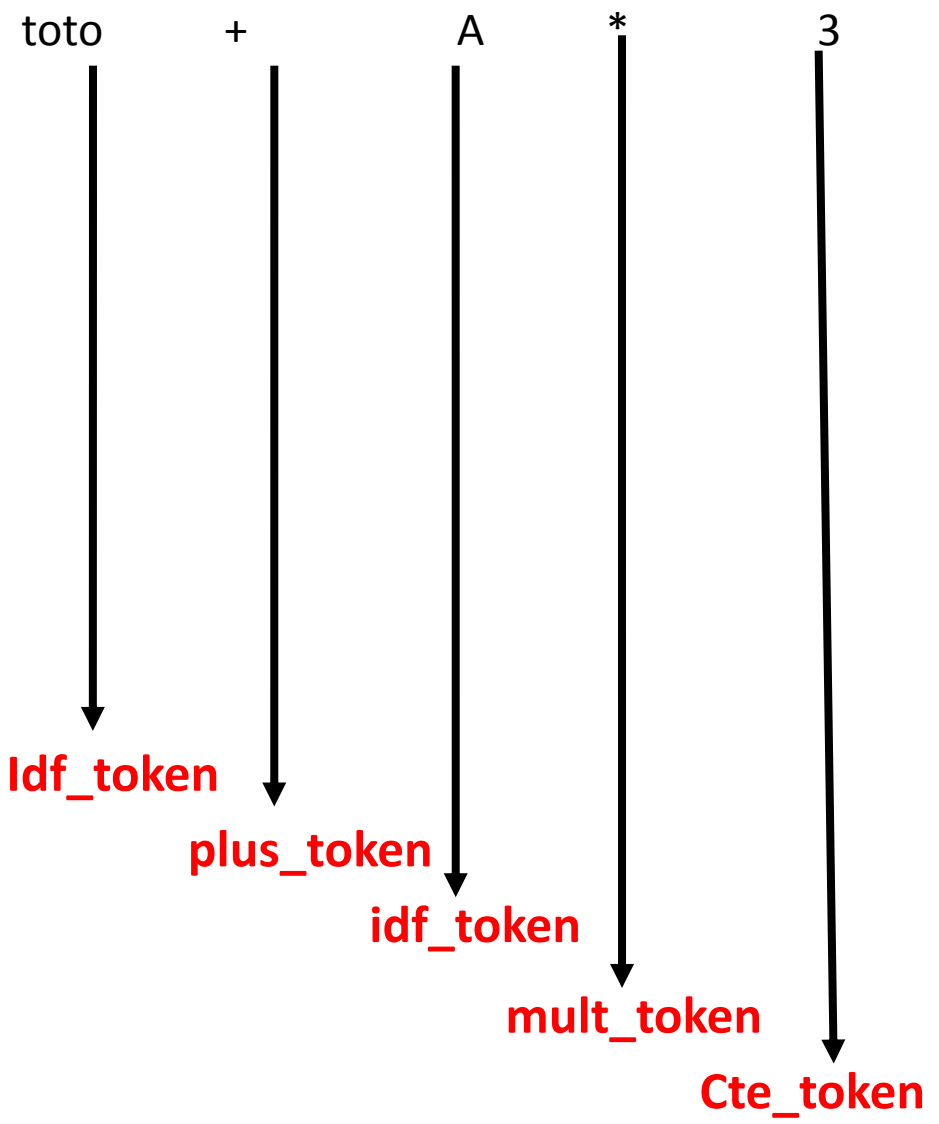
LECTURE ET RECONNAISSANCE CAR PAR CAR

VALEUR DU SYMBOLE (NOM)

CODAGE LEXICALE

CODE

RESULTAT: (NOM, CODE)



Analyse lexical d'un symbole

Dans les langages de programmation 5 catégories de symboles:

- les mots,
- les nombres,
- les chaînes,
- les symboles spéciaux,
- les symboles erronés

Analyse lexical d'un symbole

- Chacune des catégories sera lue par une fonction spécialisée:
  - Lire_nombre pour la lecture des nombres
  - Lire_mot pour la lecture des mots
  - Lire_chaîne pour la lecture des chaînes
  - Lire_spécial pour la lecture des symboles spéciaux
  - Lire_erroné pour la lecture des symboles erronés

Analyse lexical d'un symbole

- Codage lexical
  - Détermine le code du symbole selon la catégorie,
  - ~~*LE RANGE DANS LA TABLE DES SYMBOLES S'IL N'Y EST PAS DÉJÀ*~~

  – Le codage lexical dépend de la catégorie du symbole

LE MINI PROJET

LA GRAMMAIRE

PROGRAM	::=	**program** ID **;** BLOCK **.**
BLOCK	::=	CONSTS VARS INSTS
CONSTS	::=	**const** ID = NUM **;** { ID = NUM **;** } | **ε**
VARS	::=	**var** ID { **,** ID } **;** | **ε**
INSTS	::=	**begin** INST { **;** INST } **end**
INST	::=	INSTS | AFFEC | SI | TANTQUE | ECRIRE | LIRE | ε
AFFEC	::=	ID **:=** EXPR
SI	::=	**if** COND **then** INST
TANTQUE	::=	**while** COND **do** INST
ECRIRE	::=	**write (** EXPR { **,** EXPR } **)**
LIRE	::=	**read (** ID { **,** ID } **)**
COND	::=	EXPR RELOP EXPR
RELOP	::=	= | <> | < | > | <= | >=
EXPR	::=	TERM { ADDOP TERM }
ADDOP	::=	+ | -
TERM	::=	FACT { MULOP FACT }
MULOP	::=	* | /
FACT	::=	ID | NUM | **(** EXPR **)**

NOYAU DE LA GRAMMAIRE DU PASCAL : les règles syntaxiques

ID	::=	lettre {lettre | chiffre}
NUM	::=	chiffre {chiffre}
Chiffre	::=	**0**|..|**9**
Lettre	::=	**a**|**b**|..|**z**|**A**|..|**Z**

NOYAU DE LA GRAMMAIRE DU PASCAL : les règles lexicales

Méta-règles

Une série de règles définissent la forme d'un programme:

•Un *commentaire* est une suite de caractères encadrés des parenthèses {* et *} ;

•Un *séparateur* est un *caractère séparateur* (espace blanc, tabulation, retour chariot) ou un *commentaire* ;

•Deux ID ou *mots clés* qui se suivent doivent être séparés par au moins un *séparateur* ;

•Des *séparateurs* peuvent être insérés partout, sauf à l'intérieur de *terminaux*.
•Longueur maximale des identificateurs = 20

•Pas de distinction entre minuscule et majuscule

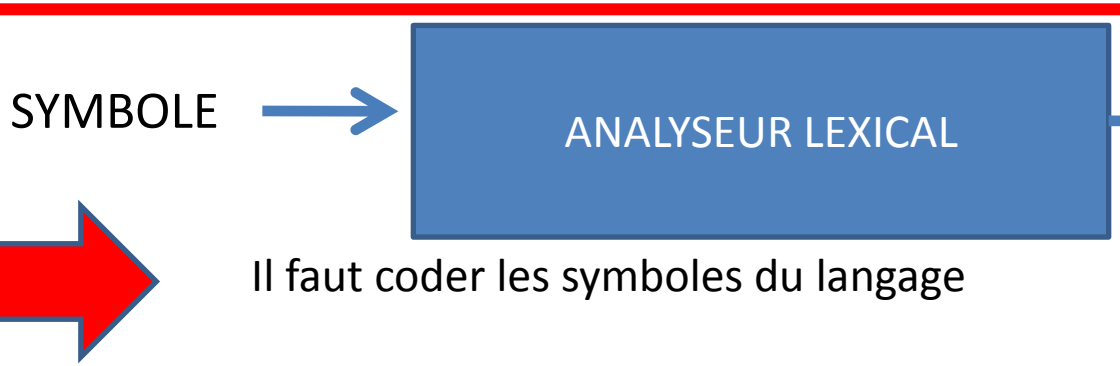
•Les constantes numériques sont entières et de longueur <=11

Exemples de programme Pascal

```
program test11;
const toto=21; titi=13;
var x,y;
Begin
  {* initialisation de x *}
  x:=toto;
  read(y);
  while x<y do begin read(y); x:=x+y+titi end;
  {* affichage des resultas
   de x et y *}
  write(x);
  write(y);
end.
```

Exemple de programme Pascal

ANALYSEUR LEXICAL
MISE EN PRATIQUE



Il faut coder les symboles du langage

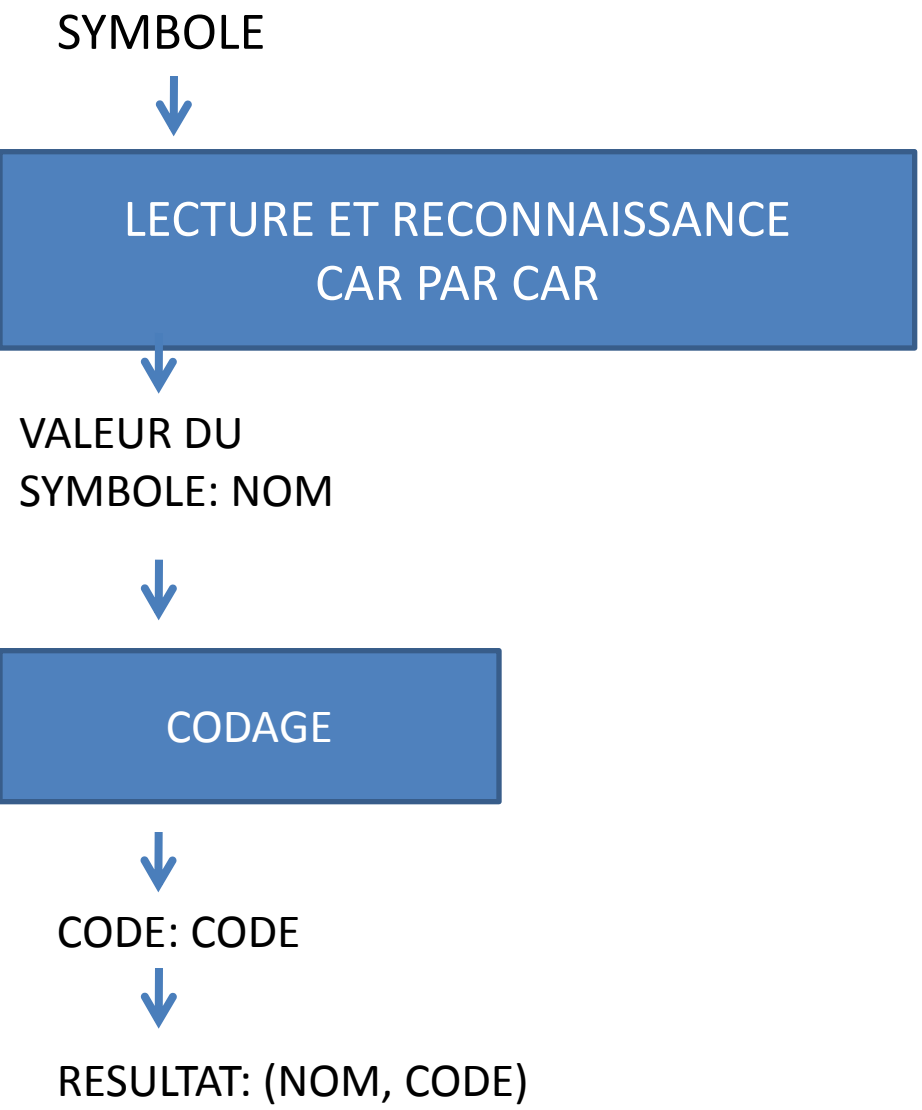
LES MOTS CLES	
program	PROGRAM_TOKEN
const	CONST_TOKEN
var	VAR_TOKEN
begin	BEGIN_TOKEN
end	END_TOKEN
if	IF_TOKEN
then	THEN_TOKEN
while	WHILE_TOKEN
Do	DO_TOKEN
read	READ_TOKEN
write	WRITE_TOKEN

LES SYMBOLES SPECIAUX	
;	PV_TOKEN
.	PT_TOKEN
+	PLUS_TOKEN
-	MOINS_TOKEN
*	MULT_TOKEN
/	DIV_TOKEN
,	VIR_TOKEN
:=	AFF_TOKEN
<	INF_TOKEN
<=	INFEG_TOKEN
>	SUP_TOKEN
>=	SUPEG_TOKEN
<>	DIFF_TOKEN
(	PO_TOKEN
)	PF_TOKEN
EOF	FIN_TOKEN

LES REGLES LEXICALES	
ID	ID_TOKEN
NUM	NUM_TOKEN

LES SYMBOLES ERRONES	
LE RESTE	ERREUR_TOKEN

```
//-------------------------------------------------------------
// DECLARATION DES CLASSES LEXICALES
//en C
//-------------------------------------------------------------
typedef enum {
                ID_TOKEN, PROGRAM_TOKEN,
                CONST_TOKEN, VAR_TOKEN,
                ……
                EOF_TOKEN, ERREUR_TOKEN
} CODES_LEX ;
```




```
char Car_Cour; //caractère courant

void Lire_Car(){
        Car_Cour=fgetc(Fichier);
}
```

SYMBOLE

LECTURE ET RECONNAISSANCE CAR PAR CAR

SELON LA CATEGORIE

LES MOTS:
commence par une lettre: les mots clés et ID

VALEUR DU SYMBOLE: NOM

CODAGE

LES NOMBRES: commence par un chiffre : NUM

LES SPECIAUX: + - , ; etc

LES ERRONES: le reste

CODE: CODE

RESULTAT: (NOM, CODE)

```
typedef  struct { CODE_LEX CODE; char NOM[20]; } TSym_Cour;
TSym_Cour            SYM_COUR;
```

MOTS

LECTURE ET RECONNAISSANCE CAR PAR CAR

LIRE_MOTS()
ID::= lettre {lettre | chiffre}

VALEUR DU SYMBOLE: NOM

CODAGE

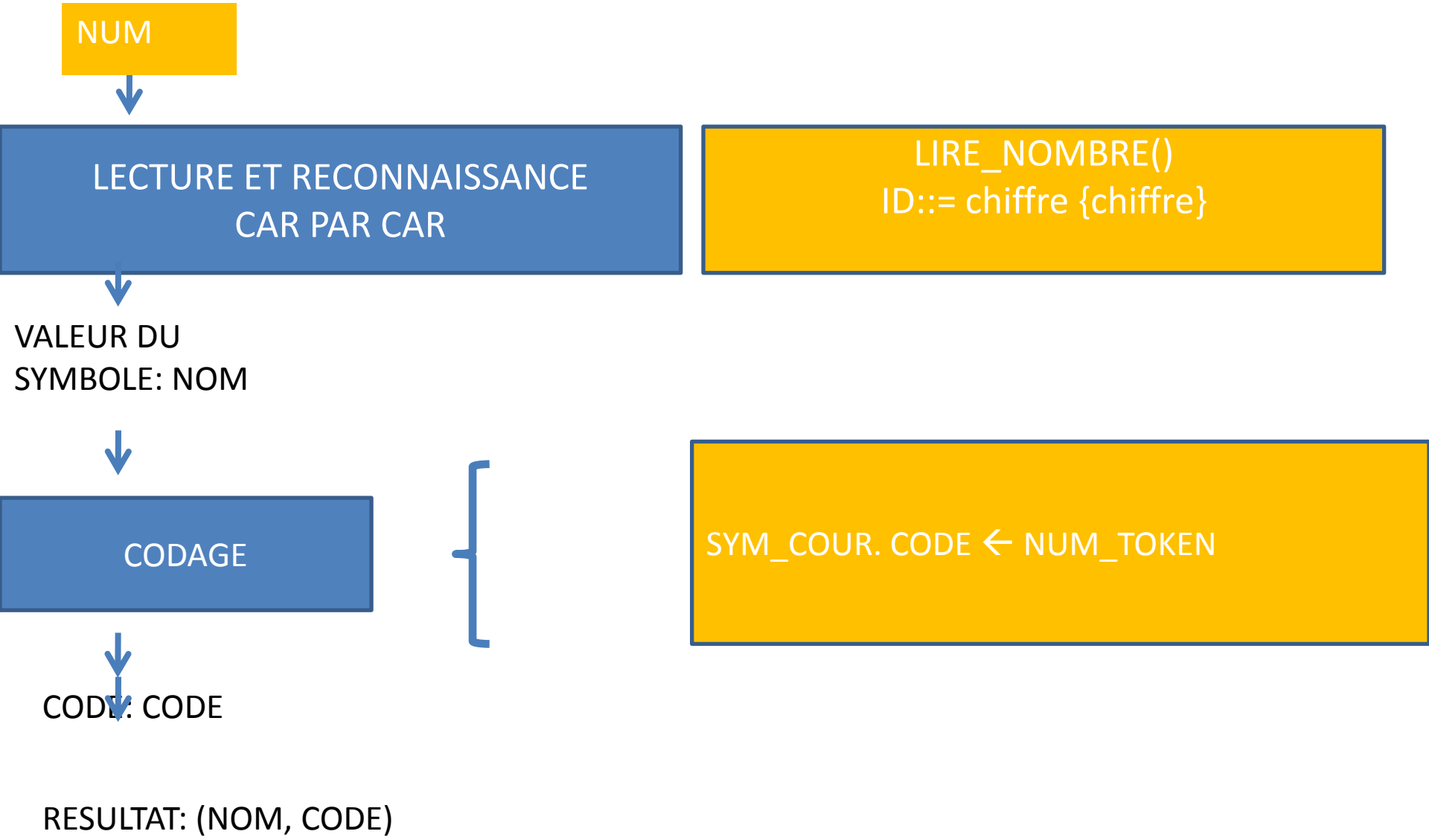
SI MOT CLE
ALORS RETOURNER LE TOKEN DU MOT CLE
SINON RETOURNER ID_TOKEN

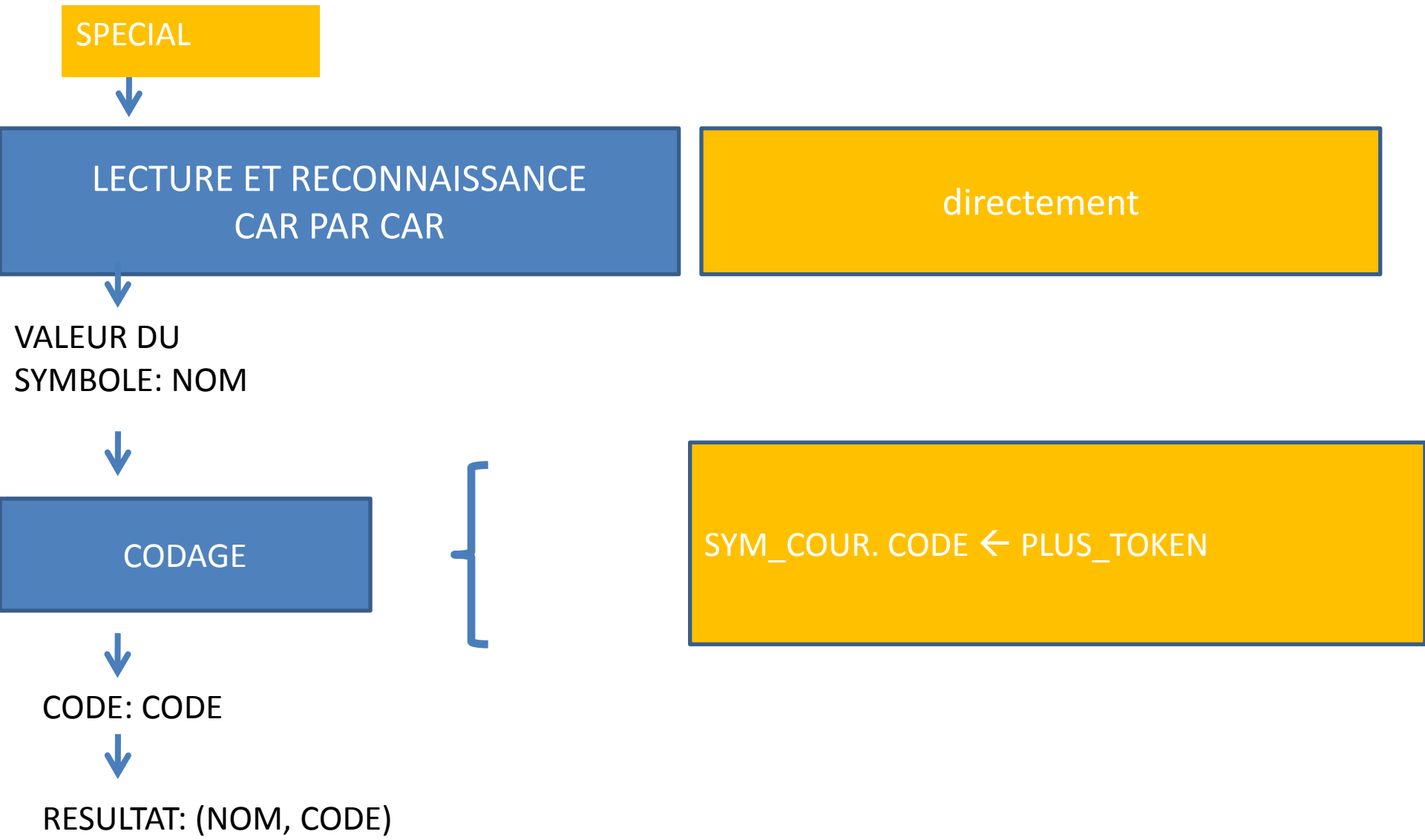
CODE: CODE

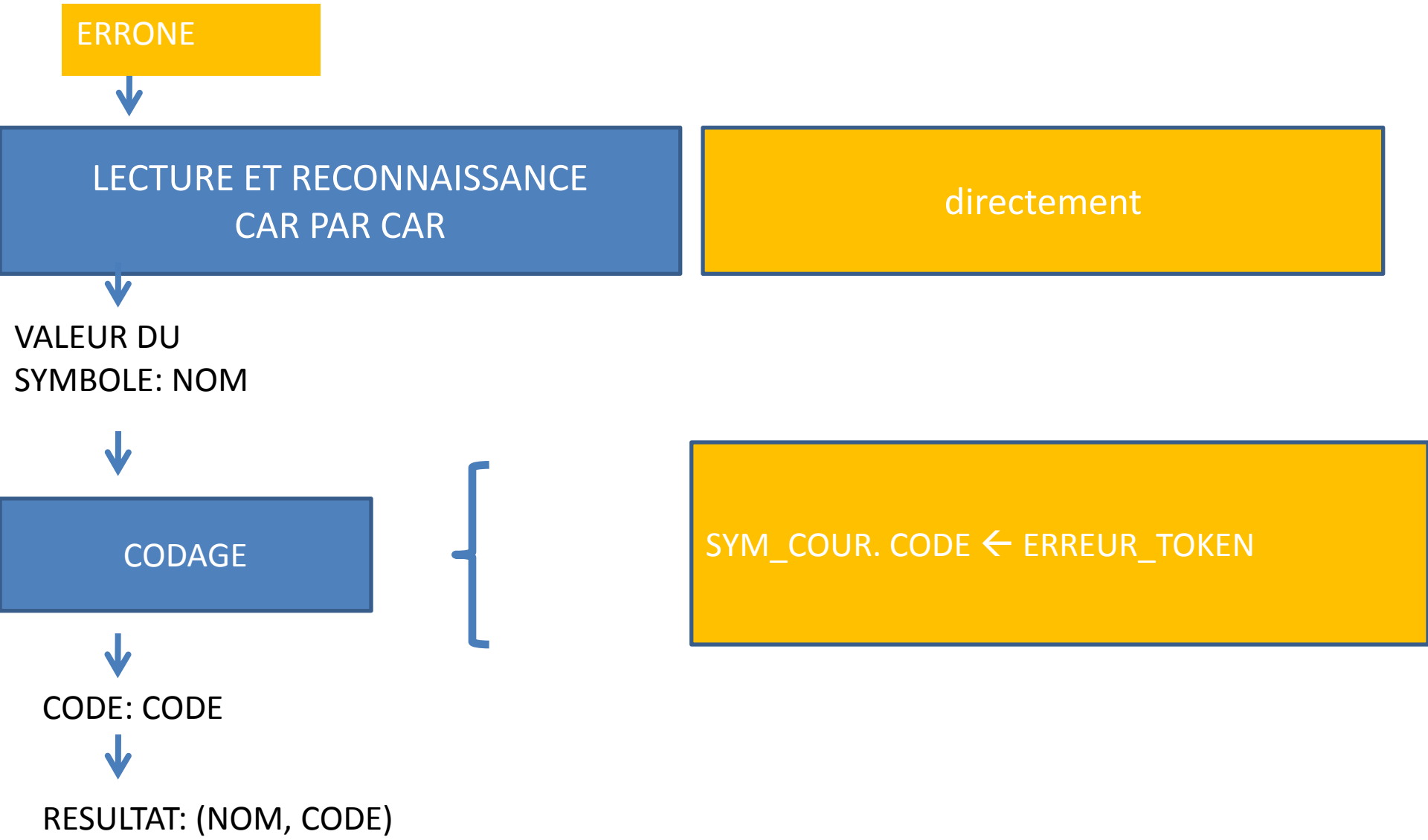
RESULTAT: (NOM, CODE

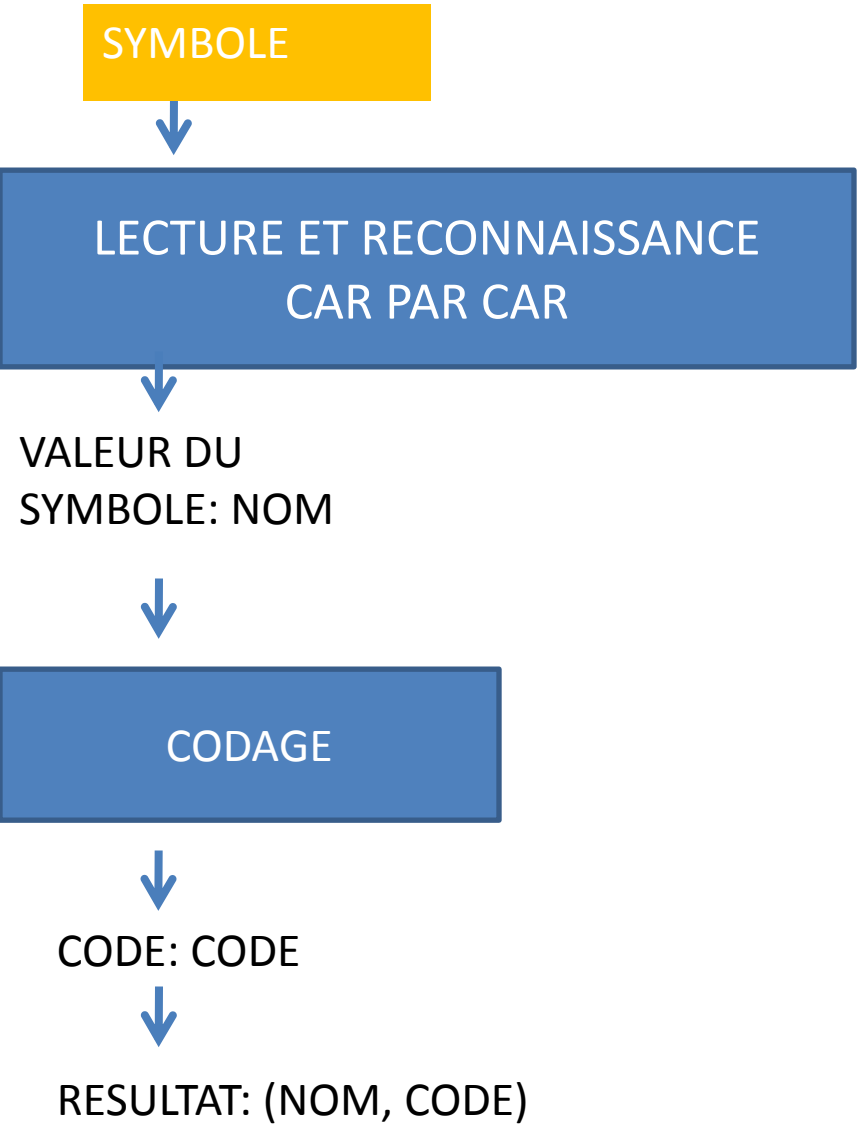
Une table des mots clés

NOM	CODE
program	PROGRAM_TOKEN
Var	VAR_TOKEN
…..	…..


NUM
LECTURE ET RECONNAISSANCE CAR PAR CAR
LIRE_NOMBRE()
ID::= chiffre {chiffre}
VALEUR DU SYMBOLE: NOM
CODAGE
SYM_COUR. CODE ← NUM_TOKEN
CODE: CODE
RESULTAT: (NOM, CODE)


SPECIAL
LECTURE ET RECONNAISSANCE CAR PAR CAR
directement
VALEUR DU SYMBOLE: NOM
CODAGE
SYM_COUR. CODE ← PLUS_TOKEN
CODE: CODE
RESULTAT: (NOM, CODE)


ERRONE
LECTURE ET RECONNAISSANCE CAR PAR CAR
directement
VALEUR DU SYMBOLE: NOM
CODAGE
SYM_COUR. CODE ← ERREUR_TOKEN
CODE: CODE
RESULTAT: (NOM, CODE)




```
void Sym_Suiv(){
-- PASSER LES SEPARATEURS
-- TRAITER SELON LA CATEGORIE
    --CATEGORIE DE MOTS
        si car_cour est une lettre : lire_mot();
    --CATEGORIE DE NOMBRE
        si car_cour est un chiffre : lire_nombre();
    --CATEGORIE DES SPECIAUX
        CAS CAR_COUR PARMI
            '+': SYM_COUR. CODE ←PLUS_TOKEN;
              Lire_Car();
            …….
            EOF: SYM_COUR. CODE ←EOF_TOKEN;

            SINON: SYM_COUR. CODE ←ERREUR_TOKEN;
                   ERREUR(CODE_ERR);
      FINDECAS
}
```

LE TEST DE L'ANALYSEUR LEXICAL

```
int main(){
        Ouvrir_Fichier("E:\\Pascal.p");
        Lire_Caractere();
        while (Car_Cour!=EOF) {
                Sym_Suiv();
                AfficherToken(Sym_Cour);
        }
        getch();
        return 1;
}
```

EXEMPLE DU TEST DE L'ANALYSEUR LEXICAL

```
program test11;
const toto=21;
var x,y;
Begin
  x:=toto;
  read(y);
end.
```


```
PROGRAM_TOKEN
 ID_TOKEN
PV_TOKEN
CONST_TOKEN
 ID_TOKEN
EG_TOKEN
NUM_TOKEN
 PV_TOKEN
VAR_TOKEN
 ID_TOKEN
VIT_TOKEN
 ID_TOKEN
PV_TOKEN
BEGIN_TOKEN
ID_TOKEN
AFF_TOKEN
ID_TOKEN
PV_TOKEN
READ_TOKEN
PO_TOKEN
ID_TOKEN
PF_TOKEN
PV_TOKEN
END_TOKEN
PT_TOKEN
EOF_TOKEN
```

LES MESSAGES D'ERREUR

CODE_ERR	MES_ERREUR
ERR_CAR_INC	"caractère inconnu"
ERR_FIC_VIDE	« fichier vide"
…..	…..

```
//-------------------------------------------------------------------------
// DECLARATION DES CLASSES DES ERREURS
//-------------------------------------------------------------------------
typedef enum {
                ERR_CAR_INC, ERR_FICH_VID, ERR_ID_LONG, …….
}Erreurs;
//-------------------------------------------------------------------------
// DECLARATION DU TABLEAU DES ERREURS
//-------------------------------------------------------------------------
typedef  struct  {   Erreurs  CODE_ERR; char mes[40]   }   Erreurs;

Erreurs        MES_ERR[100]={{ERR_CAR_INC,"caractère inconnu"}, {ERR_FICH_VID,"fichier vide",« IDF très long" },
```

```
void Erreur(Erreurs    ERR){
            int  ind_err=ERR;
            printf( "Erreur  numéro  %d \t : %s \n", ind_err, MES_ERR[ind_err] .mes);
            getch();
            exit(1);
}
```

A VOS MACHINES
et
BON COURAGE