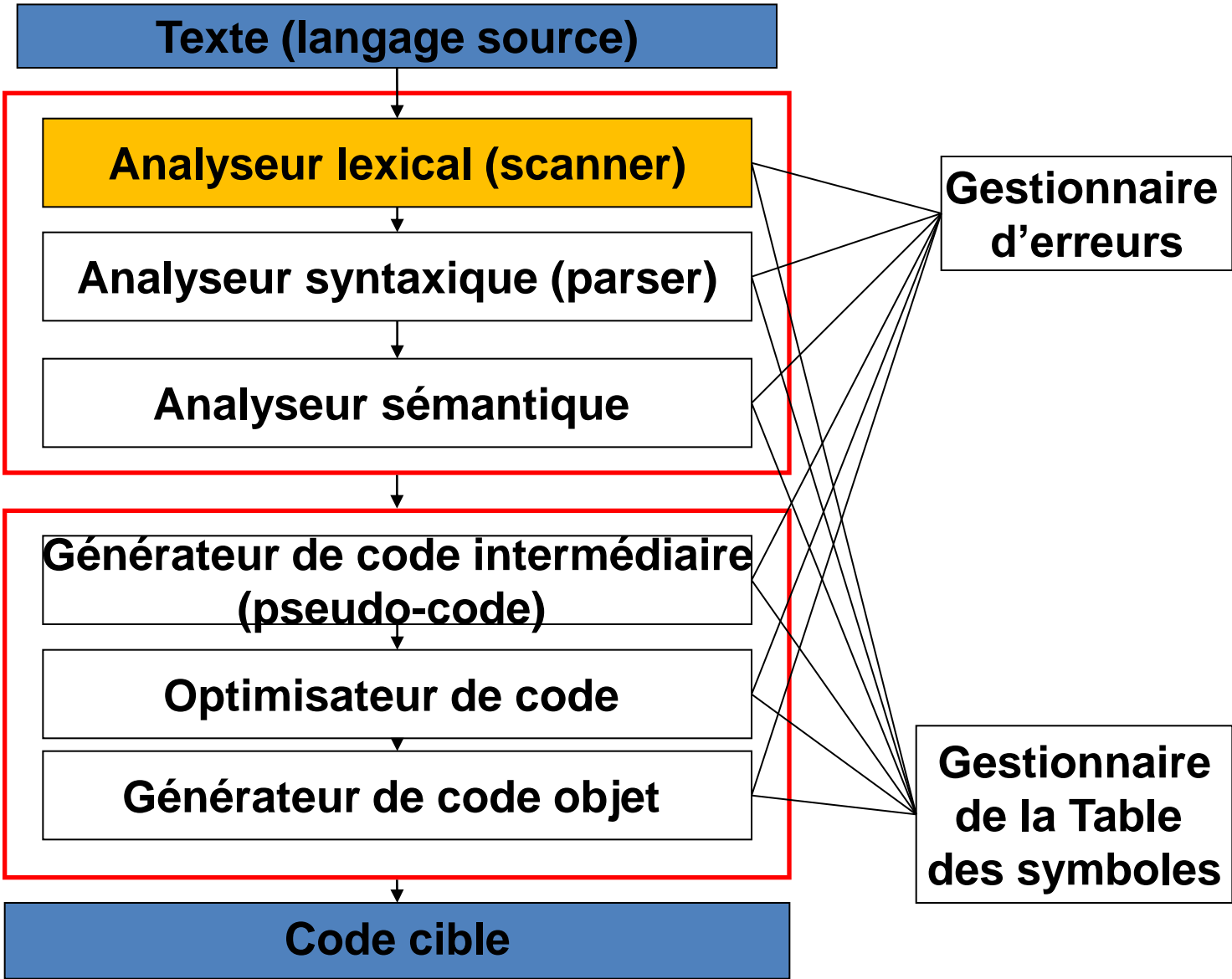


ECRITURE D'UN MINI COMPILATEUR LL(1)

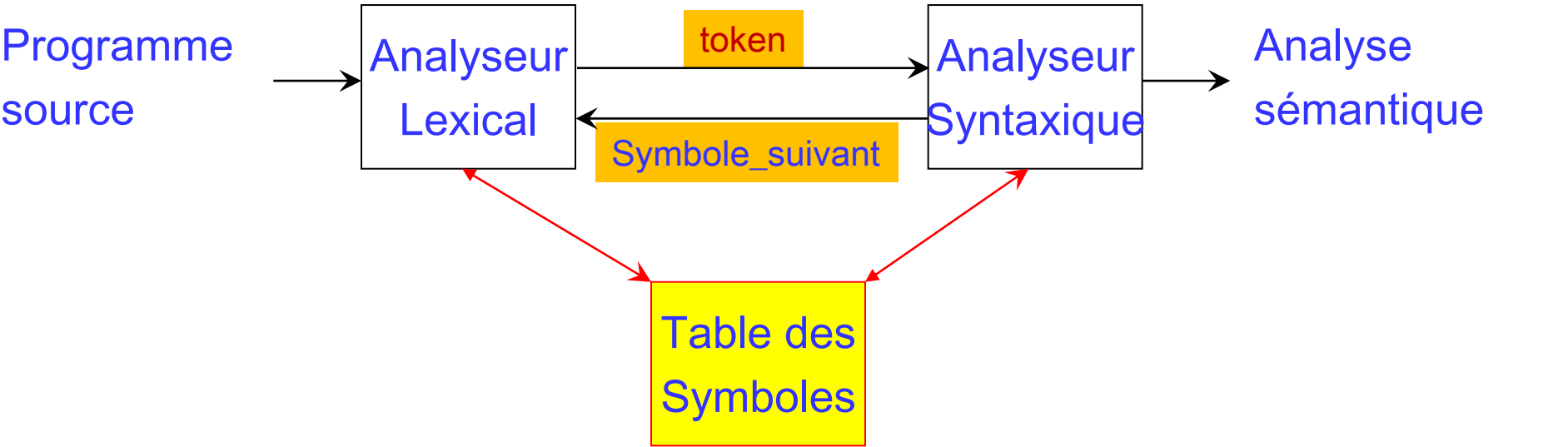
RAPPEL



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)

toto

+

A

*

3

Idf_token

plus_token

idf_token

mult_token

Cte_token

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_chîne 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
MINI COMPILATEUR
A ECRIRE EN SALLE DE TP**

LA GRAMMAIRE MINI PASCAL

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

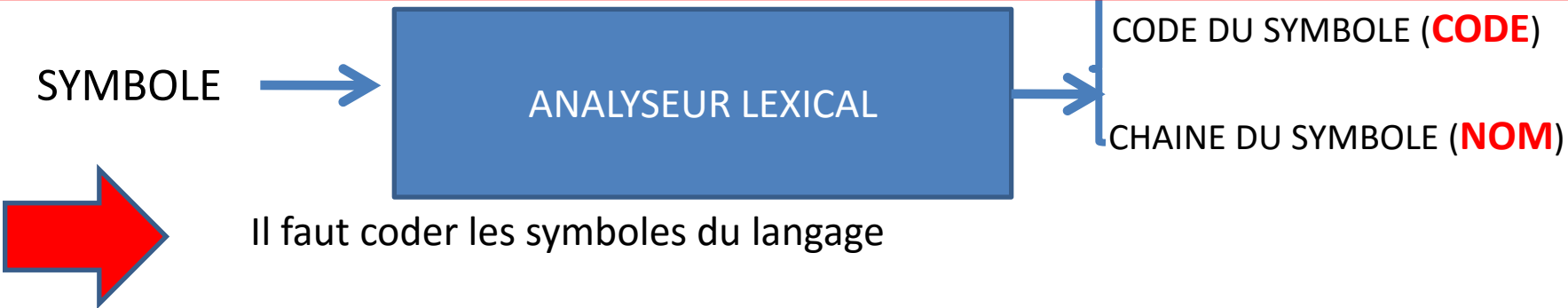
Exemples de programme Pascal

```
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



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 ;
```

SYMBOLE



LECTURE ET RECONNAISSANCE
CAR PAR CAR



VALEUR DU
SYMBOLE: NOM



CODAGE

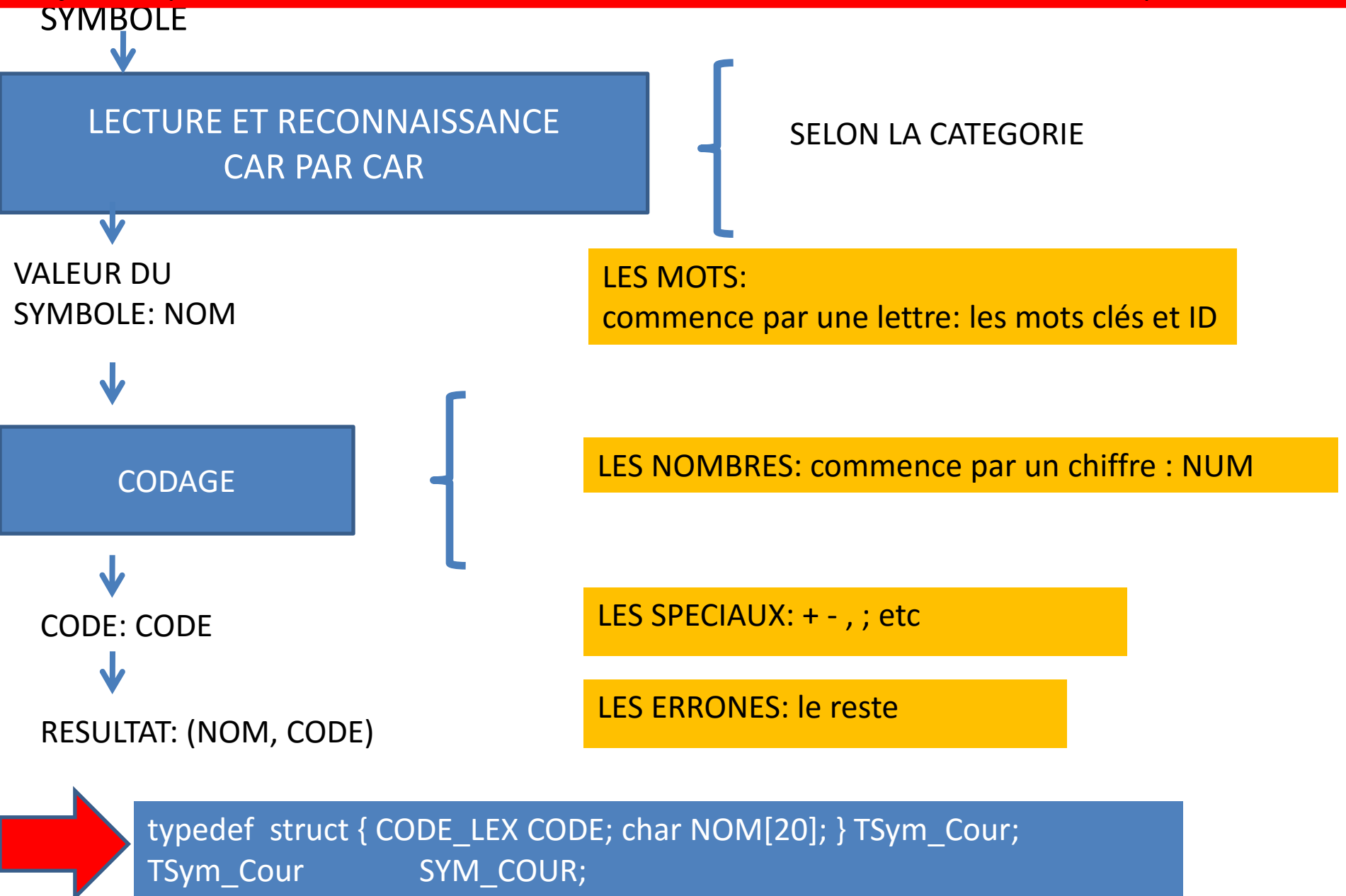


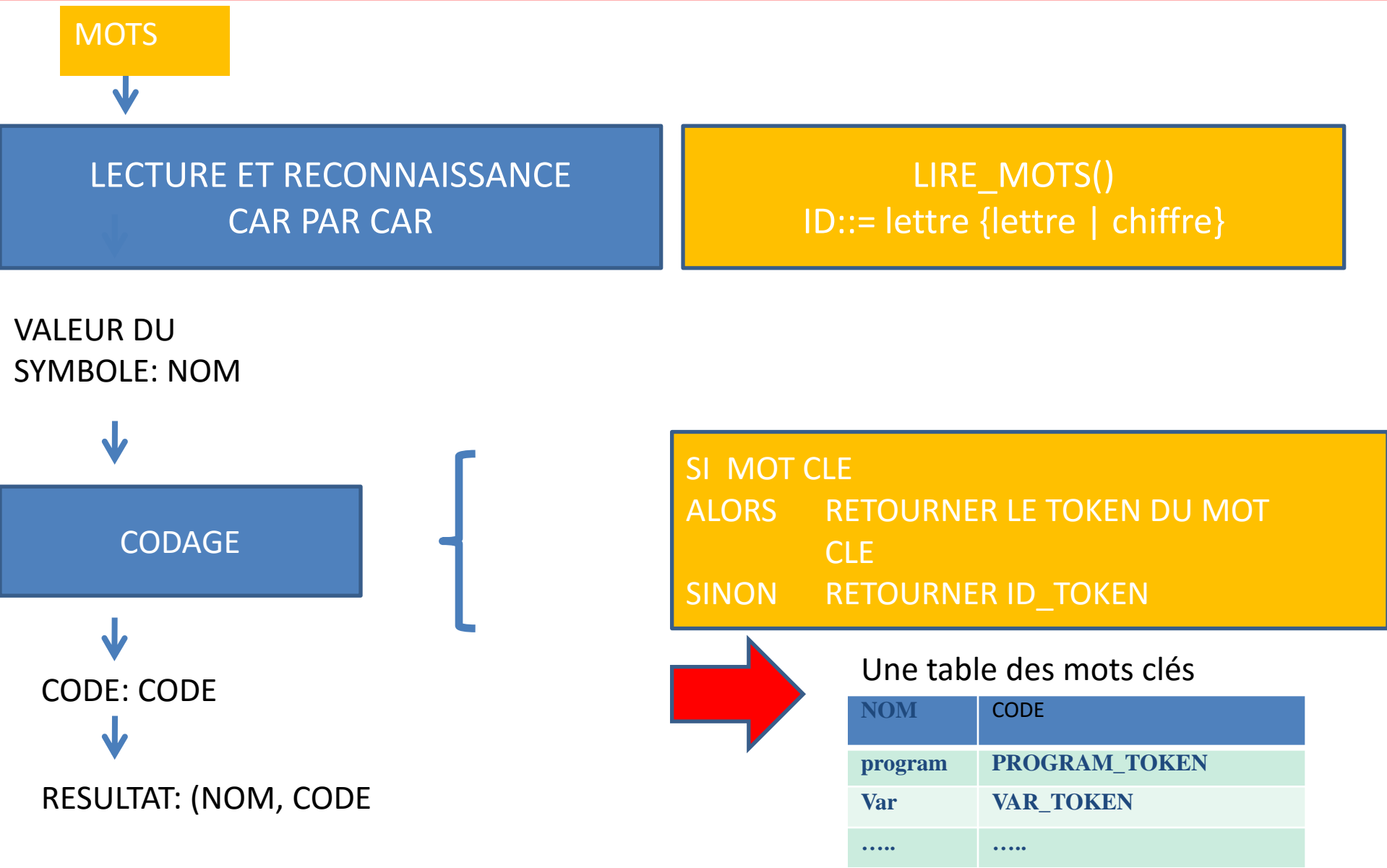
CODE: CODE

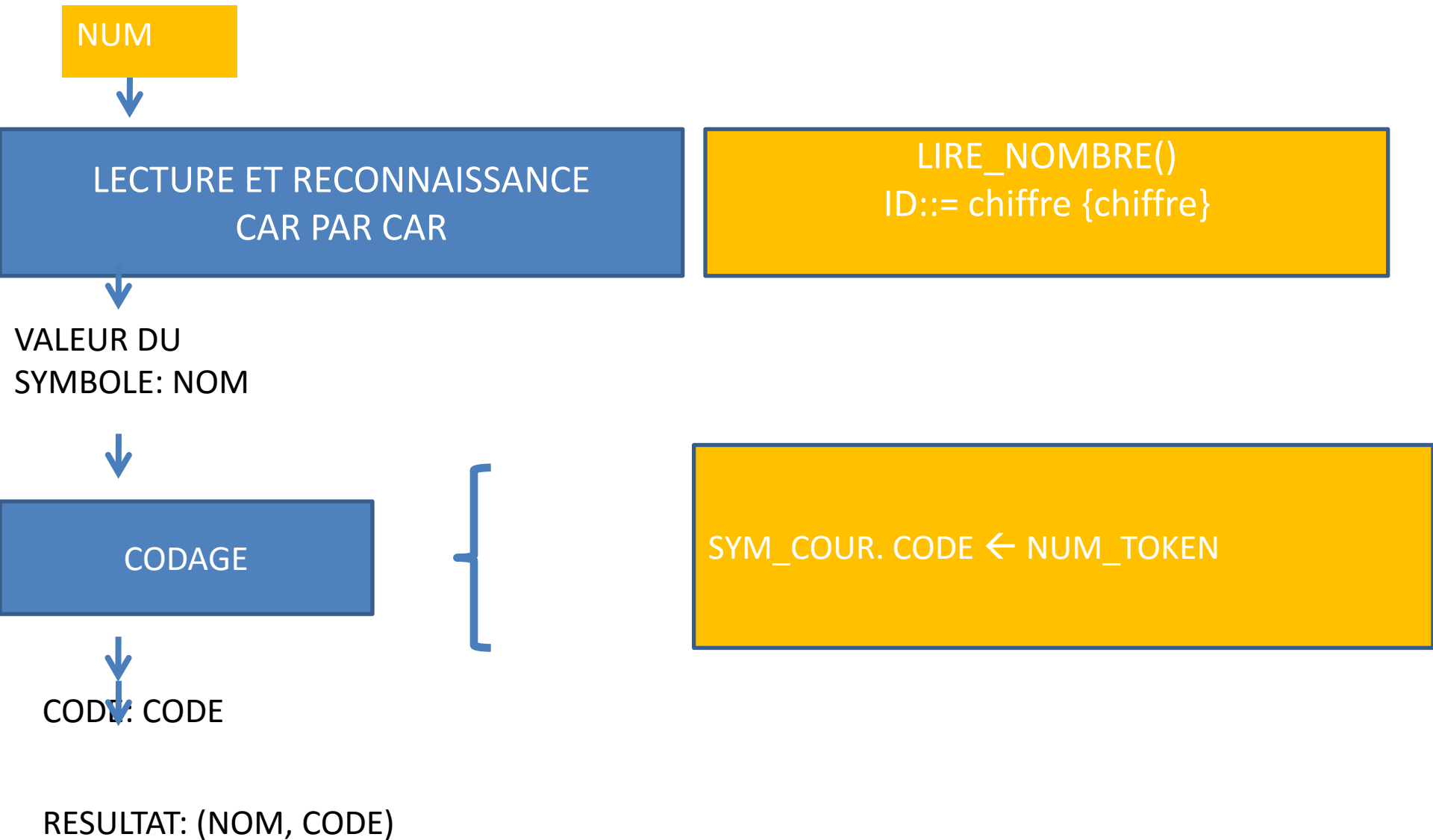


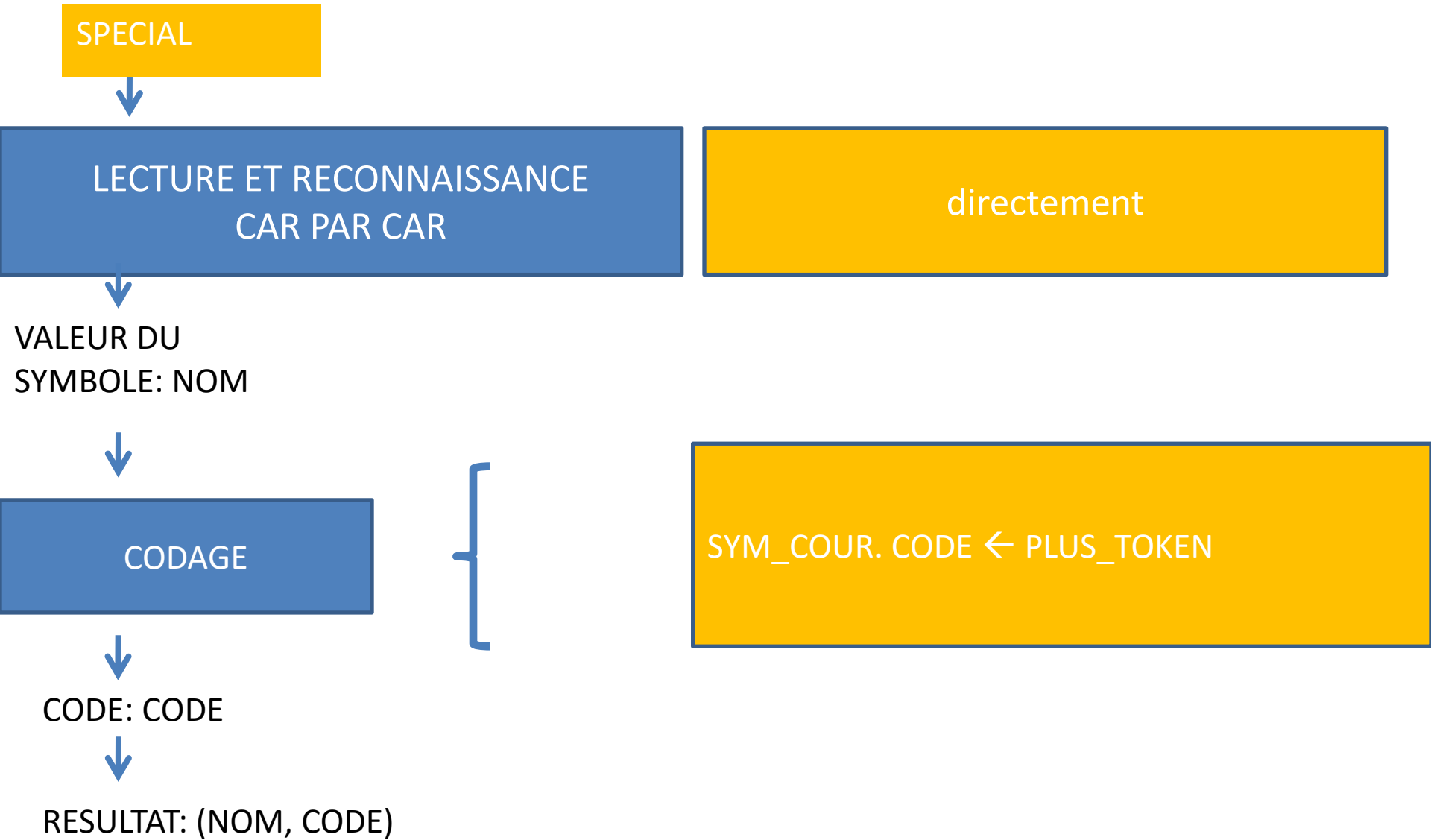
RESULTAT: (NOM, CODE)

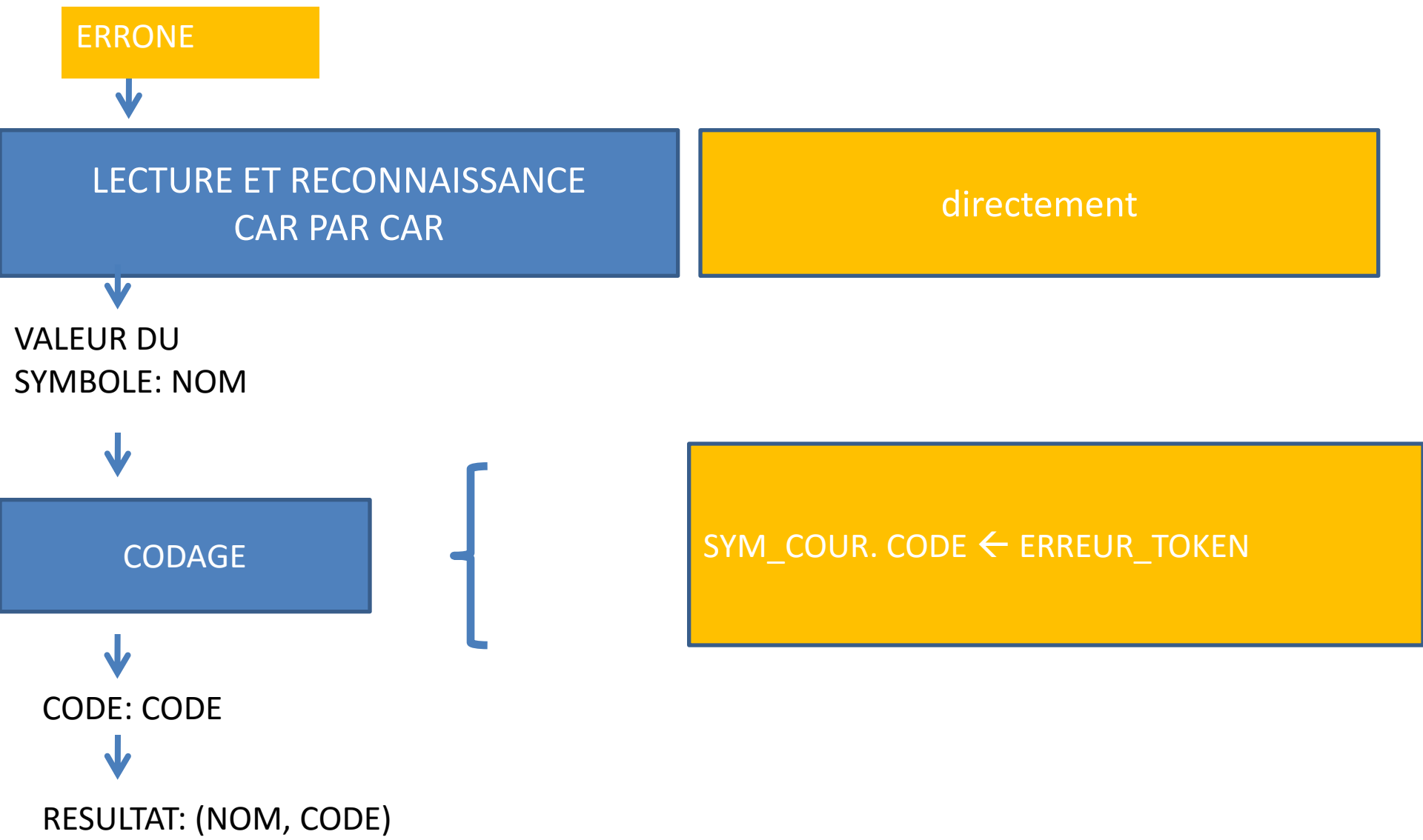
```
char Car_Cour; //caractère courant  
  
void Lire_Car(){  
    Car_Cour=fgetc(Fichier);  
}
```

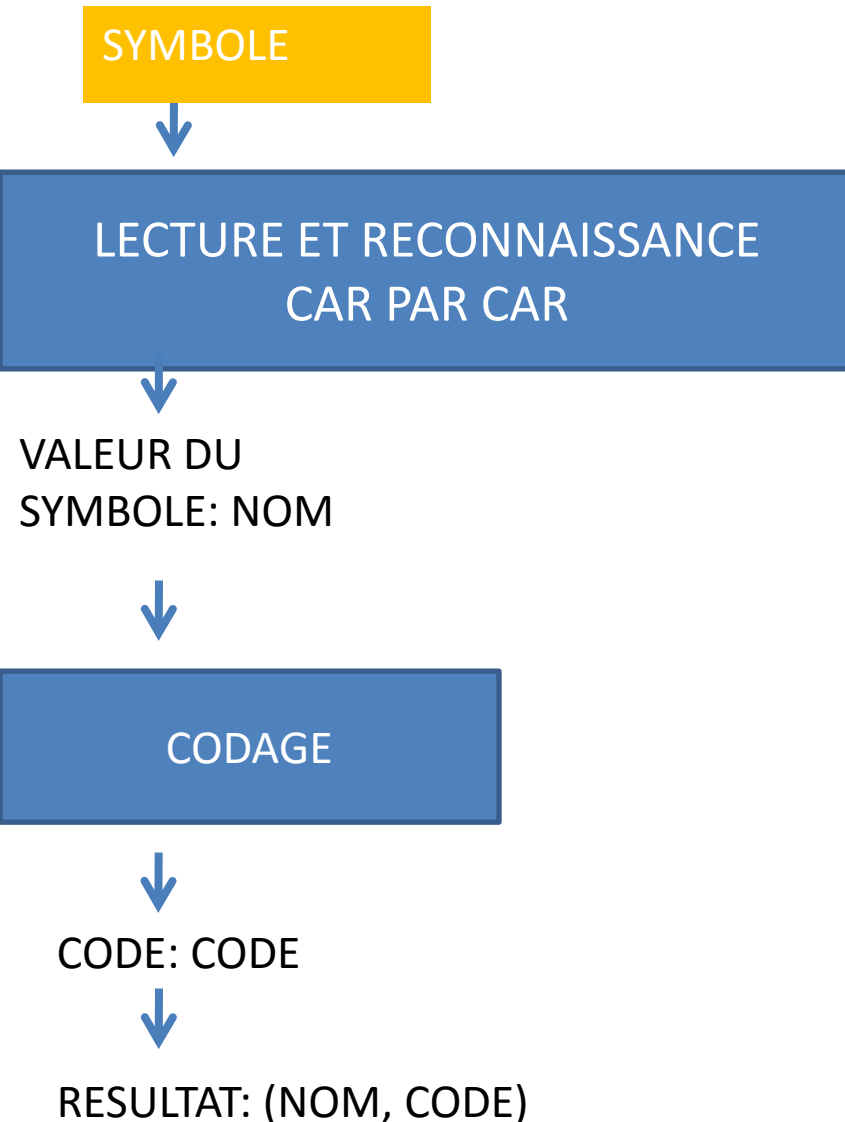












```
void Sym_Suiv(){
-- PASSER LES SEPARATEURS
-- TRAITER SELON LA CATEGORIE
--CATEGORIE DE MOTS
    si car_cour est une lettre : lire_mot();
--CATEGORIE DE NOMBRE
    si car_cour est un chiffre : lire_nombre();
--CATEGORIE DES SPECIAUX
    CAS CAR_COUR PARMi
        '+': SYM_COUR. CODE ← PLUS_TOKEN;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

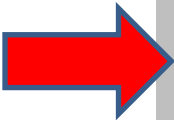
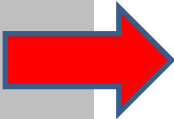
        SINON: SYM_COUR. CODE ← ERREUR_TOKEN;
                ERREUR(CODE_ERR);
    FINDECAS
}
```

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;

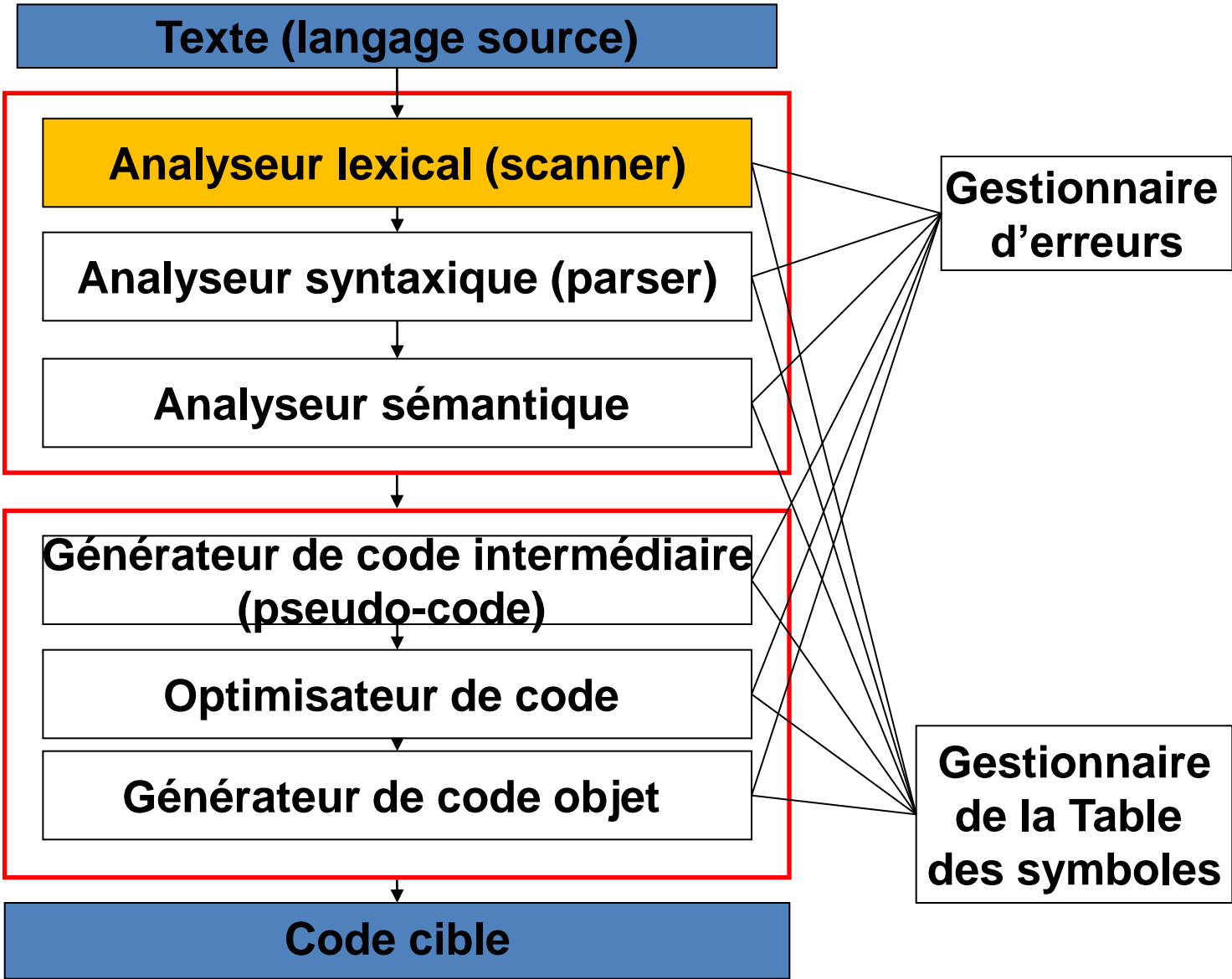
//-----
// DECLARATION DU TABLEAU DES ERREURS
//-----
typedef struct {  Erreurs  CODE_ERR; char mes[40]  }  Erreurs;
```

```
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);
}
```

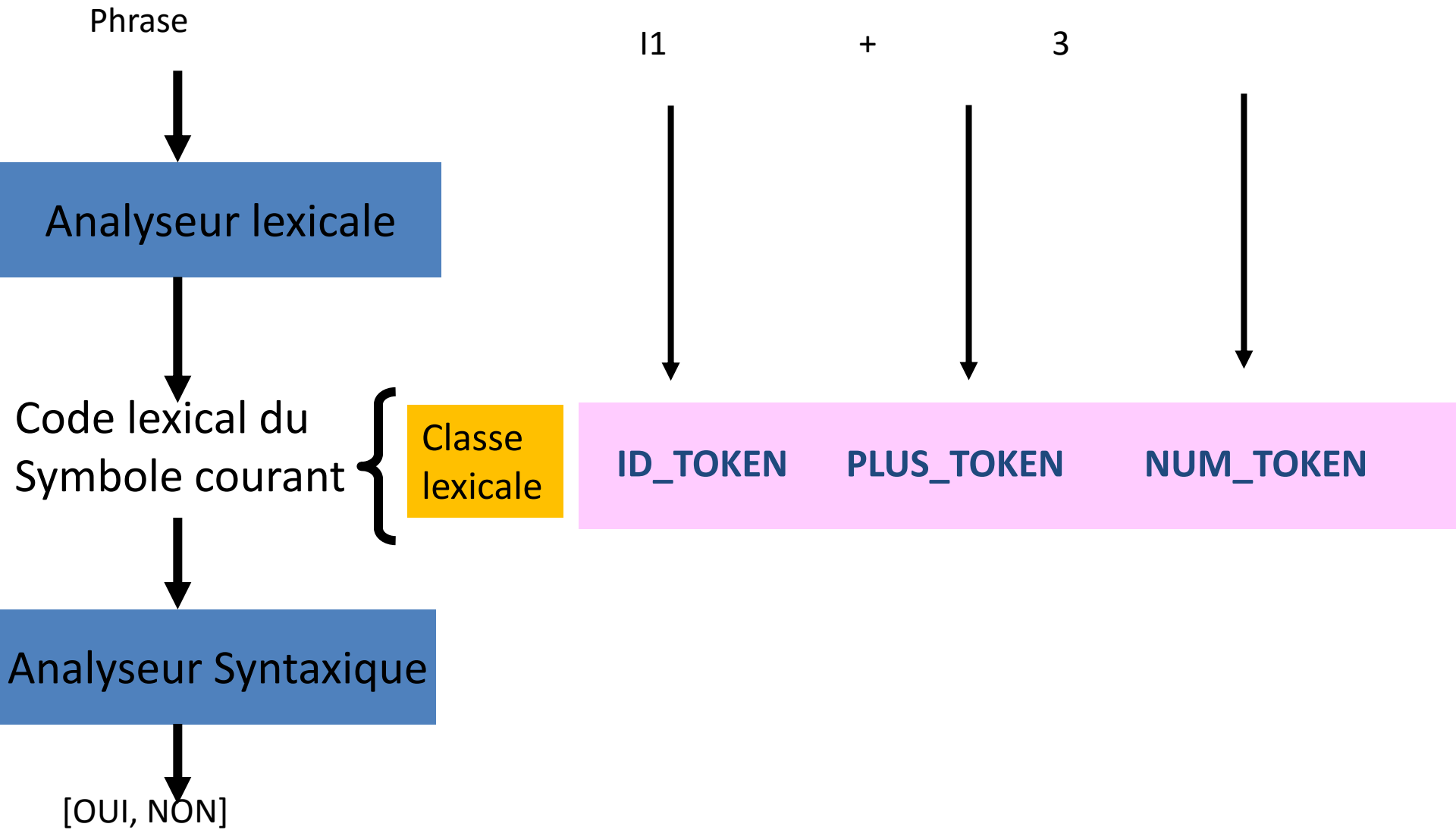
**A VOS MACHINES
et
BON COURAGE**

ANALYSEUR SYNTAXIQUE

PRINCIPE



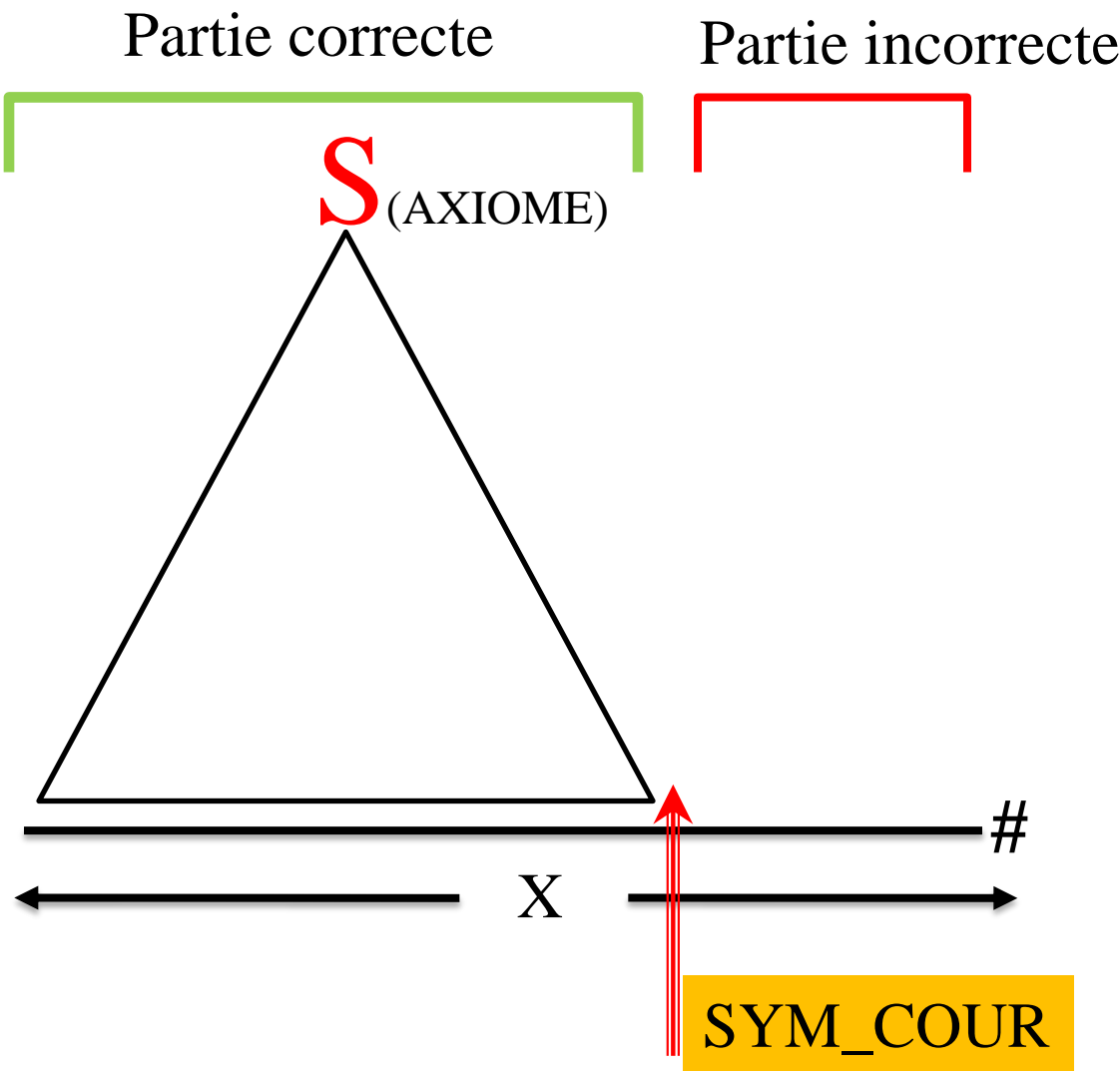
Architecture générale d'un compilateur



ANALYSEUR SYNTAXIQUE ECRITURE

PRINCIPE

SPECIFICATIONS DES TRAITEMENT DE L'ANALYSEUR



Spécification de l'analyseur syntaxique

L'analyseur LL(1) déterministe

Principe:

- A chaque règle grammaticale

$$A \rightarrow \alpha$$

on associe une procédure de la forme:

Procedure A;

Debut

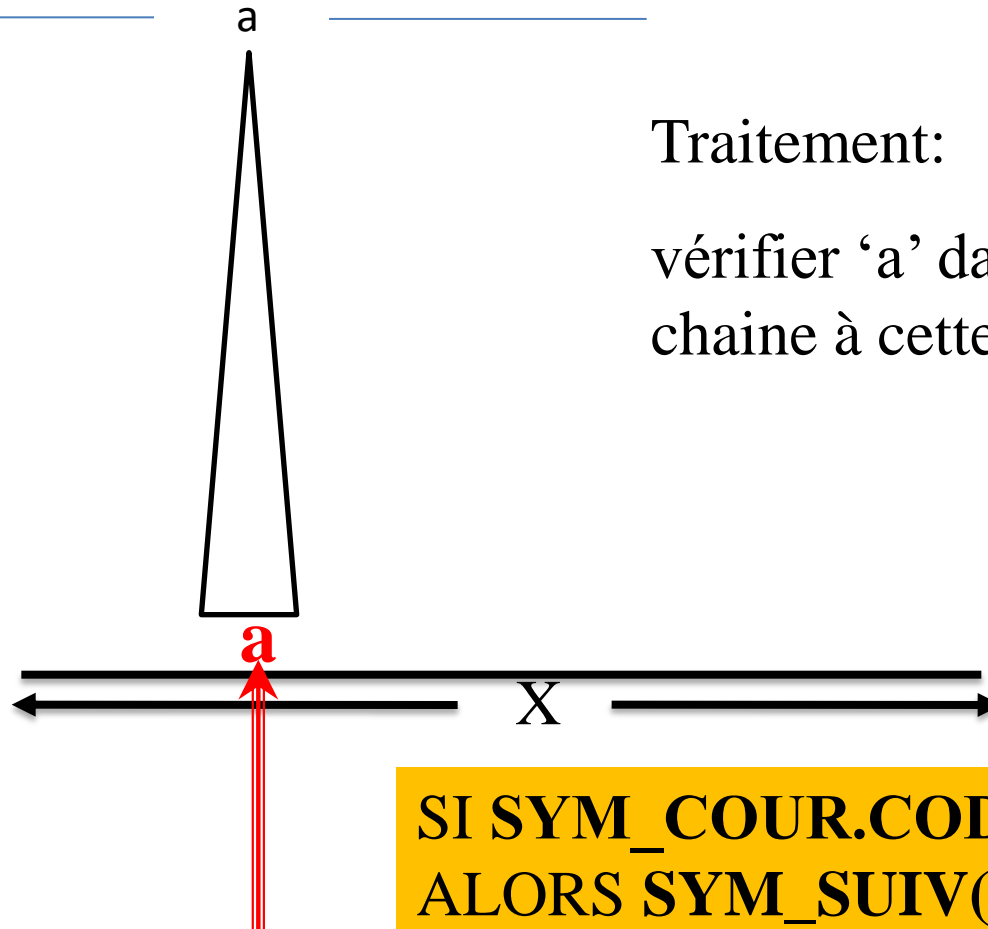
T(α);

Fin;

Où T(α) est le traitement associé à la partie droite de la règle A

Quelque soit la règle $A \rightarrow \alpha$, α contient l'une des formes suivantes

Composants de α
$a \in V_t$
$A \in V_n$
ε
$\beta_1\beta_2$
β^*
$\beta_1 \beta_2$

LE CAS D'UN TERMINAL

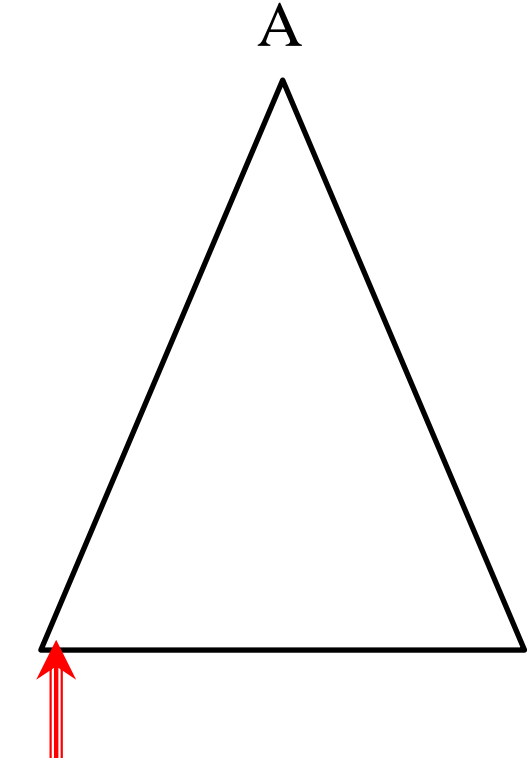
Traitement:

vérifier 'a' dans la
chaîne à cette position

```
SI SYM_COUR.CODE=CODE('a')  
ALORS SYM_SUIV()  
SINON ERREUR(mes);  
FIN SI;
```

CAS D'UN NON TERMINAL

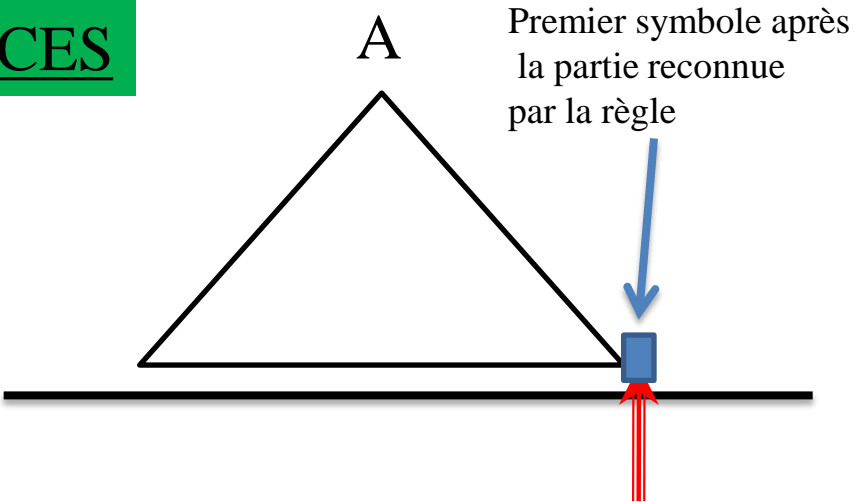
ETAT INITIAL



SYM_COUR

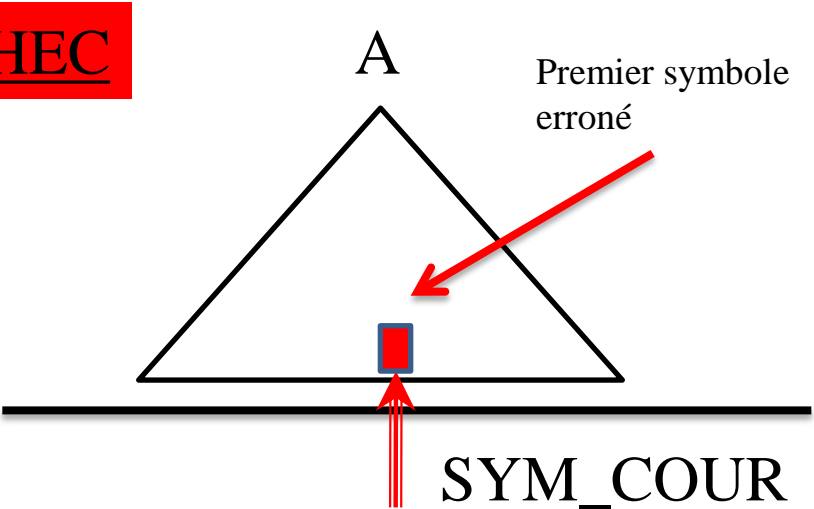
T(A)=A();

SUCCES

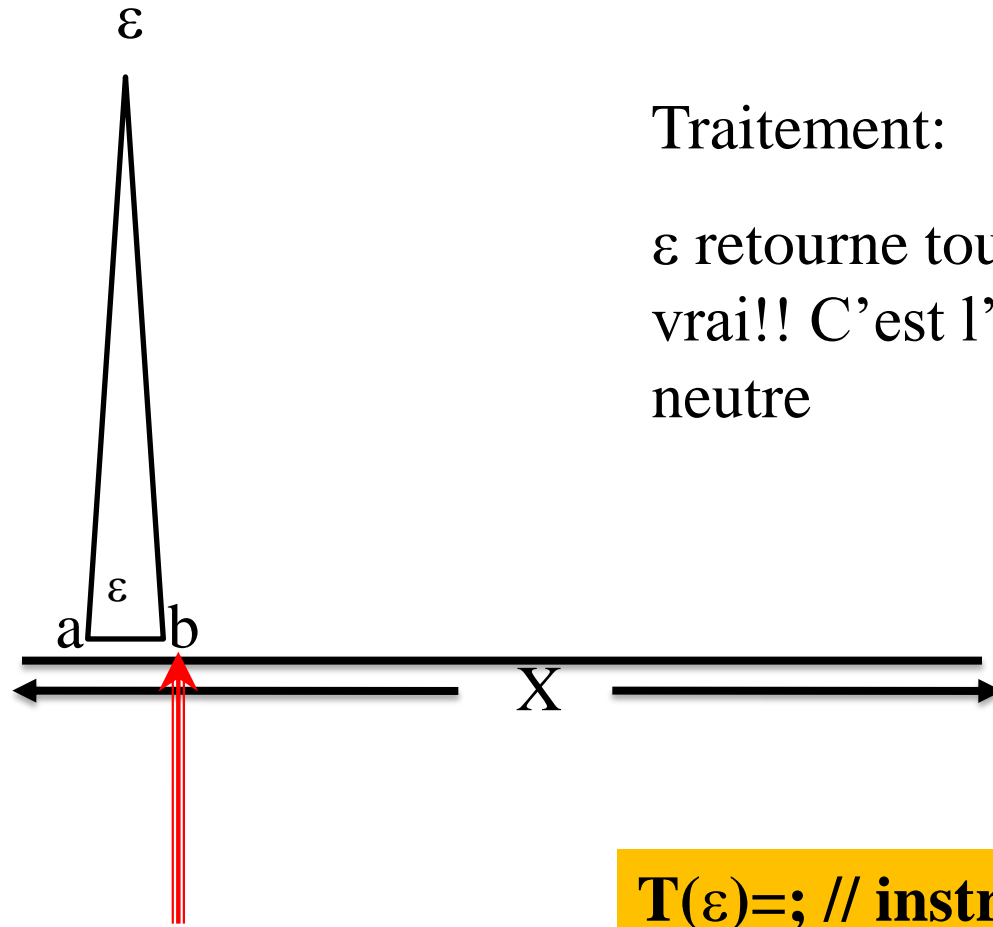


SYM_COUR

ECHEC



SYM_COUR

LE CAS DE ε 

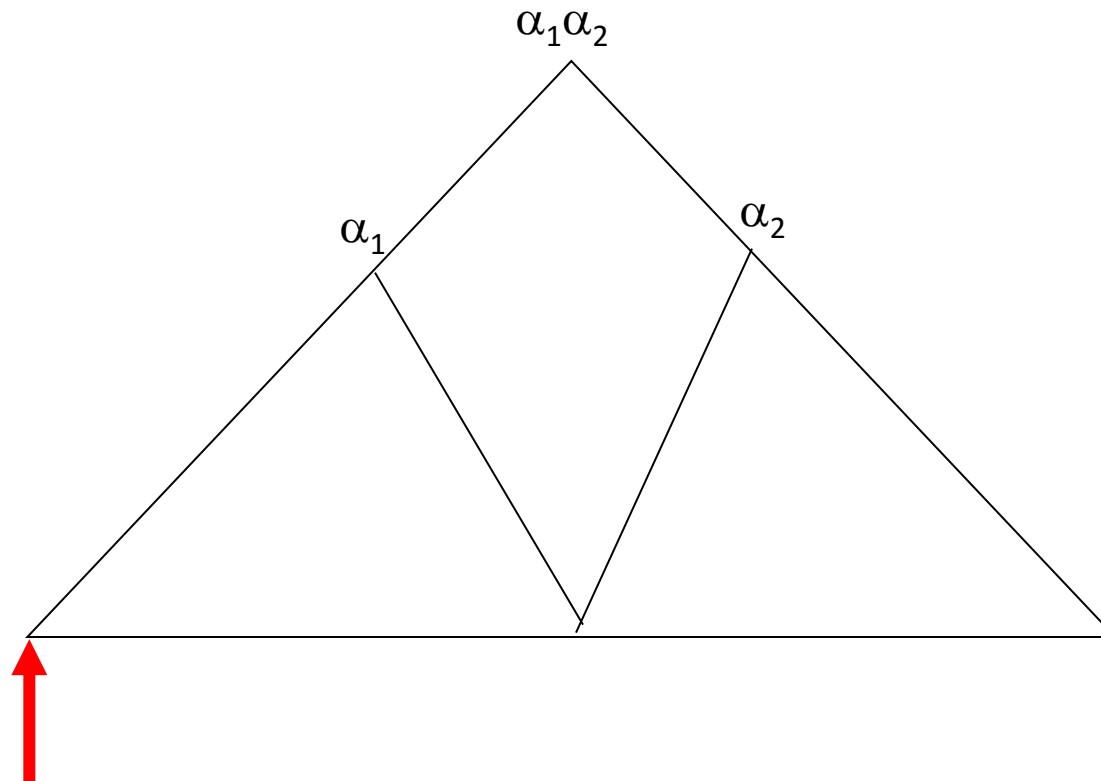
Traitement:

ε retourne toujours
vrai!! C'est l'élément
neutre

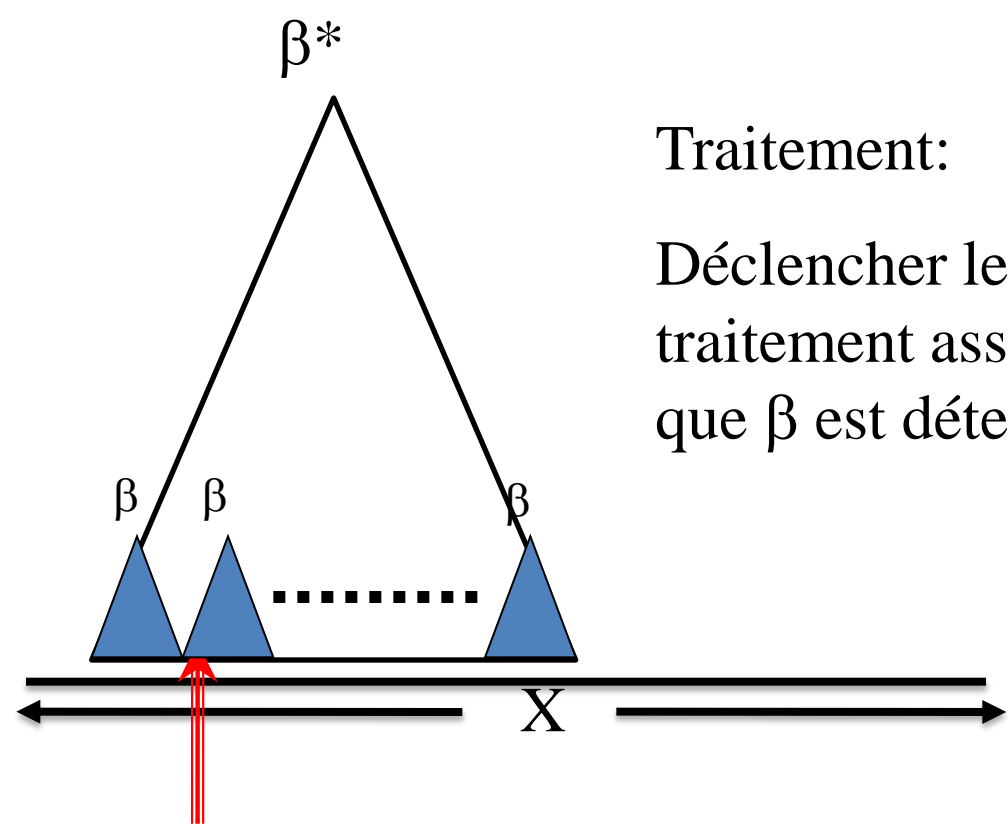
$T(\varepsilon) = ;$ // instruction vide

CAS DE $\alpha_1\alpha_2$

$$T(\alpha_1\alpha_2) = T(\alpha_1); T(\alpha_2)$$



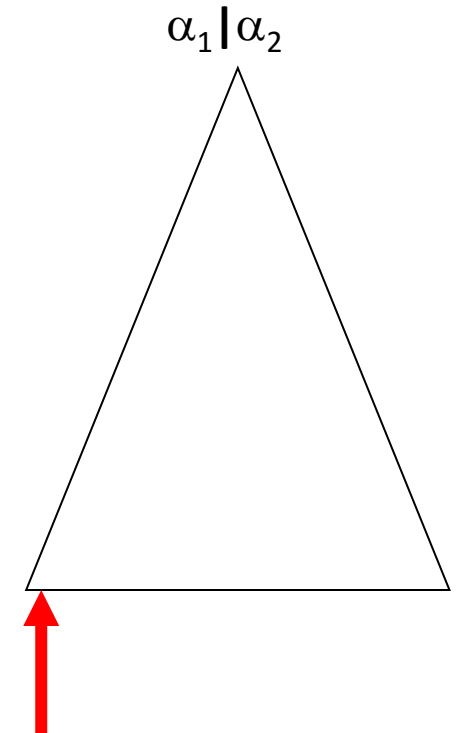
CAS DE β^*



Traitement:

Déclencher le traitement associé β tant que β est détectée!

```
T( $\beta^*$ )=TANTQUE SymCour.CODE dans FIRST( $\beta$ ) FAIRE
     $\zeta\beta$ ;
FINTANTQUE;
```

CAS DE $\alpha_1|\alpha_2$ **CAS SYM_COUR.CODE PARM****D($\alpha_1, \alpha_1|\alpha_2$): T(α_1);****D($\alpha_2, \alpha_1|\alpha_2$): T(α_2);****AUTRE CAS:****ERREUR(mes)****FIN DE CAS****SI ε appartient au L(α_2)****ALORS D($\alpha_2, \alpha_1|\alpha_2$)=FIRST($\alpha_2, \alpha_1|\alpha_2$) U FOLLOW($\alpha_2, \alpha_1|\alpha_2$)****SINON D($\alpha_2, \alpha_1|\alpha_2$)=FIRST($\alpha_2, \alpha_1|\alpha_2$)**

SYNTHESE

α	Traitement associé a α
$a \in V_t$ a_TOKEN	if (SymCour. CLS == a_TOKEN) SymboleSuivant ; else ERREUR(CODE_ERR) ;
$A \in V_n$	A(); // appel de la procédure associée à la règle A
ε	; //instruction vide
$\beta_1\beta_2$	$\zeta\beta_1$; $\zeta\beta_2$;
$\beta_1 \beta_2$	Switch (SymCour. CLS) { case D($\beta_1 \beta_2, \beta_1$) : $\zeta\beta_1$; break; case D($\beta_1 \beta_2, \beta_2$) : $\zeta\beta_2$; break; default ERREUR(CODE_ERR) }
β^*	while (SymCour.CLS in β') { $\zeta\beta$; }

ENSEMBLE DIRECTEURS EXEMPLE

Rien ne change:
on remplace les symboles par leurs classes lexicales: code

Exemple:

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**

(**const** ID = NUM ; { ID = NUM ; })' = { **CONST_TOKEN** }

Directeur(**const** ID = NUM ; { ID = NUM ; }) = { **CONST_TOKEN** }

ϵ " = { **VAR_TOKEN**, **BEGIN_TOKEN** }

Directeur(ϵ) = { **VAR_TOKEN**, **BEGIN_TOKEN** }

EXEMPLE DE PROCEDURE

```
//-----  
void Test_Symbole (Class_Lex cl, Erreurs COD_ERR){  
    if (Sym_Cour.cls == cl)  
    {  
        Sym_Suiv();  
    }  
    else  
        Erreur(COD_ERR);  
}
```

PROGRAM ::= program ID ; BLOCK .

```
void PROGRAM()  
{  
    Test_Symbole(PROGRAM_TOKEN, PROGRAM_ERR);  
    Test_Symbole(ID_TOKEN, ID_ERR);  
    Test_Symbole(PV_TOKEN, PV_ERR);  
    BLOCK();  
    Test_Symbole(PT_TOKEN, PT_ERR);  
}
```

BLOCK ::= CONSTS VARS INSTS

```
void BLOCK()
{
    CONSTS();
    VARS();
    INSTS();
}
```

CONSTS ::= const ID = NUM ; { ID = NUM ; } | ϵ

```
void CONSTS() {  
  
    switch (Sym_Cour.cls) {  
        case CONST_TOKEN : Sym_Suiv();  
                           Test_Symbole(ID_TOKEN, ID_ERR);  
                           Test_Symbole(EGAL_TOKEN, EGAL_ERR);  
                           Test_Symbole(NUM_TOKEN, NUM_ERR);  
                           Test_Symbole(PV_TOKEN, PV_ERR);  
                           while (Sym_Cour.cls==ID_TOKEN){  
                               Sym_Suiv();  
                               Test_Symbole(EGAL_TOKEN, EGAL_ERR);  
                               Test_Symbole(NUM_TOKEN, NUM_ERR);  
                               Test_Symbole(PV_TOKEN, PV_ERR);  
                           }; break;  
  
        case VAR_TOKEN:      break;  
        case BEGIN_TOKEN:   break;  
        default:             Erreur(CONST_VAR_BEGIN_ERR);break;  
    }  
}
```

RECAPITULONS

- C'est l'ensemble des procédures récursives
- Une procédure pour chaque règle syntaxique
- En général, s'il y a n non règle, il y a n procédures récursives qui s'entre appellent
- Les règles n'ont pas d'arguments;
- **SYM_COUR** est global et le code retourné par l'analyseur lexical est dans le champs **SYM_COUR.CODE**
- La procédure associée à l'axiome constitue le programme principal. C'est elle qui est appelée la première fois et celle qui appelle les autres.

```
int main(){
```

```
    Ouvrir_Fichier("C:\\PC\\Pascal.p");  
    PREMIER_SYM();
```

```
    PROGRAM();
```

```
    if (Sym_Cour.code==EOF_TOKEN)  
        printf("BRAVO: le programme est correcte!!!");  
    else  
        printf("PAS BRAVO: fin de programme erronée!!!!");
```

```
    getch();  
    return 1;  
}
```


Travail à faire:

- Programmer toutes les procédures pour toutes les règles syntaxiques.
- Tester l'analyseur syntaxique

Les erreurs:

A chaque symbole un code d'erreur et un message d'erreur

Exemples:

ERR_PROGRAM, ERR_BEGIN, ERR_ID,etc.

```
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 [= | <> | < | > | <= | >=] EXPR
EXPR         ::=  TERM { [+ | -] TERM }
TERM         ::=  FACT { [* | /] FACT }
FACT         ::=  ID | NUM | ( EXPR )
```

SI	::=	IFCOND THEN INST [ELSE INST ϵ]
REPETER	::=	REPEAT INST UNTIL COND
POUR	::=	FOR ID DO:= NUM [INTO DOWNT0] NUM DO INST
CAS	::=	CASE ID OF NUM : INST { NUM: INST } [ELSE INST ϵ] END

**A VOS MACHINES
et
BON COURAGE**

ANALYSE SEMANTIQUES

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**

- Les identificateurs et les constantes sont les objets sémantiques du programme
- Ils seront utilisés lors du calcul de l'analyse sémantique du programme



Il faut les mémoriser avec leurs propriétés



TABLE DES SYMBOLES

```
Exemple:  
program test;  
const toto=12; titi=23;  
var x, y;  
begin  
    ....  
end.
```

CH	CLS
test	ID_TOKEN
toto	ID_TOKEN
titi	ID_TOKEN
x	ID_TOKEN
y	ID_TOKEN

IDENTIFICATEURS



**IL FAUT INSERER LES IDENTIFICATEURS
DANS LA TABLE DES SYMBOLES**

MAIS



**Y a des règles sémantiques,
des contrôles sémantiques**



```
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
```

Règles sémantiques:


1. Règle 1: Toutes les déclarations dans CONSTS et VARS
2. Règle 2: PAS DE DOUBLE DECLARATIONS
3. Règle 3: Après BEGIN, tous les symboles doivent être déjà déclarés
4. Règle 4: Une constante ne peut changer de valeur dans le programme
5. Règle 5: Le ID du programme ne peut être utilisé dans le programme

Exemple 1:

```
program test;  
const tata=12;  
var x;  
begin  
  titi:=tata;  
end.
```

**ERR: identificateur titi non déclaré****Exemple 2:**

```
program test;  
const tata =12;  
var x,tata ;  
begin  
  x:=tata;  
end.
```

**ERR: double déclaration**

Exemple 3:

```
program test;  
const tata=12;  
var x;  
begin  
  tata :=15;  
end.
```

ERR: constante ne peut changer de valeur

Exemple 4:

```
program test;  
const tata=12;  
var x;  
begin  
  x :=test ;  
end.
```

ERR: nom de programme non autorisé

Exemple 5:

```
program test;  
const tata=12;  
var x;  
begin  
  read(tata );  
end.
```

ERR: constante ne peut changer de valeur

```
Exemple:  
program test;  
const toto=12; titi=23;  
var x, y;  
begin  
    ....  
end.
```

CH	TIDF
test	TPROG
toto	TCONST
titi	TCONST
x	TVAR
y	TVAR

IDENTIFICATEURS

```
Typedef enum {TPROG, TCONST, TVAR} TSYM;
```

```
Typedef struct {  
    char  NOM[20];  
    TSYM TIDF;  
} T_TAB_IDF;  
T_TAB_IDF      TAB_IDFS[NbrIDFS];
```

TRAITEMENT SEMANTIQUE DES NOMBRES

Règles sémantiques:

1. ON NE STOCKE PAS LES NOMBRE DANS LA TABLE DES SYMBOLES
2. ON CONVERTIT LES NOMBRES DANS LEUR VALEUR NUMERIQUE

```
// type du symbole courant
Typedef struct TSym_Cour{
    Class_Lex      cls;
    char           nom[20];
    int            val;
};
```

```
void Lire_Nbr(){  
  
    char mot[20];  
    int i=0;  
    while(Car_Cour>='0' && Car_Cour<='9'){  
        Sym_Cour.nom[i]=Car_Cour;  
        i++;  
        Lire_Caractere();  
    }  
    Sym_Cour.nom[i]='\0';  
    Sym_Cour.val=atoi(Sym_Cour.nom);  
    Sym_Cour.cls=NUM_TOKEN;  
}
```

A FAIRE POUR CETTE SEANCE

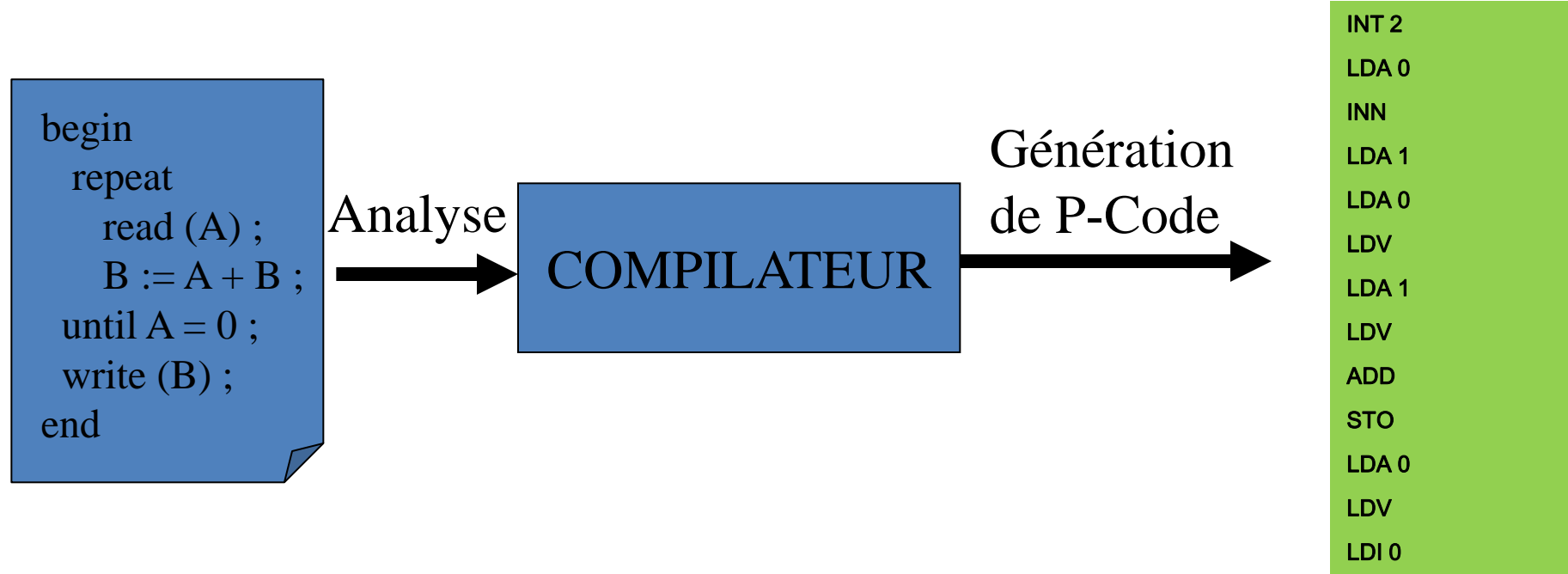
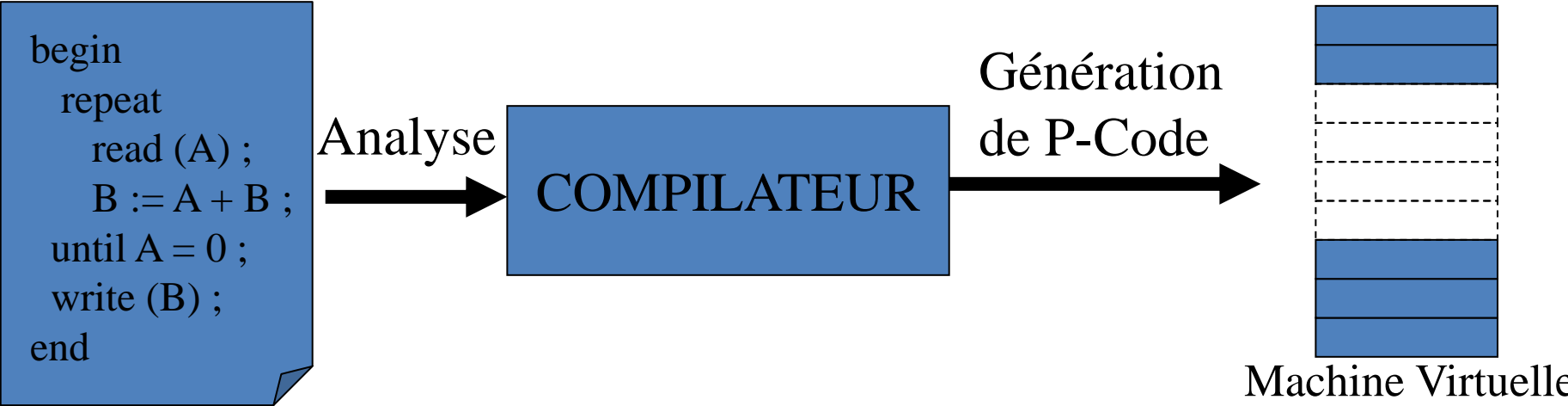
1. LES DECLARATIONS NECESSAIRES
2. MODIFIER LA FONCTION DE CODAGE LEXICALE POUR TENIR COMPTE DES CONTROLES SEMANTIQUES DES DECLARATIONS

**A VOS MACHINES
et
BON COURAGE**

GENERATION DE CODE

```
PROGRAM ::= program ID ; BLOCK .
BLOCK    ::= CONSTS VARS INSTS
CONSTS   ::= const ID = NUM ; { ID = NUM ; } |  $\epsilon$ 
VARS     ::= var ID { , ID } ; |  $\epsilon$ 
INSTS    ::= begin INST { ; INST } end
INST     ::= INSTS | AFFEC | SI | TANTQUE | ECRIRE | LIRE |  $\epsilon$ 
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 )
```

PRINCIPE DE TRAITEMENTS SEMANTIQUE ACTIONS SEMANTIQUES



AFFEC ::= ID := EXPR

CHARGER L'ADRESSE DU
ID AU SOMMET DE LA PILE

LE RESULTAT DE EXPR AU SOMMET DE
LA PILE, STOCKER LE SOMMET A
L'ADRESSE MÉMOIRE DU ID

//-----

// Procedure syntaxique de la règle:

// AFFEC ::= ID := EXPR

//-----

void AFFEC()

{

Test_Symbole(ID_TOKEN, ID_ERR);

CHARGER L'ADRESSE ID AU SOMMET DE LA PILE

Test_Symbole(AFFECT_TOKEN, AFFECT_ERR);

EXPR();

LE RESULTAT DE EXPR AU SOMMET DE LA PILE, STOCKER LE
SOMMET DE A L'ADRESSE DU ID

**ACTIONS
SEMANTIQUES**

}

AFFEC ::= ID := EXPR

CHARGER L'ADRESSE DU
ID AU SOMMET DE LA PILE



LE RESULTAT DE EXPR AU
SOMMET DE LA PILE
STOCKER LE SOMMET DE
A L'ADRESSE DU ID



AFFEC ::= ID := EXPR

GENERER1(STO)

GENERER2(LDA,TABSYM[IND_DER_SYM_ACC].ADRESSE);

AFFEC ::= ID := EXPR

GENERER1(STO)

GENERER2(LDA,TABSYM[IND_DER_SYM_ACC].ADRESSE);

// -----

// Procedure syntaxique de la règle:

// AFFEC ::= ID := EXPR

// -----

void AFFEC()

{

Test_Symbole(ID_TOKEN, ID_ERR);

GENERER2(LDA,TABSYM[IND_DER_SYM_ACC].ADRESSE);

Test_Symbole(AFFECT_TOKEN, AFFECT_ERR);

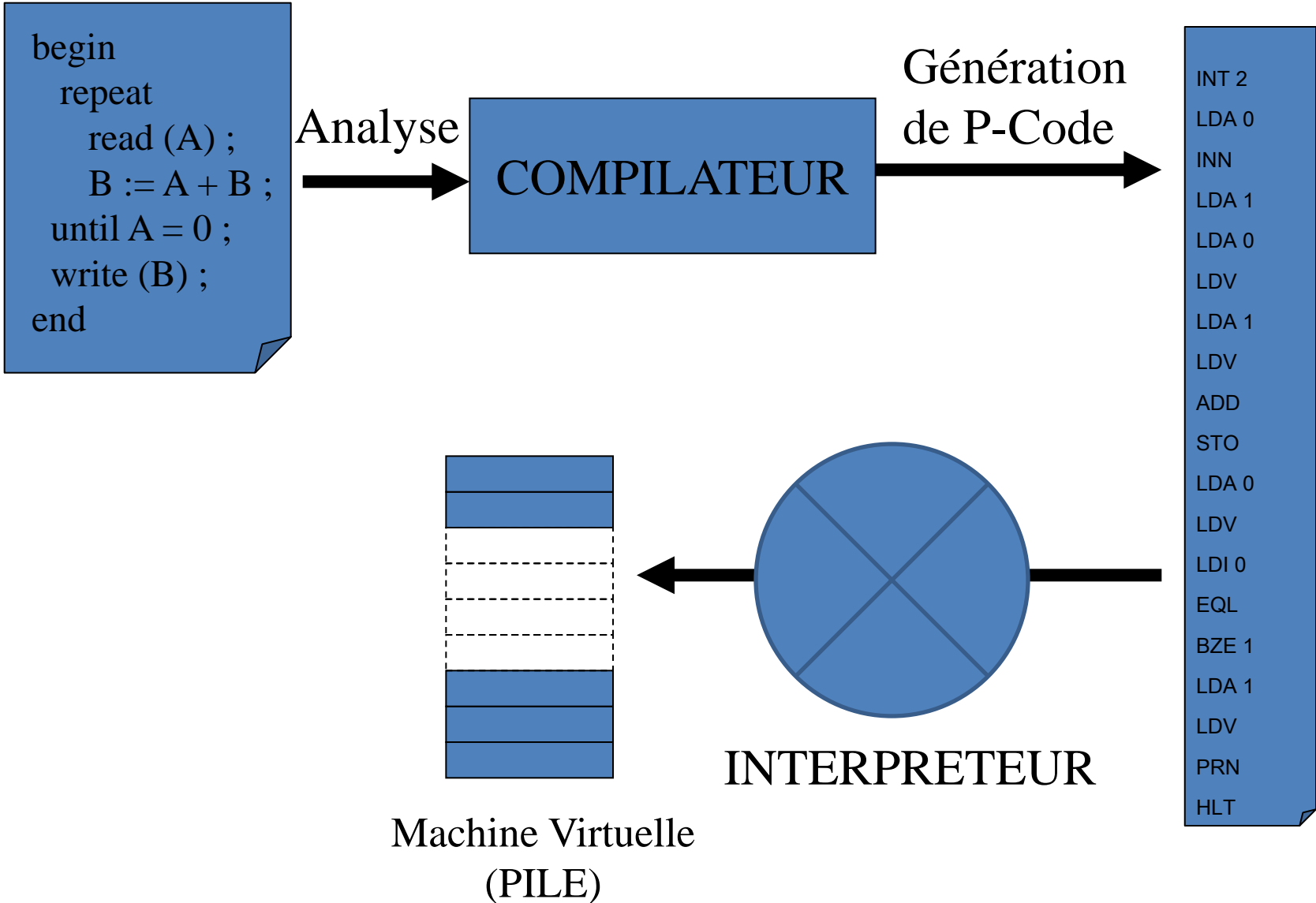
EXPR();

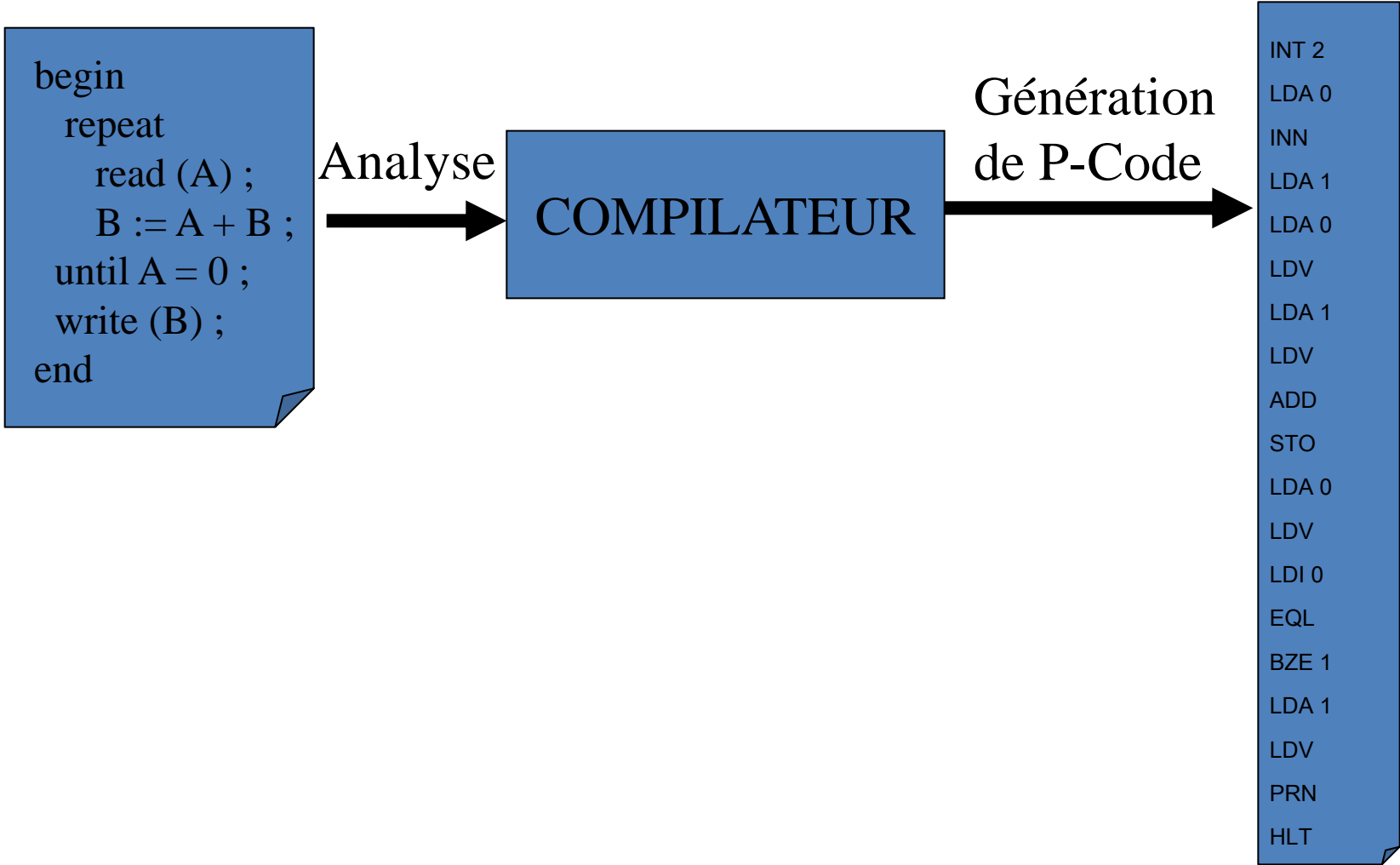
GENERER1(STO)

ACTIONS
SEMANTIQUES

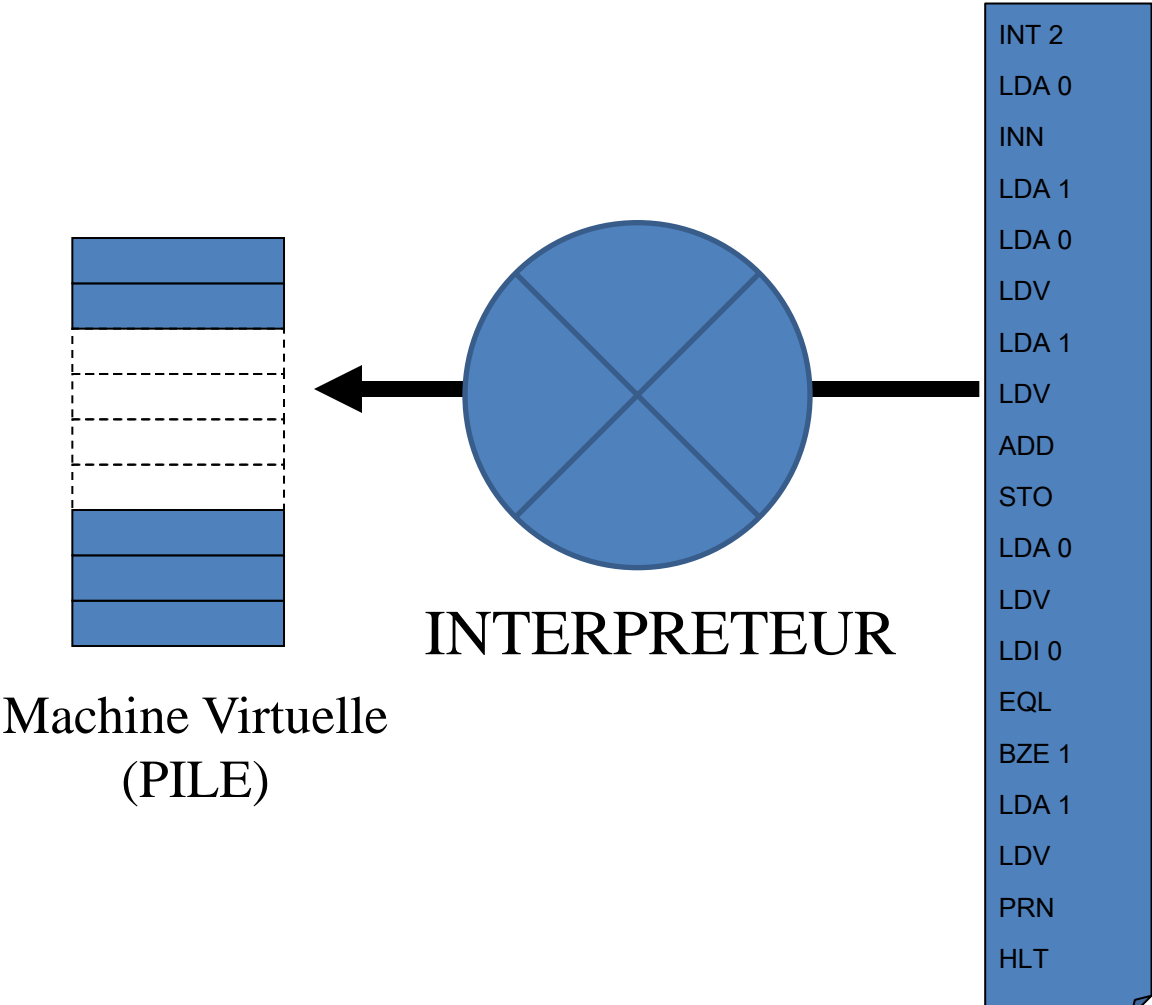
}

ARCHITECTURE GOLABLE





1^{ère} PARTIE: GENERATION DE CODE



2^{ème} PARTIE: INTERPRETATION DU CODE GENE

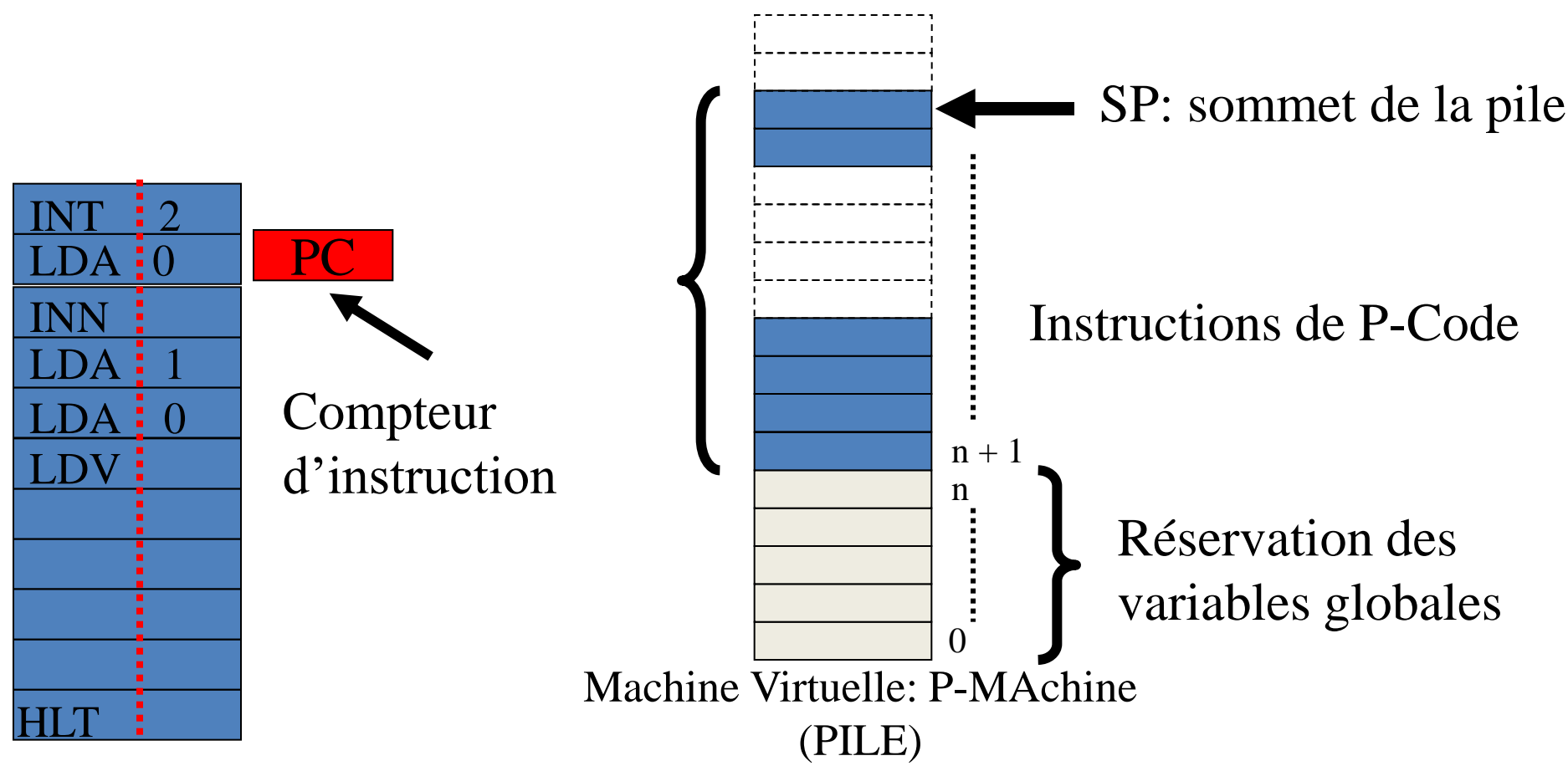
JEU DE CODE MACHINE

ADD	additionne le sous-sommet de pile et le sommet, laisse le résultat au sommet (idem pour SUB, MUL, DIV)
EQL	laisse 1 au sommet de pile si sous-sommet = sommet, 0 sinon (idem pour NEQ, GTR, LSS, GEQ, LEQ)
PRN	imprime le sommet, dépile
INN	lit un entier, le stocke à l'adresse trouvée au sommet de pile, dépile
INT c	incrémente de la constante c le pointeur de pile (la constante c peut être négative)
LDI v	empile la valeur v
LDA a	empile l'adresse a
LDV	remplace le sommet par la valeur trouvée à l'adresse indiquée par le sommet (déréférence)
STO	stocke la valeur au sommet à l'adresse indiquée par le sous-sommet, dépile 2 fois
BRN i	branchement inconditionnel à l'instruction i
BZE i	branchement à l'instruction i si le sommet = 0, dépile
HLT	halte

jeu d'instruction du P-Code simplifié

ENVIRONNEMENT D'EXECUTION ET GENERATION DE CODE

Le P-Code est le langage intermédiaire utilisé pour le Pascal.
Il est associé à la machine abstraite P-Machine composée de:



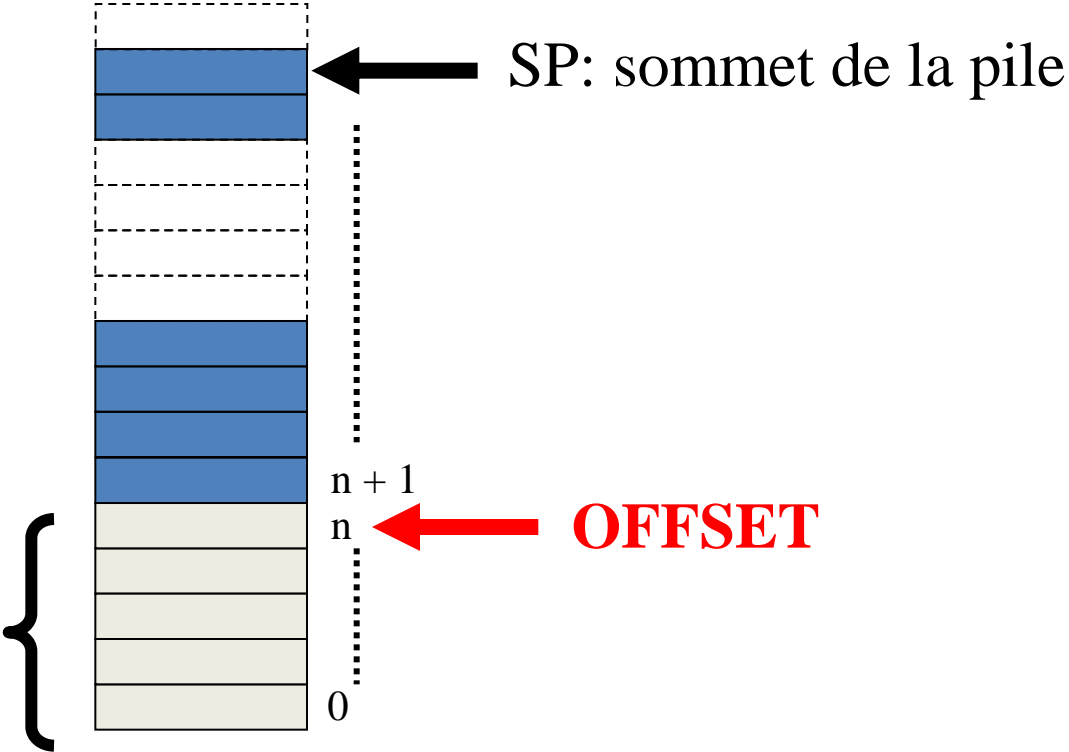
ENVIRONNEMENT

LA MEMOIRE

```
var
TABLESYM : tableau [TABLEINDEX] de enregistrement
    NOM : ALFA ;
    CLASSE : CLASSES ;
    ADRESSE : ENTIER
fin ;
OFFSET : ENTIER ;
```

Table des symboles

Réservation des
variables globales



ENVIRONNEMENT DE GENERATION

LES DECLARATIONS

Les structures de données nécessaires lors de l'écriture d'un interprète simplifié pour le P-Code sont :

un tableau MEM représentant la pile de la machine et un pointeur de pile associé

```
var
    MEM : TABLEAU [0 .. TAILLEMEM] DE ENTIER ;
    SP : ENTIER ;
```

```
Type MNEMONIQUES = (ADD,SUB,MUL,DIV,EQL,NEQ,GTR,
                    LSS,GEQ,LEQ, PRN,INN,INT,LDI,LDA,LDV,
                    STO,BRN,BZE,HLT) ;
```

```
INSTRUCTION =      enregistrement
                    MNE : MNEMONIQUES ;
                    SUITE : entier
                    fin
VAR PCODE : tableau [0 .. TAILLECODE] de INSTRUCTION ;
    PC : entier ;
```

Les structures de données nécessaires lors de l'écriture d'un interprète simplifié pour le P-Code sont :

un tableau MEM représentant la pile de la machine et un pointeur de pile associé

```
Type MNEMONIQUES = (ADD,SUB,MUL,DIV,EQL,NEQ,GTR,  
                     LSS,GEQ,LEQ, PRN,INN,INT,LDI,LDA,LDV,  
                     STO,BRN,BZE,HLT) ;
```

```
INSTRUCTION =      enregistrement  
                   MNE : MNEMONIQUES ;  
                   SUITE : entier  
                   fin
```

```
VAR PCODE : tableau [0 .. TAILLECODE] de INSTRUCTION ;  
    PC : entier ;
```

Les structures de données nécessaires lors de l'écriture d'un interprète simplifié pour le P-Code sont :

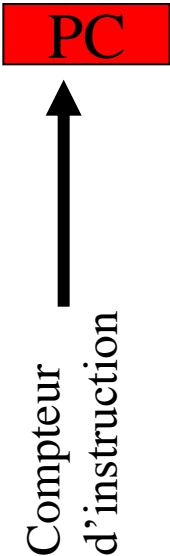
un tableau PCODE représentant les instructions de P-Code et le compteur associé PC

```
Type MNEMONIQUES = (ADD,SUB,MUL,DIV,EQL,  
                    NEQ,GTR,LSS,GEQ,LEQ, PRN,  
                    INN,INT,LDI,LDA,LDV,STO,BRN,  
                    BZE,HLT) ;
```

```
INSTRUCTION = enregistrement  
                MNE : MNEMONIQUES ;  
                SUITE : entier  
            fin  
VAR PCODE : tableau [0 .. TAILLECODE] de INSTRUCTION ;  
    PC=0 : entier ;
```

```
VAR OFFSET=-1;
```

INT	2
LDA	0
INN	
LDA	1
LDA	0
LDV	
HLT	



LES FONCTIONS DE GENERATION DE CODE

Procédures de génération de P-Code

```
procedure GENERER1 (M:MNEMONIQUES) ;  
debut  
    si PC = TAILLECODE  
        alors ERREUR ;  
    PC := PC + 1 ;  
    PCODE [PC]. MNE := M  
fin ;
```

INT	2
LDA	0
INN	
LDA	1
LDA	0
LDV	
INT	2
LDA	0
INN	
LDA	1
LDA	0
LDV	
M	



```
procedure GENERER2 (M:MNEMONIQUES ; A:entier) ;  
debut  
    si PC = TAILLECODE  
        alors ERREUR ;  
    PC := PC + 1 ;  
    PCODE [PC].MNE := M ;  
    PCODE [PC].SUITE := A  
fin;
```

INT	2
LDA	0
INN	
LDA	1
LDA	0
LDV	

PC

PC

INT	2
LDA	0
INN	
LDA	1
LDA	0
LDV	
M	A

PC

Réservation des emplacements mémoires pour les variables et les constantes

LES SYMBOLES N'EXISTENT QUE POUR
LE DEVELOPPEUR,

POUR LA MACHINE, IL N'Y A QUE DES
EMPLACEMENT MEMOIRES

→ Faut réserver une zone mémoire pour chaque
Variable et "constante"

→ LES CONSTANTES PEUVENT ETRE TRAITEES COMME DES VARIABLES INITIALISEES
OU REMPLACEES PAR LEURS VALEURS LORS DE LA GENERATION DE CODE

$\text{CONSTS} ::= \text{const ID} = \text{NUM} ; \{ \text{ID} = \text{NUM} ; \} \mid \varepsilon$

TABSYM[IND_DER_SYM_ACC].ADRESSE=++OFFSET;

RESERVATION DE LA MÉMOIRE POUR UN ID

var

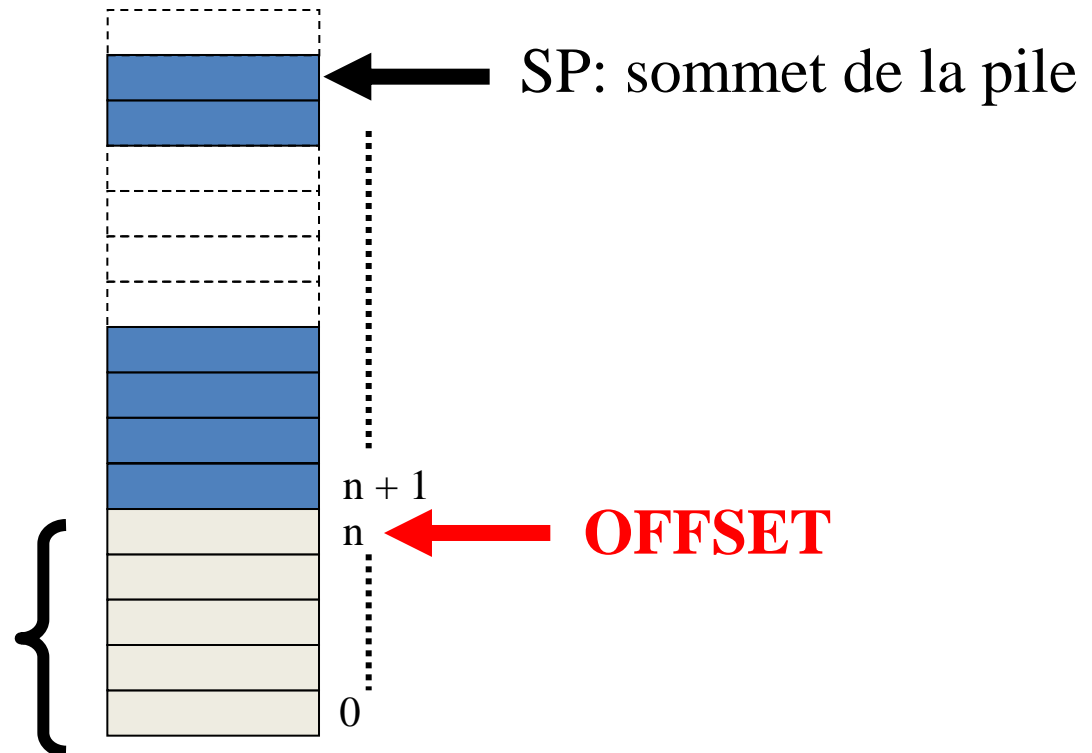
TABLESYM : tableau [TABLEINDEX] de enregistrement

NOM : ALFA ;
CLASSE : CLASSES ;
ADRESSE : ENTIER

fin ;

OFFSET : ENTIER ;

Réservation des
variables globales



$\text{VAR} ::= \text{var ID} \{ , \text{ID} \} ; \mid \varepsilon$



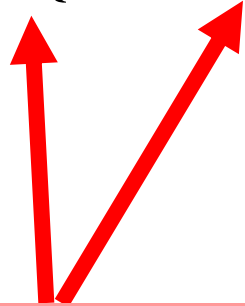
RESERVATION D'UNE PLACE MEMOIRE

Actions:

++OFFSET

stocker l'adresse réservé dans la table des symboles

$\text{VARs} ::= \text{var ID } \{ , \text{ID} \} ; \mid \varepsilon$



```
TABSYM[IND_DER_SYM_ACC].ADRESSE=++OFFSET;
```


CONST A=12; B=13;



VAR toto, tata, titi;



CONST A=12; B=13;



VAR toto, tata, titi;



⋮

⋮

Chargement de la valeur des constantes

CONSTS ::= `const ID = NUM ; { ID = NUM ; } | ε`

**RESERVATION D'UNE PLACE
MEMOIRE**

Actions:

++OFFSET
stocker l'adresse réservé dans la table
des symboles

**CHARGEMENT DE
LA VALEUR DE ID
DANS SA ZONE MEMOIRE**

ADD	additionne le sous-sommet de pile et le sommet, laisse le résultat au sommet (idem pour SUB, MUL, DIV)
EQL	laisse 1 au sommet de pile si sous-sommet = sommet, 0 sinon (idem pour NEQ, GTR, LSS, GEQ, LEQ)
PRN	imprime le sommet, dépile
INN	lit un entier, le stocke à l'adresse trouvée au sommet de pile, dépile
INT c	incrémente de la constante c le pointeur de pile (la constante c peut être négative)
LDI v	empile la valeur v
LDA a	empile l'adresse a
LDV	remplace le sommet par la valeur trouvée à l'adresse indiquée par le sommet (déréférence)
STO	stocke la valeur au sommet à l'adresse indiquée par le sous-sommet, dépile 2 fois
BRN i	branchement inconditionnel à l'instruction i
BZE i	branchement à l'instruction i si le sommet = 0, dépile
HLT	halte

jeu d'instruction du P-Code simplifié

$CONSTS ::= \text{const ID} = \text{NUM}; \{ \text{ID} = \text{NUM}; \} \mid \varepsilon$

GENERER2(LDA, TABSYM[IND_DER_SYM_ACC].ADRESSE);

GENERER1(STO);

GENERER2(LDI, VAL);

GENERATION DE DEBUT DE PROGRAMME

GENERATION DE DEBUT DE PROGRAMME

BLOCK ::= CONSTS VARS INSTS



RESERVATION DE LA
ZONE MEMOIRE

```
procedure BLOCK ;  
debut
```

```
    OFFSET := 0 ;
```

```
    CONSTS ;
```

```
    VARS ;
```

```
    PCODE[0].MNE=INT;
```

```
    PCODE[0].suite=OFFSET;
```

```
    INSTS
```

```
fin ;
```

Si 2 constantes et 3 variables, à la fin,
OFFSET=4 et la taille de la mémoire
Réservée est 5 places.



Code P-code
généré

OFFSET=4

Génération de fin de programme

Génération de fin de programme

PROGRAM ::= program ID; BLOCK .



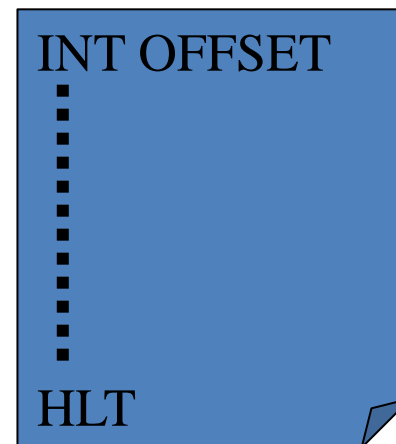
GENRATION DE L'ARRET
DU PROGRAMME

Génération de fin de programme

```
procedure PROGRAM ;  
debut
```

```
    TESTE (PROGRAM_TOKEN, PROGRAM_ERR) ;  
    TESTE_ET_ENTRE (ID_TOKEN, ID_ERR) ;  
    TEST (PT_VIRG_TOKEN, PT_VIRG_ERR) ;  
    BLOCK ;  
    GENERER1 (HLT) ;  
    TESTE_ET_ENTRE (PT_TOKEN, PT_ERR) ;
```

```
fin ;
```



Code P-code
généré

A FAIRE POUR CETTE SEANCE

- 1. LES DECLARATIONS NECESSAIRES**
- 2. MODIFIER LA FONCTION DE CODAGE LEXICALE
POUR TENIR COMPTE DES CONTROLES
SEMANTIQUES DES DECLARATIONS**
- 3. TRAITER LES REGLES: PROGRAM, CONSTS, VARS et
BLOCK**

Analyse d'une constante

Après la clause CONST

CONSTS ::= `const ID = NUM ; { ID = NUM ; } | ϵ`



RESERVATION D'UNE
PLACE MEMOIRE
++OFFSET

CHARGEMENT DE
LA VALEUR DE ID
DANS SA ZONE MEMOIRE

**CHARGEMENT DE LA VALEUR
D'UNE CONSTANTE
DANS SA ZONE MEMOIRE**

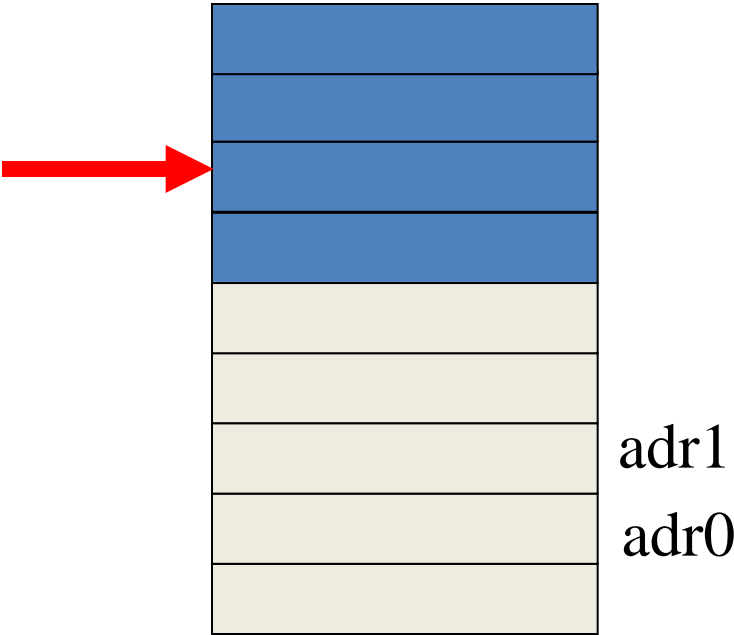
Analyse d'une CONSTANCE

Titi=45;
↑

LDA adr1
LDI 45
STO

'Toto'	'Titi'
Idf_token	Idf_token
adr0	adr1

Table des symboles



Analyse d'une CONSTANCE

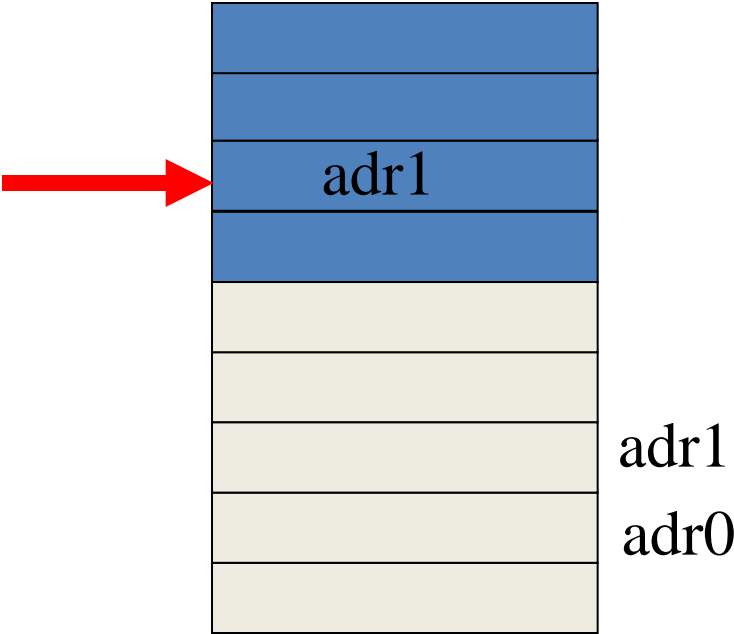
Titi=45;
↑

LDA adr1 ←
LDI 45
STO

‘Toto’
Idf_token
adr0

‘Titi’
Idf_token
adr1

Table des symboles



Analyse d'une CONSTANTE

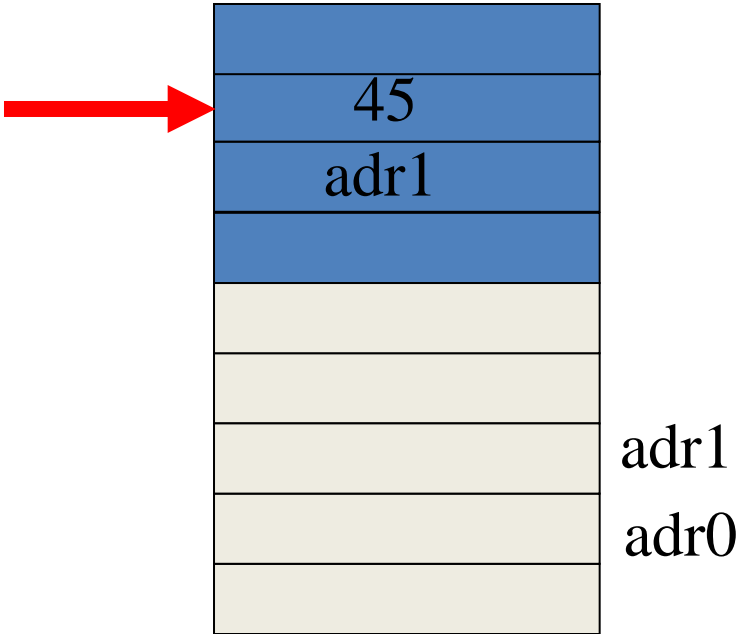
Titi=45;
↑

LDA adr1
LDI 45 ←
STO

‘Toto’
Idf_token
adr0

‘Titi’
Idf_token
adr1

Table des symboles



Analyse d'une CONSTANCE

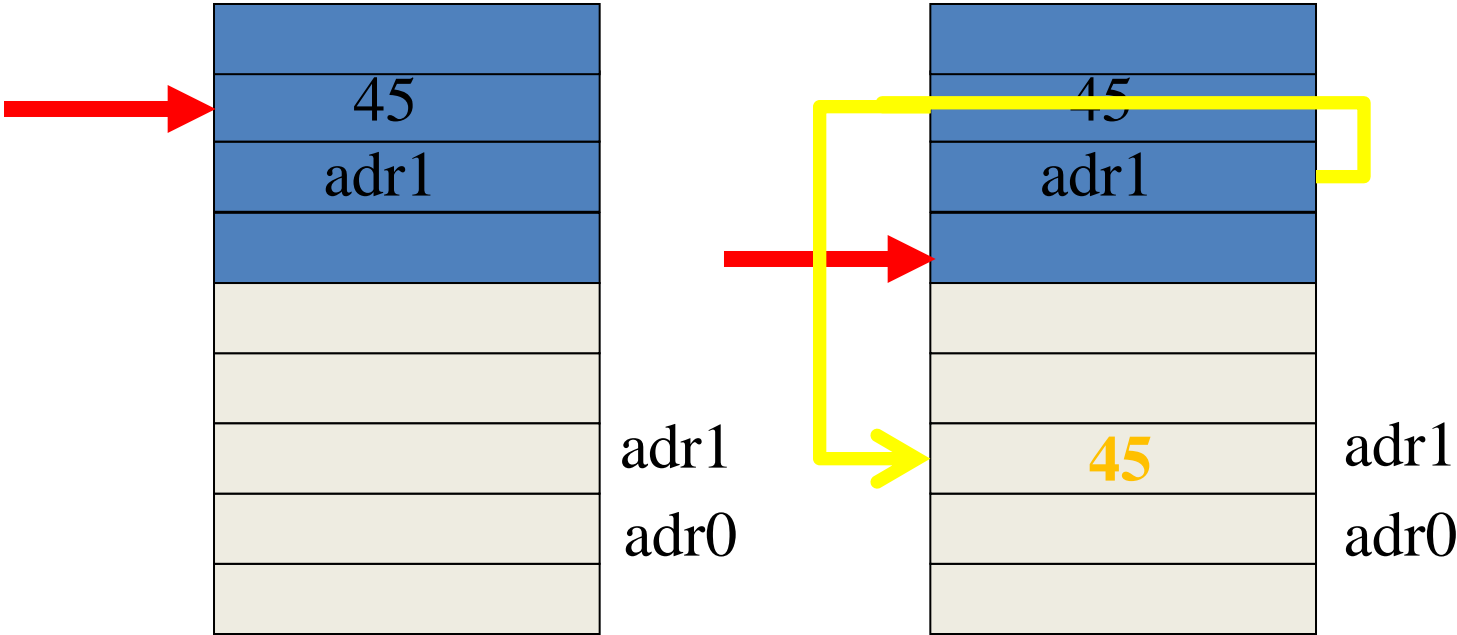
Titi=45;
↑

‘Toto’
Idf_token
adr0

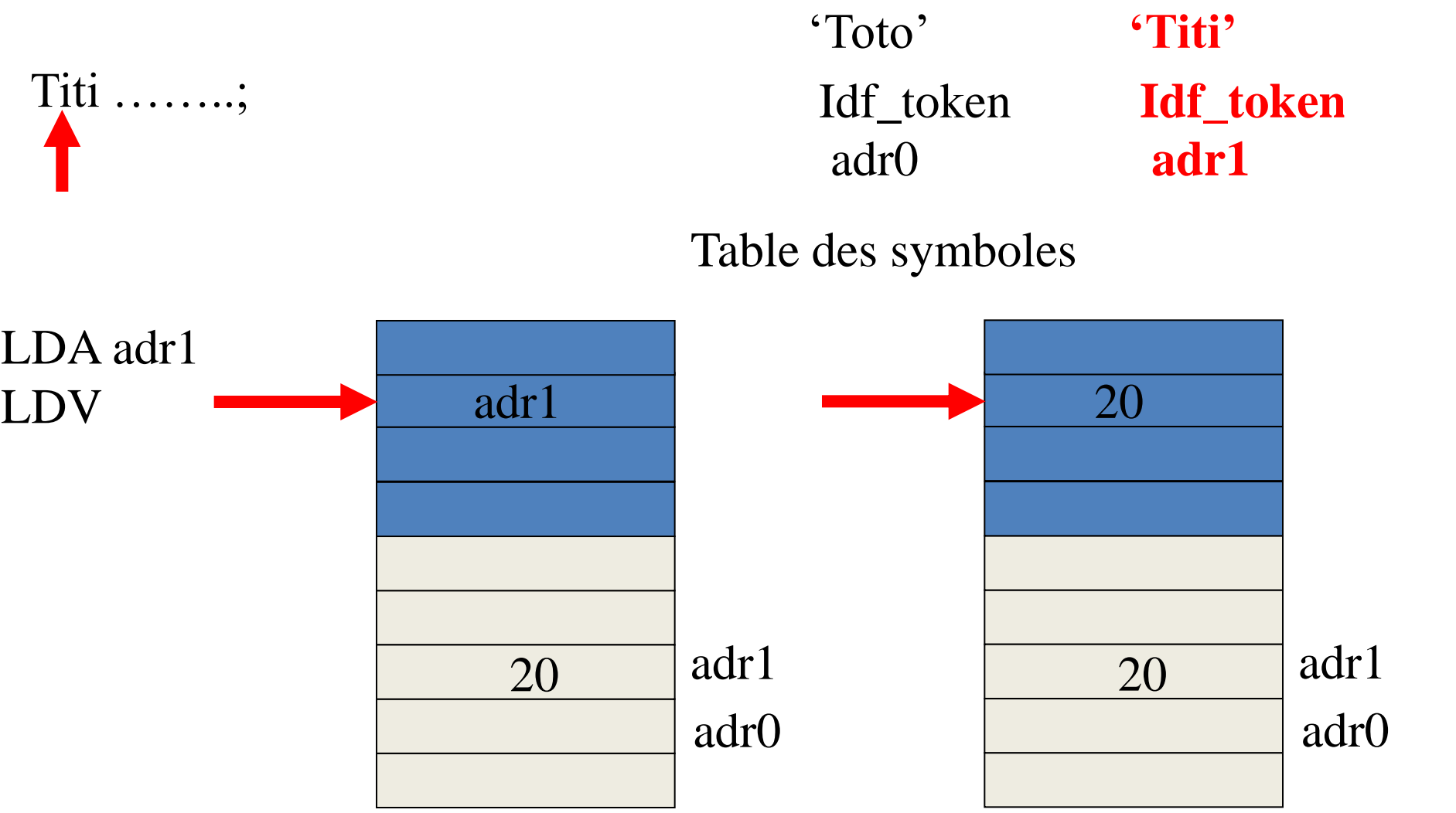
‘Titi’
Idf_token
adr1

Table des symboles

LDA adr1
LDI 45
STO ←



Analyse d'une constante Après la clause begin



Analyse d'une variable Dans la clause VAR

$\text{VAR} ::= \text{var ID} \{ , \text{ID} \} ; \mid \varepsilon$



RESERVATION D'UNE
PLACE MEMOIRE
++OFFSET

Analyse d'une variable à la déclaration

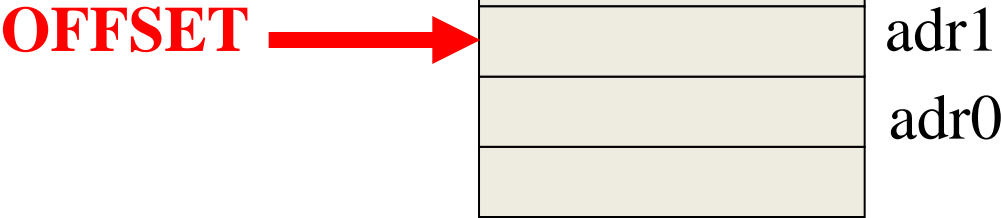
Var Titi;
↑

‘Toto’
Idf_token
adr0

‘Titi’
Idf_token
adr1

Table des symboles

OFFSET ← OFFSET+1;



Analyse d'une variable Après la clause begin

Analyse d'une variable après begin

Titi;
↑

‘Toto’

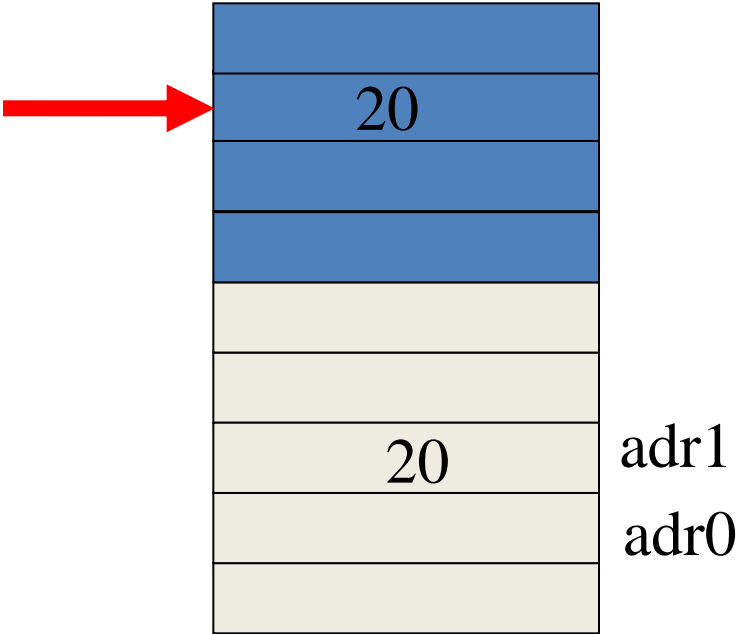
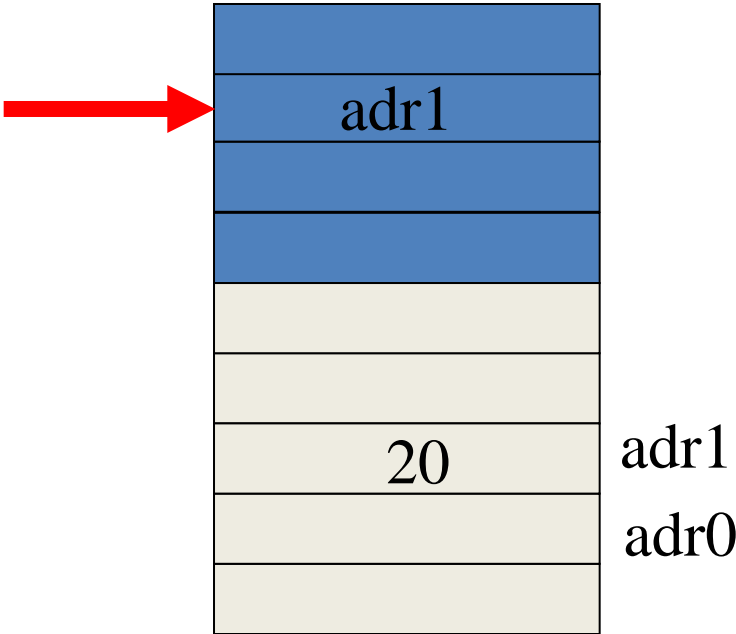
Idf_token
adr0

‘Titi’

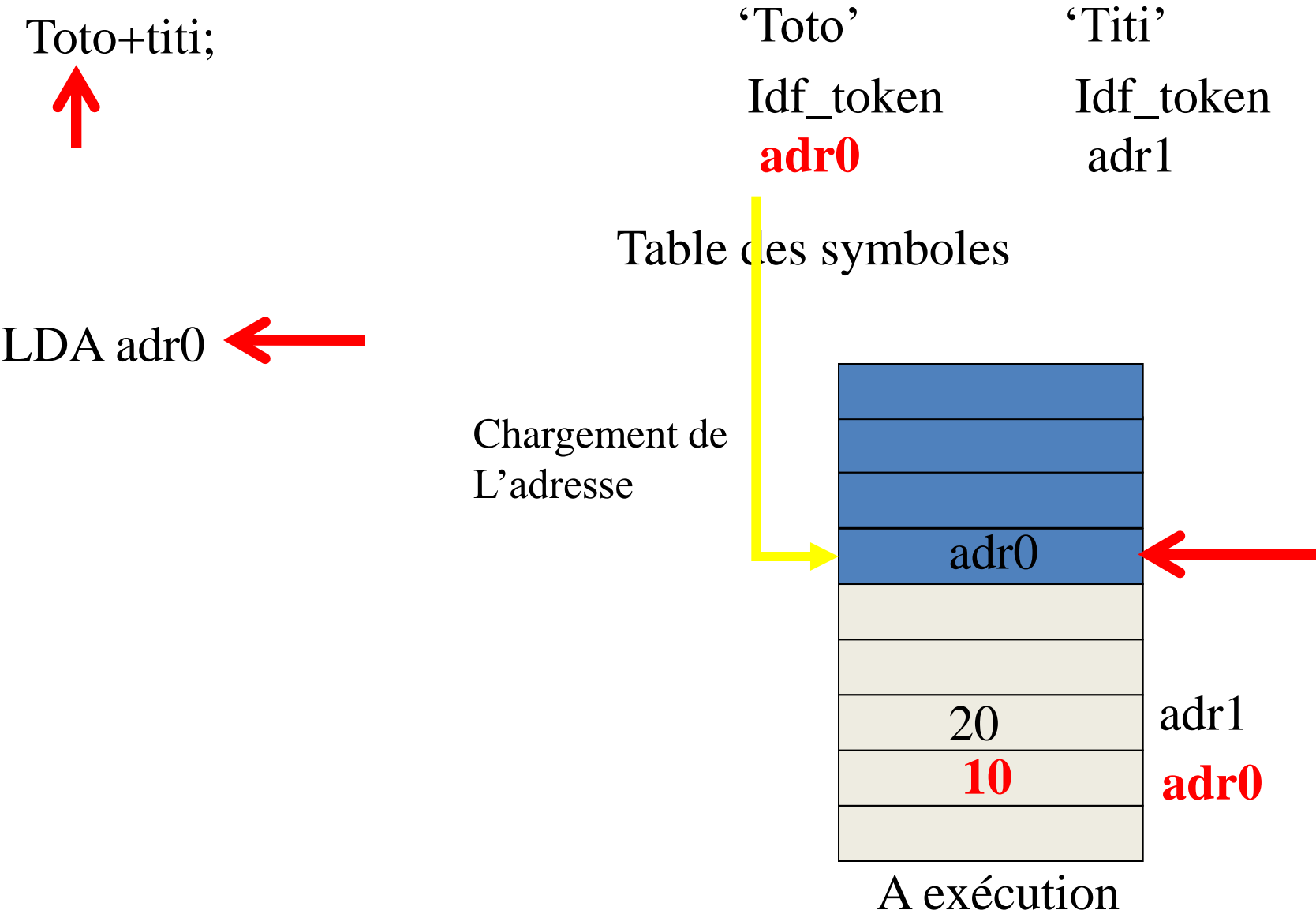
Idf_token
adr1

Table des symboles

LDA adr1
LDV



**Chargement de la valeur
D'une variable ou d'une constante
(en générale dans le programme)**



Toto+titi;



‘Toto’
Idf_token
adr0

‘Titi’
Idf_token
adr1

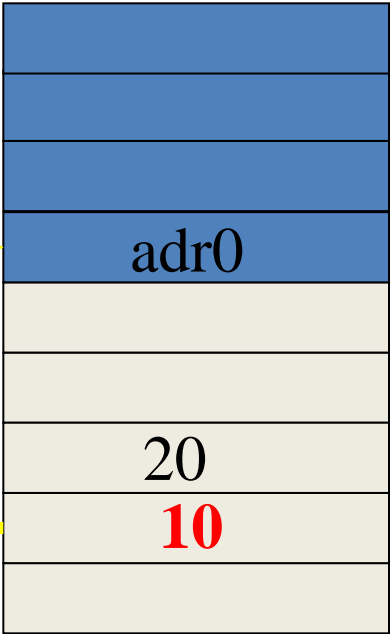
Table des symboles

LDA adr0

LDV

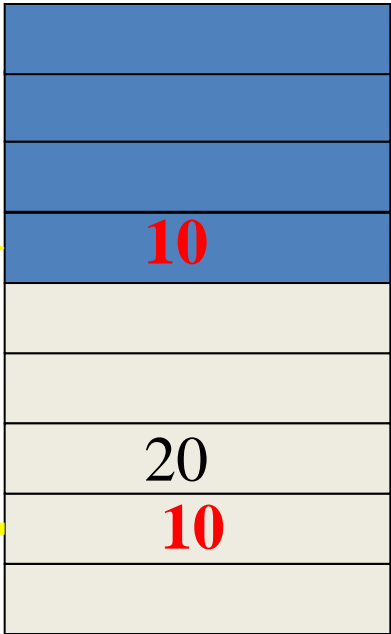


Chargement de
La valeur



adr1
adr0

A exécution



adr1
adr0

A exécution

**Chargement de la valeur
D'une variable ou d'une constante
(en général dans un programme)**

Exemple : EXPR

Toto+titi;



‘Toto’

Idf_token
adr0

‘Titi’

Idf_token
adr1

Table des symboles

LDA adr0 ←
LDV
LDA adr1
LDV
ADD

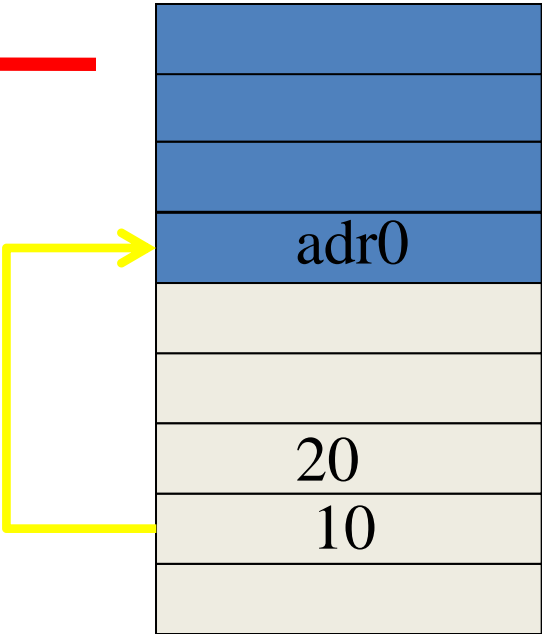
adr0	
20	adr1
10	adr0

Toto+titi;

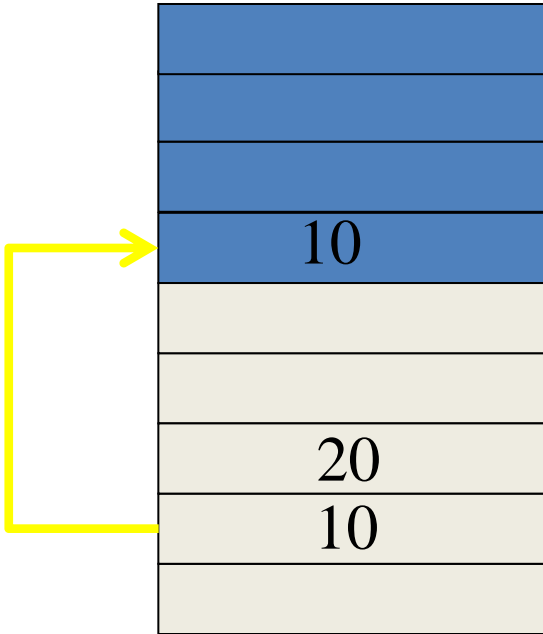
‘Toto’
Idf_token
adr0

‘Titi’
Idf_token
adr1

LDA adr0
LDV
LDA adr1
LDV
ADD



adr1
adr0



adr1
adr0

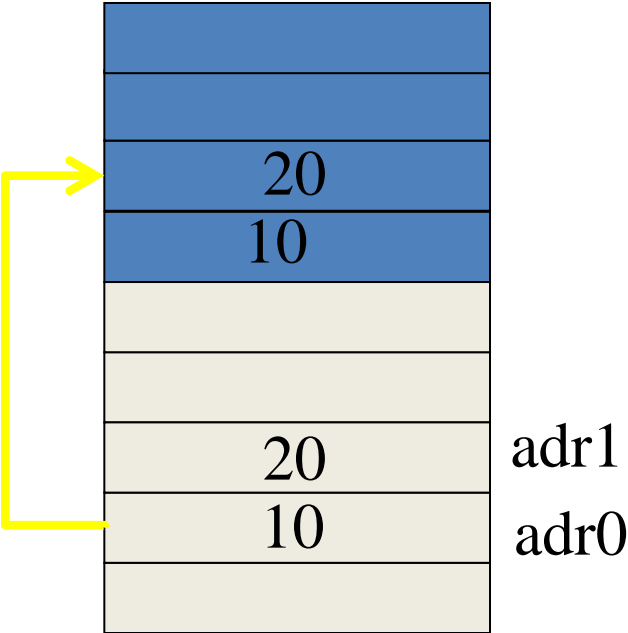
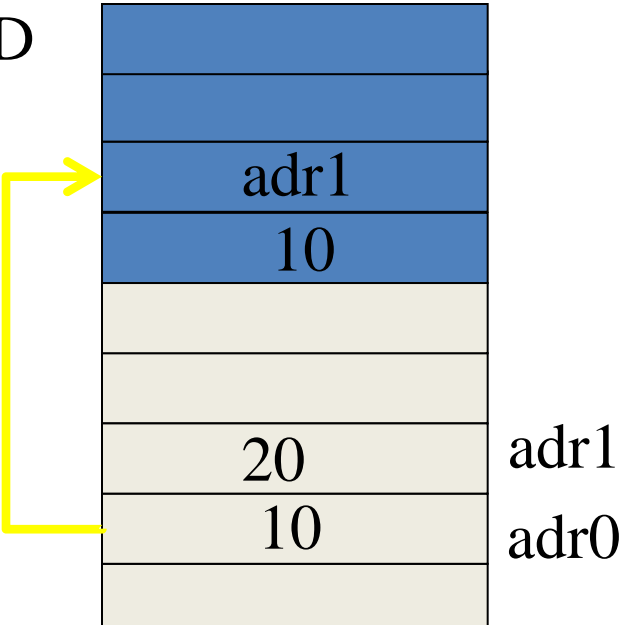
Toto+titi;

‘Toto’
Idf_token
adr0

‘Titi’
Idf_token
adr1

LDA adr0
LDV
LDA adr1
LDV
ADD

LDA adr0
LDV
LDA A adr1
LDV
ADD

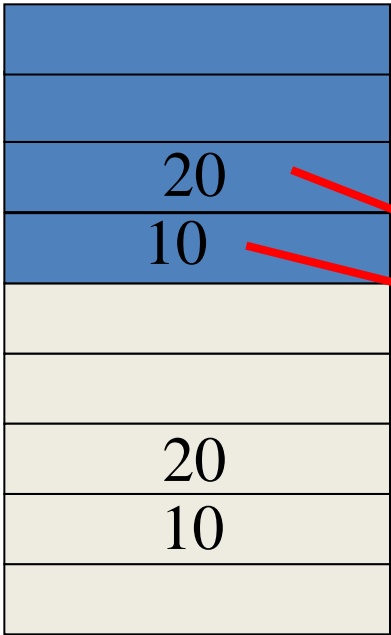


Toto+titi;

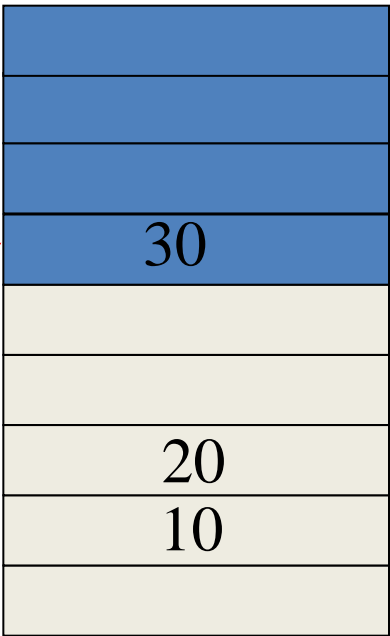
‘Toto’
Idf_token
adr0

‘Titi’
Idf_token
adr1

LDA adr0
LDV
LDA adr1
LDV
ADD ←



$10 + 20 = 30$



GENERATION DE CODE POUR LES REGLES

Génération de code pour Une affectation

AFFEC ::= ID := EXPR



CHARGER L'ADRESSE
ID AU SOMMET DE LA PILE

LE RSEULTAT DE EXPR AU
SOMMET DE LA PILE
STOCKER LE SOMMET DE
A L'ADRESSE DU ID

AFFEC ::= ID := EXPR

GENERER1(STO)

GENERER2(LDA,TABSYM[IND_DER_SYM_ACC].ADRESSE);

```
//-----  
// Procedure syntaxique de la règle:  
//      AFFEC      ::=      ID := EXPR  
//-----  
void AFFEC()  
{  
    Test_Symbole(ID_TOKEN, ID_ERR);  
    Test_Symbole(AFFECT_TOKEN, AFFECT_ERR);  
    EXPR();  
}
```

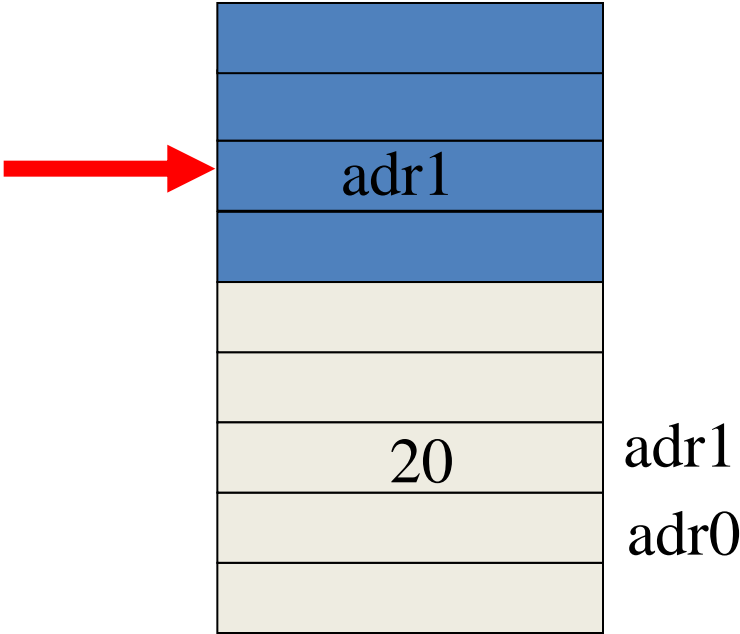
Analyse d'une variable après begin

Titi:=expression;
↑

'Toto'	'Titi'
Idf_token	Idf_token
adr0	adr1

Table des symboles

LDA adr1



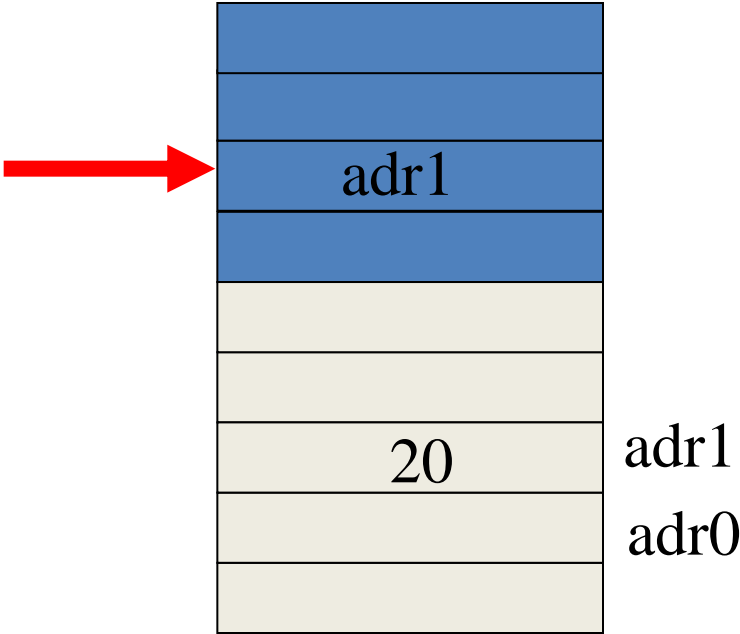
Analyse d'une variable après begin

Titi:=expression;
↑

'Toto'	'Titi'
Idf_token	Idf_token
adr0	adr1

Table des symboles

LDA adr1



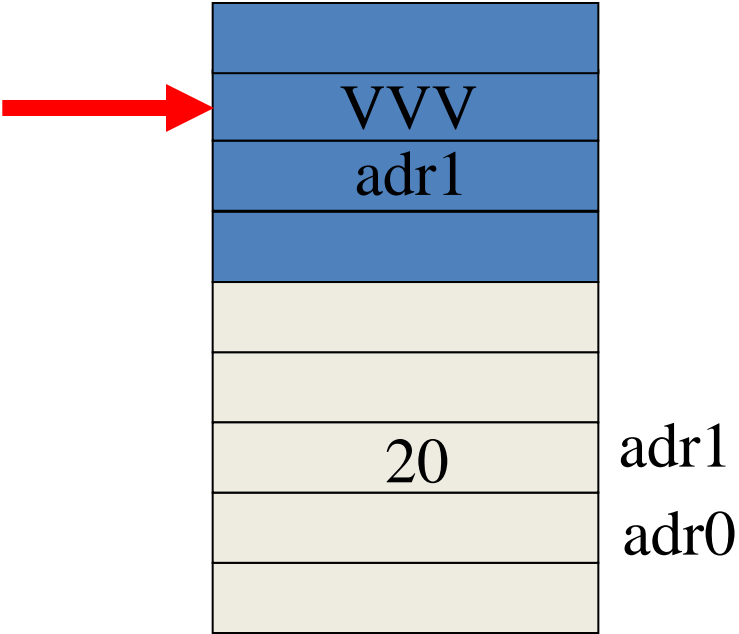
Analyse d'une variable après begin

Titi:=expression;
 ↑

'Toto'	'Titi'
Idf_token	Idf_token
adr0	adr1

Table des symboles

LDA adr1
...
...
...
<résultat de expression au sommet de la pile>



Analyse d'une variable après begin

Titi:=expression ;

‘Toto’

Idf token

‘Titi’

Idf token

Table des symboles



• • •

<résultat de expression au sommet de la pile>

STO ←



VVV

adr 1

```
//-----  
// Procedure syntaxique de la règle:  
//      AFFEC      ::=      ID := EXPR  
//-----  
void AFFEC()  
{  
    Test_Symbole(ID_TOKEN, ID_ERR);  
  
    Test_Symbole(AFFECT_TOKEN, AFFECT_ERR);  
    EXPR();  
}
```

Génération de code pour EXPR

EXPR ::= TERM { **ADDOP** TERM }



...

...

...

<résultat de TERM au sommet de la pile>

EXPR ::= TERM { **ADDOP** TERM }



OP=SYM_COUR.CL

...

...

...

<résultat de TERM au sommet de la pile>

EXPR ::= TERM { **ADDOP** TERM }



OP=SYM_COUR.CL

...

...

...

<résultat de TERM 1 au sommet de la pile>

...

...

...

<résultat de TERM 2 au sommet de la pile>

EXPR ::= TERM { **ADDOP** TERM }



OP=SYM_COUR.CL

...

...

...

<résultat de TERM 1 au sommet de la pile>

...

...

...

<résultat de TERM 2 au sommet de la pile>

GENERER1(OP)

```
//-----  
// Procedure syntaxique de la règle:  
//      EXPR ::=      TERM { ADDOP TERM }  
//      ADDOP      ::=      + | -  
//-----  
void EXPR()  
{  
    TERM();  
    while ( (Sym_Cour.cls==PLUS_TOKEN) || (Sym_Cour.cls==MOINS_TOKEN) )  
        {  
            Sym_Suiv();  
            TERM();  
        }  
}
```

Génération de code pour TERM

TERM ::= FACT {**MULOP** FACT}



...

...

...

<résultat de FACT1 au sommet de la pile>

TERM ::= FACT {**MULOP** FACT}



OP=SYM_COUR.CL

...

...

...

<résultat de FACT1 au sommet de la pile>

TERM ::= FACT {**MULOP** FACT}



OP=SYM_COUR.CL

...

...

...

<résultat de FACT 1 au sommet de la pile>

...

...

...

<résultat de FACT 2 au sommet de la pile>

TERM ::= FACT { **MULOP** FACT }



OP=SYM_COUR.CL

...
...
...
<résultat de FACT1 au sommet de la pile>
...
...
...
<résultat de FACT2 au sommet de la pile>

GENERER1(OP)

```
//-----  
// Procedure syntaxique de la règle:  
//      TERM ::=      FACT { MULOP FACT }  
//      MULOP      ::=      * | /  
//-----  
void TERM()  
{  
    FACT();  
    while ( (Sym_Cour.cls==MULTI_TOKEN) || (Sym_Cour.cls==DIV_TOKEN) )  
    {  
        Sym_Suiv();  
        FACT();  
    }  
}
```


Génération de code pour FACT

FACT ::= ID | NUM | (EXPR)



**CHARGER LA VALEUR DE ID
AU SOMMET DE LA PILE**



**CHARGER LA VALEUR DU NUM
AU SOMMET DE LA PILE**

FACT ::= ID | NUM | (EXPR)

**GENERER2(LDA, TABSYM[IND_DER_SYM_ACC].ADRESSE);
GENERER1(LDV);**

GENERER2(LDI, VAL);

```
//-----  
// Procedure syntaxique de la règle:  
//      FACT      ::=      ID | NUM | ( EXPR )  
//-----  
void FACT()  
{  
    switch (Sym_Cour.cls) {  
        case ID_TOKEN:      Sym_Suiv();  
                           break;  
        case NUM_TOKEN:     Sym_Suiv();  
                           break;  
        case PRG_TOKEN:     {Sym_Suiv();  
                           EXPR();  
                           Test_Symbole(PRD_TOKEN, PRD_ERR);  
                           }; break;  
        default: Erreur(ID_NUM_PRG_ERR);break;  
    }  
}
```

Génération de code pour Ecrire

ECRIRE ::= WRITE (EXPR { , EXPR })

ADD	additionne le sous-sommet de pile et le sommet, laisse le résultat au sommet (idem pour SUB, MUL, DIV)
EQL	laisse 1 au sommet de pile si sous-sommet = sommet, 0 sinon (idem pour NEQ, GTR, LSS, GEQ, LEQ)
PRN	imprime le sommet, dépile
INN	lit un entier, le stocke à l'adresse trouvée au sommet de pile, dépile
INT c	incrémente de la constante c le pointeur de pile (la constante c peut être négative)
LDI v	empile la valeur v
LDA a	empile l'adresse a
LDV	remplace le sommet par la valeur trouvée à l'adresse indiquée par le sommet (déréférence)
STO	stocke la valeur au sommet à l'adresse indiquée par le sous-sommet, dépile 2 fois
BRN i	branchement inconditionnel à l'instruction i
BZE i	branchement à l'instruction i si le sommet = 0, dépile
HLT	halte

jeu d'instruction du P-Code simplifié

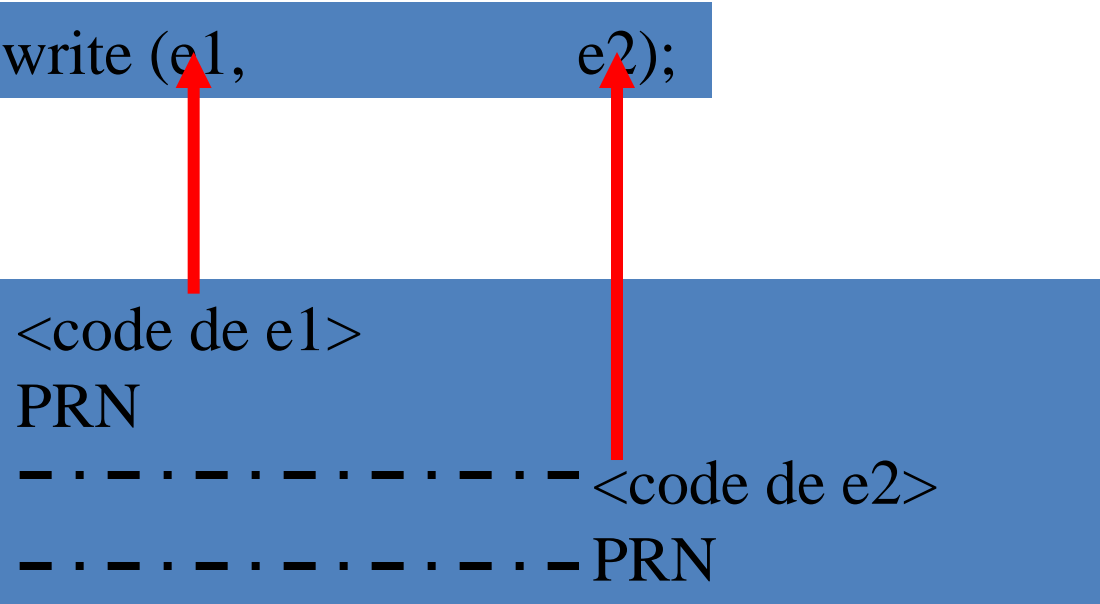
ECRIRE ::= WRITE (EXPR { , EXPR })



IMPRIMER: PRN

IMPRIMER: PRN


```
//-----  
// Procedure syntaxique de la règle:  
//          ECRIRE ::= write ( EXPR { , EXPR } )  
//-----  
void ECRIRE()  
{  
    Test_Symbole(WRITE_TOKEN, WRITE_ERR);  
    Test_Symbole(PRG_TOKEN, PRG_ERR);  
    EXPR();  
  
    while (Sym_Cour.cls==VIRG_TOKEN){  
        Sym_Suiv();  
        EXPR();  
    }  
    Test_Symbole(PRD_TOKEN, PRD_ERR);  
}
```



Exemple

Write(**Toto+titi**);

LDA adr0
LDV
LDA adr1
LDV
ADD
PRN

‘Toto’	‘Titi’
Idf_token	Idf_token
adr0	adr1

Génération de code pour LIRE

$$\text{LIRE} ::= \text{READ} (\text{ID } \{ , \text{ID } \})$$

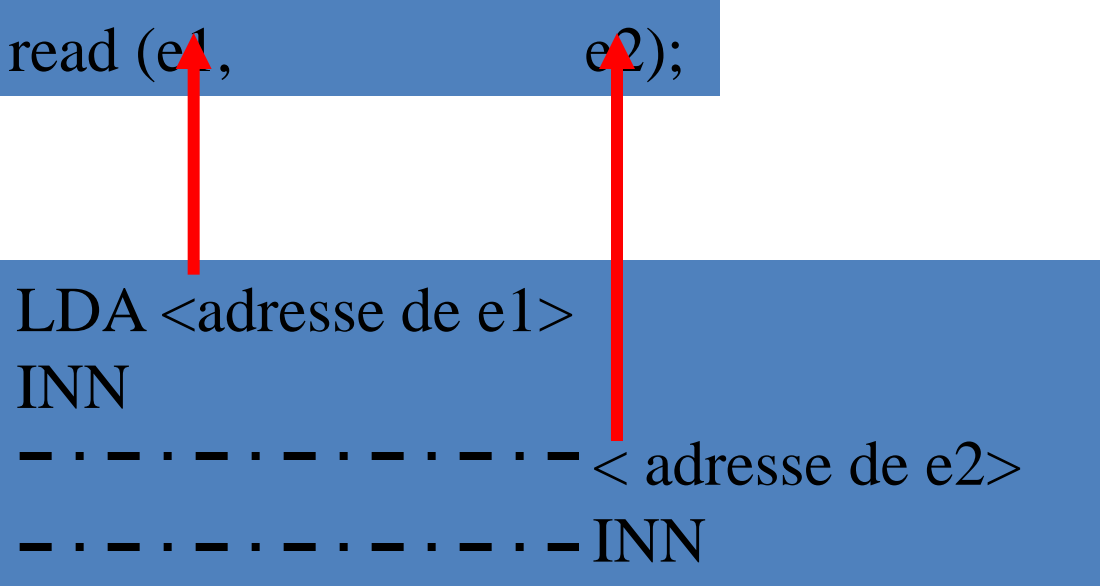
ADD	additionne le sous-sommet de pile et le sommet, laisse le résultat au sommet (idem pour SUB, MUL, DIV)
EQL	laisse 1 au sommet de pile si sous-sommet = sommet, 0 sinon (idem pour NEQ, GTR, LSS, GEQ, LEQ)
PRN	imprime le sommet, dépile
INN	lit un entier, le stocke à l'adresse trouvée au sommet de pile, dépile
INT c	incrémente de la constante c le pointeur de pile (la constante c peut être négative)
LDI v	empile la valeur v
LDA a	empile l'adresse a
LDV	remplace le sommet par la valeur trouvée à l'adresse indiquée par le sommet (déréférence)
STO	stocke la valeur au sommet à l'adresse indiquée par le sous-sommet, dépile 2 fois
BRN i	branchement inconditionnel à l'instruction i
BZE i	branchement à l'instruction i si le sommet = 0, dépile
HLT	halte

jeu d'instruction du P-Code simplifié

LIRE ::= READ (ID { , ID })



**LIRE ET STOCKER
A L'ADRESSE DE ID**



Exemple
read(Toto, titi);

LDA adr0
INN
LDA adr1
INN

‘Toto’	‘Titi’
Idf_token	Idf_token
adr0	adr1

```
//-----  
// Procedure syntaxique de la règle:  
//          LIRE ::=      read ( ID { , ID } )  
//-----  
void LIRE()  
{  
    Test_Symbole(READ_TOKEN, READ_ERR);  
    Test_Symbole(PRG_TOKEN, PRG_ERR);  
    Test_Symbole(ID_TOKEN, ID_ERR);  
    while (Sym_Cour.cls==VIRG_TOKEN){  
        Sym_Suiv();  
        Test_Symbole(ID_TOKEN, ID_ERR);  
    }  
    Test_Symbole(PRD_TOKEN, PRD_ERR);  
}
```


Génération de code pour COND

$\text{COND} ::= \text{EXPR } \textbf{RELOP} \text{ EXPR}$

MEMORISER RELOP



GENERER RELOP



$\text{RELOP} ::= = \mid <> \mid < \mid > \mid <= \mid >=$

=	EQL
<>	NEQ
>	GTR
>=	GEQ
<	LSS
<=	LEQ

Génération de code pour IF ... THEN

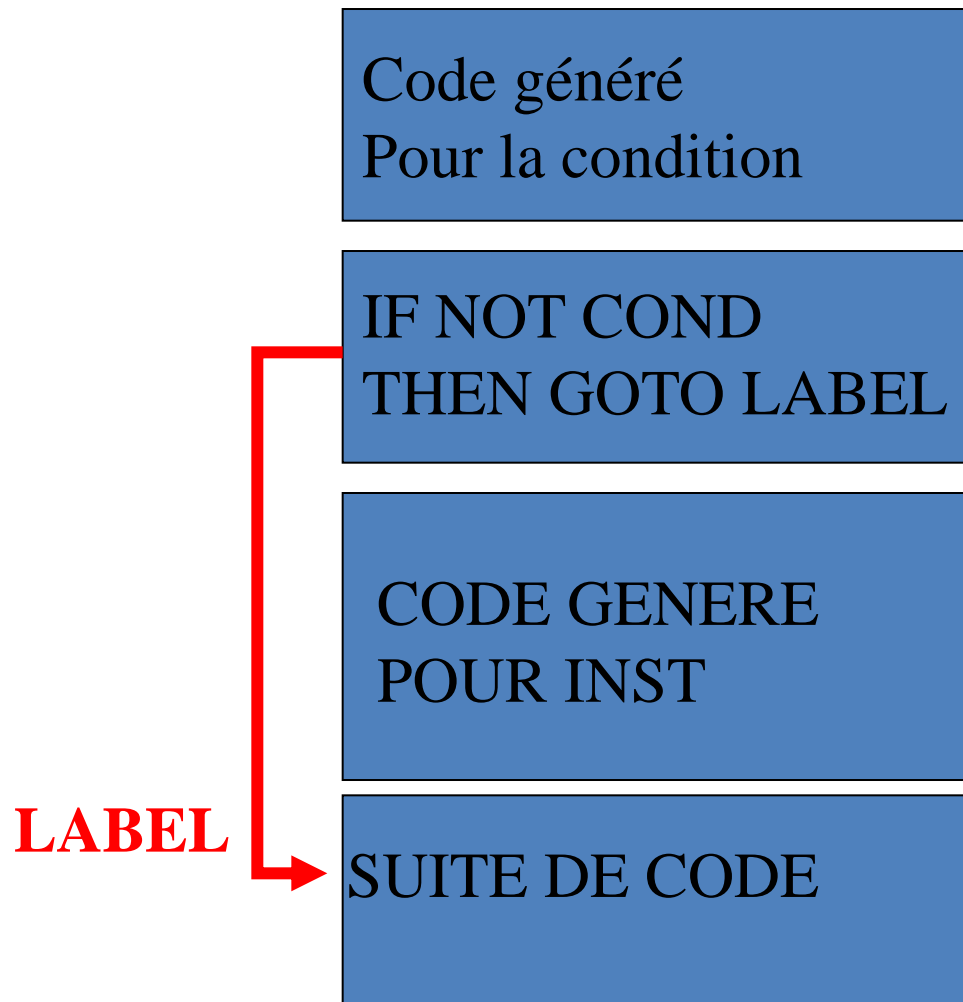
IF <COND> THEN <INST>;

Code généré
Pour la condition

Problème:
La taille de code de INST
ne pouvant pas être connu
à l'avance!!!!!!

CODE GENE
POUR INST

IF <COND> THEN <INST>;



Problème:
La taille de code de INST
ne pouvant pas être connu
à l'avance!!!!!!

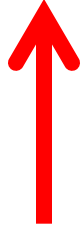
IF **<COND>** **THEN** **<INST>** ;



**BRANCHEMENT
GENERER BZE**

**REVENIR ET
COMPLETER BZE**

IF <COND> THEN <INST> ;



CODE DE
COND

IF <COND> THEN <INST> ;



CODE DE
COND

IND_BZE ← PC

BZE??

← PC

IF <COND> THEN <INST> ;



CODE DE
COND

BZE PC+1

CODE DE
INST

← PC

~~‘Toto’~~

~~‘Titi’~~

Idf_token
adr0

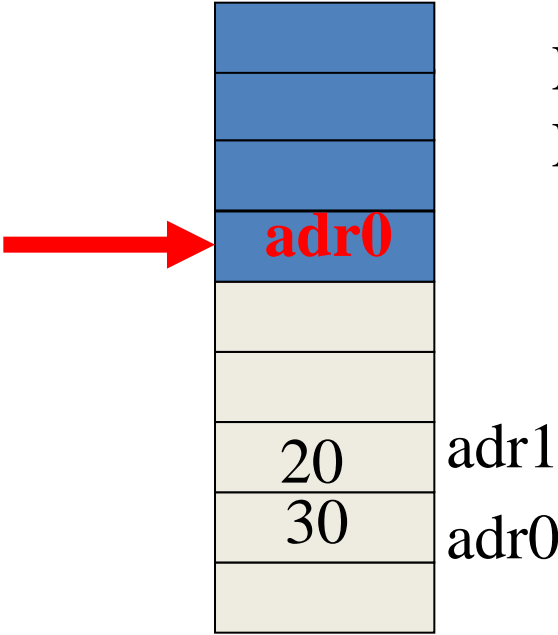
Idf_token
adr1

Table des symboles

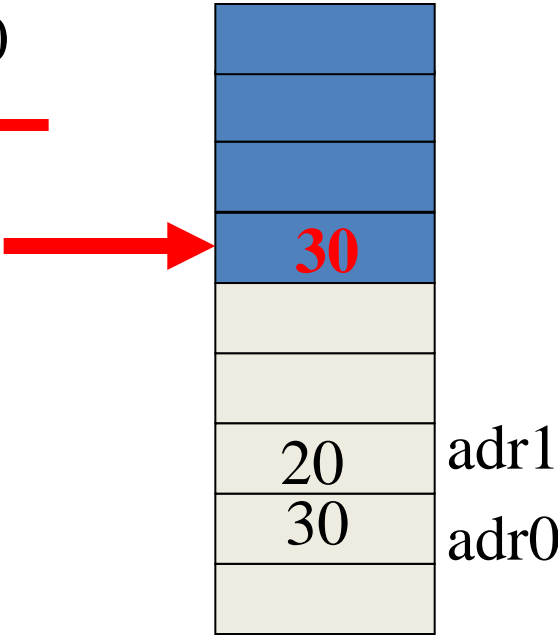
IF Toto=45 THEN titi:=90;



LDA adr0 ←
LDV



LDA adr0
LDV ←



~~‘Toto’~~

~~‘Titi’~~

Idf_token
adr0

Idf_token
adr1

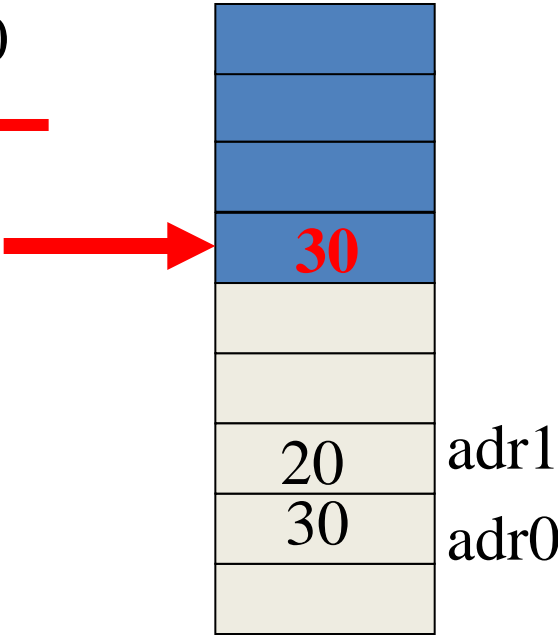
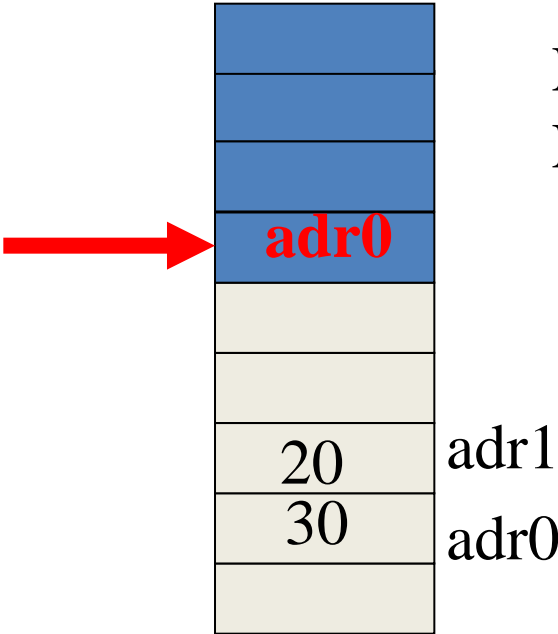
Table des symboles

IF Toto=45 THEN titi:=90;

EQ_token op

LDA adr0
LDV

LDA adr0
LDV



~~‘Toto’~~

~~‘Titi’~~

Idf_token
adr0

Idf_token
adr1

Table des symboles

EQ_token op

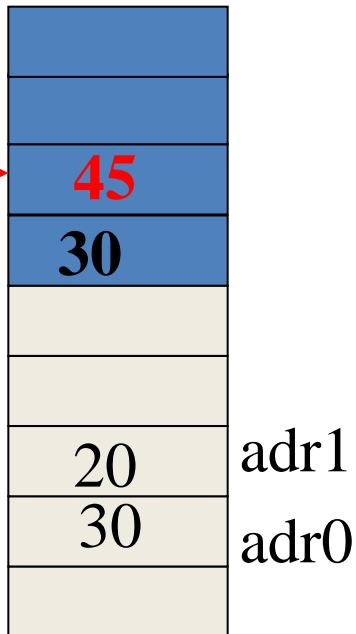
IF Toto=45 THEN titi:=90;



LDA adr0

LDV

LDI 45 ←



~~‘Toto’~~

~~‘Titi’~~

Idf_token
adr0

Idf_token
adr1

Table des symboles

EQ_token

op

IF Toto=45 THEN titi:=90;



LDA adr0
LDV
LDI 45
EQ ←



0
20
30

45 = 30 ???

adr1
adr0

~~‘Toto’~~

~~‘Titi’~~

Idf_token
adr0

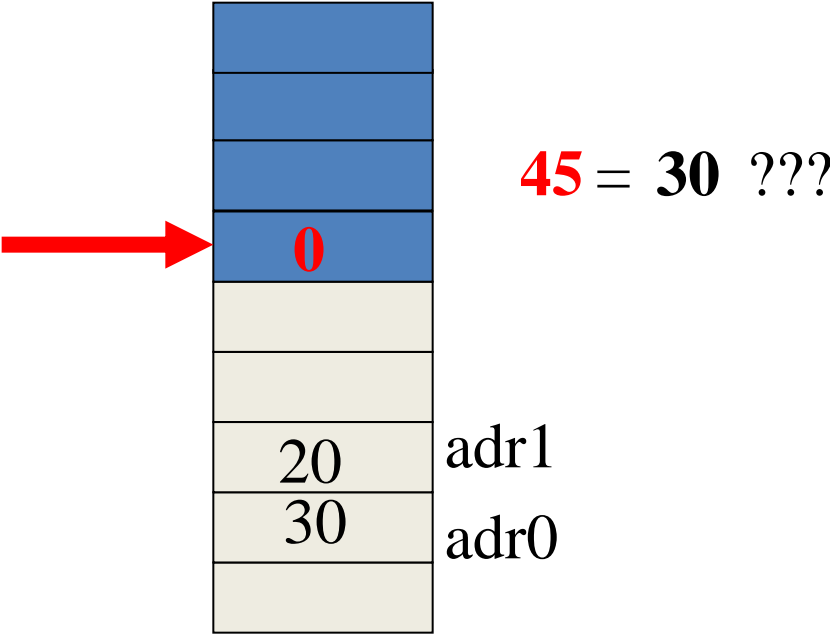
Idf_token
adr1

Table des symboles

IF Toto=45 THEN titi:=90;



LDA adr0
LDV
LDI 45
EQ ←



~~‘Toto’~~

~~‘Titi’~~

Idf_token
adr0

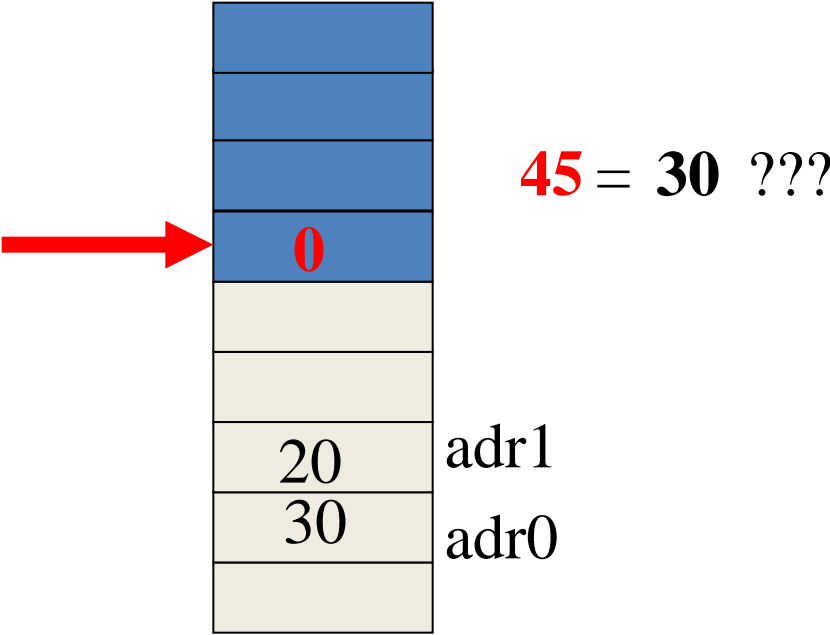
Idf_token
adr1

Table des symboles

IF Toto=45 THEN titi:=90;



LDA adr0
LDV
LDI 45
EQ ←



~~‘Toto’~~

~~‘Titi’~~

Idf_token
adr0

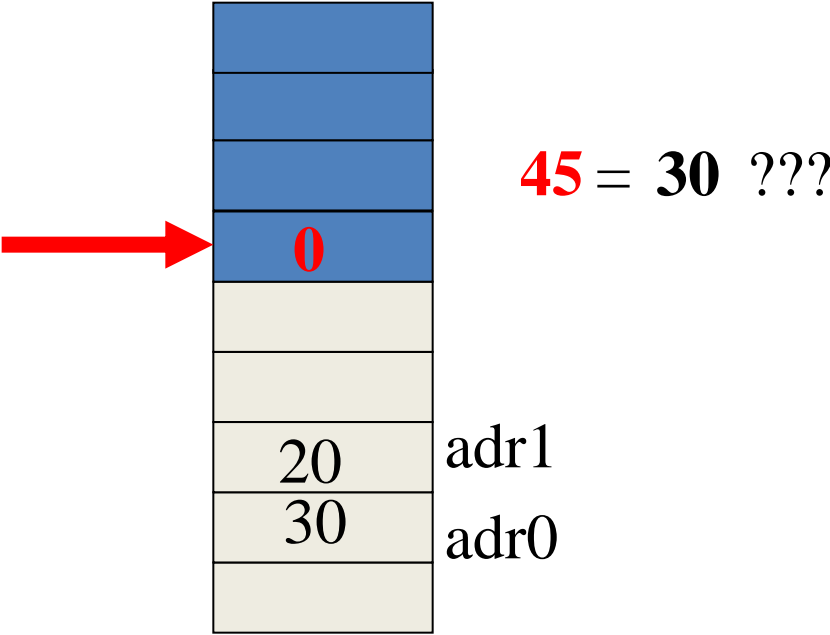
Idf_token
adr1

Table des symboles

IF Toto=45 THEN titi:=90;



LDA adr0
LDV
LDI 45
EQ
BZE????



~~‘Toto’~~

~~‘Titi’~~

Idf_token
adr0

Idf_token
adr1

Table des symboles

IF Toto=45 THEN titi:=90;



LDA adr0

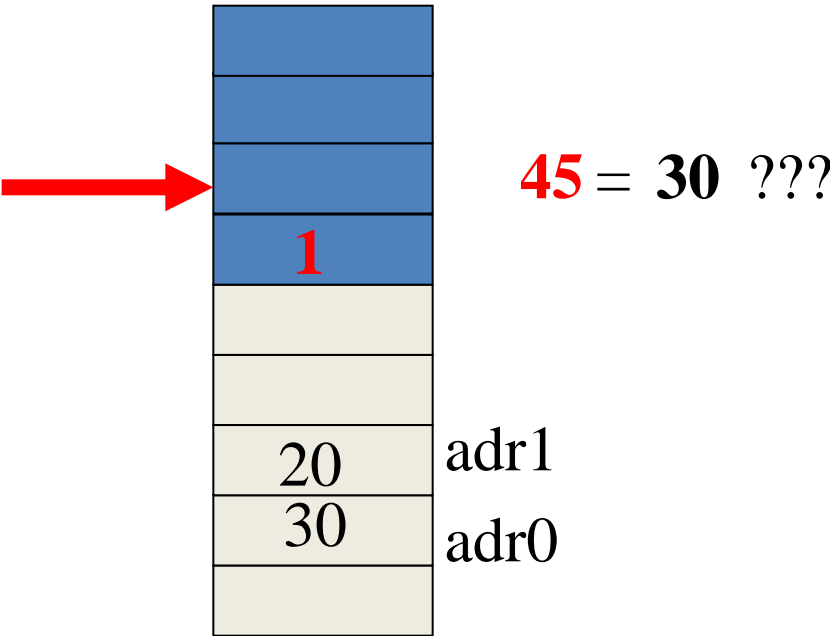
LDV

LDI 45

EQ

BZE?????

LDA adr1



Idf_token
adr0

Idf_token
adr1

Table des symboles

IF Toto=45 THEN titi:=90;



LDA adr0

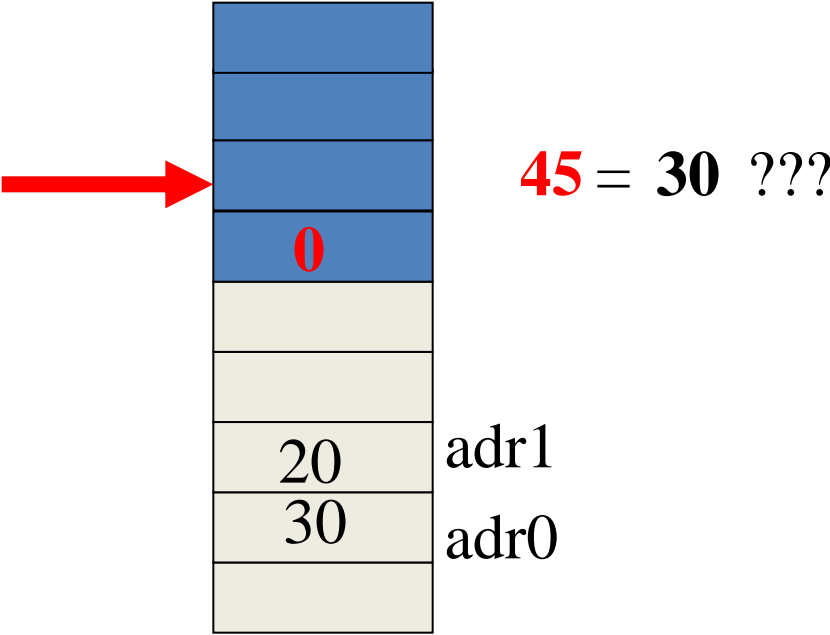
LDV

LDI 45

EQ

BZE?????

LDA adr1



~~‘Toto’~~

~~‘Titi’~~

Idf_token
adr0

Idf_token
adr1

Table des symboles

IF Toto=45 THEN titi:=90;



LDA adr0

LDV

LDI 45

EQ

BZE?????

LDA adr1

LDI 90 ←



0
20
30

45 = 30 ???

adr1

adr0

~~‘Toto’~~

~~‘Titi’~~

Idf_token
adr0

Idf_token
adr1

Table des symboles

IF Toto=45 THEN titi:=90;



LDA adr0

LDV

LDI 45

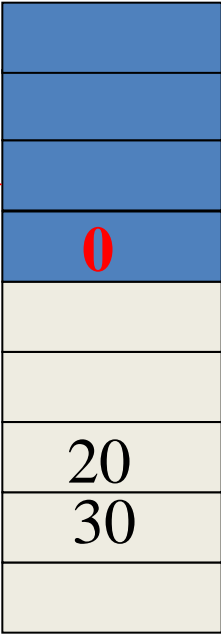
EQ

BZE?????

LDA adr1

LDI 90

STO ←



45 = 30 ???

~~Toto~~

~~Titi~~

Idf_token
adr0

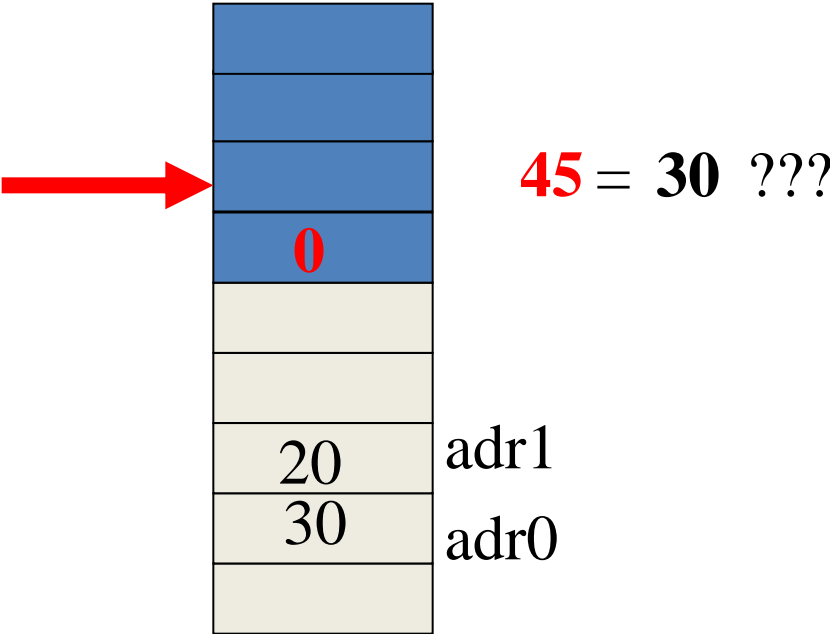
Idf_token
adr1

Table des symboles

IF Toto=45 THEN titi:=90;

LDA adr0
LDV
LDI 45
EQ
BZE?????
LDA adr1
LDI 90
STO
SUITE

LABEL



~~‘Toto’~~

~~‘Titi’~~

Idf_token
adr0

Idf_token
adr1

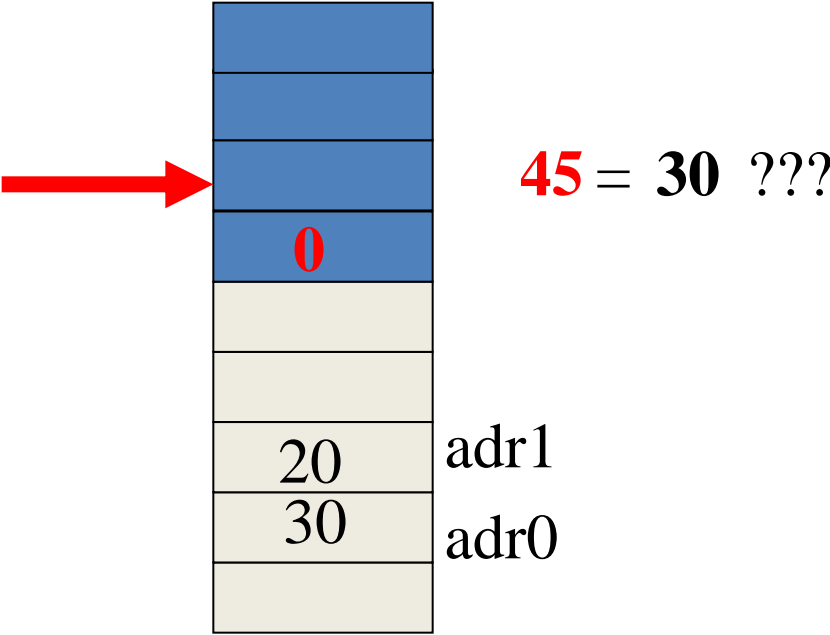
Table des symboles

IF Toto=45 THEN titi:=90;

LDA adr0
LDV
LDI 45
EQ
BZE LABEL
LDA adr1
LDI 90
STO

LABEL

SUITE

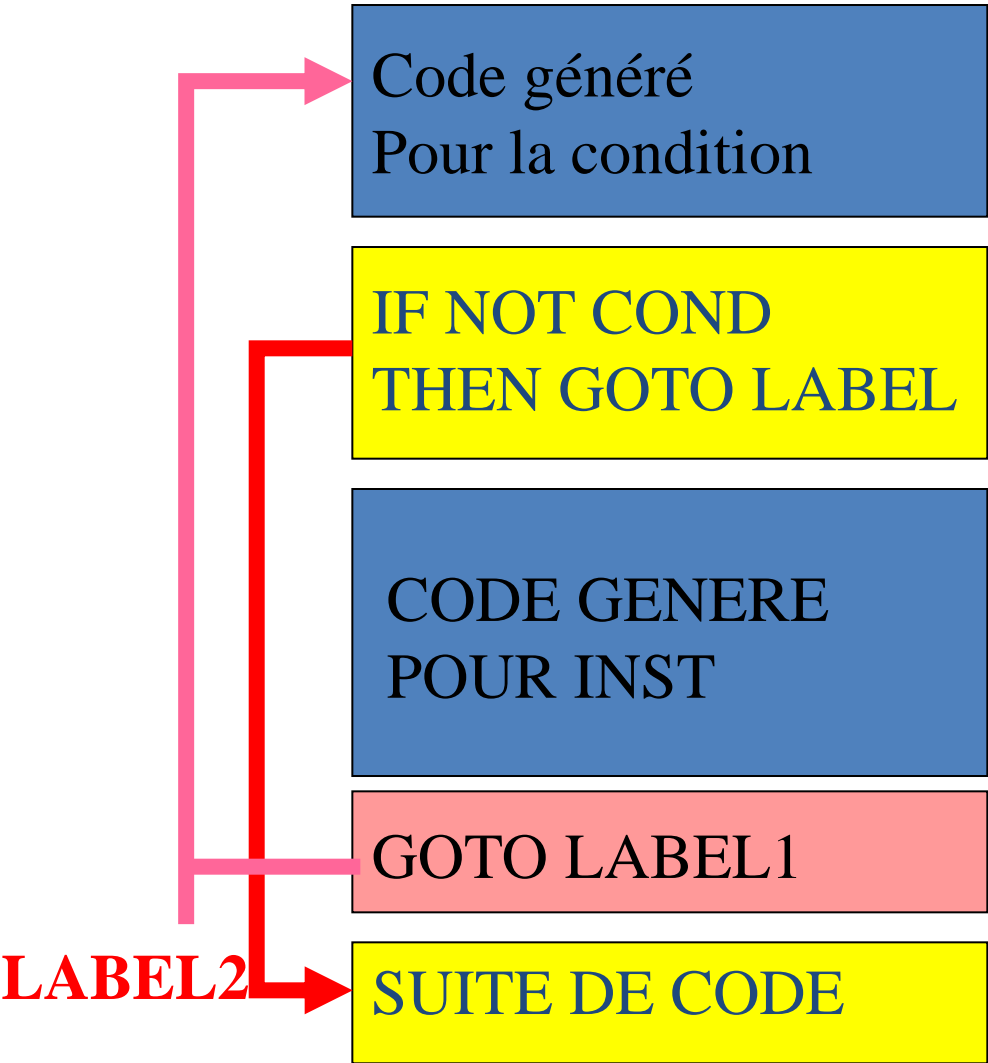


```
//-----  
// Procedure syntaxique de la règle:  
//          SI ::= if COND then INST  
//-----  
void SI()  
{  
    Test_Symbole(IF_TOKEN, IF_ERR);  
    COND();  
    Test_Symbole(THEN_TOKEN, THEN_ERR);  
    INST();  
}
```


Génération de code pour **WHILE ... DO**

WHILE <COND> DO <INST>;

LABEL1



Problème:
La taille de code de INST
ne pouvant pas être connu
à l'avance!!!!!!

WHILE <COND> DO <INST> ;

LABEL_BRN=PC+1



GENERER1(BZE);
INDICE_BZE=PC;

GENERER2(BRN, LABEL_BRN);
PCODE[INDICE_BZE].SUITE=PC+1;

~~‘Toto’~~

~~‘Titi’~~

Idf_token
adr0

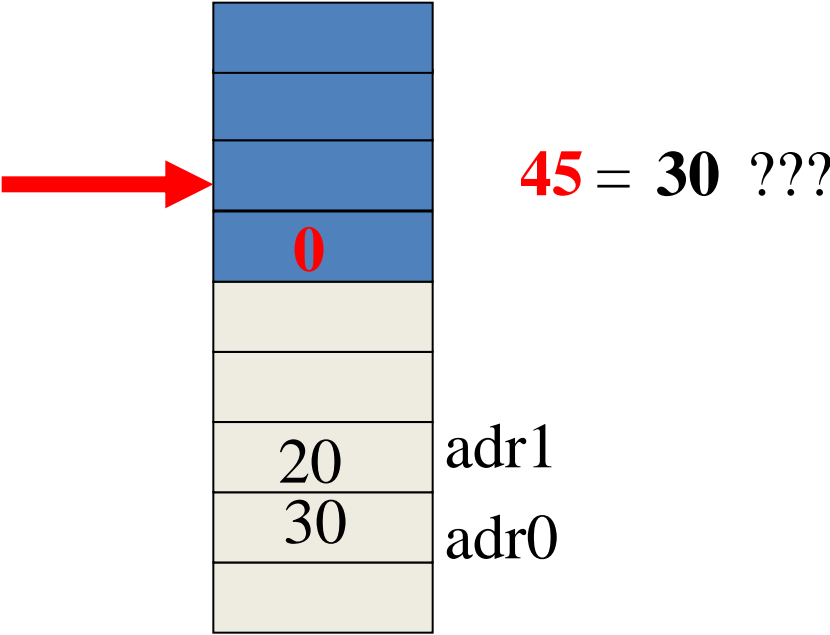
Idf_token
adr1

WHILE TOTO=45 DO titi:=90;

```
LABEL 1 LDA adr0
        LDV
        LDI 45
        EQ
        BZE LABEL 2
        LDA adr1
        LDI 90
        STO
        BRN LABEL 1
```

LABEL 2 SUITE

Table des symboles



```
//-----  
// Procedure syntaxique de la règle:  
//          TANTQUE ::=      while COND do INST  
//-----  
void TANTQUE()  
{  
    Test_Symbole(WHILE_TOKEN, WHILE_ERR);  
  
    COND();  
    GENERER1(BZE);  
  
    Test_Symbole(DO_TOKEN, DO_ERR);  
    INST();  
}
```

void CONSTS()

```
{
switch (Sym_Cour.cls) {
case CONST_TOKEN : { Sym_Suiv();
                    Test_Symbole(ID_TOKEN, ID_ERR);

                    TABSYM[IND_DER_SYM_ACC].ADRESSE=++OFFSET;
                    GENERER2(LDA, TABSYM[IND_DER_SYM_ACC].ADRESSE);
                    Test_Symbole(EGAL_TOKEN, EGAL_ERR);
                    Test_Symbole(NUM_TOKEN, NUM_ERR);
                    GENERER2(LDI, VAL);
                    GENERER1(STO);
                    Test_Symbole(PV_TOKEN, PV_ERR);

                    while (Sym_Cour.cls==ID_TOKEN){
                        Sym_Suiv();
                        TABSYM[IND_DER_SYM_ACC].ADRESSE=++OFFSET;
                        GENERER2(LDA, TABSYM[IND_DER_SYM_ACC].ADRESSE);
                        Test_Symbole(EGAL_TOKEN, EGAL_ERR);
                        Test_Symbole(NUM_TOKEN, NUM_ERR);
                        GENERER2(LDI, VAL);
                        GENERER1(STO);
                        Test_Symbole(PV_TOKEN, PV_ERR);
                    }; break;
                    }
case VAR_TOKEN:break;
case BEGIN_TOKEN: break;
default: Erreur(CONST_VAR_BEGIN_ERR);break;
}
}
```

```
void Codage_Lex(char mot[20]){
    int indice_token=-1;
    indice_token=RechercherSym(mot);

    if (indice_token!=-1)
    {
        if ((AVANT_BEGIN==1) && (indice_token>10) ) ERREUR(DD_ERR);
        else {    SYM_COUR.CLS=TABSYM[indice_token].CLS;
                 IND_DER_SYM_ACC=indice_token;
                }
    }
    else
    {
        if (AVANT_BEGIN==1) { SYM_COUR.CLS=ID_TOKEN;
                             IND_DER_SYM_ACC=index_Mots;
                             AJOUTER();
                             }
        else ERREUR(ND_ERR);
    }
}
```

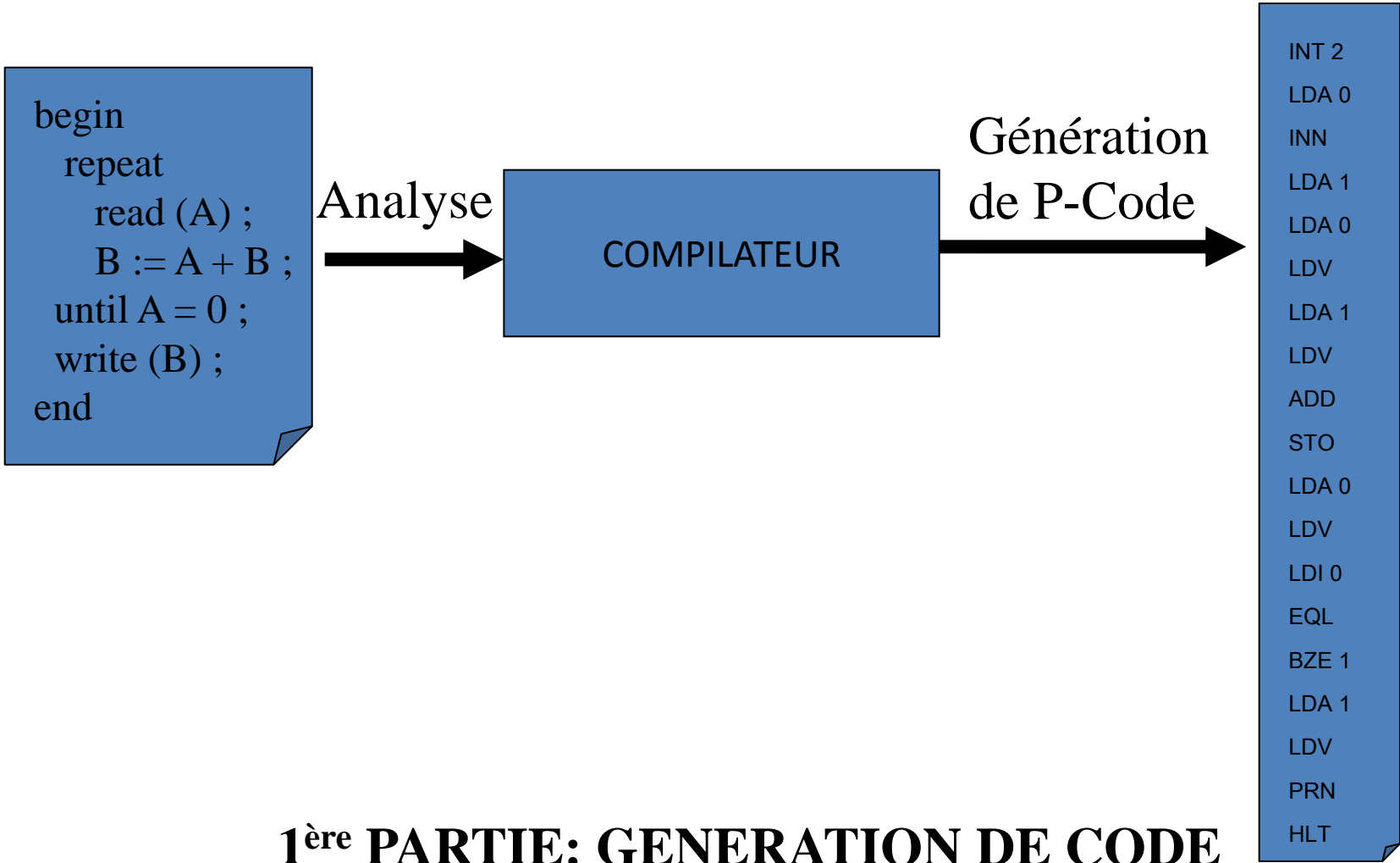
FIN
A VOUS DE FAIRE LE REST

A AJOUTER

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 | POUR | REPETER | ϵ
AFFEC ::= ID := EXPR
SI ::= **if** COND **then** INST [**else** INST | ϵ]
REPETER ::= **Repeat** INST **until** COND
POUR ::= **For** ID=NUM [**to** | **downto**] NUM **do** 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)

ECRITURE DE L'INTERPRETEUR

SAUVEGARDER LE CODE GENERE DANS UN FICHIER



```
void SaveInstToFile(INSTRUCTION INST, int i)
{
    switch( INST.MNE){
        case LDA:      fprintf(FICH_SORTIE, "%s \t %d \n", "LDA", INST.SUITE); break;
        case LDI:      fprintf(FICH_SORTIE, "%s \t %d \n", "LDI", INST.SUITE); break;
        case INT:      fprintf(FICH_SORTIE, "%s \t %d \n", "INT", INST.SUITE); break;
        case BZE:      fprintf(FICH_SORTIE, "%s \t %d \n", "BZE", INST.SUITE); break;
        case BRN:      fprintf(FICH_SORTIE, "%s \t %d \n", "BRN", INST.SUITE); break;
        case LDV:      fprintf(FICH_SORTIE, "%s \n", "LDV");          break;
        case ADD:      fprintf(FICH_SORTIE, "%s \n", "ADD");          break;
        case SUB:      fprintf(FICH_SORTIE, "%s \n", "SUB");          break;
        case MUL:      fprintf(FICH_SORTIE, "%s \n", "MUL");          break;
        case DIV:      fprintf(FICH_SORTIE, "%s \n", "DIV");          break;
        case LEQ:      fprintf(FICH_SORTIE, "%s \n", "LEQ");          break;
        case GEQ:      fprintf(FICH_SORTIE, "%s \n", "GEQ");          break;
        case NEQ:      fprintf(FICH_SORTIE, "%s \n", "NEQ");          break;
        case LSS:      fprintf(FICH_SORTIE, "%s \n", "LSS");          break;
        case GTR:      fprintf(FICH_SORTIE, "%s \n", "GTR");          break;
        case HLT:      fprintf(FICH_SORTIE, "%s \n", "HLT");          break;
        case STO:      fprintf(FICH_SORTIE, "%s \n", "STO");          break;
        case INN:      fprintf(FICH_SORTIE, "%s \n", "INN");          break;
        case PRN:      fprintf(FICH_SORTIE, "%s \n", "PRN");          break;
        default: Erreur(INST_PCODE_ERR);          break;
    }
}
```

SAUVEGARDER LE CODE GENERE DANS UN FICHIER

```
FILE *FICH_SORTIE;
```

```
FICH_SORTIE=fopen("C:\\fichierSortie.op", "w+" );
```

```
void SavePCodeToFile() {
```

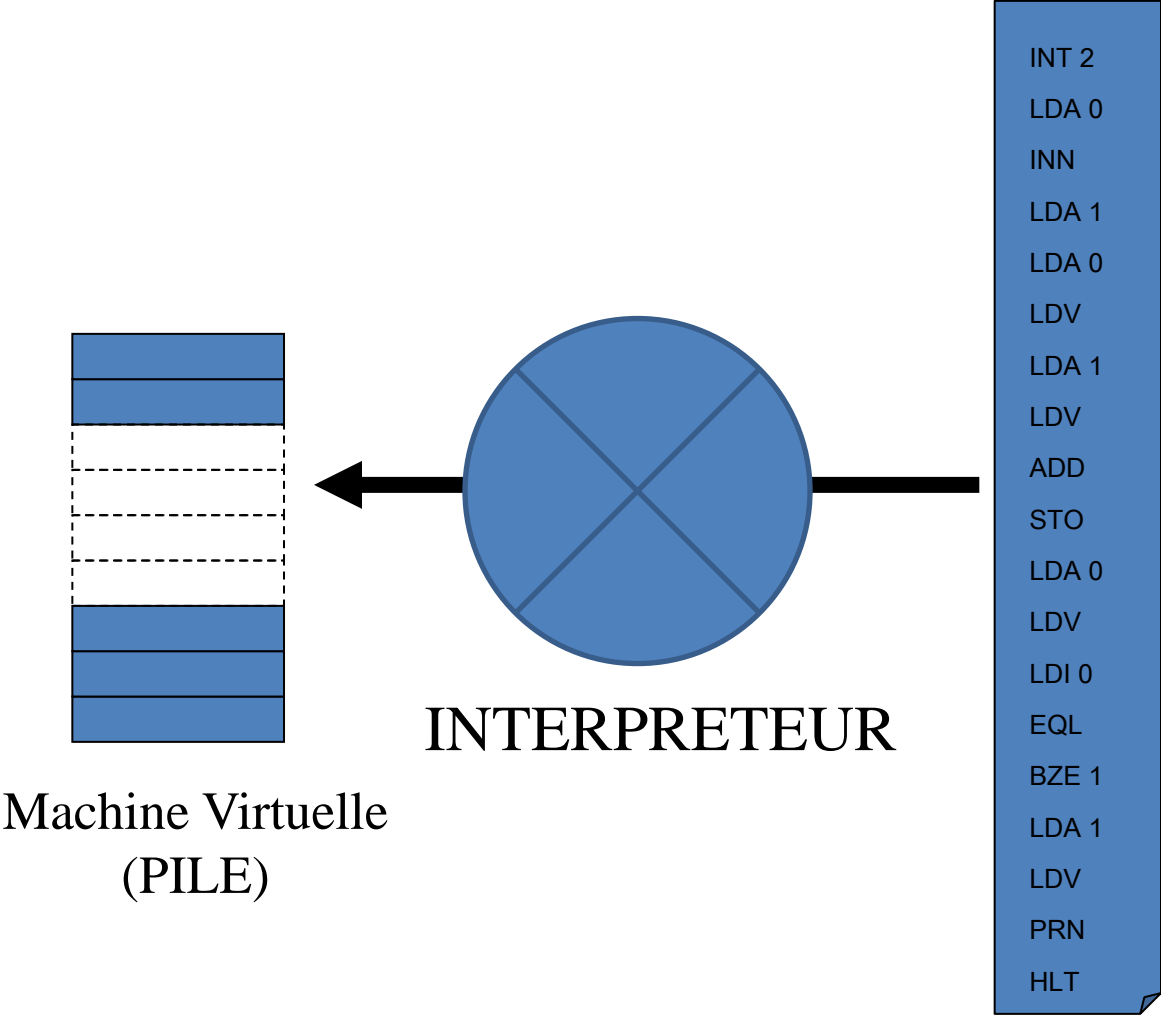
```
int i;
```

```
for (i=0; i<=PC; i++) {SaveInstToFile(PCODE[i])}  
}
```

```
Fclose(FICH_SORTIE);
```

- 1. SAUVEGARDER LE CODE GENERE DANS UN FICHIER**
- 2. DEFINIR UNE GRAMMAIRE QUI DECRIT LE CONTENU DU FICHIER**
- 3. ECRIRE UN CHARGEUR (LOADER) DU PCODE D'UN FICHIER DANS LE TABLEAU PCODE**
- 4. ECRIRE L'INTERPRETEUR**

ARCHITECTURE DE L'INTERPRETEUR



2^{ième} PARTIE: INTERPRETATION DU CODE GENERE

LA GRAMMAIRE

PCODE ::= **INT** NUM {INST_PCODE} **HLT**
INST_PCODE ::= **ADD** | **SUB**|**EQL**|...| [**LDA** | **BZE**|**BRN**|**LDI**] NUM
NUM ::= CHIFFRE {CHIFFRE}
CHIFFRE ::= **1**|..**9**

- INT 2
- LDA 0
- INN
- LDA 1
- LDA 0
- LDV
- LDA 1
- LDV
- ADD
- STO
- LDA 0
- LDV
- LDI 0
- EQL
- BZE 1
- LDA 1
- LDV
- PRN
- HLT

LES DECLARATIONS

Les structures de données nécessaires lors de l'écriture d'un interprète simplifié pour le P-Code sont :

un tableau MEM représentant la pile de la machine et un pointeur de pile associé

```
var
    MEM : TABLEAU [0 .. TAILLEMEM] DE ENTIER ;
    SP : ENTIER ;
```

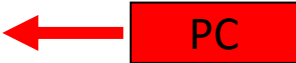
```
Type CLASS_LEX = (ADD,SUB,MUL,DIV,EQL,NEQ,GTR,
                  LSS,GEQ,LEQ, PRN,INN,INT,LDI,LDA,LDV,
                  STO,BRN,BZE,HLT, NUM) ;
```

```
INSTRUCTION =    enregistrement
                  MNE : CLASS_LEX ;
                  SUITE : entier
                fin
VAR PCODE : tableau [0 .. TAILLECODE] de INSTRUCTION ;
    PC : entier ; OFFSET=SP=-1; PC=-1;
```

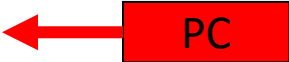
Procédures de chargement de P-Code

```
procedure ECRIRE1 (M:MNEMONIQUES) ;  
debut  
    si PC = TAILLECODE  
        alors ERREUR ;  
    PC := PC + 1 ;  
    PCODE [PC]. MNE := M  
fin ;
```

INT	2
LDA	0
INN	
LDA	1
LDA	0
LDV	

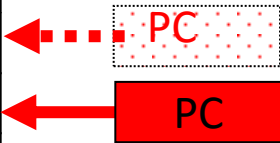


M



```
procedure ECRIRE2 (M:MNEMONIQUES ; A:entier) ;  
debut  
    si PC = TAILLECODE  
        alors ERREUR ;  
    PC := PC + 1 ;  
    PCODE [PC].MNE := M ;  
    PCODE [PC].SUITE := A  
fin;
```

INT	2
LDA	0
INN	
LDA	1
LDA	0
LDV	



INTERPRETATION DES INSTRUCTIONS MNEMONIQUES

ADD	additionne le sous-sommet de pile et le sommet, laisse le résultat au sommet (idem pour SUB, MUL, DIV)
EQL	laisse 1 au sommet de pile si sous-sommet = sommet, 0 sinon (idem pour NEQ, GTR, LSS, GEQ, LEQ)
PRN	imprime le sommet, dépile
INN	lit un entier, le stocke à l'adresse trouvée au sommet de pile, dépile
INT c	incrémente de la constante c le pointeur de pile (la constante c peut être négative)
LDI v	empile la valeur v
LDA a	empile l'adresse a
LDV	remplace le sommet par la valeur trouvée à l'adresse indiquée par le sommet (déréférence)
STO	stocke la valeur au sommet à l'adresse indiquée par le sous-sommet, dépile 2 fois
BRN i	branchement inconditionnel à l'instruction i
BZE i	branchement à l'instruction i si le sommet = 0, dépile
HLT	halte

jeu d'instruction du P-Code simplifié

```
void INTER_INST(INSTRUCTION INST){
    int val1, adr, val2;

    siwtch(INST.MNE){
        case INT: OFFSET=SP=INST.SUITE;                PC++;break;
        case LDI: MEM[++SP]=INST.SUITE;                PC++;break;
        case LDA: MEM[++SP]=INST.SUITE;                PC++; break;
        case STO: val1=MEM[SP--]; adr=MEM[SP--];MEM[adr]=val1;
        PC++;break;
        case LDV: adr=MEM[SP--]; MEM[++SP]=MEM[adr];
        PC++;break;
        case EQL: val1=MEM[SP--];val2=MEM[SP--];
                    MEM[++SP]=(val1==val2);                PC++;break;
        case LEQ: val2=MEM[SP--];val1=MEM[SP--];
                    MEM[++SP]=(val1<=val2);                PC++;break;
        case BZE: if (MEM[SP--]==0) PC=INST.SUITE;
                    else PC++;
                    break;
        case BRN: PC=INST.SUITE;
                    break;
        .....
        .....
    }
}
```

INTERPRETATION DES DE TOUT LE PCODE

```
void INTER_PCODE(){  
    PC=0;  
    while (PCODE[PC].MNE!=HLT)  
        INTER_INST(PCODE[PC]);  
}
```



FIN