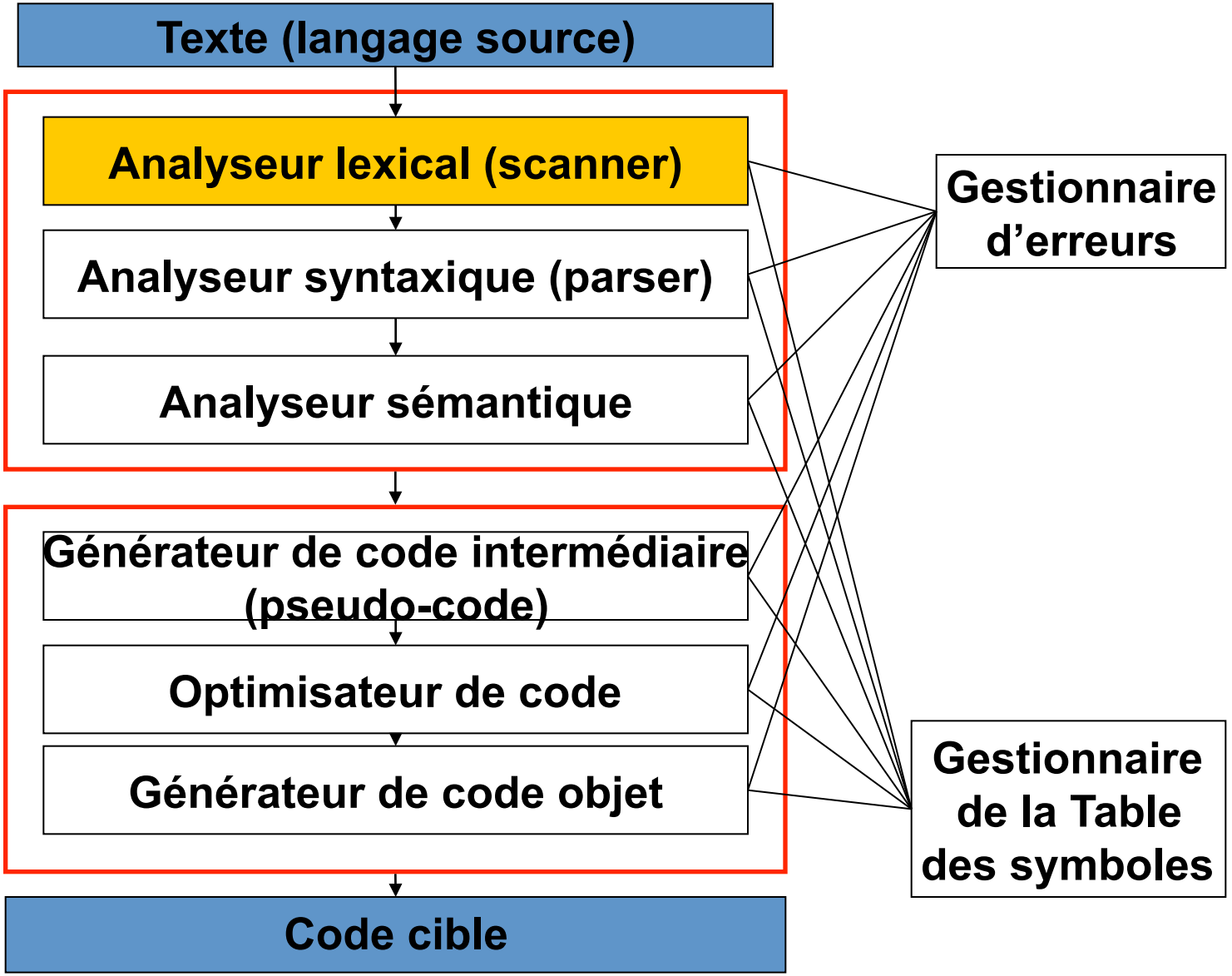
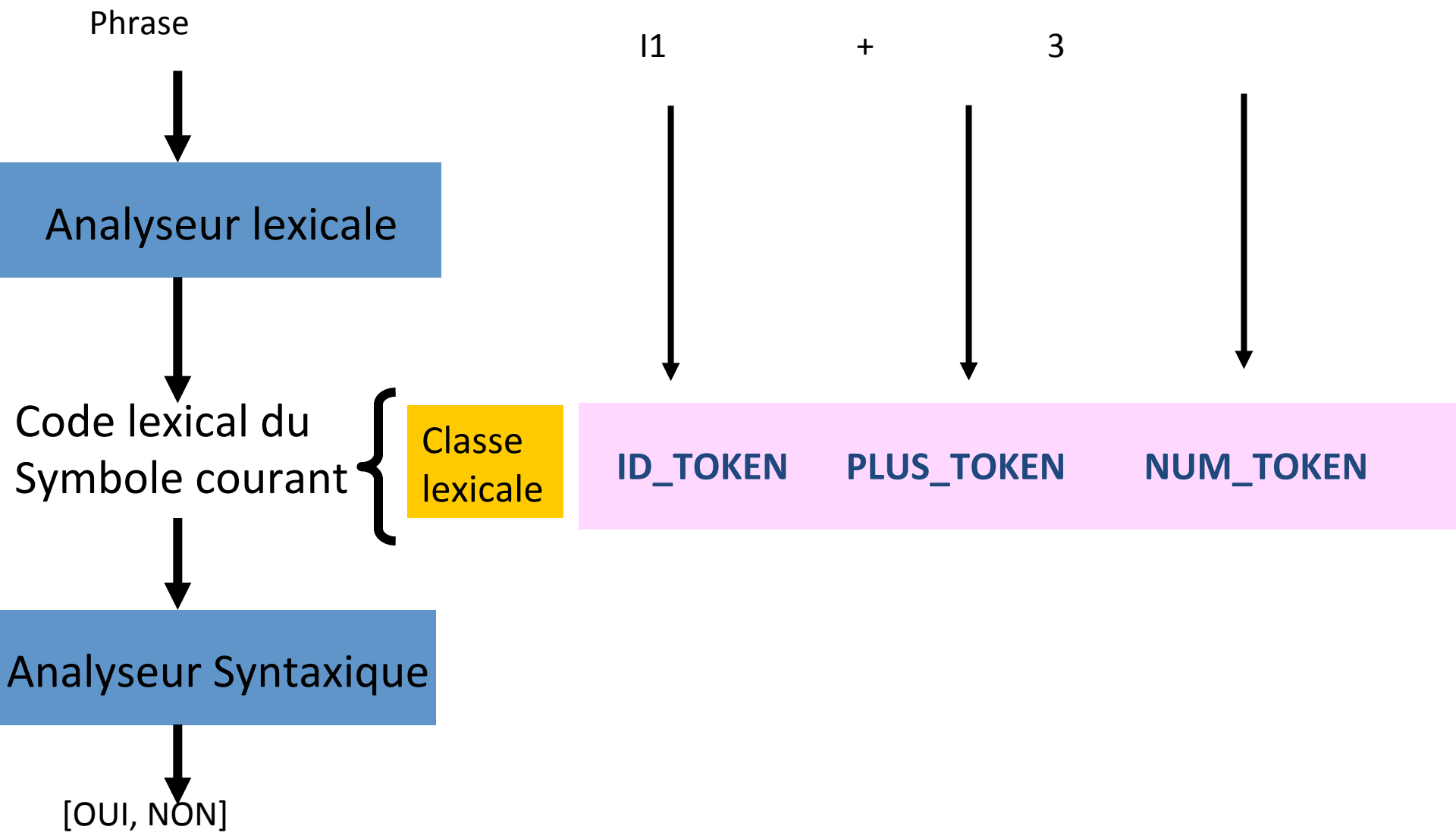
ECRITURE D'UN MINI COMPILATEUR

ANALYSEUR SYNTAXIQUE PRINCIPE



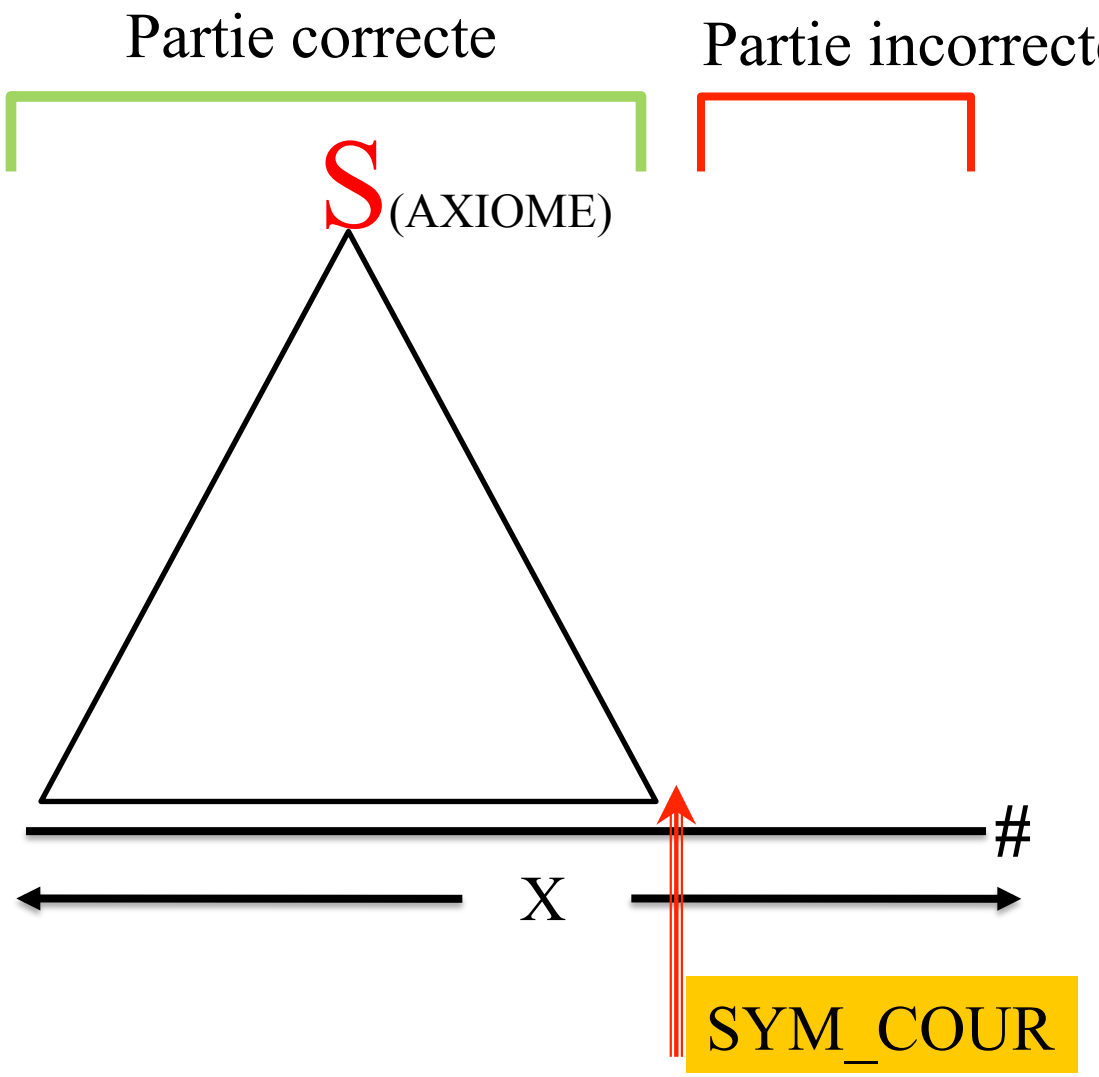
Architecture générale d'un compilateur


Phrase
Analyseur lexicale
Code lexical du
Symbole courant
Classe
lexicale
Analyseur Syntaxique
[OUI, NON]
I1
+
3
ID_TOKEN
PLUS_TOKEN
NUM_TOKEN

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:

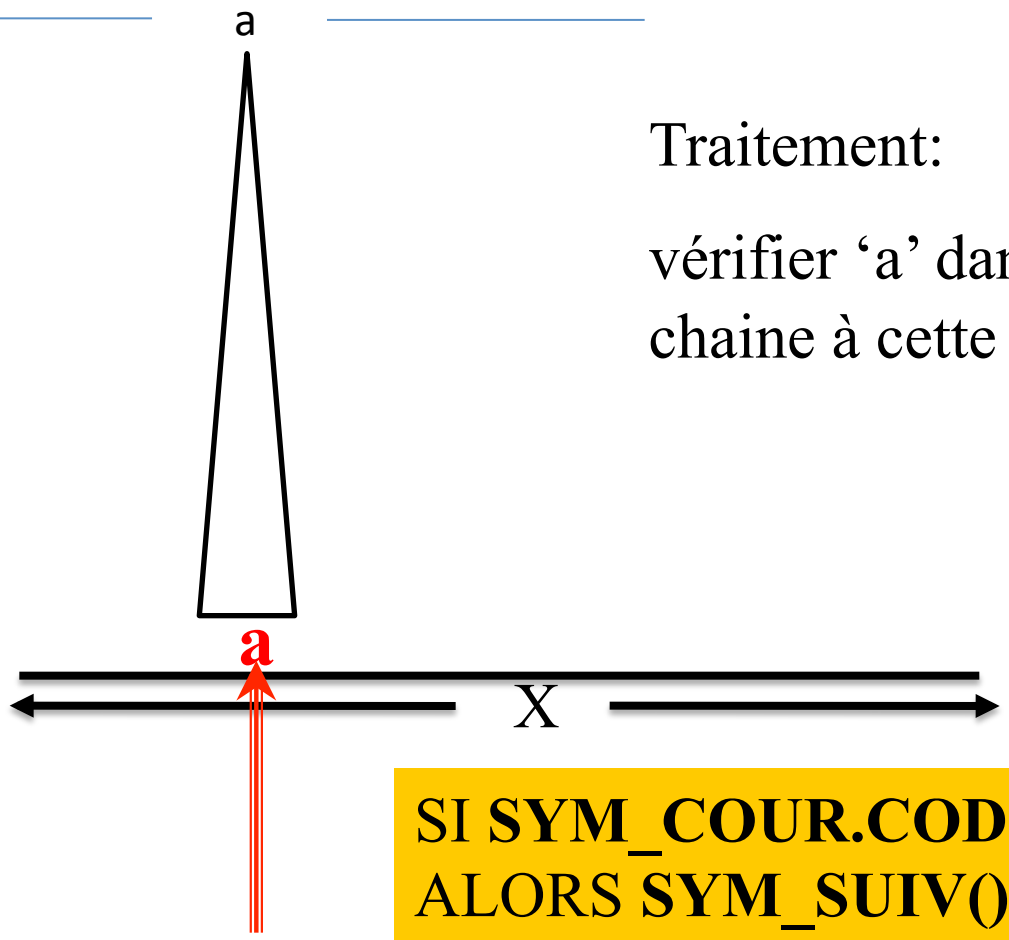
```
Procedure A;
Debut
        T(α);
Fin;
```

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

Quelque soit la règle A→ α, α contient l'une des formes suivantes

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

LE CAS D'UN TERMINAL



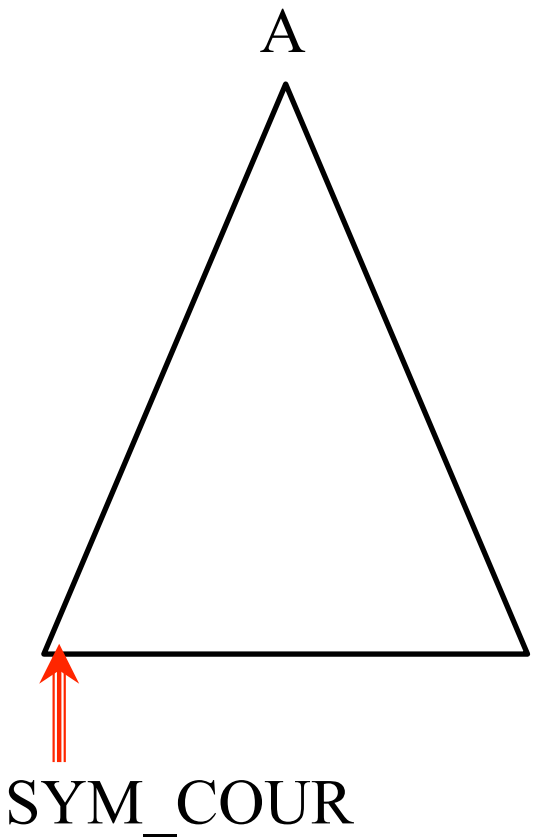
Traitement:

vérifier 'a' dans la chaine à cette position

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

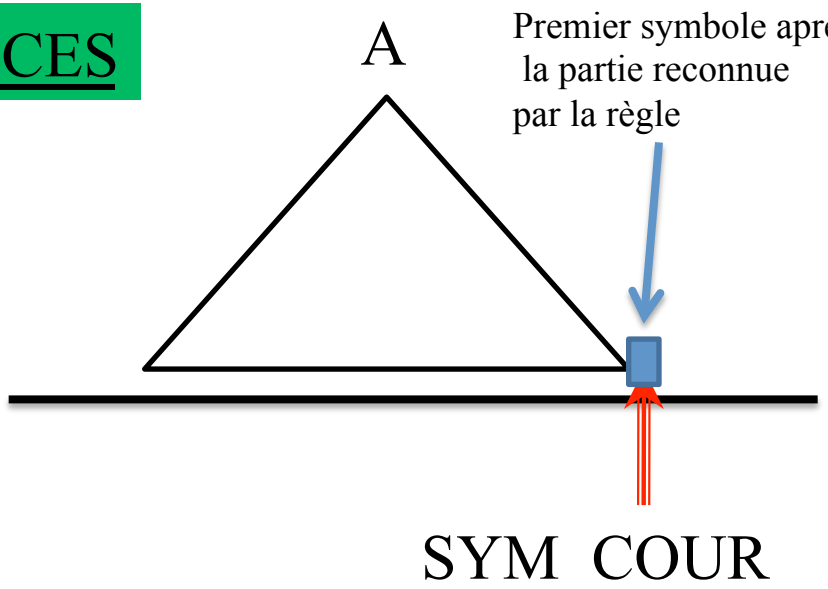
CAS D'UN NON TERMINAL

ETAT INITIAL

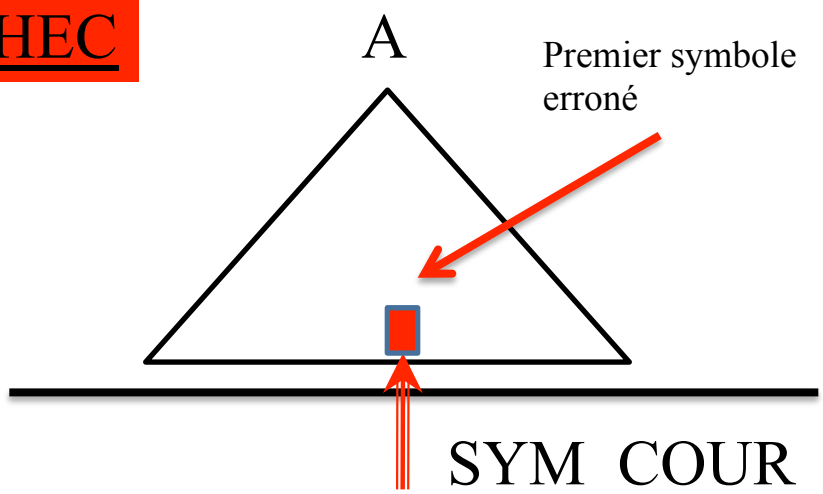


T(A)=A();

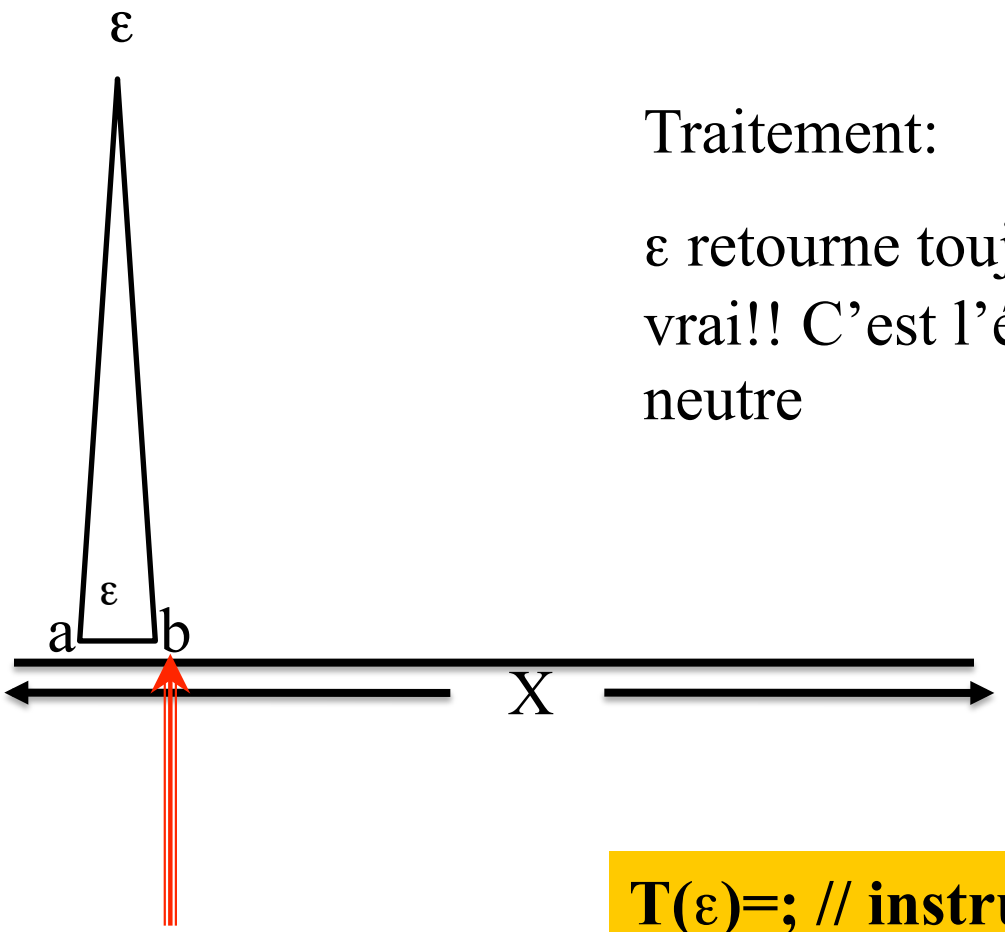
SUCCES



ECHEC



LE CAS DE ε



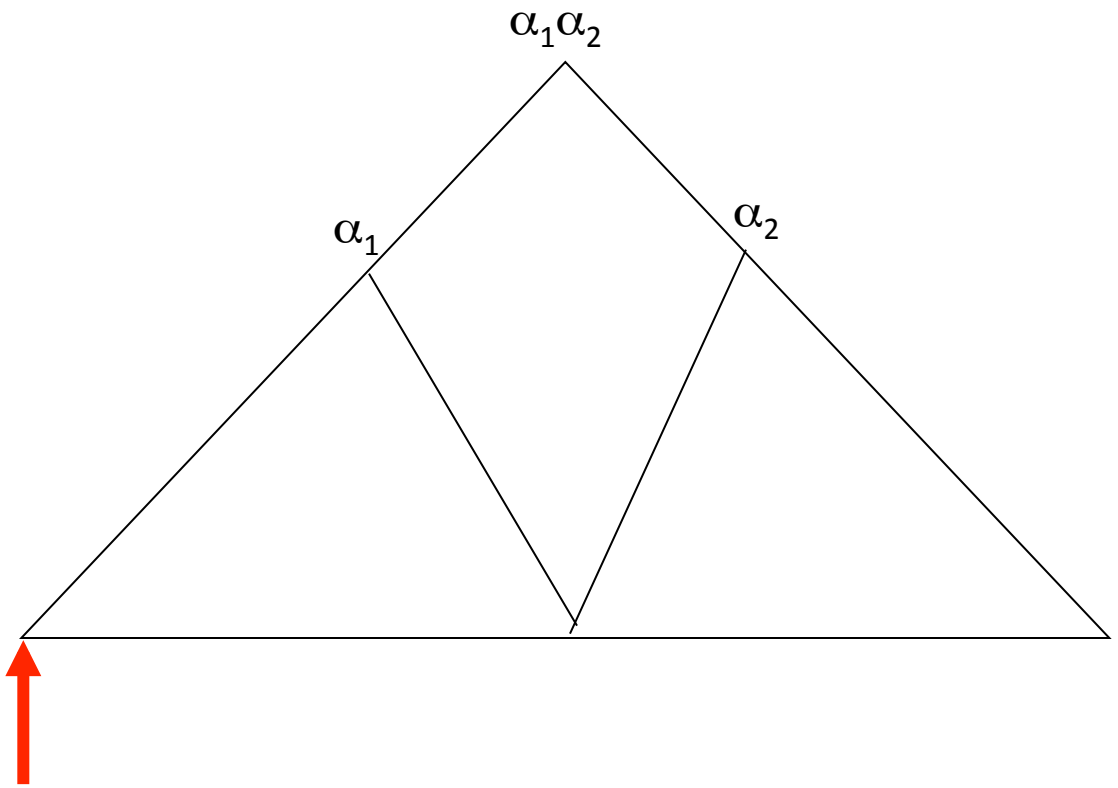
Traitement:

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

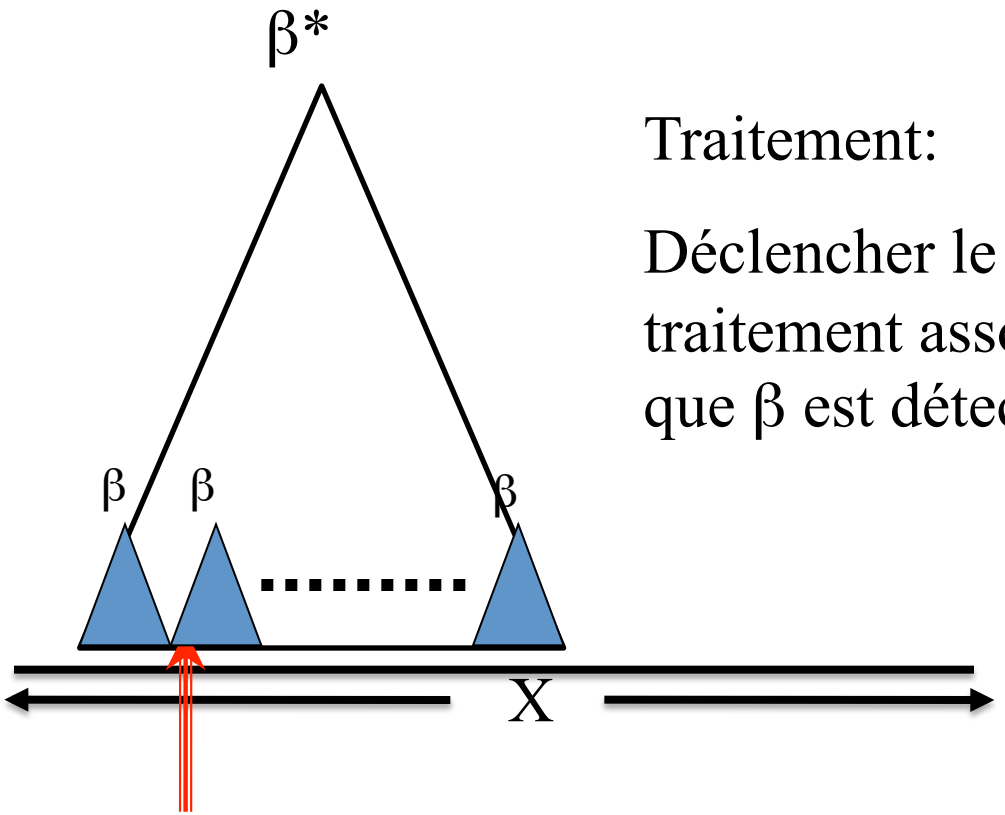
T(ε)=; // instruction vide

CAS DE $\alpha_1\alpha_2$

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



CAS DE β*



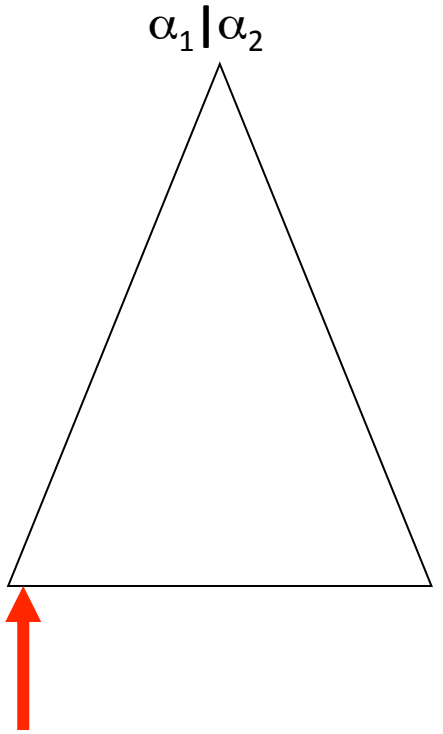
Traitement:

Déclencher le traitement associé β tant que β est détectée!

T(β*)=TANTQUE SymCour.CODE dans FIRST(β) FAIRE
ζβ;
FINTANTQUE;

CAS DE $\alpha_1|\alpha_2$

CAS SYM_COUR.CODE PARMI
D(α_1, $\alpha_1|\alpha_2$): T(α_1);
D(α_2, $\alpha_1|\alpha_2$): T(α_2);
AUTRE CAS:
ERREUR(mes)
FIN DE CAS



SI ε appartient au L(α_2)
ALORS D(α_2, $\alpha_1|\alpha_2$)=FIRST(α_2, $\alpha_1|\alpha_2$) U FELLOW(α_2, $\alpha_1|\alpha_2$)
SINON D(α_2, $\alpha_1|\alpha_2$)=FIRST(α_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
$\beta_1\beta_2$	$\zeta\beta_1$; $\zeta\beta_2$;

$\beta_1 \mid \beta_2$	Switch (SymCour.**CLS**) {
	case **D($\beta_1 \mid \beta_2, \beta_1$)** : $\zeta\beta_1$; break;
	case **D($\beta 1 \mid \beta 2, \beta 2$)** : $\zeta\beta_2$; 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 **)**

A VOS MACHINES
et
BON COURAGE