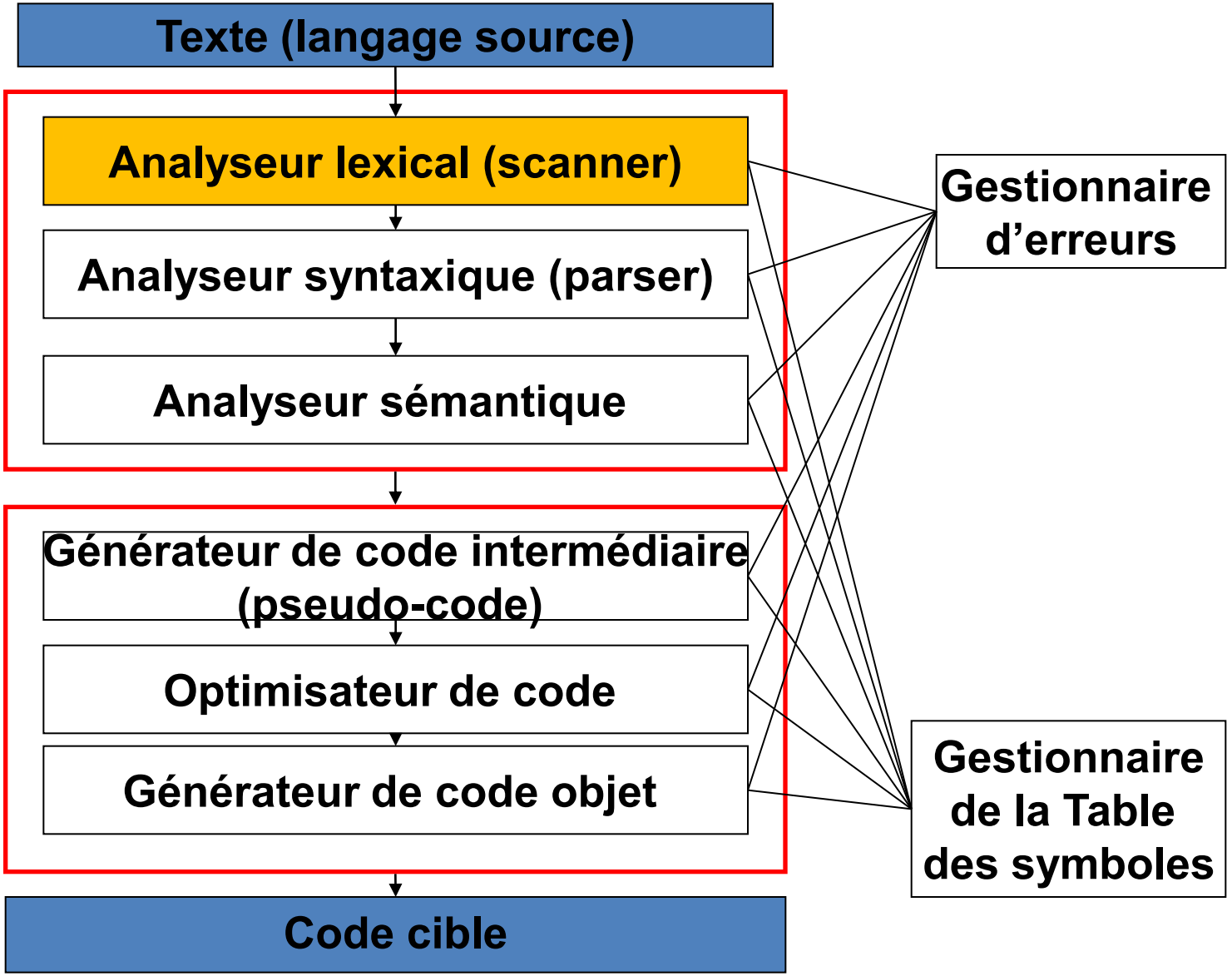
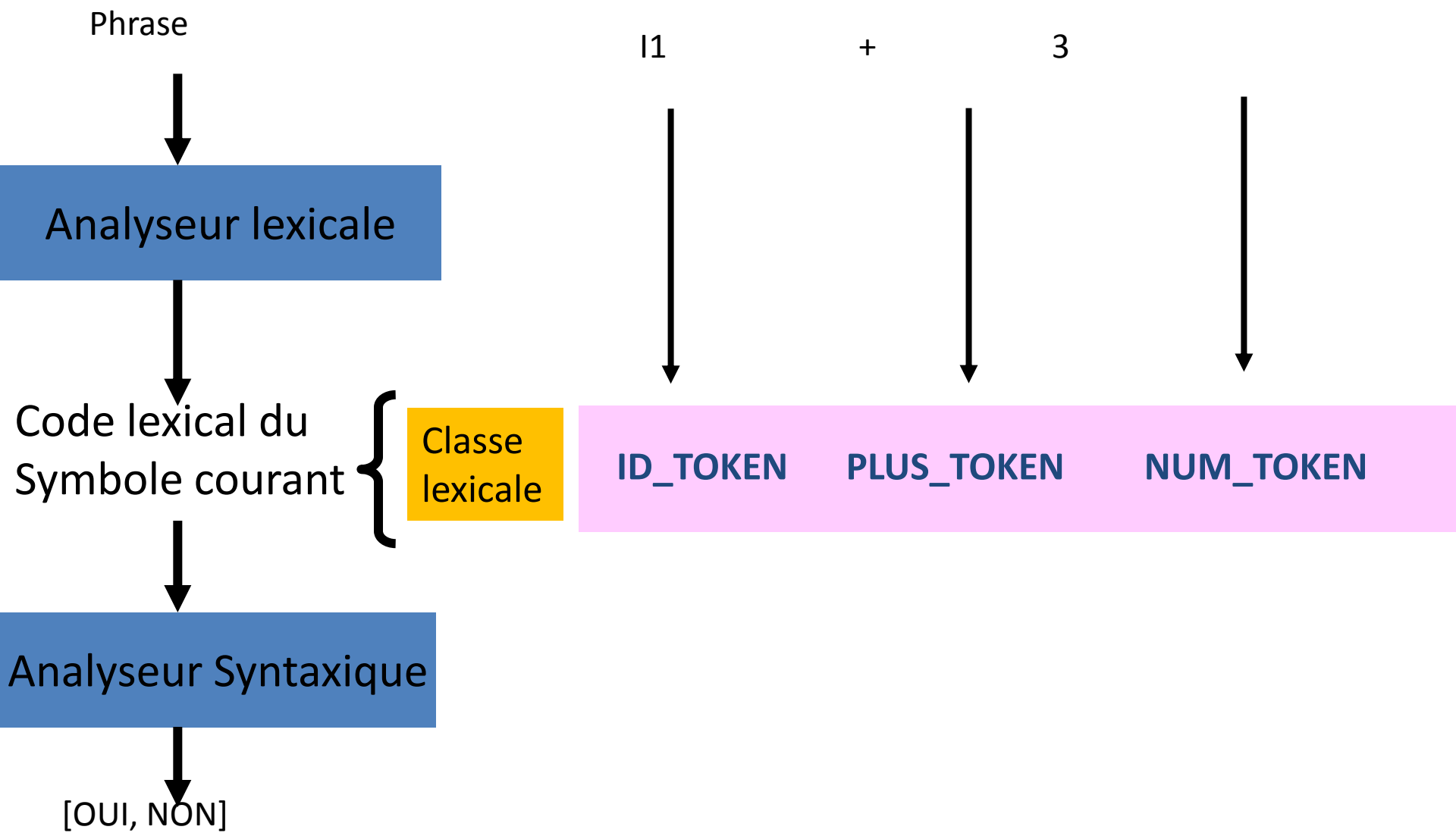
# ECRITURE D'UN MINI COMPILATEUR

# 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

# Traduction de la partie droite D'une règle

| $\alpha$ | Traitement associé a $\alpha$ |
|---|---|
| $a \in V_t$<br>a_TOKEN | if (SymCour.**CLS**==**a_TOKEN**) SymboleSuivant ;<br>else ERREUR(CODE_ERR) ; |
| $A \in V_n$ | A(); // appel de la procédure associée à la règle A |
| $\varepsilon$ | ; //instruction vide |
| $\beta_1\beta_2$ | $\zeta\beta_1$; $\zeta\beta_2$; |
| $\beta_1 \mid \beta_2$ | Switch (SymCour.**CLS**) {<br>case **D($\beta_1 \mid \beta_2, \beta_1$)** : $\zeta\beta_1$; break;<br>case **D($\beta 1 \mid \beta 2, \beta 2$)** : $\zeta\beta_2$; break;<br>default ERREUR(CODE_ERR)<br>} |
| $\beta^*$ | while (**SymCour.CLS in β'** ) { $\zeta\beta$; } |

# ENSEMBLE DIRECTEURS VS CLASSES LEXICALES

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.CLS**
- 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.cls==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.

Les erreurs:

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

| | | |
|---|---|---|
| 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