

3. COMPILEATION ET INTERPRETATION DES EXPRESSIONS ARITHMETIQUES.

L'objectif est de compiler et interpréter des programmes sources écrits dans le langage mini-Pascal dont la syntaxe du langage est réduite:

- aux expressions arithmétiques faisant intervenir des constantes et des variables entières, des opérateurs arithmétiques, des parenthèses;
 - aux instructions permettant d'initialiser une variable entière: instruction de lecture d'une variable entière et instruction d'affectation d'une variable entière;
 - à l'instruction d'écriture d'une constante entière ou de type chaîne de caractères, d'une variable entière, d'une expression arithmétique.

⇒ Définition des constantes entières et de type chaîne de caractères

Le programmeur peut définir des constantes dans la partie déclaration de constantes, qui est la première partie du programme.

⇒ Définition des variables entières

Le programmeur peut définir des variables entières dans la partie déclaration de variables du programme, qui se trouve au début du programme après la partie déclaration de constantes.

⇒ Les expressions arithmétiques

Les expressions sont évaluées de gauche à droite par le compilateur

Tous les opérateurs binaires ont la même priorité et sont associatifs à droite

Exemples:	$1+2+3$	est évalué comme $1+(2+3)$,	soit: 6
	$3*2-1$	est évalué comme $3*(2-1)$,	soit: 3
	$2-3*5+10/2^5$	est évalué comme $2-(3*(5+(10/(2^5))))$,	soit: -16
	$(2-3)*5+(10/2)^5$	est évalué comme $(2-3)*(5+((10/2)^5))$,	soit: -30

L'opérateur unaire - a une priorité supérieure aux quatre opérateurs binaires précédents.

Exemple: $-5--2+3/1+-2$ est évalué comme $(-5)-((-2)+(3/(1+(-2))))$, soit: 0

⇒ L'instruction d'affectation

Une instruction d'affectation est de l'une des formes suivantes:

identificateur_de_variable := nombre_entier
identificateur_de_variable := expression_arithmétique
identificateur_de_variable := constante_entière
identificateur_de_varaible := variable_entière

Cette instruction permet d'affecter une valeur à une variable entière.

Exemples: (avec les déclarations: VAR i; j;) i := 4; j := i + 5;

⇒ L'instruction de lecture

L'instruction LIRE permet à l'utilisateur d'affecter une valeur à une variable, en entrant cette valeur au clavier de l'ordinateur. Si plusieurs variables figurent dans la liste de l'instruction LIRE, le programme demandera à l'utilisateur d'entrer successivement des valeurs au clavier, chaque valeur devant être terminée par un "Enter".

Exemple: (avec les déclarations: VAR i, j;) LIRE(i, j);

⇒ L'instruction d'écriture

→ **L'instruction d'affichage**
Cette instruction provoque l'affichage à l'écran de la valeur des expressions, des constantes ou des variables figurant dans la liste entre parenthèses.

L'instruction ECRIRE lorsqu'elle est utilisée sans argument provoque le déplacement du curseur au début de la ligne suivante à l'écran.

Exemples: `ECRIBRE(); ECRIBRE('Le carré de ' i ' vaut ' carre(i));`

⇒ Un bloc d'instructions

→ **Un bloc d'instructions**
Un bloc d'instructions est une séquence d'instructions placée entre les deux mots réservés DEBUT et FIN.

Un bloc d'instructions est :

```
DEBUT      i := 1;  
          j := i * 2  
FIN
```

⇒ La grammaire du langage mini-Pascal

La grammaire G_0 du langage mini-Pascal est définie par le quadruplet $(T_0, N_0, \text{PROG}, P_0)$ où:

- T_0 est l'ensemble des symboles terminaux:

```
T0 = { 'PROGRAMME', 'CONST', 'VAR', 'DEBUT', 'FIN', 'LIRE', 'ECRIRE', 'IDENT', 'ENT', ',', ';', '=', ':', '(', ')', '+', '-', '*', '/' }
```

- N_0 est l'ensemble des symboles non terminaux:

$N_0 = \{ \text{PROG}, \text{DECL_CONST}, \text{DECL_VAR}, \text{BLOC}, \text{INSTRUCTION}, \text{AFFECTATION}, \text{LECTURE}, \text{ECRITURE}, \text{ECR EXP}, \text{EXP}, \text{TERME}, \text{OP BIN} \}$

- PROG est le symbole non terminal initial (PROG N₀)

- P_0 est l'ensemble des règles de production:

PROG	$\rightarrow \text{'PROGRAMME' } \text{'IDENT' } \{ \text{ DECL_CONST } \text{ DECL_VAR } \text{ BLOC }$
DECL_CONST	$\rightarrow \text{'CONST' } \text{'IDENT' } \text{ '=' } (\text{'ENT' } \mid \text{'CH'}) \{ \text{ 'IDENT' } \text{ '=' } (\text{'ENT' } \mid \text{'CH'}) \} \{$
DECL_VAR	$\rightarrow \text{'VAR' } \text{'IDENT' } \{ \text{ 'IDENT' } \} \{$
BLOC	$\rightarrow \text{'DEBUT' } \text{ INSTRUCTION } \{ \text{ 'INSTRUCTION' } \} \text{ 'FIN'}$
INSTRUCTION	$\rightarrow \text{AFFECTATION } \mid \text{LECTURE } \mid \text{ECRITURE } \mid \text{BLOC}$
AFFECTATION	$\rightarrow \text{'IDENT' } \text{ ':' } \text{ EXP}$
LECTURE	$\rightarrow \text{'LIRE' } \text{ '(' } \text{'IDENT' } \{ \text{ 'IDENT' } \} \text{ ')'$
ECRITURE	$\rightarrow \text{'ECRIRE' } \text{ '(' } [\text{ ECR_EXP } \{ \text{ 'ECR_EXP' } \}] \text{ ')'$
ECR_EXP	$\rightarrow \text{EXP } \mid \text{'CH'}$
EXP	$\rightarrow \text{TERME } \text{ SUITE_TERME}$
SUITE_TERME	$\rightarrow \varepsilon \mid \text{ OP_BIN } \text{ EXP}$
TERME	$\rightarrow \text{'ENT' } \mid \text{'IDENT' } \mid (\text{ 'EXP' }) \mid \text{ '-' } \text{ TERME}$
OP_BIN	$\rightarrow \text{ '+' } \mid \text{ '-' } \mid \text{ '*' } \mid \text{ '/' }$

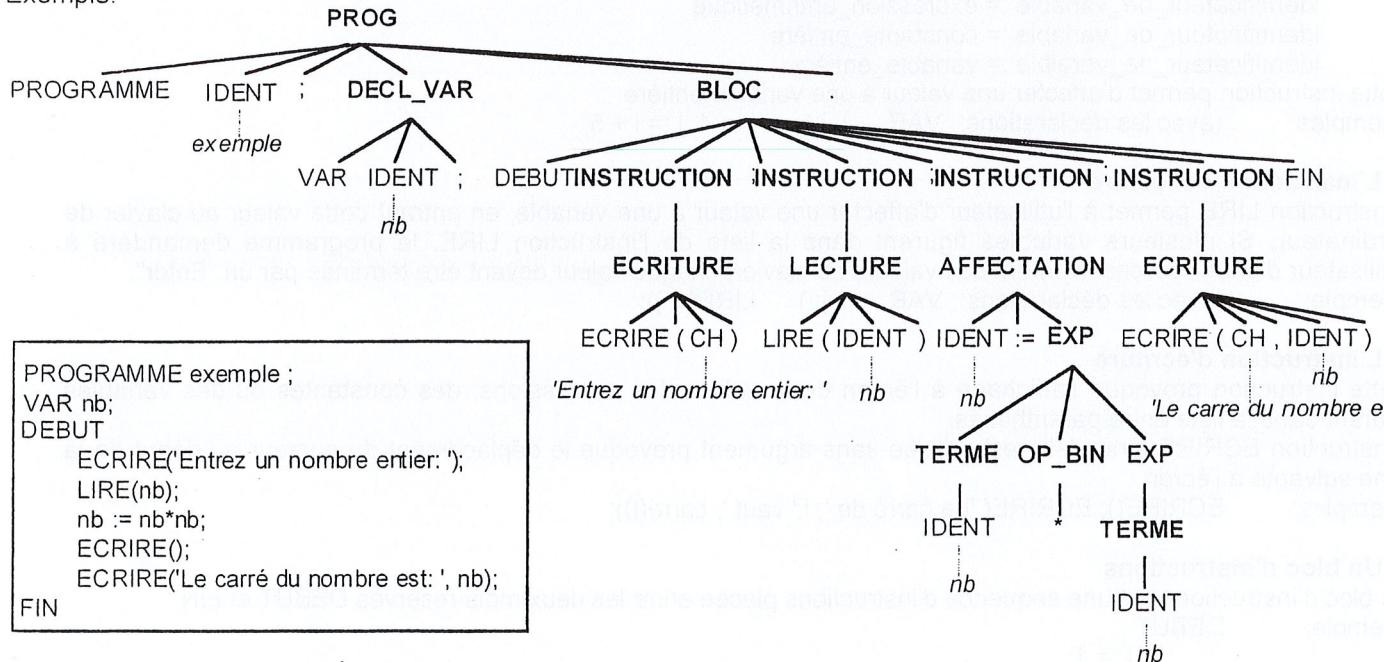
[] = optional

{ } = bundle > 0

Le formalisme utilisé pour décrire les règles de production est défini par:

	symbole de réécriture
'suite_de_lettres'	un symbole terminal
suite_de_lettres	un symbole non terminal
[x]	zéro ou une occurrence de x
{ x }	zéro, une ou plusieurs occurrences de x

Exemple:



Arbre syntaxique pour le programme exemple

Le travail de l'analyseur syntaxique consiste en quelque sorte à construire l'arbre syntaxique (appelé aussi arbre de dérivation) à partir de la grammaire du langage, pour l'ensemble des phrases du programme source. L'analyse du programme peut se faire de plusieurs manières, nous choisissons:

- d'effectuer l'**analyse syntaxique de gauche à droite**: les unités lexicales sont déterminées en parcourant le programme source de gauche à droite;
 - d'effectuer une **analyse syntaxique descendante**: il s'agit de construire l'arbre syntaxique en partant de la racine (le symbole non terminal initial PROG) et en descendant vers les feuilles (les symboles terminaux);
 - d'utiliser une **méthode d'analyse déterministe**: à chaque étape de l'analyse, l'analyseur n'a qu'une possibilité (il n'a pas à choisir parmi plusieurs possibilités ce qui l'obligerait à revenir sur ces pas pour essayer une autre possibilité dans la mesure où il aurait fait un mauvais choix: technique de retour en arrière (backtracking)).
- La méthode descendante déterministe que nous allons utiliser est la **méthode de la descendante récursive**.

⇒ Grammaire LL(1)

L'utilisation de la méthode descendante récursive impose une contrainte sur le type de la grammaire. Cette méthode ne peut être utilisée qu'avec les **grammaires LL(1)**. Le premier L (Left) signifie que l'analyse se fait de gauche à droite, le deuxième L (Left) que l'analyseur effectue des dérivations gauches, avec une inspection d'un (1) symbole en avant.

Exemple de dérivation gauche:

Soit le programme en mini-Pascal suivant:

```
PROGRAMME exemple;
DEBUT
    ECRIRE();
    ECRIRE('Coucou')
FIN
```

Une dérivation gauche est:

```
PROG → 'PROGRAMME' 'IDENT' ';' 'DEBUT' INSTRUCTION ';' INSTRUCTION 'FIN'
      → 'PROGRAMME' 'IDENT' ';' 'DEBUT' ECRIRE '(' ')' ';' INSTRUCTION 'FIN'
      → 'PROGRAMME' 'IDENT' ';' 'DEBUT' ECRIRE '(' ')' ';' ECRIRE '(' CH ')' 'FIN'
```

L'analyseur syntaxique doit fonctionner de façon déterministe en disposant d'un symbole en avance sur celui qu'il est en train de traiter. Pour toutes les productions de la forme: $X \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$, il doit être certain qu'une seule de ces productions peut être utilisée.

1ère contrainte: Pour toutes les productions de la forme: $X \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$,

il faut que $\text{PREMIER}(\alpha_i) \cap \text{PREMIER}(\alpha_j) = \emptyset$, pour tout i différent de j .

$\text{PREMIER}(\alpha)$ où α est une chaîne de symboles quelconques, est l'ensemble de tous les terminaux qui sont le début d'une chaîne dérivée à partir de α .

Exemple: $\text{PREMIER}(\text{INSTRUCTION}) = \{ \text{'IDENT'}, \text{'LIRE'}, \text{'ECRIRE'}, \text{'DEBUT'} \}$

Cette condition n'est pas suffisante pour garantir le déterminisme. Considérons, par exemple, la grammaire $S \rightarrow [x] x$. Ceci est, en fait une écriture plus condensée de: $S \rightarrow X a \quad (1) \quad X \rightarrow a \mid \phi \quad (2)$. Lorsque l'analyseur examine la phrase a , il ne sait pas si le symbole a est dérivé à partir de (1) ou de (2):

$S \rightarrow X a \rightarrow \phi a \quad (\text{à partir de } (1))$
 $S \rightarrow a \quad (\text{à partir de } (2))$

2ème contrainte: Pour chaque symbole non terminal X à partir duquel on peut dériver la chaîne vide,

$\text{PREMIER}(X) \cap \text{SUIVANT}(X) = \emptyset$

$\text{SUIVANT}(X)$ est l'ensemble des terminaux figurant immédiatement après le non terminal X dans un enchaînement de productions de la grammaire. Exemple: $\text{SUIVANT}(\text{INSTRUCTION}) = \{ ',', \text{'FIN'} \}$

La grammaire est LL(1) si et seulement si toutes les productions vérifient les deux contraintes ci-dessus. Si la grammaire est LL(1), l'analyse syntaxique peut être effectuée sans retour en arrière, i.e. de façon déterministe, en utilisant la technique descendante récursive.

Remarque: Une grammaire récursive à gauche n'est pas LL(1).

Vous pourrez vérifier que la grammaire G_0 du mini-Pascal est LL(1).

Exemple:

$\text{LECTURE} \rightarrow \text{'LIRE'} '(' \text{'IDENT'} \{ ',', \text{'IDENT'} \} ')$ est une forme condensée de: $\text{LECTURE} \rightarrow \text{'LIRE'} '(' \text{'IDENT'} \cdot X \cdot ')$
 $X \rightarrow \cdot \mid ',' \text{'IDENT'} X \mid \phi$

or $\text{SUIVANT}(X) = \{ ',' \}$ et $\text{PREMIER}(X) = \{ ',' \}$ donc $\text{SUIVANT}(X) \cap \text{PREMIER}(X) = \emptyset$

↳ Méthode d'analyse syntaxique descendante recursive

Etant une grammaire LL(1), il est toujours possible d'écrire directement un programme qui simule la construction de l'arbre syntaxique propre à la chaîne donnée en entrée. Ce programme est établi de façon systématique selon les règles suivantes. Il utilise la procédure ANALEX qui lui fournit l'unité lexicale suivante dans le programme source et une variable globale UNILEX qui contient l'unité lexicale courante.

Il faut définir une fonction booléenne récursive sans paramètres, pour chaque non terminal de la grammaire. La résultat de la fonction est faux dès qu'une erreur syntaxique est détectée dans la fonction. Si la production est $A \rightarrow W$, le corps de la fonction est défini en fonction de la forme de W , selon les principes suivants:

Exemples: **function INSTRUCTION : boolean;**

```

begin
    INSTRUCION := (AFFECTION or LECTURE or ECRITURE or BLOC);
end;

function AFFECTION : boolean;
begin
    if UNILEX = IDENT then
        begin
            UNILEX := ANALEX;
            if (UNILEX = AFF) then
                begin
                    UNILEX := ANALEX;
                    AFFECTATION := EXP
                end
            else
                AFFECTATION := false { erreur syntaxique dans une instruction
                                      d'affectation; := attendu }
        end
    else
        AFFECTATION := false { erreur syntaxique dans une instruction d'affectation:
                               identificateur attendu }
end;
else
    AFFECTATION := false { erreur syntaxique dans une instruction d'affectation:
                           identificateur attendu }

function LECTURE : boolean;
var fin, erreur: boolean;
begin
    if (UNILEX = MOTCLE) and (CHAINE = 'LIRE') then
        begin
            UNILEX := ANALEX;
            if (UNILEX = PAROUV) then
                begin
                    UNILEX := ANALEX;
                    if (UNILEX = IDENT) then
                        begin
                            UNILEX := ANALEX;
                            fin := false;
                            erreur := false;
                            repeat

```

```

        if (UNILEX = VIRG) then
        begin
            UNILEX := ANALEX;
            if (UNILEX = IDENT) then
                UNILEX := ANALEX;
            else
                begin
                    fin := true;
                    erreur := true
                end
            end
            fin := true
        until fin;
        if erreur then
            LECTURE := false { erreur syntaxique dans instruction de
                               lecture: identificateur attendu }
        else if ( UNILEX = PARFER ) then
        begin
            UNILEX := ANALEX;
            LECTURE := true
        end
        else
            LECTURE := false { erreur syntaxique dans instruction de
                               lecture: ) attendu }
    end
    else
        LECTURE := false { erreur syntaxique dans instruction de
                           lecture: identificateur attendu }
end
else
    LECTURE := false { erreur syntaxique dans instruction de lecture: (
                           attendu )}

end;
else
    LECTURE := false { erreur syntaxique dans instruction de lecture: mot-clé
                           LIRE attendu }

end;

function ECRITURE : boolean;
var fin, erreur: boolean;
begin
    if (UNILEX = MOTCLE) and (CHAINE = 'ECRIRE') then
    begin
        UNILEX := ANALEX;
        if (UNILEX = PAROUV) then
        begin
            UNILEX := ANALEX;
            erreur := false;
            if ( ECR_EXP ) then
            begin
                UNILEX := ANALEX;
                fin := false;
                repeat
                    if (UNILEX = VIRG) then
                    begin
                        UNILEX := ANALEX;
                        erreur := not ECR_EXP;
                        if erreur then
                            fin := true;
                    end
                    else
                        fin := true
                until fin;
            end;
            if erreur then
                ECRITURE := false { erreur syntaxique dans instruction
                                   d'écriture: expression incorrecte }
            else if (UNILEX = PARFER) then

```

```

begin
    UNILEX := ANALEX;
    ECRITURE := true
end
else
    ECRITURE := false      { erreur syntaxique dans instruction
                                d'écriture: ) attendu }
end
else
    ECRITURE := false      { erreur syntaxique dans instruction d'écriture: ( attendu }
end
else
    ECRITURE := false      { erreur syntaxique dans instruction d'écriture: mot-clé
                                ECRIRE attendu }
end;

```

■ Ajout de la variable globale UNILEX

UNILEX, la dernière unité lexicale reconnue dans le programme source, de type T_UNILEX

■ Ecriture des fonctions associées à chaque non terminal de la grammaire G_0

Il faut écrire une fonction booléenne récursive sans paramètres, pour chaque non terminal de G_0 en utilisant la méthode explicitée ci-dessus, soit 12 fonctions.

■ La procédure ANASYNT

```

procedure ANASYNT;
begin
    UNILEX := ANALEX;
    if (PROG) then
        writeln('Le programme source est syntaxiquement correct');
    else
        ERREUR(3); { erreur n°3: erreur syntaxique }
end

```

■ Le programme principal

La programme principal de l'analyseur syntaxique doit:

- appeler la procédure INITIALISER,
- appeler la procédure ANASYNT,
- appeler la procédure TERMINER.

De façon à tester l'analyseur syntaxique, on pourra ajouter dans chacune des 12 fonctions associées aux symboles non terminaux de la grammaire G_0 , un affichage du nom de la fonction. Ainsi, le programme affichera à l'écran la séquence d'appels des différentes fonctions lors de la compilation des programmes mini-Pascal.

A ce point, le programme est capable de lire un fichier source contenant un programme mini-Pascal et de vérifier si les phrases du programme sont conformes aux règles de production contenues dans la grammaire G_0 du langage mini-Pascal. Le programme affiche un message d'erreur indiquant qu'une erreur syntaxique a été détectée, dans le cas où une phrase est mal formée. Le programme s'arrête à la première erreur syntaxique trouvée.

2. Analyse sémantique

Le langage mini-Pascal contient des contraintes qui ne peuvent pas être formalisées dans les règles de production d'une grammaire LL(1). Une de ces contraintes est, par exemple, le fait qu'on ne peut pas utiliser dans le programme un identificateur de variable n'ayant pas été défini préalablement comme variable. Dans une instruction d'affectation, le langage exige que les parties gauche et droite du signe d'affectation soient du même type. La phase d'analyse sémantique va vérifier ces contraintes. L'analyse sémantique est introduite au sein du programme d'analyse syntaxique.

■ Fonction associée au non terminal DECL_CONST de la grammaire G_0

Lors de la déclaration d'une constante, il faut effectuer la vérification sémantique suivante:

le nom de la constante ne doit pas être un nom d'identificateur déjà présent dans la table des identificateurs.

Si ce nom est déjà défini, alors il faut produire l'erreur « identificateur déjà déclaré ». Sinon, ce nom n'a pas encore été défini dans la table des identificateurs, il faut ajouter la constante dans la table des identificateurs en remplissant les différents champs spécifiques à cette constante:

- le nom de la constante (nom);
- le type de l'identificateur (typ) est: constante;