République Algérienne Démocratique et Populaire

Ministère de l'enseignement supérieur et de la recherche scientifique

Université de Jijel

Faculté des sciences de l'ingénieur
Département de l'informatique
4ième année génie informatique
Module de compilation

Les grammaires et les compilateurs

Créé par :

ARIES Abd elkarim

Introduction

Il existe 4 types de grammaires : Type0 c'est le plus général et il n'y a aucune restriction sur ses règles, Type1 ou grammaire contextuelle qui a des règles de la forme $uAv \to uWv$, avec $A \in N;\ u, v \in (V \cup N)^*;\ W \in (V \cup N)^+$, Type1 qui s'écrit de la forme $A \to W$ avec $W \in (V \cup N)^*$ et $A \in N$. Ce type de grammaire s'appelle la grammaire hors contexte et le Type3 qui s'écrit de la forme $A \to vB\,(\text{resp}\ A \to Bv)$ ou $A \to v$ avec $A, B \in N$ et $v \in V^*$.

On s'intéresse ici par les grammaires de type2, on essaye de créer un logiciel qui vérifie si une grammaire est de type2, si oui on va appliquer des fonctions sur elle. Les fonctions qu'on peut effectuer sont rendre la réduite par la suppression des règles non productives et celles non accessibles, propre est ceci est obtenu par la suppression des epsilon transitions et des règles unitaires, rendre la pas récursive à gauche et factoriser à gauche. Après tous ces opérations on la rendre sous forme de Chomsky puis sous forme de Greibach.

La deuxième partie consiste à construire un compilateur d'un simple langage qui est très identique à l'algorithmiques, à un minimum des variables et de structure, avec un environnement de développement attaché à ce langage dirigé vers la correction des erreurs qu'un programmeur sur ce langage peut effectuer, et puis il faire une transformation du code algorithmique compilé positive vers un langage de programmation tel que Pascal pour le compiler et l'exécuter.

PARTIE1 :
Les grammaires

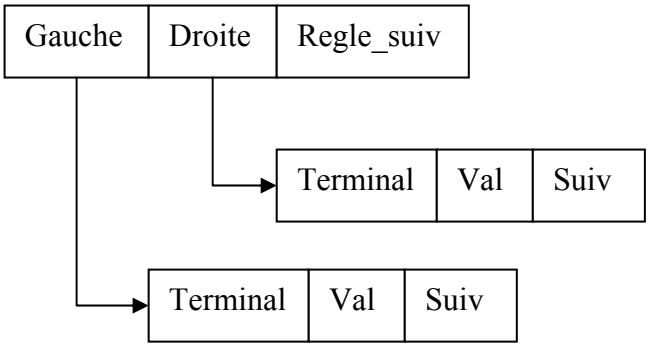
1. **La structure utilisée.**
2. **Les fonctions sur la grammaire.**
3. **Le logiciel.**

(01) La structure utilisée.

La grammaire est une suite des règles, et pour cette raison on peut créer une grammaire comme étant une liste chaînée d'un enregistrement qui s'appelle « **Regle** » et le type qui lui pointe est nommé « **La_regle** » ; donc on peut définir la grammaire comme suite « **Grammaire : La_regle ;** ». Et l'enregistrement « **Regle** » contient préalablement un pointeur sur la règle suivante qui se définit «**Regle_suiv :La regle**».

Une règle contient deux parties dont ils ont une structure identique ; donc on l'appelle « **Partie** » qui est en réalité un pointeur sur un enregistrement « **Part** » cette dernière contient un booléen qui s'appelle « **terminal** » et qui indique si le symbole est terminale ou non, une chaîne de caractère « **val** » qui indique la valeur de ce symbole et finalement un pointeur de type **Partie** s'appelle « **Suiv** » qui pointe sur le symbole suivant. Et donc l'enregistrement « **Regle** » se compose aussi de deux pointeurs de type « **Partie** » ont les noms « **Gauche** » et « **Droite** ».

```
partie=^part;
part=record
       terminal: boolean;
         val: string[10];
         suiv: partie;
       end;
la_regle=^regle;
regle=record
         gauche: partie;
        droite: partie;
         regle_suiv: la_regle;
         end;
```



(02) Les fonctions sur la grammaire :

(a)Vérification si de type 2 :

La fonction qui vérifier si une grammaire est de type 2 donnée par :
« *function est_type2(gram:la_regle):boolean;* »

Son algorithme est comme suite :

- S'il y a un terminal à gauche elle est prend la valeur « false ».
- Sinon s'il y a plus d'un non terminal, elle prend la valeur « false ».
- Sinon elle prend la valeur « true ».

(b) Rendre une grammaire productive :

La procédure sert à créer une grammaire productive à partir d'une autre grammaire, son entête est donné par :
« *procedure productive(gram:la_regle;var g_fct:la_regle;axiome:string);* »

Son algorithme est donné par :

- Créer un ensemble « ens »des non terminaux qui donnent un terminal c'est-à-dire « $ens = \{ A \to \alpha , \alpha \in (V)^* \}$ ».
- Ajouter « ens » les non terminaux qui déterminent des terminaux et/ou des non terminaux de cet ensemble « $ens = ens \cup \{ A \to \alpha , \alpha \in (V \cup ens)^* \}$ ».
- Supprimer toutes les règles que sa partie gauche n'appartient pas à « ens » ; c'est-à-dire : « $P = P - \{A \to \alpha / A \notin ens\}$ ».
- Si l'axiome appartient à « ens » donc on élimine toutes les règles et on crée une seule règle « $Axiome \to \varepsilon$ ».

(c) Rendre une grammaire accessible :

La procédure sert à rendre une grammaire accessible, son entête est donné par :
« *procedure accessible(var g_fct:la_regle;axiome:string);* »

Son algorithme est donné par :

- Crée une liste qui ne contient que l'axiome c'est-à-dire « $ens = \{Axiome\}$ ».
- Ajouter à l'ensemble des accessibles les non terminaux qui s'apparaissent à la partie droite d'une règle dont sa partie gauche appartient à cet ensemble ; c'est-à-dire « $ens = ens \cup \{B / A \to \alpha , A \in ens , B \in \alpha \}$ ».
- Tous les règles dont la partie gauche n'égale pas à un non terminal appartenant à l'ensemble des accessibles. C'est-à-dire supprimer « $A \to \alpha / A \notin ens$ ».

(d) Rendre une grammaire sans ε-transition :

La procédure sert à rendre une grammaire sans ε-transition, son entête est donné par : « *procedure sans_epsilon(var g_fct:la_regle; var axiome:string);*»

Son algorithme est donné par :

- Créer un ensemble « ens » telle que « $ens = ens \cup \{A / A \to \varepsilon\}$ ».
- Ajouter à l'ensemble les non terminaux qui donnent un ε ; c'est-à-dire « $ens = ens \cup \{A \to \alpha , \alpha \in (ens)^*\}$ ».
- Supprimer tous les règles tel que la partie gauche appartient à « ens » ; c'est-à-dire supprimer « $A \to \alpha / A \in ens$ »
- Supprimer les non terminaux appartenant à « ens » de la partie droite des règles restantes ; par exemple « $A \to aBC$ » et « $B \to \varepsilon$ » donc « $A \to aC$ ».
- Si l'axiome appartient à « ens » donc on supprime toutes les autres règles et on crée la règle suivante « $Axiome \to \varepsilon$ ».

(e) Rendre une grammaire non unitaire :

Une règle unitaire c'est lorsque son non terminal gauche donne un seul autre non terminal à la partie droite de cette règle. Cette procedure élimine ces règles, son entête est donné par : « *procedure pas_unitaire(var g_fct:la_regle;axiome:string);* ».

Son algorithme est le voici :

- Créer une ensemble des non terminaux pour chaque non terminal « $N_0(x) = \{ x \}$».
- Ajouter à chaque ensemble des non terminaux de la façon suivante : « $N_{i+1}(x) = N_i(x) \cup \{Y \in N / A \to Y , A \in N_i(x)\}$ ».
- Supprimer toutes les règles unitaires.
- Si on a « $B \in N(A) / B \to \alpha$ » on va créer « $A \to \alpha$ ».
- Supprimer les règles qui seront non accessibles.

(f) Rendre une grammaire non récursive à gauches :

On peut dire qu'une règle est récursive à gauche si on trouve que le non terminal gauche est le même pour le premier de la partie droite ; on dit donc qu'il y a une récursivité à gauche immédiate. Aussi si après certain nombre de remplacement du non terminal premier s'il existe, on trouve une règle récursive à gauche ; c'est la récursivité à gauche non immédiate. Cette procédure garantie qu'une grammaire sera non récursive à gauche. Son entête est le suivant :
« *procedure recur_gauche(var g_fct:la_regle);* ».

Son algorithme est décrit comme suivant :

- Récursivité immédiate : si « $A_i \to A_i\ \alpha$ » pour chaque « $A_i \to \beta$ » on va créer quatre règles comme suite « $A_i \to \beta\ R_i$ » « $A \to \beta$ » « $R_i \to \alpha\ R_i$ » « $R_i \to \alpha$ » et on supprime les deux premières.
- Remplacement : Remplacer toute règle de la forme « $A_i \to A_j\ \alpha$ » par A_j si on trouve « $A_i \to A_i\ \alpha$ » on va vers récursivité immédiate sinon on va vers la règle suivante.

(g) Rendre une grammaire sous FNC :

La fonction sert à créer une grammaire sous forme normale de Chomsky à partir d'une autre grammaire son entête est donné par :
« *procedure FNC(gram:la_regle;var g_fnc:la_regle);* »

Et l'algorithme de faire ça est donné par :

- Si la règle est de la forme « $A \to B\,C / A,B,C \in N$ » ou « $A \to x / x \in V$ » on la copie dans la grammaire de sortie.
- Sinon, si elle est de la forme « $A \to x\ \alpha / x \in V$ » on éclate cette règle en deux ou trois règles :
  - Si ($\alpha \in N$ et $\|\alpha\| = 1$) « $A \to C_i \alpha$ » « $C_i \to x$ ».
  - Sinon « $A \to C_i C_{i+1}$ » « $C_i \to x$ » « $C_{i+1} \to \alpha$ ».
- Si elle est de la forme « $A \to B\ \alpha / B \in N$ » on éclate cette règle en deux ou trois règles : « $A \to B\,C_i$ » « $C_i \to \alpha$ ».
- On fait retour à l'étape une jusqu'à la fin des règles de la grammaire.

(h) Rendre une grammaire sous FNG :

La fonction sert à créer une grammaire sous forme normale de Greibach à partir d'une autre grammaire son entête est donné par :
« *procedure FNG(gram:la_regle;var g_fng:la_regle);* »

Et l'algorithme de faire ça est donné par :

- Si la règle est de la forme « $A \to xB / B \in N^*, x \in V$ » on la copie dans la grammaire de sortie.
- Si la règle est de la forme « $A \to B\alpha / B \in N, \alpha \in (V \cup N)^*$ » on remplace le « B » jusqu'à obtenir « $A \to x\ \alpha / x \in V, \alpha \in (V \cup N)^*$ ».
- Si la règle est de la forme « $A \to x\ \alpha / x \in V, \alpha \in (V \cup N)^*$ » on remplace chaque terminal « x » de « α » par un non terminal « G_i » et on crée pour chacun une nouvelle règle « $G_i \to x$ ».

(03) Le logiciel :

1) La création « Création » :

1-1) Remise à zéro des grammaires « Nouv » :

Pour effacer les grammaires étant créée pendant l'exécution, il suffit d'appuyer sur ce bouton.

1-2) Créer un terminal « V »:

Il suffit de cliquer sur son bouton et écrire votre terminal et cliquer sur « OK » ; si vous cliquer sans insertion il n'ajoute aucune chose.

1-3) Créer un non terminal « N »:

Même méthode que les terminaux.

1-4) Créer ou modifier l'axiome « S »:

Il suffit de cliquer sur son bouton et à la partie des non terminaux, cliquer sur un d'eux.

1-5) Créer une nouvelle règle « R »:

Il suffit de cliquer sur son bouton et à la partie des non terminaux, cliquer sur un d'eux et la même chose pour les terminaux, et pour la construire :

- Passage vers la partie droite en cliquant sur le bouton droite de la souris ou en frappant la touche « ENTER ».
- Annulation par une frappe sur la touche « ESCAPE ».
- Pour faire entrer le ε à la partie droite il suffit de ne pas insérer aucune chose et frapper « ENTER » ou par un clique droite de la souris, on sortant de l'insertion.

2) Le travaille demandé « Fonction » :

2-1) Est-ce de type 2 ? « Type2 ? » :

- S'il n'existe pas de règle ou l'axiome, son rôle est d'informer l'usager.
- Sinon elle renvoie si la grammaire déjà insérée est de type 2.

1-2) Des fonctions sur la grammaire « FCT » :

Donne une grammaire G' réduite, propre et non récursive à gauche à partir de la grammaire G qu'on a inséré.

1-3) La forme normale de Chomsky « FNC » :

Donne une grammaire G1 sous forme de Chomsky à partir de la grammaire G' qu'on a lui transformé à partie de la règle originale.

1-4) La forme normale de Greibach « FNG » :

Donne une grammaire G2 sous forme de Greibach à partir de la grammaire G1 (celle sous forme de Chomsky).

PARTIE2 :
Le compilateur

1. **Définition du langage.**
2. **La grammaire du langage.**
3. **Quelques procédures explicatives.**
4. **La sémantique.**

(01) Définition du langage :

Ce langage est près dans sa syntaxe de la méthodologie d'écriture des algorithmes, il se compose de :

- Mots réservées : {algorithme , debut , fin , const , def , proc , vrai , faux , comme , @ , lire , lig , ecrire , si , alors , sinon , fsi , tq , faire , ftq , pour , vers , fpour , ent **,** reel **,** car **,** log **,** tab , et , ou , not , div , = , <> , < , > , <= , >= , + , - , * , /}.
- Les opérations : {+ , - , * , / , div}.
- Comparaison : { = , <> , < , > , <= , >= }.
- Logique : {et , ou , non , vrai , faux}
- Affectation : { = }.
- Les types : entier « ent », réel « reel », caractère « car », logique « log », tableau des entiers « tab(#) ».

On peut citer quelques contraintes sur ce langage :

1) Il y a une différence entre la majuscule et la minuscule donc « reel »≠ « Reel ».
2) On doit commencer avec le mot « algorithme » obligatoirement suivi de son identificateur.
3) A la fin de chaque instruction on doit revenir à la ligne.
4) Les mots « debut » et « fin » sont obligatoires.
5) L'identificateur doit être commencé par une lettre, suivi d'une collection des lettres et des chiffres.
6) Pas de « if » imbriquée.
7) On ne peut pas mettre un mot réservé « const » « def » sans mettre au moins un élément constante ou définition respectivement.
8) Le mot réservé « lig » qui suive « ecrire » ou « lire » veut dire « à la ligne ».

(02) La grammaire du langage :

La grammaire du langage décrit précédemment est donnée sous forme de Bakus et North (BNF) :

01) PROG ::= algorithme **IDENT** \n **BLOC**
02) IDENT ::= LETTRE {CAR}
03) BLOC ::= DECL debut \n **{INST}** fin
04) LETTRE ::= a |…| z | A | … | Z
05) CAR ::= LETTRE | CHIFFRE
06) DECL ::= [CONST][VAR][PROC]
07) INST ::= SIMPLE | COMPOSE
08) CHIFFRE ::= 0 |...| 9
09) CONST ::= const \n **CNST {CNST}**
10) CNST::= IDENT : **IDCONST** \n
11) VAR::= def \n **SUITVAR** \n **{SUITVAR** \n **}**
12) PROC::= proc **IDENT [**(**PARAM{**,**PARAM}**) **]** \n **BLOC** \n
13) SIMPLE ::= LECT|ECR|AFF|CALLPROC
14) COMPOSE ::= IF|WHILE|FOR
15) IDCONST::= NOMBRE | '**CAR{CAR}**' | vrai | faux
16) SUITVAR ::= IDENT{,**IDENT}** comme **TYPE**
17) PARAM ::= [@**] SUITVAR**
18) LECT ::= lire (**IDENT{**,**IDENT}**) **[**lig**]** \n
19) ECR ::= ecrire (**IDENT{**,**IDENT}**) **[**lig**]** \n
20) AFF ::= IDENT = **EXPR** \n
21) CALLPROC ::= IDENT [(**IDENT{**,**IDENT}**)**]** \n
22) IF ::= si **COND** alors \n **{SIMPLE} [** sinon \n **{SIMPLE}]** fsi \n
23) WHILE ::= tq **COND** faire \n **{SIMPLE}** ftq \n
24) FOR ::= pour **AFF** vers **ARITH** faire \n **{SIMPLE}** fpour \n
25) TYPE ::= ent | reel | car | log | tab (**NBR**)
26) EXPR ::= ARITH | LOGI
27) COND ::= CND [{(et | ou **) CND}]**
28) NOMBRE ::= [+ | - **] CHIFFRE {CHIFFRE} [**.**{CHIFFRE}]**
29) ARITH ::= MULT {(+ | - **) MULT}**
30) CND ::= (**ARITH OPER ARITH**) | (**LOGI [(** = | <> **) LOGI]**)
31) LOGI::= ET { ou **ET}**
32) MULT::= OPDA { (* | div | / **) OPDA }**
33) ET::= OPDB { et **OPDB}**
34) OPER::= = | <> | < | > | <= | >=
35) OPDA::= NOMBRE | IDENT | (**ARITH**)
36) OPDB::= [non **] (** **LOGI**) | vrai | faux | **[** non **] IDENT**

(03) Quelques procédures explicatives:

Voici la table des premiers de chaque non terminal :

Non terminal	Premiers
PROG	algorithme
IDENT	a…z ,A…Z
BLOC	const, def, proc
LETTRE	a…z ,A…Z
CAR	a…z ,A…Z, 0…9
DECL	const, def, proc
INST	lire, ecrire, a…z ,A…Z, si, pour, tq
CHIFFRE	0…9
CONST	const
CNST	a…z ,A…Z
VAR	def
PROC	proc
SIMPLE	lire, ecrire, a…z, A…Z
COMPOSE	si, tq, pour
IDCONST	‘ , vrai, faux, +, -, 0…9
SUITVAR	a…z ,A…Z
PARAM	@, a…z ,A…Z
LECT	lire
ECR	ecrire
AFF	a…z ,A…Z
CALLPROC	a…z ,A…Z
IF	si
WHILE	tq
FOR	pour
TYPE	ent, reel, car, log, tab
EXPR	a…z ,A…Z, 0…9, (, non, vrai, faux
COND	(
NOMBRE	+, -, 0…9
ARITH	a…z ,A…Z, 0…9, (
CND	(
LOGI	non, (, vrai, faux, a…z ,A…Z
MULT	a…z ,A…Z, 0…9, (
ET	non, (, vrai, faux, a…z ,A…Z
OPER	=, <>, <, >, <=, >=
OPDA	a…z ,A…Z, 0…9,(
OPDB	non, (, vrai, faux, a…z ,A…Z

Voici quelques procédures pour prendre une vision comment se fait l'analyse syntaxique :

1) PROG ::= algorithme IDENT \n BLOC

```
Procedure PROG
Début
    Si  Symbole_courant = 'algorithme' Alors
            Symbole_courant := chercher_symbole
            IDENT
            Si   Symbole_courant =retour_ligne  Alors
                    BLOC
            Sinon   Erreur 'Il faut un retourner à la ligne'
    Sinon   Erreur 'Pas de mot réservée « algorithme »'
Fin
```

2) IDENT ::= LETTRE {CAR}

```
Procedure IDENT
Début
    LETTRE
    Tant que  symbole_courant est dans {a..z, A..Z, 0..9} faire
            Symbole_courant := chercher_symbole
     FTQ
Fin
```

3) BLOC ::= DECL debut \n {INST} fin

```
Procedure BLOC
Début
    DECL
    Si   Symbole_courant = 'debut' Alors
          Symbole_courant := chercher_symbole
         Si  Symbole_courant =retour_ligne  Alors
              Symbole_courant := chercher_symbole
             Tant que Symbole_courant dans {ecrire, lire, si, pour, proc, tq} Faire
                    Symbole_courant := chercher_symbole
                    INST
             FinTq
             Si  Symbole_courant <> 'fin' Alors
                    Erreur 'la fin du programme ou procédure n'existe pas'
              Fsi
        Sinon  Erreur 'Il faut un retour ligne'
    Sinon Erreur 'pas de symbole « debut »'
Fin
```

04) LETTRE ::= a |…| z | A | … | Z
Procedure LETTRE
Début
Si symbole_courant est dans {a..z, A..Z} Alors
symbole_courant :=chercher_symbole
Sinon erreur 'un identificateur doit avoir aux moins une lettre au premier'
Fin

06) DECL ::= [CONST][VAR][PROC]
Procedure DECL
Début
CONST
VAR
PROC
Fin

07) INST ::= SIMPLE | COMPOSE
Procedure INST
Début
Si symbole_courant est dans {lire, ecrire, a…z, A…Z, si, tq, pour} Alors
SIMPLE
COMPOSE
Sinon erreur 'Il faux au moins une instruction'
Fin

08) CHIFFRE ::= 0 |...| 9
Procedure CHIFFRE
Début
Si symbole_courant est dans {0…9} Alors
symbole_courant :=chercher_symbole
Sinon erreur 'un nombre doit avoir aux moins un chiffre au premier'
Fin

09) CONST ::= const \n **CNST {CNST}**
Procedure CONST
Début
Si symbole_courant = 'const' Alors
symbole_courant :=chercher_symbole
CNST
Tant que symbole_courant est dans { a…z ,A…Z } faire
symbole_courant :=chercher_symbole
Ftq
Fsi
Fin

10) CNST::= IDENT : IDCONST \n

```
Procedure CNST
Début
    Si symbole_courant est dans { a…z ,A…Z } Alors
        IDENT
        Si symbole_courant = ':' Alors
            symbole_courant :=chercher_symbole
            IDCONST
            Si symbole_courant = retour_ligne Alors
                symbole_courant := chercher_symbole
            Sinon erreur 'Il faux un retour à la ligne'
        Sinon erreur 'absence de :'
    Sinon erreur 'Il faux un identificateur qui commence par une lettre'
Fin
```

11) VAR::= def \n SUITVAR \n {SUITVAR \n }

```
Procedure VAR
Début
    Si symbole_courant = 'def' Alors
        symbole_courant := chercher_symbole
        Si symbole_courant = retour_ligne Alors
            symbole_courant :=chercher_symbole
            SUITVAR
            Si symbole_courant = retour_ligne Alors
                Tant que symbole_courant est dans { a…z ,A…Z } faire
                    SUITVAR
                    Si symbole_courant = retour_ligne Alors
                        symbole_courant :=chercher_symbole
                    Sinon erreur 'Il faux un retour à la ligne'
                Ftq
            Sinon erreur 'Il faux un retour à la ligne'
        Sinon erreur 'Il faux un retour à la ligne'
    Fsi
Fin
```

13) SIMPLE ::= LECT|ECR|AFF|CALLPROC

```
Procedure SIMPLE
Début
    LECT
    ECR
    AFF
    CALLPROC
Fin
```

12) PROC::= proc **IDENT [**(**PARAM{**;**PARAM}**) **]** \n **BLOC** \n

```
Procedure PROC
Début
    Si symbole_courant = 'proc' Alors
        symbole_courant := chercher_symbole
        IDENT
        Si  symbole_courant = '(' Alors
            PARAM
            Tant que symbole_courant = ' ;' faire
                PARAM
            Ftq
            Si  symbole_courant = ')' Alors
                symbole_courant := chercher_symbole
            Sinon 'il faut fermer la parenthèse'
        Fsi
        Si  symbole_courant = retour_ligne Alors
            symbole_courant := chercher_symbole
        Sinon 'il faut un retour ligne'
        BLOC
        Si  symbole_courant = retour_ligne Alors
            symbole_courant := chercher_symbole
        Sinon 'il faut un retour ligne'
    Fsi
Fin
```

14) COMPOSE ::= IF|WHILE|FOR

```
Procedure COMPOSE
Début
    IF
    WHILE
    FOR
Fin
```

16) SUITVAR ::= IDENT{,**IDENT}** comme **TYPE**

```
Procedure SUITVAR
Début
    IDENT
    Tq symbole_courant = ',' faire
        IDENT
    Ftq
    Si symbole_courant = 'comme' Alors
        symbole_courant := chercher_symbole
        TYPE
    Sinon erreur 'Il n'y a pas de mot réservé « comme »'
Fin
```

(04) La sémantique:

On observe que « **IDENT** » doit être muni par les attributs suivants :

- **Nom :** c'est la chaîne de caractère au quelle on peut savoir cette identificateur.
- **Genre :** pour savoir s'il s'agit du nom du programme (valeur 0), nom d'une variable « définition » (valeur 1), nom d'une constante (valeur 2) ou d'un nom d'une procédure (valeur 3).
- **Type :** C'est un champ qui est un peut spécialisé pour les constantes et les variables ; il sert pour savoir ses types : (pas de type :0 , ent :1 , reel :2 , car :3 , log :4, tab :5).
- **Val :** La valeur est spécialement pour les constantes.
- **Long :** La longueur du tableau s'il existe. Et si le genre est procédure (valeur 3), il signifie le nombre des variables.

Ici on va voir quelques règles d'une sémantique préalable:

01)	IDENT.Genre :=0 ;
02)	$IDENT^0$.Nom:=$IDENT^1$.Nom • CAR.Val
	IDENT.Nom :=LETTRE.Val
04)	LETTRE.Val := a |…| z | A | … | Z
05)	CAR.Val := LETTRE.Val
	CAR.Val := CHIFFRE.Val
08)	CHIFFRE.Val := 0 |...| 9
10)	IDENT.Genre :=2 ; IDENT.Val :=IDCONST.Val ; IDENT.Type :=IDCONST.Type ;
12)	IDENT.Genre :=3 ; IDENT.Long :=nbr_parametres([(PARAM{;PARAM})]) ;
15)	IDCONST.Val=NOMBRE.Val ; IDCONST.Type := NOMBRE.Type
	IDCONST.Val= CAR.Val • {CAR}.Val; IDCONST.Type :=3 ;
	IDCONST.Val= true; IDCONST.Type :=4;
	IDCONST.Val= faux; IDCONST.Type :=4 ;
16)	SUITVAR.Type :=TYPE.Type ; {IDENT}.Type := TYPE.Type ;
20)	Si non permet(EXPR.Type,IDENT.Type) Alors erreur 'type erroné' Fsi
25)	TYPE.Type :=num(ent ,reel , car , log , tab (NBR)) ; Si TYPE.Type =5 Alors TYPE.Long :=NBR ; Fsi
26	EXPR.Type := ARITH.Type
	EXPR.Type := LOGI.Type

Note : Cette étude n'est pas complète et pas sûre, mais elle est juste pour montrer comment les choses sont déroulées pendant l'analyse sémantique.