Compilation

M. Patrou

2013 / 2014

Jérémy Hébert

Sommaire

Chapitre 1 : Introduction (début et fin 12/09/2013)

Présentation Générale

Définition compilateur

Un compilateur est un logiciel qui permet de lire un flot de données écrit dans un premier langage, appelé langage source, pour le traduire en un flot de sortie écrit dans un second langage, appelé le langage cible.

Un rôle secondaire important à signaler la présence d'erreurs dans le flot d'entrée.

Beaucoup de compilateur (du fait de la variété des langages sources et cibles). Tous sont conçus sur un même schéma qui distingue deux phases :

- La phase d'analyse (ou partie frontale du compilateur) dépend principalement du langage source. Il s'agit de reconstruire sous une forme intermédiaire la structure syntaxique du flot d'entrée.
- La phase de synthèse (ou partie finale du compilateur) dépend du langage cible. Elle a pour objet l'optimisation du code intermédiaire puis la production du code final.

Flot d'entrée **->** Analyseur lexical **->** Analyseur syntaxique **->** Analyser sémantique **->** générateur de code intermédiaire **->** optimiseur de code **->** générateur de code **->** Flot de sortie **(Schéma à refaire voir Alexis).**

Toutes ces étapes utilisent le gestionnaire de la table des symboles et le gestionnaire d'erreurs.

Schéma 1

Analyseur lexical

L'analyseur lexical lit séquentiellement le flot d'entrée et le découpe en lexèmes (ou unités lexicales) qui sont des suites de symboles qui ont une signification collective.

Exemple : position := initiale + vitesse x 60

Conduit au découpage :

- Un identificateur : position   ID
- Un symbole d'affection :=   AFFECT
- Un identificateur : initiale   ID
- L'opérateur +   '+'
- Un identificateur : vitesse   ID
- L'opérateur *   '*'
- Le nombre 60   NB

Chaque lexème appartient à une classe (identifiée par une constante symbolique). Certaines classes sont des singletons et peuvent être confondues avec l'unique lexème qu'elles contiennent. D'autres contiennent plusieurs éléments.

A la sortie de l'analyseur lexical on obtient : ID1 AFFECT ID2 '+' ID3 '*' NB

ID1, ID2 et ID3 sont, en fait, associés à des entrées dans la table des symboles.

Au passage : suppression des éléments non significatifs (blancs, tabulation, retour chariot, commentaires, …).

Les classes de lexèmes sont descriptibles par des langages rationnels (utilisation d'automates).

Analyseur syntaxique

L'analyseur syntaxique consiste à construire la structure grammaticale sous-jacente au flot de lexèmes.

On a besoin d'une description grammaticale (grammaire algébrique) du langage source.

Concrètement : construction d'un arbre.

Instruction -> ID AFFECT expression

Expression-> expression '+' expression

Expression-> expression '*' expression

Expression -> ID

Expression -> NB

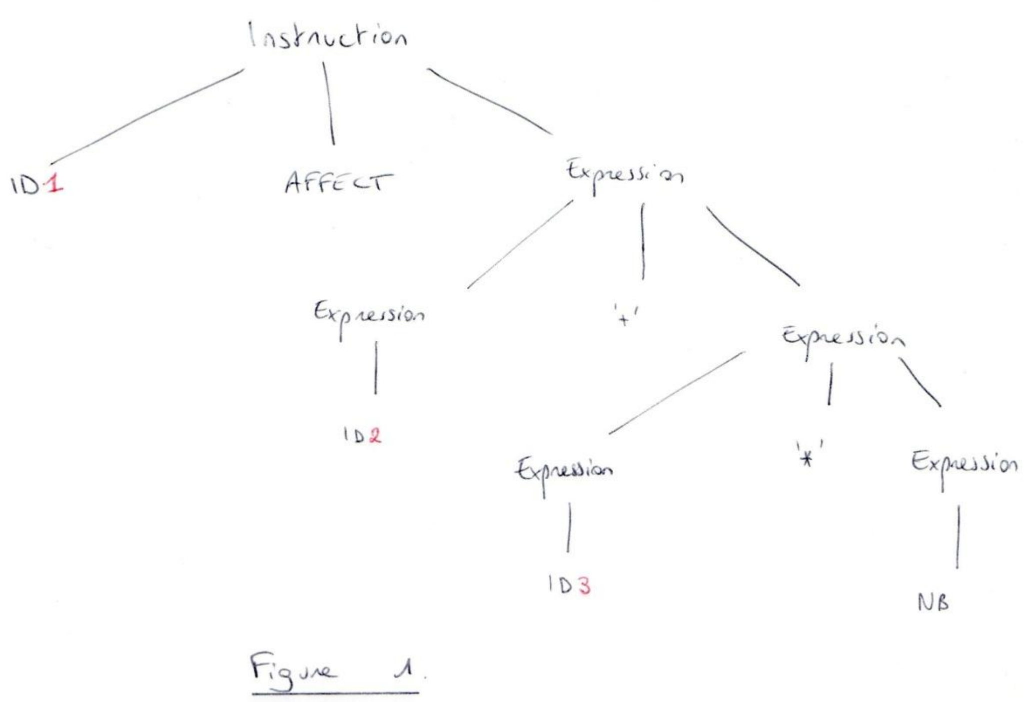


Figure 1 : arbre exemple

Analyseur sémantique

L'analyseur sémantique utilise cet arbre pour collecter diverses informations et contrôler la validité de certaines constructions (l'utilisation de réels pour indicer un tableau, une discordance sur le nombre et le type des arguments d'une fonction entre sa signature et un appel donné, …).

Une part importante de son rôle est la vérification du type des différents éléments apparaissant dans l'arbre (cette information est à stocker dans la table des symboles).

Sur notre exemple, le type des 3 identificateurs doit être vérifié pour assurer leur comptabilité. Une coercion de type (transtypage) est possible pour le nombre si ID3 est de type réel.

Générateur de code intermédiaire

Le générateur de code intermédiaire produit du code pour une machine abstraite.

```
tmp1 := entierVersReel(60)
tmp2 := Id3 * tmp1
tmp3 := Id2 + tmp2
id1 := tmp3
```

code à 3 adresses

Optimiseur de code

L'optimiseur de code recherche d'efficacité et de compacité.

```
tmp1 := id3 * 60.0
id1 := id2 + tmp1
```

Générateur de code

Le générateur de code permet :

- L'assignation des ressources mémoires
- L'affectation des registres

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

Remarques

- 6 étapes pas aussi distinctes que le laisse penser le schéma

  Par exemple, l'analyseur syntaxique pilote souvent la partie frontale du compilateur. Il veut construire un arbre interpelle l'analyse lexical en flot de ses besoins pour avancer dans sa construction. Il peut aussi stimuler l'analyseur sémantique quand il dispose d'une portion d'arbre « suffisante ».
- Le découpage entre étapes n'est pas immédiat.

Analyseur lexical	Analyseur syntaxique
Langage rationnel	Langage algébrique
automates	Grammaire algébrique

Outils d'aide à la conception de compilateur :

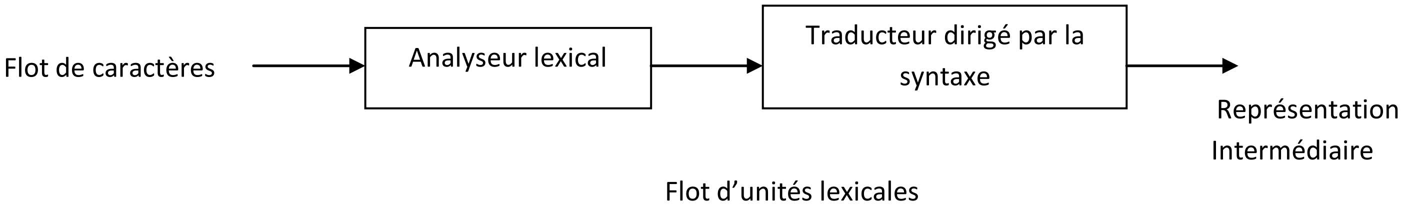
- Lex : logiciel C construisent automatiquement un analyseur lexical (également en C) à partir d'une description des lexèmes du langage source.
- Yacc (Yet Another Compiler Compiler) : construit un analyseur syntaxique (en C) à partir d'une grammaire.

Chapitre 2 : Un traducteur Infixe – post fixe (début 12/09/2013, fin 18/09/2013)

Expression infixe -> (traducteur) expression postfixe

Expression infixe : opérateurs binaires situés entre leurs opérandes.

Expression postfixe : opérateurs binaires situés après leurs opérandes.



On voit que l'analyseur syntaxique va mener les opérations. L'analyseur sémantique et le générateur du code intermédiaire lui sont tellement liés que ces 3 phases sont considérées comme n'en constituant plus qu'une.

Spécifions donc l'analyseur syntaxique. Il nous faut une grammaire décrivant le langage source :

E -> E + E | E – E | E * E | E / E | (E) | nb

Notations et définitions

- Une grammaire peut être vue comme un sujet de réécriture où chaque production de la grammaire de la forme A -> α peut être utilisée pour remplacer n'importe qu'elle occurrence de A par α.
  E $\rightarrow$E + E $\rightarrow$ nb + E $\rightarrow$ nb + nb
  Suite de 3 réécritures permettant de dériver E en nb + nb. Formellement, pour une grammaire G d'axiome S.
    - Une réécriture $\alpha A\beta \rightarrow \alpha\gamma\beta$ où $\rightarrow$ se lit « se dérive en une étape » est autorisée si A $\rightarrow$ γ est une production de G et α, β sont des chaînes quelconques.
    - Le symbole $\xrightarrow{*}$ signifie « se dérive en 0, 1 ou plusieurs étapes en».
    - Le symbole $\xrightarrow{+}$ signifie « se dérive en 1 ou plusieurs étapes en».
    - Une suite de réécritures est appelée une G-dérivation (ou plus simplement dérivation). Si $\alpha \xrightarrow{*} \beta$ alors on dit qu'il existe une dérivation de α en β (ou de β à partir de α). Et si α = S on dit simplement qu'il existe une dérivation de β.
    - Si S $\xrightarrow{*}$ α on dit qque α est une <u>proto-phrase</u> de G.
    - Une phrase de G est une proto-phrase de G ne contenant pas de symboles non-terminaux.
    - Une dérivation dans laquelle le non-terminal le plus à gauche (respectivement le plus à droite) est réécrit à chaque étape est appelée une <u>dérivation gauche</u> (resp. <u>droite</u>).
- Une grammaire peut aussi être vue comme la description d'un procédé de construction d'arbre. La racine de l'arbre est étiquetée par l'axiome et de chaque nœud étiquetée A est issu un ensemble de fils étiquetés par le symbole de la partie droite d'une A-production. Les feuilles sont étiquetées par des symboles terminaux. On parle d'arbre syntaxique ou d'arbre de dérivation.

  La suite de ses feuilles lues en ordre préfixe est le mot engendré par cet arbre.

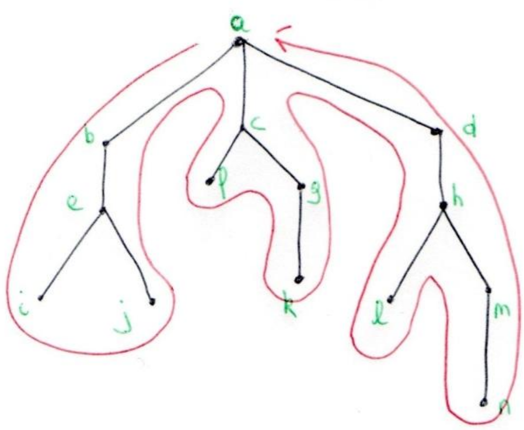


Figure 2

Figure 2 : schéma d'un arbre traitant le parcours préfixe.

- Pour une grammaire donnée, il y a bijection entre l'ensemble des arbres syntaxiques générés par celle grammaire et l'ensemble des dérivations gauches (ou droites) des phrases de cette grammaire.

2.1 Ambiguïté

Notre grammaire est ambiguë -> pb

Définition : une grammaire est ambigüe si elle permet de construire arbres syntaxiques distincts engendrant le même mot.

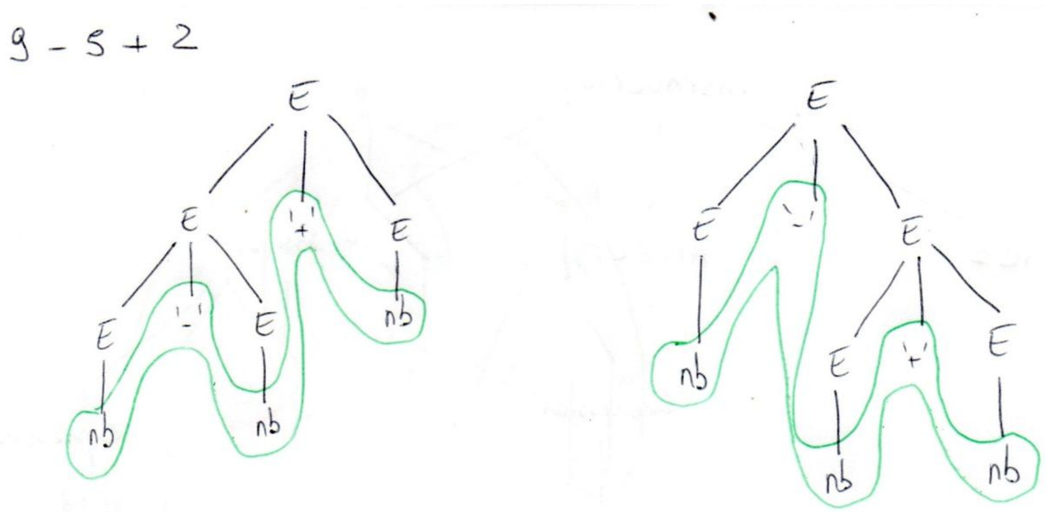


Figure 3 :

Figure 3

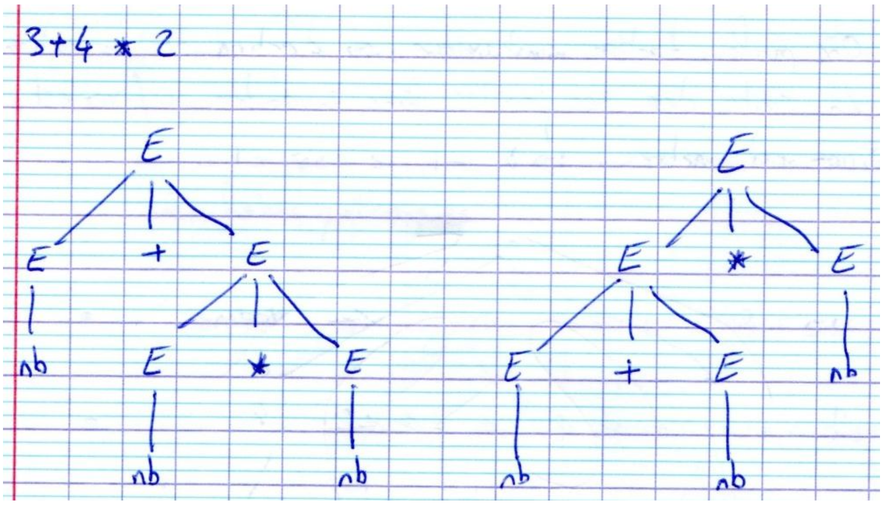


Figure 4

Un analyseur syntaxique ne peut être basé sur une grammaire ambiguë (il ne saura pas quel arbre construire).

Proposer une autre grammaire (non ambiguë)

Ici, c'est possible

En général : non -> il existe des langages inhéremment ambigus

2.2 Associativité et priorité

La grammaire est ambiguë car les propriétés d'associativité et de priorité des opérateurs ne sont pas prises en compte

<u>Priorité</u> : on règle le probleme en introduisant un symbole non – terminal supplémentaire pour chaque niveau de priorité souhaité :

E -> E + E | E – E | T

T -> T * T | T / T | F

F -> (E) | nb

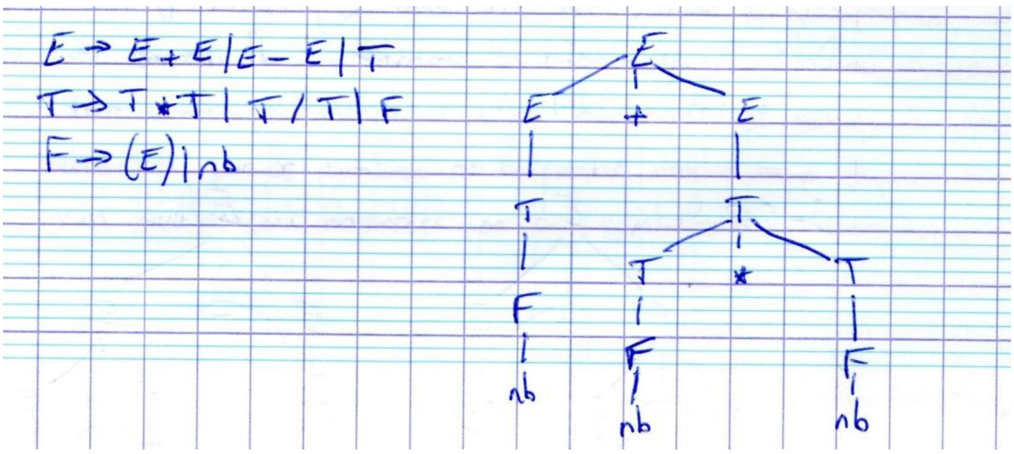


Figure 5

Associativité : conventionnellement, les 4 opérateurs arithmétiques sont associatifs à gauche.

8 – 4 – 1 => (8-4) – 1 = 3

⇨ 8 – (4 – 1) = 5

On gère cette question e cassant les symétries des parties droites de production

E -> E + T | E - T | T

T -> T * F | T / F |F

F -> (E) | nb

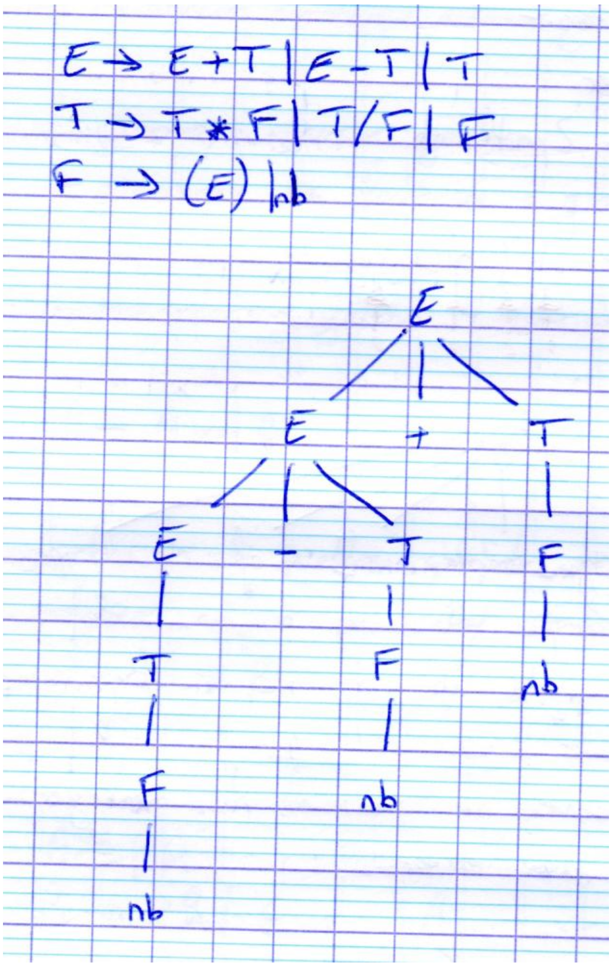


Figure 6

I -> id = I | id

Affectation associative à droite (en C)

A = b = c

2.3 Analyse syntaxique descendantes

Principe de ce type d'analyse :

Type -> type_simple

| ^id

|array[type_simple] of type

Type_simple -> integer

| char

| nb 2 points nb

Array [nb 2points nb] of integer

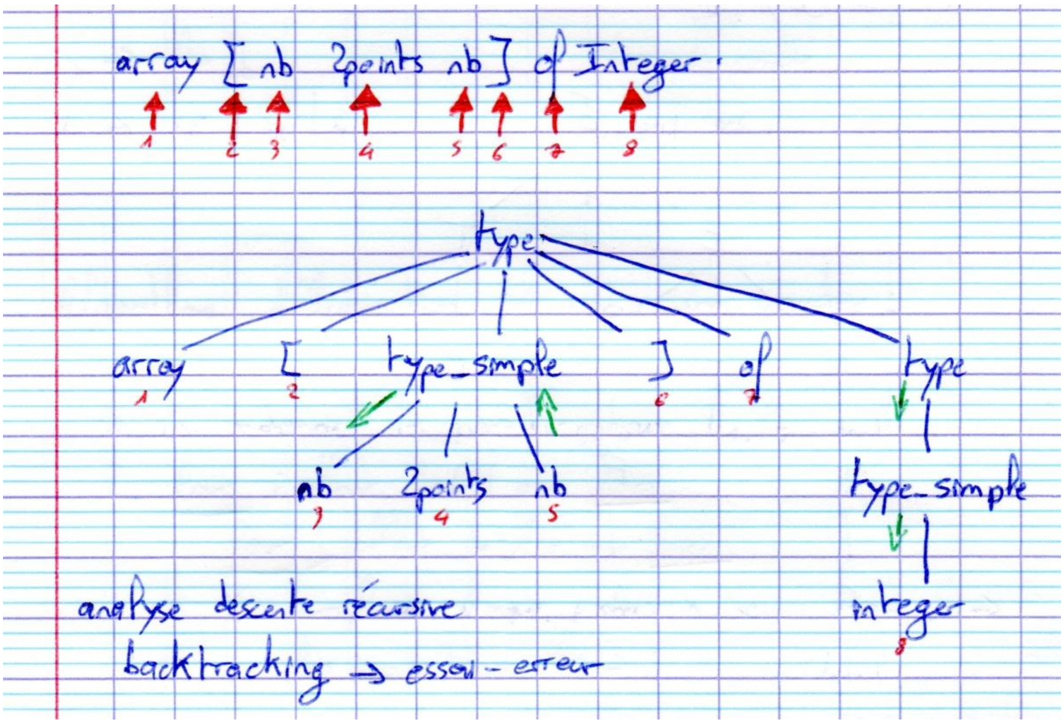


Figure 7

Backtracking -> essai – erreur.

L'arbre est construit en parallèle avec la lecture du flot d'entrée.

Le nœud courant (au départ : l'axiome à la racine) est dérivé en flot du prochain lexème lu sur l'entrée.

En général, plusieurs possibilités peuvent survenir. On procède par essais-erreur. Procédé naturel et intuitif, mais trop coûteuse dans le cas général (on envisagera, plus tard, des approches non récursives)

Pour éliminer le principe d'essais-erreurs :

PREMIER(type_simple) = { integer, char, nb}

PREMIER (^id) = { ^}

PREMIER(array[type_simple] of type) = {array}

2.4 Récursivité gauche et factorisation gauche

Pas d'analyseur syntaxique automatiquement produit en cas de :

- Grammaire récursive gauche.
- Grammaire non factorisée à gauche

Grammaire récursive gauche

Elle contient une production de la forme A -> Aα | β

Supposons que le prochain symbole de pré-vision appartienne à PREMIER(β). L'analyseur ne peut pas savoir combien de fois deriver A en Aα avant d'utiliser A -> β

Elimination de la récursive gauche :

A -> βR

R -> αR | ε

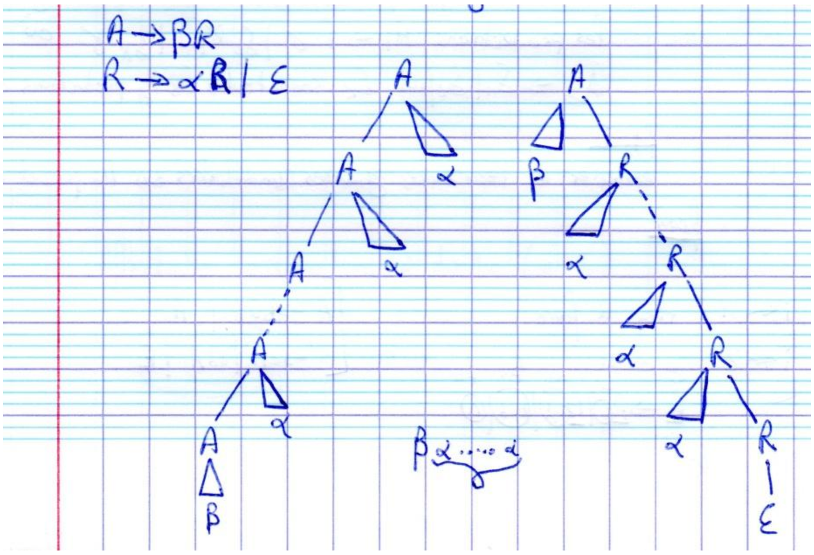


Figure 8

E -> E + T | E - T | T

T -> T * F | T / F |F

F -> (E) | nb

On applique la méthode :

E -> T E'

E' -> +T E' | - T E' | ε

T -> F T'

T' -> * F T' | / F T' | ε

F -> (E) | nb

On a :

S -> Aa | b

A -> Ac | Sd | c

Algorithme d'élimination des récursives gauches :

1 . On ordonne les non-terminaux A_1, A_2, .., A_n

2. Pour i allant de 1 à n faire

    Pour j allant de 1 à i – 1 faire

        Remplacer chaque production de la forme A_i -> $A_j\gamma$

        Les productions Ai -> δiγ | … | δkγ où Aj -> δ1 | δ2 | … | δk sont toutes les Aj productions courantes.

    Fin pour

    Eliminer les récursives gauches immédiates des Ai productions.

Fin pour

Légende :

A1, α2 : rouge

B : vert

I <- 1   rien à faire

I <- 2   j <-1

A -> Ac | Aad | bd | c

A -> bd A' | cA'

A' -> cA' | adA' | ε

Inverser boucle externe :

Pour tout k < i, s'il existe une production de la forme

Ak -> Ak'α

Alors k' > k

Inverser boucle interne :

S'il existe une production de la forme

Ai > Ak'α

Alors k4 > j – 1

J est un compteur.

Remarque : cet algorithme n'élimine les récursives gauches que si la grammaire initiale ne contient ni cycle (A $\overset{+}{\rightarrow}$ A), ni une production vide (sauf éventuellement S ->ε où S est l'axiome).

A -> A | β devient

A -> βA'

A' -> A' | ε

A -> Ca

B -> ε

C -> BA devient :

A -> Ca

B -> ε

C -> A

Il existe des algorithmes d'élimination des cycles et des ε-productions.

- Problème de l'absence de factorisation gauche : survient quand les parties droites de différentes A-production commençant par des symboles identiques.

Factorisation :

E -> E R | T

R -> + T | - T

2.5 Définition dirigée par la syntaxe

On va utiliser l'arbre produit par l'analyseur syntaxique en y ajoutant des infos supplémentaires.

On associe un ensemble d'attributs à chaque symbole grammatical. On définit des règles de calcul pour ces attributs (règles sémantiques).

$$\left.\begin{array}{l}\text{Grammaire}\\ \quad +\\ \text{Règles sémantiques}\end{array}\right\}\quad \text{définition dirigée par la syntaxe.}$$

Productions :	Règles sémantiques
E -> E_1 + T	E.t := E_1.t || T. t || '+'
E -> E_1 – T	E.t := E_1.t || T.t || '-'
E -> T	E.t := T.t
T -> T_1 * F	T.t := T_1.t || F.t || '*'
T -> T / F	T.t := T_1.t || F.t || '/'
T -> F	T.t := F.t
F -> (E)	F.t := E.t
F -> nb	F.t = nb vallex (valeur lexicale).

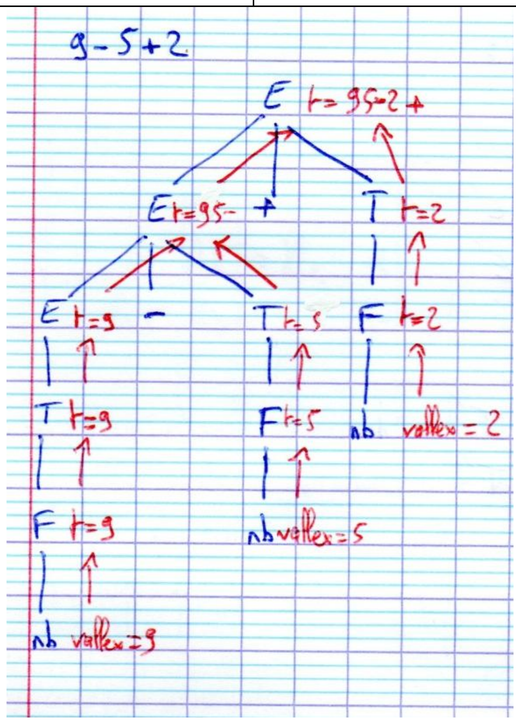


Figure 9

2.6 Schéma de traduction

On souhaite insérer des actions éxécutables par un traducteur dans l'arbre syntaxique. Par R -> + T { imprimer ('+')} R

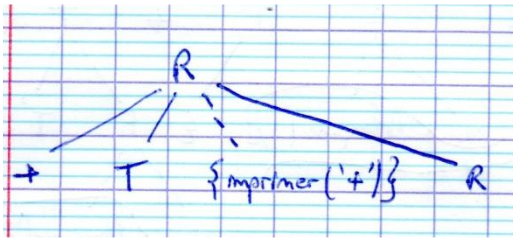


Figure 10

Le schéma de traduction associé à la définition dirigée par la syntaxe précédente :

E -> E + T { imprimer ('+')}

E -> E - T { imprimer ('-')}

E -> T

T -> T * F { imprimer ('*')}

T -> T / F { imprimer ('/')}

T -> F

F -> (E)

F -> nb { imprimer (nb.vallex)}

Sans la récursive gauche :

E -> T E'

E' -> +T { imprimer ('+')} E'

E' -> ε

T -> F T'

T' -> * F { imprimer ('*')} T'

T' -> / F { imprimer ('/')} T'

T' -> ε

F -> (E)

F -> nb {imprimer(nb.vallex)}

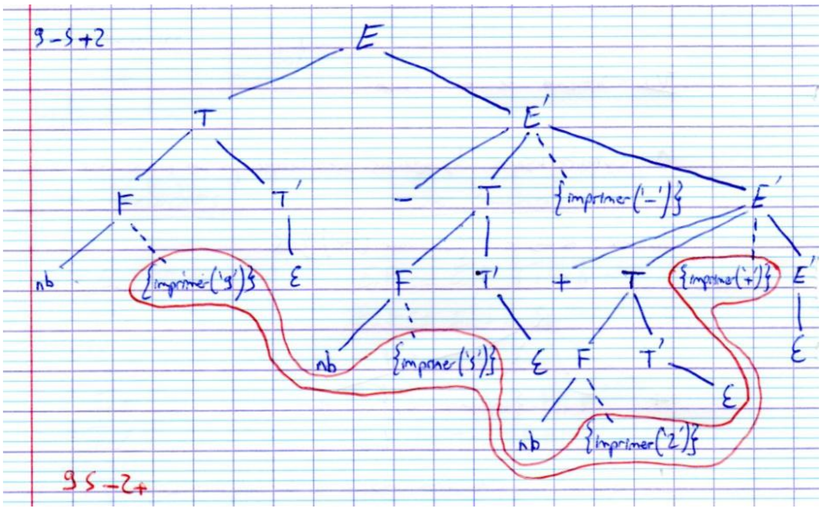


Figure 11

2.7 Une version C du traducteur

Voir Poly.

2.8 L'analyseur lexical et la table des symboles 18 / 09

L'analyseur lexical doit éliminer les caractères non significatifs et reconnaitre :

- Des entiers
- Les opérateurs + - * /
- Les parenthèses et les points virgule
- La fin de fichier
-

Solution : switch + quelques getChar().
Quand un entier est reconnu une constante symbolique (NB) est renvoyée à l'analyseur syntaxique accompagnée de la valeur entière (via une variable globale).
Dans les autres cas on retourne simplement le caractère repéré.

Quelques ingrédients supplémentaires :

- Des identificateurs,
- Des mots clés dans le langage source (DIV et MOD)

Quand un identificateur est détecté, l'analyseur lexical le recherche dans la table des symboles (et l'y insère s'il ne le trouve pas). Son indice dans la table fait office de valeur lexicale à retourner à l'analyseur syntaxique, accompagnée de la constante ID.
Les mots clés sont traités à l'identique, mais sont insérés dans la table à l'initialisation du traducteur.

Table des symboles : tableau d'enregistrement à 2 champs :

- Classe du lexème
- Représentation (string) du lexème.

Traducteur peu robuste, peu ergonomique
3_+_toto_– 2
Problème si l'utilisateur oublie ce blanc (le scanf « mange » tout jusqu'à rencontrer un blanc ou un retour chariot).
2_+_2 ;
x_*_3 ;
; => erreur (les expressions vides ne sont pas prévues par la grammaire).

Chapitre 3 : Analyse lexicale (début et fin : 18/09/2013)

Rôle principal : transformer un flot de caractère en un flot de lexèmes.

Services supplémentaires :

- Elimination des caractères non significatifs
- Décompte des lignes (pour permettre la localisation d'erreur syntaxique).
- Stockage des lexèmes dans la table des symboles
- Signalisation d'erreurs.

En pratique, c'est un sous-programme du service de l'analyseur syntaxique mais une conception séparée accroît la modularité de compilation et présente plusieurs avantages :

- Simplicité (intégrer la spécification des unités lexicales dans la grammaire l'alourdit considérablement).
- Efficacité (un module spécialisé est plus facile à optimiser)
- Portabilité (les spécificités lexicales sont confinées à l'analyseur lexical).

Les unités lexicales sont, en principe, spécifiables par des automates.

Exemples :

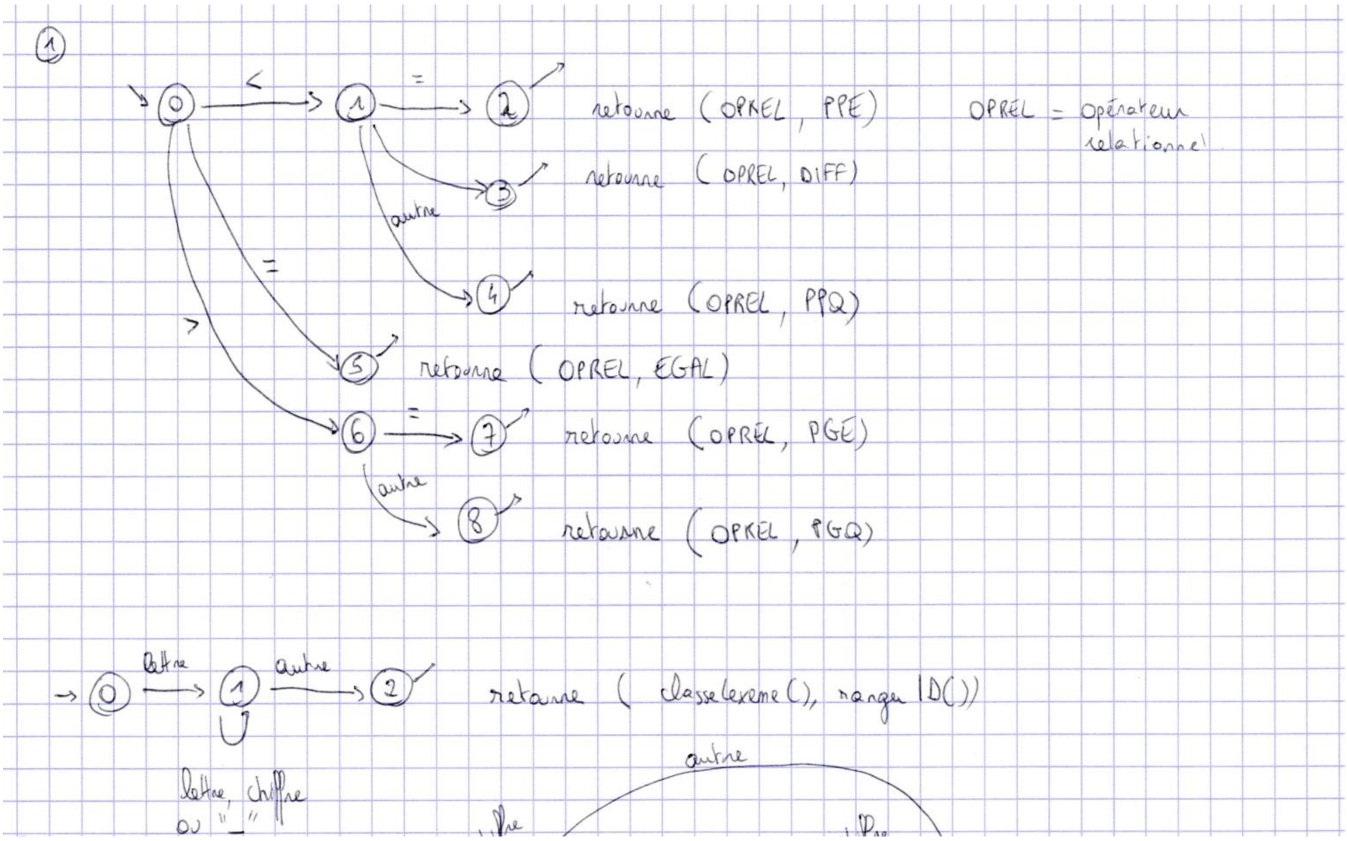
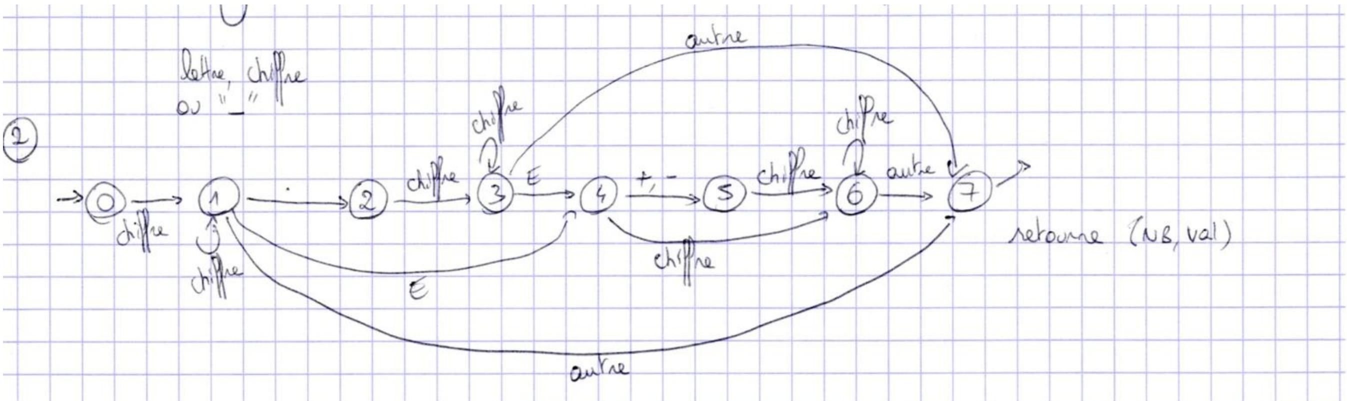


Figure 1

**/ ! ** à « autre » : penser à renvoyer sur le flot de caractère le symbole lu (il appartient probablement au prochain lexème).



Ces automates sont à regrouper pour n'en faire plus qu'un, qu'il faut déterminer pour pouvoir l'utiliser comme support de détection des lexèmes.

L'analyseur lexical parcourt cet automate au fil des caractères lus sur l'entrée pour détecter les lexèmes.

Technique d'optimisation

- De la lecture
- De stockage (compression)

De l'automate

Technique de gestion d'erreurs.

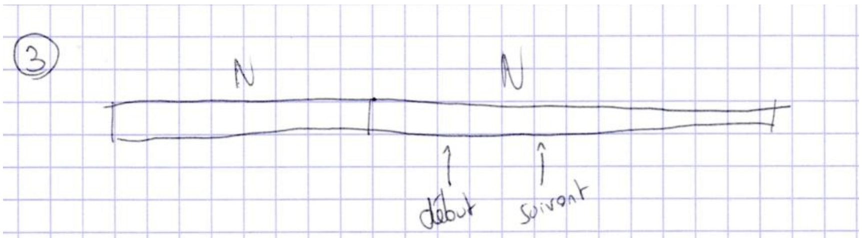
Remarque : les lexèmes repérés doivent être les plus longs possibles

3.14

_._X

___

Remarque : ce module est le plus gourmand en temps, à cause des Entrées/sorties (à optimiser).



Algorithme de gestion des pointeurs :

<u>Si</u> avant est à la fin de la 1ère moitié <u>alors</u>

  Changer la 2nde moitié

  Avant <- avant + 1

<u>Sinon</u>

  <u>Si</u> avant est à la fin de la 2nde moitié <u>alors</u>

    Changer la 1ère moitié

    Déplacer avant au début de la 1ère moitié

  <u>Sinon</u>

    Avant <- avant + 1

  <u>Fin si</u>

<u>Fin si</u>

Chapitre 4 : Analyse syntaxique

(début 18/09/2013, 25/09/2013, 02/10/2013, 16/10/2013 fin : 16/10/2013)

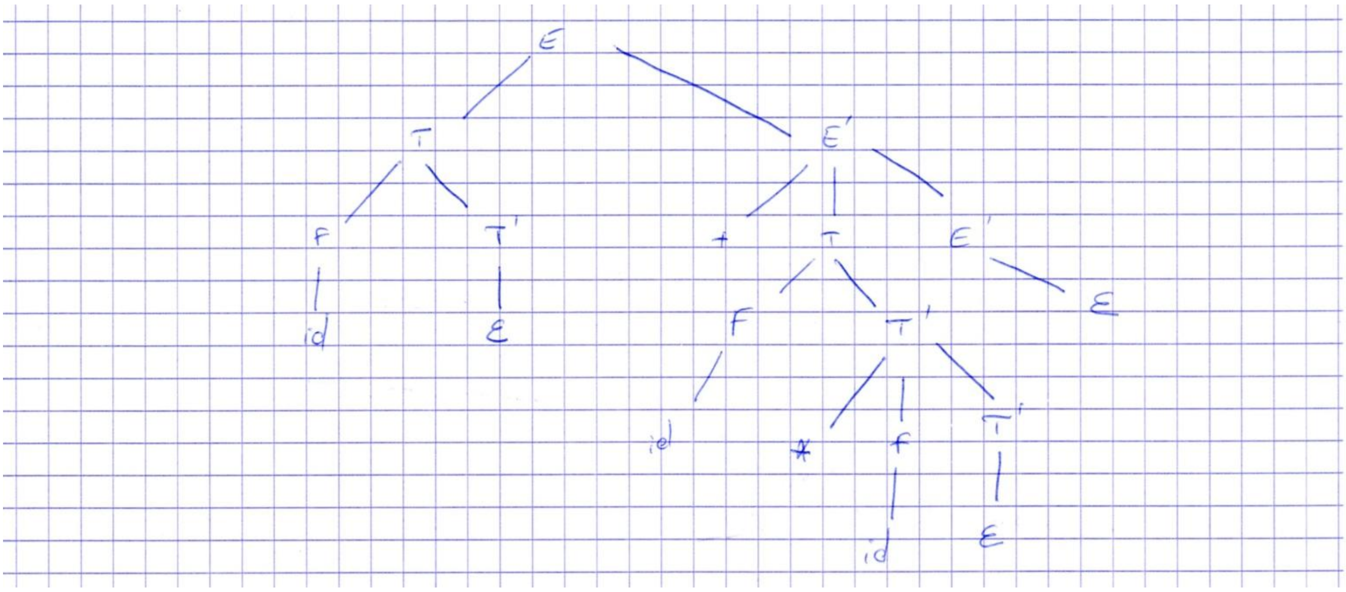
Deux grandes catégories :

- Analyseurs descendants (plus faciles à appréhender)
- Analyseurs ascendants (plus puissants).

4.1 Analyse descendante

Un analyseur descendant fonction selon le mécanisme du traducteur (chapitre 2).

On en présente ici une version non récursive. Plutôt que de générer une pile d'appels récursifs correspondant aux nœuds de l'arbre syntaxique en cours de construction et parcourus en ordre préfixe nous allons gérer explicitement la pile des symboles étiquetant ces nœuds.



- Tampon d'entrée : flot d'unité lexicales (terminé par un symbole spécial : $)
- Pile de symboles grammaticaux : permet de gérer le parcours en profondeur
- Table d'analyse : contient les infos qi permettent à l'analyseur de choisir la production à utiliser à chaque étape selon l'unité lexicale.
- Sortie : l'analyseur peut produire une sortie à chaque étape. Nous nous contenterons de produire la production utilisée.

Ex :

E -> T E'
E' -> + T E' | ε
T -> F T'
T' -> * F T' | ε
F -> (E) | id

Algo : analyse descendante non récursive

Données : une chaîne **w** et une table d'analyse **M** pour une grammaire **G**.

Résultat : une dérivation gauche pour le mot **w** si w € L(G), une indication d'erreur sinon.

Initialisation :

- $$ dans la pile
- w$ dans la pile

Positionner un pointeur source ps sur le 1er symbole de w$.

Répéter :

```
    Soit X le symbole en sommet de pile et a le symbole repéré par ps.
    Si X est un terminal ou $ alors
        Si X = a alors
            défiler(X)
            avancer(ps)
        Sinon
            Erreur()
        Fin si
    Sinon
        Si M[X, a] = X -> Y1Y2..Yn alors
            Défiler(X)
            Empiler(Yk, Yk-1, …, Y1) (avec Y1 au sommet)
            Emettre(X -> Y1Y2…Yk)
        Sinon
            Erreur()
        Finsi
Jusqu'à X = $
```

Id + id * id

(Associé au schéma 1 du 25/09)

Pile	**Entrée**	**Sortie**
$E	id + id * id$	
$E'T	id + id * id$	E -> TE'
$E'T'F	id + id * id$	T -> FT'
$E'T' id	id + id * id$	F -> id
$E'T'	+ id * id$	
$E'	+ id * id$	T' -> ε
$E'T+	id * id$	E' -> TE'
$E'T	id * id$	
$E'T'F	id * id$	T -> FT'
$E'T' id	Id * id$	F -> id
$E'T	* id$	
$E'T'F*	*id$	T' -> *FT'
$E'T'F	id$	
$E'T' id	Id$	F -> id
$E'T'	$	
$E'	$	T' -> ε
$	$	E' -> ε
/	/	

4.2 Grammaire LL(1)

Pour remplir la table d'analyse, on s'appuie sur 2 fonctions associées à la grammaire : PREMIER et SUIVANT.
Etant donné une grammaire G = <N, T, P, S> et une production A -> α appartenant à P, la fonction PREMIER doit permettre de dire s'il est pertinent ou non de dériver A en α à la vue d'un symbole A.
Il s'agit donc d'une fonction définie de P vers P(T). que nous construisons par extension successives du domaine de définition :

1/ PREMIER : N U T -> P(T)
2/ PREMIER : (N U T)* -> P(T U {ε})
3/ PREMIER : P -> P(T U {ε})

1. LA 1ère version à tout symbole X de la grammaire l'ensemble des symboles terminaux qui se trouvent en tête d'au moins une phrase dérivée de X.
   Fonction préliminaire (déterminant les symboles effaçables de G).
   EFFACABLE <- Ø
   Répéter
   Pour toute production A -> α faire
   Si tous les symboles de α sont déjà dans EFFACABLE ou si α = ε alors
   Mettre A dans EFFACABLE
   Finsi
   Fin pour
   Jusqu'à stabilisation de EFFACABLE

Algorithme de point fixe, qui termine car N est fini.

Pour tout symbole X de la grammaire faire
  Si X est terminal alors
    PREMIER(X) = { X}
  Sinon
    PREMIER(X) = Ø
  Fin si
Fin pour
Répéter
  Pour toute production A -> X1…Xk (avec k ≥ 1) faire
    Ajouter PREMIER(Xi) à PREMIER(A) pour tout i tel que {X1, …, Xi-1} ⊆ EFFACABLE
  Fin pour
Jusqu'à stabilisation de tous les ensembles PREMIER.

Algorithme de point fixe qui termine car T est fini.

2. PREMIER(ε) = {ε}

   PREMIER(Xα) = PREMIER(version 1) (X) ∪ Q

   Où Q = PREMIER (α) si X est Effaçable

   Ø sinon

3. Expression triviale :

   PREMIER (A -> α) = PREMIER (version 2) (α)

Exemple :

E -> T E'

E' -> + T E' | ε

T -> F T'

T' -> * F T' | ε

F -> (E) | id

⇨ EFFACABLE = {E', T'}

PREMIER(+) = {+}

PREMIER(*) = {*}

PREMIER(() = {(}

PREMIER ()) = {)}

PREMIER (id) = {id}

PREMIER(E) = { (, id}

PREMIER (E') = {+}

PREMIER (T) = {(, id}

PREMIER (T') = {*}

PREMIER (F) = {(, id}

PREMIER (E -> T E') = { (, id}

PREMIER (E' -> + T E') = { +}

PREMIER (E' -> ε) = { ε }

PREMIER (T -> F T') = { (, id}

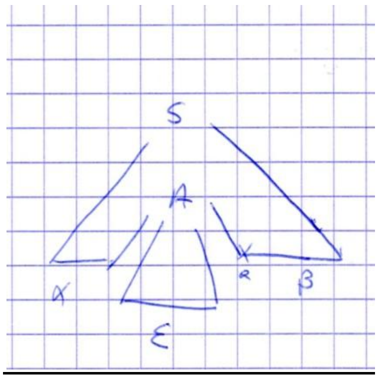
PREMIER (T’ -> * F T’) = { (, id}
PREMIER (T’ -> ε) = { ε }
PREMIER (F -> (E)) = { (}
PREMIER (F -> id) = { id }

S -> A |BD
A -> ε | aB | aA
B -> bB | ADe
D -> AA | dD | dS
EFFACABLE = {A, D, S}

PREMIER(S) = {a, b, e, d}
PREMIER (A) = {a}
PREMIER (B) = {b, a, e, d}
PREMIER (D) = {a, d}

La fonction SUIVANT détermine pour chaque symbole non-terminal A, l’ensemble des symboles terminaux susceptibles de suivre A dans une proto-phrase issue de l’axiome.

SUIVANT(A) est l’ensemble des symboles a pour lesquels il existe une dérivation de la forme S $\overset{*}{\rightarrow}$ α A a β



Pour tout non-terminal A de la grammaire faire
    SUIVANT(A) = ∅
Fin pour
SUIVANT (S) = { S }
Pour toute production A -> αBβ faire
    Ajouter PREMIER(β) \ {ε} à SUIVANT(B)
Fin pour
Repéter
    Pour toute production A -> αBβ avec ε ∈ PREMIER(β) faire
        Ajouter SUIVANT(A) à SUIVANT(B)
    Fin pour
Jusqu’à stabilisation de tous les ensembles.

SUIVANT (E) = { $,)
SUIVANT (E’) = {$,)
SUIVANT (T) = {+, $,)
SUIVANT (T’) = {+, $,)
SUIVANT (F) = {*, +, $,)

02/10/2013

S -> A | BD
A -> ε | aB | aA
B -> bB | Ade
D -> AA | dD | dS

EFFACABLE = { S, A, D}

PREMIER(S) = { a, b, d, e}
PREMIER(A) = {a}
PREMIER(B) = {b, a, e, d}
PREMIER(D)= {a, d}

SUIVANT(S)={$, e}
SUIVANT(A) = {a, d, e, $}
SUIVANT(B) = {a, d, $, e}
SUIVANT(D) = {e, $}
Algo : Remplissage d'une table d'analyse descendante
Donnée : Une grammaire G
Résultat : une table d'analyse M pour G.

Pour chaque production A -> α de G faire
    Pour chaque terminal a de PREMIER(α) faire
        Ajouter A -> α à M[A, a]
    Fin pour
    Si ε ∈ PREMIER(α) alors
        Pour chaque symbole b de SUIVANT(A) faire
            Ajouter A -> α à M[A, b]
        Fin pour
    Fin si
Fin pour

EFFACABLE + {E', T'}

E -> T E'	PREMIER(E) = { (, id}	SUIVANT (E) = { $,)
E' -> + TE' | ε	PREMIER (E') = {+}	SUIVANT (E') = {$,)
T -> F T'	PREMIER (T) = {(, id}	SUIVANT (T) = {+, $,)
T' -> *FT' | ε	PREMIER (T') = {*}	SUIVANT (T') = {+, $,)
F -> (E) | id	PREMIER (F) = {(, id}	SUIVANT (F) = {*, +, $,)

	id	+	*	(	)	$
E	E -> TE'			E -> TE'		
E'		E' -> TE'			E' -> ε	E' -> ε
T	T -> FT'			T -> FT'		
T'		T' -> ε	T'->*FT'		T' -> ε	T' -> ε
F	F -> id			F -> (E)		

Une telle table n'est utilisable pour réaliser une analyse descendante que si elle ne contient pas d'entrée multiples.
Une grammaire :

- Ambigüe
- Récursive à gauche
- Ou non factorisée à gauche

produit toujours une entrée multiple.

Définition : une grammaire algébrique dont la table d'analyse descendante ne contient pas d'entrée multiple est dite LL(1). (L pour Left, le 1er left est pour entrée lue de gauche à droite, le deuxième est la position d'une dérivation gauche et le 1 est le nombre de symbole(s) de prévision).

Exemple classique de grammaire non LL(1).

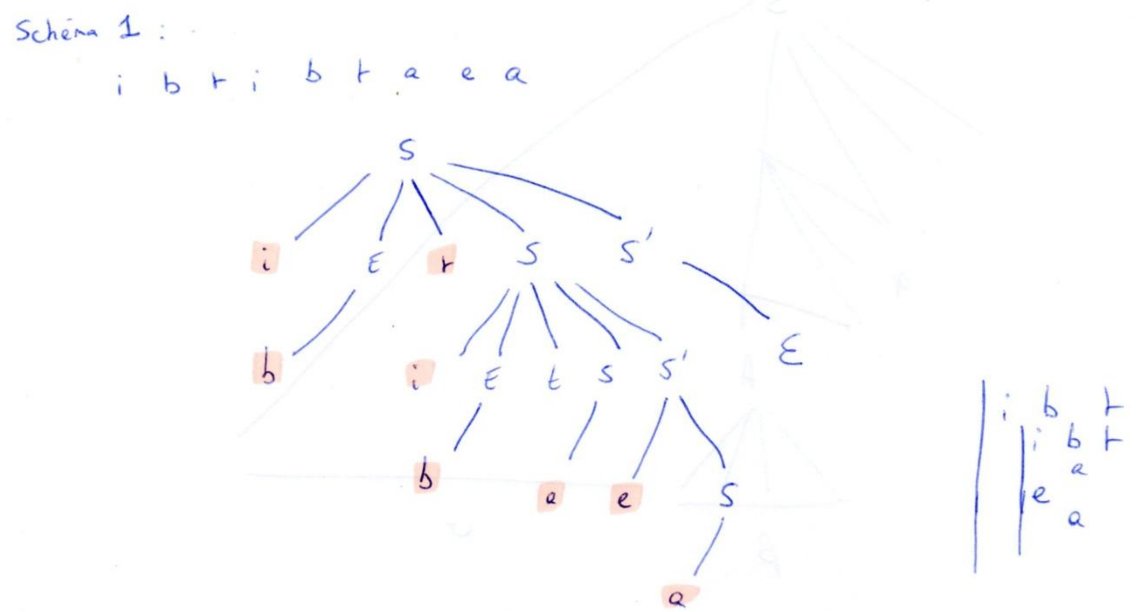
$S \rightarrow i\ E\ t\ S\ S' \mid a$

$S' \rightarrow e\ S \mid \varepsilon$

$E \rightarrow b$

PREMIER (S) = { i, a} PREMIER(S') = {e} PREMIER(E) = { b} EFFACABLE = {S'}	SUIVANT(S) = {$, e} SUIVANT(S') = {$, e} SUIVANT(E) = {t}

	a	b	e	i	t	$
S	$S \rightarrow a$		$S \rightarrow i\ E\ t\ S\ S'$			
S'		$S' \rightarrow e\ S$ $S' \rightarrow \varepsilon$				$S' \rightarrow \varepsilon$
E		$E \rightarrow b$				



Solutions possibles :

- Chercher une grammaire équivalente non ambigue (il en eiste une, ici).
- Forcer le conflit en retenant S' -> eS quand on doit dériver S' à la vue d'un e (revient à choisir d'associer chaque else au then le plus proche, qui précède et qui n'est pas déjà associé à un else).
    - Possible ici, mais pas en général.

4.3 Analyse ascendante

Construction d'un arbre syntaxique des feuilles vers la raine. On va produire une dérivation droite inversée du texte source.

La classe des grammaires analysables sera notée LR. L'analyseur ascendante (encore appelée analyseur décalage – réduction) consiste en une suite d'étapes permettant de réduire une chaîne initiale en l'axiome de la grammaire. A chaque étape un facteur correspondant à la partie droite d'une production est réduit en le symbole à gauche de cette production.

$S \rightarrow aAE$

$A \rightarrow aA \mid Bd \mid aB$

$B \rightarrow b$

$E \rightarrow e$

$\omega = a\ a\ \underline{b}\ d\ c \xrightarrow{réduction} a\ a\ \underline{B\ d}\ e \xrightarrow{réduction} a\ \underline{a\ A}\ e \xrightarrow{réduction} a\ \underline{A\ e} \xrightarrow{réduction} a\ \underline{A\ E} \xrightarrow{réduction} S$

On peut faire de mauvais choix.

$\omega = a\ a\ \underline{b}\ d\ c \xrightarrow{réduction} a\ \underline{a\ B}\ d\ e \xrightarrow{réduction} \underline{a\ A}\ d\ e \xrightarrow{réduction} A\ d\ E \xrightarrow{réduction} A\ d\ E$

$\omega = a\ a\ \underline{b}\ d\ c \xrightarrow{réduction} a\ a\ \underline{B\ d}\ e \xrightarrow{réduction} a\ \underline{a\ A}\ e \xrightarrow{réduction} a\ \underline{A}\ e \xrightarrow{réduction} A\ \underline{e} \xrightarrow{réduction} A\ E$

Définition : on appelle manche d'une proto-phrase droite γ toute production A -> β associée à une position γ où β peut être trouvée et remplacée par A pour produire la proto-phrase droite précédente dans une dérivation droite de γ.

$S \overset{*}{\rightarrow} (d)\ \alpha A v \underset{d}{\rightarrow} \alpha\beta v$

La production A -> β dans la position suivant α est un manche de αβv.

Remarque :

i) V ne contient que des symboles terminaux.

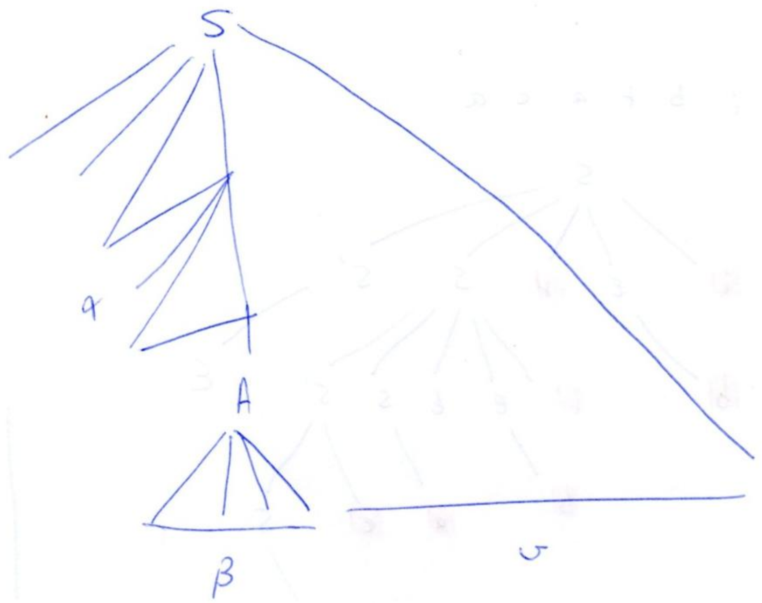
ii) Une proto-phrase peut posséder plusieurs manches auquel cas il lui correspond plusieurs dérivations droite, ce qui signifie que la grammaire est ambigüe.

Ex : E -> E + E | E * E | (E) | id

$\omega = id + id * id$

$E \rightarrow \underline{E + E}$		$E \rightarrow \underline{E * E}$
$\rightarrow E + \underline{E * E}$		$\rightarrow E * \underline{id}$
$\rightarrow E + E * \underline{id}$	<- 2 manches différents ->	$\rightarrow \underline{E + E} * id$
$\rightarrow E + \underline{id} * id$		$\rightarrow E + \underline{id} * id$
$\rightarrow \underline{id} + id * id$		$\rightarrow \underline{id} + id * id$

Représentation schématique d'un manche :



A est le symbole non-terminal le plus à droite de αAv.
La prochaine étape consiste à réduire β en A (élagage de manche).

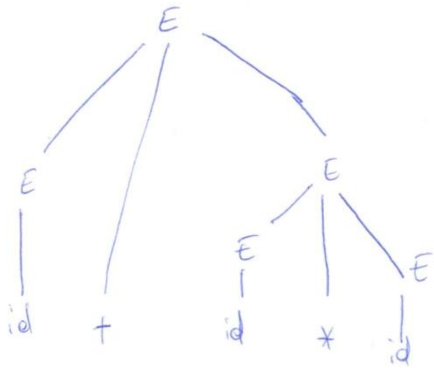
Etape d'une analyse ascendante :

1) On dispose la phrase à analyser dans un tampon d'entrée.
2) On décale un à un les symboles de tampon vers une pile.
3) Chaque fois qu'un manche apparaît au sommet de la pile, on la réduit.
4) On s'arrête lorsque le tampon est vide et que la pile contient l'axiome (succès) ou lorsque les symboles grammaticaux situés en sommet de pile n'ont aucune chance de contribuer à l'apparition d'un manche (erreur).

09/10/2013

E → E + E | E * E | (E) | id
$\omega = id + id * id$

Pile	Entrée	Actions
$	id + id * id$	décale
$id	+ id * id$	réduire par E → id
$E	+ id * id$	décaler
$E +	id * id$	décaler
$E + id	* id$	réduire par E → id
$E + E	id$	décaler
$E + E *	id$	décaler
$E + E * id	$	réduire par E → id
$E + E * E	$	réduire par E → E * E
$E + E	$	réduire par E → E + E
$E	$	accepter



E → E + E → E + E * E → E + E * id → E + id * id → id + id * id

$\omega = (id +) * id$

<u>Pile</u>	<u>Entrée</u>	<u>Actions</u>
\$	(id+)*id\$	décaler
\$(	id+)*id\$	décaler
\$(id	+)*id\$	réduire E → id
\$(E	+)*id\$	décaler
\$(E+	)*id\$	erreur

Impossible de réduire, pas de manche en sommet de pile.

Inutile de décaler (E+) n'est pas un préfixe viable.

<u>Définition</u> : Un préfixe viable est un préfixe d'une proto-phrase droite qui ne s'étend pas au-delà de l'extrémité droite du manche le plus à droite de cette proto-phrase.

<u>Remarque</u> : Ce sont exactement les préfixes qui peuvent apparaître sur la pile d'un analyseur par décalage-réduction.

<u>En théorie</u> : chaque étape d'une analyse ascendante consiste à repérer un manche pour la réduire.

<u>En pratique</u> : ce n'est pas toujours possible automatiquement. Il est impossible de repérer un manche de façon sûre pour une grammaire quelconque quand on ne possède qu'un préfixe de proto-phrase droite.

Seule une sous-classe des grammaires algébriques pourra être traitée.

Quand on traitera une grammaire n'appartenant pas à cette classe :

- Conflits réduire / décaler
- Conflits réduire / réduire

Technique LR(k)

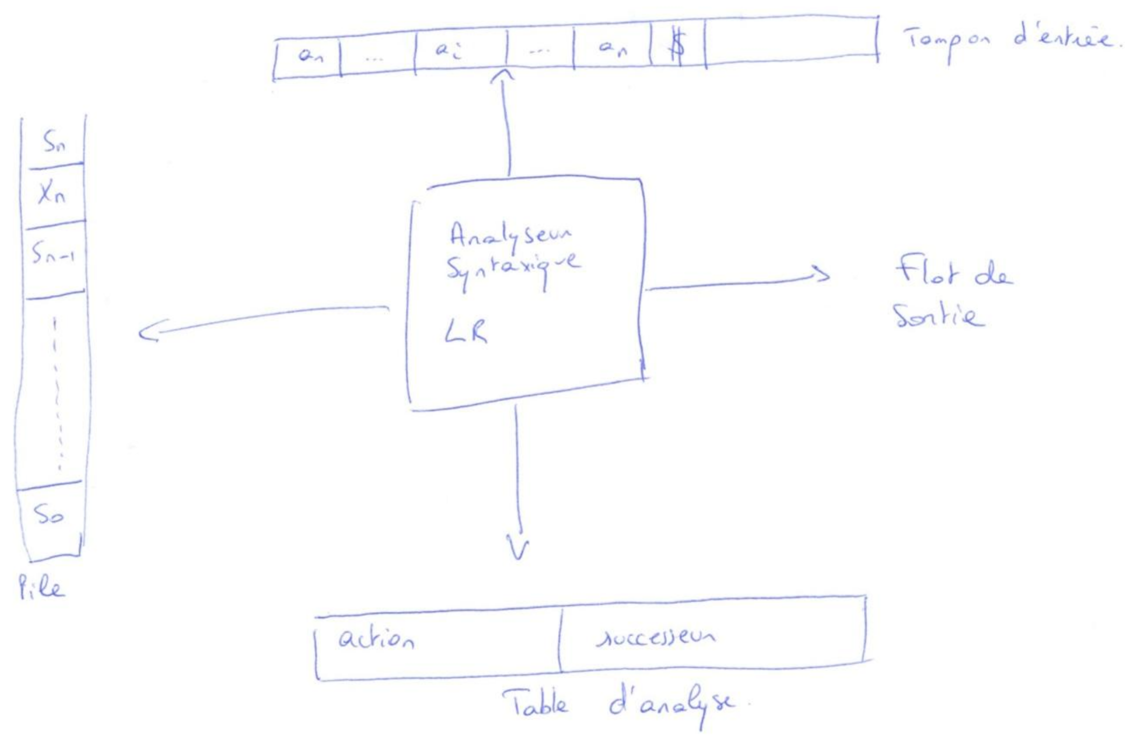
L → Left (Flot de lexèmes ratés de gauche à droite)

R → Right(Construction d'une dérivation droite).

k -> nombre de symboles de pré-vision (k = 1, ici).

a1	…	ai	…	an	\$	

Tampon d'entrée



Les unités sont lues unes à unes pour produire en sortie la dérivation droite de la chaîne d'entrée à l'aide d'une pile. Dans celle-ci, sont intercalés les états d'un automate entre les symboles grammaticaux. Cet automate reconnaît les préfixes viables de la grammaire (pour toute grammaire algébrique, le langage de ses préfixes viables est rationnel). En pratique, seuls les états de l'automate sont utiles dans la pile. On y fait apparaitre les symboles grammaticaux pour une meilleure visibilité des algorithmes.
Tables d'analyses :

- Actions : détermine s'il faut réduire ou décaler
- Successeur : détermine l'état à placer en sommet de pile après une réduction.

	Action						Successeur		
	id	+	*	(	)	$	E	T	F
0	d5			d4			1	2	3
1		d6				acc			
2		r2	d7		r2	r2			
3		r4	r4		r4	r4			
4	d5			d4			8	2	3
5		r6	r6		r6	r6			
6	d5			d4				9	3
7	d5			d4					10
8		d6			d11				
9		r1	d7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Algorithme Analyse LR
Données : Une chaîne w et une table d'analyse LR pour une grammaire G
Résultat : une dérivation droite inversée pour w si w ∈ L(G) une indication d'erreur, sinon.
Initialisation :

- A_0 dans la pile
- w$ dans le tampon
  positionner le pointeur source ps sur le 1er symbole de w$

```
répéter indéfiniment
        s ← sommet de la pile
        a ← symbole pointé par ps
        si (action[s, a] = décaler s') alors
                empiler(a)
                empiler(s') (s' est un état quelconque)
                incrémenter(ps)
        sinon
                si (action[s, a] = réduire par A → β) alors
                        dépiler 2 * |β| symboles
                        s' ← sommet de la pile
                        empiler(A)
                        empiler(successeur[s', A])
                        émettre(A→β)
                sinon
                        si (action[s, a] = accepter) alors
                                fin()
                        sinon
                                erreur()
                        fin si
                fin si
        fin si
fin repéter
```

Ex :

E → E + T (1)
E → T (2)
T → T * F (3)
T → F (4)
F → (E) (5)
F → id (6)

Pile	Entrée	Actions
0	id + id * id$	décaler
0id5	+ id * id$	réduire par F → id
0F3	+ id * id$	réduire par T → F
0T2	+ id * id$	réduire par E → T
0E1	+ id * id$	décaler
0E1+6	id * id$	décaler
0E1+6id5	* id$	réduire par F → id
0E1+6F3	* id$	réduire par T → F
0E1+6T9	* id$	décaler
0E1+6T9*7	id$	décaler
0E1+6T9*7id5	$	réduire par F → id
0E1+6T9*7F10	$	réduire par T → T * F
0E1+6T9	$	réduire par E → E + T
OE1	$	accepter

Il nous reste à voir comment remplir les tables action et successeur (nécessitent l'automate des préfixes viables).

Définition : on appelle <u>item</u> d'une grammaire G, toute production de G à l'intérieur de la quelle une position est maquée dans sa partie droite.

Ex : A → X Y Z

Fournit 4 items.

A -> .X Y Z
A -> X .Y Z
A -> X Y .Z
A -> X Y Z.

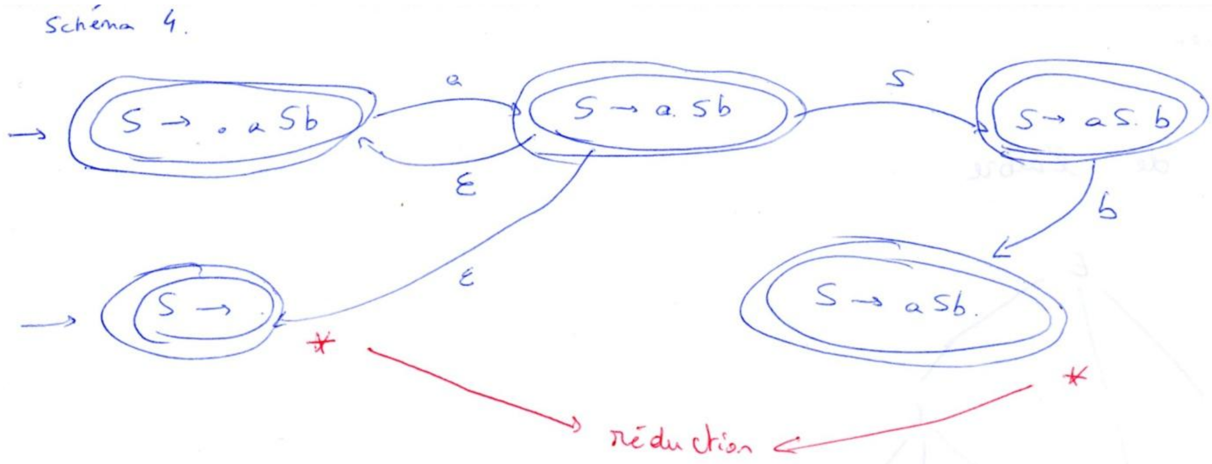
A -> ε

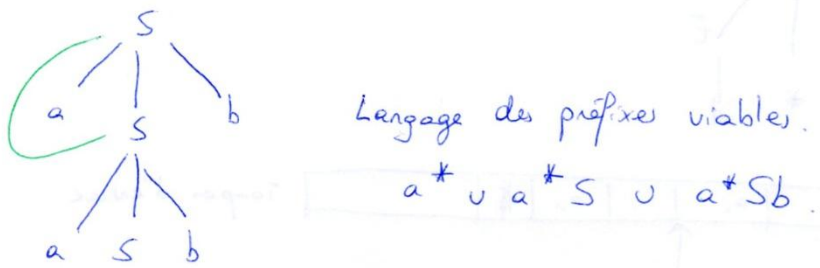
Fournit 1 item. A -> .

Intuitivement, les items permettent de repérer la « quantité » de partie droite d'une production déjà reconnue par l'analyseur.

L'ensemble des items d'une grammaire G constitue l'ensemble des états d'un AFN (Automate Fini non déterministe) reconnaissent les préfixes viables de cette grammaire.

S -> a S b | ε





S → aSb → aaSbb → aaaSbbb → … -> aaa….a (nb de a : n – 1) S b | bbbbbb..b (nb de b = n-1) -> a…a(n)|b…b(n)
Préfixes viables.

16/10/2013

Définition : Si G est une grammaire d'axiome S, on note G' la grammaire augmentée de G obtenue en ajoutant à G un nouvel axiome S' et une production supplémentaire S' → S

L'opération de fermeture

I : ensemble d'items → Fermeture(I) : ensemble d'items
Fermeture(I) ← I
Repéter
    Pour tout item de Fermeture(I) de la forme A → α.Bβ faire
        Pour toute production B → γ faire
            Ajouter B → .γ à Fermeture(I)
        Fin pour
    Fin pour
Jusqu'à stabilisation de l'ensemble

Ex :
E' → E
E → E + T | T
T → T * F | F
F → (E) | id
I = {E' →.E}
Fermeture(I) = {E'→.E, E → E + T, E →.T, T → .T * F, T →.F, F →.(E), F → id }

L'opération de Transition

Si I est un ensemble d'items et X un symbole de G alors Transition(I, X) est définie comme fermeture de l'ensemble de tous les items de la forme A →α.Xβ appartenant à I

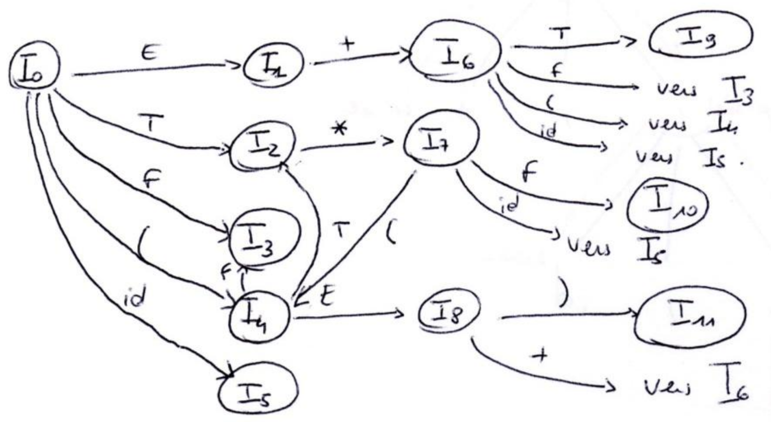
Exemple :

I = { E → T., T → T.*F}

Transition (I, *) = { T → T*.F, F→.(E), F → id }

Algorithme Ensemble d'items
Données Une grammaire augmentée G'
Résultat Une collection d'ensembles d'items correspondant aux états de l'automate fini déterministe (AFD) reconnaissant les préfixes viables de G'.
C ← Fermeture ({S' →.S})
Répeter
    Pour tout ensemble d'items I de C faire
        Pour tout symbole X de G faire
            Si Transition(I, X) ≠ ∅ alors
                Ajouter Transition(I, X) à C
            Fin si
        Fin pour
    Fin pour
Jusqu'à stabilisation de C

Schéma 1



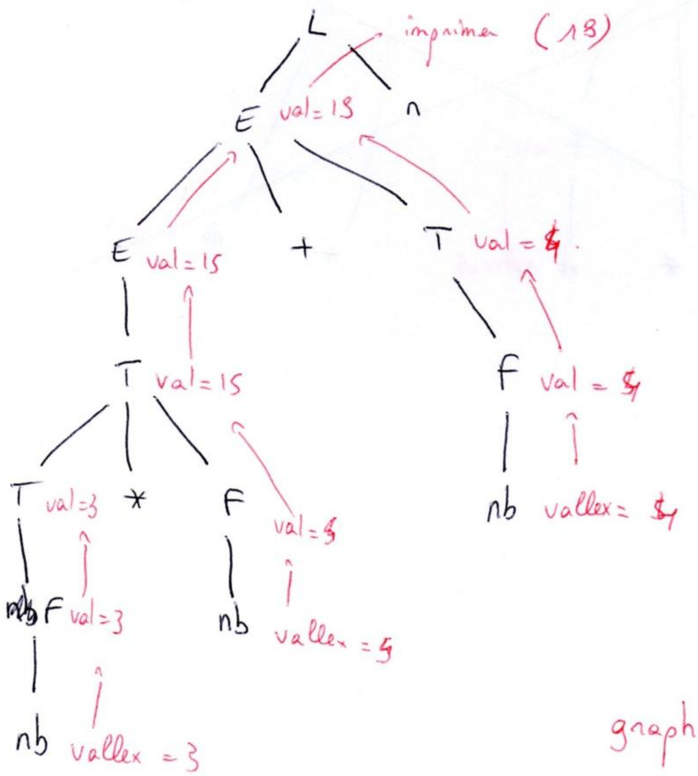
Reconnait les préfixes viables quand tous les états sont terminaux.

I_0 E' → .E	I_6
E → .E + T	E → E +. T
E → .T	T →.T*F
E → .T * F	T →.F
T → .F	F →.(E)
F → .(E)	F→.id
F → .id	
I_1 E' → E.	I_7

E $\rightarrow$ E.+T I_2 E $\rightarrow$ T. T $\rightarrow$ T.*F I_3 T $\rightarrow$ F. I_4 F $\rightarrow$ (.E) E $\rightarrow$ .E+T E $\rightarrow$ .T T $\rightarrow$ .F F $\rightarrow$.(E) F $\rightarrow$ .id I_5 F $\rightarrow$ id.	T $\rightarrow$ T*.F F $\rightarrow$.(E) F $\rightarrow$.id I_8 F $\rightarrow$(E.) E $\rightarrow$ E.+T I_9 E $\rightarrow$ E + T. T $\rightarrow$ T.*F I_{10} T $\rightarrow$ T *F. I_{11} F $\rightarrow$ (E).

Schéma 2:

3 * 5 + 4 n



graphe de dépendances.

```
Algorithme Construction d'une table d'analyse LR
Données une grammaire augmentée G'
Résultat Les tables Action et Successeur de G'
Construire C = {I0, …, In} la collection d'ensemble d'items pour G'
Pour i allant de 0 à n faire
        Pour tout symbole X de G' faire
                Si Transition(Xi, X) = Ij alors
                        Si X est terminal alors
                                Mettre « décaler j » dans Action[i, X}
                        Sinon
                                Mettre « j » dans successeur [i, X]
                        Fin si
                Fin si
        Fin pour
        Pour tout item A → α. ∈ I_i faire
                Pour tout a ∈ SUIVANT(A) faire
                Mettre « réduire par A → α » dans Action [i, a]
                Fin pour
        Fin pour
        Si S' → S. ∈ I_i alors
                Mettre « accepter » dans Action[i, $]
        Fin si
Fin pour
```

PREMIER ET SUIVANT

	Action						Successeur		
	id	+	*	(	)	$	E	T	F
0	d5			d4			1	2	3
1		d6				acc			
2		r2	d7		r2	r2			
3		r4	r4		r4	r4			
4	d5			d4			8	2	3
5		r6	r6		r6	r6			
6	d5			d4				9	3
7	d5			d4					10
8		d6			d11				
9		r1	d7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Chapitre 5 : Traduction dirigée par la syntaxe (début 16/10/2013 – fin)

Parcourir l'arbre syntaxique autant de fois que nécessaire pour évaluer les règles sémantiques du langage à ses nœuds.

Remarque : Cette phase d'analyse sémantique n'est pas nécessaire dissociée de la phase d'analyse syntaxique, l'arbre syntaxique peut même ne pas être construit explicitement.

5.1 Attributs

On associe des attributs aux symboles grammaticaux. Ils représentent des informations liées aux symboles telles que nombre, types, adresses mémoires, …

L'évaluation des attributs sur un arbre syntaxique est un processus appelé « décoration ». Il est réalisé selon les règles sémantiques du langage qui supposent des dépendances entre attributs (décrites par un graphe de dépendances).

2 types d'attributs :

- Les attributs <u>synthétisés</u> dont la valeur dépend de celles des attributs aux fils du nœud considéré.
- Les attributs <u>hérités</u> dont la valeur dépend de celles des attributs au père et aux frères du nœud considéré.

Formellement : dans une définition dirigée par la syntaxe chaque production $A \rightarrow \alpha$ possède un ensemble de règles sémantiques de la forme $b \leftarrow f(c_1, \ldots, c_k)$ où f est une fonction, b est soit un attribut synthétisé de A, soit une attribut hérité d'un symbole de α et $c_1, \ldots, c_k$ sont des attributs de symboles quelconques de la production.

Exemple :

Productions	Règles sémantiques
$L \rightarrow E\ n$	imprimer(L.val)
$E \rightarrow E_1 + T$	$E.val \leftarrow E_1.val + T.val$
$E \rightarrow T$	$E.val \leftarrow T.val$
T -> T_1 * F	$T.val \leftarrow T_1.val * F.val$
T -> F	$T.val \leftarrow F.val$
F -> (E)	$F.val \leftarrow E.val$
F -> nb	$F.val \leftarrow nb.vallex$

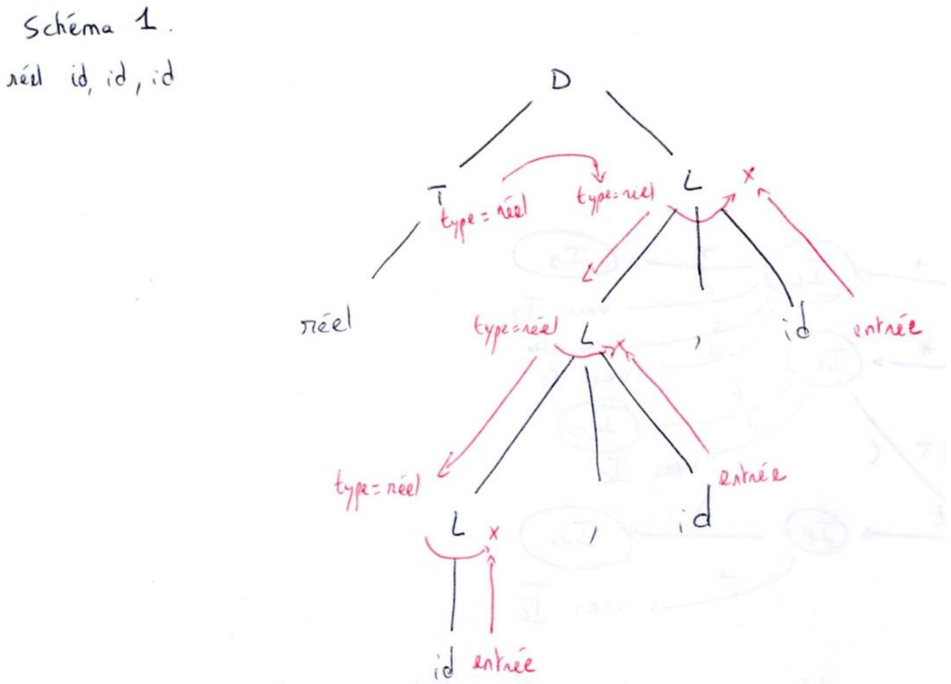
23/10/2013

<u>Définition</u> : une définition dirigée par la syntaxe n'utilisant que des attributs synthétisés est appelée une définition « S-attribuée ».

<u>Remarque</u> : Il est toujours possible de se ramener à une définition S-attribuée mais on peut perdre en intuition. Il est souvent plus naturel d'employer des attributs hérités.

Ex :

Productions	Règles sémantiques
D → TL	L.type ← T.type
T → entier	T.type ← entier
T → réel	T.type ← réel
L → L_1, id	L_1.type ← L.type
L → id	ajouter_type(id.entrée, L.type)
	ajouter_type(id.entrée, L.type)



Une règle sémantique peut induire des effets de bord. Lorsqu'elle ne sert qu'à cela (cf. ajouter_type) on lui associe un attribut factice qui apparait dans le graphe de dépendances. Ceci afin d'insérer l'appel à cette fonction dans le traducteur qui sera déduit de la définition dirigée par la syntaxe. L'ordre dans lequel sont évalués les attributs dépend des contraintes décrites par le graphe de dépendances.
Si ces contraintes ne sont pas contredites, l'ordre peut être induit par l'analyse syntaxique.
Seule impossibilité : cycle dans le graphe de dépendances.
Il existe des algorithmes (peu efficaces) pour tester la non-circularité des graphes de dépendances associés à une définition dirigée par la syntaxe.
Nous ne les étudierons pas et nous limiterons à des définitions compatibles avec les analyses LL et/ou LR.

5.2 Définitions S-attribuées

Attribut synthétisé → évalués des feuilles vers la racine.
A priori compatible avec un parcours suffixe (or l'anal. LR produit les nœuds précisément dans cet ordre).
Il suffit de gérer une pile d'attributs en parallèle avec la pile d'anal. synt. Pour une analyse LR.
Pour une analyse LL les nœuds sont générés en ordre préfixe. L'astuce consistera à laisser une post-it dans la pile pour mémoriser le fait qu'un attribut reste à calculer après le traitement de tous les fils d'un nœud.

5.2.1 Analyse ascendante

Schéma de traduction :

L → E n { imprimer(sommet()) ; dépiler()}

E → E + T {op2 ← sommet() ; dépiler() ; op1 ←sommet() ; dépiler() ; empiler(op1 + op2) }

E → T

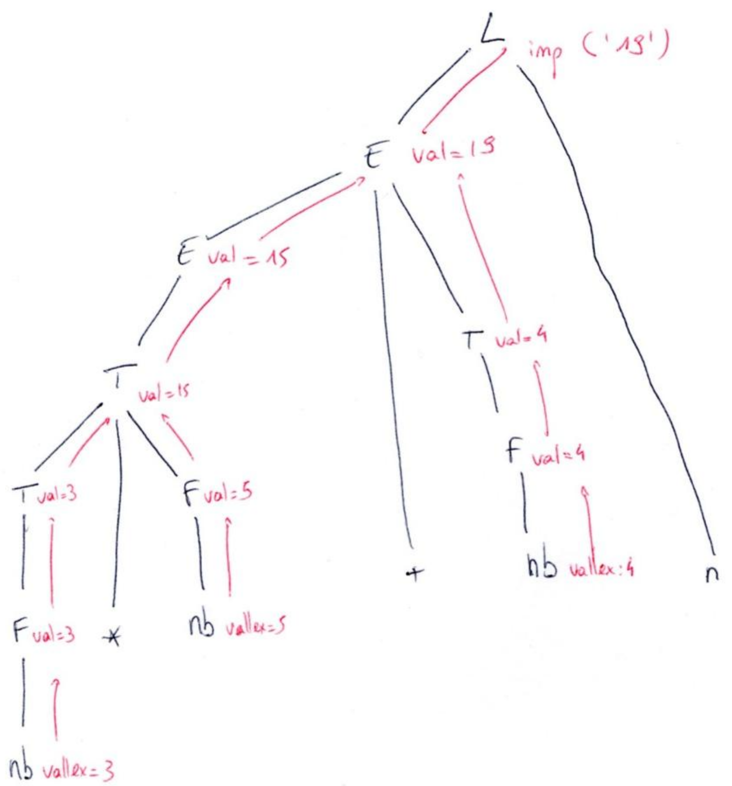
T → T * F {op2 ← sommet() ; dépiler() ; op1 ←sommet() ; dépiler() ; empiler(op1 * op2) }

T → F

F → (E)

F → nb

Pile d'analyse	Pile d'attributs	Entrée	Productions
		3 * 5 + 4n	
nb	3	* 5 + 4n	F -> nb
F	3	* 5 + 4n	T -> F
T	3	* 5 + 4n	
T *	3	5 + 4n	
T * nb	3 5	+ 4n	F -> nb
T * F	3 5	+ 4n	T -> T * F
T	15	+ 4n	E -> T
E	15	+ 4n	
E +	15	4n	
E + nb	15 4	n	F -> nb
E + F	15 4	n	T -> F
E + T	15 4	n	E -> E + T
E	19	n	
En	19	/	L -> En
L	/	/	



5.2.2 Analyse descendante

La grammaire est récursive gauche (pas d'anal desc possible) élimine la récursivité gauche ? Non ! (Apparition d'attri hérités -> cf + loin).

Autre exemple : évaluateur d'expressions préfixes.

Production	Règles sémantiques
L -> En L -> + E_1E_2 E -> * E_1E_2 E -> nb	imprimer(E.val) E.val ← E_1.val + E_2.val E.val ← E_1.val * E_2.val E.val ← nb.vallex

Schéma de traduction correspondant :

L -> En { imprimer(sommet()) ; dépiler() ;} (0)

E -> + EE {op2 ← sommet() ; dépiler() ; op1 ← sommet() ; dépiler() ; empiler(op1 + op2) ; } (1)

E -> * EE { op2 ← sommet() ; dépiler() ; op1 ← sommet() ; dépiler() ; empiler(op1 * op2) ; } (2)

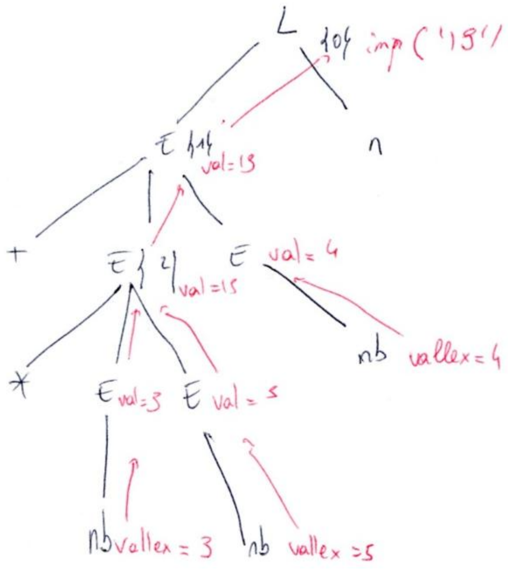
E -> nb

Table d'analyse LL :

	+	*	nb	n
L	L -> En	L -> En	L -> En	
E	E -> + EE	E -> *EE	E -> nb	

Pile d'analyse	Pile d'attributs	Entrée	Productions
L		+ * 3 5 4 n	L -> En
{0} n E		+ * 3 5 4 n	
{0}n {1} EE+		+ * 3 5 4 n	E -> + EE
{0}n {1} EE		* 3 5 4 n	E -> * EE
{0}n {1} E {2} EE *		* 3 5 4 n	
{0}n {1} E {2} EE		3 5 4 n	E -> nb
{0}n {1} E {2} Enb		3 5 4 n	
{0}n {1} E {2} E	3	5 4 n	E -> nb
{0}n {1} E {2} nb	3	5 4 n	
{0}n {1} E {2}	3 5	4 n	
{0}n {1} E	15	4 n	E -> nb
{0}n {1}	15 4	n	
{0}n	19	n	
{0}	19	/	
/	/	/	

Schéma 3 :



5.3 Définitions L-attribuées

(06/11/2013)

Quand des attributs hérités interviennent dans une définition il n'est pas toujours possible de réaliser une traduction dirigée par la syntaxe en parallèle avec l'analyse syntaxique.

Des définitions avec lesquelles cela reste souvent possible sont les définitions L-attribuées.

Définition : Une définition dirigée par la syntaxe est L-attribuée si tout attribut hérité de X_j dans la production $A \rightarrow X_1 \ldots X_n$ ne dépend que :

- Des attributs de $X_1, .., X_{j-1}$
- Des attributs hérités de A.

Une définition L-attribuée permet encore de décorer un arbre en réalisant un parcours en profondeur à main gauche.
Les attributs semblent se propager depuis la gauche (vers la droite) -> d'où le « L ».
Toute définition S-attr. est L-attr.

5.3.1 Analyse descendante

On sait déjà gérer les attributs synth.
Les attributs hérités se propagent en suivant à peu près l'ordre dans lequel les nœuds sont produits lors d'une analyse LL => facile, en principe.

Une manipulation qui fait apparaître des attributs hérités à partir d'une définition basée sur une grammaire récursive gauche -> l'élimination de la rec. gauche :

$A \rightarrow A_1\ Y\ \{A.a \leftarrow g(A_1.a, Y.y)\}$
$A \rightarrow X\ \{A.a \leftarrow f(X.x)\}$

A → X R { R.h ←f(X.x), A.a ← R.s }
R -> Y R_1 {R1.h ← g(R.h, Y.y), R.s ←R_1.s}
R -> ε {R.s ← R.h }

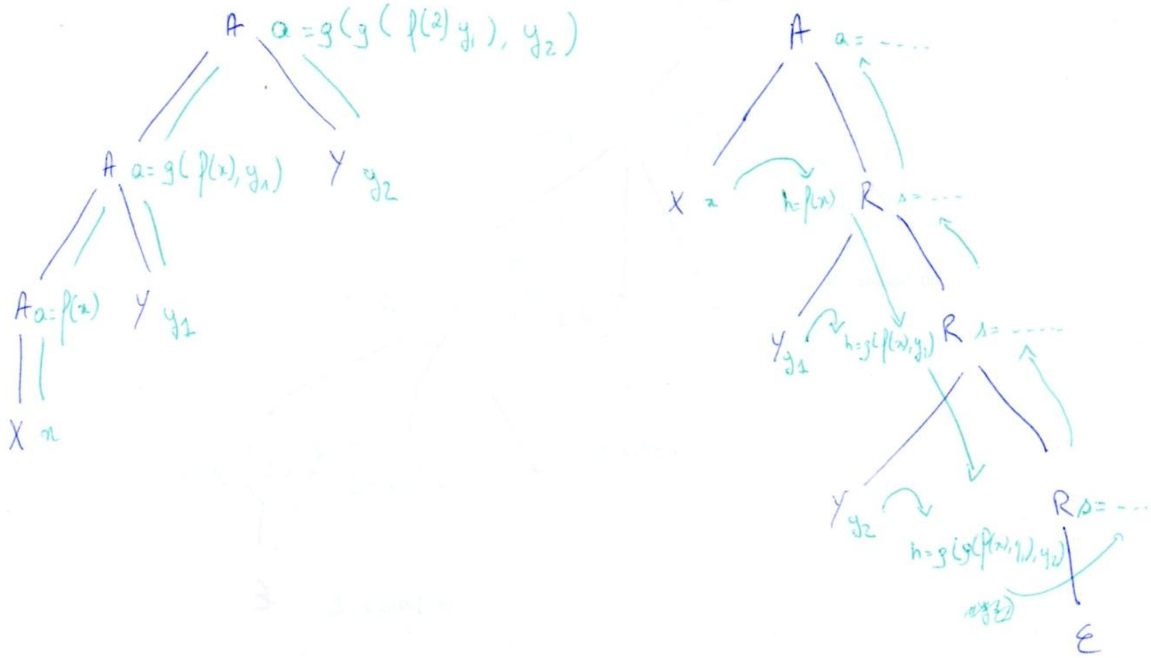


Schéma 1

Ex :
E → E_1 + T { E.val ← E_1.val + T.val}
E → E_1 – T {E.val ← E_1.val – T.val}
E → T {E.val ← T.val }
T → (E) { T.val ←E.val}
T → nb {T.val ← nb.vallex}

Elimination de la récursivité gauche :
E → TE' { E'.h ← T.val, E.val ←E'.val }
E' → + TE_1' {E1'.h ← E'.h + T.val, E'.val ← E1'.val}
E' → - TE_1' { E1'.h ← E'.h - T.val, E'.val ← E1'.val}
E'→ ε { E'.val ← E'.h }
T → (E) { T.val ←E.val}
T → nb {T.val ← nb.vallex}

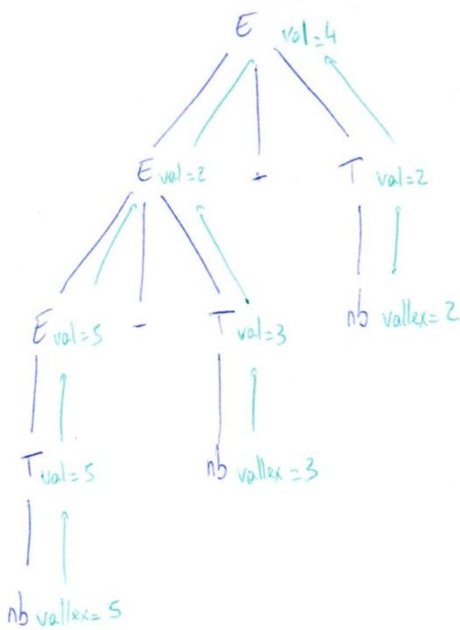


Schéma de traduction :

E → T T.val devient E'.h au sommet
E' E'.val devient E.h au sommet
E' →+
T {op2 ←sommet() ; depiler() ; op1←sommet(); depiler(); empiler(op1 + op2) } (0)
E_1' E_1'.val devient E'.h au sommet
E' → -
T {op2 ←sommet() ; depiler() ; op1←sommet(); depiler(); empiler(op1 - op2) }(1)
E_1' E_1'.val devient E'.h au sommet
E' → ε E'.val devient E'.val au sommet
T -> (
E E.val devient T.val au sommet
)
T -> nb T.val apparait au sommet avec la valeur de nb.vallex

Pile d'analyse	Pile d'attributs	Entrée	Productions
$E		5 – 3$	E -> TE'
$E'T		5 – 3$	T -> nb
$E'nb		5 – 3$	
$E'	5	– 3$	E' -> -TE'
$E' {1}T -	5	– 3$	
$E' {1}T	5	3$	T -> nb
$E' {1}nb	5	3$	
$E' {1}	5 3	$	
$E'	2	$	E' -> ε
$	2	$	
/	2	/	

5.3.2 Analyse ascendante

Problème :

- l'arbre est construit en remontant des feuilles vers la racine.
- Les attributs hérités sont évalués en « descendant ».

Difficile d'évaluer des attributs dépendant d'autres attributs associés à des nœuds non encore produits par l'analyse syntaxique.

Observation : seules les actions sont importantes in fine.

On peut donc se contenter de calculer les attributs synthétisés (les actions peuvent être assimilées à des attributs synth.) et se ramener à la situation des définitions S-attribuées.

Pourquoi ça peut marcher (fréquemment) ?

Lorsque la valeur d'un attribut hérité est nécessaire au calcul d'un attribut, cet attribut dépend directement ou indirectement d'attributs liés à des symboles tous situés dans la pile. On peut remonter aussi loin que nécessaire dans la pile pour rechercher les val. synth. dont on a besoin. (ça ne marchera pas, en cas d'attribut hérité « spontané »).

13/11/2013

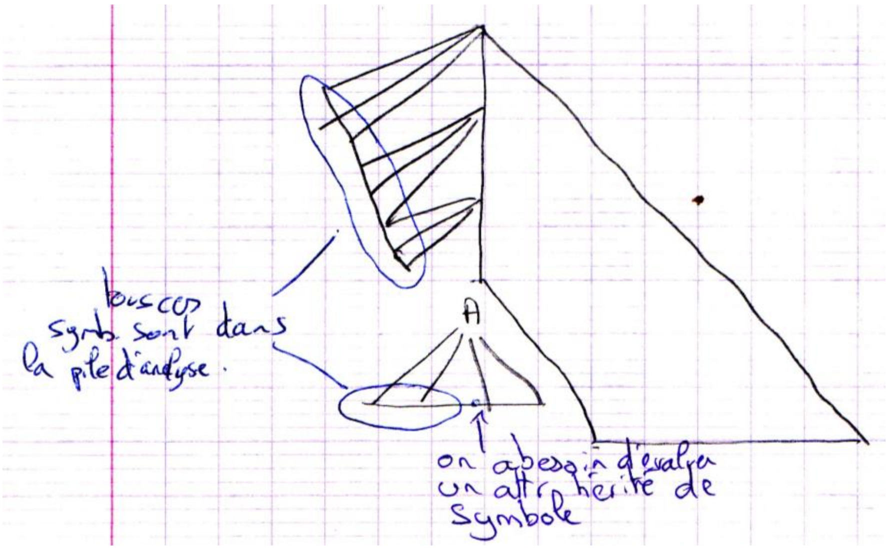


Schéma 1 (Voir tito).

Idée générale :

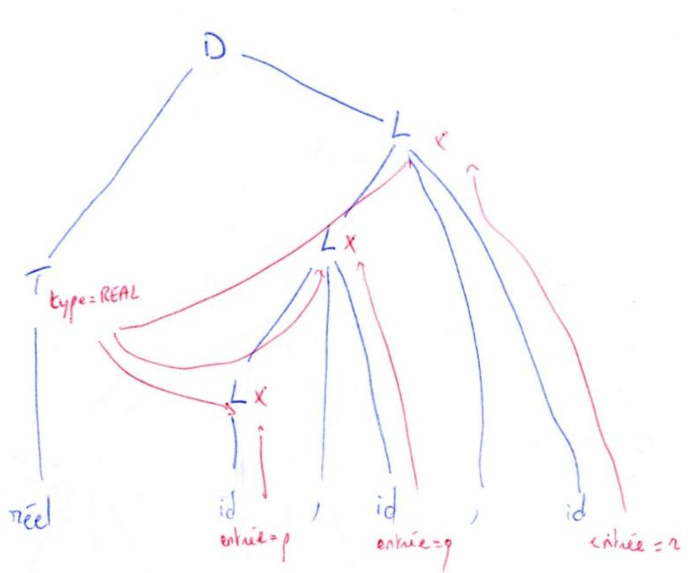
- Ne calculer que les attributs synthétisés
- Si la valeur d'un attribut herité est nécessaire, remonter dans la pile pour y trouver les valeurs nécessaires

Certains schémas de traduction peuvent être directement construits sur cette base :

D -> TL	L.type ← T.type
T-> entier	T.type ← entier
T -> réel	T.type ← réel
L -> L_1, id	L_1.type ← L.type
	ajouter_type(id.entrée, L.type)
L -> id	ajouter_type(id.entrée, L.type)

D -> TL {dépiler()}
T-> entier {empiler(INT)}
T->réel {empiler(REAL)}
L -> L,id {v ← sommet() ; dépiler() ; ajouter_type(v, sommet());}
L -> id {v ← sommet() ; dépiler() ; ajouter_type (v, sommet());}

Pile d'analyse	Pile d'attributs	Entrée	Production	Actions
		réel p, q, r		
réel		p, q, r	T -> réel	
T	REAL	p, q, r		
T id	REAL p	, q, r	L -> id	
T L	REAL	, q, r		ajouter_type(p, REAL)
T L ,	REAL	q, r		
T L , id	REAL q	, r	L -> L,id	
T L	REAL	, r		ajouter_type(q, REAL)
T L ,	REAL	r		
T L , id	REAL r	/	L -> L, id	
T L	REAL			ajouter_type(r, REAL)
D	/			



D'autres définitions ne permettent pas de déterminer facilement les positions dans la pile de certains attributs indispensables au calcul. -> Mise en place de marqueurs.

Ex :

S -> a A C C.h <- A.s

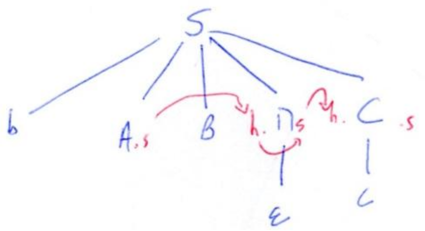
S -> b A B C C.h < A.s

C -> c C.s <- g(C.h)

Quand intervient une réduction par C -> c, l'attribut A.s nécessaire au calcul de C.h et donc de C.s se trouve, au choix, au sommet de la pile d'attributs ou juste en dessous, selon la production suivant laquelle on réduira ultérieurement. (sol : marqueurs !)

s-> a A C	C.h <- A.s
S -> b A B M C	M.h <- A.s
	C.h <- M.s
C -> c	C.s <- g(C.h)
M -> ε	M.s <- M.h

Schéma 4



<u>Rq</u> : Les marqueurs peuvent aussi être utilisés lorsque les règles sémantiques ne sont pas de sémantiques ne sont pas de simples copies.

Ex :

S -> a A C	C.h <- f(A.s)
Devient	
S -> a A N C	N.h <- f(A.s)
	C.h <- N.s
N -> ε	N.s <- N.h

E -> T E’	{E’.h <- T.val, E.val <- E’.val}
E’ -> + TE_1’	{ E_1’.h <- E’.h + T.val, E’Val <- E_1’.val }
E’ -> - TE_1’	{ E_1’.h <- E’.h - T.val, E’Val <- E_1’.val }
E’ -> ε	{E’.val <- E’.h }
T -> (E)	{T.val <- E.val}
T -> nb	{T.val <- nb.vallex}

E -> T E’ {}

E’ -> + T N_+E’ {}

E’ -> - T N_-E’ {}

E’ -> ε

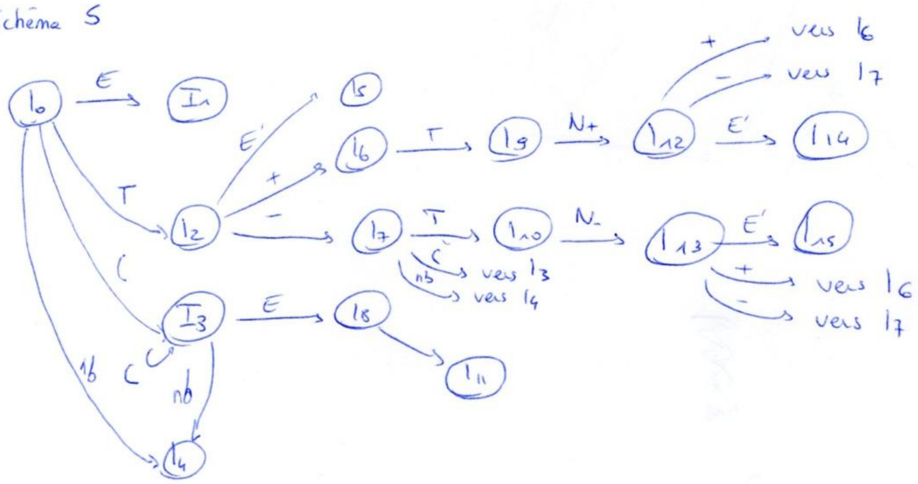
T -> (E) {}

T -> nb{}

N_+ -> ε {op2 <- sommet() ; dépiler() ; op1 <- sommet() ; dépiler() ; empiler(op1 + op2) }

N_- -> ε {op2 <- sommet() ; dépiler() ; op1 <- sommet() ; dépiler() ; empiler(op1 - op2) }

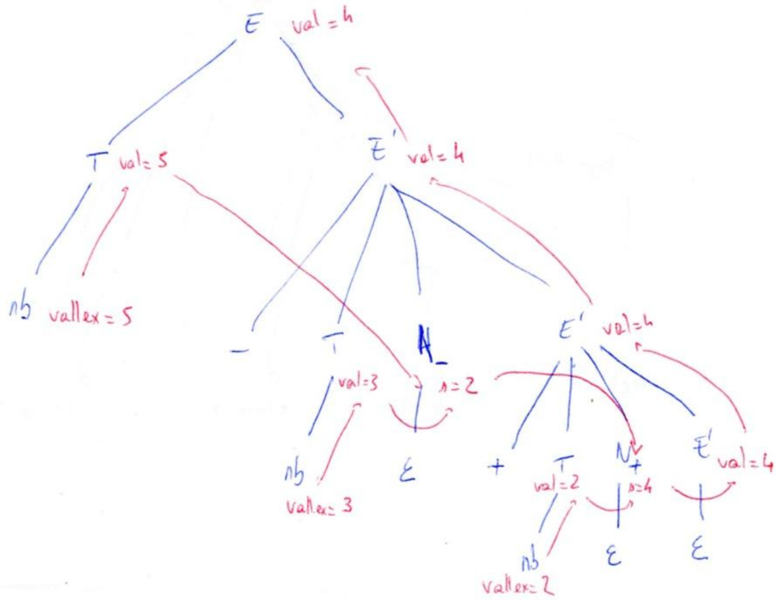
Schéma 5



I_0 S->.E E -> .TE’ T -> .(E) T->.nb	I_1 S -> E.	I_2 E -> T.E’ E’ -> .+TN$_+$E’ E’ -> .-TN$_-$E’ E’ -> .	I3 T -> (.E) E -> .TE’ T -> .(E) T->.nb	I4 T -> nb.	I5 E -> TE’.
I6 E’ -> +.TN$_+$E’ T -> .(E) T->.nb	I7 E’ -> .-TN$_-$E’ T -> .(E) T->.nb	I8 T -> (E.)	I9 E’ -> +T.N$_+$E’ N$_+$ -> .	I10 E’ -> -T.N$_-$E’ N$_-$ -> .	I11 T->(E).
I12 E’ -> +TN$_+$.E’ E’ -> .+TN$_+$E’ E’ -> .-TN$_-$E’ E’ -> .	I13 E’ -> - T N$_-$.E’ E’ -> .+TN$_+$E’ E’ -> .-TN$_-$E’ E’ -> .	I14 E’ -> + TN$_+$E’.	I15 E’ -> - TN$_-$E’.		

Pile d’analyse	Pile d’attributs	Entrée	Production
0		5 – 3 + 2	
0 nb 4	5	- 3 + 2	T -> nb
0 T 2	5	-3 + 2	
0 T 2 – 7	5	3 + 2	
0 T 2 – 7 nb 4	5 3	+ 2	T -> nb
0 T 2 – 7 T 10	5 3	+ 2	N$_-$ -> ε
0 T 2 – 7 T 10 N$_-$ 13	2	+ 2	
0 T 2 – 7 T 10 N$_-$ 13 + 6	2	2	
0 T 2 – 7 T 10 N$_-$ 13 + 6 nb 4	2 2	/	T -> nb
0 T 2 – 7 T 10 N$_-$ 13 + 6 T 9	2 2	/	N$_+$ -> ε
0 T 2 – 7 T 10 N$_-$ 13 + 6 T 9 N$_+$ 12	4	/	E’ -> ε
0 T 2 – 7 T 10 N$_-$ 13 + 6 T 9 N$_+$ 12 E’ 14	4	/	E’ -> + T N$_+$E’
0 T 2 – 7 T 10 N$_-$ 13 E’ 15	4		E’ -> - T N$_+$E’
0 T 2 E’ 5	4		E -> TE’
O E 1	4		accepter

Schéma 6



E val = 4
T val = 5
E' val = 4
nb vallex = 5
-
T
A_
E' val = 4
val = 3
s = 2
nb
vallex = 3
ε
+
T
val = 2
A+
s = 4
E' val = 4
nb
vallex = 2
ε
ε