Compilation ~ Résumé 2010.

Dubuc Xavier
contact@xavierdubuc.com

28 novembre 2018

Table des matières

---

1 Introduction

1.1 Structure générale d'un compilateur

Un compilateur est composé généralement de 6 entités réalisant le processus de traduction intéragissant avec 2 entités globales qui sont la table des identifiants et le gestionnaire des erreurs. Ces 6 entités se regroupent en 2 catégories,

1. l'analyse qui comporte
   - *l'analyse lexicale*,
   - *l'analyse syntaxique*,
   - *l'analyse sémantique* ;
2. la synthèse qui comporte
   - *le générateur de code intermédiaire*,
   - *l'optimiseur du code*,
   - *le générateur du code*.

2 Analyse lexicale

2.1 Buts

- lire les symboles du programme source, un à un,
- écarter les blancs (au sens large : espace simple, tabulation, retour à la ligne,...) et les commentaires,
- produire en sortie, un à un, les lexèmes formant le programme source,
- stocker dans la table des identificateurs tous les identificateurs (au sens large : identificateur de variable, de classe, de méthode, ...) du programme source,
- détecter et gérer les erreurs lexicales.

(Un lexème étant un symbole ou un ensemble de symboles reconnus par l'analyse lexicale)

2.2 Outils théoriques

2.2.1 Les expressions régulières (Regex)

Notations et vocabulaire

- un alphabet A est un ensemble fini de symboles *(par exemple, notre alphabet est l'ensemble* $\{a, b, c, ..., z\}$*)*,
- un mot w sur l'alphabet A est une suite de symboles de A *(par exemple, si* $A = \{0, 1\}$*, un exemple de mot :* $w = 01001$ *ou* $w = \epsilon$*,* ϵ *représentant le mot vide, c'est-à-dire aucun symbole)*,
- la concaténation de 2 mots w_1 et w_2 sur l'alphabet A, notée $w_1 \cdot w_2$ ou $w_1 w_2$ est le résultat du placement de w_2 après w_1 *(par exemple :* $w_1 = 01001$*,* $w_2 = 11 \Rightarrow w_1 w_2 = 0100111$*)*,
- A^* désigne l'ensemble de tous les mots possibles sur l'alphabet A
  (par exemple $\{0, 1\}^* = \{\epsilon, 0, 1, 00, 11, 01, 10, 000, ...\}$*)*,
- un langage $L \subseteq A^*$ est un sous-ensemble de A^* *(par exemple* $L = \{w | w$ *de longueur paire*$\}$*)*,

$\Rightarrow$ **une expression régulière** r **est une notation définissant un certain langage** $L(r)$**.**

Ces expressions régulières sont définies par induction comme suit :
Soit A un alphabet fixé,

- $r = a$ où $a \in A$ est une expression régulière définissant le langage $L(r) = \{a\}$
  $r = \epsilon$ est une expression régulière définissant le langage $L(r) = \{\epsilon\}$
- Soit r et s deux expressions régulières définissant les langages $L(r)$ et $L(s)$, alors, par ordre de priorité décroissante :
  - r^* est une expression régulière définissant le langage $L(r^*) = \{w_1 ... w_n | n \geq 0, \forall i\ w_i \in L(r)\}$,
  - $r.s$ est une expression régulière définissant le langage $L(r.s) = \{w_1 w_2 | w_1 \in L(r), w_2 \in L(s)\}$,
  - $r|s$ est une expression régulière définissant le langage $L(r|s) = L(r) \cup L(s)$
  - $r?$ est une abréviation de $(r|\epsilon)$
  - $[A - Za - z0 - 9]$ est une abréviation de $A|B|..|Z|a|b|..|z|0|...|9$
  - r^+ est une abréviation de $r.r^*$

Exemple : *On souhaite définir une expression régulière* $\boldsymbol{a}$ *définissant les constantes réelles non-signées. Elles sont constituées d'une partie entière formée d'un nombre* ≥ 1 *quelconque de chiffres suivie d'une partie fractionnaire optionnelle formée d'une virgule suivie d'un nombre* ≥ 1 *quelconque de chiffres, suivie d'une partie exposant optionnelle formée du symbole E suivi d'un signe optionnelle + ou - suivi d'un nombre* ≥ 1 *quelconque de chiffres.*

$$
\begin{aligned}
&s = 0|1|...|9\\
&r = s.s*\\
&pf =, .r\\
&pe = E.(+| - |\epsilon).r\\
&a = r.(pf|\epsilon).(pe|\epsilon)
\end{aligned}
$$

ou avec les abréviations :

$$
\begin{aligned}
&r = ([0-9])+\\
&pf =, .r\\
&pe = E.(+|-)?.r\\
&a = r.pf?.pe?
\end{aligned}
$$

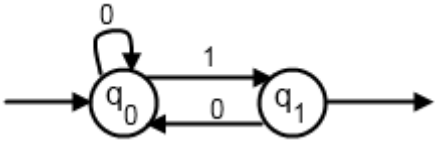
Remarque : *ces expressions ont une puissance limitée, en effet le langage formé par les mots bien parenthésés sur l'alphabet* $\{(,)\}$ *ni le langage* $L = \{0^n 1^n | n \geq 0\}$ *(en effet* $r = 0 * 1*$ *définit le langage* $L(r) : \{0^n 1^m | m, n \geq 0\}$*). L'intuition vient du fait que l'on est incapable au travers de ces expressions de «compter».*

2.2.2 Les automates finis

Un automate fini $\mathscr{A} = (Q, I, F, T, A)$ est constitué de
- Q : un ensemble fini d'états,
- $I \subseteq Q$: sous-ensemble des états initiaux,
- $F \subseteq G$: sous-ensemble des états finaux,
- $T \subseteq (Q \times A \times Q$ est la relation de transition,
- A un alphabet.

$\Rightarrow \mathscr{A}$ **est souvent représenté comme un graphe.**

Exemple :



$$
\begin{aligned}
&A = \{0, 1\}\\
&Q = \{q_0, q_1\}\\
&I = \{q_0\}\\
&F = \{q_1\}\\
&(q_0, 1, q_1) \in T\\
&(q_0, 0, q_0) \in T\\
&(q_1, 0, q_0) \in T
\end{aligned}
$$

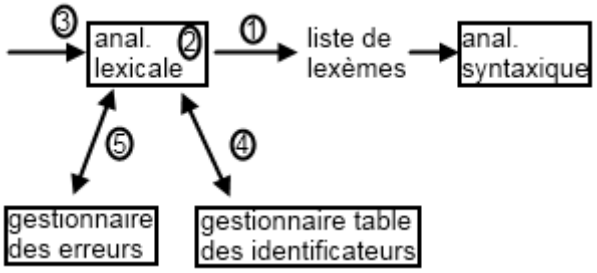
Un automate $\mathscr{A}$ reconnait un langage de mot de A^* noté $L(\mathscr{A})$:

$$L(\mathscr{A}) = \{w \in A^* | w \text{ est l'étiquette d'un chemin dans } \mathscr{A} \text{ allant d'un état initial à un état final.}\}$$

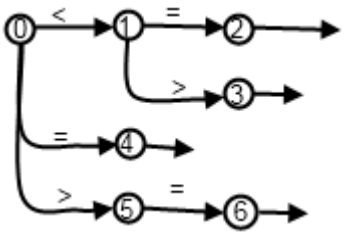
Lien avec l'analyse lexicale : l'analyseur dispose d'une série d'automates, un par type de lexèmes. Le programme source est lu symbole par symbole, en faisant fonctionner l'un de ces automates. Si cet automate accepte ainsi un w, on aura détecté le lexème w dans le programme source.

Théorèmes
1. Soit L un langage, alors L est défini par une expression régulière ssi L est accepté par un automate. **(théorème de Kleene, 1956)**
2. Soit $\mathscr{A}$ un automate non-déterministe, alors $\exists$ un automate déterministe qui accepte le même langage. (**Théorème de Rabin-Scott, 1959**)
3. Soit L un langage accepté par un $\mathscr{A}$, alors il existe un unique automate déterministe de taille minimale (en nombre d'états) qui calcule L.
Toutes les preuves sont **effectives** (c'est-à-dire que les preuves donnent un algorithme).

2.3 Algorithme pour l'analyse lexicale



1. A chaque lexème détecté, l'analyseur lexical va fournir en sortie son type et sa valeur *(par exemple type : constante entière, valeur : 1998 ; on lit 4 symboles : 1, 9, 9 et 8 que l'on interprête comme le lexème* 1998*)*,
2. Étant donné le langage de programmation utilisé pour le programme source, à chaque type de lexème autorisé par ce langage, on possède une expression régulière qui le définit, c'est-à-dire (via le théorème) un automate déterministe de taille minimale l'acceptant.
   L'analyseur lexical dispose de tous ces automates, placés dans un certain ordre. Au fur et à mesure de la lecture du programme source, il tente de faire fonctionner l'un après l'autre ces automates jusqu'à ce que :
   — soit l'un ait fonctionné et a donc détecté un lexème d'un certain type,
   — soit aucun ne fonctionne et on a détecté une erreur lexicale
   Concernant l'ordre, par exemple, l'automate des mots-clé doit précéder celui des identificateurs *(sinon tous les mots-clés seront des identificateurs)*.



Exemple d'algorithme : *(pour les opérateurs relationnels vis à vis de l'automate ci-haut, les états 7 et 9 correspondent à d'autres automates)*

```
etat_initial <- 0, erreur <- faux, lex_f <- faux
caractere <- SUIVANT()
Tant que caractere = " ", "\n" ou tab faire caractere <- SUIVANT()
   Tant que non erreur et non lex_f faire
      selon que etat_initial vaut
         0 : type_lexeme <- op_rel
             Si caractere = '<' alors
                caractere <- SUIVANT()
                Si caractere = '=' alors val_lexeme <- 'LE'
                Sinon Si caractere = '>' alors val_lexeme <- 'NE'
                Sinon
                   caractere <- PRECEDENT()
                   val_lexeme <- 'LT'
                lex_f <- vrai
             Sinon si caractere = '=' ...
             Sinon si caractere = '>' ...
             Sinon etat_initial <- 7
          7 : ...
          9 : ...
          Sinon erreur <- vrai
```

3. Un automate n'est rien d'autre qu'un petit bout de l'algorithme d'analyse lexiale d'où l'interêt d'avoir des automates déterministes et de taille minimale. On se rappelle que pour détecter un lèxème, on est parfois amené à lire un caractère supplémentaire du programme source. Ce caractère supplémentaire pouvant faire partie du lexème suivant, il faut utiliser PRECEDENT(). (Le programme source étant lu caractère par caractère)
4. **Table des identificateurs :** à chaque nouveau lexème détecté qui est de type identificateur il faut stocker celui-ci dans la table des identificateurs. Comme en SDD, les tables de hachage sont préconisées. Concernant les mots-clé, une solution proposée indique de ne pas utiliser d'automate mais à la place, l'automate des identificateurs et de faire la distinction entre eux et les identificateurs grâce à la table des identificateurs.

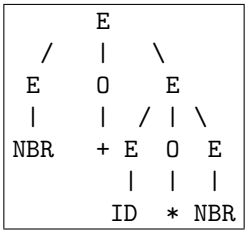
Lexème	Caractéristiques
class	*mot-clé*
void	*mot-clé*
...	...
...	...
position	
...	...

5. **Erreurs :**
   - **détection** : quand aucun automate n'a fonctionné (erreur lexicale)
     Exemples :
     - ... << ... → pas d'erreur détectée, on lit 2 fois le lexème <.
     - ... WIHLE ... → pas d'erreur détectée (nouvel identificateur détecté)
     - ... x... où x n'est autorisé par aucun automate → erreur détectée.

     ⇒ en pratique, il y a peu d'erreur lexicale.
   - **récupération** : passer le caractère posant problème et les suivants *(si nécessaires)* jusqu'à ce que l'un des automates fonctionne à nouveau *(méthode simple, il en existe des plus élaborées).* (Rappel : cette analyse ne détecte pas un oubli de «;» ou de «)»)

3 Analyse syntaxique

L'analyse syntaxique reçoit une liste de lexèmes en entrée, et en fonction de celles-ci, elle calcule un arbre de dérivation. Pour ce faire on va voir un outil théorique, les grammaires ainsi que 2 analyses syntaxiques, l'analyse syntaxique descendante et l'analyse syntaxique ascendante. Par exemple, pour une expression arithmètique : $2 + x * 3$ l'arbre pourrait être :



3.1 Les grammaires

Une grammaire décrit la syntaxe du langage de programmation utilisé. On les définit comme suit : soient :

— T, un alphabet terminal (dont les symboles sont appelés symboles terminaux),
— V, un alphabet de variables (dont les symboles sont appelés variables),
— S, un axiome, $S \in V$ (S comme **S**tart)

La grammaire est formée d'un ensemble de règles (**les règles syntaxiques**) de la forme $X \to \alpha$ où $X \in V$ et α est un mot sur $T \cup V$.

Par exemple pour les expressions arithmétiques comme ci-dessus, la grammaire associée est :

```
E -> EOE
E -> (E)
E -> ID
E -> NBR
O -> +
O -> -
O -> /
O -> *
```

Ici, $T = \{ID, NBR, (,), +, -, *, /\}$, $V = \{E, 0\}$, **E** est l'axiome. On remarque également que les expressions arithmétiques (E) sont définies récursivement et que les lexèmes (NBR, ID, +, *, -, /, (,)) reviennent.

Le but de l'analyse syntaxique est de vérifier que la suite de lexèmes calculées par l'analyseur lexical respecte bien la syntaxe du langage de programmation utilisé telle que définie par la grammaire donnée.

3.1.1 Les dérivations et les arbres de dérivation

Les dérivations sont définies comme suit :

$\gamma_1 X \gamma_2 \Rightarrow \gamma_1 \alpha \gamma_2$ où γ_1, γ_2 sont des mots sur $T \cup V$ et $X \to \alpha (X \in V)$ est une règle de la grammaire.

Le but de l'analyseur syntaxique est de trouver les dérivations à effectuer qui, en partant de l'axiome, donnent la liste des lexèmes donnée.
(Pour spécifier que l'on effectue plusieurs dérivations, on utilise $\to^*$)

L'arbre de dérivation est formé de noeuds internes qui sont étiquetés par des symboles de variables et de feuilles qui sont étiquetées par des symboles terminaux ou ϵ, et il est tel que la relation père-fils vérifie :

1. la règle $X \to a$ ($a = a_1 a_2 ... a_k$ symboles du mot a)

```
      X
 / | ... \
a1 a2 ... ak
```

2. la règle $X \to \epsilon$

```
   X
   |
epsilon
```

Remarque importante : $T = \{\text{symboles terminaux}\} = \{\text{types de lexèmes}\}$, *un lexème à ce stade n'est plus vu comme un mot mais comme un symbole terminal.*

3.1.2 Retour aux grammaires

Soit G une grammaire d'axiome S, le langage $L(G)$ défini par G est l'ensemble :

$$\{w \in T^* | S \to^* w\}$$

c'est-à-dire l'ensemble des suites de lexèmes w telles que à partir de l'axiome S, il existe une série de dérivations permettant d'arriver à w.

Terminologie en anglais

```
Grammaire (hors-contexte) -> (context-free) grammar
Analyse syntaxique -> parsing
Arbre de dérivation -> parsing tree
```

3.1.3 Puissance des grammaires

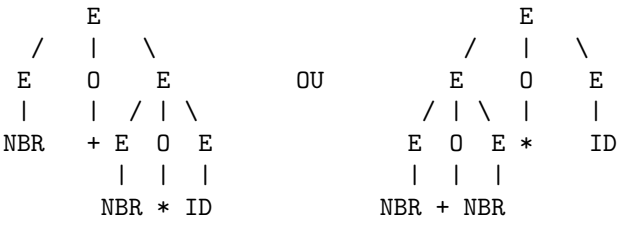
Propriété 1 : si L est un langage défini par une expression régulière, alors il existe une grammaire G qui permet également de définir L. **Les grammaires sont donc au moins aussi puissantes que les expressions régulières.**
Propriété 2 : il existe des langages L définis par des grammaires pour lesquels il n'existe aucune expression régulière les définissant. **Les grammaires sont donc plus puissantes que les expressions régulières.**
Propriété 3 : il existe des langages qui ne peuvent pas être définis par des grammaires. **Les grammaires ont donc une puissance limitée.** *(par exemple, une grammaire spécifiant un langage spécifiant les mots du type wCw avec $w \in \{a, b\}^*$. En effet, rien ne permet d'assurer que les 2 mots w soient identiques.*

Point de vue de compilation : plusieurs langages de programmation demandent de déclarer le type d'un identifiant avant de l'utiliser, ici wCw représente la déclaration de l'id, un bout de programme et l'usage de l'id. Ceci ne pouvant être vérifié par une grammaire, donc si on commet l'erreur d'oublier la déclaration d'un id, cette erreur ne peut pas être détectée lors de l'analyse syntaxique. Par contre, vérifier que les accolades sont bien organisées dans un programme pourra l'être.

On peut conclure que les expressions régulières et les automates ne savent pas compter, tandis que les grammaires le peuvent, mais de manière limitée grâce à une pile.

3.1.4 Grammaires ambigües

Une grammaire ambigüe est un grammaire telle que si pour $w \in T^*$, il existe 2 arbres de dérivation différents montrant que $S \rightarrow^* w$. Par exemple, pour les expressions arithmétiques, la suite de lexèmes $NBR + NBR * ID$ possède 2 arbres de dérivation :



En marge de cela, il faudra résoudre les cas d'ambiguité, via 2 solutions possibles :
— soit on remplace la grammaire par une grammaire non-ambigüe,
— soit on garde la grammaire ambigüe et aider l'algorithme de compilation à faire le bon choix.

Transformation de la grammaire des expressions arithmétiques en une grammaire non-ambigüe.

On remarque aisément que les opérateurs $*$ et $/$ sont plus prioritaires que $+$ et $-$ et que la grammaire est doit être associative gauche (5-4-3 doit correspondre à (5-4)-3 et pas à 5-(4-3)). On peut donc dresser le tableau suivant :

Opérateurs	$+$	$-$	$*$	$/$
Priorité	1	1	2	2
Associativité	gauche	gauche	gauche	gauche

On définit donc la grammaire suivante :

```
// Priorité 1
E -> E+T
E -> E-T
E -> T
// Priorité 2
T->T*F
T->T/F
T->F
// Priorité la plus haute
F->(E)
F->ID
F->NBR
```

On peut également y ajouter l'opérateur d'exponentiation, celui-ci étant prioritaire sur les autres et ayant une associativité droite (en effet $2^{3^4} = 2^{(3^4)}$). La grammaire devient donc :

```
E -> E+T
E -> E-T
E -> T
T -> T*F
T -> T/F
T -> F
F -> X^F
F -> X
X -> (E)
X -> ID
X -> NBR
```

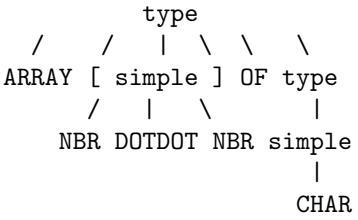
3.2 Analyse syntaxique descendante

Problème : Soit une grammaire G et $w \in T^*$ la suite de lexèmes produite par l'analyse lexicale. Il faut vérifier si $w \in L(G)$ et si c'est le cas donner l'arbre de dérivation correspondant ; sinon détection et récupération d'erreur.
→ Théorie des grammaires : il existe un algorithme qui teste si $w \in L(G)$ en $O(n^3)$ (trop couteux !) où n est le nombre de lexèmes produit par l'analyse lexicale. Nous allons, pour certaines familles de grammaires qui englobent généralement les grammaires des langages de programmation, on va détailler de tels algorithmes en $O(n)$ c'est-à-dire que l'on effectue un travail en $O(1)$ sur chaque lexème. Dans cette section, nous allons étudier une famille de grammaire où l'algorithme va construire l'arbre de dérivation de haut en bas et de gauche à droite.

Exemple introductif

```
(1) type -> simple
(2) type -> î ID
(3) type -> ARRAY [simple] OF type
(4) simple -> INTEGER
(5) simple -> CHAR
(6) simple -> NBR DOTDOT NBR
```

(î signifie pointeur)
$T = \{\text{î}, ID, ARRAY, [,], OF, INTEGER, CHAR, NBR, DOTDOT\}$
$V = \{type, simple\}$ (type est l'axiome).
Un exemple accepté : $ARRAY[2..17]OF\ CHAR$
→ $w = ARRAY[NBR\ DOTDOT\ NBR]OF\ CHAR$ L'arbre de dérivation est donc :



L'« *algorithme* » pour le construire de haut en bas et de gauche à droite est globalement comme suit :

1. Choisir entre (1), (2), (3) :
   — on écarte (1) car $simple \not\to ARRAY$,
   — on écarte (2) car il n'y a pas de pointeurs,
   — *on choisit donc (3)*
2. Matching entre le lexème lu et le lexème proposé par l'arbre («[»)
3. Choisir entre (4), (5), (6) pour $simple \to$ *on choisit (6)*
4. Matching pour $DOTDOT$, NBR,] et OF
5. Choisir entre (1), (2), (3) pour $type \to$ *on choisit (1)*
6. Choisir entre (4), (5), (6) pour $simple \to$ *on choisit (5)*

Il y a donc 2 types d'actions possibles :

1. **Match** entre le lexème lu et le lexème courant de l'arbre,
2. Étant donné un noeud X de l'arbre, avec $X \in V$, choisir la bonne règle $X \to \alpha$ (c-à-d α tel que $\alpha \to^* a\beta$ où a est le lexème lu).

Concernant cette 2ème action, le choix à faire est aisé dès lors que la **table M** a été construite :

X\a	**lexèmes possibles**
Symboles de variables possibles	

Dans notre cas :

X\a	î	ID	ARRAY	[	]	OF	INTEGER	CHAR	NBR	DOTDOT
type	(2)		(3)				(1)	(1)	(1)	
simple							(4)	(5)	(6)	

Algorithme d'analyse syntaxique sur l'exemple.
(lex_lu représente le lexème lu et lex_arbre le lexème courant de l'arbre)

```
Algorithme Match(lex_lu,lex_arbre)
   Si lex_lu = lex_arbre alors lex_lu <- <lexème suivant dans w>
   Sinon <erreur>

Algorithme Type(lex_lu)
   Si lex_lu appartient à {INTEGER, CHAR, NBR} alors Simple(lex_lu)
   Sinon Si lex_lu = î alors
      Match(lex_lu,î)
      Match(lex_lu,ID)
   Sinon Si lex_lu = ARRAY alors
      Match(lex_lu,ARRAY)
      Match(lex_lu,[)
      Simple(lex_lu)
      Match(lex_lu,])
      Match(lex_lu,OF)
      Type(lex_lu)
   Sinon <erreur>

Algorithme Simple(lex_lu)
   Si lex_lu = ...
   Sinon Si lex_lu  = ...
   Sinon Si lex_lu = ...
   Sinon <erreur>
```

Au niveau du fonctionnement, étant donné w et a son premier lexème, on appelle $Type(a)$. L'analyse s'est bien passée si l'algorithme **s'arrête sans tomber sur une erreur** et que **tous les lexèmes sont de w ont été traités**.

De manière générale, étant donné une grammaire, on construit sa table M et on écrit les différents algorithmes : *Match* et un par *symbole de variable*.

Il existe cependant des grammaires qui ne sont pas facilement traitables avec cette méthode, en effet si 2 règles ont un début identique (par exemple $A \to a$ et $A \to ab$), le compilateur devra faire le choix mais si il prend le mauvais choix, il devra revenir en arrière (*backtracking*) et essayer avec l'autre règle. Cela peut être très couteux si le *backtracking* demande de revenir très en arrière. On ne traitera donc pas de telles grammaires avec cette méthode, mais par d'autres méthodes mieux adaptées.

3.2.1 Grammaires $LL(1)$, First & Follow

Les grammaires qui peuvent être traitées par cette méthode d'analyse syntaxique descendante sont celles telles que la table M possède au maximum 1 règle par case. Ces grammaires sont appellées grammaires LL(1) *(on construit l'arbre de gauche (left) à droite et on lit 1 seul lexème à la fois dans w de la gauche vers la droite)*. Pour ces grammaires, l'algorithme d'analyse syntaxique descendante est en $O(1)$ où n est la taille de w.

Comment construire la table M ?

1. Cas simple
   On fait ici l'hypothèse que la grammaire ne possède pas de règle $Y \to \epsilon$, on introduit une notion importante :

FIRST(α)	=	$\{b \in T \mid \alpha \to^* b\gamma$ pour un certain $\gamma \in (T \cup V)^*\}$
	=	{**lexèmes** b tels que l'on peut dériver α0, 1 ou plusieurs fois de telle sorte à faire apparaître ce b en 1ère position}

   Dès lors, $M(X, a)$ contient $X \to \alpha$ si $a \in FIRST(\alpha)$ (c'est-à-dire $\alpha \to^* a\beta$).

2. Cas compliqué
   Ici la grammaire possède une ou plusieurs $Y \to \epsilon$, on introduit ainsi une nouvelle notion :

FOLLOW$(\mathbf{X})$	=	$\{b \in T \mid S \to^* \beta X b\gamma$ avec $\beta, \gamma \in (T \cup V)^*\}$
	=	{**lexèmes** b tels qu'en dérivant 0, 1 ou plusieurs fois l'axiome, on peut faire apparaître b juste après X}

   On rajoute $ si on a $S \to^* \beta X$ où $\beta \in (T \cup V)^*$ (c-à-d quand X n'est suivi d'aucun lexème)
   Dès lors, $M(X, a)$ contient $X \to \epsilon$ si $a \in FOLLOW(X)$.

Propositions

1. Toute grammaire ambigüe n'est pas **LL(1)**.
2. Toute grammaire récursive gauche (grammaire possédant des règles de la forme $X \to X\alpha$, $X \to \beta$) n'est pas **LL(1)**.
   Remarque : on peut toujours rendre une grammaire récursive gauche, non-récursive gauche :

   ```
   X -> XG
   X -> B
   ```

   Cette grammaire peut être résumée en : $X \to^* BGG...G$ (avec 0, 1 ou plusieurs G), elle peut être transformé en :

   ```
   X -> BX'
   X' -> epsilon
   X' -> GX'
   ```

3.2.2 Détection et récupération d'erreurs

Une erreur est **détectée** si il n'y a pas de match entre le lexème lu dans w et le lexème courant de l'arbre ou si il n'y a pas de règle proposée dans la table M lors du traitement du noeud X dans l'arbre et le lexème a dans w, autrement dit $M(X, a)$ est vide.

Récupération
Les erreurs possibles sont l'oubli d'un lexème, l'ajout d'un lexème en trop ou mettre un lexème à la place d'un autre, dans notre cas, nous allons faire l'hypothèse qu'il s'agit uniquement d'un oubli de lexème ou d'un lexème en trop.

1. En cas de **mismatch**, dans le cas d'un *oubli*, l'action de récupération est de passer a dans l'arbre et traiter le noeud suivant. Dans le cas d'un *lexème en trop*, on ignore b dans w et on passe au lexème suivant.
2. Lorsque $M(X, a)$ **est vide**, dans le cas d'un *oubli*, l'action de récupération est d'oublier X(le noeud en cours de dérivation) et son sous-arbre dans l'arbre et de passer au noeud suivant et dans w passer des lexèmes jusqu'à ce que le lexème à traiter $\in FOLLOW(X)$. Dans le cas d'un *lexème en trop*, on ignore le lexème et les lexèmes suivants jusqu'à lire un lexème $c \in FIRST(X)$

Toute la difficulté réside dans le fait de savoir dans quelle situation on se trouve, pour cela, des heuristiques basées sur la connaissance des fautes fréquentes dans les programmes sont utilisées.

3.3 Analyse syntaxique ascendante

Dans cette analyse, on construit l'arbre de dérivation du bas vers le haut et de la gauche vers la droite. On ajoute pour celà un nouvel axiome S' et une nouvelle règle $S' \to S$. Pour ce faire, il y a 2 actions importantes :

1. Lire le lexème suivante de le placer comme nouvelle feuille dans l'arbre de dérivation → SHIFT.
2. Remonter dans l'arbre de dérivation grâce à une règle de la grammeaire → REDUCE.
   (de α vers X avec $X \to \alpha$ une règle de la grammaire permise)

La difficulté est de n'utiliser qu'une règle permise à ce moment de l'analyse (règle qui n'amènera pas à un blocage plus loin dans la méthode). L'algorithme va utiliser une **pile** pour symboliser le passé et un **automate** (différent de ceux de l'analyse lexicale) qui l'aidera à bien faire les 2 actions qui sont à leur disposition. La **pile**, en cours d'exécution, contient la partie supérieure de ce qui a été construit dans l'arbre de dérivation et l'**automate** contient des états et des transitions étiquetées par des symboles $\in T \cup V$ ainsi que l'état initial I_0. Les états qu'elle contient sont en fait des états de l'automate qui sont intercalés entre chaque transition.

Structure de la pile :

I_0	α_1	I_x	α_2	I_y	α_3	...	α_k

Actions à effectuer

Pile :

	...	I	

Lexèmes à lire :

a	...		$

— Si, étant donnés I et a, I indique qu'il faut faire un **SHIFT**, alors on empile a et I' tel que l'automate possède la transition (I, I') avec comme étiquette a. On lit ensuite le lexème suivant.
— Sinon, s'il indique un **REDUCE** avec $X \to \alpha$, alors on dépile α et tous les états intercalés. Soit I', l'état au sommet de la pile après dépilement, on empile X et I'' tel que l'automate possède une transition (I', I'') avec C comme étiquette.
— On est en arrêt avec succès si I indique que l'on a fini (c'est-à-dire $a = \$$).
— On est dans une situation d'erreur sinon.

3.3.1 Méthode SLR

Méthode basée sur un automate **SLR**, automate qui contient des états et des transitions et à chaque état est associé l'ensemble des règles marquées de la forme $X \to \beta.\gamma$ (qui représentent en fait passé, présent, futur). Au début de l'algorithme, aucun lexème n'a été lu et I_0 est le contenu de la pile, dès lors les règles marquées ont toutes leur point à gauche (en effet, il n'y a pas de passé).

Construction de l'automate

— **États :**
  1. état initial I_0 qui contient la règle marquée $S' - > .S$,
  2. état I tel que : si I contient une règle marquée de la forme $X \to \alpha.Y\beta$ $(Y \in V, \alpha, \beta \in (T \cup V)^*)$ alors I doit contenir toute règle marquée de la forme $Y \to .\gamma$ telle que $Y \to \gamma$ est une règle de la grammaire.

— **Transitions :**
  Étiquetées par des symboles de $T \cup V$, si l'état I contient une règle marquée de la forme $X \to \alpha.\sigma\beta$ où $\sigma \in T \cup V$, alors il existe I' dans l'automate, possédant une règle marquée de la forme $X \to \alpha\sigma.\beta$, et une transition (I, I') avec σ comme étiquette.

On construit l'automate en partant de I_0 et en appliquant les règles définies ci-dessus au fur et à mesure.

Cette méthode, basée sur l'automate **SLR** fonctionne quand il n'y a pas de backtracking c'est-à-dire quand il n'y a pas de conflit **shift-reduce** ou de conflit **reduce-reduce**. Si c'est le cas, la grammaire associée est dite **SLR(1)** (on dérive toujours l'élément de droite et on lit un lexème à la fois de la gauche vers la droite)

Pour les grammaires non **SLR(1)**, il faut trouver d'autres méthodes d'analyse syntaxique ascendante. **Proposition :** les grammaires ambigües ne sont pas **SLR(1)**. Au travers de l'exemple des affectations en **C** *(voir cours)*, on remarque que l'automate **SLR** autorise des actions (dans ce cas-ci un reduce avec la règle $R \to L.$ dans l'état S_2) qui mène à des blocages. Autrement dit l'automate se trompe en autorisant cette action en cet état. On en conclut que la méthode **SLR** est trop simple et on va envisager une nouvelle méthode plus évoluée et plus puissante, la méthode **Canonical LR**.

3.3.2 Méthode Canonical LR

L'idée de l'automate Canonical LR est de remplacer la condition $a \in FOLLOW(X)$ du **reduce** par une condition plus contraignante. Dit autrement, au lieu de travailler avec l'entièreté de **FOLLOW(X)**.
Définition de l'automate :

- État : information = ensemble de couples $(X \to \alpha.\beta, b)$ avec $b \in T$ *(b lexème)* et $b \in FOLLOW(X)$.
- État initial I_0 tel que $(S' \to .S,\$) \in I_0$ ce qui est normal vu que $FOLLOW(S') = \{\$\}$
- L'automate **canonical LR** est construit petit à petit à partir de l'état initial selon les 2 principes suivants :
  1. Si $I \ni (X \to \alpha.Y\beta, b)$ avec $Y \in V$ alors I contient aussi $(Y \to .\gamma, c)$ tel que $Y \to \gamma$ est une règle de la grammaire et $c \in FIRST(\beta b)$.
  2. Si $I \ni (X \to \alpha.\sigma\gamma, b)$ avec $\sigma \in T \cup V$, alors il existe dans l'automate une transition (I, I') avec σ comme étiquette et $I' \ni (X \to \alpha\sigma.\gamma, b)$.

Actions possibles
Soit a le lexème courant, lu dans w et I l'état au sommet de la pile.

1. faire l'action «arrêt avec succès» si $a =\$$ et $I \ni (S' \to S.,\$)$,
2. faire l'action **shift** si l'automate possède une transition (I, I') avec comme étiquette $a \Rightarrow$ **empiler** a et I',
3. faire l'action **reduce** si $I \ni (X \to \alpha., a)$, dans ce cas, **dépiler** α et les états intercalés puis **empiler** X et I'' tel qu'il existe une transition (I', I'') avec X comme étiquette. (I' état au sommet de la pile après dépilement)
4. erreur sinon.

On aura une grammaire **canonical LR** si elle peut être traitée par cette méthode avec un automate sans conflit.
Conflits dans l'automate canonical LR

- conflit **shift-reduce** si transaction (I, I') avec a en étiquette et que $I \ni (X \to \alpha., a)$,
- conflit **reduce-reduce** si $I \ni (X \to \alpha., a)$ et $I \ni (X \to \beta., a)$.

3.3.3 Méthode LALR

On construit l'automate canonical LR et on fusionne ensuite les états de cet automate qui ont les mêmes règles marquées (regroupant toutes les 2èmes composantes possibles).

3.3.4 Comparaison entre les 3 méthodes

- L'automate canonical LR est en général plus gros (à cause des $2^{\text{èmes}}$ composantes) et on voit des sous graphes équivalents qui apparaissent.
- Dans l'automate canonical LR, $I \ni (X \to \alpha.\beta, b)$ indique $b \in FOLLOW(X)$; on n'a pas nécessairement tout $FOLLOW(X)$ apparaissent en $2^{\text{ème}}$ composante.
- L'automate LALR a pour but de garder la richesse de l'automate canonical LR, tout en fusionnant les sous-graphes équivalents dans le but de réduire la taille de l'automate.
- Au niveau de la puissance :

$$\{grammaires\ SLR\} \subsetneq \{grammaires\ LALR\} \subsetneq \{grammaires\ SLR\} \subsetneq \{grammaires\}$$

3.3.5 Grammaires ambigües

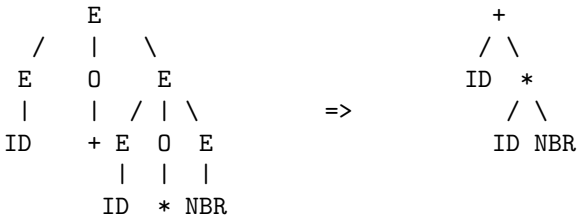
Dans l'analyse syntaxique descendante, on se contentait de rendre les grammaires non ambigües, dans l'analyse syntaxique ascendante, on peut également lever les conflits. (Par exemple lorsque l'on doit faire un reduce ou un shift avec + dans l'était I_7 pour les expressions arithmétiques, le reduce est à préconiser vu que + est associatif à gauche)

3.3.6 Gestion des erreurs

On détecte une erreur si aucune action de type shift, reduce ou arrêt avec succès pas possible. On dresse le tableau des actions possibles en fonction des états et des lexèmes et toutes les cases vides sont les erreurs détectées. Pour récupérer d'une erreur, l'idée générale est de faire comme si on avait traiter X, c'est-à-dire dépiler α, empiler X et passer les lexèmes jusqu'à lire un lexème b qui suit X tel que $b \in FOLLOW(X)$. Mais en général on effectue une analyse plus fine grâce à une étude des erreurs commises et récupération d'erreur adaptée.

4 Analyse sémantique

1. **Compilateur classique :** les tâches importantes sont le contrôle de type et la construction de l'arbre abstrait.
   La **déclaration de type** est traitée à ce niveau (il y a donc interaction avec la table des identificateurs). Une erreur de type peut venir d'un identificateur non déclaré, un nombre de paramètres d'une méthode qui n'est pas respecté, si, par exemple, la fonction modulo est appliquée à des entiers. L'**arbre abstrait** est construit à partir de l'arbre de dérivation, « en compactant » celui-ci. Par exemple :



2. **Variantes :** par exemple traduire un programme *html* en **latex**. Dans ces cas là, l'analyse sémantique joue le rôle de traduction. (ici traduire la liste en html en une liste en latex)

4.1 Grammaires attribuées

Il s'agit d'un ensemble de couple (règle syntaxique, règle sémantique). Exemple de grammaire attribuée permettant de traduire vers le résultat de l'expression arithmétique (attribut valeur, noté .val) :

```
E1 -> E2+T      E1.val <- E2.val + T.val
E -> T          E.val <- T.val
T1 -> T2*F      T1.val <- T2.val * F.val
T -> F          T.val <- F.val
F -> (E)        F.val <- E.val
F -> NBR        F.val <- valeur reconnue lors de l'analyse lexicale pour NBR
```

Un attribut *att* est dit synthétisé si dans la grammaire attribuée on a :

$$X \to \alpha_1\alpha_2...\alpha_n \qquad X.att \leftarrow f(\alpha_1.att, \alpha_2.att, ...\alpha_n.att)$$

Un attribut *att* est dit hérité si dans la grammaire attribuée on a :

$$X \to \alpha_1\alpha_2...\alpha_n \qquad \alpha_i.att \leftarrow f(X, \alpha_j.att)(j \neq i)$$

4.2 Contrôle de type

Le but est de compléter la table des identificateurs concernant leur type et de repérer les erreurs sémantiques liées au type, on utilise pour ça un attribut *type* qui contient soit le *type* soit *error* s'il y a eu une erreur soit *void* s'il n'y a pas de type. *(cf exemple du cours)*

4.3 Implémentation, évaluation des grammaires attribuées

1. **Méthode générale.** *(cf illustration sur l'exemple du cours)*
   (a) Construire un graphe dont les noeuds sont ceux de l'arbre de dérivation et donc les flèches sont telles que si on a le calcule d'attribut de la forme $X \leftarrow fct(y_1, y_2, ...y_n)$ alors flèche de y_i à X.
   (b) Tri topologique de ce graphe → numéroter les sommets de telle manière que $A \rightarrow B \Rightarrow n(A) < n(B)$. Ce tri permet de savoir dans quel ordre il faut effectuer les calculs d'attributs de l'arbre de dérivation.
   (c) En déduire l'algorithme des calculs d'attributs pour l'arbre de dérivation donné.

   Méthode générale **mais** ne peut être appliquée qu'après avoir construit l'arbre de dérivation en entier *(pour les grammaires L ou S- attribuées, on peut appliquer la méthode en même temps que l'on construit l'arbre de dérivation).*

2. **Méthode pour les grammaires S-attribuées.**
   Une grammaire attribuée est dite S-attribuée si tous ses attributs sont **synthétisés**. On va combiner le calcul de ces attributs avec une des méthodes d'analyse syntaxique ascendante. D'un point de vue implémentation il suffit d'enrichir la pile utilisée par l'une de ces méthodes avec un champ supplémentaire dédiée aux attributs.
   Exemple *(pour 2*3+7 dont on synthétise la notation postfixe)* :

E	*	E	
2		3	

⇒

E	
2 3 *	

   (cf exemple complet dans le cours)

3. **Méthode pour les grammaires L-attribuées.**
   Une grammaire attribuée est dire L-attibuée si chaque fois que les/l' attribut(s) hérité(s) de α_i est/sont fonction uniquement des attributs hérités du père X et des attributs hérités ou synthétisés de ses frères gauches $\alpha_1...\alpha_{i-1}$. *(Les attributs synthétisés sont autorisés)* Le calcul des attributs va être combiné avec la méthode d'analyse syntaxique descendante. Ce sera également possible avec les méthodes d'analyse syntaxique ascendante. On en sort l'algorithme :

```
Algorithme CalculAtt(n, attr hérités de n)
 Entrée : noeud n de l'arbre de dérivation et ses attributs hérités.
 Sortie : ses attributs synthétisés.
   Pour chaque fils m de n, de gauche à droite
      évaluer les attributs hérités de m
      att synt de m <- CalculAtt(m, att hérités de m)
   évaluer les attributs synthétisés de m
   retourner les valeurs calculées
```

   Pour combiner cet algorithme avec l'analyse syntaxique descendante, on a besoin des schémas de traduction qui sont obtenus à partir des grammaires **L-attribuées** en venant insérer les règles sémantiques à l'intérieur des règles syntaxiques au vu de l'algorithme *CalculAtt. (Voir exemple du cours)*

   On peut également adapter ce calcul d'attributs à l'analyse syntaxique ascendante, comme vu dans l'exemple du cours, celui-ci s'appuie sur le fait que les attributs à utiliser pour calculer un attribut hérité sont trouvables dans la pile au même endroit à chaque fois. (ce n'est pas toujours le cas, dans ce cas là on utilise des marqueurs pour se ramener à la situation précédente, *cf exemple du cours*)