ANALYSE SYNTAXIQUE ASCENDANTE DETERMINISTE

Dans cette partie nous décrivons une famille de méthodes d'analyse syntaxique, dites « ascendantes » qui construisent l'arbre syntaxique en partant des feuilles vers la racine. Ces méthodes sont « déterministes » c'est-à-dire qu'elles imposent des contraintes sur les grammaires prises en compte qui garantissent que l'analyseur syntaxique peut toujours connaître la prochaine action à exécuter « à coup sûr ». En particulier ces méthodes ne s'appliquent pas à une grammaire ambiguë, dont au moins un mot du langage associé admet deux arbres syntaxiques, donc deux séquences d'actions d'acceptation du mot.

À cette étape on ne vérifie que la syntaxe du programme à compiler, en laissant à des étapes ultérieures la vérifications des aspects « contextuels » (analyse de portée des identificateurs, typage, etc). L'analyse syntaxique détecte les erreurs de syntaxe d'un programme : mauvaise imbrication de parenthèses, construction incorrecte, symbole oublié ou mal placé, mot-clef mal orthographié, etc. Une difficulté est d'être capable de corriger une telle erreur, ou d'en limiter l'impact, afin de pouvoir continuer l'analyse syntaxique de la suite du programme de façon pertinente, sans provoquer une cascade de messages d'erreurs, éventuellement incorrects.

Comme pour l'analyse lexicale, l'analyseur syntaxique est souvent obtenu grâce à un « générateur d'analyseurs syntaxiques » prenant en entrée une grammaire respectant certaines contraintes techniques. En plus de vérifier si le mot appartient au langage défini par la grammaire, ces systèmes permettent d'exécuter des actions au fur et à mesure de l'utilisation des productions, par exemple pour construire l'arbre syntaxique, ou un AST, du programme.

Dérivations « droites » d'un mot

Lorsqu'une forme sentencielle comporte plus d'un non-terminal, le choix de celui à dériver à la prochaine étape est a priori arbitraire. Il existe deux méthodes naturelles de choisir le prochain non-terminal : soit on remplace systématiquement le non-terminal le plus à gauche, soit on remplace systématiquement le non-terminal le plus à droite. On obtient deux manières de procéder, appelées respectivement « dérivations gauches » et « dérivations droites ». Dans la suite nous n'utiliserons que de « dérivations droites ». Si la grammaire est non-ambiguë, tout mot du langage a exactement une dérivation droite. Nous illustrons cette notion avec notre grammaire G3 et la dérivation du mot `ident+ident*ident`. Le non-terminal dérivé à chaque étape est souligné:

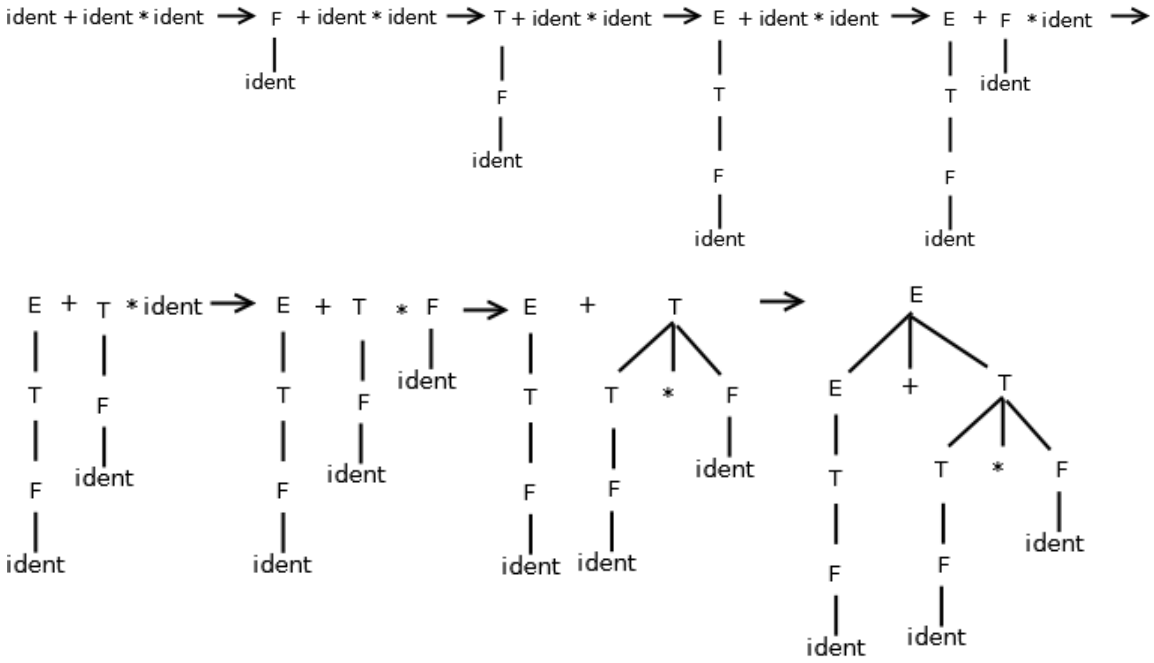
$\underline{E} \rightarrow E + \underline{T} \rightarrow E + T * \underline{F} \rightarrow E + \underline{T} * ident \rightarrow E + \underline{F} * ident$

$\rightarrow \underline{E} + ident * ident \rightarrow \underline{T} + ident * ident \rightarrow \underline{F} + ident * ident$

$\rightarrow ident + ident * ident$

Sur cet exemple nous connaissions la séquence de dérivations à utiliser pour ce mot. Le problème de l'analyse syntaxique est de déterminer si une telle séquence existe ! Il n'est pas très pratique avec une « dérivation droite » de contrôler qu'une tentative de dérivation est raisonnable puisque la partie constituée **uniquement** de terminaux apparaît progressivement **de la droite vers la gauche**. Ainsi, l'utilisation à la première étape de la règle $E \rightarrow E + T$ n'est justifiée que si le mot à analyser comporte au moins un symbole +, ce qui demande de connaître le sous-mot associé au non-terminal E de la partie droite $E + T$. Dans une « dérivation droite », pour contrôler plus facilement un début d'analyse syntaxique, on devrait logiquement parcourir le mot à reconnaître de **droite à gauche**, ce qui est peu réaliste quand le « mot » est un programme de taille respectable. Cet inconvénient disparaît si on pratique une telle dérivation **à rebours**, en « remontant » le sens des flèches (voir ci-dessous) ! Dans ce cas on ne manipule plus **un** arbre syntaxique, mais une **forêt d'arbres** dont les racines sont étiquetées par des symboles terminaux (vus comme des arbres restreints à une feuille) ou non-terminaux. Les lettres du mot sont progressivement transformées en arbres, de la **gauche vers la droite**. Dans l'exemple ci-dessous, on remarque que les étiquettes des racines des arbres de la première ligne sont exactement aux symboles utilisés dans la dérivation initiale : à certaines étapes, des arbres « indépendants » deviennent des sous-arbres d'un nouvel arbre. La première étape accède

au premier symbole du mot, `ident`, en le voyant comme l'étiquette d'un arbre réduit à sa racine, avant avant que les suivantes en fassent un arbre d'un arbre de racine `F` puis `T`.



Dans une analyse vue comme une dérivation droite à rebours, nous aurons deux types d'actions :

- empiler la lettre courante du mot sous la forme d'un arbre réduit à sa racine. Ceci afin de constituer progressivement une partie droite de règle, pour pouvoir procéder ultérieurement à un enracinement.
- dépiler une **partie droite** de production et empiler sa **partie gauche** : les arbres dont les étiquettes des racines forment une partie **droite** de production deviennent sous-arbres d'un arbre dont la racine est la partie **gauche** de la production. Cet « **enracinement** » se fait toujours de la **droite vers la gauche** : on ne peut pas enraciner un sous-arbre tant que tous les arbres à sa droite ne l'ont pas été : la forêt d'arbres est gérée en **pile**.

Les fonctions First et Follow

Ci-dessus, nous avons occulté le problème du choix de l'action à effectuer (enraciner une forêt ou empiler la lettre courante du mot), en connaissance la dérivation initiale. Dans la suite, ce choix se fera « à coup sûr » grâce à la connaissance des prochaines lettres du sous-mot qui reste à analyser. On appelle ces lettres des "caractères d'avance " (« *lookahead characters* »). Pour des raisons de performances, on se limite en général à un seul caractère d'avance, même s'il existe des systèmes qui travaillent avec plus d'un caractère d'avance. Pour certaines grammaires, il faut cependant connaître un nombre de caractères d'avance plus important (voire non borné) pour faire le choix à coup sûr.

Nous définissons deux fonctions, appelées *First* et *Follow*, qui permettront d'énoncer les propriétés de déterminisme recherchées, avant de donner les relations pour les calculer. Pour garantir qu'il existera toujours un « caractère d'avance », nous ajoutons un **symbole terminal supplémentaire** en fin de mot, noté `$` dans la suite (en pratique il s'agit du caractère de fin de fichier), ce qui revient à ajouter une production `S' ::= S` et à faire jouer à `S'` le rôle d'axiome joué par `S`. Le symbole `$` sert de butoir et ne sera jamais empilé.

Définitions :

$First(X) = \{ a \in V_t \ / \exists \alpha \in V_t^* : X \rightarrow^* a\,\alpha \} \cup \{ \varepsilon \}$ si $X \rightarrow^* \varepsilon$,

$\{ a \in V_t \ / \exists \alpha \in V_t^* : X \rightarrow^* a\,\alpha \}$ sinon.

$Follow(X) = \{ a \in V_t \ / \ \exists \alpha, \beta \in V_t^* : S \rightarrow^* \alpha\, X\, a\, \beta \}$

Nous commençons par calculer *First*, à valeurs dans $V_t \cup \{ \varepsilon \}$, en étendant sa définition à toute séquence de symboles avec les définitions par induction suivantes ($a \in V_t$, $X \in V_N$, $\alpha \in (V_t \cup V_N)^*$)

First(ε) = { ε }
First(*a*) = { *a* }
First(*X*) = ∪ *First*(α), union sur toutes les règles de la forme `X ::=` α dans G
First(*a*α) = { *a* }
First(*X*α) = (*First*(*X*) - { ε }) ∪ *First*(α) si $X \rightarrow^* \varepsilon$
First(*X*) sinon.

La dernière règle exprime que le premier caractère posé par *X*α provient soit de *X*, soit de α quand $X \rightarrow^* \varepsilon$. Quand ε est dans *First*(X), il ne se retrouve pas toujours dans *First*(*X*α) puisque α peut poser une lettre qui empêcherait d'avoir *X*α $\rightarrow^*$ ε. Si de plus α $\rightarrow^*$ ε, alors ε est dans *First*(α) et se retrouve dans *First*(*X*α).

La difficulté de ces règles est qu'il n'y a pas d'ordre dans lequel appliquer les troisième et cinquième cas, des non-terminaux pouvant se définir mutuellement. On peut montrer que *First* peut être calculé par itérations, en démarrant avec un ensemble *First* vide pour chaque non-terminal. Une étape de calcul applique les règles aux valeurs courantes des *First* pour définir leurs nouvelles valeurs. Ceci est répété jusqu'à ce qu'aucun ensemble *First* ne puisse plus être enrichi. Pour les exemples des grammaires `G3` et `G2` vues auparavant, on obtient :

```
G3:
S ::= E $                          First(S) = { ident, ( }
E ::= E + T | T                    First(E) = { ident, ( }
T ::= T * F | F                    First(T) = { ident, ( }
F ::= ident | ( E )                First(F) = { ident, ( }
```

```
G2:
S ::= E $                          First(S)  = { ident, ( }
E ::= T E'                         First(E)  = { ident, ( }
E'::= + T E' | ε                   First(E') = { +, ε }
T ::= F T'                         First(T)  = { ident, ( }
T'::= * F T' | ε                   First(T') = { *, ε }
F ::= ident | ( E )                First(F)  = { ident, ( }
```

Considérons maintenant la relation *Follow*, à valeurs dans V_t (donc jamais de ε dans un *Follow*) :

$$Follow(X) = \{ a \in V_t \ / \ \exists \alpha, \beta \in V_t^* : \ S \rightarrow^* \alpha X a \beta \}$$

Follow(X) ne se réduit pas à « *a* est **voisin** de *X* » dans une règle : soit les règles `X ::= X1 a` et `X1 ::= X2`. La première règle donne `a` ∈ *Follow*(`X1`) et la dérivation `X` → `X1a` → `X2a` entraîne `a` ∈ *Follow*(`X2`). De même, avec une règle `X ::= X1 X2 X3 a`, et `X2` $\rightarrow^*$ ε, `X3` $\rightarrow^*$ ε on aura `a` dans le *Follow* de `X1`, `X2` et `X3`.

On calcule *Follow* par application itérée des relations suivantes aux règles de G :

si X ::= α Y β ∈ G, pour tout `a` dans *First(β), a* ≠ ε*, alors* `a` ∈ *Follow(Y)*
si X ::= α Y β ∈ G et β $\rightarrow^*$ ε*, pour tout* `a` *dans Follow(X)*, alors `a` ∈ *Follow(Y)*
si X ::= α Y ∈ *G, pour tout* `a` *dans Follow(X), alors* `a` ∈ *Follow(Y)*

Le troisième cas n'est qu'un cas particulier du deuxième avec β = ε ! Nous l'ajoutons explicitement pour éviter tout risque d'oubli. Pour une production donnée, il faut considérer **toutes** les manières de fixer *α*, *β* et *Y* dans une production. Ainsi, pour `E ::= E + T`, on applique le premier cas avec α = ε, Y = `E` et β = `+T`, ce qui entraîne + ∈ *Follow*(`E`), et aussi le troisième cas avec α = `E+` et Y = `T` ce qui entraîne que tout élément de *Follow*(`E`) est dans *Follow*(`T`).

Sur les grammaires G3 et G2 on obtient :

```
G3:
S ::= E $                        Follow(S) = ∅
E ::= E + T | T                  Follow(E) ={$,+,)}
T ::= T * F | F                  Follow(T) ={$,+,*,)}
F ::= ident | '(' E ')'          Follow(F) ={$,+,*,)}

G2:
S ::= E $                      Follow(S) =∅
E ::= T E'                     Follow(E) = {$, )}
E'::= + T E' | ε               Follow(E')= {$, )}
T ::= F T'                     Follow(T) = { +, $, )}
T'::= * F T' | ε               Follow(T')= {$, ), +}
F ::= ident | ( E )            Follow(F) = {*, ), $, +}
```

Seule *Follow* est utilisée dans la suite, mais son calcul nécessite *First*.

Méthode d'analyse syntaxique SLR(1)

Nous présentons maintenant la méthode SLR(1) – pour *Simple Left-to-right parsing, Rightmost derivation with 1 lookahead character*, la variante de la famille des méthodes de type LR la plus simple à expliquer et qui demande le moins de calculs.

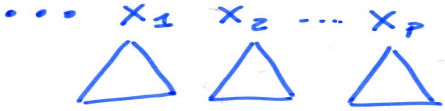
Résumons la situation générale d'un analyseur ascendant. Par principe des dérivations droites, à tout moment de l'analyse on manipule d'une part **une forêt d'arbres**, d'autre part **un suffixe du mot initial** (le début du mot a déjà été reconnu et ses lettres apparaissent aux feuilles des arbres de la forêt). Certains des arbres peuvent être restreints à une feuille et la forêt est gérée en **pile** (avec le sommet de la pile à droite sur le dessin) :

- le premier mouvement possible consiste à **empiler** (*shift*) la lettre courante du mot pour en faire un arbre réduit à sa racine, et on avance dans le mot. Ci-dessous, dans le dessin résumant la situation après empilement, w_i doit être vu comme un arbre réduit à sa racine

Au départ : Pile d'analyse / Mot restant

$w_i w_{i+1} \ldots w_n$

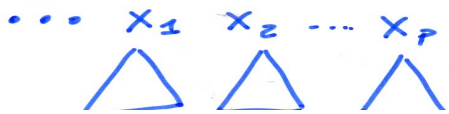
Après empilement :



w_i / $w_{i+1} \ldots w_n$

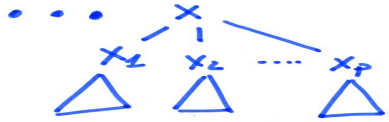
- la seconde possibilité est de dépiler des arbres (dont la séquence des racines forme une **partie droite de production**), à les **enraciner** (les « **réduire** », *reduce* en anglais) sous une nouvelle racine dont l'étiquette est la partie gauche de cette production et à empiler l'arbre ainsi obtenu (ci-dessous, `X ::= X1…Xp` est une production de G). Le mot restant à analyser est sans changement. **Seules les étiquettes des racines importent dans la pile pas les sous-arbres**

Au départ : Pile d'analyse mot restant



/ $w_i w_{i+1} \ldots w_n$

Après ré



/ $w_i w_{i+1} \ldots w_n$

Pour obtenir une analyse déterministe, il faut savoir résoudre deux types de conflits:

- **choisir entre empiler et réduire** quand les deux possibilités existent, c'est-à-dire quand une partie droite de production apparaît en sommet de pile ; Il faut notamment savoir choisir quand faire des réductions par des productions dont la partie droite est ε puisque le mot vide est toujours virtuellement présent en sommet de pile !
- **choisir entre plusieurs réductions possibles** lorsque en sommet de pile on a une partie droite de production à laquelle on peut associer deux parties gauches distinctes (par exemple `N ::= ε` et `M ::= ε`) ou quand une partie droite de production est **suffixe** d'une autre partie droite (cas de `E ::= E+T` et `E := T`, avec `E+T` présent en sommet de pile : réduit-on `E+T` en `E` ou seulement `T` en `E` ?).

Le principe des méthodes LR est d'utiliser les états d'un automate fini pour mémoriser l'avancée dans les différentes parties droites possibles des productions à un moment donné, plutôt que d'avoir à chercher cette information dans la pile. La progression dans une production sera parfois être suspendue (i.e. empiler !) le temps de reconnaître un non-terminal. Lorsqu'une partie droite est obtenue, les caractères d'avance, via la relation *Follow*, permettent de savoir s'il est utile de procéder à une réduction, et si oui à laquelle. Nous présentons la méthode à l'aide d'exemples, en renvoyant le lecteur à un des ouvrages pour une présentation plus algorithmique et systématique de la méthode.

L'automate des items LR(0)

Les méthodes LR utilisent une pile d'analyse qui peut être vue comme contenant des **préfixes de parties droites** de productions. Pour déterminer la prochaine action (empilement ou réduction) sans parcourir la pile, on mémorise le contexte courant dans les états d'un « automate à états finis » (la pile d'analyse est en réalité une **pile d'états** de cet automate); les transitions de l'automate correspondent aux empilements, soit d'un symbole terminal, soit de l'étiquette de la racine d'un arbre enraciné. Un état de cet automate est un ensemble fini d'éléments appelés " items LR(0) " qui symbolisent l'avancée dans la reconnaissance d'une partie droite de production. Dans la suite on utilise un marqueur, ici le point ".", pour séparer la partie déjà reconnue (à gauche du marqueur) de la partie restant à reconnaître (à droite du marqueur) :

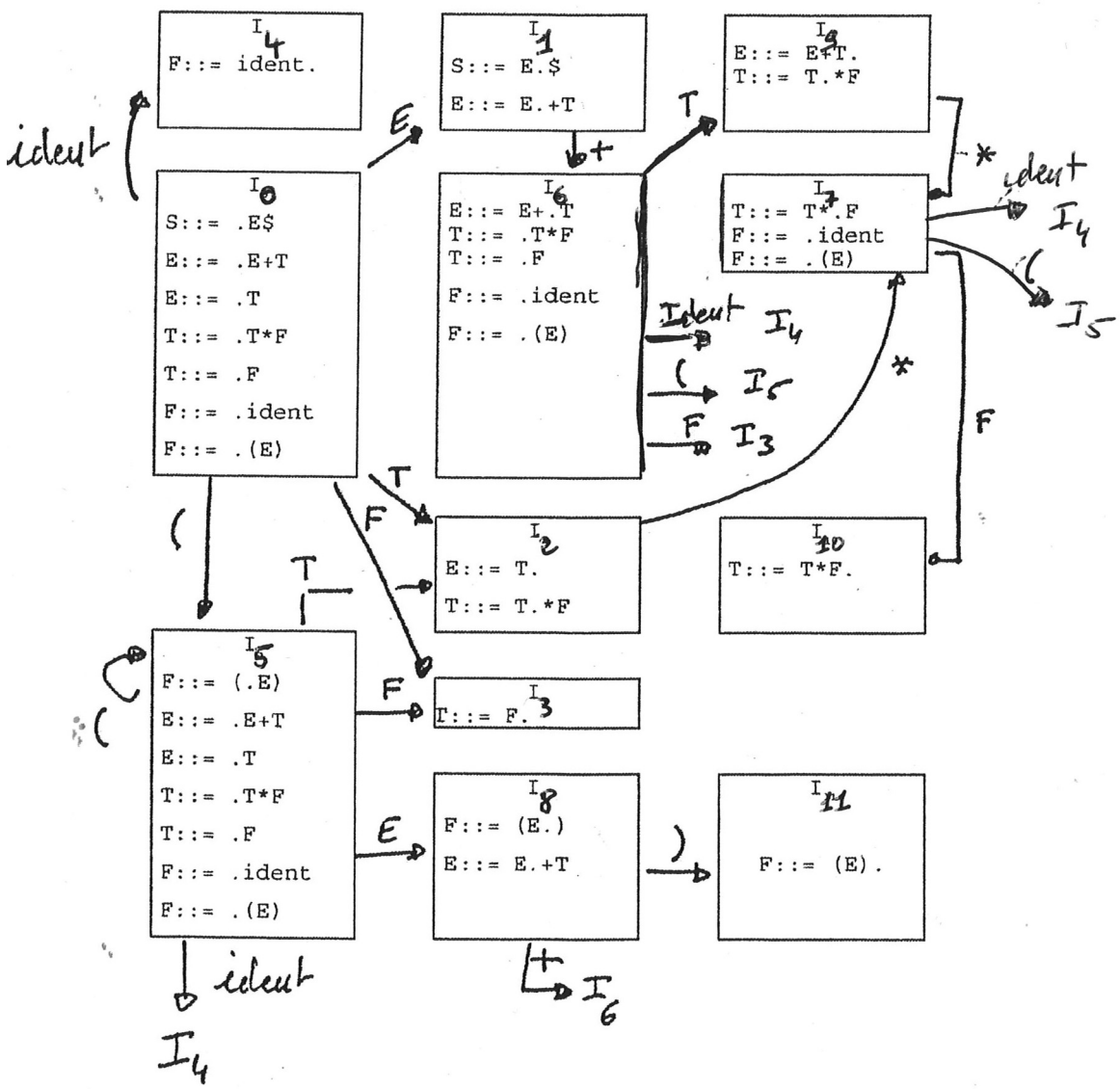
Définition : On appelle **item LR(0)**, un élément de la forme `[X: α . β]` où `X ::= α β` est une règle de grammaire de G.

Remarque : pour une production `X ::= ε`, on ne doit pas distinguer `[X: .ε]` et `[X: ε .]` puisque ε ne représente pas un symbole mais l'absence de symbole. On note `[X: .]` un tel item.

Lors d'une analyse, soit I l'état en sommet de pile : si I contient un item de la forme `[X: α.aβ]`, où *a* est un terminal, on peut empiler `a` si `a` est la lettre courante du mot ; il existe alors pour I une transition étiquetée `a` menant vers un état qui contient l'item `[X: α a.β]`. Si I contient un item de la forme `[X: α.Yβ]`, où `Y` est un non-terminal, c'est qu'on attend une instance de `Y`. Pour empiler `Y` il faut obtenir une partie droite de production dont `Y` est partie gauche, afin que le mécanisme de réduction fasse le travail. Être en attente de `Y`, c'est être en attente de toute partie droite de production dont Y est partie gauche : on ajoute donc dans I un item `[Y: .γ]` pour toute production `Y ::= γ` de G. Une possibilité de réduction est indiquée par la présence dans I d'un item de la forme `[X: δ.]` avec le marqueur en fin de partie droite. La configuration initiale de l'analyse est représentée par une pile initialisée avec l'état contenant l'item `[S': .S$]`, `S'` étant le nouvel axiome, et on se place au début du mot `w$`. Le succès est représenté par la présence en sommet de pile de l'unique état contenant l'item `[S': S.$]` et `$` doit être le caractère courant du mot.

L'automate LR(0) pour la grammaire G3 (`S` est le nouvel axiome) est donné ci-dessous.

Conditions de déterminisme SLR



Les conditions de déterminisme de la méthode SLR se vérifient sur l'automate LR(0) et permettent d'exclure tout conflit lors d'une analyse, à l'aide du caractère d'avance dans le mot : à l'analyse, en cas de possibilité de réduction, le caractère d'avance est utilisé pour vérifier s'il est pertinent de la faire une réduction, et laquelle :

Définition : une grammaire est dite **SLR(1)** si pour tout état I de l'automate LR(0) les deux propriétés suivantes sont vérifiées :

Si I contient un couple d'items `[X: α.aβ]` et `[Y: γ .]` alors $a \notin Follow(Y)$

Si I contient un couple d'items `[Y: γ .]` et `[Z: δ .]` alors $Follow(Y) \cap Follow(Z) = \varnothing$

Exercice *:* Montrer que les grammaires ci-dessous ne sont pas SLR :

```
S ::= L a R | R        S ::= A a A b | B b B a
L ::= b R | b          A ::= ε
R ::= L                B ::= ε
```

En pratique, on construit à partir de l'automate LR(0) et de la relation *Follow* une table qui donne l'action à effectuer pour chaque état et symbole. Si la grammaire est SLR, l'algorithme ci-dessous

associe au plus une action à chaque case. Une case sans action correspond à la détection d'une erreur syntaxique. Dans la table, on note S (*shift*) l'action d'empiler et R (*reduce*) l'action de réduire. On a numéroté les productions de la grammaire pour y faire référence pour les réductions.

> **Définition** : la **table d'actions SLR** est un tableau A indexé par les états de l'automate LR(0) et les symboles de $V_t \cup V_N$. On définit A[I, *a*], où I est un état et *a* un symbole terminal ou non-terminal, de la façon suivante :
> - s'il existe dans I un item `[X: α.aβ]`, A[I, *a*] contient (S, J), où J est la cible de la transition étiquetée par *a* à partir de I
> - pour tout item `[Y: γ .]` de I et *a* de *Follow*(*Y*), A[I, *a*] contient (R, *k*) où *k* est le numéro de la production `Y ::= γ`
> - si I contient l'item `[S': S.$]`, A[I, $] = (succès).

La table d'actions SLR pour `G3` est donnée ci-dessous en numérotant les productions comme suit :

```
S ::= E    $          1
E ::= E + T           2
   | T                3
T ::= T * F           4
   | F                5
F ::= ident           6
   | ( E )            7
```

	ident	(	)	+	*	$	E	T	F
I_0	(S, I_4)	(S, I_5)					(S, I_1)	(S, I_2)	(S, I_3)
I_1				(S, I_6)		(succès)			
I_2			(R, 3)	(R, 3)	(S, I_7)	(R, 3)			
I_3			(R, 5)	(R, 5)	(R, 5)	(R, 5)			
I_4			(R, 6)	(R, 6)	(R, 6)	(R, 6)			
I_5	(S, I_4)	(S, I_5)					(S, I_8)	(S, I_2)	(S, I_3)
I_6	(S, I_4)	(S, I_5)						(S, I_9)	(S, I_3)
I_7	(S, I_4)	(S, I_5)							(S, I_{10})
I_8			(S, I_{11})	(S, I_6)					
I_9			(R, 2)	(R, 2)	(S, I_7)	(R, 2)			
I_{10}			(R, 4)	(R, 4)	(R, 4)	(R, 4)			
I_{11}			(R, 7)	(R, 7)	(R, 7)	(R, 7)			

L'analyseur syntaxique consulte la table d'actions à chaque étape jusqu'à aboutir à la case de la table correspondant au succès, ou à tomber sur une case vide qui correspond à la découverte d'une erreur syntaxique. On rappelle qu'en plus de la table d'actions, l'analyseur utilise une pile pour sauvegarder l'item courant quand on lance la reconnaissance d'un non-terminal. Comme à chaque empilement d'un symbole on empile un état, quand il est temps de faire une réduction pour une partie droite de production, il suffit de dépile autant d'états de la pile que de symboles de la partie droite pour retrouver l'item qui a lancé la reconnaissance de cette partie droite.

L'algorithme ci-dessous détermine si le mot à analyser appartient ou non au langage de la grammaire. En pratique, il faudrait construire l'arbre syntaxique pour le programme, au fur et à mesure des réductions effectuées par l'analyseur syntaxique.

Définition : la **procédure d'analyse SLR** est définie comme suit :

procédure Analyse($w_1...w_n$\$) : bool
-- retourne vrai *si le mot* $w_1...w_n$\$ *appartient au langage engendré par la grammaire à*
-- partir de laquelle la table d'actions a été construite, et faux *sinon.*
-- **A** est la table d'actions de l'analyseur

```
Pile<Etat> P ;  Etat s, z, z'; Entier i ; Char c ;
P := pilevide; empiler(P, I0 ) ;          -- I0 est l'état initial de l'automate
i := 1; c := w1;                          -- initialiser c avec la première lettre du mot
répéter
s := sommet(P);                           -- récupérer l'état courant de l'automate
si A[s, c] = (S, z')
   empiler(P, z'); i := i + 1; c := wi;   -- empiler le nouvel état et avancer dans le mot
sinon si A[s, c] = (R, k)
     -- supposons que la k ème règle est Y ::= δ
     -- dépiler | δ | symboles de la pile. Soit z le nouveau sommet de pile, alors
     -- A[z, Y] est forcément de la forme (S, z') ; empiler z'.
     depiler| δ |(P); z := sommet(P); z' := A[z, Y]; empiler(P, z');
sinon si A[s, c] = (succès) renvoyer(vrai);
sinon renvoyer(faux);
fin
```

Exercices :

1. Donner la trace de l'analyse syntaxique du mot `id + id * id $`
2. Indiquez l'état de l'analyse lors de l'échec de la reconnaissance du mot `id * id id`.

Autres méthodes de type LR

Dans la méthode SLR, *Follow* sert pour remplir la table d'actions : pour un état dans lequel une réduction est possible, on ajoute la réduction tous les caractères du *Follow* de la partie gauche. La liste de ces caractères ne dépend pas de l'état dans lequel la réduction intervient. Dans la méthode LR(1), le caractère d'avance fait partie des items qui sont de la forme `[X ::=` $\alpha.\beta$, a], où a est un élément de *Follow*(*X*). L'item indique que la réduction ne sera à faire, une fois la partie droite $\alpha\beta$ obtenue, que si le caractère d'avance est a, et non plus pour tout caractère de *Follow*(X). Pour chaque état on calcule précisément les caractères d'avance pertinents pour une réduction. La construction de la table d'actions est similaire à celle poursle cas SLR et la procédure qui exploite la table est identique.

Exemple : la grammaire suivante n'est pas SLR, à cause d'un conflit *reduce-reduce* entre `A::= ε` et `B::=ε` dans l'état initial pour `a` et `b`. Pourtant, dans l'état initial si le caractère d'avance est `'a'` seule la réduction par `A ::= ε` est utile. Le caractère `'b'` du *Follow*(`A`) n'est utile que pour la réduction de ε en `A` du deuxième `A`. De même pour `'b'` et la réduction par `B ::= ε`.

```
S ::= A a A b | B b B a
A ::= ε
B ::= ε
```

Dans la méthode LR(1), les items sont amorcés par `[S' ::= .S$, ]`[1] et conduisent à l'automate ci-dessous, sans conflit :

```
I0 [S' ::= .S$,  ]        I1 [S' ::= S.$, ]
       [S  ::= .AaAb, $]
```

[1] Il n'y a pas de caractère d'avance pour cet item puisqu'il n'y a pas de réduction par `S' ::= S $ .`

```
            [S  ::= .BbBa, $]      I2 [S::= A.aAb, $]
            [A  ::= ., a]                                  I3 [S::=
B.bBa, $]
            [B  ::= ., b]

                                I4  [S ::= Aa.Ab, $]       I5 [S ::=
Bb.Ba, $]
                                    [A ::= ., b]               [B ::= .,
a]

                                I6  [S ::=AaA.b, $]        I7
[S ::=BbB.a, $]

                                I8  [S ::= AaAb., $]       I9
[S ::=BbBa., $]
```

Sur cet exemple les deux méthodes produisent un automate avec le même nombre d'états ; la méthode SLR indique un conflit dans l'état initial puisque *Follow*(A) = *Follow*(B) = { a, b }. Le calcul précis des caractères d'avance des items LR(1) déduit que dans l'état initial, pour le caractère d'avance a, seule la réduction par A ::= ε est utile, et pour b, seule B ::= ε importe, évitant ainsi tout conflit. La méthode LR(1) évite des réductions inutiles là où la méthode SLR peut faire des réductions inutiles avant de se bloquer par absence de transition dans l'automate.

L'automate LR(1) a en général (beaucoup) plus d'états que l'automate LR(0) correspondant, puisque à un état LR(0) peuvent correspondre plusieurs états LR(1) qui ne se distinguent que par les seconds composants de leurs items : la grammaire { S ::= AaAb, A ::= c } engendre l'unique mot cacb. Son automate LR(1) est le suivant :

```
I0  [S': .S$,  ]            I1 [S: S.$, ]
    [S: .AaAb, $]
    [A: .c, a]              I3 [A: c., a]


I2 [S: A.aAb, $]            I4 [S: Aa.Ab, $]          I5 [A: c., b]
                               [A: .c, b]


I6 [S: AaA.b, $]            I7  [S: AaAb., $]
```

On distingue les états [A ::= c., a] et [A ::= c., b] pour différencier les deux contextes de réduction par A ::= c, là où l'automate LR(0) n'a qu'un état. Plus le calcul est précis, plus la méthode accepte de grammaires sans conflits, mais plus le nombre d'états de l'automate est grand.

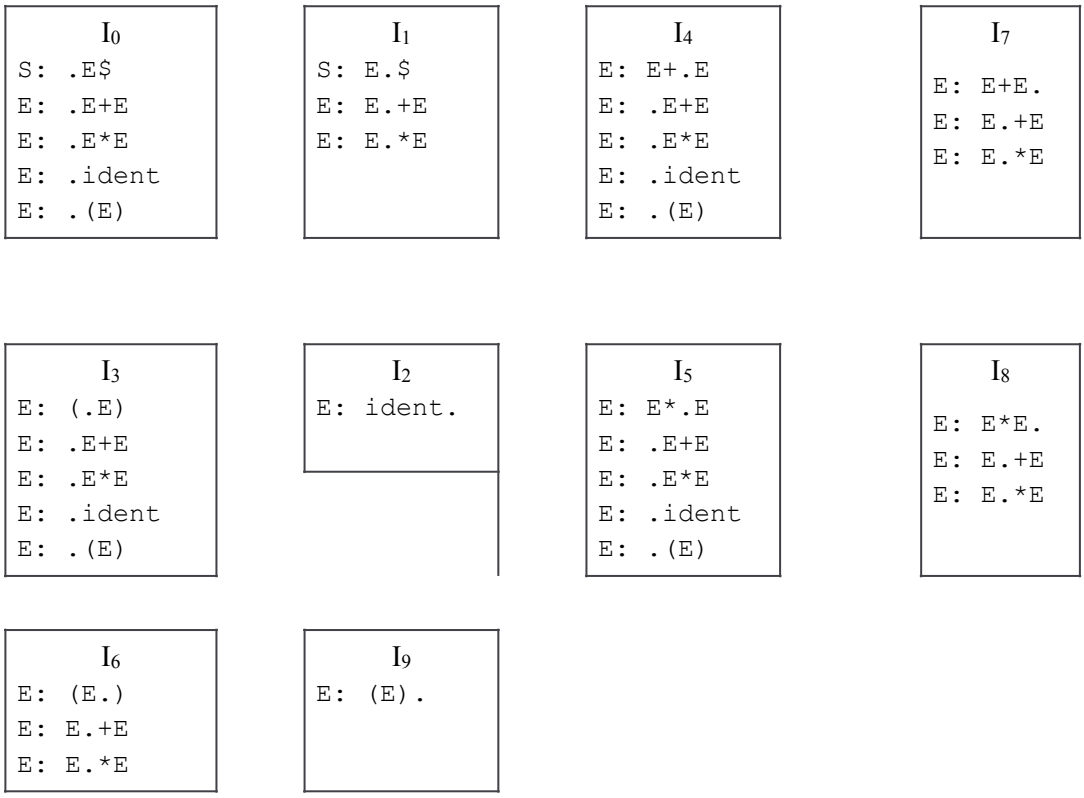
Gestion des conflits liés à la précédence et à l'associativité des opérateurs

Lorsqu'une grammaire ne vérifie pas les conditions de déterminisme, il n'y a pas a priori d'autre solution qu'essayer de trouver une grammaire équivalente avec les bonnes propriétés. Quand le conflit est lié à un problème limité d'ambiguïté dans la grammaire, on peut essayer de procéder autrement en examinant chaque conflit dans la table pour décider **statiquement** (*ie* à la construction de la table) d'en privilégier une, pour retomber sur une table déterministe. Puisqu'on modifie la table d'actions, en général, l'analyseur ne reconnaît plus tout le langage engendré par la grammaire de départ ; lorsque les conflits sont liés à l'absence dans la grammaire des précédences ou associativités d'opérateurs, le langage reconnu peut rester le même, la suppression d'actions dans la table revenant à forcer une interprétation pour chaque expression ambiguë. Le choix de l'action à conserver doit être effectué en examinant les contextes possibles pour la pile d'analyse. Cette manière de procéder ne doit être utilisée **qu'à bon escient**, en contrôlant explicitement son effet sur les piles d'analyse possibles.

Lorsque le procédé peut être appliqué, il a l'avantage de permettre de travailler avec une grammaire plus concise et plus naturelle qu'une grammaire qui serait SLR stricto sensu (on peut par exemple comparer les grammaires G1 et G3). Nous illustrons ce procédé à l'aide de la grammaire G1.

```
S ::= E $
E ::= E + E                    First(E)  = { ident, ( }
E ::= E * E                    Follow(E) = { $, +,
      *, ) }
E ::= ident
```

L'automate LR(0) est donné ci-dessous. La grammaire n'est pas SLR : l'état I_7 présente deux conflit " shift/reduce " sur les symboles + et * ; il en est de même que I8.



Analysons par exemple les conflits de l'état I_7 : par principe de construction de l'automate LR(0), une pile SLR avec I_7 au sommet représente une pile d'analyse de la forme ... `E+E` (puisque I_7 contient l'item `E+E.)` où ... représente ce qui précède `E+E` dans la pile et auquel nous n'avons pas accès tant que ce qui précède n'aura pas été dépilé.

- conflit shit-reduce sur + : le mot *w* à considérer est donc de la forme *+ w'*.

Si on empile le symbole le `'+'`, le couple (pile, mot) devient (... `E+E+`, *w'*). Si *w* est correct, le + doit être suivi d'une sous-expression ; à un instant ultérieur, le contexte aura la forme (... `E+E+E`, *w''*). Les réductions se faisant de la droite vers la gauche, les deux `E` de droite seront réduits en `E`, puis le `E` de gauche sera combiné avec cet arbre. L'opérateur + associerait donc à droite.

Si au contraire on réduit par `E ::= E+E`. Le couple (pile, mot) devient (... `E,` *+w'*) qui évoluera, comme précédemment vers (... `E+E`, *w''*) et la réduction pourra alors s'effectuer. L'opérateur + associerait dans ce cas à gauche.

L'opérateur + associant à gauche, en I_7 on privilégie la réduction sur l'empilement pour `'+'`.

- conflit shit-reduce sur `*` : le mot *w* est de la forme **w'*. Selon le même raisonnement que précédemment, on note qu'une réduction donnerait une plus forte précédence à + par rapport à `*`, tandis qu'un empilement ferait le choix inverse : en I_7 quand le caractère d'avance est * il faut privilégier l'empilement ;

Un examen des conflits dans l'état I_8 montre qu'avec le caractère d'avance +, il faut privilégier la réduction par `E ::= E*E` pour respecter la précédence de `*` par rapport à +. Quand le caractère d'avance est `*`, il faut aussi choisir la réduction qui donnera un opérateur `*` associant à gauche, par rapport à l'empilement qui donnerait un opérateur associant à droite.

Les générateurs d'analyseurs syntaxiques

Comme on a pu le voir dans les pages précédentes, l'analyse syntaxique SLR consiste à calculer des relations pour vérifier si les conditions de déterminisme sont satisfaites. Si ce n'est pas le cas, la méthode n'est pas applicable et la grammaire doit être transformée, en espérant que la grammaire résultat aura les propriétés attendues (il est indécidable de savoir s'il existe une grammaire SLR équivalente à une grammaire donnée !). Si les conditions sont satisfaites, la méthode peut être encodée par la table d'actions qui dirige l'analyseur syntaxique qui devient une simple procédure d'exploitation de la table d'actions selon le caractère d'avance. L'ensemble de ces opérations peut être automatisée, (sauf trouver une grammaire SLR équivalente à une grammaire qui ne le serait pas!) et il est possible de réaliser des " générateurs d'analyseurs syntaxiques " qui fournissent un analyseur à partir d'une description de la grammaire. Il existe de nombreux générateurs d'analyseurs, parmi lesquels nous pouvons citer le couple FLEX+BISON qui engendre des analyseurs LaLR (une extension de SLR) écrits sous forme de programmes C, C++ OU JAVA. Dans le monde OCAML le couple associé est OCAMLLEX+MENHIR que nous utiliserons en projet. L'outil MENHIR implémente la méthode LR(1).

Dans tous les cas, l'analyseur se comporte par défaut comme un simple reconnaisseur, c'est-à-dire une procédure qui décide si le mot en entrée appartient ou non au langage associé à l'analyseur. Pour obtenir un véritable analyseur syntaxique il faut ajouter plusieurs autres composants :

- une interface avec un analyseur lexical, qui fournira à la demande les unités lexicales (le " caractère d'avance ") ; à chaque fois que l'analyseur syntaxique consomme l'unité courante, il fait appel à l'analyseur lexical pour obtenir l'unité suivante ;
- un moyen d'effectuer des traitements aux productions de la grammaire: on ne se contente pas d'un reconnaisseur, mais on veut obtenir en sortie un arbre syntaxique ou toute structure équivalente utile pour la suite de la compilation. Les générateurs offrent la possibilité d'attacher des fragments de code aux productions de la grammaire, par exemple pour construire un AST. Ces fragments de code seront exécutés automatiquement par l'analyseur syntaxique au fur et à mesure des **réductions**, aboutissant à une construction ascendante de l'AST;

- un moyen de récupérer des erreurs syntaxiques : il ne suffit pas d'avoir un analyseur qui reconnaisse les mots corrects mais on souhaite aussi des messages d'erreurs pertinents et des possibilités de récupérer ces erreurs pour poursuivre la suite de l'analyse. Les systèmes varient fortement selon les mécanismes qu'ils offrent au concepteur pour contrôler la récupération d'erreurs. Un des moyens consiste à ajouter une unité lexicographique spéciale " erreur " qui est insérée automatiquement par le système lorsqu'une erreur syntaxique est détectée ; le concepteur doit alors ajouter des productions incluant cette unité « d'erreur » pour spécifier les récupérations spécifiques qu'il veut traiter, un mécanisme par défaut relativement brutal (élimination successive des unités lexicographiques jusqu'à retomber en terrain connu) traitant les autres cas. L'outil MENHIR offre des mécanismes particulièrement efficaces dans ce domaine, mais nous n'aurons pas le temps de les expliciter dans ce cours.