

Introduction à la compilation

Cours 3 : Analyse syntaxique descendante

Yann Régis-Gianas
`yrg@pps.jussieu.fr`

PPS - Université Denis Diderot – Paris 7

vendredi 12 février 2010

Rappels sur l'analyse descendante

- Soit la grammaire (tirée de *Parsing Techniques* de Grune et Jacobs) :

(1)		$S \rightarrow aSQ$
(2)		$S \rightarrow abc$
(3)		$bQc \rightarrow bbcc$
(4)		$cQ \rightarrow Qc$

et l'entrée : « *aabbcc* »

⑤

- L'analyse débute avec celle du symbole S_s .
- Quelle peut être la prochaine étape ?

Rappels sur l'analyse descendante

- Soit la grammaire (tirée de *Parsing Techniques* de Grune et Jacobs) :

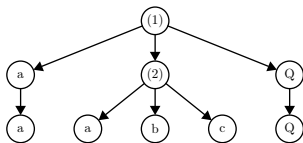
$$\begin{array}{lcl} (1) & \parallel & S \rightarrow aSQ \\ (2) & \parallel & S \rightarrow \end{array}$$

Rapports sur l'analyse descendante

- Soit la grammaire (tirée de *Parsing Techniques* de Grune et Jacobs) :

(1)	$S \rightarrow aSQ$
(2)	$S \rightarrow abc$
(3)	$bQc \rightarrow bbcc$
(4)	$cQ \rightarrow Qc$

et l'entrée : « *aabbcc* »



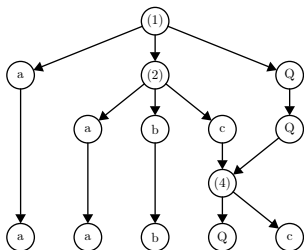
- Appliquer de nouveau la règle (1) imposerait le préfixe « *aaa* » dans l'entrée.
- On doit donc considérer la règle (2).

Rappels sur l'analyse descendante

- Soit la grammaire (tirée de *Parsing Techniques* de Grune et Jacobs) :

(1)	$S \rightarrow aSQ$
(2)	$S \rightarrow abc$
(3)	$bQc \rightarrow bbcc$
(4)	$cQ \rightarrow Qc$

et l'entrée : « *aabbcc* »



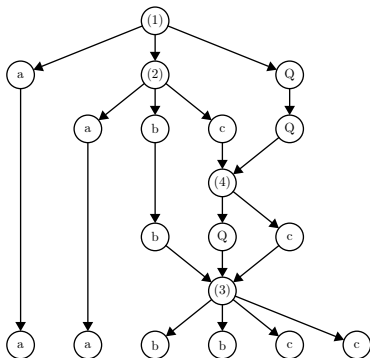
- Seule la règle (4) s'applique.
(C'était difficile à prévoir !)

Rappels sur l'analyse descendante

- Soit la grammaire (tirée de *Parsing Techniques* de Grune et Jacobs) :

(1)	$S \rightarrow aSQ$
(2)	$S \rightarrow abc$
(3)	$bQc \rightarrow bbcc$
(4)	$cQ \rightarrow Qc$

et l'entrée : « *aabbcc* »



- Ici, seule la règle (3) s'applique et elle mène à l'entrée.

Algorithmme de Unger

Principe

- ▶ Soit G , une grammaire **hors-contexte**, sans règle ϵ et sans boucle.
 - ▶ Une façon naturelle (et naïve) de déterminer si une entrée $c_1 \dots c_n$ peut être produite par un non-terminal S consiste à :
 - ▶ **énumérer** l'ensemble des règles de G qui définissent S ;
 - ▶ Pour chaque règle de la forme « $S \rightarrow A_1 \dots A_m$ », pour **toutes les partitions** $P = P_1 \dots P_m$ de l'entrée en **m parties**, on détermine (par récursion) si A_i peut produire P_i .
- ⇒ Une application standard de la méthode « diviser pour régner ».

Exercice

1. Quelle est la complexité de cette méthode de recherche ?
2. Est-ce que cet algorithme termine ?

(réponses dans la suite)

Grammaire d'exemple

- On s'intéresse à la grammaire suivante :

<i>exp</i>	$::=$	<i>exp</i> + <i>term</i>
		<i>term</i>
<i>term</i>	$::=$	<i>term</i> * <i>factor</i>
		<i>factor</i>
<i>factor</i>	$::=$	0 1 ...
		(<i>exp</i>)

Calculer toutes les partitions de N objets en M parties?

Exercise

Comment calculer toutes les partitions de N objets en M parties (non vides) ?

```

let rec partition m n l :  $\alpha$  partition list =
  if n < m then []
  else
    match m, l with
    | 0, []
      | [ [] ]
    | 1, l
      | [ [ l ] ]
    | m, l
      | List.flatten
        (repeat 1 n
          (fun k
            let (picked, l) = pick k l in
            List.map (fun p
              picked : : p) (partition (m - 1) (n - k) l)))

```

Implémentation (naïve et fausse) de Unger

```
let rec parse g s input : bool =  
  let match_rhs (_, rhs) =  
    let ps = partition (List.length rhs) input in  
    let subinput_matches_symbol symbol input =  
      match symbol, input with  
      | Terminal t, [u]   |> t = u  
      | NonTerminal s, _   |> parse g s input  
      | _                  |> false  
    in  
    let partition_matches_rhs p =  
      List.for_all2 subinput_matches_symbol rhs p  
    in  
    exists partition_matches_rhs ps  
  in  
  exists match_rhs (rules_about g s)
```

Exercice

1. Quelle est la complexité de cette méthode de recherche ?
2. Est-ce que cet algorithme termine ?

Règles de production vide

- Pour traiter les règles de production vide, on doit considérer les parties de taille 0 : il suffit de modifier l'algorithme qui énumère les partitions.

Exercice

Comment calculer toutes les partitions de N objets en M parties (possiblement vides) ?

```
let rec partition m n l :  $\alpha$  partition list =  
  match m, l with  
  | 0, []  
  | [ [] ]  
  | 1, l  
  | [ [ l ] ]  
  | m, l  
  |> List.flatten  
    (repeat 0 n  
      (fun k  
        let (picked, l) = pick k l in  
        List.map (fun p    picked : : p) (partition (m - 1) (n - k) l)))
```

Règles de production vide

- ▶ Par exemple, si on étend la grammaire à l'aide des règles :

$factor$	\rightarrow	$INT\ plusplus$
$plusplus$	\rightarrow	ϵ
$plusplus$	\rightarrow	$BANG\ plusplus$

- ▶ L'algorithme boucle ! Pourquoi ?

Observation sur la trace d'exécution

- ▶ Le programme se pose sans cesse les mêmes questions.
 - ▶ Si on cherche une dérivation pour « $1!$ » à partir de « exp » alors une des voies explorées par l'algorithme envisage la possibilité que cette entrée soit produite par la règle « $exp \rightarrow exp + term$ ». En étudiant la partition « $\epsilon|1!$ », on doit résoudre le problème de la génération de « ϵ » par « exp ».
- ⇒ Ce dernier problème se réduit immédiatement sur lui-même, ce qui provoque la non terminaison de l'algorithme !

Se donner une mémoire

- ▶ Une solution très simple – et dans cette situation, très efficace – consiste à se souvenir des réponses issues des analyses déjà effectuées.
- ▶ Quand on se pose la question :
« Est-ce que le non-terminal E peut produire w ? »
on vérifie d'abord dans une table si on ne s'est pas déjà posé cette question.
- ▶ Il y a alors trois cas possibles :

1. On ne s'est jamais posé la question :
Il faut faire le calcul et enregistrer le résultat dans la table.
2. On s'est déjà posé cette question :
On peut renvoyer le résultat enregistré dans la table.
3. On est déjà en train de se poser cette question :
Cela signifie qu'il y a une boucle : on peut s'arrêter et répondre que cette branche de recherche ne fournit pas de solution.

Se donner une mémoire en OCaml

```
type answer =  
  | BeingAnswered  
  | Answered of bool  
  
let h = Hashtbl.create 13 in  
  
let rec result_from_memory (s, input) =  
  try match Hashtbl.find h (s, input) with  
    | BeingAnswered → false  
    | Answered b → b  
  with Not_found →  
    Hashtbl.add h (s, input) BeingAnswered;  
    let y = traverse (s, input) in  
    Hashtbl.add h (s, input) (Answered y);  
    y
```

Algorithme de Unger avec mémoire

```
and traverse (s, input) : bool =  
  let rules = rules_about g s in  
  let match_rhs ((_, (rhs : ('t, 'n) rhs))) =  
    let m = List.length rhs in  
    let ps = partition m input in  
    let subinput_matches_symbol symbol input =  
      match symbol, input with  
      | Terminal t, [u]   t = u  
      | NonTerminal s, _   result_from_memory (s, input) (* Changement. *)  
      | _                  false  
    in  
    let partition_matches_rhs p =  
      List.for_all2 subinput_matches_symbol rhs p  
    in  
    exists partition_matches_rhs ps  
  in  
  exists match_rhs rules  
in  
result_from_memory (s, input)
```

Procédé de « memoization »

- ▶ La méthode que nous venons d'employer s'apparente à de la **programmation dynamique** dans le sens où on réutilise les résultats de **sous-problèmes partagés** entre plusieurs problèmes pour factoriser les calculs.
- ▶ Cependant, il ne faut pas perdre de vue qu'on utilise un important espace mémoire.
- ▶ On peut écrire cette fonction en toute généralité de la façon suivante :

```
exception Loop

type  $\alpha$  answer =
  BeingAnswered
  Answered of  $\alpha$ 

let rec memoize f =
  let h = Hashtbl.create 13 in
  fun x
    try match Hashtbl.find h x with
      BeingAnswered   raise Loop
      Answered b      b
    with Not_found
      Hashtbl.add h x BeingAnswered ;
      let y = f x in
      Hashtbl.add h x (Answered y) ;
      y
```

Inefficacité de l'algorithme de Unger

- ▶ L'algorithme de Unger se pose **beaucoup** de questions.
- ▶ Par exemple, pour déterminer si « $exp \rightarrow^* (42)$ » :

```
expr  expr + term / ( INT )
expr  expr + term /
expr  term /
term  term factor /
term  factor /
factor ( expr ) /
factor INT /
expr  expr + term / (
expr  term / (
term  term factor / (
term  factor / (
factor ( expr ) / (
factor INT / (
expr  expr + term / ( INT
expr  term / ( INT
```

```
term  term factor / ( INT
term  factor / ( INT
factor ( expr ) / ( INT
expr  expr + term / INT
expr  term / INT
term  term factor / INT
term  factor / INT
factor ( expr ) / INT
factor INT / INT
factor INT / ( INT
expr  term / ( INT )
term  term factor / ( INT )
term  factor / ( INT )
factor ( expr ) / ( INT )
```

(Ce qui se trouve après le symbole « / » correspond à la partie d'entrée considérée.)

Incapacité de l'algorithme de Unger

- ▶ Par une « simple » observation du premier caractère de l'entrée, on peut **prévoir** que l'étude de certaines règles ne va pas aboutir.
- ▶ Par exemple, en considérant la taille minimale d'un mot produit par le côté droit d'une règle, on déduit que les règles suivantes ne peuvent pas produire l'entrée associée :

expr	expr + term / (INT)
✓ expr	expr + term /
✓ expr	term /
✓ term	term factor /
✓ term	factor /
✓ factor	(expr) /
✓ factor	INT /
✓ expr	expr + term / (
expr	term / (
✓ term	term factor / (
term	factor / (
factor	(expr) / (
factor	INT / (
✓ expr	expr + term / (INT
expr	term / (INT

✓ term	term factor / (INT
term	factor / (INT
✓ factor	(expr) / (INT
✓ expr	expr + term / INT
expr	term / INT
✓ term	term factor / INT
term	factor / INT
✓ factor	(expr) / INT
factor	INT / INT
✓ factor	INT / (INT
expr	term / (INT)
term	term factor / (INT)
term	factor / (INT)
factor	(expr) / (INT)

Analyse descendante prédictive

Analyse prédictive

- ▶ L'algorithme de Unger est **non directionnel**.
 - ▶ Il se donne la possibilité de lire l'entrée plusieurs fois et d'une façon arbitraire.
- ⇒ Il faut donc avoir sous la main l'entrée dans sa totalité.
- ▶ Nous allons nous restreindre à une lecture de l'entrée de **gauche à droite**.
 - ▶ Dans ce cadre, on peut modéliser l'**analyse prédictive** ainsi :

Reste de l'entrée ...
Prédiction ...

- ▶ Le reste de l'entrée est formé de terminaux.
- ▶ La prédiction est une phrase intermédiaire contenant des symboles terminaux et non-terminaux.

Analyse prédictive : exemple

- Soit la grammaire :

$$\begin{array}{lcl} S & ::= & aB \mid bA \\ A & ::= & a \mid aS \mid bAA \\ B & ::= & b \mid bS \mid aBB \end{array}$$

Exercice

Quel est le langage reconnu par cette grammaire ?

Analyse prédictive : exemple

$$\begin{aligned} S &::= aB \mid bA \\ A &::= a \mid aS \mid bAA \\ B &::= b \mid bS \mid aBB \end{aligned}$$

a a b b
S

- Essayons de calculer la dérivation de « aabb » par S .

Analyse prédictive : exemple

$$\begin{aligned} S &::= aB \mid bA \\ A &::= a \mid aS \mid bAA \\ B &::= b \mid bS \mid aBB \end{aligned}$$

a a b b
S

- ▶ On doit remplacer S par l'un de ses membres droits.
- ▶ Par observation du premier lexème "a", seule la première règle est applicable.

Analyse prédictive : exemple

$$\begin{aligned} S &::= aB \mid bA \\ A &::= a \mid aS \mid bAA \\ B &::= b \mid bS \mid aBB \end{aligned}$$

a	a b b
a B	

- ▶ La prédiction et l'entrée commence par le même symbole.
- ▶ On **accepte** le terminal "a".

Analyse prédictive : exemple

$$\begin{aligned} S &::= aB \mid bA \\ A &::= a \mid aS \mid bAA \\ B &::= b \mid bS \mid aBB \end{aligned}$$

a	a	b b
a	B	

- ▶ Le premier symbole de la prédiction est un non-terminal.
- ▶ Trois possibilités mais une seule est compatible avec le lexème "a".

Analyse prédictive : exemple

$$\begin{aligned} S &::= aB \mid bA \\ A &::= a \mid aS \mid bAA \\ B &::= b \mid bS \mid aBB \end{aligned}$$

a	a	b b
a	a	B B

- On accepte de nouveau le lexème “a”.

Analyse prédictive : exemple

$$\begin{aligned} S &::= aB \mid bA \\ A &::= a \mid aS \mid bAA \\ B &::= b \mid bS \mid aBB \end{aligned}$$

a a	b	b
a a	B	B

- ▶ Par observation du premier lexème de l'entrée, il reste deux choix.
- ▶ Cependant, si on choisit « bS », on aurions au moins un nouveau “a” à reconnaître. Or, la suite de l'entrée ne contient plus de “a”.

Analyse prédictive : exemple

$$\begin{aligned} S &::= aB \mid bA \\ A &::= a \mid aS \mid bAA \\ B &::= b \mid bS \mid aBB \end{aligned}$$

a a	b	b
a a	b	B

- On accepte le lexème “b”.

Analyse prédictive : exemple

$$\begin{aligned} S &::= aB \mid bA \\ A &::= a \mid aS \mid bAA \\ B &::= b \mid bS \mid aBB \end{aligned}$$

a a b	b
a a b	B

- On doit encore appliquer première règle de B.

Analyse prédictive : exemple

$$\begin{aligned} S &::= aB \mid bA \\ A &::= a \mid aS \mid bAA \\ B &::= b \mid bS \mid aBB \end{aligned}$$

a a b b	
a a b b	

- ▶ On accepte finalement le dernier lexème “b”.
- ⇒ La prédiction est vide, tout comme l’entrée : l’analyse est un succès.
- ▶ La dérivation est :

$$S \rightarrow aB \rightarrow aaBB \rightarrow aabB \rightarrow aabb$$

Un modèle de calcul : l'automate à pile

- ▶ Les deux actions de l'algorithme d'analyse prédictif s'apparentent aux deux mécanismes sur lesquels s'appuie les **automates à pile**.
- ▶ Un automate à pile est une machine qui, en fonction d'un symbole d'entrée et du sommet de sa pile, empile ou dépile des symboles sur cette dernière.
- ▶ Un automate à pile **accepte** une séquence de symboles si elle mène à la pile vide.
- ▶ Il peut y avoir plusieurs choix d'action à effectuer pour un même symbole d'entrée. Dans ce cas, on parle d'automate déterministe.

Définition formelle des automates à pile utilisés ici

- ▶ Nous allons utiliser la formulation suivante de la définition des automates à pile.
- ▶ Un automate suit un alphabet d'entrée (les terminaux) et un alphabet de pile (les non-terminaux et les terminaux) est la donnée de règle de la forme :

$$(i, s) \rightarrow s_1 \dots s_n$$

où i est soit un terminal, soit vide.

s est un terminal ou un non-terminal.

s_i est un terminal ou un non-terminal pour tout i .

Exemple d'automates à piles

- Le langage de la grammaire :

$$\begin{aligned} S &::= aB \mid bA \\ A &::= a \mid aS \mid bAA \\ B &::= b \mid bS \mid aBB \end{aligned}$$

- est reconnu par l'automate à pile **non déterministe** :

$$\begin{aligned} (, S) &\rightarrow aB \\ (, S) &\rightarrow bA \\ (, A) &\rightarrow a \\ (, A) &\rightarrow aS \\ (, A) &\rightarrow bAA \\ (, B) &\rightarrow b \\ (, B) &\rightarrow bS \\ (, B) &\rightarrow aBB \\ (a, a) &\rightarrow \\ (b, b) &\rightarrow \end{aligned}$$

Environnement d'évaluation de l'automate à pile

- ▶ Un automate à pile ne nécessite que l'état de sa pile et son entrée pour fonctionner.
- ▶ Toutefois, pour mieux comprendre les algorithmes d'analyse syntaxique les utilisant, nous allons nous donner un environnement d'évaluation plus riche, qui se souvient des analyses déjà effectuées par l'automate.
- ▶ Pour cela, on définit une **description instantanée** par un tableau :

Entrée déjà analysée	Entrée restante
Règles utilisées	Prédictions

- ▶ La seconde ligne de ce tableau peut contenir plusieurs couples (analyse, prédiction) concurrents.
- ▶ On introduit un marqueur final '#' à la fin de l'entrée et de la prédiction.
- ▶ Ainsi, une description instantanée qui accepte l'entrée correspond une acceptation de '#'.
- ▶ Pas de cas particulier pour vérifier le succès à la fin de l'analyse.

Domaine de recherche des dérivations

- ▶ Le domaine de recherche des dérivations peut donc se modéliser par un arbre dont les nœuds sont les descriptions instantanées et chaque arête correspond à un choix d'expansion d'une certaine règle d'un non-terminal.
- ▶ On choisit d'en faire un parcours **en profondeur** ou un parcours **en largeur**.

Exercice

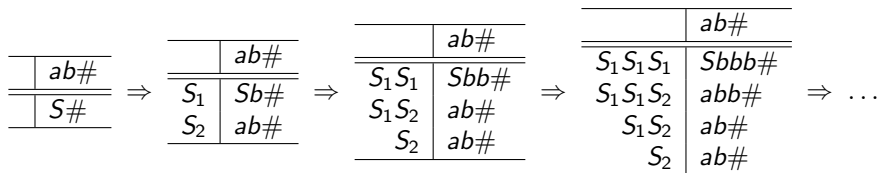
Est-ce que cet arbre est **fini** ? À **branchement fini** ?

Un cas de non-terminaison

- Considérons la grammaire :

$$S ::= Sb \mid a$$

- Sur l'entrée « ab », on s'engage dans une **séquence infinie de prédictions** :



Problème de la récursion à gauche

- ▶ Un non-terminal qui peut dériver une phrase commençant par lui-même est **récuratif gauche**.
 - ▶ Il y a deux sortes de récursion gauche, la **récursion gauche immédiate (ou directe)** d'un non-terminal A apparaît lorsqu'une règle de A possède un membre droit débutant par A .
 - ▶ Un non-terminal récuratif gauche qui n'est pas récuratif gauche de façon immédiate, l'est de façon **indirect**.
- ⇒ Bonne nouvelle : on peut toujours **supprimer la récursion gauche**, c'est-à-dire transformer une grammaire possédant des non-terminaux récuratifs à gauche en une grammaire équivalente non réursive gauche.

Élimination de la récursion gauche

Élimination de la récursion directe

- ▶ Nous allons voir une méthode de suppression de la récursion immédiate à gauche qui suppose que la grammaire considérée ne contient pas de règles de la forme :
 - ▶ « A » : règle de production vide.
 - ▶ « $A \rightarrow B$ » : règle unitaire.
où les symboles A et B sont des non-terminaux.
- ⇒ Heureusement, il est aussi possible de transformer toute grammaire avec de telles règles en une grammaire équivalente sans règles unitaires ou de production vide.

Élimination des règles de production vide

- Soit une grammaire G contenant une règle de la forme :

$$A \rightarrow \epsilon$$

- Pour toute règle de la forme :

$$B \rightarrow \alpha A \beta$$

- On peut rajouter la règle suivante dans la grammaire :

$$B \rightarrow \alpha \beta$$

- On introduit alors un non-terminal A avec les mêmes règles que A sauf celles de production vide. On peut remplacer toutes les occurrences de A par A dans les règles de la grammaire.

Terminaison ?

- ▶ Le processus suivant peut introduire de nouvelles règles de production vide.
- ▶ On doit donc **itérer** le processus pour supprimer toutes ces règles.

Exercice

Est-ce que cet algorithme termine ?

Terminaison de la suppression des règles de production vide

- ▶ Le nombre de non-terminaux de la grammaire qui produisent ϵ est fini.
- ▶ Ce nombre décroît à chaque itération.

Élimination des règles unitaires

- Soit une grammaire G possédant une règle :

$$A \rightarrow B$$

où A et B sont des non-terminaux.

- Supposons que B soit défini ainsi :

$$B \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$$

- On peut mettre en ligne la définition de B en rajoutant la règle :

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$$

Terminaison ?

- ▶ Encore une fois, cette transformation de grammaires peut introduire de nouvelles règles unitaires.

Exercice

Est-ce que cet algorithme termine ?

Terminaison de l'élimination des règles unitaires

- ▶ On peut retomber sur la règle « $A \rightarrow B$ ».
- ▶ Ce cas témoigne de la présence de dérivations infinies.
- ▶ On peut, sans modifier le langage, supprimer cette nouvelle occurrence de la règle « $A \rightarrow B$ ».

Élimination des règles unitaires et de production vide :

Exemple

<i>exp</i>	$::=$	<i>exp</i> + <i>term</i>
		<i>term</i>
<i>term</i>	$::=$	<i>term</i> * <i>factor</i>
		<i>factor</i>
<i>factor</i>	$::=$	<i>int plusplus</i>
		(<i>exp</i>)
<i>plusplus</i>	$::=$	ϵ
		! <i>plusplus</i>

Élimination des règles unitaires et de production vide :

Exemple

<i>exp</i>	$::=$	<i>exp</i> + <i>term</i>
		<i>term</i> * <i>factor</i>
		i
		i <i>plusplus</i>
		(<i>exp</i>)
<i>term</i>	$::=$	<i>term</i> * <i>factor</i>
		i
		i <i>plusplus</i>
		(<i>exp</i>)
<i>factor</i>	$::=$	i
		i <i>plusplus</i>
		(<i>exp</i>)
<i>plusplus</i>	$::=$! <i>plusplus</i>

Élimination de la récursion immédiate gauche

- Soit une grammaire G , sans règle unitaire, ni règle de production vide.
- Pour tout non-terminal (récursif immédiat), on peut partitionner ses règles ainsi :

$$\begin{aligned} A &\rightarrow A\alpha_1 \mid \dots \mid A\alpha_m \\ A &\rightarrow \beta_1 \mid \dots \mid \beta_n \end{aligned}$$

où $\alpha_i \in (T \cup N)^+$.

$\beta_i \in (T \cup N)^+$ et ne commencent pas par le non-terminal A .

- On peut remplacer cet ensemble de règles par les règles suivantes :

$$\begin{aligned} A &\rightarrow A_{\text{tête}}A_{\text{suites}} \mid A_{\text{tête}} \\ A_{\text{tête}} &\rightarrow \beta_1 \mid \dots \mid \beta_n \\ A_{\text{suite}} &\rightarrow \alpha_1 \mid \dots \mid \alpha_m \\ A_{\text{suites}} &\rightarrow A_{\text{suite}}A_{\text{suites}} \mid A_{\text{suite}} \end{aligned}$$

où $A_{\text{tête}}$, A_{suite} et A_{suites} sont des non-terminaux fraîchement introduits.

Élimination de la récursion immédiate : exemple

$$\begin{aligned} \text{exp} &::= \text{exp} + \text{term} \\ &| \text{term} * \text{factor} \\ &| \mathbf{i} \\ &| \mathbf{i} \text{ plusplus} \\ &| (\text{exp}) \end{aligned}$$
$$\begin{aligned} \text{exp}_{\text{tête}} &::= \text{term} * \text{factor} \\ &| \mathbf{i} \\ &| \mathbf{i} \text{ plusplus} \\ &| (\text{exp}) \\ \text{exp}_{\text{suite}} &::= + \text{term} \\ \text{exp}_{\text{suites}} &::= \text{exp}_{\text{suite}} \text{exp}_{\text{suites}} \\ \text{exp} &::= \text{exp}_{\text{tête}} \text{exp}_{\text{suites}} \mid \text{exp}_{\text{tête}} \end{aligned}$$

Élimination de la récursion gauche

- ▶ Dans notre cas, il suffit de supprimer la récursion gauche immédiate de *exp* et *term* pour supprimer toutes récursions gauches de notre grammaire.
- ▶ Pour supprimer toute récursion gauche indirecte, il faut itérer ce procédé en suivant l'ordre induit par la relation de dépendance entre non-terminaux définie de la façon suivante :

A dépend en tête de B si il existe une règle « $B \rightarrow A\alpha$ ».

Checkpoint

- ▶ À ce stade, nous savons donc construire un automate à pile **non déterministe** pour toute grammaire hors-contexte (éventuellement préalablement transformée en une grammaire équivalente adéquate).
 - ▶ On peut implanter cet automate sous la forme d'un programme composée de fonctions mutuellement récursives correspondant à l'analyse de chaque non-terminal.
 - ▶ La pile des appels récursifs modélise alors les points de choix de l'arbre de recherche.
- ⇒ La complexité en pire cas est exponentielle.
- ⇒ On aimerait transformer le processus de recherche en un algorithme déterministe, quitte à restreindre la classe des grammaires reconnues.

Analyse descendante prédictive dirigée par une table

Sources d'information pour diriger la recherche

- ▶ Pour transformer notre méthode de recherche en un algorithme déterministe, il faut trouver les sources d'information qui permettent de faire un choix (et le bon) sans avoir à revenir dessus.
- ▶ Il y a deux sources essentielles d'information pour décider qu'elle est la bonne règle de grammaire à utiliser :
 - ▶ La dérivation qui a été construite jusqu'à maintenant (la partie analysée) ;
 - ▶ Les premiers lexèmes de l'entrée restant à analyser.

Cas simpliste

- Considérons la grammaire :

$$\begin{aligned} S &::= aB \mid bA \\ A &::= a \mid aS \mid bAA \\ B &::= b \mid bS \mid aBB \end{aligned}$$

- Toutes les règles de cette grammaire débutent par un terminal.
- Si la prédiction commence par un non-terminal A et l'entrée par un terminal a , le couple (A, a) caractérise un ensemble de règles à essayer.

Cas simpliste

- ▶ On peut construire la table suivante pour diriger l'analyse syntaxique :

	a	b	$\#$
S	$S_1 \rightarrow aB$	$S_2 \rightarrow bA$	
A	$A_1 \rightarrow a$ $A_2 \rightarrow aS$	$A_3 \rightarrow bAA$	
B	$B_3 \rightarrow aBB$	$B_1 \rightarrow b$ $B_2 \rightarrow bS$	

- ▶ Une fois cette **table d'analyse syntaxique** construite, la grammaire n'est plus nécessaire : les étapes de prédiction utilisent désormais une table.
- ▶ Si l'analyse pointe sur une case sans règle alors l'entrée est rejetée.
- ▶ Avoir plusieurs règles dans une case rend l'analyse non déterministe.

La classe des grammaires LL(1)

- ▶ Une grammaire est LL(1) si sa table d'analyse syntaxique contient au plus une règle par case.
- ▶ L'appellation « LL(1) » s'explique ainsi :
 - ▶ *Left to right* : une analyse directionnelle qui lit l'entrée de gauche à droite.
 - ▶ *Leftmost* : on imite la dérivation gauche.
 - ▶ (1) : on lit un lexème en avant pour prendre ses décisions.

La classe des grammaires SLL(1)

- ▶ La restriction « toute règle commence par un terminal » est trop restrictive.
 - ▶ La classe des grammaires la respectant s'appelle *Simple LL(1)* (SLL(1)).
- ⇒ La fin de cette séance va viser à relâcher cette restriction.

Analyse descendante prédictive $LL(1)$ totale

L'idée à retenir

- ▶ Les grammaires SLL(1) permettent de calculer aisément la table d'analyse syntaxique : il suffit de lire chaque règle et de la placer dans la bonne case de la table en observant le premier lexème de sa partie droite.
- ▶ On peut généraliser cette idée : pour construire la table d'analyse LL(1), il suffit de savoir calculer :

Les lexèmes
que le membre droit d'une règle peut produire en première position.

Un cas particulier

- ▶ Commençons par nous intéresser aux grammaires dont aucun non-terminal ne produit le mot vide.
- ▶ Cela signifie que pour tout membre droit α d'une règle, nous avons deux cas à considérer :
 - ▶ Soit a et le premier lexème produit par la règle est "a".
 - ▶ Soit A et l'ensemble des lexème produit en première position par la règle est le même que celui de A .

Équations entre ensemble de terminaux

- La définition précédente peut s'exprimer à l'aide de ces équations définissant $FIRST(\alpha)$ où α est un mot de $(T \cup N)^+$:

$$\begin{aligned} FIRST(a\alpha) &= \{a\} \\ FIRST(A\alpha) &= FIRST(A) \\ FIRST(\alpha) &\subseteq FIRST(A) \quad \text{Si } A \rightarrow \alpha \text{ dans } G \end{aligned}$$

Exercice

Comment calculer cette fonction FIRST ?

Calcul de point fixe

- ▶ Une solution aux systèmes d'inéquations précédent est fournit par le plus petit point fixe de la fonction suivante :

$$\begin{aligned} FIRST^n(a\alpha) &= \{a\} \\ FIRST^n(A\alpha) &= FIRST^{n-1}(A) \\ FIRST^n(A) &= FIRST^{n-1}(A) \cup FIRST^{n-1}(\alpha) \quad \text{Si } A \rightarrow \alpha \text{ dans } G \end{aligned}$$

- ▶ Pour calculer ce plus petit point fixe, on se donne une fonction $FIRST^0$ qui associe la fonction constante « $w \mapsto \emptyset$ ».
 - ▶ On itère ensuite la définition de cette fonction pour calculer une nouvelle fonction $FIRST^n$. Cette fonction associe à tous les mots des ensembles plus grands que la fonction de l'itération précédente.
 - ▶ Or, la taille de ces ensembles est bornée par le cardinal de l'alphabet (fini) des non-terminaux.
- ⇒ Cette itération termine donc et est une solution du système d'inéquations.

Conflit FIRST/FIRST

- ▶ Si les ensembles FIRST de deux règles d'un non-terminal partagent un symbole, on dit que l'on a un **conflit FIRST/FIRST**.
- ▶ Une telle grammaire n'est donc pas LL(1).

Construction d'une table LL(1)

<i>Session</i>	$::=$	<i>Fact Session</i>
		<i>Question</i>
		<i>(Session)Session</i>
<i>Fact</i>	$::=$	<i>!STRING</i>
<i>Question</i>	$::=$	<i>?STRING</i>

Exercice

1. Est-ce que cette grammaire est adaptée à l'analyse LL ?
2. Calculer la fonction FIRST.
3. Calculer la table LL correspondante.
4. Analyser l'entrée « (!FIRE ?WATER) !WATER ?WOOD ».

Traitement des non-terminaux produisant le mot vide

- ▶ Face à une règle de la forme $A \rightarrow \epsilon$, on ne peut pas décider quels terminaux peuvent suivre A .
- ▶ Il faut aller voir les terminaux qui **peuvent suivre** A en observant les règles qui utilisent A .
- ▶ Pour effectuer cette inspection efficacement, on étend le co-domaine de la fonction *FIRST* : on étend l'ensemble des terminaux à l'aide d'un symbole ϵ qui signifie que le non terminal peut produire le mot vide.
- ▶ On modifie aussi la définition de *FIRST* :

$FIRST(\epsilon)$	$=$	$\{\epsilon\}$	
$FIRST(a\alpha)$	$=$	$\{a\}$	
$FIRST(A\alpha)$	$=$	$FIRST(A)$	Si $\epsilon \notin FIRST(A)$
$FIRST(A\alpha)$	$=$	$(FIRST(A) \setminus \{\epsilon\}) \cup FIRST(\alpha)$	Si $\epsilon \in FIRST(A)$
$FIRST(\alpha)$	\subseteq	$FIRST(A)$	Si $A \rightarrow \alpha$ dans G

Un premier algorithme

- ▶ À l'aide de cette nouvelle version de la fonction FIRST, on peut définir un algorithme d'analyse syntaxique qui utilise la grammaire.
- ▶ À chaque étape de prédiction, si α est la prédiction courante, on calcule $\text{FIRST}(\alpha)$ pour déterminer la règle de la grammaire à utiliser.
- ▶ Si il y a plusieurs règles, on échoue : la grammaire n'est pas LL(1).

Algorithme LL(1) en présence de règles ϵ

$Session$	$::=$	$Facts \ Question$ \mid $(Session)Session$
$Facts$	$::=$	$Fact \ Facts \mid \epsilon$
$Fact$	$::=$	$!STRING$
$Question$	$::=$	$?STRING$

Exercice

1. Est-ce que cette grammaire est adaptée à l'analyse LL ?
2. Analyser l'entrée « !WATER (!FIRE ?WATER) ?WOOD » en calculant *FIRST* pour chaque étape.

Défauts de l'algorithme

- ▶ Le recalcul de *FIRST* à chaque étape de l'analyse est une source d'inefficacité en comparaison à l'analyse dirigée par une table.
- ▶ On peut cependant calculer une fonction *FOLLOW* pour chaque non-terminal A produisant le mot vide qui représente l'ensemble des terminaux qui peuvent suivre A .
- ▶ Ainsi, pour construire la table, il suffit de rajouter chaque règle $A \rightarrow \alpha$ dans toutes les cases (A, a) telles que $a \in FIRST(\alpha FOLLOW(A))$.
- ▶ Si pour un non terminal A pouvant produire le mot vide par une règle $A \rightarrow \alpha$ et qu'il existe un terminal qui est à la fois dans $FIRST(A)$ et dans $FOLLOW(A)$ alors on a un conflit *FIRST/FOLLOW*.
- ▶ Si un non terminal possède deux règles distinctes de la forme $A \rightarrow \alpha$ et $A \rightarrow \alpha$ qui peuvent produire le mot vide, alors il s'agit conflit *FOLLOW/FOLLOW*.

Définition de la fonction *FOLLOW*

- ▶ On utilise une définition similaire à *FIRST* à l'aide d'inéquations :

$$\begin{array}{lll} FIRST(\beta) & \subseteq & FOLLOW(A) \quad \text{Si } B \rightarrow \alpha A \beta \text{ dans } G. \\ FOLLOW(B) & \subseteq & FOLLOW(A) \quad \text{Si } B \rightarrow \alpha A \beta \text{ dans } G. \\ & & \text{et } \epsilon \in FIRST(\beta) \end{array}$$

- ▶ On peut encore résoudre ce système par itération du système :

$$\begin{array}{ll} FOLLOW^n(A) & = FOLLOW^{n-1}(A) \cup FIRST(\beta) \\ & \text{Si } B \rightarrow \alpha A \beta \text{ dans } G. \\ \\ FOLLOW^n(B) & = FOLLOW^{n-1}(B) \cup FOLLOW^{n-1}(A) \\ & \text{Si } B \rightarrow \alpha A \beta \text{ dans } G. \\ & \text{et } \epsilon \in FIRST(\beta) \end{array}$$

Construction de la table LL(1) en présence de règles ϵ

<i>Session</i>	$::=$	<i>Facts Question</i>
	$ $	$(Session)Session$
<i>Facts</i>	$::=$	<i>Fact Facts</i> $ \epsilon$
<i>Fact</i>	$::=$	$!STRING$
<i>Question</i>	$::=$	$?STRING$

Exercice

1. Calculer les fonctions FIRST et FOLLOW.
2. Calculer la table LL(1).
3. Analyser l'entrée « $(!FIRE ?WATER) !WATER ?WOOD$ ».

Synthèse

Conclusion

- ▶ Nous avons présenté l'algorithme de Unger, très simple mais qui n'utilise pas l'entrée pour prédire les chemins de recherche les plus probables.
- ▶ Ceci nous a conduit à l'analyse prédictive, d'abord non déterministe, puis déterministe.
- ▶ La classe des grammaires LL(1) conduit à des analyseurs syntaxiques très efficaces mais certaines grammaires hors-contexte ne sont pas LL(1).
- ▶ Il y a des méthodes pour essayer de transformer certaines grammaires hors-contexte en grammaires LL(1).