Introduction à la compilation
Cours 3 : Analyse syntaxique descendante

Yann Régis-Gianas
yrg@pps.jussieu.fr

PPS - Université Denis Diderot – Paris 7

Rappels sur l'analyse descendante

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

$$
\begin{array}{r||rcl}
(1) & S & \rightarrow & aSQ \\
(2) & S & \rightarrow & abc \\
(3) & bQc & \rightarrow & bbcc \\
(4) & cQ & \rightarrow & Qc
\end{array}
$$

et l'entrée : « *aabbcc* »

S

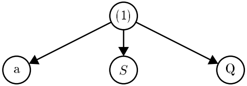
- L'analyse débute avec celle du symbole S_s.
- Quelle peut être la prochaine étape ?

Rappels sur l'analyse descendante

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

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

et l'entrée : « *aabbcc* »



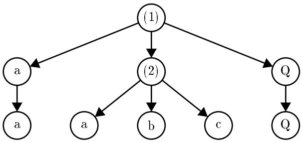
- Comme « abc » n'est pas un préfixe de l'entrée, la règle (2) ne s'applique pas.
- Seule la règle (1) peut s'appliquer.

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* »



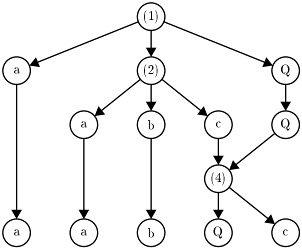
- Appliquer de nouveau la règle (1) imposerait le préfixe « aaa » dans l'entrée.
- On doit donc considérer la règle (2).

Rappels sur l'analyse descendante

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

$$\begin{array}{r||rcl} (1) & S & \rightarrow & aSQ \\ (2) & S & \rightarrow & abc \\ (3) & bQc & \rightarrow & bbcc \\ (4) & cQ & \rightarrow & Qc \end{array}$$

et l'entrée : « *aabbcc* »



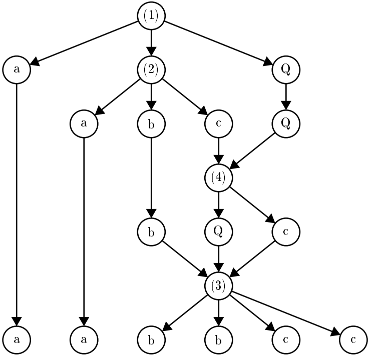
- Seule la règle (4) s'applique.
(C'était difficile à prévoir !)

Rappels sur l'analyse descendante

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

$$\begin{array}{c||rcl} (1) & S & \rightarrow & aSQ \\ (2) & S & \rightarrow & abc \\ (3) & bQc & \rightarrow & bbcc \\ (4) & cQ & \rightarrow & Qc \end{array}$$

et l'entrée : « *aabbcc* »



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

Algorithme 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 $c_1 \ldots c_n$ 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 « $S \rightarrow A_1 \ldots A_m$ », pour tous les découpages $\mathcal{P} \equiv P_1 ... P_m$ de l'entrée en **m** parties, on détermine (par récursion) si A_i peut produire P_i.

$\Rightarrow$ 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 :

$$\begin{array}{rcl}
exp & ::= & exp \ + \ term \\
 & | & term \\
term & ::= & term \ * \ factor \\
 & | & factor \\
factor & ::= & 0 \mid 1 \mid \ldots \\
 & | & (\ exp\ )
\end{array}$$

Calculer toutes les partitions de N objets en M parties ?

Exercice

Comment calculer toutes les partitions de N objets en M parties (non vides) ? (en supposant $M > 0$)

Calculer toutes les partitions de N objets en M parties ?

Exercice

Comment calculer toutes les partitions de N objets en M parties (non vides) ? (en supposant M > 0)

```
let rec splitting m n l : α splitting list =
  assert (m > 0);
  if (n < m) then []
  else match m, l with
    | 0, l →
      assert false
    | 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)
               (splitting (m - 1) (n - k) l)))
```

Implémentation (naïve) de Unger

```
let rec parse g s input : bool =
  let rules : ('t, 'n) rule list = rules_about g s in
  let match_rhs ((_, (rhs : ('t, 'n) rhs))) =
    let m = List.length rhs in
    let ps = splitting m input in
    let subinput_matches_symbol symbol subinput =
      match symbol, subinput with
        | Terminal t, [u] → t = u
        | NonTerminal s, _ → parse g s subinput
        | _ → false
    in
    let splitting_matches_rhs p =
      List.for_all2 subinput_matches_symbol rhs p
    in
    List.exists splitting_matches_rhs ps
  in
  List.exists match_rhs rules
```

Exercice

Quelle est la complexité de cette méthode de recherche ?

Règles de production vide

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

Exercice

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

```
let rec splitting m n l : α splitting list =
  match m, l with
    | 0, [] → [ [] ]
    | 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)
               (splitting (m - 1) (n - k) l)))
```

Règles de production vide

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

$$\boxed{\begin{array}{rcl} \mathit{factor} & \rightarrow & \mathit{INT}\ \mathit{plusplus} \\ \mathit{plusplus} & \rightarrow & \epsilon \\ \mathit{plusplus} & \rightarrow & \mathit{BANG}\ \mathit{plusplus} \end{array}}$$

- 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* ».

$\Rightarrow$ Ce dernier problème se réduit immédiatement sur lui-même, ce qui provoque la non terminaison de l'algorithme !

Forme des dérivations « qui bouclent »

- Les chemins de recherche de dérivation, à partir de S produisant w, qui sont exhibés par le phénomène précédent correspondent à des dérivations de la forme :
$$S \longrightarrow \ldots \longrightarrow \alpha S \beta$$
où α et β peuvent produire le mot vide.
- Si α et β ont effectivement produit le mot vide alors il existe une infinité de dérivations de cette forme : nous pouvons nous intéresser uniquement à celle sans boucle.
- Si α et β n'ont pas produit le mot vide alors il ne sert à rien de se poser la question « $S \overset{?}{\rightarrow} w$ » pour le S entre α et β.

$\Rightarrow$ On peut donc <u>sans danger</u> couper l'arbre de recherche en ce point.

$\Rightarrow$ Comment modifier l'algorithme pour prendre en compte cette remarque ?

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)) as r) =
    trace_rule input r;
    let m = List.length rhs in
    let ps = splitting 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)
        | _ → false
    in
    let splitting_matches_rhs p =
      trace_splitting p;
      List.for_all2 subinput_matches_symbol rhs p
    in
    List.exists splitting_matches_rhs ps
  in
  List.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 α answer = BeingAnswered | Answered of α

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 déteminer si « $exp \to^\star (42)$ » :

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.)

Inefficacité 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 la totalité de l'entrée.

- 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 :

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

Exercice

Quel est le langage reconnu par cette grammaire ?

Analyse prédictive : exemple

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

a a b b
S

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

Analyse prédictive : exemple

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

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

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

a	a b b
a B	

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

Analyse prédictive : exemple

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

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

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

a	a	b b
a	a	B B

- On accepte de nouveau de nouveau le lexème “a”.

Analyse prédictive : exemple

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

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

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

a a	b	b
a a	b	B

- On accepte le lexème "b".

Analyse prédictive : exemple

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

a a b	b
a a b	B

- On doit encore appliquer première règle de B.

Analyse prédictive : exemple

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

a a b b	
a a b b	

- On accepte finalement le dernier lexème “b”.

$\Rightarrow$ La prédiction est vide, tout comme l'entrée : l'analyse est un succès.

- La dérivation est :

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

Caractéristiques de l'analyse prédictive gauche-droite

- Nous avons produit la dérivation gauche.
- L'algorithme d'analyse est une succession de deux actions distinctes :
  - Soit la prédiction commence par un terminal "a" :
    - Si l'entrée commence aussi par "a" alors on continue.
    - Sinon on échoue.
  - Soit la prédiction commence par un non-terminal "A", alors on remplace ce non-terminal dans la prédiction par la partie droite de ses règles.
- Dans notre exemple, nous avons su choisir "magiquement" une seule partie droite à chaque étape. Cependant, en toute généralité, cet algorithme est encore un processus de recherche qui doit envisager toutes les possibilités.

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 non déterministe.

Définition formelle des automates à pile utilisés ici

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

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

  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 :

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

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

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

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 :

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

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

$$
\begin{array}{c|l}
 & ab\# \\
\hline\hline
 & S\#
\end{array}
\Rightarrow
\begin{array}{r|l}
 & ab\# \\
\hline\hline
S_1 & Sb\# \\
S_2 & ab\#
\end{array}
\Rightarrow
\begin{array}{r|l}
 & ab\# \\
\hline\hline
S_1 S_1 & Sbb\# \\
S_1 S_2 & ab\# \\
S_2 & ab\#
\end{array}
\Rightarrow
\begin{array}{r|l}
 & ab\# \\
\hline\hline
S_1 S_1 S_1 & Sbbb\# \\
S_1 S_1 S_2 & abb\# \\
S_1 S_2 & ab\# \\
S_2 & ab\#
\end{array}
\Rightarrow \dots
$$

Problème de la récursion à gauche

- Un non-terminal qui peut dériver une phrase commencant par lui-même est récursif 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 récursif gauche qui n'est pas récursif 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 récursifs à gauche en une grammaire équivalente non récursive 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 :
  - « $A \rightarrow \epsilon$ » : règle de production vide.
  - « $A \rightarrow B$ » : règle unitaire.

  où les symboles A et B sont des non-terminaux.

$\Rightarrow$ 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 :

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

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

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

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

$$
\begin{array}{rcl}
\mathit{exp} & ::= & \mathit{exp}\ +\ \mathit{term} \\
 & | & \mathit{term} \\
\mathit{term} & ::= & \mathit{term}\ *\ \mathit{factor} \\
 & | & \mathit{factor} \\
\mathit{factor} & ::= & \mathit{int}\ \mathit{plusplus} \\
 & | & (\ \mathit{exp}\ ) \\
\mathit{plusplus} & ::= & \epsilon \\
 & | & !\ \mathit{plusplus}
\end{array}
$$

Élimination des règles unitaires et de production vide : Exemple

$$
\begin{array}{rcl}
exp & ::= & exp \ + \ term \\
 & \mid & term \ * \ factor \\
 & \mid & \mathbf{i} \\
 & \mid & \mathbf{i} \ plusplus' \\
 & \mid & (\ exp \ ) \\
term & ::= & term \ * \ factor \\
 & \mid & \mathbf{i} \\
 & \mid & \mathbf{i} \ plusplus' \\
 & \mid & (\ exp \ ) \\
factor & ::= & \mathbf{i} \\
 & \mid & \mathbf{i} \ plusplus' \\
 & \mid & (\ exp \ ) \\
plusplus' & ::= & ! \ plusplus'
\end{array}
$$

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 :

$$\begin{array}{rcl} A & \rightarrow & A\alpha_1 \mid \ldots \mid A\alpha_m \\ A & \rightarrow & \beta_1 \mid \ldots \mid \beta_n \end{array}$$

  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 :

$$\begin{array}{rcl} A & \rightarrow & A_{\text{tête}} A_{\text{suites}} \mid A_{\text{tête}} \\ A_{\text{tête}} & \rightarrow & \beta_1 \mid \ldots \mid \beta_n \\ A_{\text{suite}} & \rightarrow & \alpha_1 \mid \ldots \mid \alpha_m \\ A_{\text{suites}} & \rightarrow & A_{\text{suite}} A_{\text{suites}} \mid A_{\text{suite}} \end{array}$$

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

Élimination de la récursion immédiate : exemple

$$
\begin{array}{lcl}
\mathit{exp} & ::= & \mathit{exp} \;+\; \mathit{term} \\
 & | & \mathit{term} \;*\; \mathit{factor} \\
 & | & \mathbf{i} \\
 & | & \mathbf{i}\ \mathit{plusplus}' \\
 & | & (\ \mathit{exp}\ )
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{exp}_{\text{tête}} & ::= & \mathit{term} \;*\; \mathit{factor} \\
 & | & \mathbf{i} \\
 & | & \mathbf{i}\ \mathit{plusplus}' \\
 & | & (\ \mathit{exp}\ ) \\
\mathit{exp}_{\text{suite}} & ::= & +\ \mathit{term} \\
\mathit{exp}_{\text{suites}} & ::= & \mathit{exp}_{\text{suite}}\,\mathit{exp}_{\text{suites}} \\
\mathit{exp} & ::= & \mathit{exp}_{\text{tête}}\,\mathit{exp}_{\text{suites}} \mid \mathit{exp}_{\text{tête}}
\end{array}
$$

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 « $B \to A\alpha$ ».

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.

$\Rightarrow$ La complexité en pire cas est exponentielle.

$\Rightarrow$ 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 :

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

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

- 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 :
  - ***L**eft to right* : une analyse directionnelle qui lit l'entrée de gauche à droite.
  - ***L**eftmost* : 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)).

$\Rightarrow$ 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

- Commencons 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 $\alpha \equiv a\beta$ et le premier lexème produit par la règle est "a".
  - Soit $\alpha \equiv A\beta$ 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 $(T \cup N)^+$:

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

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 :

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

- 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.

$\Rightarrow$ 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 partage 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)

$$
\begin{array}{rcl}
\mathit{Session} & ::= & \mathit{Fact}\ \mathit{Session} \\
& | & \mathit{Question} \\
& | & (\mathit{Session})\mathit{Session} \\
\mathit{Fact} & ::= & !\mathit{STRING} \\
\mathit{Question} & ::= & ?\mathit{STRING}
\end{array}
$$

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.

Non-terminaux annulables

- Si un non terminal A peut produire le mot vide, on dit qu'il est annulable.
- On définit le prédicat $NULLABLE(w)$ ainsi :

$$\boxed{\begin{array}{rcl} NULLABLE(\epsilon) & = & \top \\ NULLABLE(a) & = & \bot \\ NULLABLE(A) & = & \exists w, NULLABLE(w) \wedge A \rightarrow w \in G \\ NULLABLE(s\alpha) & = & NULLABLE(s) \wedge NULLABLE(\alpha) \end{array}}$$

Nouvelle fonction *FIRST*

$$\boxed{\begin{array}{rcll} FIRST(a\alpha) & = & \{a\} & \\ FIRST(A\alpha) & = & FIRST(A) & \text{Si } \neg NULLABLE(A) \\ FIRST(A\alpha) & = & FIRST(A) \cup FIRST(\alpha) & \text{Si } NULLABLE(A) \\ FIRST(\alpha) & \subseteq & FIRST(A) & \text{Si } A \to \alpha \text{ dans } G \end{array}}$$

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 FIRST(α) 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

$$
\begin{array}{rcl}
\mathit{Session} & ::= & \mathit{Facts}\ \mathit{Question} \\
 & | & (\mathit{Session})\mathit{Session} \\
\mathit{Facts} & ::= & \mathit{Fact}\ \mathit{Facts} \mid \epsilon \\
\mathit{Fact} & ::= & !\mathit{STRING} \\
\mathit{Question} & ::= & ?\mathit{STRING}
\end{array}
$$

Exercice

1. Analysez 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 $A \to \alpha$ dans toutes les cases (A, a) telles que $a \in FIRST(\alpha\, FOLLOW(A))$.
- Si il existe un non terminal A pouvant produire le mot vide par une règle $A \to \alpha$ 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 $A \to \alpha$ et $A \to \alpha'$ qui peuvent produire le mot vide, alors il s'agit d'un conflit $FOLLOW/FOLLOW$.

Définition de la fonction *FOLLOW*

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

$$
\begin{array}{rcll}
\mathit{FIRST}(\beta) & \subseteq & \mathit{FOLLOW}(A) & \text{Si } B \rightarrow \alpha A \beta \text{ dans } G. \\
\mathit{FOLLOW}(B) & \subseteq & \mathit{FOLLOW}(A) & \text{Si } B \rightarrow \alpha A \beta \text{ dans } G \\
 & & & \text{et } \mathit{NULLABLE}(\beta).
\end{array}
$$

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

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

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

$$
\begin{array}{rcl}
\mathit{Session} & ::= & \mathit{Facts}\ \mathit{Question} \\
 & | & (\mathit{Session})\mathit{Session} \\
\mathit{Facts} & ::= & \mathit{Fact}\ \mathit{Facts} \mid \epsilon \\
\mathit{Fact} & ::= & !\mathit{STRING} \\
\mathit{Question} & ::= & ?\mathit{STRING}
\end{array}
$$

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).