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) :

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

et l'entrée : « *aabbcc* »

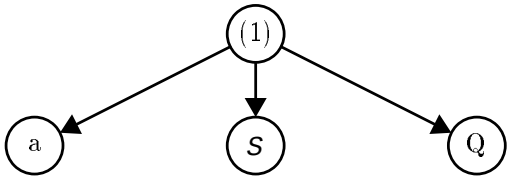
s

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* »



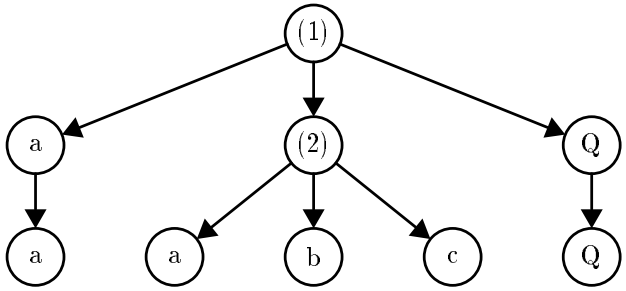
- Comme « abc » n'est pas un préfixe de l'entrée, la règle (2) ne s'applique pas.
- Seule la règle (1) peut s'appliquer.

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* »



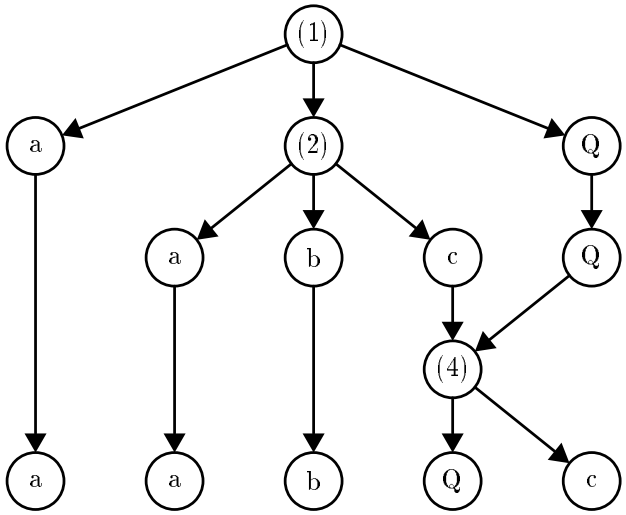
- Appliquer de nouveau la règle (1) imposerait le préfixe « aaa » dans l'entrée.
- On doit donc considérer la règle (2).

Rappels sur l'analyse descendante

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

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

et l'entrée : « *aabbcc* »



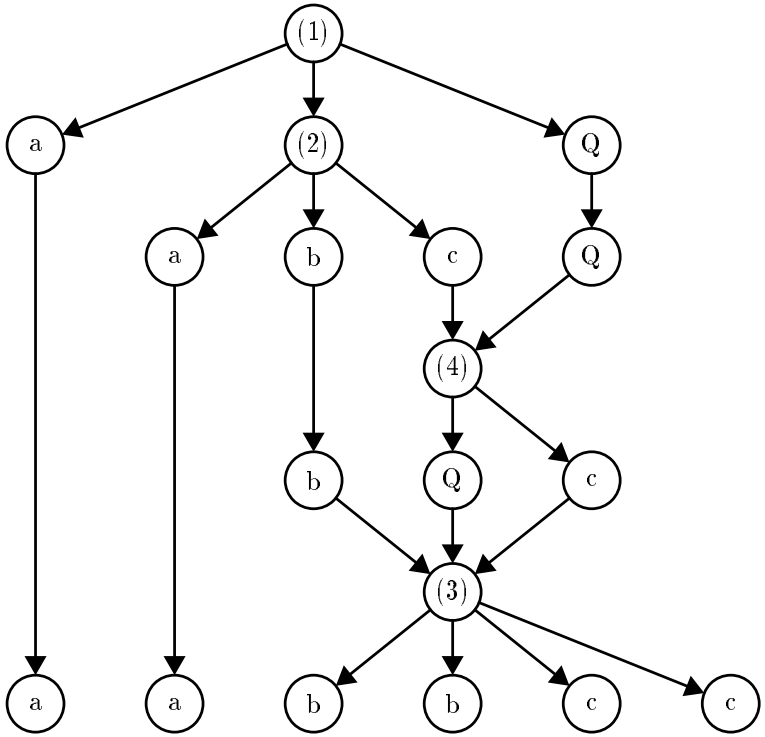
- Seule la règle (4) s'applique.
  (C'était difficile à prévoir !)

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.

Algorithme de Unger

Principe

- Soit G, une grammaire hors-contexte, sans règle 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 « $S \to A_1 \ldots A_m$ », pour toutes les découpages $\mathcal{P} \equiv P_1 \ldots P_m$ 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 :

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

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 :

$$\begin{array}{rcl} \textit{factor} & \rightarrow & \textit{INT plusplus} \\ \textit{plusplus} & \rightarrow & \\ \textit{plusplus} & \rightarrow & \textit{BANG plusplus} \end{array}$$

- 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* → *exp* + *term* ». En étudiant la partition « |1|! », on doit résoudre le problème de la génération de « » par « *exp* ».

  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.

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

  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

- 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 :

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

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”.

  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.

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 :

	$ab\#$
	$S\#$

	$ab\#$
S_1	$Sb\#$
S_2	$ab\#$

	$ab\#$
S_1S_1	$Sbb\#$
S_1S_2	$ab\#$
S_2	$ab\#$

	$ab\#$
$S_1S_1S_1$	$Sbbb\#$
$S_1S_1S_2$	$abb\#$
S_1S_2	$ab\#$
S_2	$ab\#$

$\ldots$

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.

  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 :

$$A \rightarrow$$

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

Élimination des règles unitaires et de production vide : Exemple

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

Élimination des règles unitaires et de production vide : Exemple

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

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 :

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

  où $\alpha_i \quad (T \quad N)^+$.

  $\beta_i \quad (T \quad N)^+$ 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

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

$$
\begin{array}{rcl}
exp_{\text{tête}} & ::= & term \quad factor \\
 & | & \mathbf{i} \\
 & | & \mathbf{i} \ plusplus' \\
 & | & (\ exp\ ) \\
exp_{\text{suite}} & ::= & + \ term \\
exp_{\text{suites}} & ::= & exp_{\text{suite}} exp_{\text{suites}} \\
exp & ::= & exp_{\text{tête}} exp_{\text{suites}} \ | \ exp_{\text{tête}}
\end{array}
$$

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.

  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

- 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 :
  - ***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)).

  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 $(T \quad N)^+$:

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

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 :

$$\begin{array}{rcl} FIRST^n(a\alpha) & = & \{a\} \\ FIRST^n(A\alpha) & = & FIRST^{n-1}(A) \\ FIRST^n(A) & = & FIRST^{n-1}(A) \quad FIRST^{n-1}(\alpha) \quad \text{Si } A \to \alpha \text{ dans } G \end{array}$$

- Pour calculer ce plus petit point fixe, on se donne une fonction $FIRST^0$ qui associe la fonction constante « $w \to \quad$ ».
- 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 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 $A \rightarrow \quad$, 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 rajoute un nouveau symbole $\dot{}$ à l'ensemble des terminaux.
- Si $\dot{} \quad FIRST(A)$, cela signifie que le non terminal A peut produire le mot vide. On dit qu'il est annulable.
- On modifie aussi la définition de $FIRST$:

$$
\begin{array}{rcll}
FIRST(\ ) & = & \{\dot{}\} & \\
FIRST(a\alpha) & = & \{a\} & \\
FIRST(A\alpha) & = & FIRST(A) & \text{Si } \dot{} \ / \ FIRST(A) \\
FIRST(A\alpha) & = & (FIRST(A) \setminus \{\dot{}\}) \quad FIRST(\alpha) & \text{Si } \dot{} \quad FIRST(A) \\
FIRST(\alpha) & & FIRST(A) & \text{Si } A \rightarrow \alpha \text{ dans } G
\end{array}
$$

Un premier algorithme

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

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).