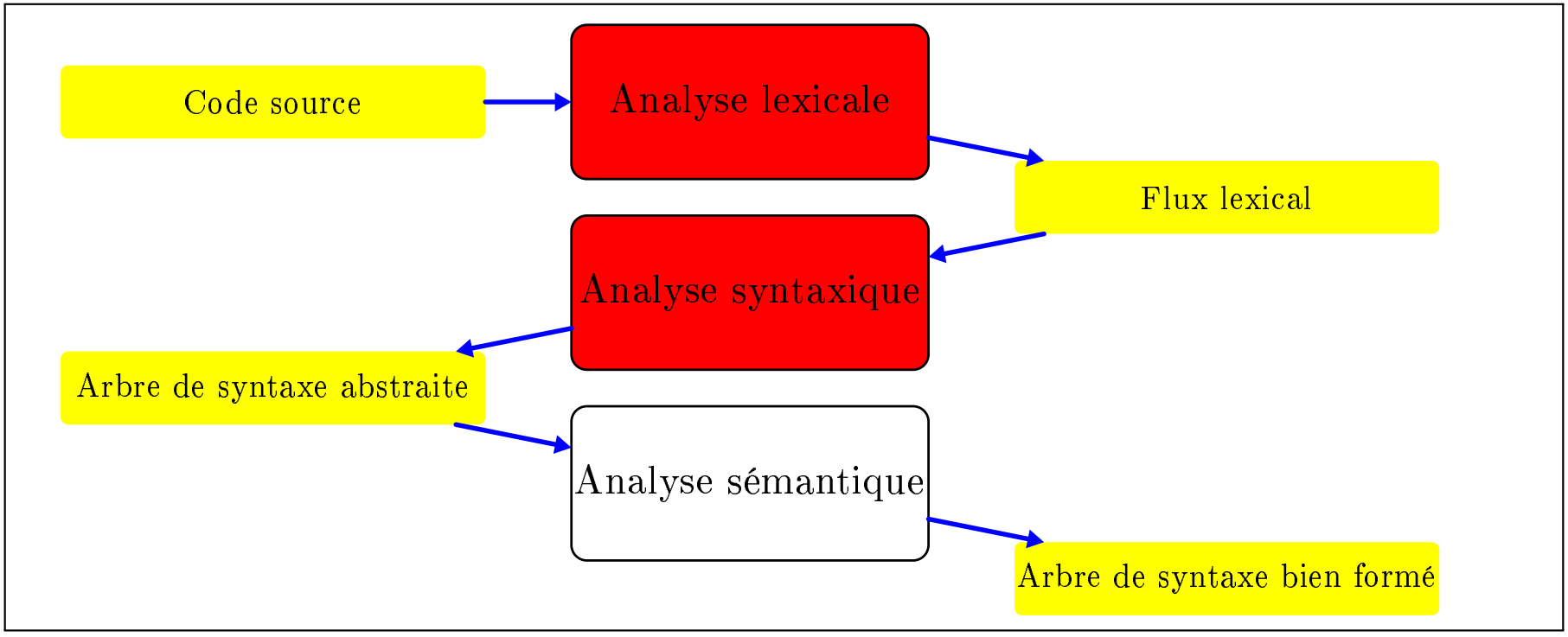
Introduction à la compilation
Cours 2 : Analyse lexicale et syntaxique

Yann Régis-Gianas
yrg@pps.univ-paris-diderot.fr

PPS - Université Denis Diderot – Paris 7

Rappel du dernier cours



Première partie du cours :

L'analyse lexicale et l'analyse syntaxique dans la partie avant d'un compilateur.

Vue d'ensemble

Grandes lignes

- L'analyse lexicale traduit un flux de caractères en un flux de lexèmes.
- L'analyse syntaxique traduit un flux de lexèmes en un arbre de syntaxe abstraite.

$\Rightarrow$ Dans les deux cas, il s'agit d'exhiber la structure implicite d'un flux.

- Cette structure nous servira à donner un sens aux programmes.

$\Rightarrow$ Pour le moment, nous nous focalisons sur les problèmes suivants :

1. Comment spécifier la structure de tous les programmes d'un langage $\mathcal{L}$?
2. Comment reconnaître la structure d'un programme et la reconstruire ?

Décrire une syntaxe

- Pour voir un programme comme un arbre, on doit répondre à ces questions :

  - Quelles sont ses feuilles ? – Les symboles terminaux.

  $\Rightarrow$ les mots-clés du langage, sa « ponctuation », ses atomes de sens.

  - Quels sont ses nœuds ? – Les symboles non terminaux.

  $\Rightarrow$ ses catégories syntaxiques (expressions, déclarations, types, . . .).

Décrire une syntaxe : exemple du langage MARTHA

- Terminaux :
  - Mots-clés : **sum**, **from**, **to**, **do**, **def**.
  - Identifiants : x, y, z, foo, bar, ...
  - Constantes entières : 0, 1, 2, 3, ...
  - Symboles : ‘+‘, ‘*‘, ‘/‘, ‘−‘, ‘(‘, ‘)‘, ‘,‘, ‘=‘.
- Non terminaux :
  - Expression : les expressions arithmétiques formées :
    - par application des symboles d'opérateurs appliqués à des sous-expressions, en respectant les priorités entre ces symboles ;
    - par application de l'opérateur **sum** suivi d'un identifiant, du mot-clé **from**, d'une expression, du mot-clé **to**, d'une expression, du mot-clé **do** et d'une expression ;
    - par application d'une fonction à une liste d'expressions, séparées par des virgules et entourée de parenthèses.
  - Définition de fonctions : le mot-clé **def** suivi d'un identifiant, d'une liste d'identifiants entre parenthèses et séparés par des virgules, du symbole ‘=‘ et d'une expression.
  - Programme : une liste de définitions de fonction suivie d'une expression.

⇒ Une description peu concise ... et incomplète !

(Quelles sont les priorités respectives des symboles ? Comment les décrire ?)

Grammaires

Une grammaire (générative, syntagmatique, formelle, ...) est un quadruplet (N, T, R, S) où :

- N et T sont des ensembles de symboles disjoints.
- R est un ensemble de paires (P, Q) avec $P \in (N \cup T)^+$ et $Q \in (N \cup T)^*$ [1]
- Il existe un symbole $S \in N$.

Les règles de la grammaire (P, Q) de R, sont en général notés « $P \to Q$ ».

- N est l'ensemble des symboles non-terminaux.
- T est l'ensemble des symboles terminaux.
- R est l'ensemble des règles.
- S est le symbole d'entrée.

1. A^+ représente l'ensemble des séquences non vides d'éléments de A tandis que A^* représente l'ensemble des séquences vides d'éléments de A.

La grammaire du langage MARTHA

- $Keywords \equiv \{\textbf{sum}, \textbf{from}, \textbf{to}, \textbf{do}, \textbf{def}\}$
- $f, x \in Identifiers \equiv \{a, \ldots, z\}^+ \setminus Keywords$ [2]
- $i \in Integers \equiv \{0, \ldots, 9\}^+$
- $T \equiv \{\text{`+'}, \text{`*'}, \text{`/'}, \text{`-'}, \text{`('}, \text{`)'}, \text{`,'}, \text{`='}\} \cup Keywords \cup Identifiers$
- $N \equiv$
  $\{Expression, Factor, Term, SExpressions\} \cup$
  $\{SIdentifiers, Definition, Definitions, Program\}$
- R est :

$$
\begin{array}{rcl}
Expression & \rightarrow & Expression \text{ `+' } Term \\
Expression & \rightarrow & Expression \text{ `-' } Term \\
Expression & \rightarrow & Term \\
Term & \rightarrow & Term \text{ `*' } Factor \\
Term & \rightarrow & Term \text{ `/' } Factor \\
Term & \rightarrow & Factor \\
Factor & \rightarrow & i \\
Factor & \rightarrow & x \\
Factor & \rightarrow & f \text{ `(' } SExpressions \text{ `)'} \\
Factor & \rightarrow & \text{`(' } Expression \text{ `)'}
\end{array}
$$

2. f, x sont des "méta-variables" qui dénotent des éléments pris dans l'ensemble *Identifiers*.

La grammaire du langage MARTHA (suite)

SExpressions	$\rightarrow$	*Expression*
SExpressions	$\rightarrow$	*Expression* ‘,‘ *SExpressions*
SIdentifiers	$\rightarrow$	*x*
SIdentifiers	$\rightarrow$	*x* ‘,‘ *SIdentifiers*
Definition	$\rightarrow$	**def** *f* ‘(‘ *SIdentifiers* ‘)‘ ‘=‘ *Expression*
Definitions	$\rightarrow$	ϵ
Definitions	$\rightarrow$	*Definition Definitions*
Program	$\rightarrow$	*Definitions Expression*

La grammaire du langage MARTHA, écriture condensée

Expression	$\rightarrow$	*Expression* ‘+‘ *Term* | *Expression* ‘−‘ *Term* | *Term*
Term	$\rightarrow$	*Term* ‘∗‘ *Factor* | *Term* ‘/‘ *Factor* | *Factor*
Factor	$\rightarrow$	*i* | *x* | *f* ‘(‘ *SExpressions* ‘)‘ | ‘(‘ *Expression* ‘)‘
SExpressions	$\rightarrow$	*Expression* | *Expression* ‘,‘ *SExpressions*
SIdentifiers	$\rightarrow$	*x* | *x* ‘,‘ *SIdentifiers*
Definition	$\rightarrow$	**def** *f* ‘(‘ *SIdentifiers* ‘)‘ ‘=‘ *Expression*
Definitions	$\rightarrow$	ϵ | *Definition Definitions*
Program	$\rightarrow$	*Definitions Expression*

- On peut déduire T et N par simple observation des règles.
- On utilise le symbole '|' pour dénoter une alternative dans la grammaire.

La grammaire du langage MARTHA

Exercice

⇒ Modifiez la grammaire pour accepter la définition et l'utilisation de fonctions d'arité nulle.[3]

3. C'est-à-dire dont le nombre d'arguments formels vaut zéro.

Grammaires génératives en OCaml

- On peut donner un type OCaml à ces objets :

```
type ('t, 'n) symbol =
  | Terminal of 't
  | NonTerminal of 'n

type ('t, 'n) lhs =
    ('t, 'n) symbol × ('t, 'n) symbol list

type ('t, 'n) rhs =
    ('t, 'n) symbol list

type ('t, 'n) rule =
    ('t, 'n) lhs × ('t, 'n) rhs

type ('t, 'n) grammar =
    ('t, 'n) rule list
```

Grammaire de MARTHA en OCAML

```
type ('t, 'n) rule_alternative = 'n → ('t, 'n) rule list

(* ('t, 'n) rhs → ('t, 'n) rule_alternative *)
let ( ! ) rhs =
  fun lhs → [((NonTerminal lhs, []), rhs)]

(* ('t, 'n) rule_alternative → ('t, 'n) rhs *)
(* → ('t, 'n) rule_alternative *)
let ( *| ) rhs1 rhs2 =
  fun lhs → (rhs1 lhs @ ( ! rhs2) lhs)

(* ('t, 'n) rule_alternative → 'n → ('t, 'n) rule list *)
let ( ⟶ ) lhs rhs = rhs lhs
```

Quelques notations.

Terminaux et non-terminaux de MARTHA en OCAML

```
type terminal =
    | Int
    | Identifier
    | Lparen | Rparen | Comma
    | Plus | Minus | Star | Slash
    | Sum | From | To | Do | Def

type nonterminal =
    | Expression
    | Term
    | Factor
    | SExpressions
    | SIdentifiers
    | Definition
    | Definitions
    | Program
```

Grammaire de MARTHA en OCAML

```
let martha = [
  Expression −→
   ! [ NonTerminal Expression ; Terminal Plus ; NonTerminal Term ]
  *| [ NonTerminal Expression ; Terminal Minus ; NonTerminal Term ]
  *| [ NonTerminal Term ] ;

  Term −→
   ! [ NonTerminal Term ; Terminal Star ; NonTerminal Factor ]
  *| [ NonTerminal Term ; Terminal Slash ; NonTerminal Factor ]
  *| [ NonTerminal Factor ] ;

  Factor −→
   ! [ Terminal Int ]
  *| [ Terminal Identifier ]
  *| [ Terminal Identifier ; Terminal Lparen ; NonTerminal SExpressions ; Terminal Rparen ]
  *| [ Terminal Lparen ; NonTerminal Expression ; Terminal Rparen ] ;

  SExpressions −→
   ! [ NonTerminal Expression ]
  *| [ NonTerminal Expression ; Terminal Comma ; NonTerminal SExpressions ] ;

  SIdentifiers −→
   ! [ Terminal Identifier ]
  *| [ Terminal Identifier ; Terminal Comma ; NonTerminal SIdentifiers ] ;

  Definition −→
   ! [ Terminal Def ; Terminal Identifier ; Terminal Lparen ; NonTerminal SIdentifiers ; Terminal Rparen ] ;

  Definitions −→
   ! [ ]
  *| [ NonTerminal Definition ; NonTerminal Definitions ] ;

  Program −→
   ! [ NonTerminal Definitions ; NonTerminal Expression ]
]
```

Produire un mot à partir d'une grammaire

- En suivant les règles de la grammaire de MARTHA, on peut engendrer un programme syntaxiquement correct de ce langage :

Forme intermédiaire	Règle utilisée
Program	
Definitions Expression	*Program* $\rightarrow$ *Definitions Expression*
Expression	*Definitions* $\rightarrow \epsilon$
Expression ‘+‘ *Term*	*Expression* $\rightarrow$ *Expression* ‘+‘ *Term*
Expression ‘+‘ *Factor*	*Term* $\rightarrow$ *Factor*
Expression ‘+‘ 21	*Factor* $\rightarrow$ *i*
Term ‘+‘ 21	*Expression* $\rightarrow$ *Term*
Factor ‘+‘ 21	*Term* $\rightarrow$ *Factor*
21 ‘+‘ 21	*Factor* $\rightarrow$ *i*

Produire un mot à partir d'une grammaire

- On aurait pu obtenir le même programme par un chemin différent :

Forme intermédiaire	Règle utilisée
Program	
Definitions Expression	*Program* $\rightarrow$ *Definitions Expression*
Expression	*Definitions* $\rightarrow \epsilon$
Expression ‘+‘ *Term*	*Expression* $\rightarrow$ *Expression* ‘+‘ *Term*
Term ‘+‘ *Term*	*Expression* $\rightarrow$ *Term*
Term ‘+‘ *Factor*	*Term* $\rightarrow$ *Factor* (à l'occurrence 2)
Term ‘+‘ 21	*Factor* $\rightarrow$ *i*
Factor ‘+‘ 21	*Term* $\rightarrow$ *Factor*
21 ‘+‘ 21	*Factor* $\rightarrow$ *i*

(Notons que l'occurrence précise d'application d'une règle peut être nécessaire.)

Les règles en toute généralité

- Rien ne nous empêche d'écrire une règle « AddAnd » de la forme :

  Identifier ‘,‘ *Identifier* ‘)‘$\rightarrow$ *Identifier* **and** *Identifier* ‘)‘

$\Rightarrow$ En utilisant plusieurs symboles à la gauche d'une règle, on peut définir un contexte d'application de cette règle.

Dérivation

- Soit une grammaire $G \equiv (T, N, R, S)$
- Soient u et v, deux séquences de $(T \cup N)^*$.
- On écrit « $u \rightarrow v$ », qui se lit « u dérive v », si :
  - la séquence u s'écrit « $w_1\, i_1 \ldots i_m\, w_2$ » ;
  - la séquence v s'écrit « $w_1\, p_1 \ldots p_n\, w_2$ » ;
  - la règle « $i_1 \ldots i_m \rightarrow p_1 \ldots p_n$ » est dans R.
- En d'autres termes, v est u dont une sous-séquence a été réécrite par une règle de R.
- La relation « $\rightarrow$ » est appelée relation de dérivation immédiate de G.
- La fermeture transitive et réflexive de cette relation est appelée relation de dérivation de G. Elle est notée « $\rightarrow^\star$ ».
- Le langage engendré par G, que nous noterons généralement $\mathcal{L}_G$, est :

$$\{v \in T^* \mid S \rightarrow^\star v\}$$

Dérivation en OCaml

```
(* Une application de règle est déterminée par *)
(* son lieu d'application et la règle appliquée. *)
type ('t, 'n) rule_application =
    int × ('t, 'n) rule

(* Une dérivation peut être représentée par *)
(* une liste d'applications de règle. *)
type ('t, 'n) derivation =
    ('t, 'n) rule_application list
```

Dérivation en OCaml

```
let rule_number g index =
  List.nth g index

let rules_about (g : ('t, 'n) grammar) t =
  List.filter (function (
  | (NonTerminal rt, []), _) → rt = t
  | _ → false)
  g

let rule_about g ?(index=0) t =
  List.nth (rules_about g t) index
```

Quelques accesseurs de règle dans une grammaire.

Dérivation en OCaml

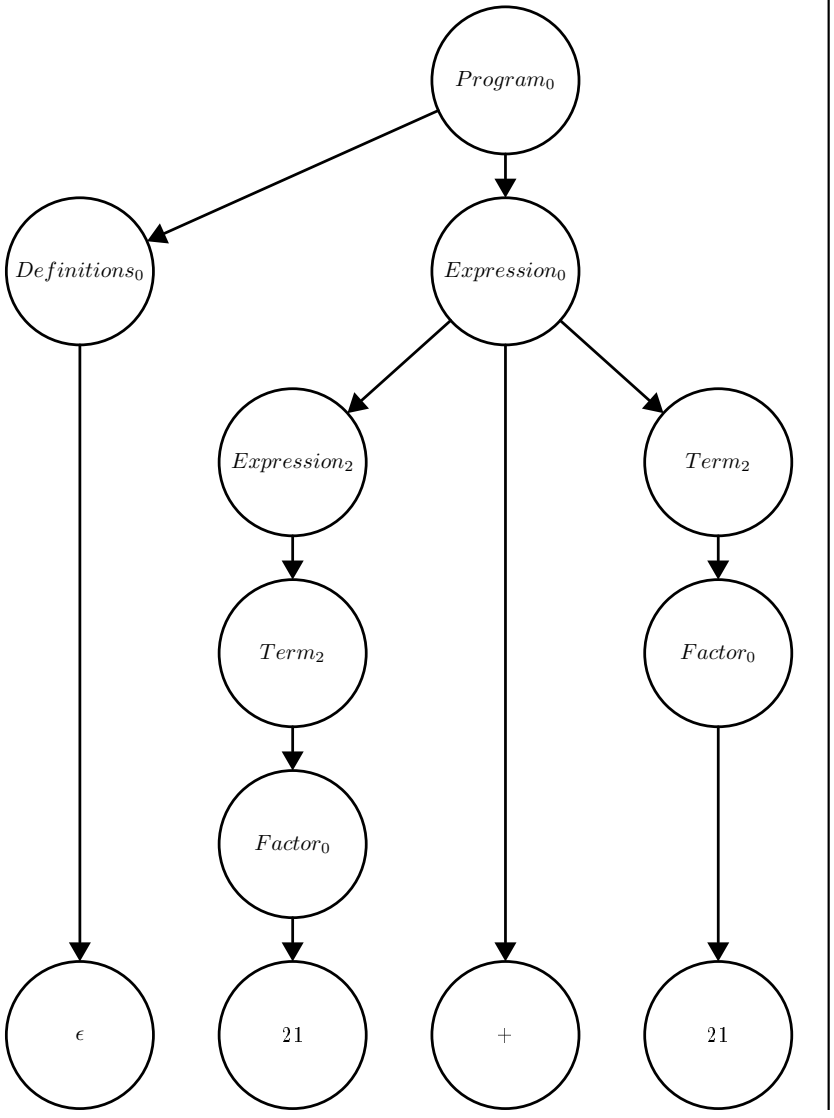
```
let derivation_example = [
  0, rule_about martha Program ;
  0, rule_about martha ~index: 0 Definitions ;
  0, rule_about martha ~index: 0 Expression ;
  2, rule_about martha ~index: 2 Term ;
  2, rule_about martha ~index: 0 Factor ;
  0, rule_about martha ~index: 2 Expression ;
  0, rule_about martha ~index: 2 Term ;
  0, rule_about martha ~index: 0 Factor ;
]
```

La dérivation, représentée à l'aide d'une valeur OCaml.

Dérivation en OCaml

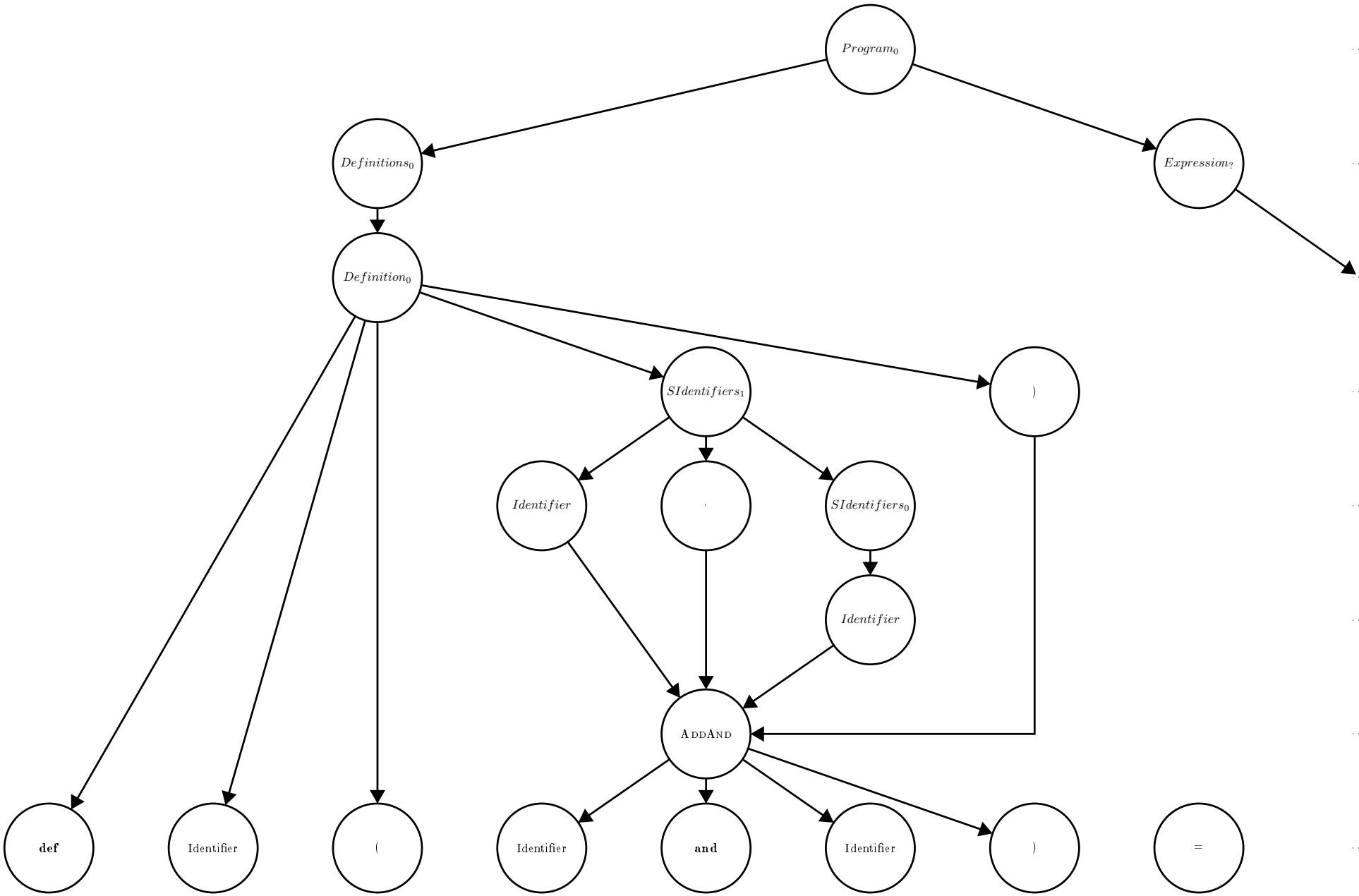
```
let apply_rule occ ((start, seq), result) s =
  let prefix, focus = split occ s in
  let suffix =
    let rec aux (pattern, s) =
      match (pattern, s) with
      | [], suffix → suffix
      | p :: ps, sym :: syms when p = sym → aux (ps, syms)
      | _ → failwith "Invalid rule application."
    in
    aux (start :: seq, focus)
  in
  prefix @ result @ suffix

let rec interpret g s = function
  | [] → s
  | (occ, r) :: d → interpret g (apply_rule occ r s) d
```

Représentation graphique



- Un graphe (acyclique) de production associé à une dérivation est formé :
  - de nœuds correspondants aux règles appliquées ;
  - d'arêtes symbolisant les entrées et sorties des règles.
- Par abus de notation, un nœud contenant un symbole correspond à la règle de reconnaissance de ce symbole.
- Dans cette représentation, les deux dérivations précédentes sont identiques.

Exemple : une règle à plusieurs entrées



Construire la représentation graphique d'une dérivation

Exercice

Instrumentez l'interprétation des dérivations que nous avons écrit en OCaml de façon à construire un graphe de production.
(Vous pouvez produire un fichier dans le format d'entrée de GraphViz[4].)

4. http://www.graphviz.org/

Représentations canoniques des graphes de production

- En effectuant un parcours *préfixe* du graphe de production, on obtient la dérivation gauche, c'est-à-dire la dérivation qui réécrit en priorité les symboles les plus à gauche.
- En effectuant un parcours *postfixe*, on obtient l'image miroir de la dérivation droite, c'est-à-dire la dérivation qui réécrit en priorité les symboles les plus à droite.

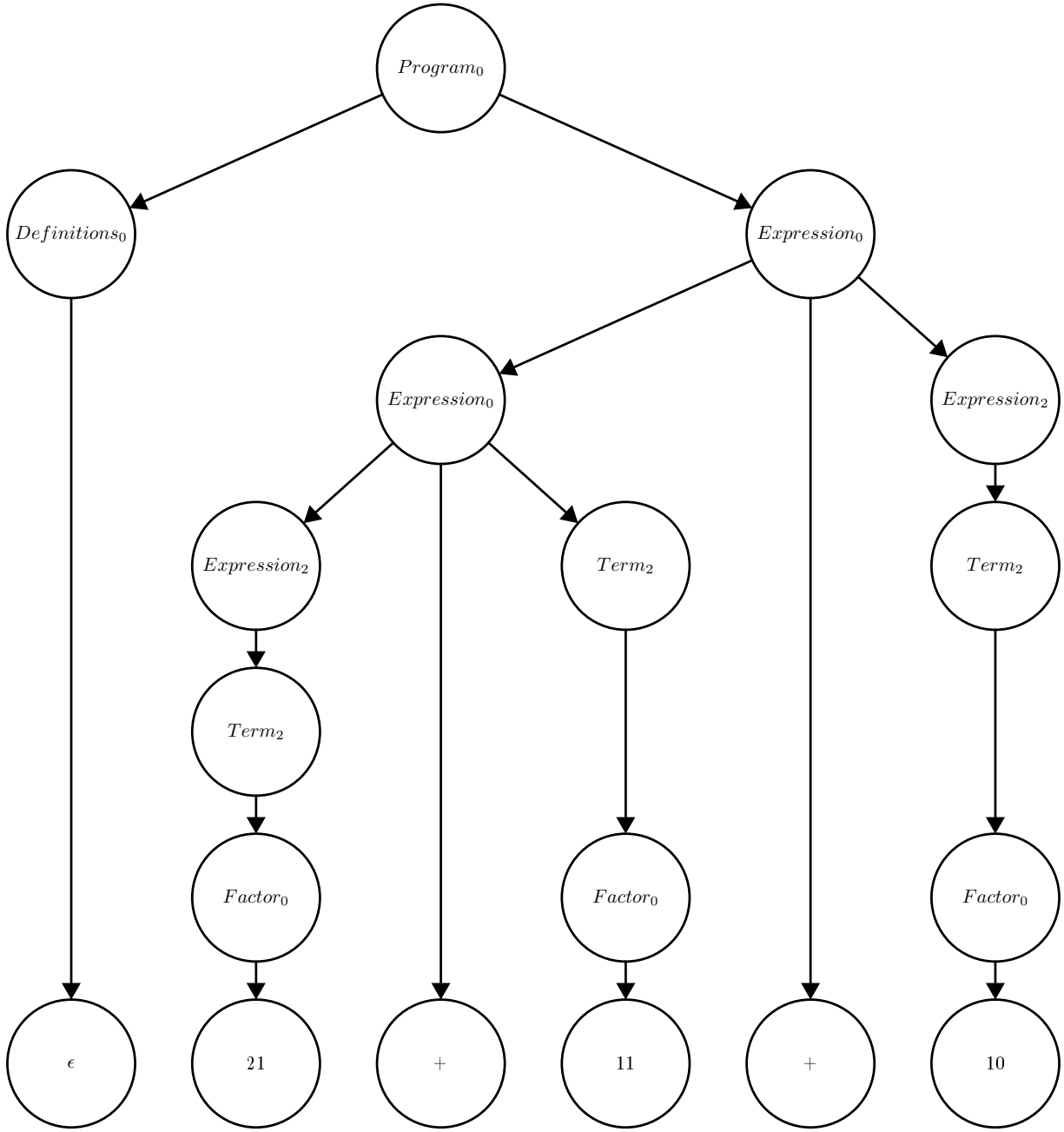
$\Rightarrow$ Ces dérivations sont des représentants canoniques pour les dérivations équivalentes (*i.e.* qui possèdent le même arbre de production).

La grammaire du langage MARTHA, modifiée

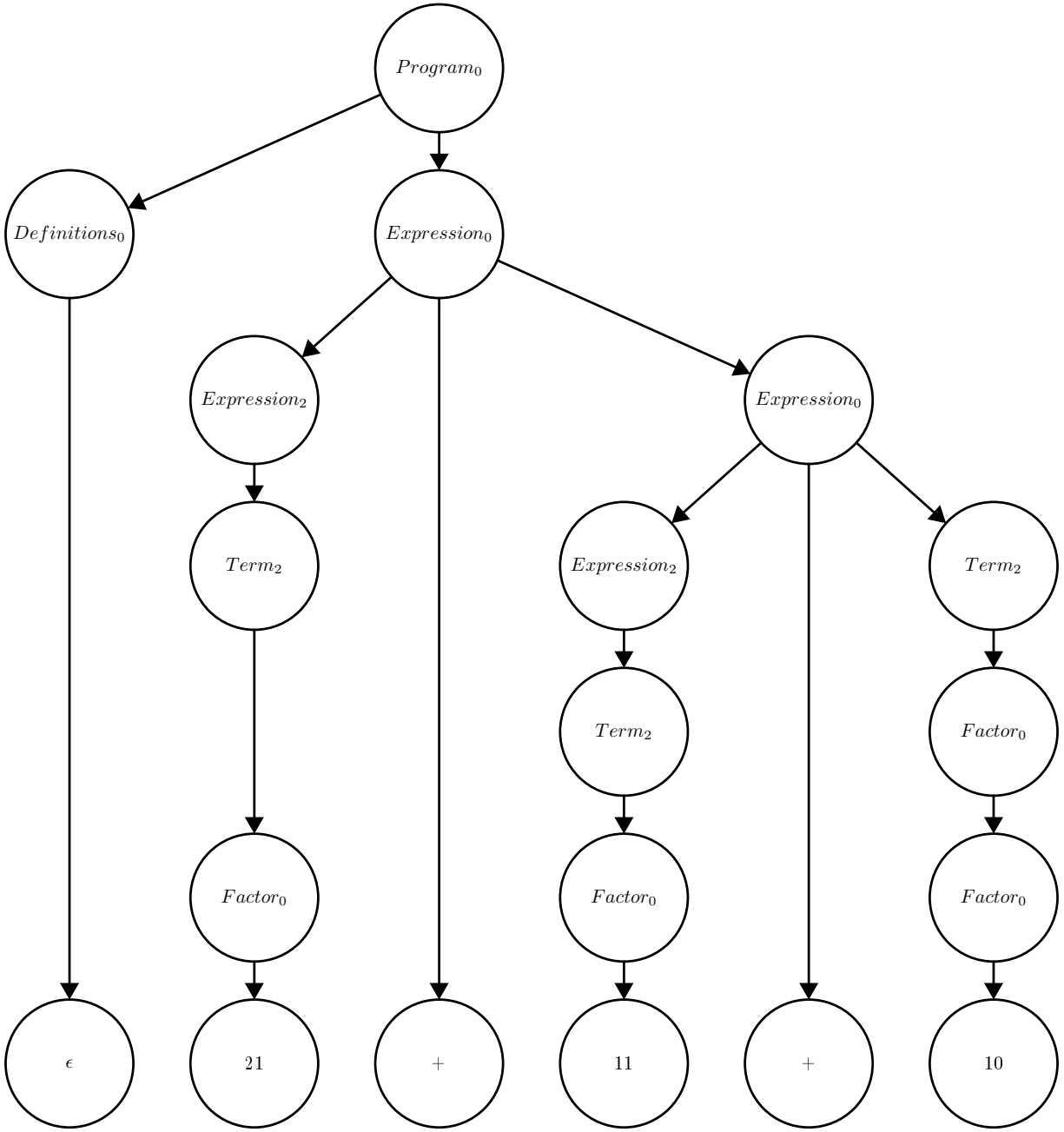
- On modifie la grammaire de MARTHA de la façon suivante :

$$
\begin{array}{lcl}
 & \ldots & \\
\textit{Expression} & \rightarrow & \textit{Expression} \text{ `+` } \textit{Expression} \\
\textit{Expression} & \rightarrow & \textit{Expression} \text{ `−` } \textit{Expression} \\
\textit{Expression} & \rightarrow & \textit{Term} \\
 & \ldots &
\end{array}
$$

Exemple : une réelle ambiguïté



Exemple : une réelle ambiguïté



Ambiguïté

- Une grammaire G est ambiguë si un mot est dérivable par deux arbres de production différents.
- Dans l'exemple précédent, on a reconnu deux parenthésages différents de l'expression « 21 + 11 + 10 » : « (21 + 11) + 10 » et « 21 + (11 + 10) ».
- Ce type d'ambiguité est bénigne dans le sens où les deux façons d'analyser l'expression conduisent à une même interprétation.
- Cependant, certaines ambiguités ne le sont pas : les expressions « 21 - 11 - 10 » et « (21 - 11) - 10 » valent 0 tandis que « 21 - (11 - 10) » vaut 20.

Spécification du problème de la reconnaissance

- Le problème de l'analyse syntaxique se spécifie généralement ainsi :
  - Entrées :
    Une grammaire $G \equiv (T, N, R, S)$, une entrée $I \in (T \cup N)^*$.
  - Précondition :
    La grammaire G est non ambiguë.
  - Sortie :
    Un arbre de production t ou bien une erreur.
  - Postcondition :
    Si la sortie est un arbre t alors il correspond à la dérivation [5] de I par les règles de G.
    Sinon, il n'existe pas de dérivation de I par les règles de G.

5. Plus précisément à un ensemble de dérivations équivalentes

Spécification du problème de la reconnaissance

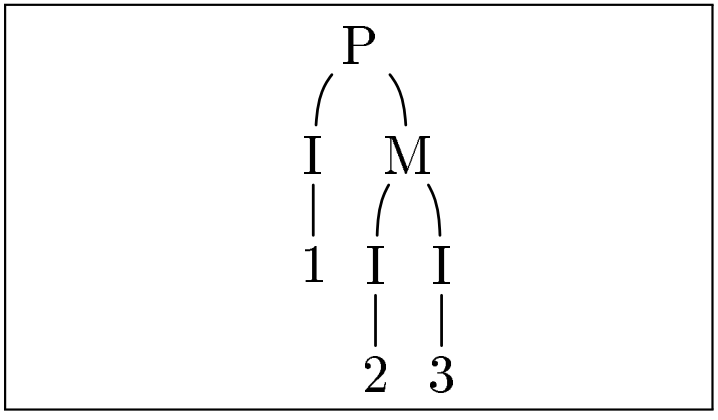
- Le problème de l'analyse syntaxique peut aussi se spécifier ainsi :
  - Entrées :
    Une grammaire $G \equiv (T, N, R, S)$, des règles de suppression des ambiguités, une entrée $I \in (T \cup N)^*$.
  - Précondition :
    Pas de précondition.
  - Sortie :
    Un arbre de production F ou bien une erreur.
  - Postcondition :
    Si la sortie est un arbre de production alors il existe plusieurs dérivations de I par les règles de G mais les règles de suppression des ambiguïtés ont permis d'effectuer une discrimination parmis ces dérivations non équivalentes.
    Sinon, il n'existe pas de dérivation de I par les règles de G.

Exemple de règles de suppression des ambiguités

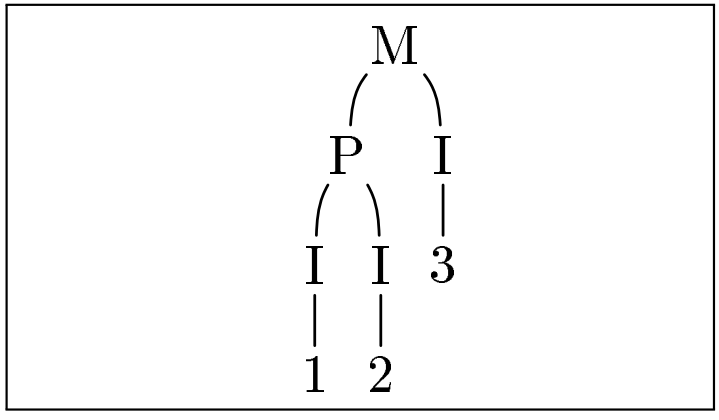
- Soit la grammaire suivante :

$$
\begin{array}{lcll}
E & \rightarrow & i & (\mathrm{I}) \\
 & | & E \text{ `*` } E & (\mathrm{M}) \\
 & | & E \text{ `+` } E & (\mathrm{P})
\end{array}
$$

- Voici deux arbres de syntaxe abstraite obtenus pour l'entrée « 1 + 2 * 3 » :



Dérivation « PIMII »



Dérivation « MPIII »

- La règle « la multiplication a une priorité plus forte que l'addition » s'exprime en termes de l'arbre de production :

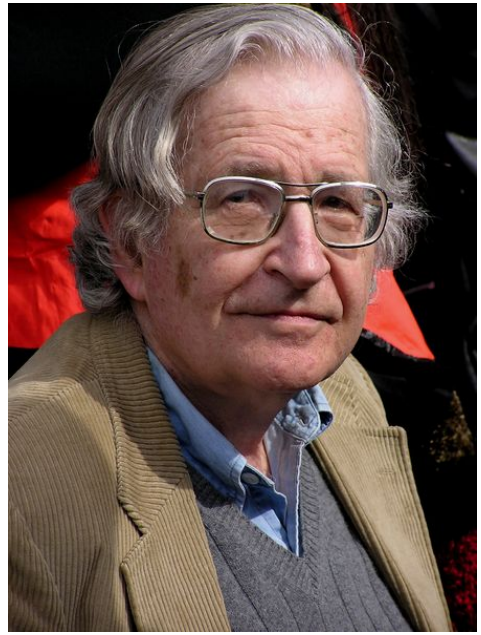
  « Un nœud "P" doit ne doit pas être le successeur direct d'un nœud "M". »

Exemple de règle de suppression des ambiguïtés

Exercice

Comment rajouter les expressions parenthésées au langage précédent ? Est-ce que la règle de suppression des ambiguïtés fonctionne toujours ?

Hiérachie de Chomsky



- Dans les années 60, Noam Chomsky a défini une classification des grammaires génératives s'appuyant sur des restrictions progressives de la syntaxe des règles dans le but de faciliter leur utilisation sans pour autant restreindre de façon déraisonnable leur expressivité (leur pouvoir génératif).
- Les grammaires de type 0 sont non restreintes.

Grammaire de type 1

- Il y a deux définitions équivalentes des grammaires de type 1.
- Une grammaire est de type 1 (monotone) si elle ne contient aucune règle dont le membre gauche est plus long que le membre droit.
- Une grammaire est de type 1 (dépendante du contexte) si ses règles peuvent dépendre du contexte. Une règle est dépendante du contexte si elle est de la forme « $uSw \rightarrow uvw$ » où u, v et w sont des séquences de symboles (terminaux ou non terminaux) et S est un non terminal.

$\Rightarrow$ Trouver un exemple simple de langage qui ne peut pas être généré par une grammaire de type 1 est difficile.

Grammaire de type 2

- Nous nous intéresserons surtout aux grammaires de type 2 car ce sont celles utilisées pour définir la syntaxe des langages de programmation.
- Une grammaire est type 2 si toutes ses règles sont de la forme « $N \rightarrow w$ » où w est une séquence de symboles (terminaux et non terminaux) et N est un non terminal.
- C'est une grammaire de type 1 dont toutes les règles ont des contextes vides.
- On les appelle des grammaires hors-contexte.
- Dans le graphe de production, ce que produit un nœud est indépendant de ce que produisent ses voisins : le graphe de production est donc un arbre.

Grammaire de type 2

- On peut partitionner les règles d'une grammaire de type 2 suivant le non-terminal à gauche de la flèche. Ce sont les règles qui définissent ce non-terminal.
- Soit un non-terminal A. Les règles de A peuvent être regroupées en une règle de la forme :

$$A \quad \rightarrow \quad w_1 \mid \ldots \mid w_n$$

où chaque w_i est de la forme $v_1 \ldots v_{k_i}$.

- Dès lors, on peut définir le langage associé à ce non-terminal A :

$$\mathcal{L}_A \quad = \quad \mathcal{L}_{w_1} \cup \ldots \cup \mathcal{L}_{w_n}$$

où $\mathcal{L}_{w_1} = \mathcal{L}_{v_1} \ldots \mathcal{L}_{v_{k_i}}$, c'est-à-dire la concaténation des langages des symboles $(v_i)_{i \in 1 \ldots k_i}$.

- Le langage $\mathcal{L}_T$ associé à un terminal T est bien sûr « $\{T\}$ ».

Définition récursive

- La restriction des grammaires hors-contexte n'interdit pas l'utilisation récursive d'un non-terminal A dans sa définition.
- Ainsi, on peut décider si :
  - un non-terminal A est récursif à gauche, c'est-à-dire si il peut produire une chaîne commençant par le non-terminal A ;
  - un non-terminal A est récursif à droite, c'est-à-dire si il peut produire une chaîne se concluant par le non-terminal A ;
  - un non-terminal A est auto-imbriqué, c'est-à-dire si il peut produire une chaîne dans laquelle le terminal A se situe à gauche et à droite de deux séquences non vides.
- Les grammaires hors-contextes permettent donc d'exprimer l'imbrication, une construction très courante des syntaxes de langage de programmation.

Notation BNF pour les grammaires hors-contextes

- La notation BNF (Backus-Naur Form) est utilisée traditionnellement pour représenter les grammaires hors-contextes. Il en existe de très nombreuses variantes.
- Les non-terminaux sont écrits entre chevrons : « <T> ».
- Les flèches sont remplacées par le symbole « ::= ».
- On regroupe les règles comme indiqué plus tôt.
- Il existe une version étendue de BNF (notée souvent EBNF) qui inclut les opérateurs "*" (zéro ou plusieurs fois), "+" (au moins une fois) et " ?" (au plus une fois), ainsi que la possibilité de parenthéser les sous-séquences auxquelles on désire appliquer ces opérateurs.

$\Rightarrow$ Toutes ces notations sont équivalentes.

Spécification YACC de grammaire hors-contexte

- Dans les années 70, de nombreuses recherches ont porté sur le problème de reconnaissance des langages décrits par les grammaires hors-contexte pour les appliquer à la construction des compilateurs.
- L'outil YACC (*Yet Another Compiler Compiler*) a été développé à l'époque en s'appuyant sur l'algorithme d'analyse syntaxique appelé LALR(1), que nous aborderons dans ce cours.
- Cet algorithme ne traite pas la totalité des grammaires hors-contextes mais la classe des langages LALR(1) est un bon compromis entre expressivité et efficacité de l'algorithme d'analyse.
- Nous utiliserons une version plus moderne de YACC appelée MENHIR.
- MENHIR est dédié au langage OCAML et reconnaît une classe plus large de langages, les langages LR(1).

Spécification YACC de grammaire hors-contexte

- YACC a la spécification suivante :
  - Entrées :
    Une grammaire hors-contexte G, un ensemble de règle de suppression d'ambiguité.
  - Précondition :
    Pas de précondition.
  - Sortie :
    Un programme qui implémente un analyseur syntaxique pour G ou bien une liste de conflits.
  - Postcondition :
    Si la sortie est un programme alors cela signifie que la grammaire est dans la classe des langages LALR(1) et que les règles de suppression des ambiguités sont complètes. Sinon, cela signifie que les règles de suppression des ambiguités ne sont pas suffisantes ou que la grammaire n'est pas dans la classe LALR(1).

Transformations de grammaire

- Plusieurs grammaires peuvent reconnaître le même langage.
- Il est parfois nécessaire de transformer une grammaire pour la rendre compatible avec la classe de grammaires traitée par un algorithme.
- Ces transformations nécessitent souvent l'ajout de nouveaux non-terminaux et la décomposition de certaines règles.

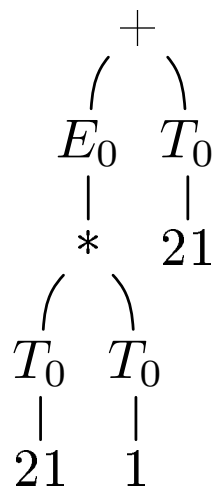
$\Rightarrow$ On s'éloigne parfois énormément de la grammaire initiale, la plus naturelle.

- La syntaxe d'un langage de programmation est ainsi souvent décrite par une grammaire hors-contexte de spécification qui définit les différentes catégories syntaxiques du langage. Cette grammaire est à destination du programmeur puisqu'elle lui donne une vue synthétique des programmes syntaxiquement corrects. Elle sert aussi à définir la forme des arbres de syntaxe abstraite sur lesquels s'appuient les phases suivantes du compilateur.
- L'analyse syntaxique utilise une grammaire d'analyse compatible avec les algorithmes d'analyse syntaxique connus et reconnaissant le même langage que la grammaire de spécification.

De l'arbre de production à l'arbre de syntaxe abstraite

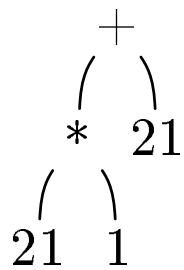
- La remarque précédente implique que l'arbre de production issu de l'analyse syntaxique fait référence aux règles de la grammaire d'analyse et est donc différent de l'arbre de syntaxe abstraite qui fait référence à la grammaire de spécification.
- Exemple :

$$
\begin{array}{lcl}
E & ::= & T \\
 & | & E \text{ `+` } T \\
T & ::= & F \\
 & | & T \text{ `*` } F \\
F & ::= & i
\end{array}
$$



?

$$
\begin{array}{lcl}
E & ::= & i \\
 & | & E \text{ `*` } E \\
 & | & E \text{ `+` } E
\end{array}
$$



Action sémantique

- En pratique, peu de systèmes produisent *explicitement* l'arbre de production.
- Les outils de la famille YACC exigent une action sémantique associée à chaque règle. Elle sert à calculer une valeur sémantique associée à chaque non-terminal reconnu en utilisant les valeurs sémantiques des non-terminaux intervenant de la règle qui le produit.
- En MENHIR, on écrit ainsi :

```
expr: lhs=expr PLUS rhs=term
{ (* L'action sémantique est une expression OCaml. *)
  (* Elle peut faire référence à [lhs] et [rhs], *)
  (* les valeurs sémantiques des deux sous-expressions. *)
  (* Ici, on utilise le constructeur [Add] du type [ast]. *)
  Add (lhs, rhs)
}
```

Format des spécifications écrites à l'aide de Menhir

```
%{
  (* Ici le prélude, c'est-à-dire du code OCaml qui peut définir *)
  (* des types de données et des fonctions utilisés dans les *)
  (* actions sémantiques. *)
%}

(* Déclarations de la grammaire. *)
%token T (* Définit un token T. *)
%token<int> I (* Définit un token I auquel est attaché un entier. *)

%start e (* Déclare le non-terminal "e" d'entrée de la grammaire. *)
       (* Menhir accepte plusieurs points d'entrée. *)
%type<int> e (* Déclare le type de la valeur sémantique de e. *)

(* Des priorités entre des règles (explications plus tard) *)
%left p
%right u v
%nonassoc k
%%

(* Les règles de la grammaire. *)
e :  T x=e y=I { x + y }
  | T T x=e y=I { x * y }

%{
  (* Postlude *)
%}
```

Exemple de spécification écrite à l'aide de MENHIR

```
%{
  (* Abstract Syntax Tree. *)
  type exp =
    | Int of int
    | Add of exp × exp
    | Mult of exp × exp
%}

(* The lexing phase will produce the following tokens: *)
%token<int> INT
%token PLUS STAR EOF

(* Here are the declarations of non terminal symbols: *)
%type<exp> top_exp
%start top_exp

%%
(* Now, we are defining the rules of the grammar: *)
top_exp : e=exp EOF { e }

exp : x=INT { Int x }
| lhs=exp PLUS rhs=exp { Add (lhs, rhs) }
| lhs=exp STAR rhs=exp { Mult (lhs, rhs) }
```

Utilisation de Menhir

- On utilise la commande :

```
% menhir –explain arith.mly
Warning: 2 states have shift/reduce conflicts.
Warning: 4 shift/reduce conflicts were arbitrarily resolved.
```

⇒ Menhir nous informe qu'il a bien réussi à interpréter notre grammaire mais en faisant des choix arbitraires pour résoudre certaines ambiguités.

- Menhir a produit :
  - un fichier `arith.ml` qui réalise l'analyse syntaxique ;
  - un fichier `arith.mli` qui définit son interface ;
  - un fichier `arith.conflicts` qui explique comment la résolution des conflits a été faite.

⇒ Il y a très peu de chances pour que ce soit ces choix arbitraires soient les bons. Regardons ce dernier fichier !

Explication d'un conflit particulier par MENHIR

```
** Conflict (shift/reduce) in state 7.
** Tokens involved: STAR PLUS
** The following explanations concentrate on token STAR.
** This state is reached from top_exp after reading:

exp PLUS exp

** The derivations that appear below have the following common factor:
** (The question mark symbol (?) represents the spot where the derivations begin to
differ.)

top_exp
exp EOF
(?)

** In state 7, looking ahead at STAR, reducing production
** exp → exp PLUS exp
** is permitted because of the following sub-derivation:

exp STAR exp // lookahead token appears
exp PLUS exp .

** In state 7, looking ahead at STAR, shifting is permitted
** because of the following sub-derivation:

exp PLUS exp
          exp . STAR exp
```

Fonctionnement d'un analyseur produit par MENHIR

- Un analyseur syntaxique produit par MENHIR lit le flot de lexèmes de gauche à droite.

$\Rightarrow$ Il fait partie de la famille des analyseurs directionnels.

- Il essaie de reconstruire l'arbre de production en partant des feuilles.

$\Rightarrow$ C'est une analyse ascendante.

- Pour cela, il maintient une pile contenant les sous-arbres de l'arbre de production déjà reconnus et prend une décision en fonction du sommet de cette pile et du lexème suivant.

$\Rightarrow$ L'analyseur est un automate à pile déterministe qui lit un lexème en avant.

- Il y a deux types de décision :
  - La décision "avance" (*shift*) : on pousse le lexème suivant sur la pile.
  - La décision "réduit" (*reduce*) : on dépile N sous-arbres de la pile pour en construire un autre que l'on pose au sommet de la pile.

Résolution d'un conflit par MENHIR

- Le conflit précédent s'explique ainsi :
- Après avoir reconnu « exp PLUS exp » (trois éléments sur la pile), si le prochain lexème est « STAR » alors, MENHIR ne sait pas choisir entre :
  - L'action "avance" : on empile *STAR*.
  - L'action "réduit" : on construit une nouvelle expression correspondant au sous-arbre « exp + exp » que l'on met sur la pile.
- De toute évidence, la seconde action n'est pas acceptable.
- Un autre conflit apparaît après avoir lu « exp STAR exp » et si le prochain lexème est « PLUS ». Dans ce cas, on veut plutôt réduire.

$\Rightarrow$ La règle de réduction de "*" doit avoir une priorité plus forte que celle de "+". Il suffit d'écrire :

```
(* On utilise le terminal le plus à droite de la règle *)
(* pour parler de celle-ci. Nous verrons un mot-clé %prec, plus élégant. *)
%nonassoc PLUS (* Les règles du haut sont moins prioritaires ... *)
%nonassoc STAR (* ... que les règles du bas. *)
```

$\Rightarrow$ 2 conflits sont traités sur 4.

Résolution des conflits par Menhir

- Les 2 autres conflits parlent de l'interaction entre PLUS/PLUS et STAR/STAR.

Exercice

Sur quoi porte ces conflits ? Quel effet a eu le mot-clé `%nonassoc` ?

Grammaire de type 3

- Les grammaires hors-contexte décrivent des langages dans lesquels les mots peut-être imbriqués. Lorsque l'on analyse un sous-mot d'un mot de ces langages, la forme des règles de la grammaire nous permet de nous souvenir de ce qui doit suivre le sous-mot une fois qu'il sera reconnu.
- La restriction des grammaires de type 3 supprime cette mémoire.
- On formalise cette restriction de la façon suivante :

  Une règle ne peut produire qu'un ou plusieurs terminaux suivis d'un unique non terminal optionnel.

$\Rightarrow$ En d'autres termes, les règles doivent être linéaires droites.

- Ces grammaires correspondent aux langages rationnels (ou réguliers) que vous avez déjà étudiés.
- Les automates finis sont les outils de prédilection pour la construction des analyseurs syntaxiques de ces langages.
- L'expressivité des langages rationnels n'est en général pas suffisante pour décrire la syntaxe des langages de programmation. Par contre, elle suffit à l'analyse lexicale.

Grammaire de type 4

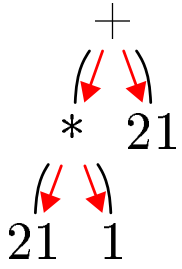
- La restriction des grammaires de type 4 interdit la présence d'un non-terminal à droite de la flèche.

$\Rightarrow$ Ces grammaires représentent des langages finis.

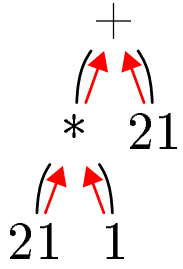
- Elles fournissent un cadre pour décrire des énumérations.

Deux directions possibles pour l'analyse syntaxique

- Rappel :
  Le problème de l'analyseur syntaxique est de construire l'arbre de production d'un mot w par une grammaire non ambiguë G.
- Il y a deux grandes techniques pour résoudre ce problème :
  - L'analyse descendante essaie d'imiter la dérivation (supposée) du mot w en partant du symbole d'entrée de la grammaire G en reconstruisant l'arbre de production de sa racine vers ses feuilles.
  - L'analyse ascendante tente une reconstruction de la dérivation à partir des feuilles de l'arbre (le mot w) en remontant vers la racine.



Analyse descendante



Analyse ascendante

Exemple d'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* »

Ⓢ

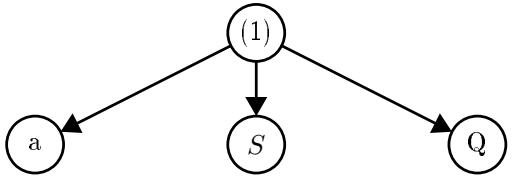
- L'analyse débute avec celle du symbole S_s.
- Quelle peut être la prochaine étape ?

Exemple d'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* »



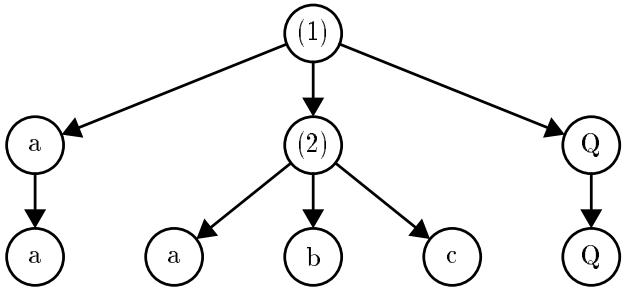
- Comme « abc » n'est pas un préfixe de l'entrée, la règle (2) ne s'applique pas.
- Seule la règle (1) peut s'appliquer.

Exemple d'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* »



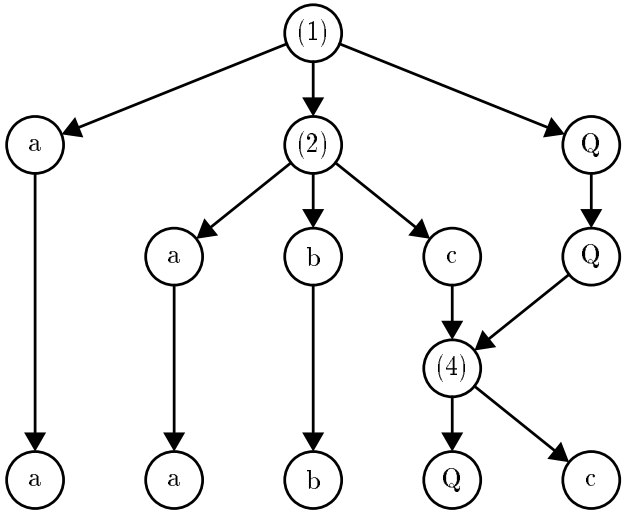
- Appliquer de nouveau la règle (1) imposerait le préfixe « aaa » dans l'entrée.
- On doit donc considérer la règle (2).

Exemple d'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* »



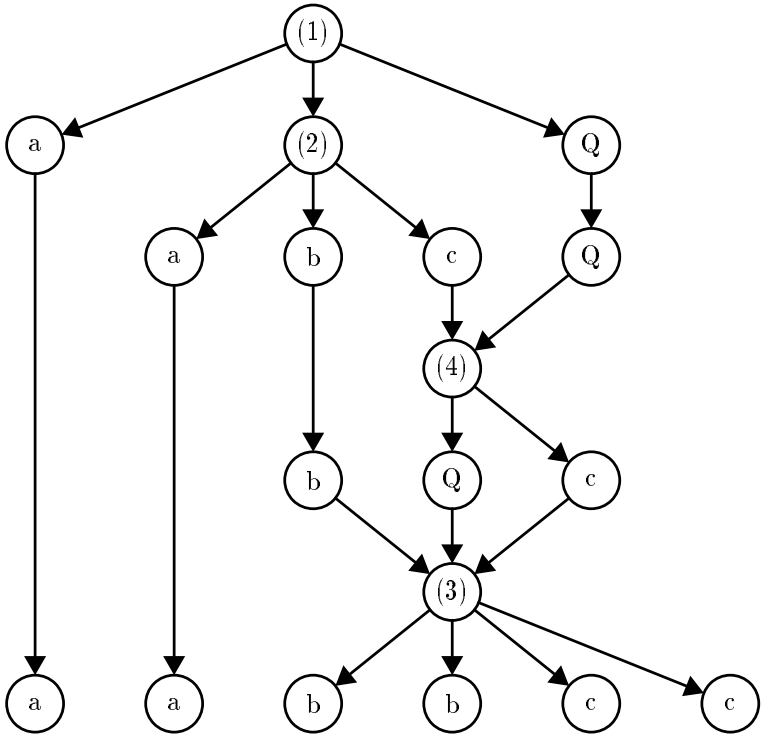
- Seule la règle (4) s'applique.
  (C'était difficile à prévoir !)

Exemple d'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.

Exemple d'analyse ascendante

- 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* »

ⓐ ⓐ ⓑ ⓑ ⓒ ⓒ

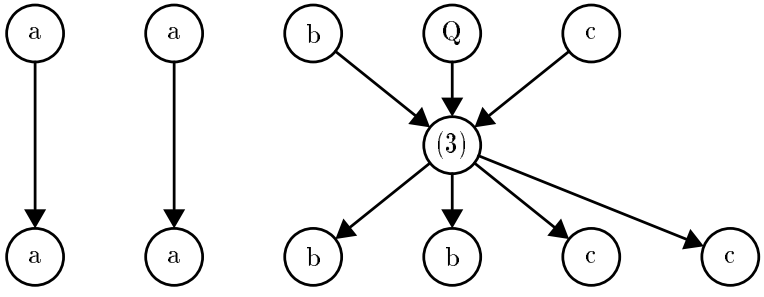
- Quelle règle peut produire une sous-séquence de non-terminaux de ce mot ?

Exemple d'analyse ascendante

- 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* »



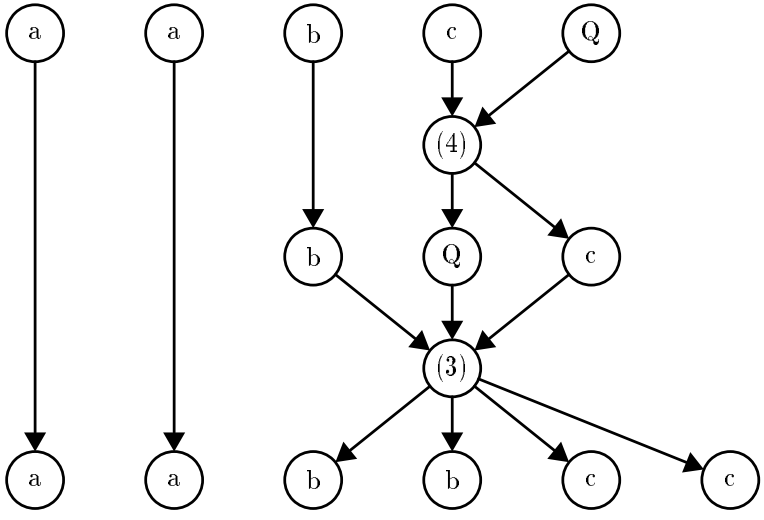
- De même, seule la règle (4) a pu produire le mot « Q c ».

Exemple d'analyse ascendante

- 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* »



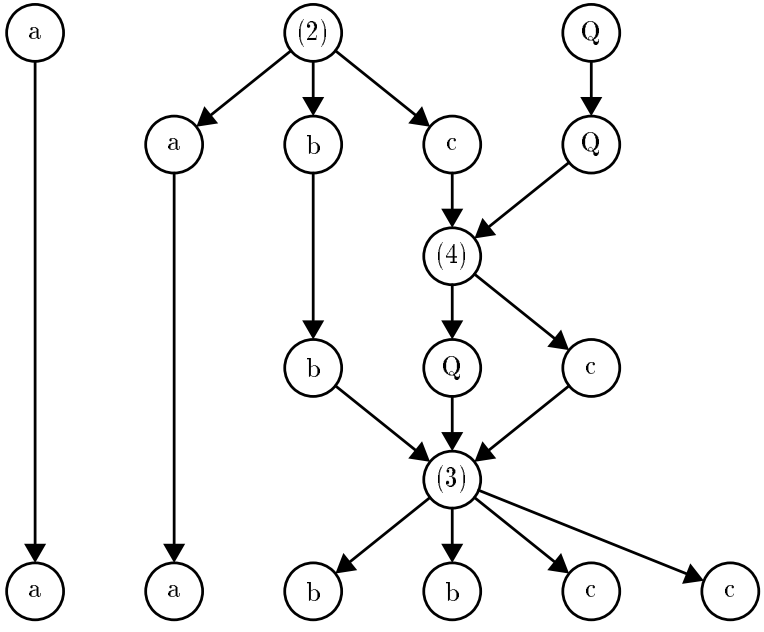
- Ici, seule la règle (3) s'applique et elle mène à l'entrée.

Exemple d'analyse ascendante

- 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* »



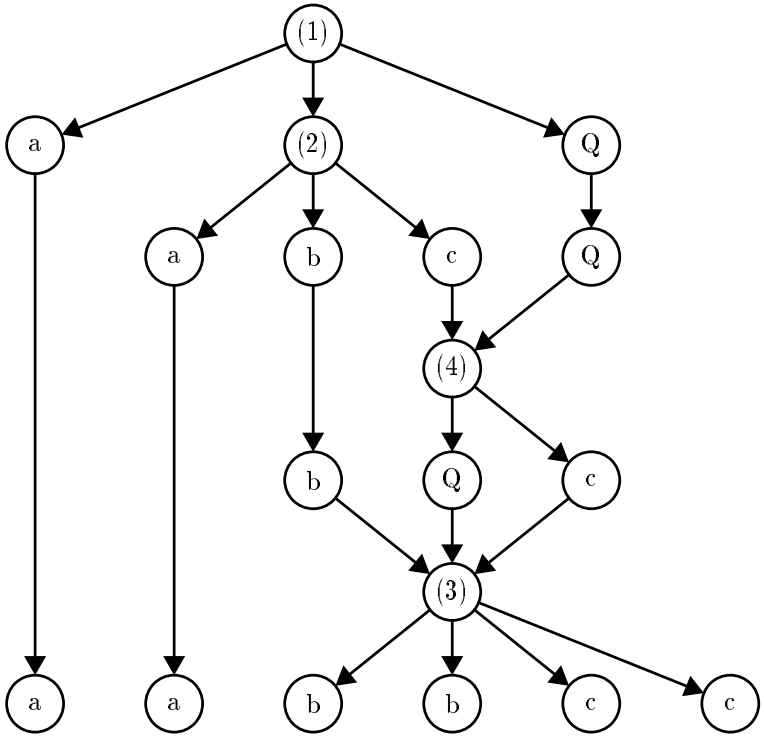
- Encore une fois, une seule règle s'applique, la règle (2).

Exemple d'analyse ascendante

- 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* »



- Pour finir, seule la règle (1) produit le mot « aSQ ».

Synthèse

Grammaire et analyse syntaxique

- Nous avons (re)vu le concept de grammaire algébrique.
- Le problème de l'analyse syntaxique consiste à construire un arbre de production à partir d'une grammaire et d'un mot qui retrace la reconnaissance de ce mot par la grammaire, c'est-à-dire sa dérivation.
- Menhir permet de spécifier des grammaires hors-contexte.
- Il ne les traite pas dans leur totalité.
- Il existe deux grands types d'analyse :

  Les analyses descendantes et ascendantes

$\Rightarrow$ La définition de certains algorithmes qui les réalisent sera l'objet du prochain cours.

Bibliographie

- *Parsing techniques, A pratical guide*, Grune, Jacobs.
  VU University Amsterdam, Amsterdam, The Netherlands
  (Première édition est disponible en ligne : `http://www.few.vu.nl/~dick/PTAPG.html`)
- *Théorie des langages*, Demaille, Yvon.
  Notes de cours.
  (`http://www.lrde.epita.fr/~akim/thl/theorie-des-langages-1.pdf`)

QCM : Question 1

L'analyse syntaxique :

a. produit un flux de lexème.

b. peut être ascendante ou descendante.

c. s'appuie sur les grammaires de type 1.

d. peut utiliser un automate (à pile).

QCM : Question 2

On peut résoudre un conflit dans une spécification de grammaire :

a. en modifiant le langage à reconnaître.

b. en réécrivant la grammaire en une grammaire équivalente.

c. en renommant des non-terminaux.

d. en spécifiant des priorités entre les règles.

QCM : Question 3

Un graphe de production est :

a. l'arbre de syntaxe abstraite.
b. un graphe potentiellement cyclique.
c. un graphe acyclique.
d. formé de nœuds qui dénotent des applications de règles de la grammaire.
e. formé d'arêtes étiquetées par des noms de règles de la grammaire.

QCM : Question 4

L'analyse lexicale produit :

a. un non-terminal.
b. un flux de non-terminaux.
c. un flux de terminaux.
d. un flux de caractères.

QCM : Question 5

En parcourant l'arbre de production :

a. suivant un parcours préfixe, on obtient une dérivation droite.
b. suivant un parcours préfixe, on obtient une dérivation gauche.
c. suivant un parcours infixe, on obtient une dérivation gauche.
d. suivant un parcours infixe, on obtient une dérivation droite.