SÉANCE 11

INTRODUCTION À L'ANALYSE SYNTAXIQUE

SUJETS

Grammaires non-contextuelles

Dérivations

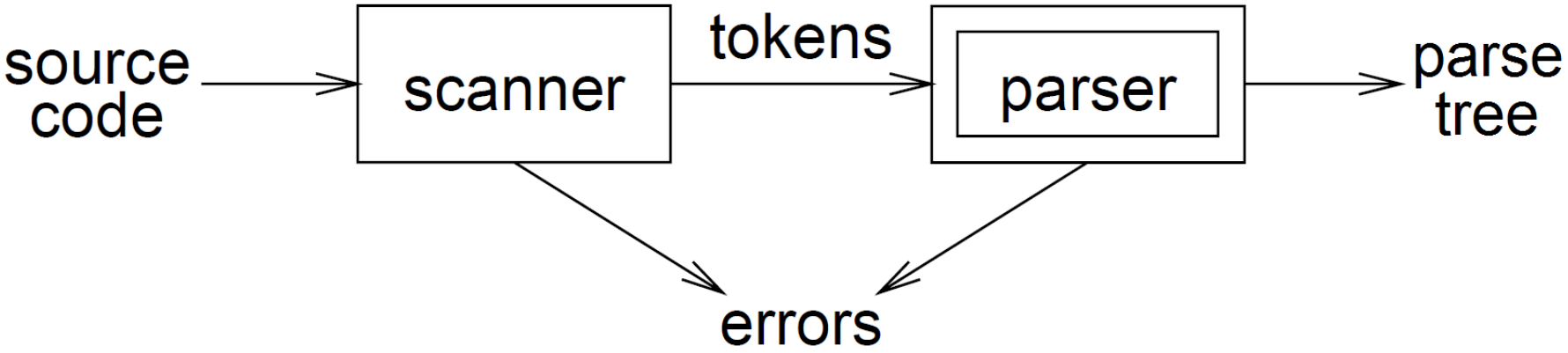
Arbres de parsage

Ambiguïté

Parseur descendant

Récursivité gauche

LE RÔLE DE L'ANALYSEUR SYNTAXIQUE



GRAMMAIRES NON-CONTEXTUELLES

uOttawa
L'Université canadienne
Canada's university

Une grammaire non-contextuelles (GNC) se consiste de:

- Terminaux
- Non-terminaux
- Symbole de début
- Productions

Un langage qui peut être généré par une grammaire non-contextuelle est appelé un langage non-contextuel

GRAMMAIRES NON-CONTEXTUELLES

uOttawa
L'Université canadienne
Canada's university

Terminaux: sont les symboles de base d'où les chaînes sont formées

- Ceux-ci sont les tokens qui ont été produits par l'Analyseur Lexical

Non-terminaux: sont des variables syntactiques qui dénote une série de symboles de grammaire

- Un de ces non-terminaux est distingué comme étant le **symbole de départ**

Les productions d'une grammaire spécifient la manière dont le terminal et non-terminal peuvent être combinés afin de former des chaînes

EXEMPLE DE GRAMMAIRE

uOttawa
L'Université canadienne
Canada's university

La grammaire avec les productions suivantes définit des expressions arithmétiques simples

```
⟨expr⟩  ::=  ⟨expr⟩  ⟨op⟩  ⟨expr⟩
⟨expr⟩  ::= id
⟨expr⟩  ::= num
  ⟨op⟩  ::= +
  ⟨op⟩  ::= -
  ⟨op⟩  ::= *
  ⟨op⟩  ::= /
```

Dans cette grammaire, les symboles terminaux sont : *num, id* + - * /

Les symboles non-terminaux sont ⟨*expr*⟩ et ⟨*op*⟩ , et ⟨*expr*⟩ est le symbole de départ

DÉRIVATIONS

⟨expr⟩ ⇒ ⟨expr⟩ ⟨op ⟩ ⟨expr⟩

est lue *"expr dérivés expr op expr"*

⟨expr⟩ ⇒ ⟨expr⟩ ⟨op ⟩ ⟨expr⟩
⇒ id ⟨op ⟩ ⟨expr⟩
⇒ id * ⟨expr⟩
⇒ id*id

est appelée une dérivation de id*id d'après *expr*.

DÉRIVATIONS

Si $A::=\gamma$ est une production et α et β sont des chaînes arbitraires de symboles de grammaire, on peut dire que:

$$\alpha A \beta \Rightarrow \alpha\gamma\beta$$

Si $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$, on dit que α_1 *dérive* α_n.

DÉRIVATIONS

$\Rightarrow$ veut dire "dérive en une seule étape."

$\overset{*}{\Rightarrow}$ veut dire "dérive en zéro ou plusieurs étapes."

- *si* $\alpha \overset{*}{\Rightarrow} \beta$ *et* $\beta \Rightarrow \gamma$ *donc* $\alpha \overset{*}{\Rightarrow} \gamma$

$\overset{+}{\Rightarrow}$ veut dire "dérive en une ou plusieurs étapes."

Si $S \overset{*}{\Rightarrow} \alpha$, où α peut contenir des non-terminaux, alors on dit que α est en forme phrastique

- Si α ne contient pas des non-terminaux, on dit que α est une phrase

DÉRIVATIONS

G: grammaire

S: symbole de départ

L(G): le langage généré par G

Les chaînes dans L(G) peuvent contenir uniquement des symboles terminaux de G

Une chaîne de terminaux *w* appartient à L(G) si et seulement si $S \overset{+}{\Rightarrow} w$

- Comme on a déjà mentionné, la chaîne *w* est appelée une phrase de G

Un langage qui peut être généré par une grammaire est connu comme étant un langage non-contextuel

- Si deux grammaires génèrent le même langage, les grammaires sont considérées équivalentes

DÉRIVATIONS

Nous avons déjà vu les règles de production suivantes:

⟨expr⟩ ::= ⟨expr⟩ ⟨op⟩ ⟨expr⟩ | id | num

⟨op⟩ ::= + | - | * | /

La chaîne `id+id` est une phrase de grammaire ci-haut parce que

⟨expr⟩ ⇒ ⟨expr⟩ ⟨op⟩ ⟨expr⟩

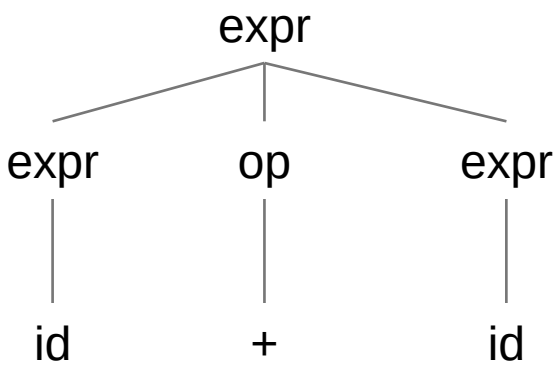
⇒ id ⟨op⟩ ⟨expr⟩

⇒ id + ⟨expr⟩

⇒ id + id

On écrit $\langle expr \rangle \overset{*}{\Rightarrow} id+id$

ARBRE DE PARSAGE



Ceci est appelé:
Dérivation la plus à gauche

DEUX ARBRES DE PARSAGE

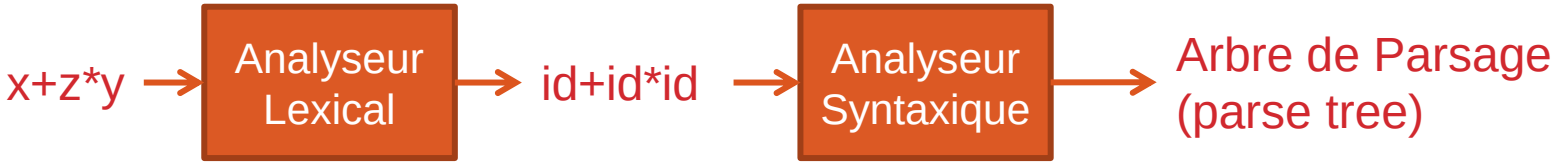
uOttawa
L'Université canadienne
Canada's university

Considérons encore une fois l'expression arithmétique de grammaire.

Pour la ligne de code:

x+2*y

(nous ne considérons pas le point-virgule pour le moment)



Grammaire:

```
⟨expr⟩ ::= ⟨expr⟩ ⟨op⟩ ⟨expr⟩ | id | num
⟨op⟩ ::= + | - | * | /
```

DEUX ARBRES DE PARSAGE

uOttawa
L'Université canadienne
Canada's university

Considérons encore une fois l'expression arithmétique de grammaire.

La phrase id + id * id possède deux dérivations à gauche distinctes:

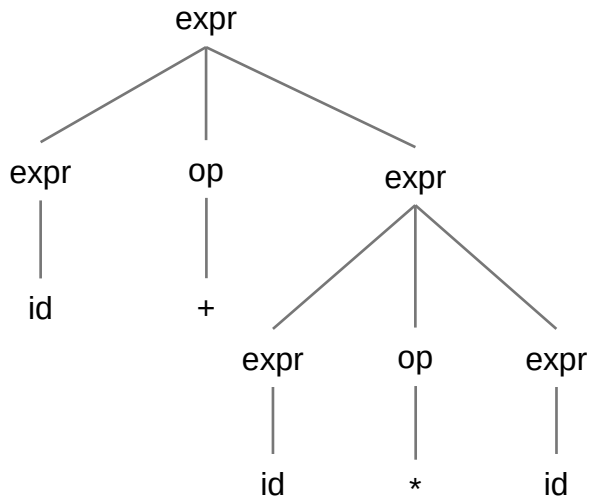
```
⟨expr⟩ ⇒ ⟨expr⟩ ⟨op⟩ ⟨expr⟩
       ⇒ id ⟨op⟩ ⟨expr⟩
       ⇒ id + ⟨expr⟩
       ⇒ id + ⟨expr⟩ ⟨op⟩ ⟨expr
⟩
       ⇒ id + id ⟨op⟩ ⟨expr⟩
       ⇒ id + id * ⟨expr⟩
       ⇒ id + id * id
```

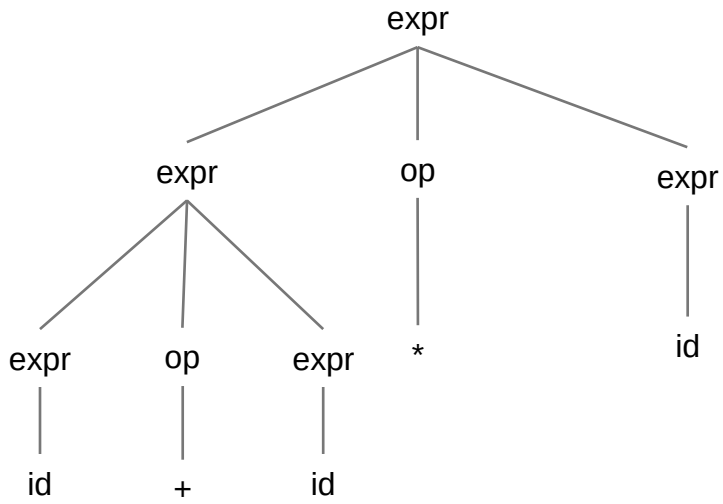
```
⟨expr⟩ ⇒ ⟨expr⟩ ⟨op⟩ ⟨expr⟩
       ⇒ ⟨expr⟩ ⟨op⟩ ⟨expr⟩ ⟨op⟩ ⟨expr⟩
       ⇒ id ⟨op⟩ ⟨expr⟩ ⟨op⟩ ⟨expr⟩
       ⇒ id + ⟨expr⟩ ⟨op⟩ ⟨expr⟩

       ⇒ id + id ⟨op⟩ ⟨expr⟩
       ⇒ id + id * ⟨expr⟩
       ⇒ id + id * id
```

Grammaire:

```
⟨expr⟩ ::= ⟨expr⟩ ⟨op⟩ ⟨expr⟩ | id | num
⟨op⟩ ::= + | - | * | /
```

DEUX ARBRES DE PARSAGE





Equivalent à:
id+(id*id)

Equivalent à:
(id+id)*id ✗

Grammaire:

```
⟨expr⟩ ::= ⟨expr⟩ ⟨op⟩ ⟨expr⟩ | id | num
⟨op⟩ ::= + | - | * | /
```

PRÉCÉDENCE

L'exemple précédent surligne un problème dans la grammaire:

- Elle n'impose pas la précédence
- Elle n'implique pas l'ordre d'évaluation

On peut améliorer les règles de production pour ajouter la précédence

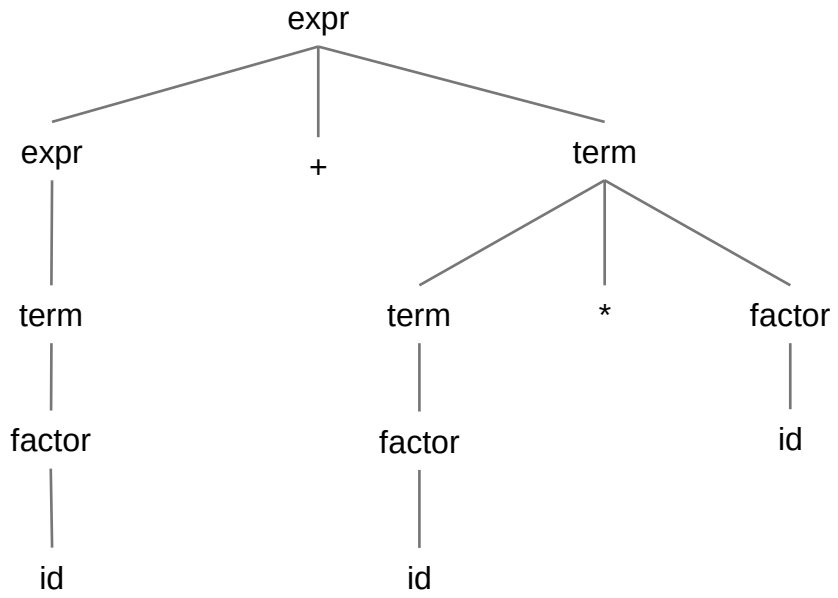
$$
\begin{array}{lcl}
\langle \text{expr} \rangle & ::= & \langle \text{expr} \rangle + \langle \text{term} \rangle \\
& | & \langle \text{expr} \rangle - \langle \text{term} \rangle \\
& | & \langle \text{term} \rangle \\
\langle \text{term} \rangle & ::= & \langle \text{term} \rangle * \langle \text{factor} \rangle \\
& | & \langle \text{term} \rangle / \langle \text{factor} \rangle \\
& | & \langle \text{factor} \rangle \\
\langle \text{factor} \rangle & ::= & \texttt{num} \\
& | & \texttt{id}
\end{array}
$$

APPLIQUER LA MISE À JOUR DE PRÉCÉDENCE

uOttawa
L'Université canadienne
Canada's university

La phrase id + id * id possède seulement une seule dérivation à gauche maintenant:

```
⟨expr⟩  ⇒  ⟨expr⟩  + ⟨term⟩
        ⇒  ⟨term⟩  + ⟨term⟩
        ⇒  ⟨factor⟩ + ⟨term⟩
        ⇒ id + ⟨term⟩
        ⇒ id + ⟨term⟩ * ⟨factor⟩
        ⇒ id + ⟨factor⟩ * ⟨factor⟩
        ⇒ id + id * ⟨factor⟩
        ⇒ id + id * id
```



Grammaire:

```
⟨expr⟩    ::= ⟨expr⟩ + ⟨term⟩ | ⟨expr⟩ - ⟨term⟩ | ⟨term⟩
⟨term⟩    ::= ⟨term⟩ * ⟨factor⟩ | ⟨term⟩ / ⟨factor⟩ | ⟨factor⟩
⟨factor⟩  ::= num | id
```

AMBIGUÏTÉ

Une grammaire qui produit plus qu'un arbre de parsage pour une phrase est considérée d'être *ambiguë*

Exemple:

$$
\begin{array}{lcl}
\langle \text{stmt} \rangle & ::= & \texttt{if } \langle \text{expr} \rangle \texttt{ then } \langle \text{stmt} \rangle \\
& | & \texttt{if } \langle \text{expr} \rangle \texttt{ then } \langle \text{stmt} \rangle \texttt{ else } \langle \text{stmt} \rangle \\
& | & \texttt{other stmts}
\end{array}
$$

Considérez la déclaration suivante:

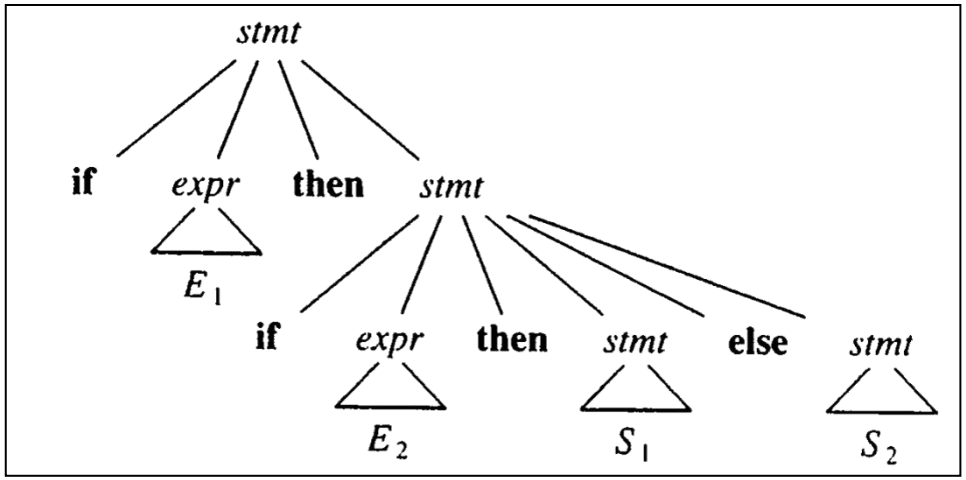
$$\texttt{if } E_1 \texttt{ then if } E_2 \texttt{ then } S_1 \texttt{ else } S_2$$

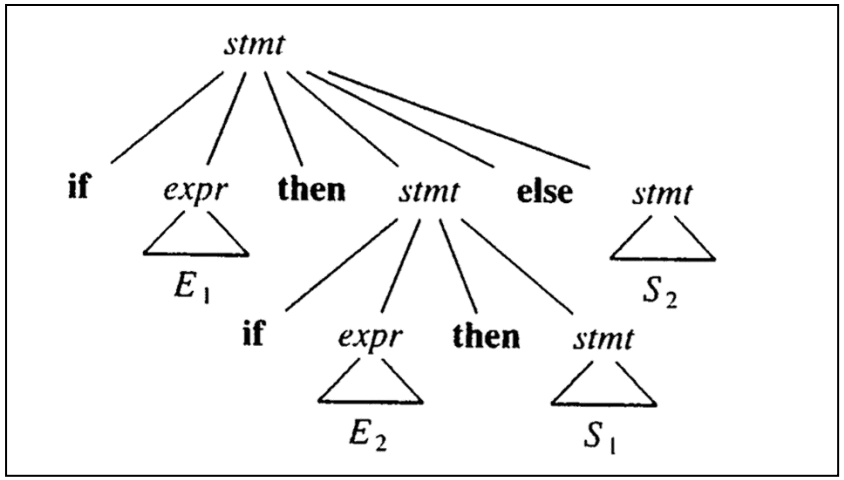
Elle possède deux dérivations

- C'est une ambiguïté non-contextuelle

AMBIGUÏTÉ

Une grammaire qui produit plus qu'un arbre de parsage pour une phrase est considérée d'être *ambiguë*





ÉLIMINER L'AMBIGUÏTÉ

uOttawa
L'Université canadienne
Canada's university

Parfois, une grammaire ambiguë peut être réécrite afin d'éliminer l'ambiguïté.

- Ex. "matcher chaque « else » avec « then » le plus proche"
- Ceci est probablement l'intention du programmeur

```
⟨stmt⟩      ::=  ⟨matched⟩
               |  ⟨unmatched⟩
⟨matched⟩   ::=  if ⟨expr⟩ then ⟨matched⟩ else ⟨matched⟩
               |  other stmts
⟨unmatched⟩ ::=  if ⟨expr⟩ then ⟨stmt⟩
               |  if ⟨expr⟩ then ⟨matched⟩ else ⟨unmatched⟩
```

APPLICATION DANS JAVA

uOttawa
L'Université canadienne
Canada's university

Dans Java, les règles de grammaire sont un peu différentes de l'exemple précédent

Une version (très simplifiée) de ces règles est présentée ci-dessous

```
<stmnt>          ::= <matched>
                 | <unmatched>
<matched>        ::= if ( <expr> ) <matched> else <matched>
                 | other stmnts
<unmatched>      ::= if ( <expr> ) < stmnt >
                 | if ( <expr> ) <matched> else <unmatched>
```

APPLICATION DANS JAVA

uOttawa
L'Université canadienne
Canada's university

Pour la portion de code suivante

```
if (x==0)
   if (y==0)
      z = 0;
   else
      z = 1;
```

Après avoir exécuter l'analyseur lexical, nous obtenons la liste de tokens suivante:

```
if ( id == num ) if (id == num) id = num ; else id = num ;
```

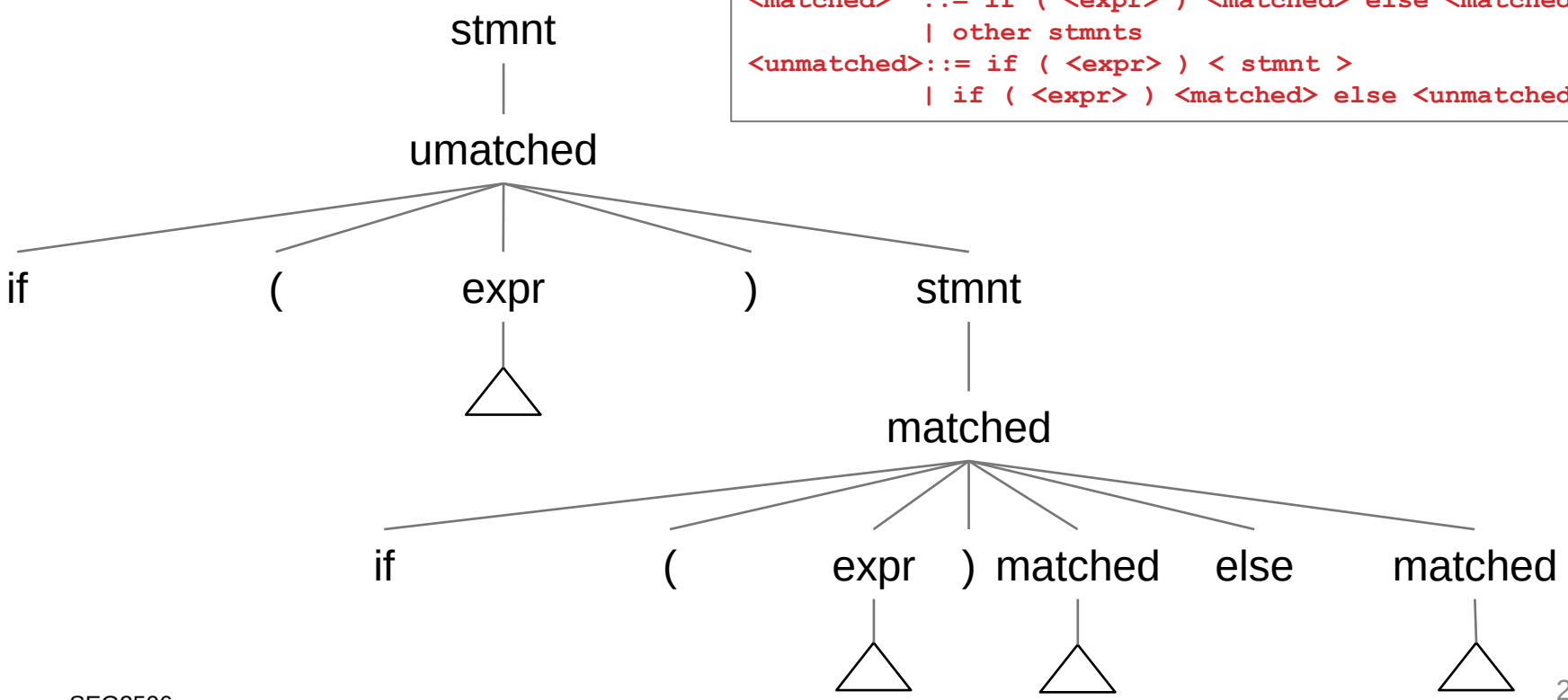
EXEMPLE JAVA

Chaîne de tokens

```
if ( id == num ) if (id == num) id = num ; else id = num ;
```

Grammaire

```
<stmnt>     ::= <matched>
              | <unmatched>
<matched>   ::= if ( <expr> ) <matched> else <matched>
              | other stmnts
<unmatched>::= if ( <expr> ) < stmnt >
              | if ( <expr> ) <matched> else <unmatched>
```



UN AUTRE EXEMPLE JAVA

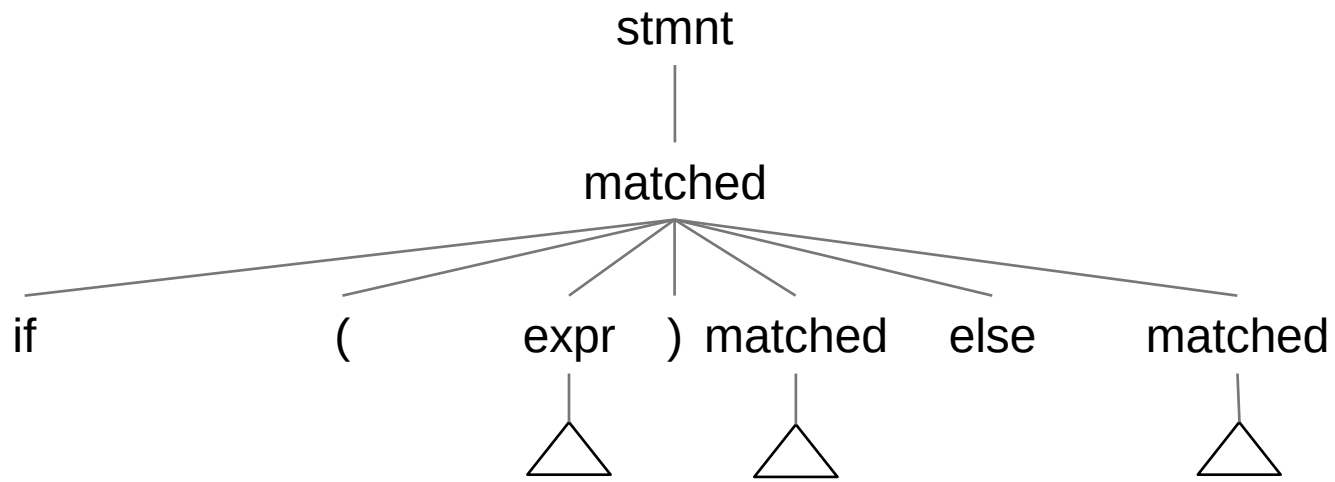
uOttawa
L'Université canadienne
Canada's university

Chaîne de tokens

```
if ( id == num ) else id = num ;
```

Grammaire

```
<stmnt>     ::= <matched>
              | <unmatched>
<matched>   ::= if ( <expr> ) <matched> else <matched>
              | other stmnts
<unmatched>::= if ( <expr> ) < stmnt >
              | if ( <expr> ) <matched> else <unmatched>
```



PARSEUR DESCENDANT

Un parseur descendant commence avec la racine de l'arbre de parsage, marquée avec le symbole de départ

Pour construire un arbre de parsage, on répète les étapes suivantes jusqu'à ce que les feuilles de l'arbre de parsage matches la chaîne de données

1. À un nœud marqué par *A*, sélectionnez une production A::=α et construisez l'enfant approprié pour chaque symbole de α
2. Lorsqu'un terminal ajouté à l'arbre de parsage ne matche pas la chaîne de données, reculer
3. Trouvez le non-terminal suivant à être développer

PARSEUR DESCENDANT

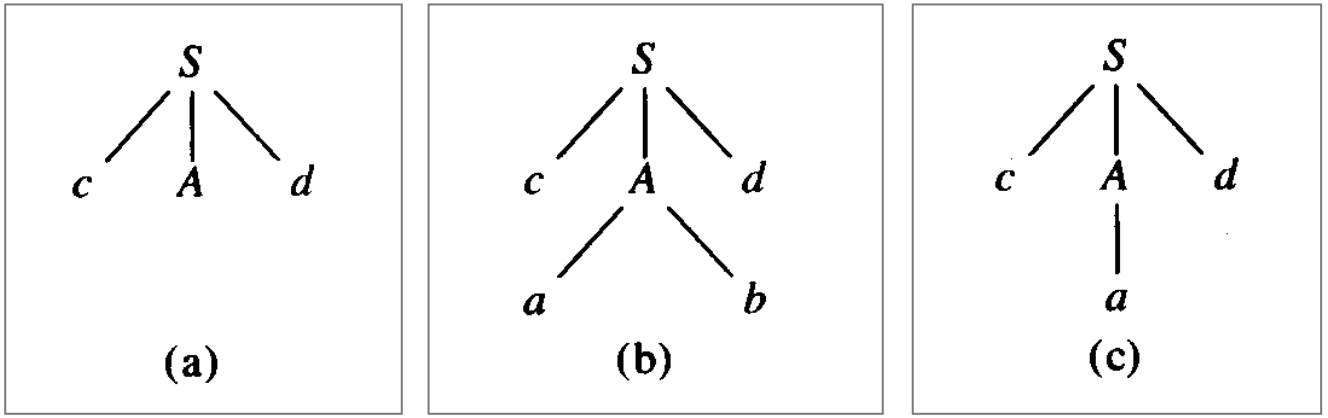
Le parsage descendant peut être considéré come une tentative pour trouver une dérivation à gauche pour une chaîne de données

Exemple:

Grammaire:

```
<S> ::= c<A>d
<A> ::= ab | a
```

Chaîne de données
`cad`



On a besoin de reculer!

GRAMMAIRE DE L'EXPRESSION

uOttawa
L'Université canadienne
Canada's university

Rappelez-vous de notre grammaire pour les expressions simples:

$$
\begin{array}{lcl}
\langle \text{expr} \rangle & ::= & \langle \text{expr} \rangle + \langle \text{term} \rangle \\
& | & \langle \text{expr} \rangle - \langle \text{term} \rangle \\
& | & \langle \text{term} \rangle \\
\langle \text{term} \rangle & ::= & \langle \text{term} \rangle * \langle \text{factor} \rangle \\
& | & \langle \text{term} \rangle / \langle \text{factor} \rangle \\
& | & \langle \text{factor} \rangle \\
\langle \text{factor} \rangle & ::= & \texttt{num} \\
& | & \texttt{id}
\end{array}
$$

Considérez la chaîne de données:

```
id - num * id
```

EXEMPLE

Prod'n	Sentential form	Input
1	⟨expr⟩	↑ `id - num * id`

Reference Grammar

⟨expr⟩	::=	⟨expr⟩ + ⟨term⟩
	|	⟨expr⟩ − ⟨term⟩
	|	⟨term⟩
⟨term⟩	::=	⟨term⟩ ∗ ⟨factor⟩
	|	⟨term⟩ /⟨factor⟩
	|	⟨factor⟩
⟨factor⟩	::=	`num`
	|	`id`

EXEMPLE

Reference Grammar

⟨expr⟩	::=	⟨expr⟩ + ⟨term⟩
	|	⟨expr⟩ − ⟨term⟩
	|	⟨term⟩
⟨term⟩	::=	⟨term⟩ ∗ ⟨factor⟩
	|	⟨term⟩ /⟨factor⟩
	|	⟨factor⟩
⟨factor⟩	::=	num
	|	id

Prod'n	Sentential form	Input
1	⟨expr⟩	↑ id − num ∗ id
2	⟨expr⟩ + ⟨term⟩	↑ id − num ∗ id
4	⟨term⟩ + ⟨term⟩	↑ id − num ∗ id
7	⟨factor⟩ + ⟨term⟩	↑ id − num ∗ id
9	id + ⟨term⟩	↑ id − num ∗ id
–	id + ⟨term⟩	id ↑ − num ∗ id
–	⟨expr⟩	↑ id − num ∗ id
3	⟨expr⟩ − ⟨term⟩	↑ id − num ∗ id
4	⟨term⟩ − ⟨term⟩	↑ id − num ∗ id
7	⟨factor⟩ − ⟨term⟩	↑ id − num ∗ id
9	id − ⟨term⟩	↑ id − num ∗ id
–	id − ⟨term⟩	id ↑ − num ∗ id
–	id − ⟨term⟩	id − ↑num ∗ id
7	id − ⟨factor⟩	id − ↑num ∗ id
8	id − num	id − ↑num ∗ id
–	id − num	id − num ↑∗ id
–	id − ⟨term⟩	id − ↑num ∗ id
5	id − ⟨term⟩ ∗ ⟨factor⟩	id − ↑num ∗ id
7	id − ⟨factor⟩ ∗ ⟨factor⟩	id − ↑num ∗ id
8	id − num ∗ ⟨factor⟩	id − ↑num ∗ id
–	id − num ∗ ⟨factor⟩	id − num ↑∗ id
–	id − num ∗ ⟨factor⟩	id − num ∗ ↑id
9	id − num ∗ id	id − num ∗ ↑id
–	id − num ∗ id	id − num ∗ id↑

EXEMPLE

Une autre analyse syntaxique possible pour id – num * id

Prod'n	Sentential form	Input
1	⟨expr⟩	↑ id - num * id
2	⟨expr⟩ + ⟨term⟩	↑ id - num * id
2	⟨expr⟩ + ⟨term⟩ + ⟨term⟩	↑ id - num * id
2	⟨expr⟩ + ⟨term⟩ + ⋯	↑ id - num * id
2	⟨expr⟩ + ⟨term⟩ + ⋯	↑ id - num * id
2	⋯	↑ id - num * id

Reference Grammar

⟨expr⟩	::=	⟨expr⟩ + ⟨term⟩
	|	⟨expr⟩ − ⟨term⟩
	|	⟨term⟩
⟨term⟩	::=	⟨term⟩ ∗ ⟨factor⟩
	|	⟨term⟩ /⟨factor⟩
	|	⟨factor⟩
⟨factor⟩	::=	num
	|	id

Si l'analyseur syntaxique fait les mauvais choix, l'expansion ne se termine pas

- Ceci n'est pas une bonne propriété pour l'analyseur syntaxique
- Les analyseurs syntaxiques devraient se terminer, finalement…

RÉCURSIVITÉ À GAUCHE

Une grammaire est récursive à gauche si:

"Elle possède un non-terminal A de façon qu'il existe une dérivation $A \overset{+}{\Rightarrow} A\alpha$ pour une chaîne α"

Les parseurs descendants avec dérivation à gauche ne peuvent pas traiter la récursivité à gauche dans une grammaire

ÉLIMINER LA RÉCURSIVITÉ À GAUCHE

uOttawa
L'Université canadienne
Canada's university

Considérez le fragments de grammaire:

$$\begin{aligned} \langle \text{foo} \rangle &::= \langle \text{foo} \rangle \alpha \\ &\mid \beta \end{aligned}$$

Où α et β ne commence pas par ⟨foo⟩

On peut réécrire ceci comme:

$$\begin{aligned} \langle \text{foo} \rangle &::= \beta \langle \text{bar} \rangle \\ \langle \text{bar} \rangle &::= \alpha \langle \text{bar} \rangle \\ &\mid \varepsilon \end{aligned}$$

Où ⟨bar⟩ est un nouveau non-terminal

Ce Fragment ne contient pas de récursivité à gauche

EXEMPLE

Notre grammaire d'expressions contient deux cas de récursivité à gauche

$$
\begin{array}{lcl}
\langle \text{expr} \rangle & ::= & \langle \text{expr} \rangle + \langle \text{term} \rangle \\
& | & \langle \text{expr} \rangle - \langle \text{term} \rangle \\
& | & \langle \text{term} \rangle \\
\langle \text{term} \rangle & ::= & \langle \text{term} \rangle * \langle \text{factor} \rangle \\
& | & \langle \text{term} \rangle / \langle \text{factor} \rangle \\
& | & \langle \text{factor} \rangle
\end{array}
$$

Appliquer la transformation nous donne

$$
\begin{array}{lcl}
\langle \text{expr} \rangle & ::= & \langle \text{term} \rangle \langle \text{expr}' \rangle \\
\langle \text{expr}' \rangle & ::= & + \langle \text{term} \rangle \langle \text{expr}' \rangle \\
& | & - \langle \text{term} \rangle \langle \text{expr}' \rangle \\
& | & \varepsilon \\
\langle \text{term} \rangle & ::= & \langle \text{factor} \rangle \langle \text{term}' \rangle \\
\langle \text{term}' \rangle & ::= & * \langle \text{factor} \rangle \langle \text{term}' \rangle \\
& | & / \langle \text{factor} \rangle \langle \text{term}' \rangle \\
& | & \varepsilon
\end{array}
$$

Avec cette grammaire, un parseur descendant va

- Se terminer (assurément)
- Reculer pour quelques données

ALGORITHMES PRÉDICTIFS

uOttawa
L'Université canadienne
Canada's university

Nous avons vu que les parseurs descendants peuvent reculer lorsqu'ils choisissent la mauvaise production

Conséquemment, nous devons utiliser les algorithmes prédictifs afin d'éviter de reculer

Les parseur prédictifs sont utiles

- LL(1): scanner de gauche à droite, dérivation gauche, 1-token de prélecture (look ahead)
- LR(1): scanner de gauche à droite, dérivation droite, 1-token de prélecture (look ahead)

MERCI!

QUESTIONS?