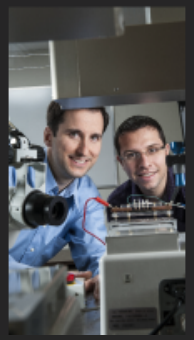





POLYTECHNIQUE MONTRÉAL

LE GÉNIE EN PREMIÈRE CLASSE

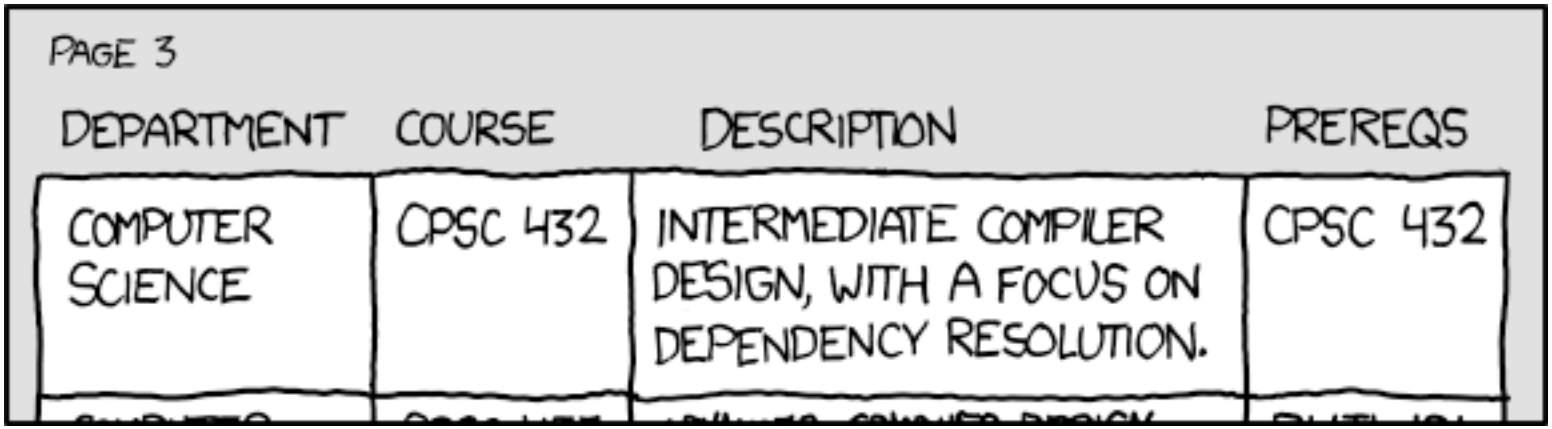
LOG3210
Éléments de langage et compilateurs

Analyse syntaxique – Parseurs descendants

Semaine 2

PLAN

1. Chapitre 2 (Suite et Fin)
   - Grammaires ambiguës
   - Associativité et précédence des opérateurs
2. Chapitre 4 – Parseurs descendants récursifs



https://xkcd.com/754/ : Dependencies

CHAPITRE 4 – INTRODUCTION

```
class Etudiant {
    string nom;
    int note;
}// ; // point-virgule manquant

int main(void) {

    return 0;
}
```

CHAPITRE 4 – INTRODUCTION

Visual Studio 2013

```
5  class Etudiant {
6      string nom;
7      int note;
8  }// ; // point-virgule manquant
9
10
11 int main(void) {
12     return 0;
13 }
14
```

(11) error C2628: 'Etudiant' suivi de 'int' n'est pas conforme **(n'auriez-vous pas oublié un ';' ?)**

(11) error C3874: le type de retour de 'main' doit être 'int' au lieu de 'Etudiant'


```
1>------ Début de la génération : Projet : sandbox, Configuration : Debug Win32 ------
1>  main.cpp
1>z:\poly\log3210\h16\sandbox\sandbox\main.cpp(11): error C2628: 'Etudiant' suivi de 'int' n'est pas conforme (n'auriez-vous pas oublié un ';' ?)
1>z:\poly\log3210\h16\sandbox\sandbox\main.cpp(11): error C3874: le type de retour de 'main' doit être 'int' au lieu de 'Etudiant'
```

CHAPITRE 4 – INTRODUCTION

Apple LLVM version 7.0.2
(clang-700.1.81)

```
5  class Etudiant {
6      string nom;
7      int note;
8  }// ; // point-virgule manquant
9
10
11 int main(void) {
12     return 0;
13 }
14
```

```
main.cpp:8:2: error: expected ';' after class
}// ; // point-virgule manquant
 ^
 ;
1 error generated.
```

CHAPITRE 4 – INTRODUCTION

gcc version 5.1.1 20150618
(Red Hat 5.1.1-4) (GCC)

```
5  class Etudiant {
6      string nom;
7      int note;
8  }// ; // point-virgule manquant
9
10
11 int main(void) {
12     return 0;
13 }
14
```

```
main.cpp:8:1: erreur: expected ';' after
  class definition
 }// ; // point-virgule manquant
 ^
```

ÉCOLE
POLYTECHNIQUE
MONTRÉAL

LOG3210
Cours 2

Analyse syntaxique – Parseurs descendants

Analyse syntaxique

Dérivations

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Il s'agit d'une séquence de remplacements produits via les règles de production
  - E → -E → -(E) → -(**id**)
- ***Leftmost* derivation** (dérivation à gauche)
  - On choisit toujours le non terminal le plus à gauche pour le remplacer
  - $E \Rightarrow_{lm} E + E \Rightarrow$ **id** $+ E \Rightarrow$ **id + id**
  - $E \Rightarrow^{*}_{lm}$ **id + id**
- ***Rightmost* derivation** (dérivation à droite)
  - On choisit toujours le non terminal le plus à droite pour le remplacer
  - $E \Rightarrow_{rm} E + E \Rightarrow E +$ **id** $\Rightarrow$ **id + id**

Exercice 1

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Donnez une dérivation à gauche et une dérivation à droite pour les grammaires et les chaînes spécifiées:

1. $S \rightarrow S\ S\ \mathbf{+} \mid S\ S\ \mathbf{*} \mid a$
   Chaîne ***aa+a****

Dérivation à gauche :

$$S \Rightarrow_{lm} S\ S\ * \Rightarrow S\ S\ \mathbf{+}\ S\ * \Rightarrow \mathbf{a}\ S\ \mathbf{+}\ S\ * \Rightarrow \mathbf{aa+}\ S\ * \Rightarrow \mathbf{aa+a*}$$

Dérivation à droite :

$$S \Rightarrow_{rm} S\ S\ * \Rightarrow S\ \mathbf{a*} \Rightarrow S\ S\ \mathbf{+a*} \Rightarrow \mathbf{a}\ S\ \mathbf{+a*} \Rightarrow \mathbf{aa+a*}$$

Exercice supplémentaire

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Donnez une dérivation à gauche et une dérivation à droite pour les grammaires et les chaînes spécifiées:

  I. $S \rightarrow S\ (\ S\ )\ S \mid \varepsilon$
  Chaîne **(()())**

- Donnez l'arbre de parsage.

- Décrivez le langage généré par cette grammaire.

- Exercices supplémentaires à la Section 4.2.8 du Dragon

Analyse syntaxique

ÉCOLE POLYTECHNIQUE MONTRÉAL

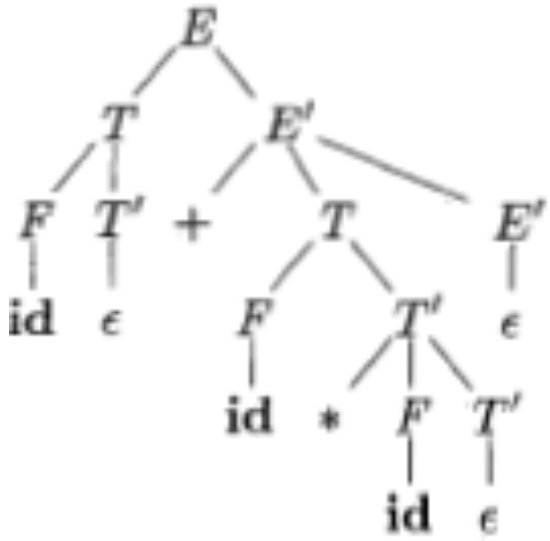
- CFG générales (applications en IA), peuvent traiter l'ambigüité
  - Earley, Cocke-Younger-Kasami
  - Espace = $O(n^2)$
  - Temps = $O(n^3)$
- LL(k) → O(n)
  - Contexte gauche (Lecture des jetons de gauche à droite)
  - Dérivation gauche
- LR(k) → O(n)
  - Contexte gauche (Lecture des jetons de gauche à droite)
  - Dérivation droite

Analyse syntaxique

ÉCOLE POLYTECHNIQUE MONTRÉAL

- On a une chaîne en entrée et une grammaire libre de contexte.
- Comment générer l'arbre de parsage?
- Exemple:
  - **id + id * id**

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \mathbf{id}
\end{array}
$$



Parseur descendant *(top-down)*

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Construisent un arbre de parsage à partir de la chaîne en entrée, en partant de la racine de l'arbre.
- Les nœuds sont créés en **préordre** (*aka* préfixe).
- Le symbole actuellement analysé est référé en tant que *lookahead*.

Parseur descendant *(top-down)*

ÉCOLE POLYTECHNIQUE MONTRÉAL

$$
\begin{array}{lcl}
\textit{stmt} & \rightarrow & \textbf{expr ;} \\
 & | & \textbf{if (expr)} \ \textit{stmt} \\
 & | & \textbf{for (} \ \textit{optexpr} \ \textbf{;} \ \textit{optexpr} \ \textbf{;} \ \textit{optexpr} \ \textbf{)} \ \textit{stmt} \\
 & | & \textbf{other} \\
 & & \\
\textit{optexpr} & \rightarrow & \epsilon \\
 & | & \textbf{expr}
\end{array}
$$

- Donnez l'arbre de parsage pour l'expression suivante:

for (; expr ; expr) other

Parseur descendant *(top-down)*

ÉCOLE POLYTECHNIQUE MONTRÉAL

$$
\begin{array}{lcl}
\textit{stmt} & \rightarrow & \textbf{expr ;} \\
 & | & \textbf{if (expr)} \ \textit{stmt} \\
 & | & \textbf{for (} \ \textit{optexpr} \ \textbf{;} \ \textit{optexpr} \ \textbf{;} \ \textit{optexpr} \ \textbf{)} \ \textit{stmt} \\
 & | & \textbf{other} \\
 & & \\
\textit{optexpr} & \rightarrow & \epsilon \\
 & | & \textbf{expr}
\end{array}
$$

Figure 2.16: A grammar for some statements in C and Java

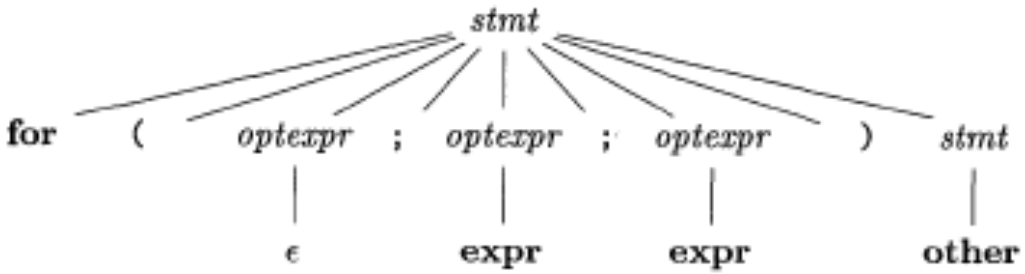


Figure 2.17: A parse tree according to the grammar in Fig. 2.16

for (; expr ; expr) other

Parseur descendant *(top-down)*

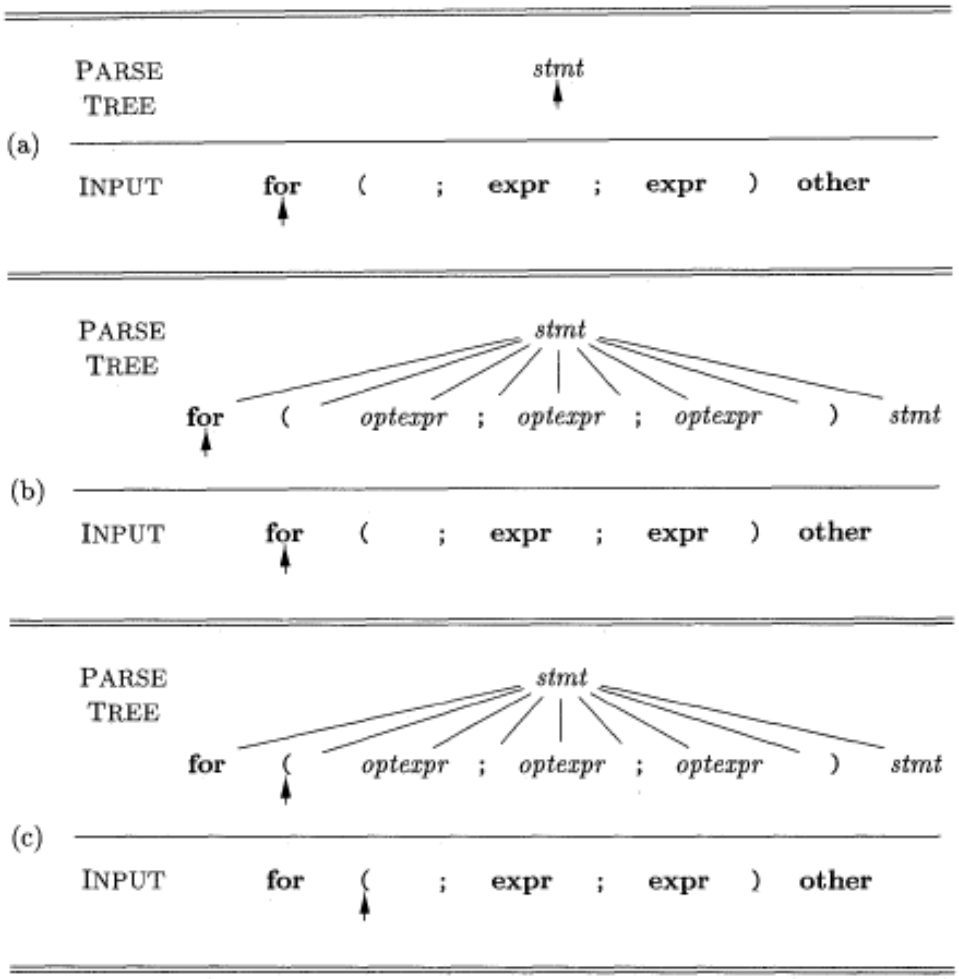


Figure 2.18: Top-down parsing while scanning the input from left to right

Parseur descendant *(top-down)* Retour en arrière

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Un parseur descendant typique peut nécessiter des retours en arrière (*backtracking*) si la grammaire n'est pas factorisée.

- Exemple:
  - $S \rightarrow c\,A\,d$ $\qquad$ $A \rightarrow a\,b \mid a$
- Si on veut reconnaître la chaîne **c a d** :

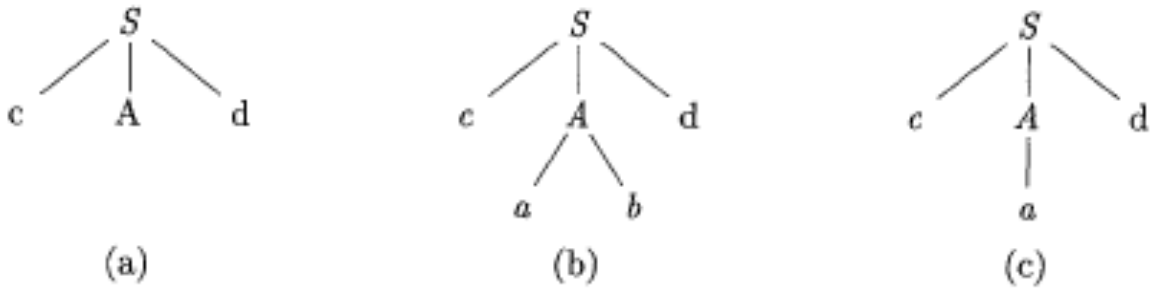


Figure 4.14: Steps in a top-down parse

Factorisation à gauche

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Pour les parseurs *top-down*, la production à utiliser n'est pas toujours évidente.

- Exemple
  - *stmt* → **if** *expr* **then** *stmt* **else** *stmt*
    | **if** *expr* **then** *stmt*

- La factorisation à gauche permet d'éliminer plusieurs cas où du *backtracking* pourrait être nécessaire.

Factorisation à gauche

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Solution: trouver le préfixe commun aux deux règles et créer un nouveau non terminal pour les parties non communes.
  - *stmt* → **if** *expr* **then** *stmt elsestmt*
    elsestmt → **else** *stmt* | ε

- Globalement:
  - $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$ α est le préfixe commun
    γ représente toutes les règles qui ne commencent pas par α

- Devient:
  - $A \rightarrow \alpha A' \mid \gamma$
  - $A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

Récursivité à gauche

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Une grammaire est récursive à gauche si elle contient un non terminal *A* tel qu'il existe une règle de production $A \rightarrow A\alpha$ pour une chaîne α quelconque.

- La récursivité à gauche mène à une boucle infinie pour les analyseurs syntaxiques récursifs en profondeur (*recursive-descent parsers*).

Élimination de la récursivité directe

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Soit les productions du type $A \rightarrow A\alpha \mid \beta$.

- Une telle récursivité à gauche peut toujours être éliminée en la remplaçant par les productions suivantes:
  - $A \rightarrow \beta A'$
    $A' \rightarrow \alpha A' \mid \varepsilon$
- Cette règle suffit pour la plupart des grammaires.

- Exemple:
  - $E \rightarrow E + T \mid T$
    $T \rightarrow T * F \mid F$
    $F \rightarrow (E) \mid \mathbf{id}$

Grammaire des expressions

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Exemple:
  - E → E + T | T
  - T → T * F | F
  - F → (E) | id

- En éliminant la récursivité à gauche:

E -> T E'

E' -> + T E' | ε

T -> F T'

T' -> * F T' | ε

F -> (E) | id

Élimination de la récursivité directe (suite)

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Dans le cas général, étant donné une règle de production du type : $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \beta_1 \mid \beta_2 \mid \beta_3$

- Il suffit de remplacer la règle par :

  $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A'$

  $A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \varepsilon$

Élimination de la récursivité indirecte

ÉCOLE POLYTECHNIQUE MONTRÉAL

- La récursivité à gauche peut impliquer plus d'une étape.

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Ac \mid Sd \mid \varepsilon$$

- Le non terminal S est récursif à gauche, car $S \rightarrow Aa \rightarrow Sda$

- L'algorithme 4.19 du livre permet d'éliminer ces récursivités.

Élimination de la récursivité Algorithme

ÉCOLE POLYTECHNIQUE MONTRÉAL

Algorithm 4.19: Eliminating left recursion.

INPUT: Grammar G with no cycles or ϵ-productions.

OUTPUT: An equivalent grammar with no left recursion.

METHOD: Apply the algorithm in Fig. 4.11 to G. Note that the resulting non-left-recursive grammar may have ϵ-productions. □

```
1)  arrange the nonterminals in some order A1, A2, ..., An.
2)  for ( each i from 1 to n ) {
3)      for ( each j from 1 to i - 1 ) {
4)          replace each production of the form Ai → Aj γ by the
                productions Ai → δ1 γ | δ2 γ | ··· | δk γ, where
                Aj → δ1 | δ2 | ··· | δk are all current Aj-productions
5)      }
6)      eliminate the immediate left recursion among the Ai-productions
7)  }
```

Exercice 2

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Éliminez la récursion à gauche des grammaires suivantes

1. S → AB
   A → AA | Aa | aa
   B → BB | Bb | bb

2. D → EF
   E → Eg | Fe
   F → DF | f

Parseur descendant *(top-down)* Boucle infinie

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Expliquez pourquoi une grammaire récursive à gauche peut causer un analyseur syntaxique descendant (LL) à entrer dans une boucle infinie.
  - Ex.: E→E + T | T

- Le parseur peut dériver toujours le même non-terminal, sans consommer aucun jeton en entrée, et tomber ainsi dans une boucle infinie. (Section 4.4.1 du Dragon)

FIRST (Livre du Dragon 4.2.2)

ÉCOLE POLYTECHNIQUE MONTRÉAL

- **FIRST(α)** : l'ensemble des terminaux qui peuvent être les premiers symboles de α.

Déterminer FIRST pour tous les symboles

1. Si *X* est un terminal, alors FIRST(*X*) = { *X* }
2. Si *X* est un non terminal et que $X \rightarrow Y_1 Y_2 \ldots Y_k \ldots$ est une production, ajouter *a* dans FIRST(*X*) si pour un *i* quelconque, *a* est dans FIRST(Y_i) et ε est dans FIRST(Y_1), …, FIRST(Y_{i-1}).
   Si ε est dans FIRST(Y_j) pour j = 1, 2, …, k, alors ajouter ε à FIRST(*X*).
   Règle générale: FIRST(Y_1) est sûrement dans FIRST(X).
   Si Y_1 ne dérive pas ε, rien d'autre n'est ajouté dans FIRST(X)
   Si Y_1 dérive ε, on ajoute FIRST(Y_2). Continuer la même règle avec Y_2, etc.
3. Si $X \rightarrow \varepsilon$ est une production, ajouter ε à FIRST(X)

FIRST (Livre du Dragon 4.2.2)

ÉCOLE POLYTECHNIQUE MONTRÉAL

- **FIRST(α)** : l'ensemble des terminaux qui peuvent être les premiers symboles de α.

Déterminer FIRST pour tous les symboles

1. Si *X* est un terminal, alors FIRST(*X*) = { *X* }
2. Si *X* est un non terminal et que $X \rightarrow Y_1 Y_2 \ldots Y_k \ldots$ est une production, ajouter *a* dans FIRST(*X*) si pour un *i* quelconque, *a* est dans FIRST(Y_i) et ε est dans FIRST(Y_1), …, FIRST(Y_{i-1}).
   Si ε est dans FIRST(Y_j) pour j = 1, 2, …, k, alors ajouter ε à FIRST(*X*).
3. Si $X \rightarrow \varepsilon$ est une production, ajouter ε à FIRST(X)

- **FIRST(X_1,X_2,…X_n)** : Ajouter FIRST(X_1) sauf ε.
  Si $\varepsilon \in$ FIRST(X_1), ajouter FIRST(X_2) sauf ε, etc.

FOLLOW (Livre du Dragon 4.2.2)

ÉCOLE POLYTECHNIQUE MONTRÉAL

- **FOLLOW(*A*)** : l'ensemble des terminaux *a* pouvant apparaître immédiatement à la droite de A. Si A peut être le symbole le plus à droite, on ajoute alors **$** à l'ensemble FOLLOW.

Déterminer FOLLOW pour tous les non terminaux

1. Placer $ dans FOLLOW(*S*), où *S* est le symbole de départ et $ est le marqueur de fin.
2. S'il existe une production A → αBβ, tout ce qui est dans FIRST(β), à l'exception de ε, est dans FOLLOW(*B*).
3. S'il existe une production A → αB, ou encore une production A → αBβ où FIRST(β) contient ε, ajouter tout ce qui est dans FOLLOW(A) dans FOLLOW(B)

Exercice 3

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Calculer les FIRST et FOLLOW de la grammaire suivante:

  - E → T G
  - G → **+** T G | ε
  - T → F H
  - H → ***** F H | ε
  - F → (E) | **id**

Exercice 3 (solution)

ÉCOLE POLYTECHNIQUE MONTRÉAL

- FIRST(*F*) = FIRST(*T*) = FIRST(*E*) = { **(**, **id** }
- FIRST(*G*) = { **+**, ε }
- FIRST(*H*) = { ***, ε }

- FOLLOW(*E*) = FOLLOW(*G*) = { **)**, **$** }
- FOLLOW(*T*) = FOLLOW(*H*) = { **+**, **)**, **$** }
- FOLLOW(*F*) = { **+**, ***, **)**, **$** }

Grammaires LL(1)

ÉCOLE POLYTECHNIQUE MONTRÉAL

Les grammaires LL(1) sont un sous-ensemble des CFG.

- **Premier L**: lecture de gauche à droite (**L**eft to right)
- **Deuxième L:** dérivation à gauche (**L**efmost derivation)
- **Chiffre 1**: un seul symbole de *lookahead* est utilisé pour sélectionner la prochaine production.

- La plupart des constructions que l'on retrouve en programmation peuvent être représentées par une grammaire LL(1).
  - Attention, les grammaires LL(1) ne doivent être ni récursives à gauche, ni ambiguës. Elles ont aussi été factorisées à gauche.

Conditions pour qu'une grammaire soit LL(1)

- Pour deux productions $A \rightarrow \alpha \mid \beta$ distinctes:
  1. FIRST(α) et FIRST(β) sont deux ensembles disjoints.
     - Factorisée à gauche.
  2. Au plus un de α et β peut dériver la chaîne vide ε (**déjà spécifié en 1**).
  3. Si β peut éventuellement dériver ε (ε est dans FIRST(β)), alors FIRST(α) et FOLLOW(A) sont disjoints.
     De même, si α peut éventuellement dériver ε (ε est dans FIRST(α)), alors FIRST(β) et FOLLOW(A) sont disjoints.

Analyseur syntaxique descendant Implémentation

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Un ensemble de procédures récursives est utilisé pour reconnaître l'entrée.
- Une procédure est associée à chaque non terminal de la grammaire.
- Opération **match(jeton)** : vérifie que *jeton* est égal au symbole *lookahead* courant, puis incrémente le *lookahead*.

Analyseur syntaxique prédictif (descendant)

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Un analyseur syntaxique prédictif peut être construit pour une grammaire LL(1).
- Un analyseur syntaxique prédicitf est un analyseur descendant récursif qui ne requiert pas de retour en arrière.
- Seul le symbole courant doit être pris en compte pour choisir la règle à sélectionner.

Construction d'un analyseur syntaxique prédictif (descendant)

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Produire une méthode pour chaque non terminal *A*.

- Pour chaque production $A \rightarrow \alpha$ de la grammaire:
  1. Si *lookahead* $\in$ FIRST(α), appeler les méthodes de α.
  2. Si $\varepsilon \in$ FIRST(α) et que *lookahead* $\in$ FOLLOW(*A*), appeler les méthodes de α.
     - Si $\alpha = \varepsilon$, alors ne rien faire.

- Étant donné la méthode correspondant au non terminal *A*, tous les jetons qui n'ont pas été traités au point précédent génèrent une erreur.

Analyseur syntaxique prédictif (descendant)

ÉCOLE POLYTECHNIQUE MONTRÉAL

$$
\begin{array}{lcl}
\textit{stmt} & \rightarrow & \textbf{expr ;} \\
 & | & \textbf{if (expr)} \textit{ stmt} \\
 & | & \textbf{for (} \textit{ optexpr ; optexpr ; optexpr } \textbf{)} \textit{ stmt} \\
 & | & \textbf{other} \\
 & & \\
\textit{optexpr} & \rightarrow & \epsilon \\
 & | & \textbf{expr}
\end{array}
$$

- Écrivons un analyseur syntaxique descendant pour cette grammaire.

Analyseur syntaxique prédictif (Solution naïve)

ÉCOLE POLYTECHNIQUE MONTRÉAL

```
void stmt() {
        switch ( lookahead ) {
        case expr:
                match(expr); match(';'); break;
        case if:
                match(if); match('('); match(expr); match(')'); stmt();
                break;
        case for:
                match(for); match('(');
                optexpr(); match(';'); optexpr(); match(';'); optexpr();
                match(')'); stmt(); break;
        case other;
                match(other); break;
        default:
                report("syntax error");
        }
}

void optexpr() {
        if ( lookahead == expr ) match(expr);
}

void match(terminal t) {
        if ( lookahead == t ) lookahead = nextTerminal;
        else report("syntax error");
}
```

← Support pour le ε

Analyseur syntaxique prédictif (Solution avec détection d'erreurs)

ÉCOLE POLYTECHNIQUE MONTRÉAL

```
void stmt() {
        switch ( lookahead ) {
        case expr:
                match(expr); match(';'); break;
        case if:
                match(if); match('('); match(expr); match(')'); stmt();
                break;
        case for:
                match(for); match('(');
                optexpr(); match(';'); optexpr(); match(';'); optexpr();
                match(')'); stmt(); break;
        case other;
                match(other); break;
        default:
                report("syntax error");
        }
}

void optexpr() {
        if ( lookahead == expr ) match(expr);
}

void match(terminal t) {
        if ( lookahead == t ) lookahead = nextTerminal;
        else report("syntax error");
}
```

```
if ( lookahead == expr)
        match (expr);
else if ( lookahed == ';' )
        ; // do nothing
else if ( lookahed == ')' )
        ; // do nothing
else
        report("syntax error");
```

Exercice 4

ÉCOLE POLYTECHNIQUE MONTRÉAL

- En considérant que:
  - La fonction "boolean match(int jeton)" est déjà définie.
  - Les jetons de type int : **<id>**, **<true>**, **<false>**, **<parenthese gauche>**, **<parenthese droite>**, **<or>** et **<and>** sont déjà définis.
  - La variable globale *lookahead* contient le jeton courant.

- Écrivez le code de l'analyseur syntaxique descendant récursif correspondant à la grammaire suivante:
  - $E \rightarrow M\ F$
    $F \rightarrow \vee\ E \mid \varepsilon$
    $M \rightarrow W\ N$
    $N \rightarrow \wedge\ M \mid \varepsilon$
    $W \rightarrow$ **id** | **true** | **false** | **(** E **)**

Exercice supplémentaire

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Considérez la chaîne:
  (id $\wedge$ **true)** $\vee$ **(id** $\vee$ **false)**

1. Donnez une dérivation à gauche (*leftmost derivation*) pour la chaîne considérée.
2. Donnez une dérivation à droite (*rightmost derivation*) pour la chaîne considérée.
3. Donnez un arbre d'analyse syntaxique (*parse tree*) pour la chaîne considérée

PARSEURS DESCENDANTS DANS LE MONDE OPEN SOURCE

GCC 3.4.0 (Avril 2004)

« A **hand-written recursive-descent C++ parser** has replaced the YACC*-derived C++ parser from previous GCC releases. The new parser contains much improved infrastructure needed for better parsing of C++ source codes, handling of extensions, and clean separation (where possible) between proper semantics analysis and parsing. The new parser fixes many bugs that were found in the old parser. »

* Yacc est un générateur de parseurs LR (ascendant)

PARSEURS DESCENDANTS DANS LE MONDE OPEN SOURCE

« A single unified parser for C, Objective C, C++, and Objective C++

Clang is the "C Language Family Front-end", which means we intend to support the most popular members of the C family. We are convinced that the right parsing technology for this class of languages is a hand-built recursive-descent parser. Because it is plain C++ code, recursive descent makes it very easy for new developers to understand the code, it easily supports ad-hoc rules and other strange hacks required by C/C++, and makes it straight-forward to implement excellent diagnostics and error recovery. »

PARSEURS DESCENDANTS DANS LE MONDE OPEN SOURCE

« **Expressive Diagnostics**

In addition to being fast and functional, we aim to make Clang extremely user friendly. As far as a command-line compiler goes, this basically boils down to making the diagnostics (error and warning messages) generated by the compiler be as useful as possible. There are several ways that we do this, but the most important are pinpointing exactly what is wrong in the program, highlighting related information so that it is easy to understand at a glance, and making the wording as clear as possible. »

PARSEURS DESCENDANTS RÉCURSIFS

Lectures supplémentaires

Eli Bendersky,
« *Recursive descent, LL and predictive parsers* »,
http://eli.thegreenplace.net/2008/09/26/recursive-descent-ll-and-predictive-parsers

Josh Haberman, « *LL and LR in Context: Why Parsing Tools Are Hard* »,
http://blog.reverberate.org/2013/09/ll-and-lr-in-context-why-parsing-tools.html