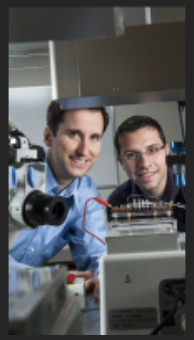





POLYTECHNIQUE MONTRÉAL

LE GÉNIE EN PREMIÈRE CLASSE

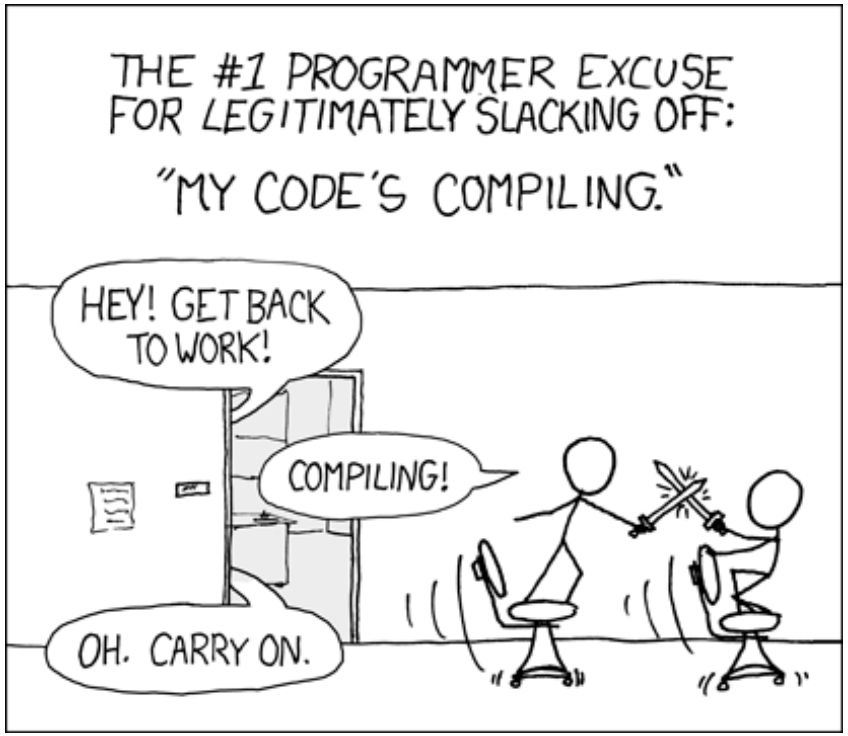
LOG3210
Éléments de langage et compilateurs

Introduction
Analyse lexicale
Grammaires

Semaine 1

PLAN

1. Présentation du cours
2. Chapitre 3 – Analyse lexicale
3. Chapitre 2 – Grammaires



https://xkcd.com/303/ : Compiling

QUI PROGRAMME EN C++ ?

- Génération de code intermédiaire (e.g. LLVM-IR)
- Génération de code machine (ou d'assembleur)
- Allocation de registres
- Optimisations (si possible)

QUI PROGRAMME EN JAVA ?

- Génération de code intermédiaire (Java *bytecode*)
- Gestion de la mémoire (*garbage collection*)

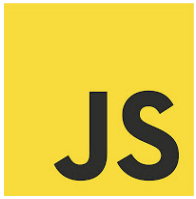
QUI PROGRAMME EN ML ?

Programmation fonctionelle

- ML, Haskell, Lisp, R

- Environnements run-time
  - *Nested procedures*
    - Chapitre 7 : *Access links*

QUI PROGRAMME EN PHP ?



Langages interprétés

- PHP, JavaScript, Python, Ruby

- Analyse lexicale et syntaxique
  - Exécution du code « *on-the-fly* »

- Analyse de flux (LOG6302 Réingénierie)

2016/01/09 https://www.python.org/community/logos/, http://www.2ality.com/2011/10/logo-js.html, http://php.net/download-logos.php

ANALYSE STATIQUE DE PROGRAMMES

Heartbleed

```
hbtype = *p++;
n2s(p, payload);
pl = p;
```

2016/01/09 http://heartbleed.com,
http://blog.cryptographyengineering.com/2014/04/attack-of-week-openssl-heartbleed.html

ANALYSE STATIQUE DE PROGRAMMES

Heartbleed : *Boundary check*!

```
/* Read type and payload length first */
if (1 + 2 + 16 > s->s3->rrec.length)
  return 0; /* silently discard */
hbtype = *p++;
n2s(p, payload);
if (1 + 2 + payload + 16 > s->s3->rrec.length)
  return 0; /* silently discard per RFC 6520
  sec. 4 */
pl = p;
```

2016/01/09 http://heartbleed.com,
http://blog.cryptographyengineering.com/2014/04/attack-of-week-openssl-heartbleed.html

POURQUOI UN COURS DE COMPILATEURS ?

- Pour comprendre un outil indispensable au programmeur :
  - Pourquoi est-ce que le ';' manquant n'est détecté que plusieurs lignes plus loin ?
  - Quelles sont les optimisations d'un compilateur ?
  - Pourquoi utiliser un compilateur plutôt qu'un autre ?
  - Quel est l'impact de la *garbage collection* sur un programme ?

POURQUOI UN COURS DE COMPILATEURS ?

- Pour écrire un nouveau langage :
  - *Domain Specific Language* (DSL) ;
  - Systèmes embarqués.

    (*Cross compilation*, port de Clang sur ARM)

- Pour faire de l'analyse de programmes :
  - Intégration dans un IDE ;
  - Analyses de sécurité.

POURQUOI UN COURS DE COMPILATEURS ?

Parce que c'est recherché dans l'industrie ! ($$)

https://developer.apple.com/swift/,
http://fbinfer.com (Facebook Infer for mobile apps)

ÉCOLE
POLYTECHNIQUE
MONTRÉAL

LOG3210
Cours 1

Introduction et grammaires

Compilateurs vs Interpréteurs

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Quelle est la différence?

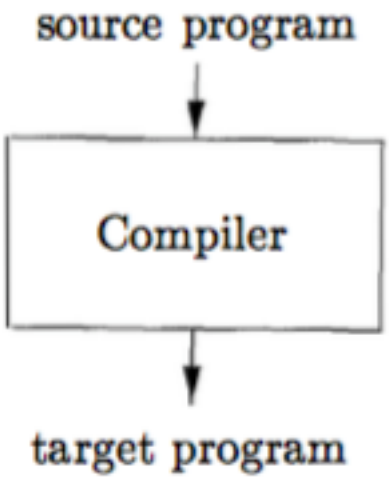


Figure 1.1: A compiler

Figure 1.2: Running the target program

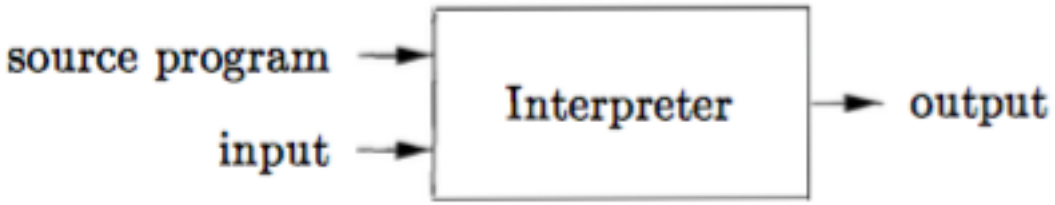


Figure 1.3: An interpreter

Compilateurs vs Interpréteurs

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Avantages – Inconvénients
  - Rapidité ? (Compilateur)
  - Diagnostiques / traitements des erreurs ?
  - Portabilité ? (Interpréteur)
- Exemples
  - C++ vs Python
  - Langages web
- Approches mixtes
  - Java (Compilation et interprétation)
  - Compilateurs *just-in-time* (JIT)

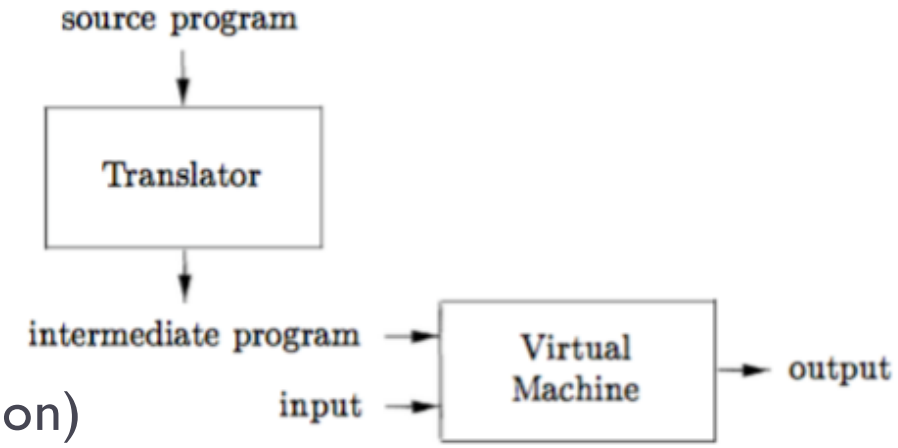
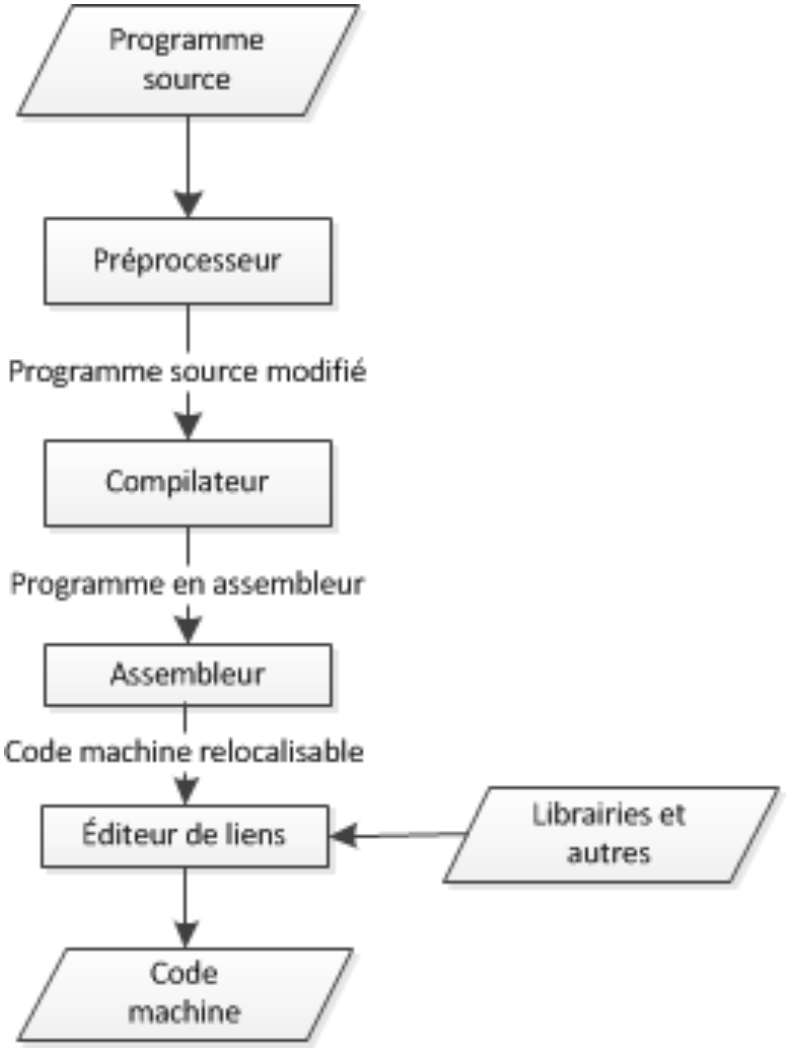
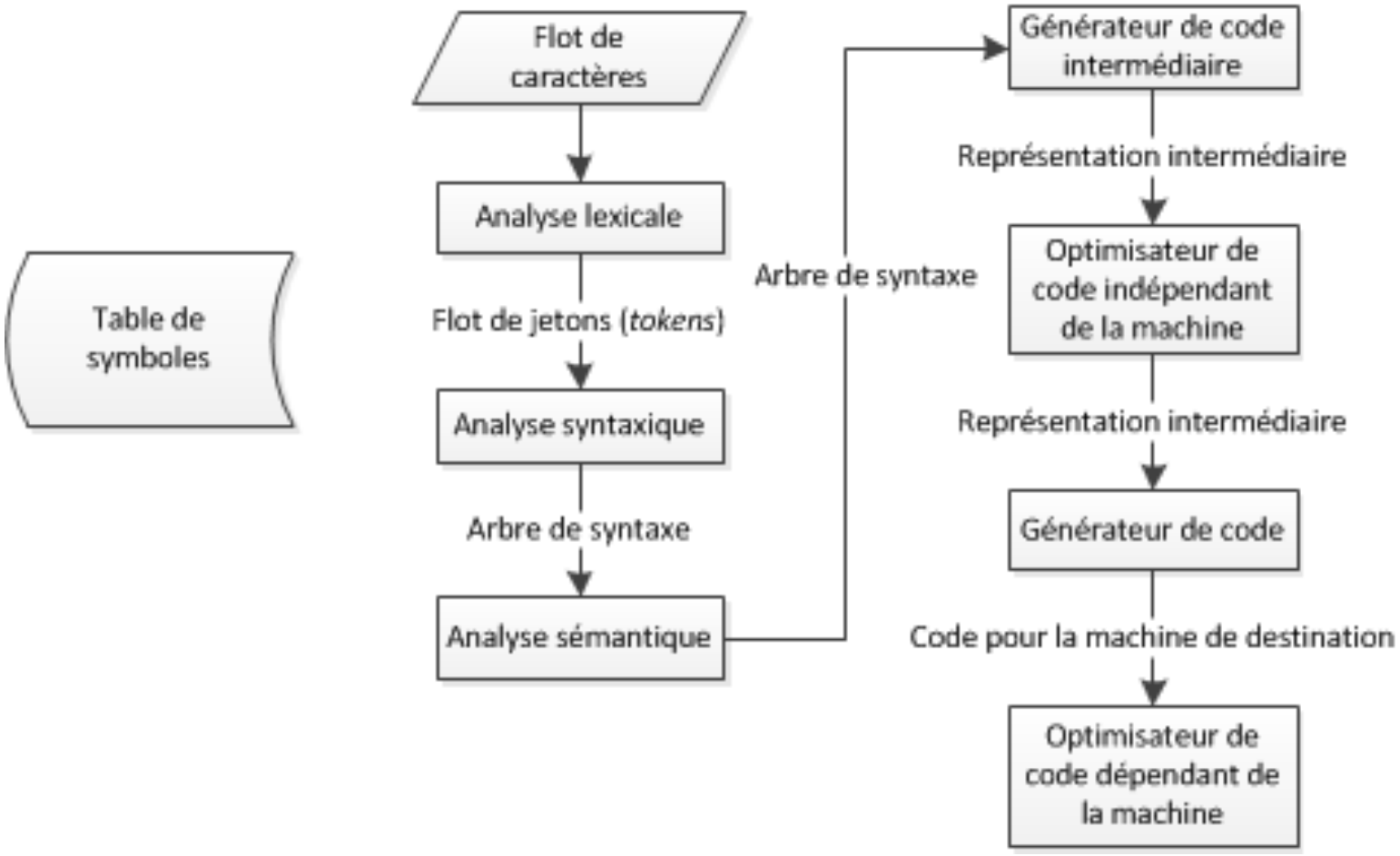


Figure 1.4: A hybrid compiler

Compilation d'un langage typique



Phases d'un compilateur



Analyse lexicale

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Permet de lire le programme source et produire des jetons (*tokens*) qui seront insérés dans la table de symboles.

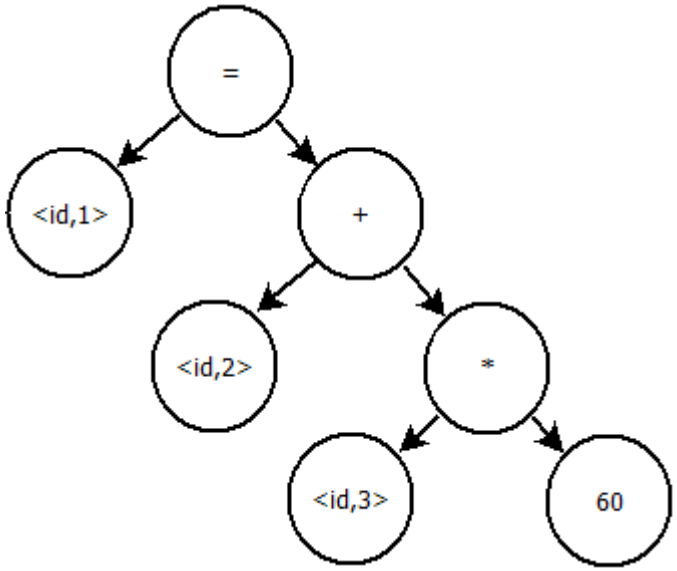
- Par exemple:
  - *Position = initial + rate * 60*

- Devient:
  - *<**id**, 1> <=> <**id**, 2> <+> <**id**, 3> <*> <60>*

Analyse syntaxique

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Communément appelé parsage (*parsing*).
- Crée un arbre de syntaxe qui permet de connaître l'ordre dans lequel les expressions sont exécutées.
- Exemple:



Analyse sémantique

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Vérifie que le programme en entrée est consistant avec le langage défini.
- Effectue les diverses vérifications sémantiques telles que:
  - Vérifications des types
  - Vérification de l'initialisation des variables
- Renvoie les erreurs et avertissements de compilation, si nécessaire.

Génération de code

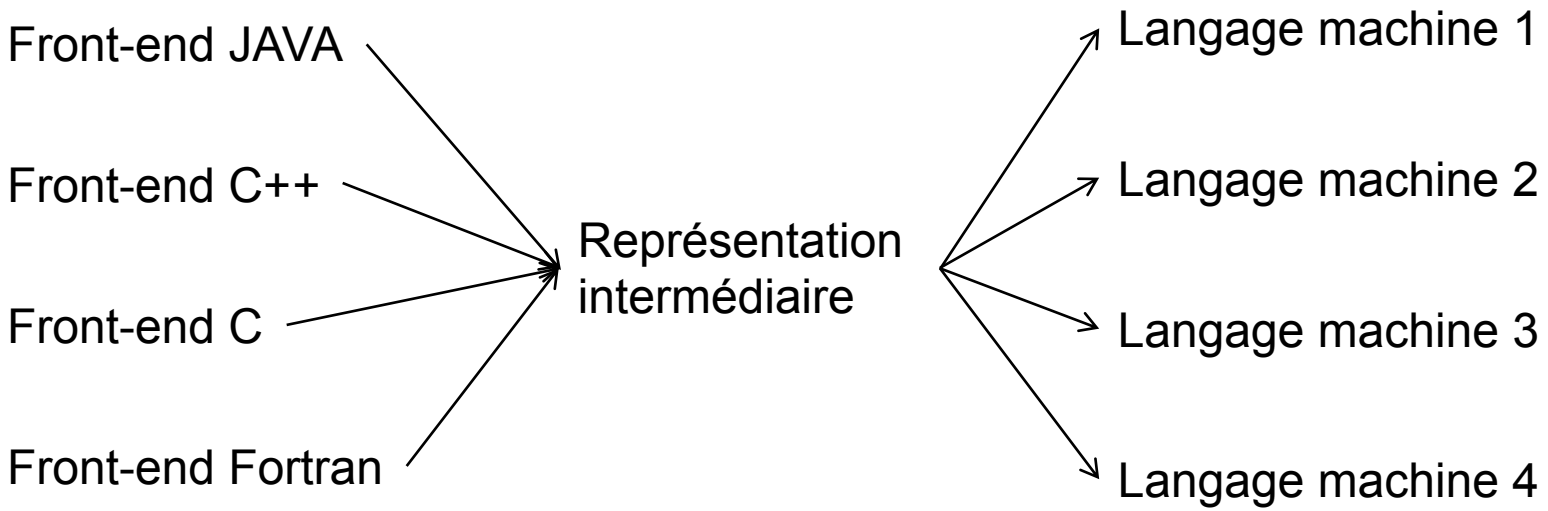
ÉCOLE POLYTECHNIQUE MONTRÉAL

- Utilise l'arbre de syntaxe pour générer
  - du code intermédiaire et,
  - éventuellement, le code machine.

Génération de code intermédiaire

ÉCOLE POLYTECHNIQUE MONTRÉAL

- But de la représentation intermédiaire: interface entre le front-end (spécifique au langage) et le back-end (spécifique à la machine).



Écrire un compilateur

ÉCOLE POLYTECHNIQUE MONTRÉAL

- L'écriture d'un compilateur est une tâche très complexe.

- L'optimisation du compilateur doit:
  - Être correcte (conserver l'intention du code original)
  - Augmenter la performance de plusieurs programmes
  - Maintenir un temps de compilation raisonnable
  - Garder le compilateur relativement simple pour permettre sa maintenance

- Les compilateurs avec optimisations contiennent parfois des erreurs. (e.g. Option –O4 sur gcc)

Applications des compilateurs

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Permettre le développement de langages de programmation de haut niveau.
- Optimiser le code pour diverses architectures
  - Parallélisme
  - Gestion des niveaux de mémoire
- Traduction de programmes (SQL, traduction binaires…)
- Outils de productivité

Analyse lexicale

Analyse lexicale

ÉCOLE POLYTECHNIQUE MONTRÉAL

- L'analyseur lexicale lit la chaîne en entrée et la convertit en unités lexicales: les jetons *(tokens*).

- Deux tâches principales:
  - *Scanning* : tâches qui ne nécessitent pas la création de jetons. Par exemple, supprimer les commentaires et les espaces inutiles.
  - Analyse lexicale: produire les jetons

Analyse lexicale - Jetons

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Pour la plupart des langages de programmation, les classes de jetons suivantes sont suffisantes:
  - Un jeton pour chaque mot clé (**if**, **else**, etc.)
  - Un jeton pour chaque opérateur
  - Un jeton pour les identificateurs
  - Un ou plusieurs jetons pour les constantes (nombres, chaînes de caractères…)
  - Un jeton pour chaque symbole de ponctuation (point, parenthèses, point-virgule…)

Analyse lexicale - Jetons

ÉCOLE POLYTECHNIQUE MONTRÉAL

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	`<` or `>` or `<=` or `>=` or `==` or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but ", surrounded by "'s	`"core dumped"`

Figure 3.2: Examples of tokens

Attributs des jetons

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Les jetons peuvent avoir un attribut permettant de les différencier.
  - Pointeur vers la table de symboles
  - Valeur d'une constante numérique

- Par exemple, E = M * C ** 2 devient:

<id, pointer to symbol-table entry for E>
<assign_op>
<id, pointer to symbol-table entry for M>
<mult_op>
<id, pointer to symbol-table entry for C>
<exp_op>
<number, integer value 2>

Spécification des jetons

ÉCOLE POLYTECHNIQUE MONTRÉAL

- On peut spécifier les jetons via des expressions régulières:

$$
\begin{array}{rcl}
\textit{letter\_} & \rightarrow & \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \mid \texttt{\_} \\
\textit{digit} & \rightarrow & \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
\textit{id} & \rightarrow & \textit{letter\_} \; (\; \textit{letter\_} \mid \textit{digit} \;)^*
\end{array}
$$

- **Exercice :** Donnez le langage spécifié par les expressions suivantes, en considérant les terminaux T = {*a*, *b*}:
  1. a | b
  2. (a | b) (a | b)
  3. a*
  4. (a | b)*
  5. a | a* b

Reconnaissance de jetons

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Le but de l'analyseur lexicale est de transformer l'entrée en une chaîne de jetons, tel quel spécifié dans à la figure 3.12 :

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
`<`	**relop**	LT
`<=`	**relop**	LE
`=`	**relop**	EQ
`<>`	**relop**	NE
`>`	**relop**	GT
`>=`	**relop**	GE

Figure 3.12: Tokens, their patterns, and attribute values

Reconnaissance de jetons

- La reconnaissance des jetons s'effectue via un automate NFA (non-déterministe) ou DFA (déterministe)

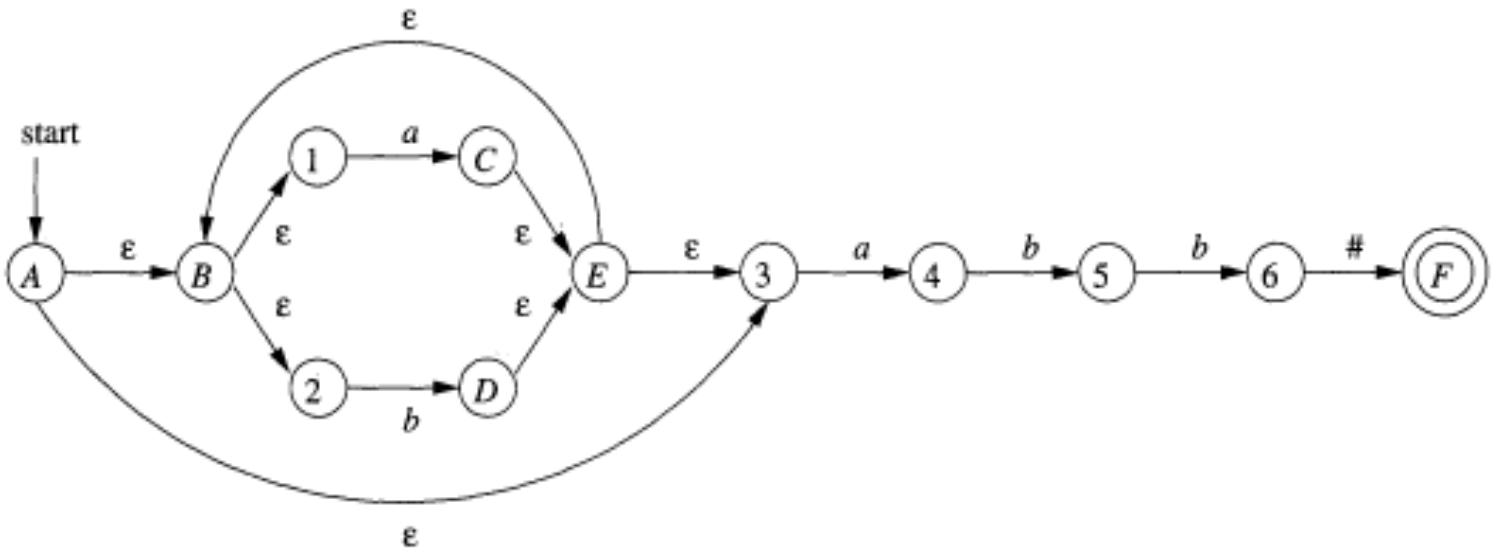


Figure 3.57: NFA constructed by Algorithm 3.23 for (**a**|**b**)***abb#**

Grammaires

Grammaires

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Une grammaire décrit la structure hiérarchique d'un langage.

- Exemple:
  - **if** (expression) statement **else** statement

- Règle de production:
  - *stmt* → **if** (*expr*) *stmt* **else** *stmt*

Grammaires libres de contexte (CFG)

ÉCOLE POLYTECHNIQUE MONTRÉAL

- CFG = (N, T, P, S) :
  - Symboles non terminaux (N)
  - Symboles terminaux (T)
  - Règles de productions P du type
    - $a \rightarrow b, \quad a \in N \quad et \quad b \in (N \cup T)$
    - Exemple:
      - E → E + E | E * E | (E) | -E | id
  - Un symbole de départ $S \in N$

Grammaires libres de contexte (CFG)

ÉCOLE POLYTECHNIQUE MONTRÉAL

- **Exemple :** Grammaire qui reconnait une chaîne de "*a*", suivie d'une addition (+), suivie d'une chaîne de "*b*".

- CFG = (N, T, P, S) :
  - Symboles non terminaux (N) : N = {S, A, B}
  - Symboles terminaux (T) : T = {a, b, +}
  - Règles de productions P :
    - S → A + B
    - A → aA | a
    - B → bB | b
  - Un symbole de départ S ∈ N

- **Question :** Combien de règles de productions ?

Exercice 1.1

ÉCOLE POLYTECHNIQUE MONTRÉAL

- En utilisant les symboles suivants, écrivez une grammaire permettant de reconnaître la multiplication et la division d'un nombre aléatoire de chiffres.

Par exemple, la chaîne 8*2*4/2*9/1/2 doit être reconnue.

  - Symboles non terminaux: *list*, *digit*
  - Symboles terminaux: *, /, les chiffres de 0 à 9

Représentation de CFG

ÉCOLE POLYTECHNIQUE MONTRÉAL

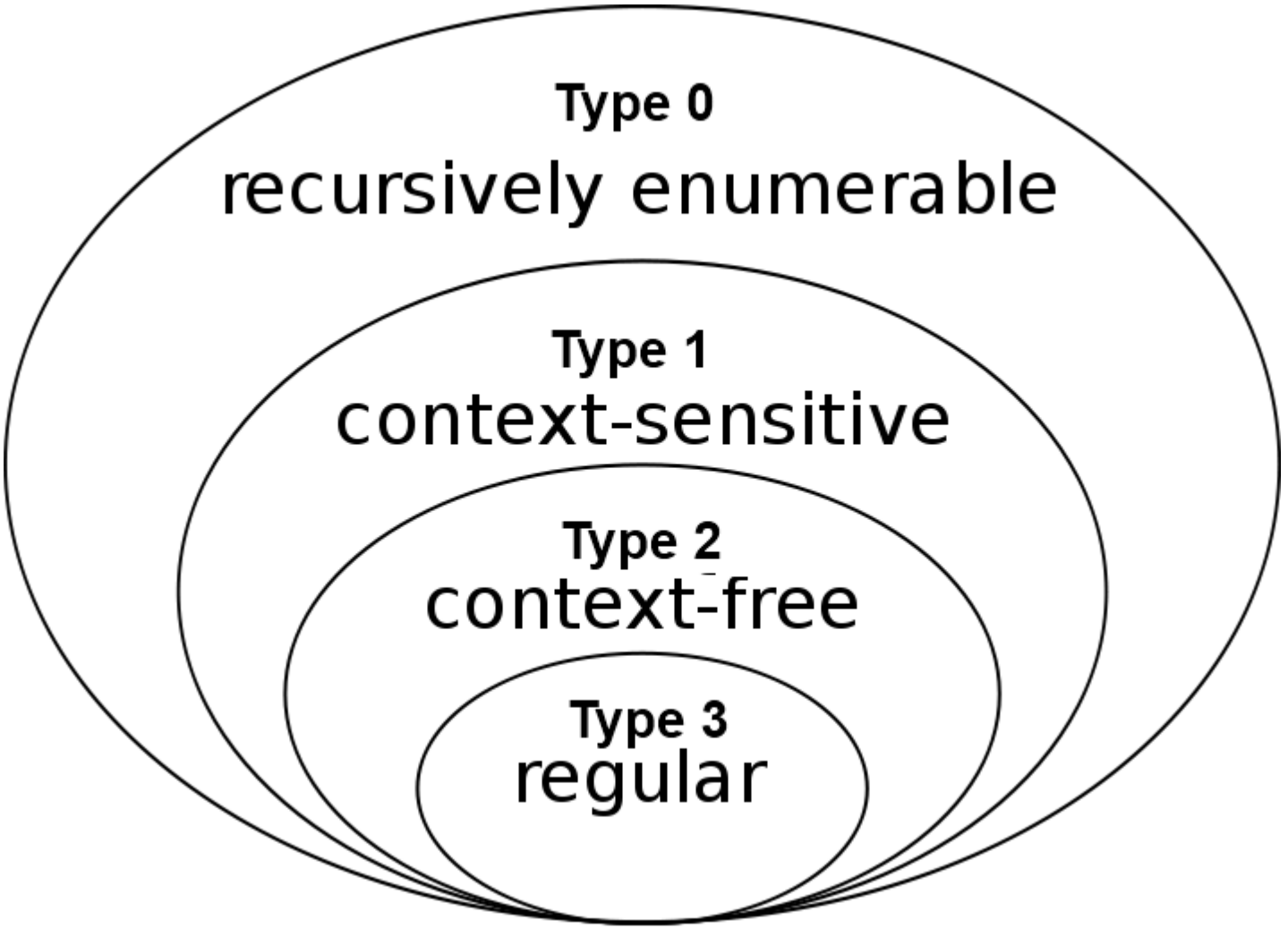
- Backus-Naur Form (BNF)
- Extended BNF (EBNF)
  - Groupage: (…) , 1 occurrence
  - Parties optionnelles: […], 0 ou 1 occurrence
  - {…}+, 1 ou plus occurrences
  - {…}n, 1 à n occurrences
  - {…}*, 0 à plus occurrences
- JavaCC permet d'utiliser la notation EBNF

Langages libres de contexte

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Un langage libre de contexte (CFL) est généré par une CFG.
  - Un langage est généré par une grammaire.
- Il existe des langages non libres de contexte.
  - Par exemple, C/C++ !

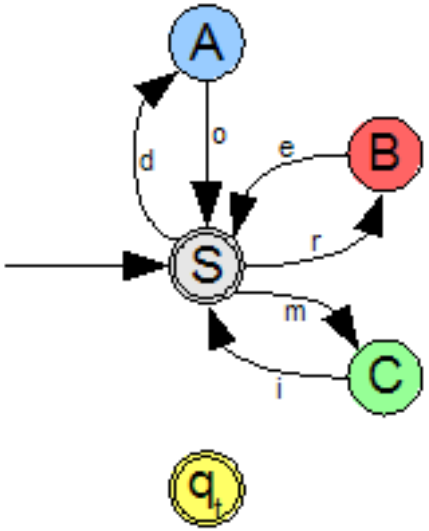
Types de grammaires



Types de grammaires

- **Grammaires régulières** (type 3)
  - Ne peuvent comprendre que des règles de production ayant la forme A → a ou A → aB
  - Il s'agit de tous les langages reconnaissables par un automate à fini (*Finite State Machine*).
  - Tous les langages réguliers peuvent être obtenus via une expression régulière.
  - Exemple:

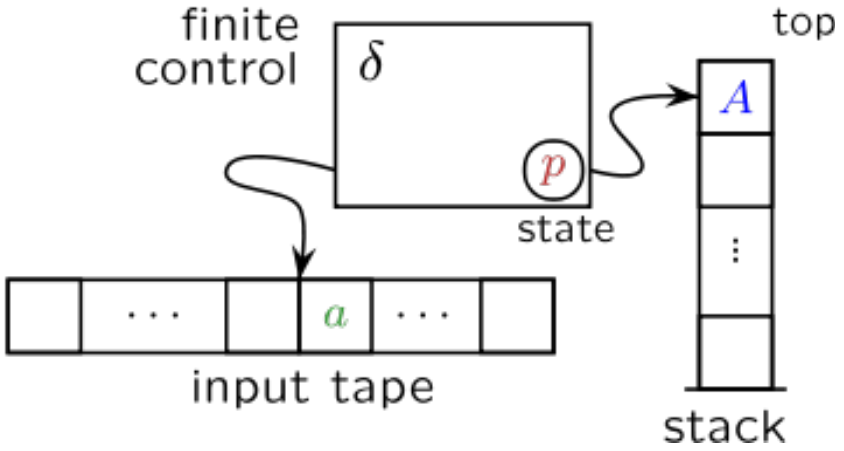
$$S \rightarrow dA|rB|mC|\epsilon$$
$$A \rightarrow oS$$
$$B \rightarrow eS$$
$$C \rightarrow iS$$



Types de grammaires

ÉCOLE POLYTECHNIQUE MONTRÉAL

- **Grammaires libres de contexte** (type 2)
  - Règles de la forme A → γ, où A est un non terminal et γ est un ensemble de terminaux et de non terminaux.
  - Il s'agit de tous les langages reconnaissables par un automate à pile (*pushdown automaton* ou parseur ascendant).
  - La plupart des langages de programmation sont libres de contexte.



Types de grammaires

ÉCOLE POLYTECHNIQUE MONTRÉAL

- **Grammaires contextuelles** (type 1)
  - Règles de la forme $\alpha A \beta \rightarrow \alpha \gamma \beta$ où A est un non terminal et α, β et γ sont un ensemble de terminaux et de non terminaux. α et β peuvent être vides, mais γ est non-vide.
  - Il s'agit de tous les langages reconnaissables par un automate linéairement borné (forme restreinte d'une machine de Turing non déterministe.
- **Grammaires syntagmatiques (ou non restreintes)** (type 0)
  - Aucune restriction sur les règles.
  - Génèrent tous les langages reconnaissables par une machine de Turing

Types de grammaires

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Grammaires régulières (Type 3)
  - Reconnues par l'analyse lexicale (automate)
- Grammaires libres de contexte (Type 2)
  - Reconnues par l'analyse syntaxique
- Grammaires contextuelles (Type 1)
  - Partiellement reconnues par l'analyse sémantique (table de symboles)
- Grammaires non restreintes (Type 0)
  - Machine de Turing

Exemples
Langages non libres de contexte

ÉCOLE POLYTECHNIQUE MONTRÉAL

- L1 = {wcw | w ∈ (a|b)* }
  - Problème de vérifier la déclaration d'un identificateur de longueur arbitraire avant son utilisation
  - Solution: table des symboles et routines associées

- L2 = {$a^n b^m c^n d^m$ | n >= 1, m >= 1}
  - Problème de vérifier la correspondance entre paramètres formels et paramètres actuels dans les définitions et les appels à procédures
  - Solution: analyse sémantique

Arbre d'analyse syntaxique

ÉCOLE POLYTECHNIQUE MONTRÉAL

- L'arbre d'analyse syntaxique (ou arbre de parsage) montre comment le symbole de départ d'une grammaire dérive une chaîne du langage.

- L'arbre de parsage représente la structure syntaxique d'une chaîne en entrée par rapport à une grammaire.

- **Exemple :** 9-5+2 pour la grammaire suivante
  - *list* → *list* + *digit*
    list → *list* – *digit*
    list → *digit*
    digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |9

Exercice 1.2

ÉCOLE POLYTECHNIQUE MONTRÉAL

- En utilisant les symboles suivants, écrivez une grammaire permettant de reconnaître la multiplication et la division d'un nombre aléatoire de chiffres.

Donnez un arbre de parsage pour la chaîne 8*2*4/2*9/1/2.

- Symboles non terminaux: *list*, *digit*
- Symboles terminaux: *, /, les chiffres de 0 à 9

Exercice 2

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Soit la grammaire CFG suivante:
  - S → AB A → AA | Aa | aa B → BB | Bb | bb

1. Identifiez les éléments N, T, P, S de la définition de la grammaire G = (N, T, P, S)

2. Dites si les chaînes suivantes appartiennent au langage et, pour celles qui y appartiennent, dessinez l'arbre d'analyse syntaxique:

a	*a a b b*
a a	*a a a b b*
a a a	*a b b b*
a b	*a a b b b*
a a b	*a a a b b b*
a a a b	*a b a b*
a b b	*a a b b a a b b*

Grammaires ambiguës

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Une grammaire est ambiguë si elle peut produire plus qu'un arbre d'analyse syntaxique pour la même chaîne de caractères en entrée (c'est-à-dire plus que une dérivation gauche ou droite).

- Le problème de déterminer si une CFG est ambiguë est indécidable (NP-complet).
  - Par contre, il ne suffit que d'un exemple pour démontrer qu'elle est ambiguë.

- Lire les sections 2.2.4 et 4.2.5 du livre.

Éliminer l'ambiguïté

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Pour un parseur, il est préférable qu'une grammaire ne soit pas ambiguë.

- Solutions à l'ambiguïté:
  - Modifier la grammaire
    - Exemple: A → A + A | A – A | a
    - Il y a deux dérivations possibles pour la chaîne « a + a + a »
    - Grammaire non ambiguë pour le même langage:
      A → A + a | A – a | a
  - Modifier le langage
  - Ajouter des règles d'élimination de l'ambiguïté
    - Élimination des arbres de parsage incorrects

Éliminer l'ambiguïté Exemple

ÉCOLE POLYTECHNIQUE MONTRÉAL

- *stmt* → **if** *expr* **then** *stmt* |
  if *expr* **then** *stmt* **else** *stmt* |
  other
- Ambiguë, car la chaîne suivante a deux arbres de parsage:

If E_1 **then if** E_2 **then** S_1 **else** S_2

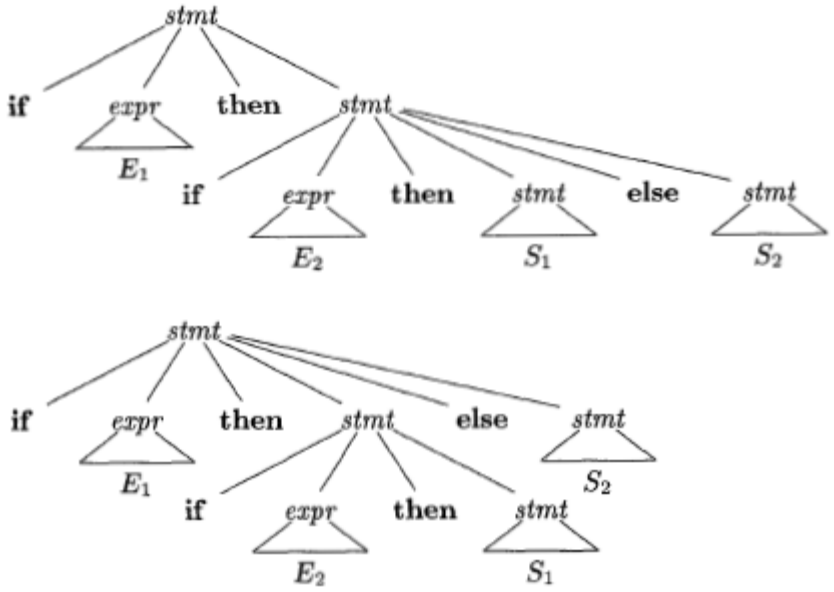


Figure 4.9: Two parse trees for an ambiguous sentence

Éliminer l'ambiguïté Exemple

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Solution en modifiant la grammaire:

$$
\begin{array}{rcl}
\textit{stmt} & \rightarrow & \textit{matched\_stmt} \\
 & | & \textit{open\_stmt} \\
\textit{matched\_stmt} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{matched\_stmt} \textbf{ else } \textit{matched\_stmt} \\
 & | & \textbf{other} \\
\textit{open\_stmt} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \\
 & | & \textbf{if } \textit{expr} \textbf{ then } \textit{matched\_stmt} \textbf{ else } \textit{open\_stmt}
\end{array}
$$

- Solution en modifiant le langage:
  - *stmt* → **if** *expr* **then** *stmt* ***fi*** |
    if *expr* **then** *stmt* **else** *stmt* **fi** |
    other
  - *stmt* → **if** *expr* **then** { *stmt* } | **if** *expr* **then** { *stmt* } **else** { *stmt* } | **other**

- Solution avec des règles d'élimination:
  - Toujours associer les **else** avec le **then** non associé le plus près.

Exercice 3

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Soit la grammaire de l'exercice 2:
  - S → AB A → AA | Aa | aa B → BB | Bb | bb

1. Montrez que la grammaire est ambiguë.

2. Éliminez l'ambiguïté en réécrivant la grammaire pour en obtenir une équivalente, mais sans ambiguïté.

3. Expliquez brièvement pourquoi votre grammaire est équivalente.

Associativité des opérateurs

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Une grammaire se doit de représenter l'associativité des opérateurs.

- **Exemple :**
  - 9+5+2 est équivalent à (9+5)+2 parce que le + est associatif à gauche.
    - *list* → *list* + *digit* | *digit*
  - Dans un langage de programmation, a=b=c est équivalent à a=(b=c) parce que l'opérateur = est associatif à droite.
    - *list* → *id* = *list* | *id*

- Écrivons les grammaires correspondantes.

Précédence des opérateurs

ÉCOLE POLYTECHNIQUE MONTRÉAL

- En arithmétique, les opérateurs * et / ont une précédence plus élevée que les opérateurs + et – .

- **Exemple :**
  - La grammaire doit donc reconnaître que 9+5*2 est équivalent à 9+(5*2).

- Quels sont les problèmes avec la grammaire suivante:
  - E → E + E | E * E | (E) | id

Précédence des opérateurs

ÉCOLE POLYTECHNIQUE MONTRÉAL

- La grammaire doit donc reconnaître que 9+5*2 est équivalent à 9+(5*2).

- Quels sont les problèmes avec la grammaire suivante:
  - E → E + E | E * E | (E) | id

1. La grammaire est ambiguë.
2. Plusieurs arbres de parsage sont possibles et ne garantissent pas la priorité des opérations.

Précédence des opérateurs - Solution

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Quels sont les problèmes avec la grammaire suivante:
  - E → E + E | E * E | (E) | id
- Une grammaire qui reconnaît la priorité des opérateurs et l'associativité à gauche:
  - E → E + T | T
  - T → T * F | F
  - F → (E) | id
- Lire le livre du Dragon section 4.1.2

TP1 – Grammaire et parseur

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Écrire la grammaire pour générer un parseur pour le « langage des expressions ».
- JavaCC est une générateur de parseurs descendants-récursifs (Semaine 2).
  - *Java Compiler Compiler.*
- Les parseurs LL (descendants-récursifs) ne supportent pas la récursivité à gauche.

Récursivité à gauche

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Quels sont les problèmes avec la grammaire suivante:
  - E → E + T | T
  - T → T * F | F
  - F → (E) | id

1. La grammaire est récursive à gauche.

Récursivité à gauche - Solution

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Quels sont les problèmes avec la grammaire suivante:
  - E → E + T | T
  - T → T * F | F
  - F → (E) | id

- Il faut éliminer la récursivité à gauche:

E -> T E'

E' -> + T E' | ε

T -> F T'

T' -> * F T' | ε

F -> (E) | id