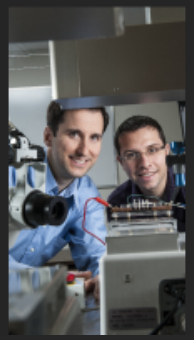
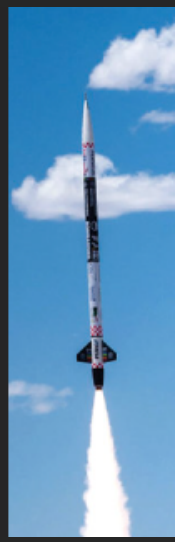




POLYTECHNIQUE MONTRÉAL

LE GÉNIE EN PREMIÈRE CLASSE

LOG3210
Éléments de langage et compilateurs

Analyse syntaxique – Parseurs ascendants

Semaine 3

PLAN

1. Chapitre 4 – Parseurs descendants (Suite et Fin)
   - Implémentation d'un parseur LL
   - Génération d'un parseur par JavaCC
2. Chapitre 4 – Parseurs ascendants (Début)

CHAPITRE 4 – JAVACC

JavaCC est un générateur de parseur

Java Compiler Compiler

JJTree ajoute des méthodes utilitaires à la grammaire *.jjt* et produit un fichier *.jj*

JavaCC génère un parseur à partir de ce fichier *.jj* (voir *LEParser.java*)

CHAPITRE 4 – JAVACC

Règle de production: Program → Block <EOF>

```
final public ASTProgram Program() throws
                                        ParseException {

  ASTProgram jjtn000 =
             new ASTProgram(this, JJTPROGRAM);
  boolean jjtc000 = true;
  jjtree.openNodeScope(jjtn000);
  try {
      Block();
      jj_consume_token(0);
      jjtree.closeNodeScope(jjtn000, true);
```

Program → Block <EOF>

CHAPITRE 4 – JAVACC

Règle de production:

E → T (<+> T)*

```
final public void AddExpr() {

  try {
      MultExpr();
  label_2:
      while (true) {
        switch (jj_ntk) {
        case PLUS:
        case MINUS:
          ;
          break;
        default:
          break label_2;
        }
        switch (jj_ntk) {
        case PLUS:
          jj_consume_token(PLUS);
          break;
        case MINUS:
          jj_consume_token(MINUS);
          break;
        default:
           jj_consume_token(-1);
          throw new
                ParseException();
        }
        MultExpr();
      }
```

ÉCOLE
POLYTECHNIQUE
MONTRÉAL

LOG3210
Cours 3

Analyse syntaxique – Parseurs ascendants

Table de parsage
Parseurs descendants

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Les parseurs *top-down* construisent une table de parsage, plus rapide à l'exécution que les méthodes récursives.

- Cette table est un tableau à deux dimensions M[*A*, *a*] où
  - *A* est un non terminal
  - *a* est un terminal ou le symbole $ (fin de l'entrée)

- Exemple:

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

Table de parsage
Parseurs descendants

ÉCOLE POLYTECHNIQUE MONTRÉAL

Example 4.32: For the expression grammar (4.28), Algorithm 4.31 produces the parsing table in Fig. 4.17. Blanks are error entries; nonblanks indicate a production with which to expand a nonterminal.

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ **id**			$F \to (E)$		

Figure 4.17: Parsing table M for Example 4.32

Construction de la table Parseurs descendants

ÉCOLE POLYTECHNIQUE MONTRÉAL

- On construit la table de parsage en suivant les étapes suivantes (algorithme 4.31 du livre):

- Pour chaque production $A \rightarrow \alpha$ de la grammaire:
  1. Pour chaque terminal *a* dans FIRST(α), ajouter $A \rightarrow \alpha$ dans M[*A*,*a*]
  2. Si ε est dans FIRST(α), alors pour chaque terminal *b* dans FOLLOW(*A*), ajouter $A \rightarrow \alpha$ dans M[*A*,*b*].
     Si ε est dans FIRST(α) et $ est dans FOLLOW(*A*), ajouter également $A \rightarrow \alpha$ à M[*A*,$]
- Si, à la fin, il n'y a aucune production dans M[*A*,*a*], insérer **error** dans M[*A*,*a*]

Exercice 1

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Bâtir la table de parsage de l'analyseur descendant pour la grammaire suivante:
  - $A \rightarrow BC$
  - $B \rightarrow bB \mid \varepsilon$
  - $C \rightarrow cC \mid \varepsilon$
- Pour chaque case où il y a **error**, donnez un message d'erreur pertinent pour l'utilisateur

Exercice 1 (solution)

- FIRST(*A*) = { *b*, *c*, ε } FOLLOW(*A*) = { $ }
- FIRST(*B*) = { *b*, ε } FOLLOW(*B*) = { *c*, $ }
- FIRST(*C*) = { *c*, ε } FOLLOW(*C*) = { $ }

Non terminaux	Symboles d'entrée b	c	$
A	A → BC	A → BC	A→BC
B	B → bB	B → ε	B → ε
C	ENCOUNTERED b, EXPECTING c or $	C→cC	C → ε

Table de parsage
Parseurs descendants

ÉCOLE POLYTECHNIQUE MONTRÉAL

- L'ambiguïté se manifeste dans la table de parsage.
  - Certains langages n'ont pas de grammaire LL(1)

NON-TERMINAL	INPUT SYMBOL					
	a	b	e	i	t	$
S	$S \to a$			$S \to iEtSS'$		
S'			$S' \to \epsilon$ $S' \to eS$			$S' \to \epsilon$
E		$E \to b$				

Figure 4.18: Parsing table M for Example 4.33

Analyse syntaxique ascendante (*bottom-up parsing*)

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Aussi appelé parsage décalage / réduction (*shift/reduce parsing*)
- Forme générale: LR parsing
- Idée:
  - Construire un arbre de parsage à partir des feuilles vers la racine

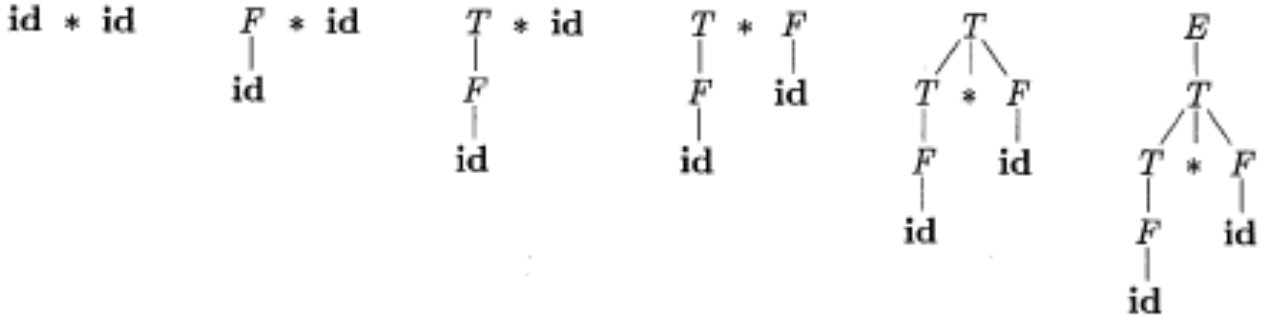
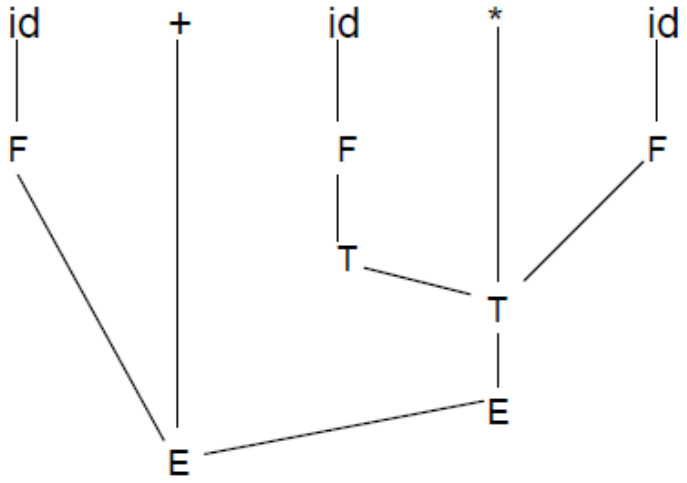


Figure 4.25: A bottom-up parse for **id * id**

Analyse syntaxique ascendante

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Exemple:



- **Décalage:** lire un jeton supplémentaire de l'entrée.
- **Réduction**: remplacer la partie droite d'une règle de production, qui correspond à une sous chaîne particulière, par la partie gauche de la règle.
- Si la sous chaîne est bien choisie, une dérivation par la droite inversée est effectuée.

Exemple

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Soit:
  - S → aABe A → Abc | b B → d
- Entrée:
  - abbcde (A → b)
  - aAbcde (A → Abc)
  - aAde (B → d)
  - aABe (S → aABe)
  - S
- On voit bien la dérivation par la droite inversée
  - S → aABe → aAde → aAbcde → abbcde

Handle (manche)

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Définition informelle
  - Sous-chaîne qui correspond à la partie droite d'une production et pour laquelle une réduction correspond à un pas vers une bonne dérivation par la droite inversée.

- Exemple:
  - S → aABe A → Abc | b B → d

- Entrée:
  - a**b**bcde (handle: b)
  - a**Abc**de (handle: Abc)
  - aA**d**e (handle: d)
  - **aABe** (handle: aABe)
  - S

Handle (manche)

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Définition formelle:
  - Un *handle* de γ est une production $A \rightarrow \beta$ et une position dans γ tels que le remplacement de β par A produit γ^{-1}, où la dérivation droite de γ^{-1} donne γ.
  - Si on a $S \underset{rm}{\overset{*}{\Rightarrow}} \alpha A w \underset{rm}{\Rightarrow} \alpha\beta w$, alors la production $A \rightarrow \beta$ dans la position qui suit α est un *handle* de $\alpha\beta\omega$
- Notes:
  - La chaîne ω doit contenir seulement des symboles terminaux.
  - Si la grammaire est ambiguë, il peut exister plus d'une dérivation à droite et donc plusieurs *handles* pour γ.
  - Si la grammaire est non-ambiguë, chaque chaîne γ a un seul *handle*.

Handle (manche) Ambiguïté

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Considérez la grammaire suivante:
  E → E **+** E | E ***** E | **(** E **)** | **id**

E	$\overset{rm}{\Rightarrow}$ **E * E**	E	$\overset{rm}{\Rightarrow}$ **E + E**
E * E	$\overset{rm}{\Rightarrow}$ E * **id**	E + E	$\overset{rm}{\Rightarrow}$ E + **E * E**
E * id	$\overset{rm}{\Rightarrow}$ **E + E** * id	E + E * E	$\overset{rm}{\Rightarrow}$ E + E * **id**
E + E * id	$\overset{rm}{\Rightarrow}$ E + **id** * id	E + E * id	$\overset{rm}{\Rightarrow}$ E + **id** * id
E + id * id	$\overset{rm}{\Rightarrow}$ **id** + id * id	E + id * id	$\overset{rm}{\Rightarrow}$ **id** + id * id

Handle pruning (élagage du manche)

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Soit la séquence de dérivations suivante:

$$S = \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \cdots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = w$$

- Le *handle pruning* permet de reconstruire la dérivation à droite inversée.

- Principe de base:
  - Identifier le *handle* β_n dans γ_n
  - Réduire β_n par la tête de la production pertinente A_n → β_n pour obtenir γ_{n-1}.

- La localisation d'un *handle* n'a pas été expliquée encore.

Exercice 2

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Soit la grammaire S → 0 S 1 | 0 1
  1. Calculer le *handle* pour chacune des dérivations à droite inversées à partir de la chaîne 000111

- Soit la grammaire S → S S + | S S * | *a*
  2. Calculer le *handle* pour chacune des dérivations à droite inversées à partir de la chaîne *aaa*a++*

Exercice 2 (solution)

ÉCOLE POLYTECHNIQUE MONTRÉAL

1. S → 0 S 1 | 0 1
   Entrée:
   - 00**01**11 (handle: 01)
   - 0**0S1**1 (handle: 0S1)
   - **0S1** (handle: 0S1)
   - S
2. S → S S + | S S * | *a*
   Entrée:
   - ***a***a*a*++ (handle: a)
   - S**a**a*a++ (handle: a)
   - SS**a***a++ (handle: a)
   - S**SS***a++ (handle: SS*)
   - SS**a**++ (handle: a)
   - S**SS+**+ (handle: SS+)
   - **SS+** (handle: SS+)
   - S

Parseur décalage/réduction (*shift-reduce parsing*)

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Problématique
  - Localiser un *handle*
  - Choisir la production à appliquer
- Structures utilisées
  - Pile (*stack*)
  - Tampon d'entrée (*input buffer*)

STACK	INPUT
$	*w* $

- Symboles spéciaux
  - Fond de la pile: $
  - Fin de l'entré: $

Parseur décalage/réduction Implémentation

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Algorithme:
  - RÉPÉTER
    1. Décaler 0 ou plus de symboles d'entrée sur la pile jusqu'à la formation d'un *handle* *B* sur la pile
    2. Réduire *B* par la partie gauche de la règle appropriée
  - Tant que (pas d'erreur) ou (état d'acceptation)

- L'état d'acceptation est défini comme le moment où la pile contient le symbole de départ et le tampon d'entrée est vide.

STACK	INPUT
$ S	$

Parseur décalage/réduction Actions possibles

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Il y a quatre actions pouvant être effectuées par le parseur:
  - **Décalage** (*shift*): le prochain symbole d'entrée est empilé sur la pile
  - **Réduction** (*reduce*): un handle se trouve sur le dessus de la pile
    1. Identifier le *handle* à réduire sur le dessus de la pile
    2. Trouve la production
    3. Remplacer la partie DROITE du handle, sur la pile
  - Note: le *handle* apparaît toujours sur le dessus de la pile, JAMAIS à l'intérieur (explication: entrée traitée de gauche à droite + dérivation à droite)

Parseur décalage/réduction
Actions possibles

ÉCOLE POLYTECHNIQUE MONTRÉAL

- **Acceptation**: terminaison du parsage avec succès

- **Erreur:** détection d'une erreur syntaxique. Mise en marche de routine de correction d'erreurs.

STACK	INPUT	ACTION
$	$\mathbf{id}_1 * \mathbf{id}_2$ $	shift
$ $\mathbf{id}_1$	$* \mathbf{id}_2$ $	reduce by $F \rightarrow \mathbf{id}$
$ F	$* \mathbf{id}_2$ $	reduce by $T \rightarrow F$
$ T	$* \mathbf{id}_2$ $	shift
$ $T *$	$\mathbf{id}_2$ $	shift
$ $T * \mathbf{id}_2$	$	reduce by $F \rightarrow \mathbf{id}$
$ $T * F$	$	reduce by $T \rightarrow T * F$
$ T	$	reduce by $E \rightarrow T$
$ E	$	accept

Figure 4.28: Configurations of a shift-reduce parser on input $\mathbf{id}_1 * \mathbf{id}_2$

Parseur décalage/réduction
Conflits durant le parsage

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Pour certaines grammaires CFG, tous les parseurs par décalage/réduction atteignent une configuration où l'état de la pile et le prochain symbole d'entrée ne permettent pas de décider entre:
  1. Décaler ou réduire (*shift/reduce conflict*)
  2. Réduire ou réduire (*reduce/reduce conflict*), à cause de plusieurs productions possibles.

- Ces grammaires ne sont pas LR(*k*).
  - Les langages de programmation ont habituellement des grammaires LR(1).
  - Une grammaire ambiguë n'est pas LR(*k*), pour tout *k*.
  - On lève l'ambiguïté *shift/reduce* en priorisant la réduction ou le décalage.

Conflits (exemple)

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Soit la grammaire:

$$
\begin{array}{lcl}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
 & | & \textbf{other}
\end{array}
$$

- Et la configuration:

STACK	INPUT
$\cdots$ **if** *expr* **then** *stmt*	**else** $\cdots$ \$

- On a un conflit décalage/réduction
  - Dans ce cas, on voudrait probablement faire un décalage afin d'associer chaque *else* au *then* précédent le plus près.

Conflits (exemple 2)

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Soit une grammaire qui utilise les parenthèses pour les appels de procédures et pour la référence à des cases dans un tableau:

(1)	*stmt*	→	**id** (*parameter_list*)
(2)	*stmt*	→	*expr* := *expr*
(3)	*parameter_list*	→	*parameter_list* , *parameter*
(4)	*parameter_list*	→	*parameter*
(5)	*parameter*	→	**id**
(6)	*expr*	→	**id** (*expr_list*)
(7)	*expr*	→	**id**
(8)	*expr_list*	→	*expr_list* , *expr*
(9)	*expr_list*	→	*expr*

Figure 4.30: Productions involving procedure calls and array references

Conflits (exemple 2)

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Soit la configuration suivante, pour une instruction p(i, j):

STACK	INPUT
··· **id** (**id**	, **id**) ···

- **id** doit être réduit, mais il y a deux productions possibles (5 et 7)
  - (5) est correcte si *p* est une procédure
  - (7) est correcte si *p* est un tableau

- On a un conflit réduction/réduction
  - Solution possible:
    Changer le jeton **id** de la production (1) par un jeton **procid**, et modifier l'analyseur lexical pour différentier les types de jetons.

Exercice 3

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Soit la grammaire S → 0 S 1 | 0 1
  1. Donner l'état de la pile et du tampon d'entrée pour chaque étape du parsage ascendant de la chaîne 000111

- Soit la grammaire S → S S + | S S * | *a*
  2. Donner l'état de la pile et du tampon d'entrée pour chaque étape du parsage ascendant de la chaîne *aaa*a++*

Parseur LR

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Capables de reconnaître presque tous les langages libres de contexte
  - Il existe des grammaires CFG qui ne sont pas LR, mais elles peuvent être évitées dans le contexte des langages de programmation.
- On utilise habituellement les parseurs à décalage/réduction sans retour-arrière (*backtracking*)
- LR(1) ⊃ LL(1)
- Les erreurs syntaxiques sont détectées aussitôt que possible dans lecture de gauche à droite de l'entrée

Parseur LR

ÉCOLE POLYTECHNIQUE MONTRÉAL

- La construction « à la main » est très exigeante; on utilise des générateurs de parseurs LR
  - Yacc, Bison, SableCC, etc.
- Types de tables d'analyse LR:
  - ***Simple* LR** (SLR): table de petite taille, moins puissante
  - ***Canonical* LR** (LR): table de grande taille, plus puissante
  - ***Look Ahead LR*** (LALR): table de taille moyenne et puissance intermédiaire
- L'algorithme de parsage est toujours le même, peu importe le type de table utilisé

Items LR(0)

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Les items permettent de déterminer quand faire un *shift* ou un *reduce*
- Un item LR(0) (ou simplement item) d'une grammaire *G* est une production de *G* avec un point dans une position quelconque de la production.
  - Pour la production A→) $A \rightarrow \cdot XYZ$ items suivants:

$$A \rightarrow X \cdot YZ$$
$$A \rightarrow XY \cdot Z$$
$$A \rightarrow XYZ \cdot$$

  - Pour la production A → ε, on a un seul item: A → .

- Un item indique jusqu'où une production a été traitée jusqu'à maintenant.
  - Exemple: *A* → *X*·*YZ* signifie qu'une chaîne dérivable à partir de *X* a été traitée et que nous espérons maintenant voir une chaîne dérivable à partir de *YZ*.

Automate LR(0)

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Automate déterministe fini permettant d'effectuer les décisions de parsage.
- Est construit grâce à l'ensemble LR(0) canonique
  - Il s'agit d'un ensemble d'ensembles (*collection of sets*) d'items
- Trois éléments doivent être définis pour construire l'ensemble LR(0) canonique:
  - Grammaire augmentée
  - Fonction CLOSURE
  - Fonction GOTO

Grammaire augmentée

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Supposons une grammaire *G* avec un symbole de départ *S*.
- La grammaire augmentée *G'* est *G* avec un nouveau symbole de départ *S'* et la production *S'* → *S*.
- But du nouveau symbole de départ: permettre au parseur de savoir quand il doit arrêter de parser et accepter l'entrée.

Fonction CLOSURE

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Supposons que *I* est un ensemble d'items d'une grammaire *G*.

- CLOSURE(*I*) est l'ensemble d'items construit à partir de *I* via les deux règles suivantes:
  1. Initialement, ajouter chaque item de *I* dans CLOSURE(*I*)
  2. Si A → $\alpha \cdot B\beta$ est dans CLOSURE(*I*) et que *B* → γ est une production, alors ajouter l'item B →$\cdot\gamma$ à CLOSURE(*I*), s'il n'est pas déjà présent.
     Appliquer cette règle jusqu'à ce qu'aucun nouvel item ne puisse être ajouté à CLOSURE(*I*)

Fonction CLOSURE Exemple

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Soit la grammaire suivante:

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

- Si I = { [$E' \rightarrow \cdot E$]}, alors CLOSURE(I) contient les éléments suivants:
  - $E' \rightarrow \cdot E$
  - $E \rightarrow \cdot E + T$ $\quad$ $E \rightarrow \cdot T$
  - $T \rightarrow \cdot T * F$ $\quad$ $T \rightarrow \cdot F$
  - $E \rightarrow \cdot (E)$ $\quad$ $E \rightarrow \cdot \mathbf{id}$

Fonction CLOSURE Algorithme

ÉCOLE POLYTECHNIQUE MONTRÉAL

```
SetOfItems CLOSURE(I) {
        J = I;
        repeat
                for ( each item A → α·Bβ in J )
                        for ( each production B → γ of G )
                                if ( B → ·γ is not in J )
                                        add B → ·γ to J;
        until no more items are added to J on one round;
        return J;
}
```

Figure 4.32: Computation of CLOSURE

Fonction GOTO

ÉCOLE POLYTECHNIQUE MONTRÉAL

- GOTO(*I*, *X*)
  - *I* est un ensemble d'items
  - *X* est un symbole de la grammaire

- GOTO(*I*, *X*) est défini comme la fermeture (CLOSURE) de l'ensemble de tous les items [$A \rightarrow \alpha X \cdot \beta$] tel que [$A \rightarrow \alpha \cdot X \beta$] est dans *I*.
  - Intuitivement, la fonction GOTO est utilisée pour définir les transitions dans l'automate LR(0).
  - Les états correspondent aux ensembles d'items, alors que GOTO(*I*, *X*) spécifie la transition de l'état *I* sous l'entrée *X*.

Fonction GOTO Exemple

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Si *I* = { [*E'* → *E*·], [*E* → *E*·+*T*] }
  alors GOTO(*I*, +) contient les éléments suivants:

$$
\begin{aligned}
E &\rightarrow E + \cdot T \\
T &\rightarrow \cdot T * F \\
T &\rightarrow \cdot F \\
F &\rightarrow \cdot (E) \\
F &\rightarrow \cdot \mathbf{id}
\end{aligned}
$$

- On a fait progresser le point « par-dessus » le symbole +, puis on calcule la fermeture (CLOSURE) sur la règle *E* → *E*+·*T*

Fonction GOTO Algorithme

ÉCOLE POLYTECHNIQUE MONTRÉAL

- L'algorithme suivant construit *C*, l'ensemble d'ensembles d'items LR(0) pour une grammaire augmentée *G'*

```
void items(G') {
        C = CLOSURE({[S' → ·S]});
        repeat
                for ( each set of items I in C )
                        for ( each grammar symbol X )
                                if ( GOTO(I, X) is not empty and not in C )
                                        add GOTO(I, X) to C;
        until no new sets of items are added to C on a round;
}
```

Détermination des ensembles d'items LR(0)

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Construisons les ensembles d'items de la grammaire augmentée suivante, en calculant aussi les GOTO:

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

Utilité des GOTO et CLOSURE

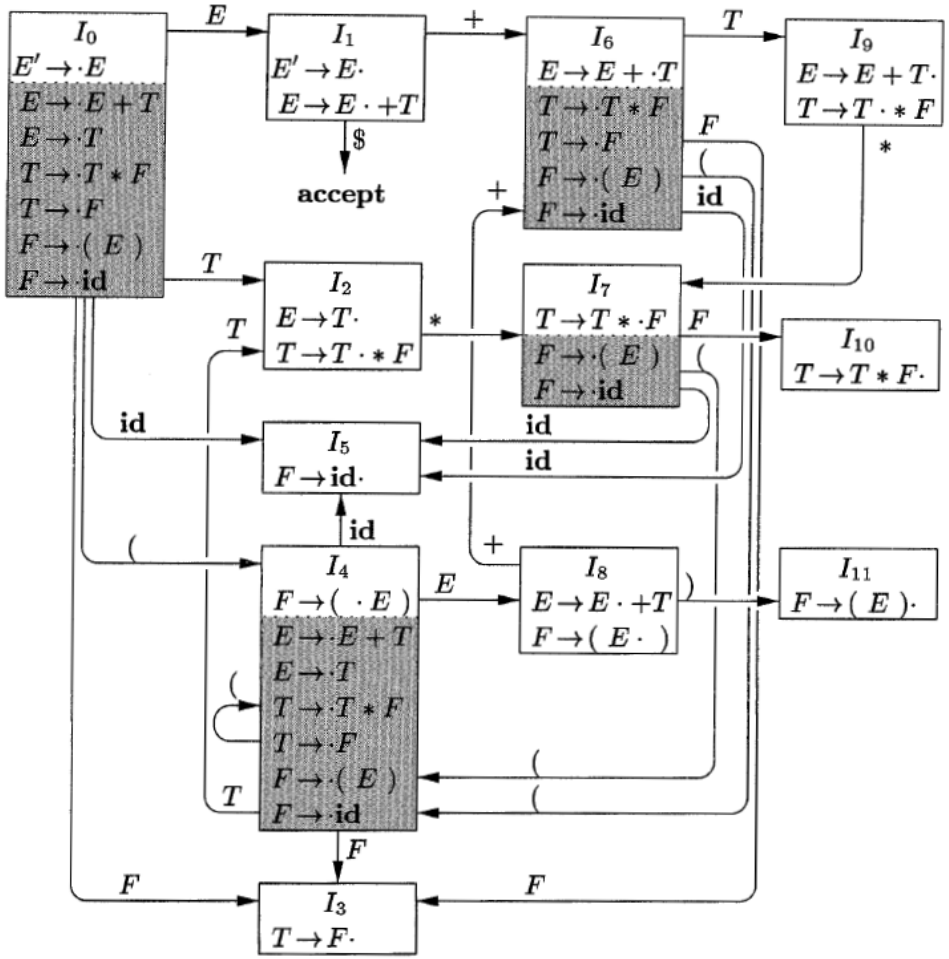


Figure 4.31: LR(0) automaton for the expression grammar (4.1)

Exercice 4

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Soit la grammaire S → S S + | S S * | *a*

1. Augmenter la grammaire
2. Calculer les ensembles d'items pour la grammaire ainsi que la fonction GOTO pour chacun des ensembles obtenus.
   Indice: commencez par I_0 = CLOSURE(*règle de départ*)