Analyse syntaxique ascendante

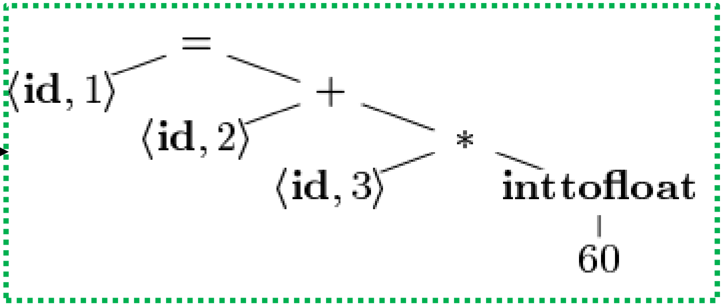
Brahimi Mohammed

Structure d’un compilateur (Front-end)

`position = initial + rate * 60`

Flot de caractères du programme source

Front-end



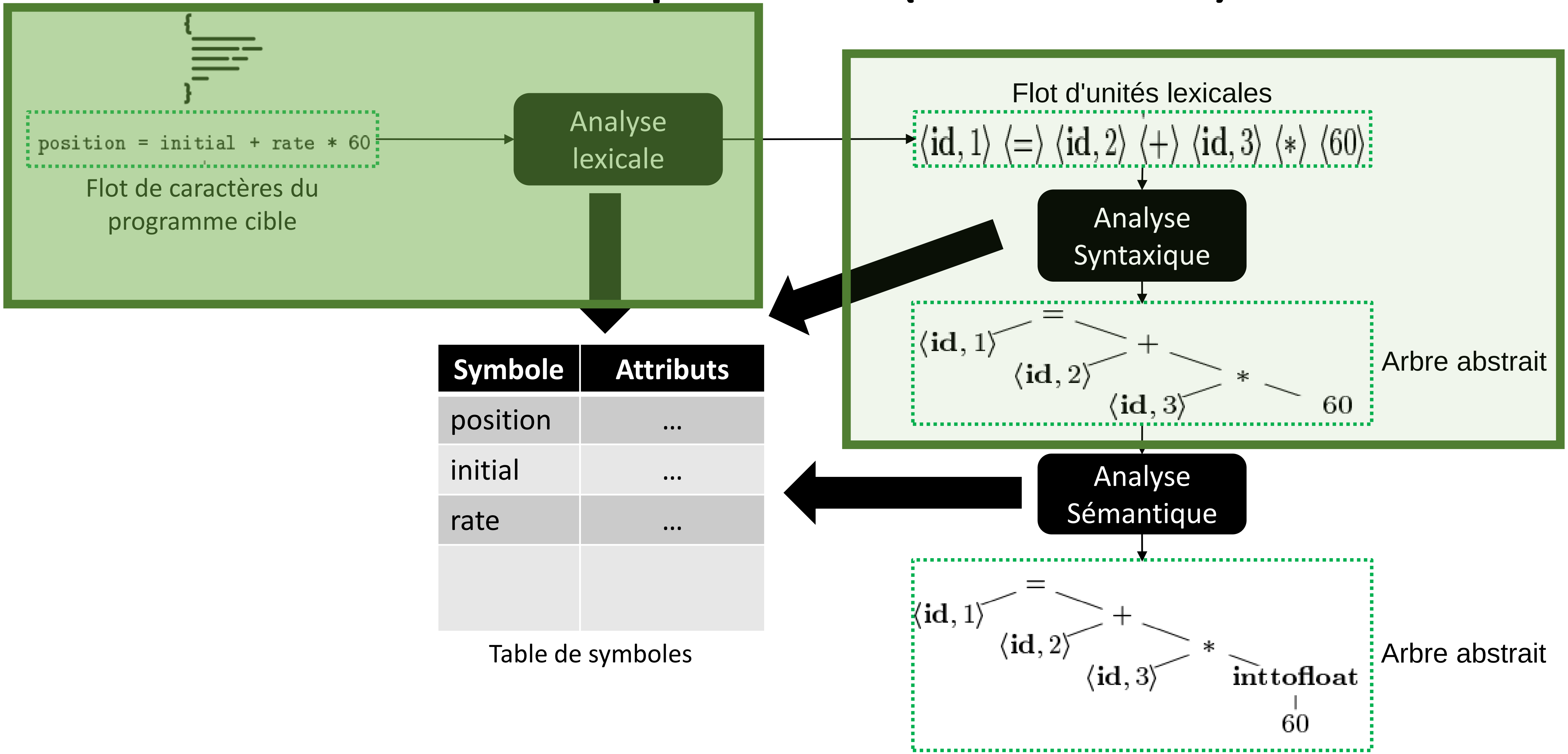
Arbre abstrait

Symbole	Attributs
position	…
initial	…
rate	…

Table de symboles

Le front-end essaye de comprendre le programme source

Structure d’un compilateur (Front-end)



Symbole	Attributs
position	…
initial	…
rate	…

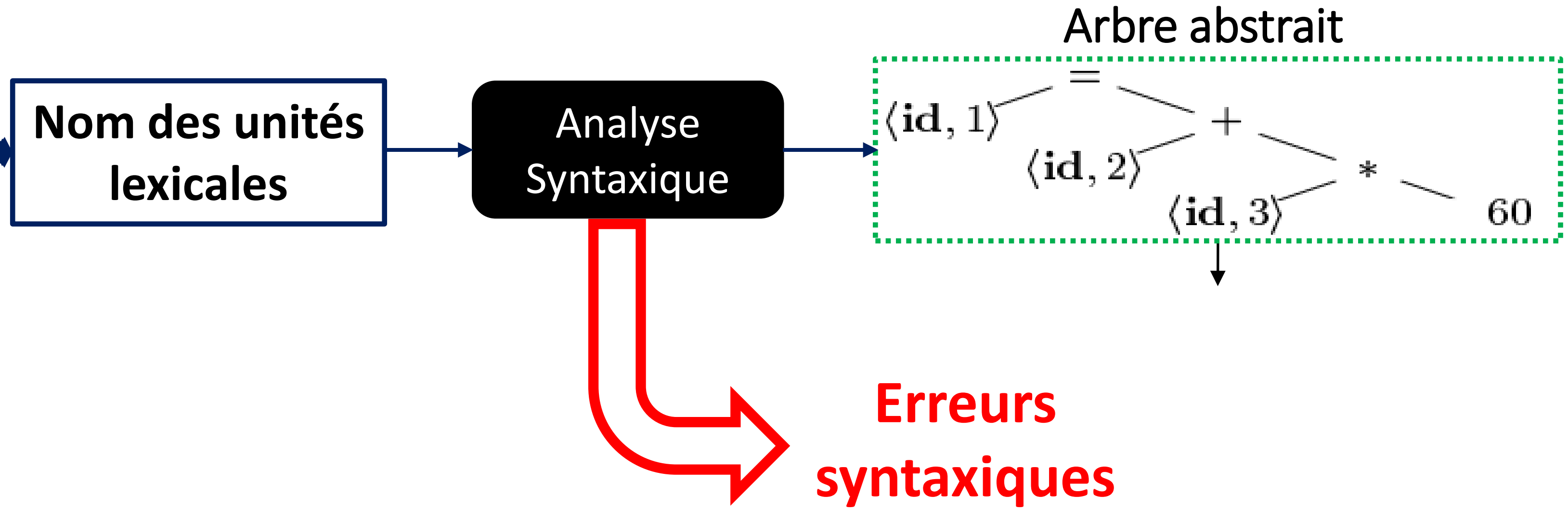
Table de symboles

C'est quoi l’analyse syntaxique ?

- Reçoit une chaine d’**unités lexicales** fournie par l’analyseur lexicale
- Vérifier si cette chaine d’**unités lexicales** respecte la **grammaire** du langage source
- Signaler les **erreurs syntaxiques**
- Construire **un arbre d’analyse** pour les phases suivantes de compilation

C'est quoi l’analyse syntaxique ?

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



Analyse syntaxique ascendante

- Commencer à partir le mot à reconnaitre (**Feuillies de l'arbre)** pour arriver à la (**Racine de l'arbre**)

- Remplacer une partie droite **«** α **»** d'une production **«** $\mathbf{A} \rightarrow \alpha$ **»**

par son non terminale du partie gauche **«**A**» (Réduction)**

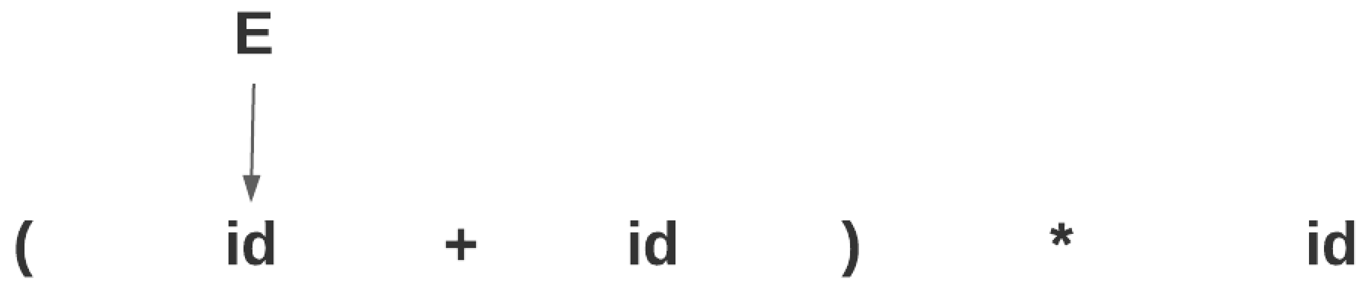
- Arrêter quand le non terminale de départ est atteint

- Construire l'arbre à partir les feuilles tous en montant vers la racine

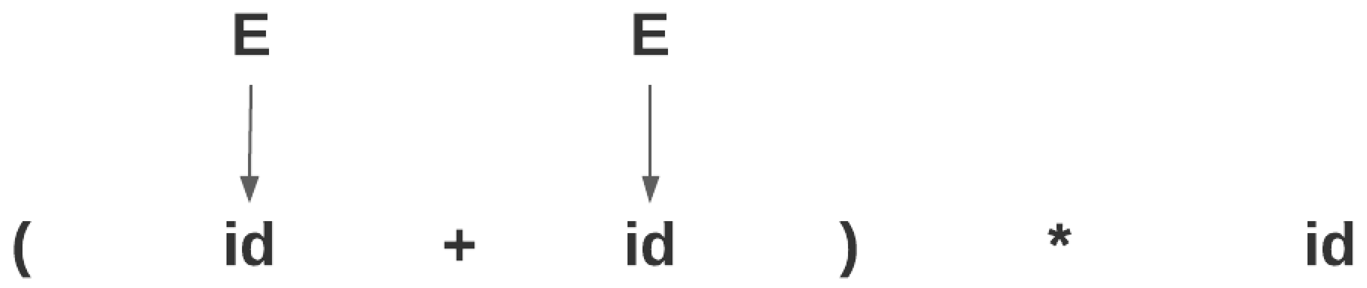
Mot dans les feuilles

(id + id) * id

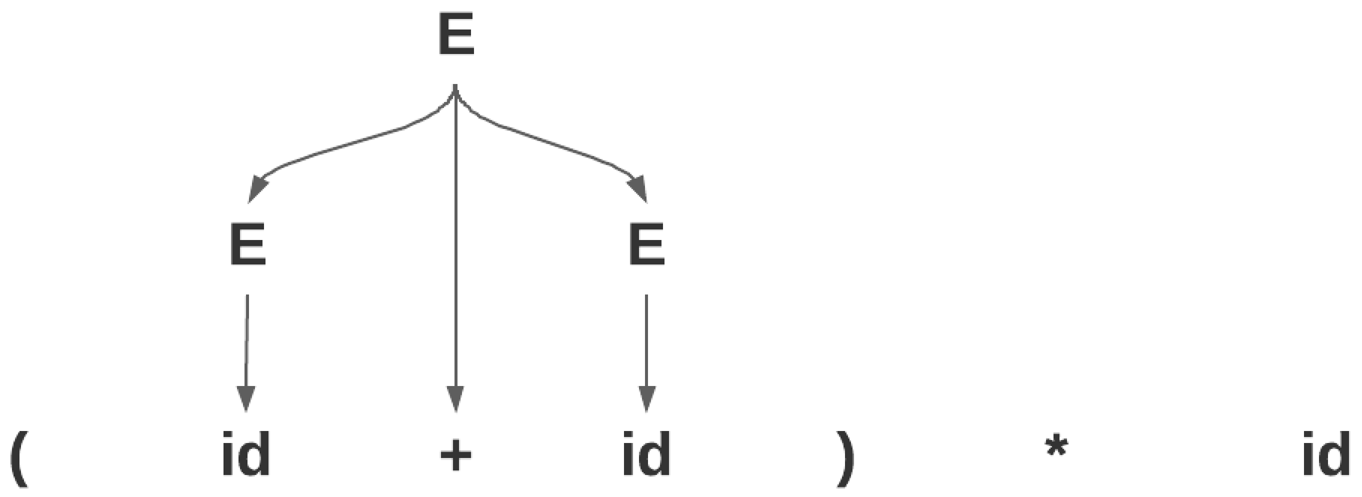
$$(\mathbf{E} + \mathbf{id}) * \mathbf{id} \Longrightarrow (\mathbf{id} + \mathbf{id}) * \mathbf{id}$$



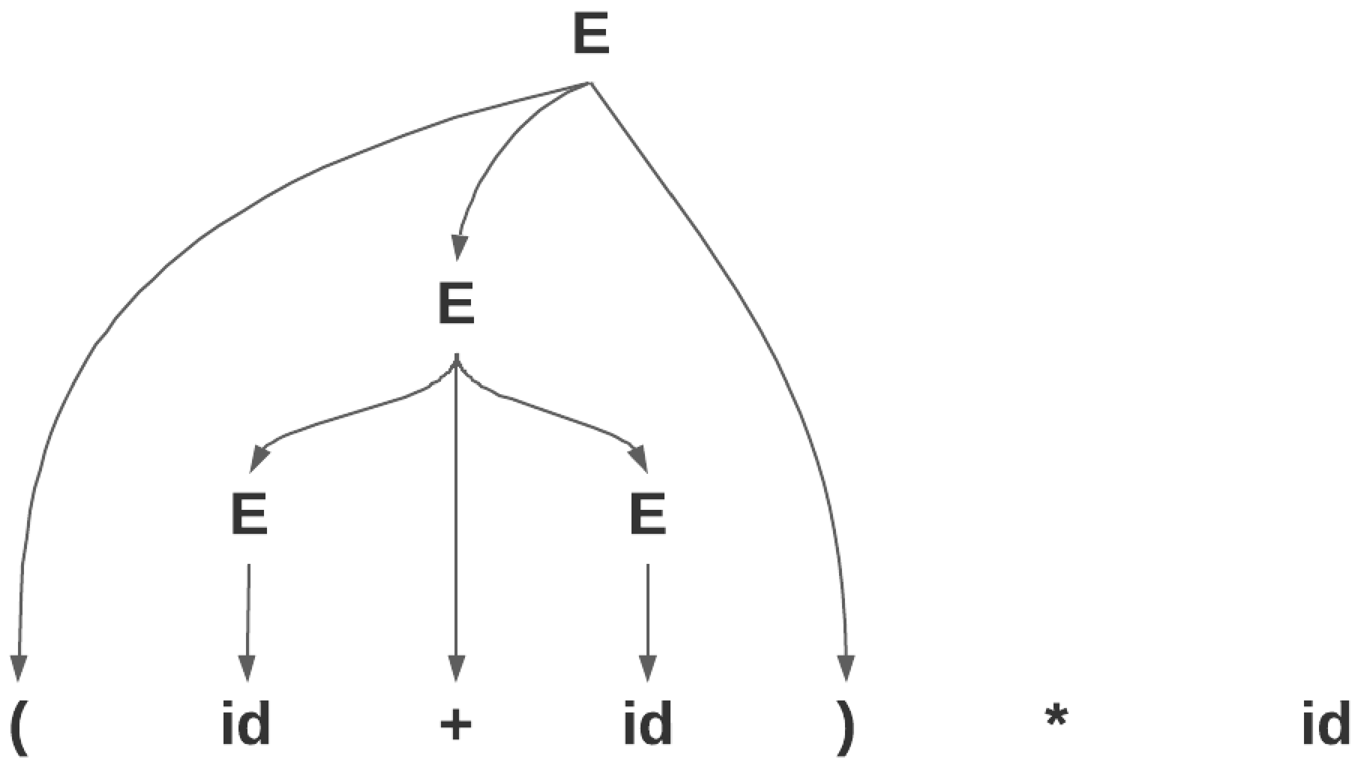
$$(\mathrm{E} + \mathbf{E}) * \mathbf{id} \Longrightarrow (\mathbf{E} + \mathbf{id}) * \mathbf{id}$$



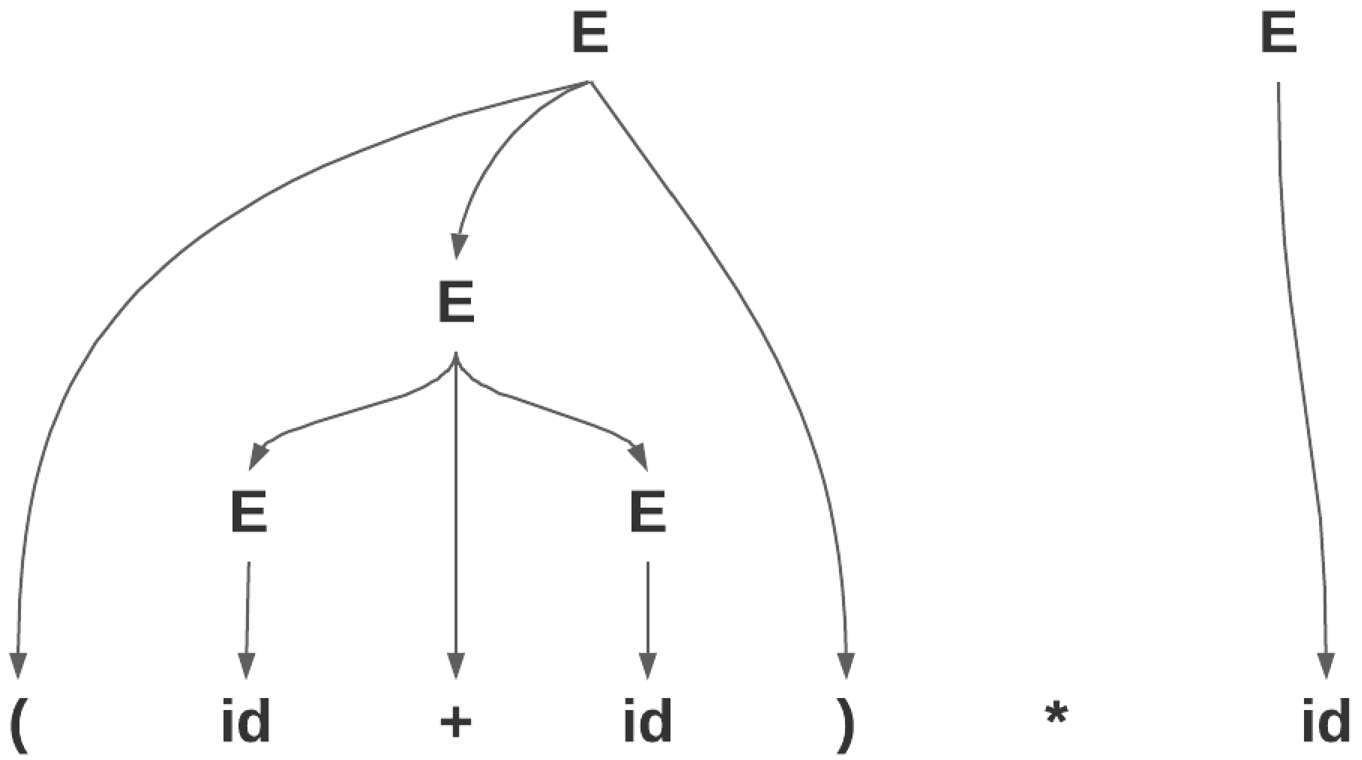
$$(\mathbf{E}) * \mathbf{id} \implies (\mathrm{E} + \mathbf{E}) * \mathbf{id}$$



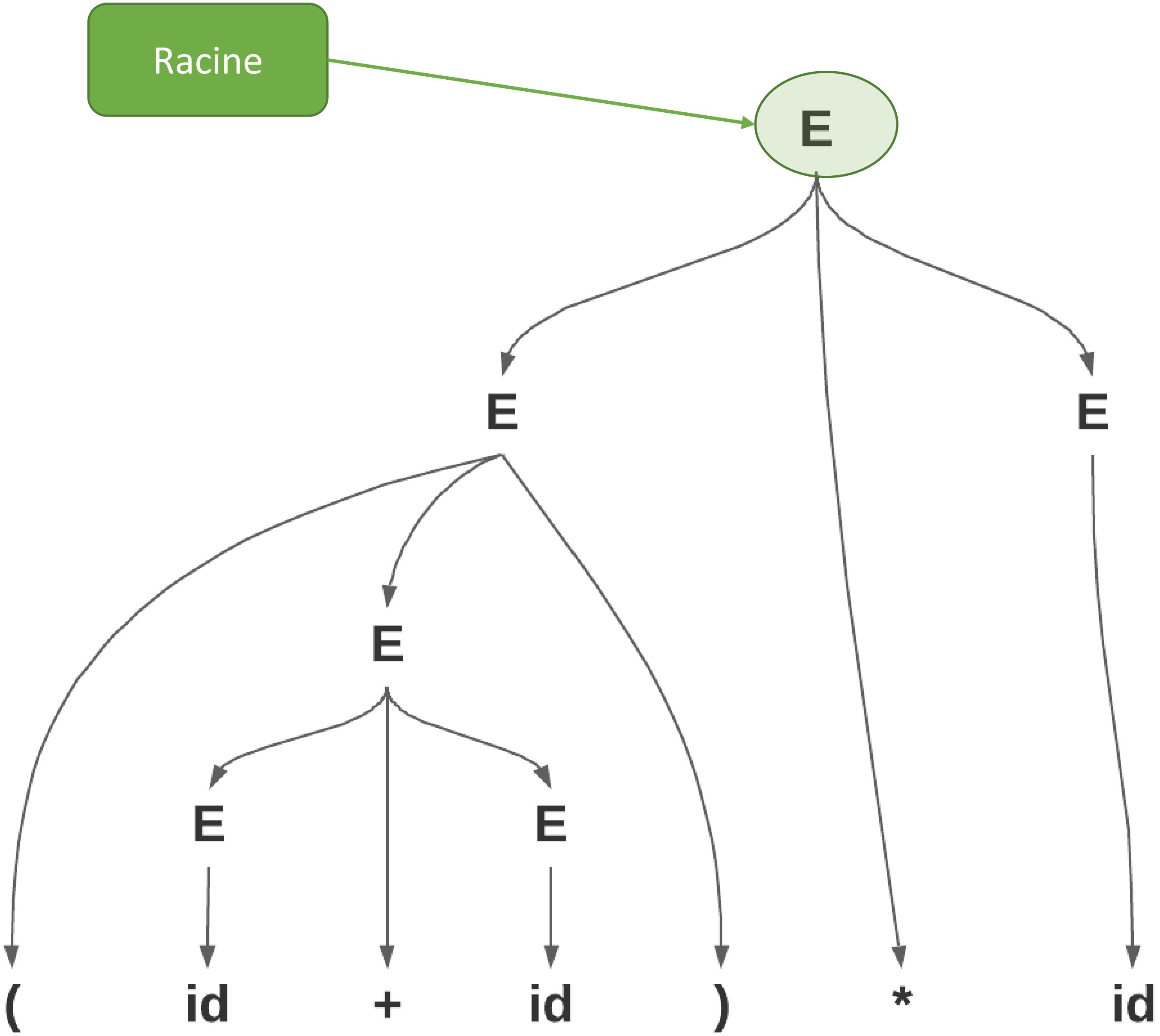
$$\mathbf{E} * \mathbf{id} \Longrightarrow (\mathbf{E}) * \mathbf{id}$$



$$E * \color{green}{\mathbf{E}} \implies \color{green}{\mathbf{E}} * \mathbf{id}$$



$$E \Longrightarrow E * \mathbf{E}$$



Analyse syntaxique ascendante

- Plus générale que l’analyse descendante

- La méthode préférée pour la construction des analyseurs syntaxiques

- Moins d’opérations de conversion de grammaire
  - L’élimination de la récursivité à gauche n’est pas nécessaire
  - La factorisation à gauche n’est pas nécessaire

Exemple

Dérivations

int * int + int (chaine)

int * $\boxed{\text{int}}$ + int $(T \to int)$

$\boxed{\text{int * } \mathbf{T}}$ + int $(T \to int * T)$

T + $\boxed{\text{int}}$ $(T \to int)$

T + $\boxed{\mathbf{T}}$ $(E \to T)$

$\boxed{\text{T + } \mathbf{E}}$ $(E \to T+E)$

$\boxed{\text{E}}$

Réductions

E→T+E | T

T→int∗T | int|

Exemple

Dérivations

int * int + int (chaine)

int * int + int ($T \rightarrow \mathrm{int}$)

int * **T** + int ($T \rightarrow \mathrm{int} * T$)

T + int ($T \rightarrow \mathrm{int}$)

T + **T** ($E \rightarrow T$)

T + **E** ($E \rightarrow T + E$)

E

Réductions

Dérivation droite

Analyse syntaxique ascendante

- **Fait important n° 1**

Un analyseur ascendante est analyseur avec

des dérivations droites (Right most dérivations) inversées

Analyse syntaxique ascendante

- Le fait important n°1 a une conséquence intéressante:
  - Soit $\alpha\beta\mathbf{w}$ une étape d'une analyse ascendante
  - Supposons que la prochaine réduction soit par $\mathbf{X} \rightarrow \beta$
  - Alors $\boldsymbol{w}$ est une chaîne de terminaux

- Pourquoi?
  - Car $\alpha\mathbf{X}\mathbf{w} \rightarrow \alpha\beta\mathbf{w}$ est une étape dans une dérivation droite

Analyseur décalage-réduction

Idée: Diviser la chaîne en deux sous-chaînes :

- Sous-chaîne droite n'est pas encore examinée par l'analyseur
- Sous-chaîne gauche contient des terminaux et des non-terminaux
- Le point de division est marqué par un pointeur d'analyse |

$$\alpha X \mid w$$

Sous-chaîne droite n'est pas encore examinée

Analyseur décalage-réduction

- Analyseur syntaxique ascendant utilise que deux actions :

Décalage

Réduction

Décalage

- Décalage : déplacer le pointeur d'analyse | **une position à droite**
- Décaler un terminal vers la sous chaine gauche

$$\mathbf{ABC \,|\, xyz} \rightarrow \mathbf{ABCx \,|\, yz}$$

Réduction

- Application d’une production inversée (Réduction) à l’extrémité droite de la sous chaine gauche

- Si $\mathbf{A \rightarrow xy}$ est une production, alors

$$CB\mathbf{xy} \mid ijk \xrightarrow{\textbf{Réduction}} CB\mathbf{A} \mid ijk$$

Exemple

| int * int + int

int * int | + int (**Réduction** $T \to int$)

int * T | + int (**Réduction** $T \to int * T$)

T + int | (**Réduction** $T \to int$)

T + T | (**Réduction** $E \to T$)

T + E | (**Réduction** $E \to T + E$)

E | (**Mot accepté**)

Exemple

| int * int + int		(Décalage)
int | * int + int		(Décalage)
int * | int + int		(Décalage)
int * int | + int		(Réduction $T \rightarrow int$)
int * T | + int		(Réduction $T \rightarrow int * T$)
T | + int		(Décalage)
T + | int		(Décalage)
T + int |		(Réduction $T \rightarrow int$)
T + T |		(Réduction $E \rightarrow T$)
T + E |		(Réduction $E \rightarrow T + E$)
E |		(Mot accepté)

Exemple

- int * int + int   **(Décalage)**

int   *   int   +   int

Exemple

| int * int + int **(Décalage)**

int | * int + int **(Décalage)**

int * int + int

Exemple

| int * int + int   **(Décalage)**

int | * int + int   **(Décalage)**

int * | int + int   **(Décalage)**

int   *   int   +   int

Exemple

| int * int + int (Décalage)

int | * int + int (Décalage)

int * | int + int (Décalage)

int * int | + int (Réduction T→int)

int * int + int

Exemple

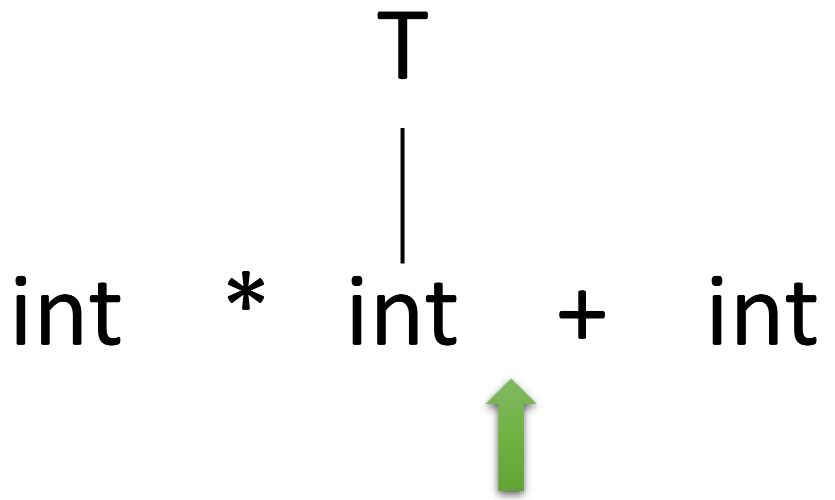
| int * int + int (Décalage)

int | * int + int (Décalage)

int * | int + int (Décalage)

int * int | + int (Réduction $T \to int$)

int * T | + int (Réduction $T \to int * T$)



Exemple

| int * int + int (Décalage)

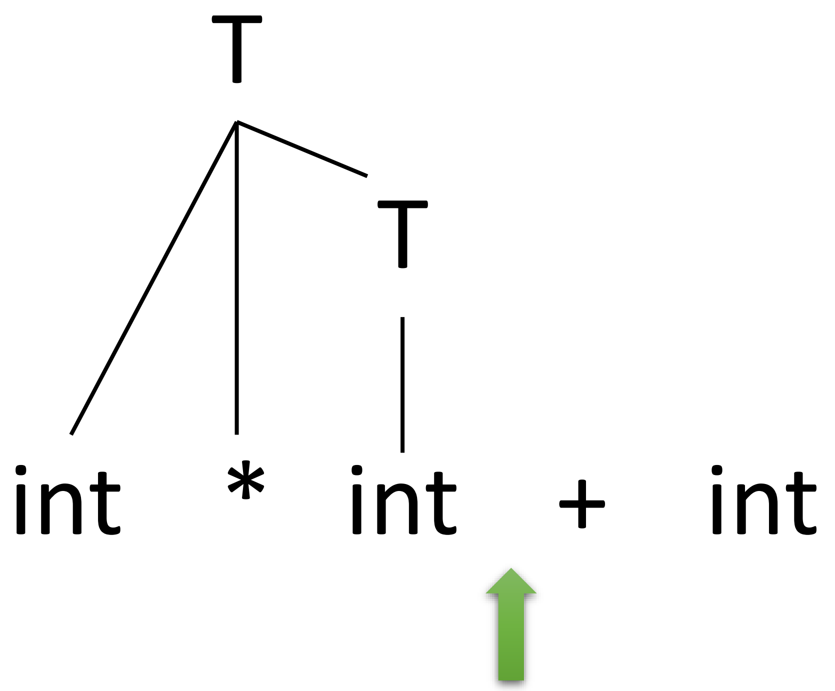
int | * int + int (Décalage)

int * | int + int (Décalage)

int * int | + int (Réduction T→int)

int * T | + int (Réduction T→int*T)

T | + int (Décalage)



Exemple

| int * int + int (Décalage)

int | * int + int (Décalage)

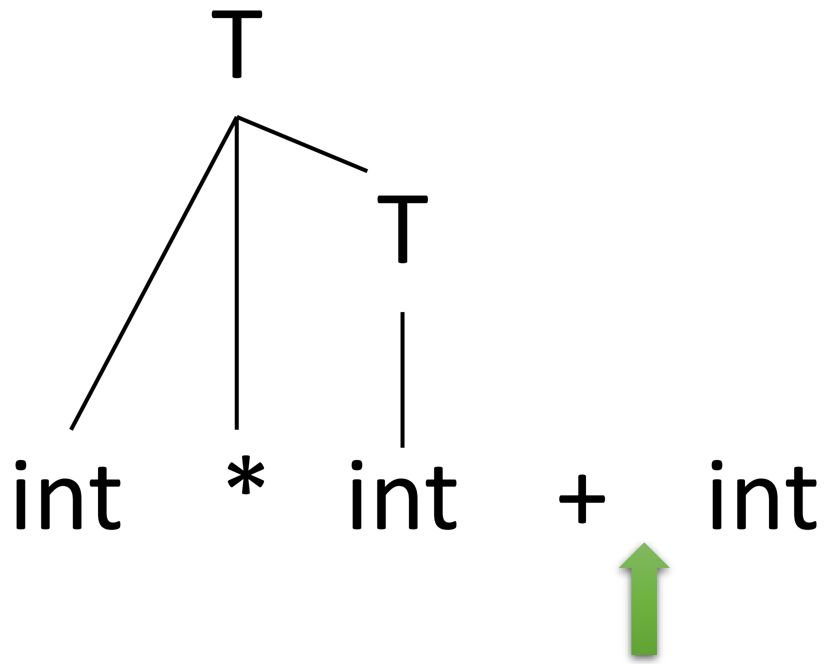
int * | int + int (Décalage)

int * int | + int (Réduction T→int)

int * T | + int (Réduction T→int*T)

T | + int (Décalage)

T + | int (Décalage)



Exemple

| int * int + int (Décalage)

int | * int + int (Décalage)

int * | int + int (Décalage)

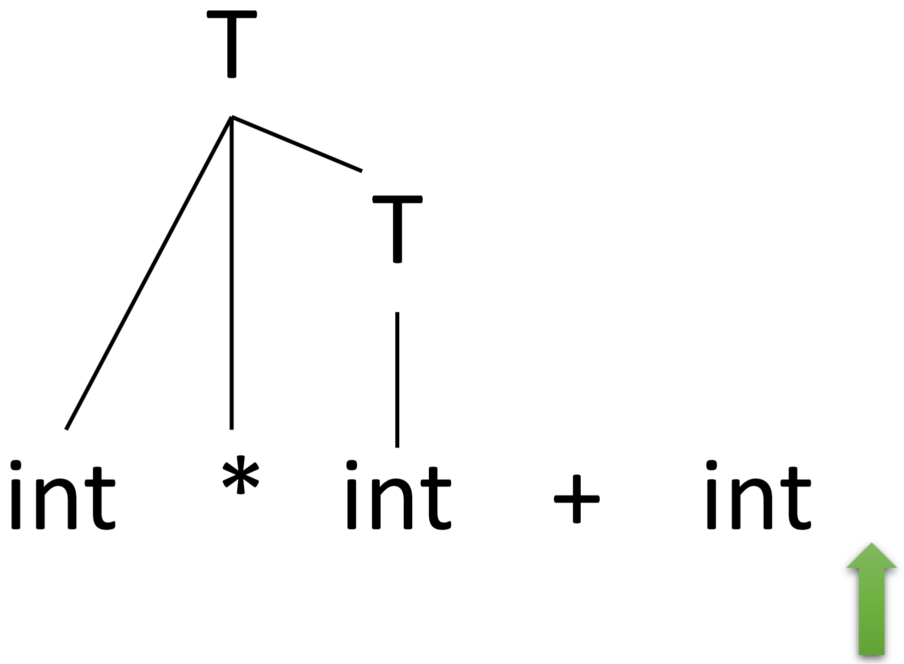
int * int | + int (Réduction $T \rightarrow int$)

int * T | + int (Réduction $T \rightarrow int * T$)

T | + int (Décalage)

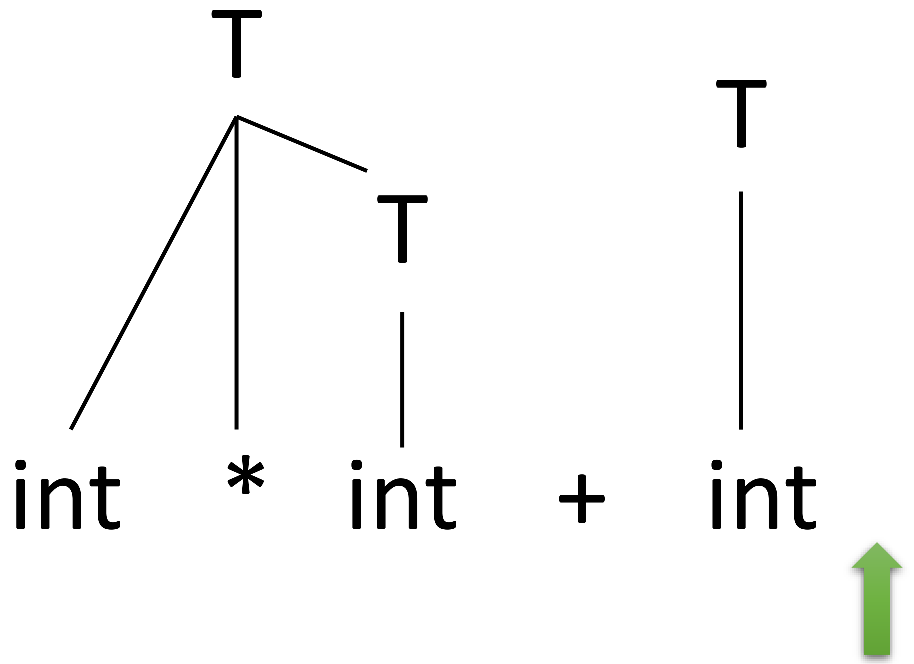
T + | int (Décalage)

T + int | (Réduction $T \rightarrow int$)



Exemple

| int * int + int	(Décalage)
int | * int + int	(Décalage)
int * | int + int	(Décalage)
int * int | + int	(Réduction T→int)
int * T | + int	(Réduction T→int*T)
T | + int	(Décalage)
T + | int	(Décalage)
T + int |	(Réduction T→int)
T + T |	(Réduction E→T)



Exemple

| int * int + int (Décalage)

int | * int + int (Décalage)

int * | int + int (Décalage)

int * int | + int (Réduction $T \rightarrow int$)

int * T | + int (Réduction $T \rightarrow int * T$)

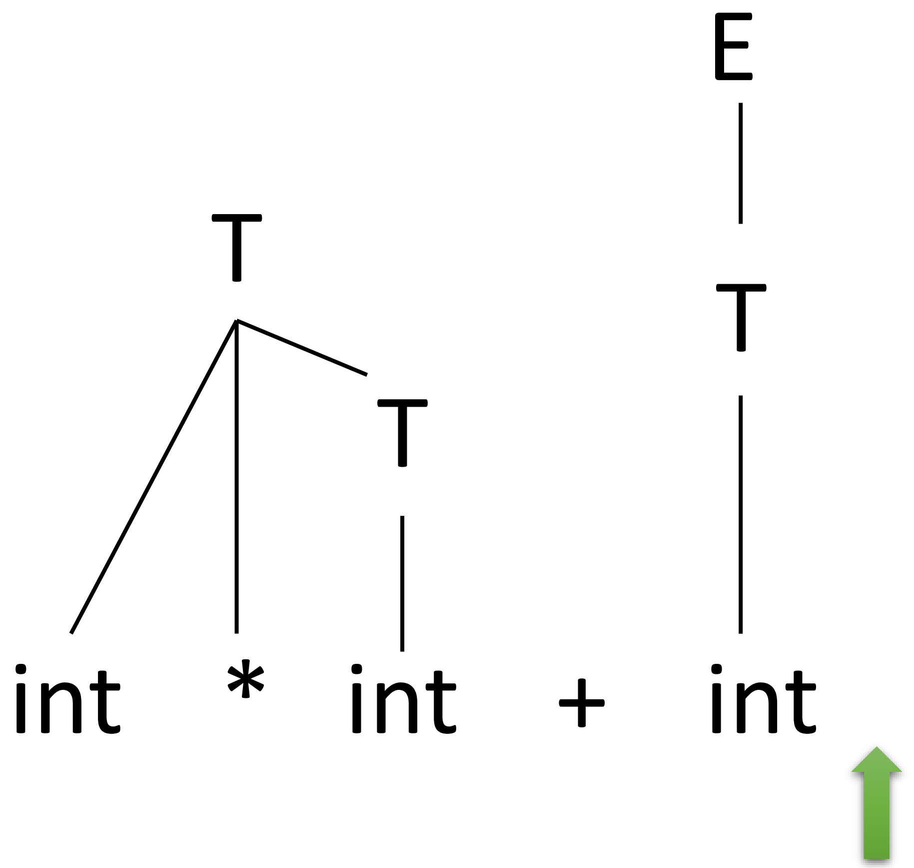
T | + int (Décalage)

T + | int (Décalage)

T + int | (Réduction $T \rightarrow int$)

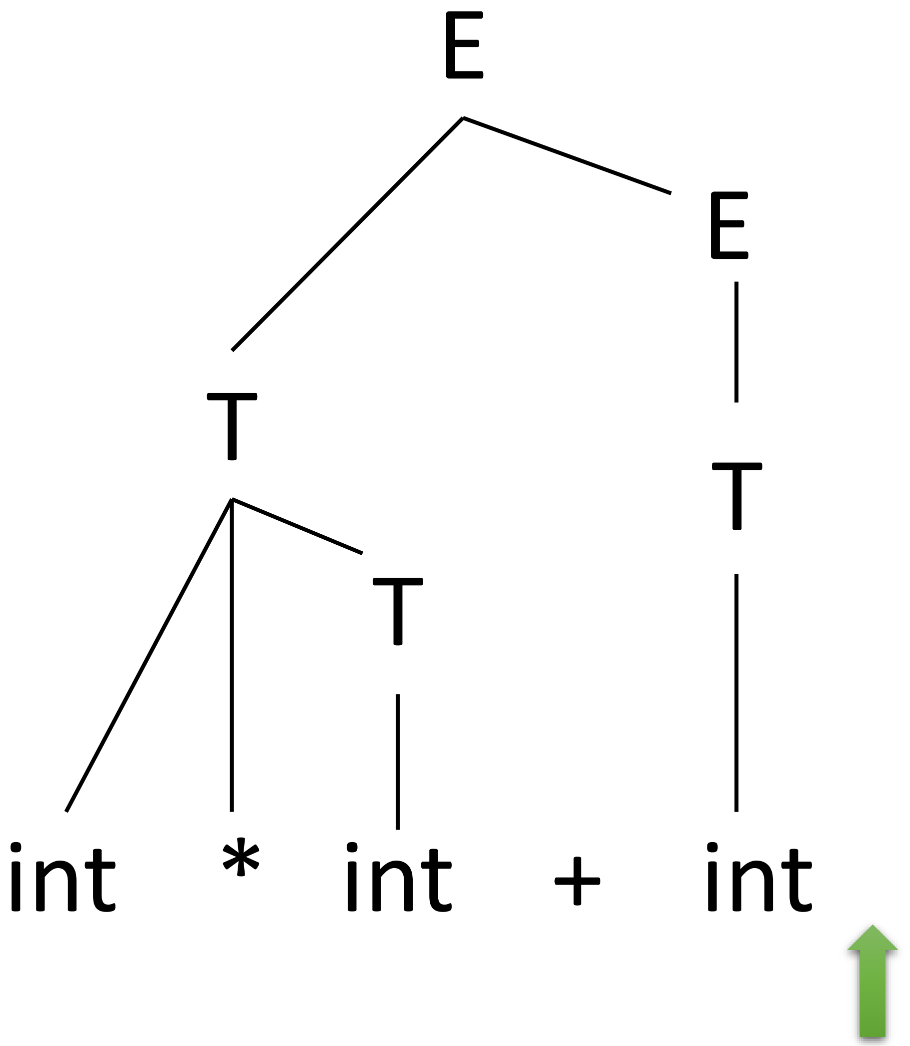
T + T | (Réduction $E \rightarrow T$)

T + E | (Réduction $E \rightarrow T+E$)



Exemple

| int * int + int	(Décalage)
int | * int + int	(Décalage)
int * | int + int	(Décalage)
int * int | + int	(Réduction T→int)
int * T | + int	(Réduction T→int*T)
T | + int	(Décalage)
T + | int	(Décalage)
T + int |	(Réduction T→int)
T + T |	(Réduction E→T)
T + E |	(Réduction E→T+E)
E |	(Mot accepté)



Analyseur décalage-réduction

- La sous chaîne gauche peut être implémentée par une pile

- Le sommet (la tête) de la pile est le |

- **Décalage** empile (push) un terminal sur la pile

- **Réduction**
  - Dépile (pop) les symboles de la pile (partie droite de production)
  - Empile un non-terminal sur la pile   (partie gauche de production)

Conflits d’analyseur décalage-réduction

- Dans un état donné, plus d'une action (Décalage ou Réduction) peut conduire à une analyse valide

- **Conflit Décalage/Réduction**:
  - S'il est légal de faire un **Décalage** ou **Réduction**

- **Conflit Réduction/Réduction**
  - S'il est légal de **réduire** de **deux productions différentes**

Comment décidons-nous quand décaler ou réduire ?

- Exemple de grammaire:
  - $\mathbf{E \to T + E \mid T}$
  - $\mathbf{T \to int * T \mid int}$

- Considérons l'étape int | * int + int
- On pourrait réduire de **T** → **int** en donnant T | *int + int

- **Une erreur fatale!**
  - Pas moyen d’arriver au symbole de départ E

Manches

- Intuition

On veut réduire seulement si on peut atteindre le non-terminal de départ avec les réductions suivantes

- Supposons la dérivation à droite suivante :

$$S \rightarrow^* \alpha X w \rightarrow \alpha\beta w$$

- Alors $\boldsymbol{\alpha\beta}$ est un **manche** de $\boldsymbol{\alpha\beta w}$

Manches

- Les manches formalisent l'intuition
  - Un manche est une réduction qui permet des réductions supplémentaires vers le symbole de départ

- Nous voulons seulement réduire au niveau **des manches**
  - Si un manche est dans la pile, on fait une réduction

- Nous avons définit le manche, mais

Comment déterminer les manches ?

Manches

- **Fait important n° 2**

Les manches apparaissent toujours au **sommet de la pille**, jamais au milieu de la pille.

- **Preuve:**
  - Exercice (preuve par récurrence)

Manches

- Dans l'analyse par réduction-décalage, les manches apparaissent toujours en haut de la pile.

- Les manches ne sont jamais à gauche du non terminal le plus à droite
  - Par conséquent, les actions Réduction / Décalage sont suffisants
  - le | il n'est jamais nécessaire de se déplacer vers la gauche

- Les algorithmes d'analyse ascendante sont basés sur la reconnaissance des manches.

Préfixes viables

- α est un préfixe viable si :
  - $\boldsymbol{\alpha|w}$ est un état de l'analyseur décalage-réduction

- $\boldsymbol{\alpha}$ est le contenu de la pille

- Une préfixe viable est une partie initiale d'un manche

- Tant que la pile de l'analyseur contient **un préfixe viable**, aucune **erreur syntaxique n'est détectée**

Manches

- **Fait important n° 3**

Les préfixes viables est un langage régulier, peut être reconnu par un automate fini .

Comment déterminer l'automate qui peut reconnaitre les préfixes viables ?

Analyseur LR(0)

- **L** : parcours de l’entré de gauche à droite (**Left to right**)
- **R** : Dérivation à droite inversée (**Rightmost derivation**)
- **0** : Aucun caractère de prédiction (la réduction ne considère pas l’entrée)

Les items LR(0)

- Un item est une production avec à « $\bullet$ » dans la partie droite.

- Les items de $A \to XYZ$ **sont :**
  - $A \to .XYZ$
  - $A \to X.YZ$
  - $A \to XY.Z$
  - $A \to XYZ.$

- **Exemple:**
  - Les items de $\mathrm{E} \to T + E$

Les items LR(0)

- Les items de $A \to \varepsilon$ **sont :**
  - $A \to .$
- **Exemple:**
  - Les items de $\mathrm{E} \to T + E$

Les items LR(0)

- Considérons la chaine (int)

$\mathbf{G:}\ \ E \rightarrow T + E \ \mid\ T$

$\qquad T \rightarrow int * T \ \mid\ int \mid (E)$

- **(E |)** est un état d'analyseur décalage-réduction

- **(E** est un préfixe de la partie droite de $\mathbf{T\rightarrow}$ **(E)**

- Item $\mathbf{T\rightarrow}$ **(E.)** signifie que nous avons déjà vu **(E** et on espère voir **)**

Les items LR(0)

- T → (.E)
- E → .T
- T → int *. T

- Signifie :
  - **T → (.E)** : nous avons vu **(** de **T → (.E)**
  - **E → .T** : nous avons vu $\boldsymbol{\varepsilon}$ de **E → .T**
  - **T → int *. T** : nous avons vu $\mathbf{int*}$ de **E → .T**

Construction d’un automate LR(0)

- Ajouter une production pour savoir quand l’analyse s’arrête

$$\boldsymbol{S' \rightarrow S}$$

- Définir les fonctions:
  - Fermeture(I)
  - Aller_à(I,X)

- **I:** ensemble d’items LR(0)
- X: un symbole de la grammaire

Fermeture(I)

- Pour définir les états de l'automate

- Initialiser la **Fermeture(I)** par les items de **I**
  - **Fermeture(I)=I**

- Si $\boldsymbol{A} \rightarrow \ \boldsymbol{\alpha}.\boldsymbol{B\beta}$ est dans **Fermeture(I)** et $\boldsymbol{B} \rightarrow \ \boldsymbol{\gamma}$ est une production
  - Ajouter l'item $\boldsymbol{B} \rightarrow .\boldsymbol{\gamma}$ dans **Fermeture(I)** si il n'est pas déjà

- Appliquer cette règle jusqu'à aucun item peut être ajouté

Algorithme de Fermeture

$J = I$;
repeat
    for (each item $A \rightarrow \alpha \cdot B\beta$ in J)
        for (each production $B \rightarrow \gamma$ of G)
            if ($B \rightarrow \cdot\gamma$ is not in J)
                add $B \rightarrow \cdot\gamma$ to J;
until no more items are added to J on one round;
return J;

Aller_à(I,X)

- Pour définir les transitions de l’automate

- Si $\boldsymbol{A} \rightarrow \ \boldsymbol{\alpha}.\boldsymbol{X\beta}$ dans **I** ajouter tous $\boldsymbol{A} \rightarrow \ \boldsymbol{\alpha X}.\boldsymbol{\beta}$ dans **Aller_à(I,X)**

- Calculer la fermeture à partir les items résultant de l’application de la règle précédente

Construction d’un automate LR(0)

- L’état initiale est déterminé par la fermeture de $\boldsymbol{S'} \to . \boldsymbol{S}$ **:**
  - I_0 = Fermeture({$\boldsymbol{S'} \to . \boldsymbol{S}$})

- Pour trouver l’état suivant avec une transition en symbole **X:**
  - I_j = Fermeture(Aller_à(I_i, X))
  - Ajouter l’état $\boldsymbol{I_j}$ comme un nouveau état de l’automate si il n’existe pas déjà

- Arrêter quand on ne peut pas ajouter des états

Algorithme de construction d'un automate LR(0)

$C = \{\text{CLOSURE}(\{[S' \to \cdot S]\})\};$

repeat

  for (each set of items I in C)

    for (each grammar symbol X)

      if ($\text{GOTO}(I, X)$ is not empty and not in C)

        add $\text{GOTO}(I, X)$ to C;

until no new sets of items are added to C on a round;

Analyse LR(0)

- **Supposons**
  - La pile contient: $\boldsymbol{\alpha}$
  - L'entrée suivante est: **t**
  - L'état de l'automate après la lecture de $\boldsymbol{\alpha}$ est: **s**

- **Réduction** avec $(\boldsymbol{A} \rightarrow \boldsymbol{\gamma})$ si
  - **s** contient l'item $(\boldsymbol{A} \rightarrow \boldsymbol{\gamma}.)$

- **Décalage** si
  - **s** contient l'item $(\mathbf{A} \rightarrow \boldsymbol{\beta}.\mathbf{tw})$
  - L'entrée suivante est: **t**

Conflits de l’analyse LR(0)

- Conflit **réduction/réduction :**
  - Un état avec deux items de réductions:
    - $\boldsymbol{X \rightarrow \beta.}$
    - $\boldsymbol{Y \rightarrow \alpha.}$
- Conflit **décalage/réduction**:
  - Un état avec un item de décalage et un autre de réduction
    - $\boldsymbol{X \rightarrow \beta.}$
    - $\boldsymbol{Y \rightarrow \omega.t\delta}$

Exemple

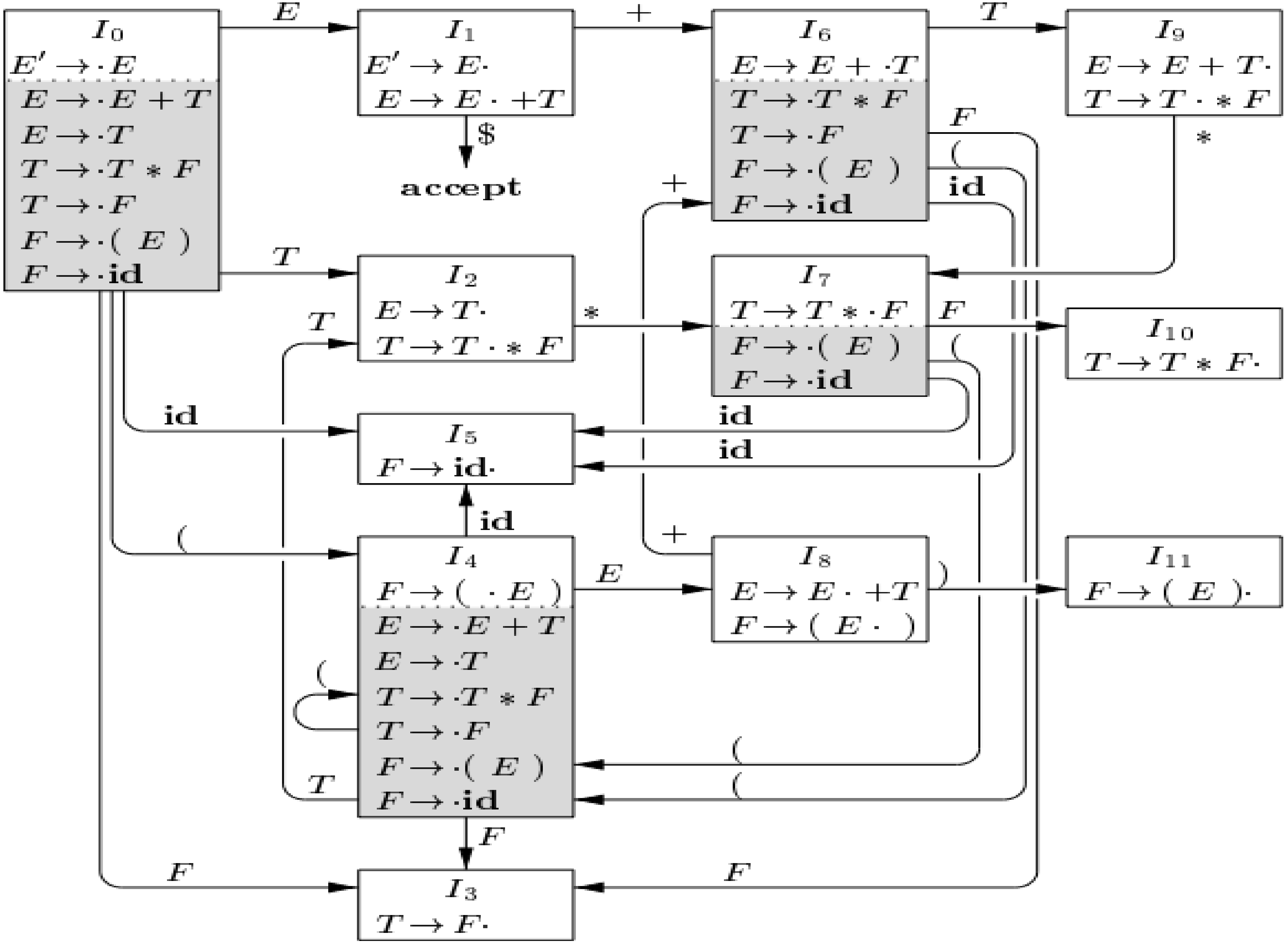
- Automate LR(0) de la grammaire :

G:

$$\mathbf{E \rightarrow E+T \mid T}$$

$$\mathbf{T \rightarrow T*F \mid F}$$

$$\mathbf{F \rightarrow (E) \mid int}$$



I0
E' → ·E
E → ·E + T
E → ·T
T → ·T * F
T → ·F
F → ·(E)
F → ·id
I1
E' → E·
E → E · +T
$
accept
I2
E → T·
T → T · * F
I3
T → F·
I4
F → (· E)
E → ·E + T
E → ·T
T → ·T * F
T → ·F
F → ·(E)
F → ·id
I5
F → id·
I6
E → E + ·T
T → ·T * F
T → ·F
F → ·(E)
F → ·id
I7
T → T * ·F
F → ·(E)
F → ·id
I8
E → E · +T
F → (E ·)
I9
E → E + T·
T → T · * F
I10
T → T * F·
I11
F → (E)·
E
T
F
id
(
+
*
)

Analyseur SLR(1)

- SLR : Simple LR

- Améliore LR(0) en considérant un caractère de prédiction

- Moins de de conflits **décalage/réduction**

Analyse LR(0)

- **Supposons**
  - La pile contient: $\boldsymbol{\alpha}$
  - L’entrée suivante est: **t**
  - L’état de l’automate après la lecture de $\boldsymbol{\alpha}$ est: **s**

- **Réduction** avec $(\boldsymbol{A} \rightarrow \boldsymbol{\gamma})$ si
  - **s** contient l’item $(\boldsymbol{A} \rightarrow \boldsymbol{\gamma}.)$

- **Décalage** si
  - **s** contient l’item $(\mathbf{A} \rightarrow \boldsymbol{\beta}.\mathbf{tw})$
  - L’entrée suivante est: **t**

Analyse SLR(1)

- **Supposons**
  - La pile contient: $\boldsymbol{\alpha}$
  - L'entrée suivante est: **t**
  - L'état de l'automate après la lecture de $\boldsymbol{\alpha}$ est: **s**

- **Réduction** avec $(\boldsymbol{A} \rightarrow \boldsymbol{\gamma})$ si
  - **s** contient l'item $(\boldsymbol{A} \rightarrow \boldsymbol{\gamma}.)$
  - **t** ∈ **Suivant(A)**
- **Décalage** si
  - **s** contient l'item (**A→ β.tw**)
  - L'entrée suivante est: **t**

Table d’analyse SLR(1)

Action | **Aller_à**

Terminaux | Non terminaux

Etats	a_0	…	a_n	$	A_0	…	A_n
0							
1							
…							
m							

Décaler j

Vide

Réduire (A→ α)

Table d'analyse SLR(1)

- Construction de l'automate **LR(0)** à avec les états $\mathbf{C} = \{\mathbf{I_0}, \mathbf{I_1}, \dots, \mathbf{I_n}\}$

- L'état initiale de l'analyseur est l'état $\mathbf{I_0}$ qui contient $[\boldsymbol{S'} \rightarrow \boldsymbol{.S}]$

- Si $[\mathbf{A} \rightarrow \boldsymbol{\alpha}.\mathbf{a}\boldsymbol{\beta}] \in \boldsymbol{I_i}$ et **Aller_à(**$\boldsymbol{I_i}, \boldsymbol{a}$**)=** $\boldsymbol{I_j}$
  - Action[i,a] = Décaler j

- Si $[\mathbf{A} \rightarrow \boldsymbol{\alpha}.] \in \boldsymbol{I_i}$
  - Pour tous **a** $\in \boldsymbol{Suivant}(\boldsymbol{A})$:
  - Action[i,a]= Réduire ($\mathbf{A} \rightarrow \boldsymbol{\alpha}$)

- Si $[\boldsymbol{S'} \rightarrow \mathbf{S}.] \in \boldsymbol{I_i}$
  - Action[i,$]= Accepter

- Si **Aller_à(**$\boldsymbol{I_i}, \boldsymbol{A}$**)=** $\boldsymbol{I_j}$
  - Aller_à[i,A]=j

- Les cases vides sont des erreurs syntaxiques