Compilation (#2,3) : Analyse syntaxique

C. Deleuze & L. Gonnord

Grenoble INP/Esisar

2022-2023

Plan

Définition informelle de la syntaxe

une instruction est définie par :

1. si *id* est un identificateur et *exp* une expression alors

   id := *exp* est une instruction

2. si *exp* est une expression et *inst* une instruction alors

   si *exp* alors *inst*
   tant que *exp* faire *inst* | sont des instructions

Définition informelle de la syntaxe

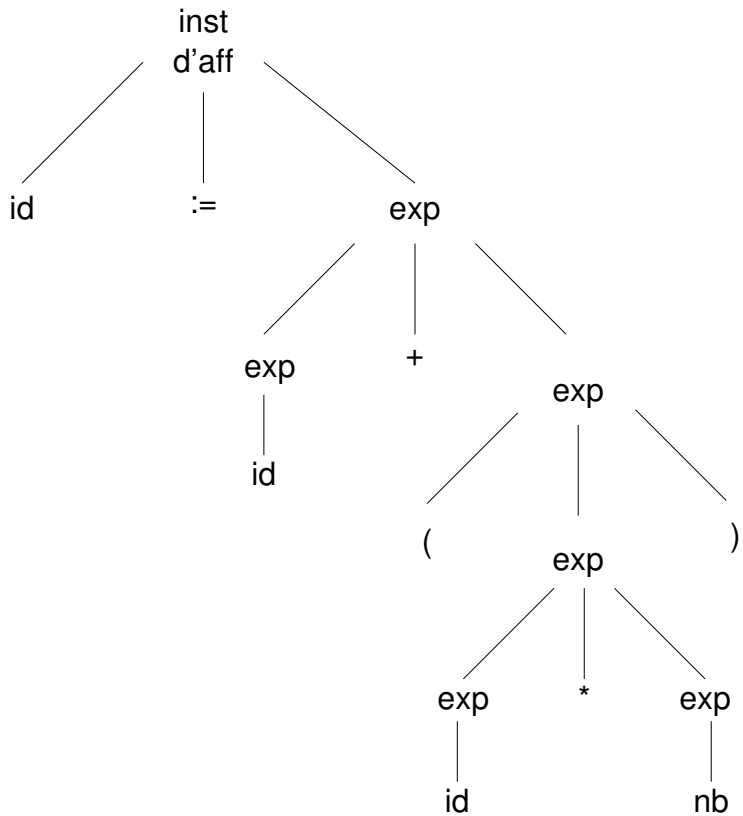
une instruction est définie par :

1. si *id* est un identificateur et *exp* une expression alors
   id := *exp* est une instruction
2. si *exp* est une expression et *inst* une instruction alors
   si *exp* alors *inst*
   tant que *exp* faire *inst*
   | sont des instructions

une expression est définie par :

3. tout identificateur est une expression
4. tout nombre est une expression
5. si *exp1* et *exp2* sont des expressions alors
   exp1 + *exp2*
   exp1 * *exp2*
   (*exp1*)
   | sont des expressions

Arbre syntaxique



Grammaires

La syntaxe des langages de programmation est définie à l'aide d'un formalisme appelé grammaire non contextuelle (ou simplement grammaire). Une grammaire comprend quatre composants :

1. un ensemble d'unités lexicales, appelées symboles terminaux,
2. un ensemble de symboles non terminaux,
3. un ensemble de productions,

   non terminal $\rightarrow$ suite de terminaux et non terminaux

   partie gauche — *partie droite* (éventuellement ϵ)
4. un non terminal est désigné comme **axiome**.

Voici la grammaire décrite informellement ci-dessus.

I $\rightarrow$ **id** := E

I $\rightarrow$ **si** E **alors** I

I $\rightarrow$ **tant que** E **faire** I

E $\rightarrow$ **id**

E $\rightarrow$ **nb**

E $\rightarrow$ E + E

E $\rightarrow$ E * E

E $\rightarrow$ (E)

Quelques définitions

- Une production **définit** un non terminal s'il apparaît en partie gauche de la production.
- Une **chaîne** d'unités lexicales est une suite éventuellement vide d'unités lexicales (notée ϵ si vide).
- Une grammaire **dérive** des chaînes en commençant par l'axiome et en remplaçant, de manière répétée, un non terminal par la partie droite d'une des productions le définissant.
- Les chaînes d'unités lexicales que l'on peut dériver ainsi constituent le **langage défini** par la grammaire.

Dérivation

Arbre d'analyse

C'est la représentation graphique d'une dérivation dans laquelle les choix concernant l'ordre de dérivation ont disparu.

- chaque nœud intérieur est étiqueté par un non terminal,
- ses fils sont étiquetés par les symboles de la partie droite de la production utilisée,
- les feuilles forment la chaîne des symboles terminaux (de gauche à droite).

Arbre d'analyse

C'est la représentation graphique d'une dérivation dans laquelle les choix concernant l'ordre de dérivation ont disparu.

- chaque nœud intérieur est étiqueté par un non terminal,
- ses fils sont étiquetés par les symboles de la partie droite de la production utilisée,
- les feuilles forment la chaîne des symboles terminaux (de gauche à droite).

aka arbre syntaxique !

Ambiguïtés

Non aux ambiguïtés !

Grammaire non ambigüe :

à chaque texte de programme correspond :

Non aux ambiguïtés !

Grammaire non ambigüe :

à chaque texte de programme correspond :

- soit un unique arbre de dérivation et le programme est (syntaxiquement) correct,
- soit aucun et le pgm contient au moins une erreur syntaxique.

Non aux ambiguïtés !

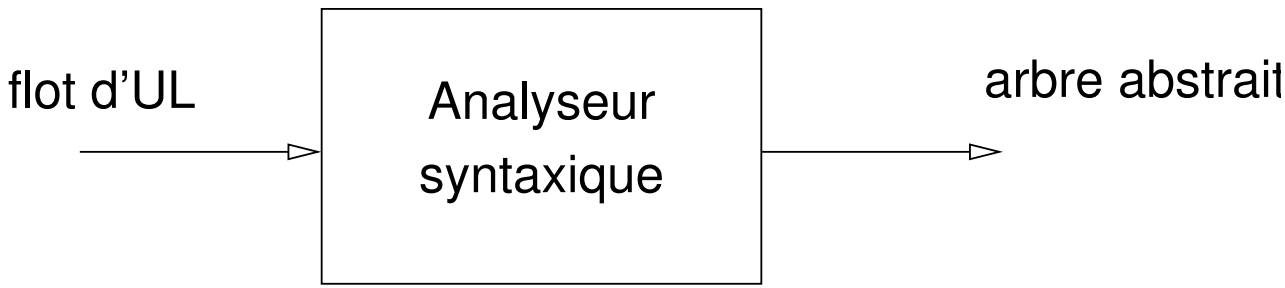
Grammaire non ambigüe :
à chaque texte de programme correspond :

- soit un unique arbre de dérivation et le programme est (syntaxiquement) correct,
- soit aucun et le pgm contient au moins une erreur syntaxique.

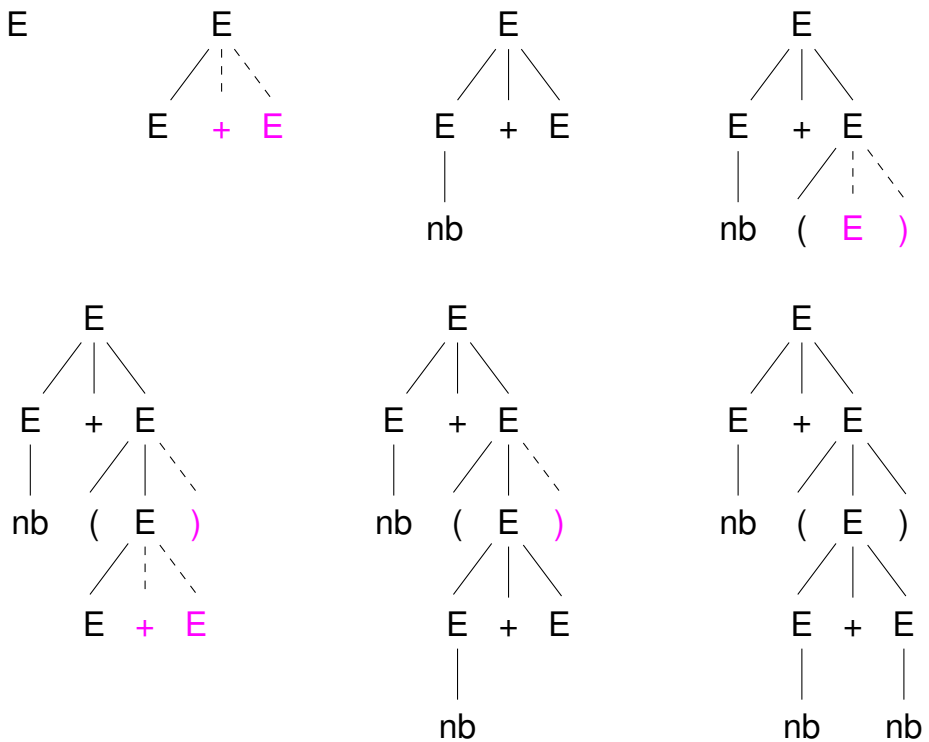
Deux façons d'éliminer les ambiguïtés :

- modifier la grammaire pour la rendre non ambiguë,
- lever les ambiguïtés avec des règles externes à la grammaire.

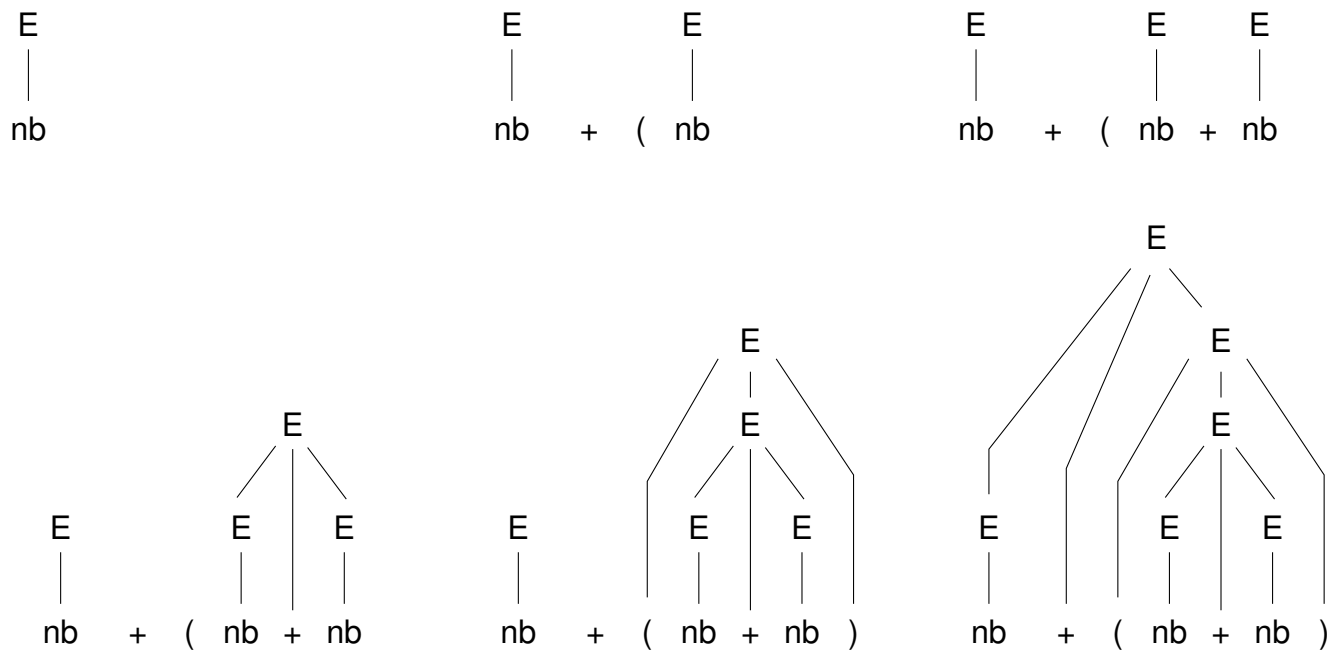
Analyseur syntaxique



Analyse descendante



Analyse ascendante



expression $\rightarrow$ terme reste_expression	E $\rightarrow$ T R
terme $\rightarrow$ expression_parenthésée | **id**	T $\rightarrow$ P | id
expression_parenthésée $\rightarrow$ '(' expression ')'	P $\rightarrow$ (E)
reste_expression $\rightarrow$ '+' expression | ε	R $\rightarrow$ + E | ε

Figure: Grammaire pour montrer l'analyse descendante

1. $\forall t$ terminal, PREM$(t) = \{t\}$
2. $\forall A$ non-terminal, PREM$(A) = \emptyset$
3. Pour les non terminaux, appliquer les règles suivantes jusqu'à ce qu'on ne puisse plus rien ajouter (A est un non terminal, X_i un symbole quelconque) :
   1. si $A \rightarrow X_1 X_2 ... X_k$ avec $k \geqslant 1$
      - ajouter PREM$(X_1)\backslash\{\varepsilon\}$ à PREM(A)
      - si $\varepsilon \in$ PREM(X_1), ajouter PREM$(X_2)\backslash\{\varepsilon\}$
      - si en plus, $\varepsilon \in$ PREM(X_2), ajouter PREM$(X_3)\backslash\{\varepsilon\}$
      - ...
      - si $\varepsilon \in$ PREM(X_i) $\forall i$, ajouter ε à PREM(A).
   2. si $A \rightarrow \varepsilon$ ajouter ε à PREM(A).

Figure: Calcul des PREMs des symboles

pour une chaîne $\alpha = X_1 X_2 \dots X_n$, avec X_i un symbole quelconque

1. initialiser PREM(α) à $\emptyset$
2. ajouter PREM$(X_1)\backslash\{\varepsilon\}$
3. si $\varepsilon \in$ PREM(X_1), ajouter PREM$(X_2)\backslash\{\varepsilon\}$
4. si en plus $\varepsilon \in$ PREM(X_2), ajouter PREM$(X_3)\backslash\{\varepsilon\}$
5. ...
6. si $\varepsilon \in$ PREM(X_i) $\forall i$, ajouter ε

Figure: Calcul des PREMs des chaînes

Initialiser le SUIV de l'axiome à {\$} et les autres SUIV à $\emptyset$ puis appliquer les règles suivantes jusqu'à ce qu'on ne puisse plus rien ajouter :

1. pour chaque production de la forme $A \to \alpha B \beta$
   - ajouter PREM$(\beta)\backslash\{\varepsilon\}$ à SUIV(B)
   - si $\varepsilon \in$ PREM(β), ajouter SUIV(A) à SUIV(B)
2. pour chaque production de la forme $A \to \alpha B$
   - ajouter SUIV(A) à SUIV(B).

Figure: Calcul des SUIVs des non terminaux

Non terminal	PREM	SUIV
expression	{ id '(' }	{ $ ')' }
terme	{ id '(' }	{ '+' $ ')' }
expression_parenthésée	{ '(' }	{ '+' $ ')' }
reste_expression	{ '+' ε }	{ $ ')' }

Figure: Ensembles PREM et SUIV de notre grammaire

```
#include "ana_lex.h"
#include "ap.h"

void parse(void) {
  expression();
  consomme_ul(FDF);
}

void expression(void) {
  switch(prochaine_ul()) {
  case ID:
  case '(':  terme(); reste_expression(); break;
  default:   erreur();
  }
}
```

```
void terme(void) {
  switch(prochaine_ul()) {
  case '(':  expression_parenthesee(); break;
  case ID:   consomme_ul(ID); break;
  default:   erreur();
  }
}

void expression_parenthesee(void) {
  switch(prochaine_ul()) {
  case '(':  consomme_ul('('); expression(); consomme_ul(')'); br
  default:   erreur();
  }
}
```

```
void reste_expression(void) {
  switch(prochaine_ul()) {
  case '+':  consomme_ul('+'); expression(); break;
  case FDF:
  case ')':  break;
  default:   erreur();
  }
}
```

Figure: Analyseur prédictif

```
void terme(void) {
    switch(prochaine_ul()) {
    case '(': expression_parenthesee(); break;
    case ID:  consomme_ul(ID); break;
    case ID:  element_indexe(); break;
    default:  erreur();
    }
}
```

Figure: Code erroné pour l'analyse d'un terme avec `élément_indexé`

```
void S(void) {
    switch(prochaine_ul()) {
    case 'a': A(); consomme_ul('a'); consomme_ul('b'); break;
    default:  erreur();
    }
}

void A(void) {
    switch(prochaine_ul()) {
    case 'a': consomme_ul('a'); break;
    case 'a': break;
    default:  erreur();
    }
}
```

Figure: Code erroné avec un conflit PREM-SUIV

Conditions pour être LL(1)

- pas de conflit PREM-PREM : $\forall$ N, PREM de tous les choix sont distincts,
- pas de conflit PREM-SUIV : $\forall$ N nullifiable, SUIV(N) distinct des PREM des autres choix,
- pas de choix nullifiables multiples.

3 Analyse descendante

- Analyseur prédictif
- Conflits LL(1)
- Analyseur prédictif itératif

non terminal	terminal en entrée				
	id	+	(	)	$
expression	terme rest_exp		terme rest_exp		
terme	id		exp_par		
exp_par			(expression)		
rest_exp		+ expression		ε	ε

Figure: Table de transition de l'analyseur prédictif itératif

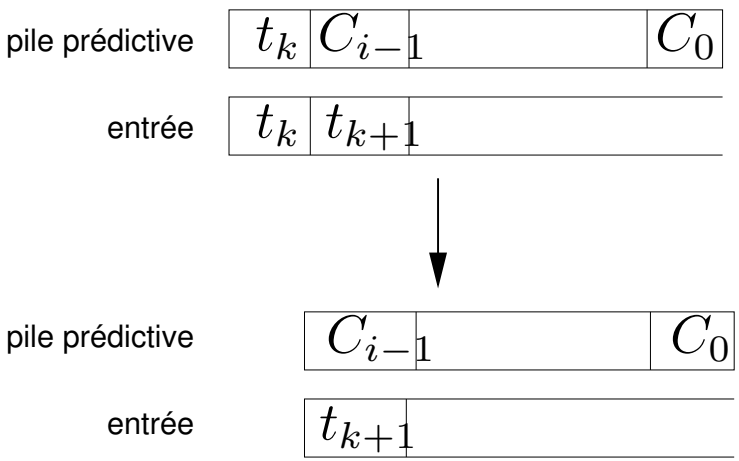
Mouvements de l'analyseur

reconnaissance Un terminal est en sommet de pile. Il doit être égal au terminal en entrée. On dépile et avance (ou erreur).

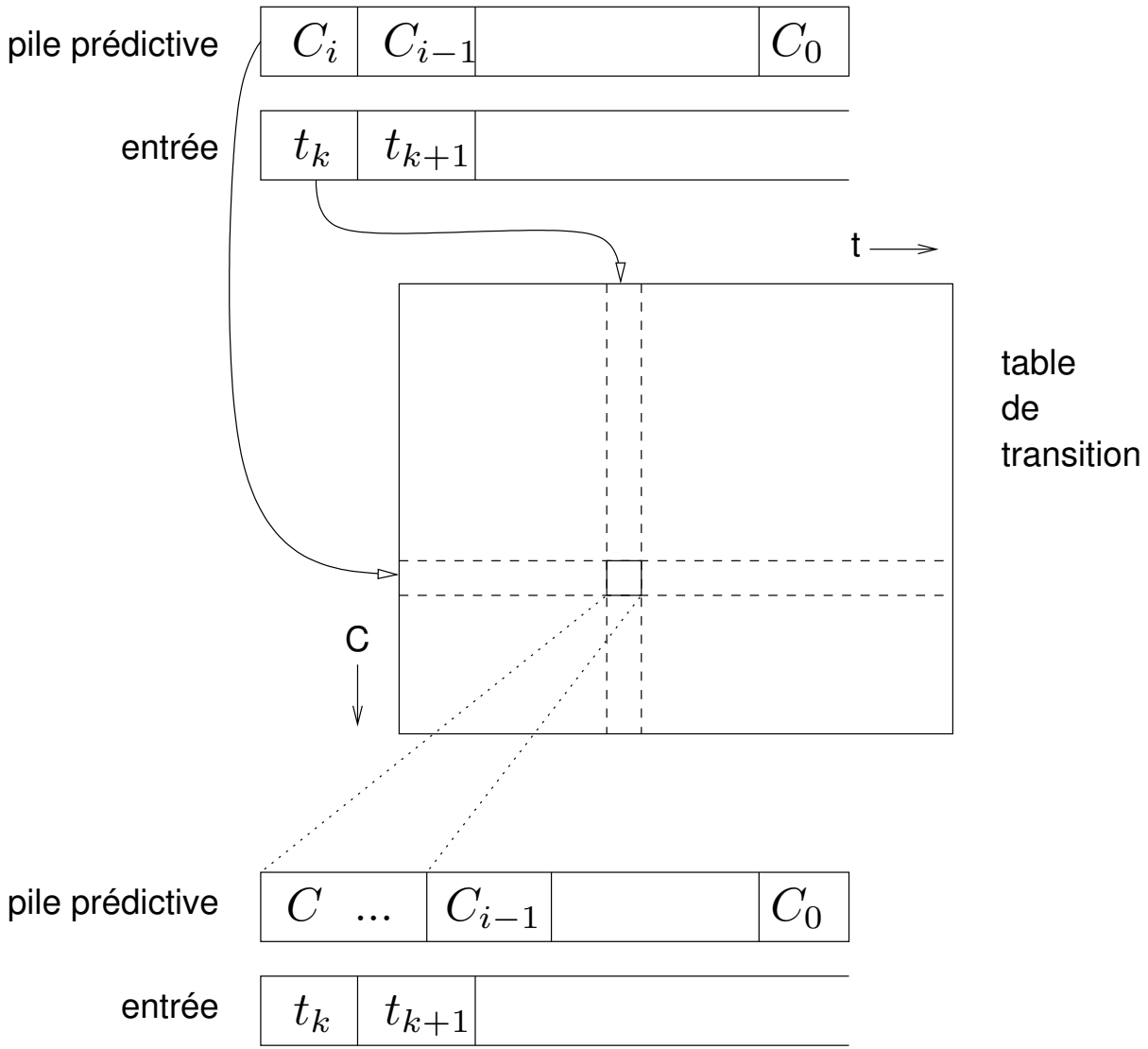
prédiction Un non-terminal est en sommet de pile. Dépile, consulte table et empile les nouveaux symboles (ou erreur).

terminaison La pile prédictive est vide.

Mouvement de reconnaissance



Mouvement de prédiction



```
importer Entrée [1..]  // depuis l'analyseur lexical
Indice_entrée ← 1
Pile_prédictive ← Pile_vide
empiler FDF puis axiome sur Pile_prédictive
tantque Pile_prédictive ≠ Pile_vide faire
    prédiction ← dépiler(Pile_prédictive)

    si prédiction est un terminal alors
        // essayer mouvement de reconnaissance
        si prédiction = Entrée[Indice_entrée] alors
                Indice_entrée ← Indice_entrée + 1
        sinon
                erreur "UL attendue non trouvée : ", prédiction
    sinon
        // essayer mouvement de prédiction
        prédiction ← table_prédictive[prédiction, Entrée[Indice_entrée]]
        si prédiction = vide alors
                erreur "UL inattendue : ", Entrée[Indice_entrée]
        sinon
                empiler les symboles de prédiction sur Pile_prédictive
```

Figure: Code de l'analyseur prédictif itératif

Exercice : analyse de `id + (id + id)`

NT	terminal				
	id	+	(	)	$
E	T R		T R		
T	id		P		
P			(E)		
R		+ E		ε	ε

Figure: Table de transition de l'analyseur prédictif itératif

Postparation

1. Ambiguïtés : montrez deux dérivations de la chaîne `id := id + id * nb` qui correspondent à des arbres syntaxiques différents (grammaire du transparent 7).
2. Exécutez à la main l'analyseur prédictif récursif sur la chaîne `id + id`, puis sur la chaîne `id + (id + id)`.
3. Soit une grammaire contenant la production $E \rightarrow E + E$. Expliquez pourquoi elle n'est pas LL(1).
4. Terminez l'analyse prédictive itérative du transparent 37,
   - et "retrouvez" la dérivation correpondante dans l'analyse.

Préparation

Lisez l'interview du créateur de Yacc, dans le document "The A-Z of Programming Languages" (document sur chamilo)

4 Analyse ascendante

- Principes
- Analyse par décalage-réduction
- Analyseurs LR
- Grammaires LR
- Construction des tables d'analyse SLR

Pivot

Le but est de trouver le nœud pas encore construit le plus à gauche dont tous les fils ont été construits. Cette suite de fils constitue le *pivot*. Créer le nœud N du parent et le relier aux fils (le pivot), c'est *réduire* le pivot vers N.

Définition

On appelle *pivot* (ou *manche* ou *poignée* – *handle* en anglais) une suite β de symboles terminaux et non terminaux qui correspond à la partie droite d'une production et dont la réduction vers le non terminal de la partie gauche de cette production représente une étape le long de la dérivation droite inverse.

Réduction du pivot

Soit une chaîne de terminaux w, appartenant au langage.
Il existe une dérivation droite (inconnue) :

$$S = \gamma_0 \underset{d}{\rightarrow} \gamma_1 \underset{d}{\rightarrow} \gamma_2 \quad \cdots \quad \underset{d}{\rightarrow} \gamma_{n-1} \underset{d}{\rightarrow} \gamma_n = w$$

Repérer le pivot β_n dans γ_n et le remplacer par le A de la production
A $\rightarrow \beta_n$ donne γ_{n-1}...
repeat

Actions de l'analyseur

décaler prochain terminal de l'entrée est retiré et empilé.

réduire le pivot est en sommet de pile. Il est remplacé par la partie gauche de la production choisie.

accepter la pile contient uniquement l'axiome et l'entrée est vide.

erreur

1,2	expression $\rightarrow$ terme | expression '+' terme	E $\rightarrow$ T | E+T
3,4	terme $\rightarrow$ **id** | '(' expression ')'	T $\rightarrow$ id | (E)

Figure: Une grammaire récursive à gauche

Pile	Entrée	Action
	(a+b)+c	

1,2 $E \rightarrow T \mid E+T$

3,4 $T \rightarrow id \mid (E)$

Pile	Entrée	Action
	(a+b)+c	D

1,2 $E \rightarrow T \mid E+T$

3,4 $T \rightarrow id \mid (E)$

Pile	Entrée	Action
	(a+b)+c	D
(	a+b)+c	

1,2 $E \rightarrow T \mid E+T$

3,4 $T \rightarrow id \mid (E)$

Pile	Entrée	Action
	(a+b)+c	D
(	a+b)+c	D

1,2 $E \rightarrow T \mid E+T$

3,4 $T \rightarrow id \mid (E)$

Pile	Entrée	Action
	(a+b)+c	D
(	a+b)+c	D
(a	+b)+c	

1,2 $E \rightarrow T \mid E+T$

3,4 $T \rightarrow id \mid (E)$

Pile	Entrée	Action
	(a+b)+c	D
(	a+b)+c	D
(a	+b)+c	R3

1,2 $E \rightarrow T \mid E+T$

3,4 $T \rightarrow id \mid (E)$

Pile	Entrée	Action
	(a+b)+c	D
(	a+b)+c	D
(a	+b)+c	R3
(T	+b)+c	

1,2 $E \rightarrow T \mid E+T$

3,4 $T \rightarrow id \mid (E)$

Pile	Entrée	Action
	(a+b)+c	D
(	a+b)+c	D
(a	+b)+c	R3
(T	+b)+c	R1

1,2 $E \rightarrow T \mid E+T$

3,4 $T \rightarrow id \mid (E)$

Pile	Entrée	Action
	(a+b)+c	D
(	a+b)+c	D
(a	+b)+c	R3
(T	+b)+c	R1
(E	+b)+c	

1,2 $E \rightarrow T \mid E+T$

3,4 $T \rightarrow id \mid (E)$

Pile	Entrée	Action
	(a+b)+c	D
(	a+b)+c	D
(a	+b)+c	R3
(T	+b)+c	R1
(E	+b)+c	D

1,2 $E \rightarrow T \mid E+T$

3,4 $T \rightarrow id \mid (E)$

Pile	Entrée	Action
	(a+b)+c	D
(	a+b)+c	D
(a	+b)+c	R3
(T	+b)+c	R1
(E	+b)+c	D
(E+	b)+c	

1,2 $E \rightarrow T \mid E+T$

3,4 $T \rightarrow id \mid (E)$

Pile	Entrée	Action
	(a+b)+c	D
(	a+b)+c	D
(a	+b)+c	R3
(T	+b)+c	R1
(E	+b)+c	D
(E+	b)+c	D

1,2 $E \rightarrow T \mid E+T$

3,4 $T \rightarrow id \mid (E)$

1,2 $E \rightarrow T \mid E+T$

3,4 $T \rightarrow id \mid (E)$

Pile	Entrée	Action
	(a+b)+c	D
(	a+b)+c	D
(a	+b)+c	R3
(T	+b)+c	R1
(E	+b)+c	D
(E+	b)+c	D
(E+b	)+c	

1,2 $E \rightarrow T \mid E+T$

3,4 $T \rightarrow id \mid (E)$

Pile	Entrée	Action
	(a+b)+c	D
(	a+b)+c	D
(a	+b)+c	R3
(T	+b)+c	R1
(E	+b)+c	D
(E+	b)+c	D
(E+b	)+c	R3

1,2 $E \rightarrow T \mid E+T$

3,4 $T \rightarrow id \mid (E)$

Pile	Entrée	Action
	(a+b)+c	D
(	a+b)+c	D
(a	+b)+c	R3
(T	+b)+c	R1
(E	+b)+c	D
(E+	b)+c	D
(E+b	)+c	R3
(E+T	)+c	

1,2 $E \rightarrow T \mid E+T$

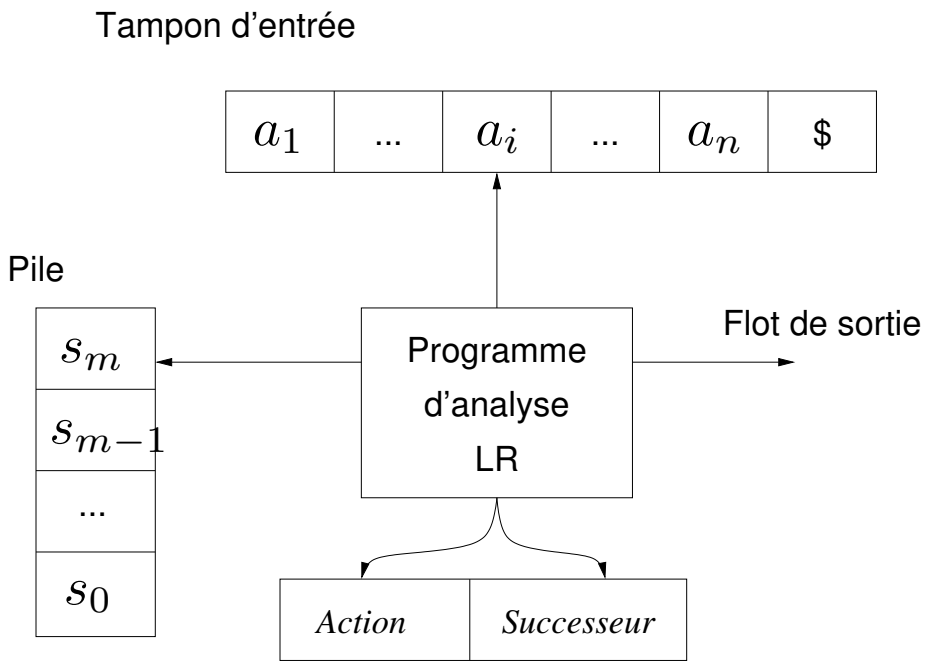
3,4 $T \rightarrow id \mid (E)$

Pile	Entrée	Action
	(a+b)+c	D
(	a+b)+c	D
(a	+b)+c	R3
(T	+b)+c	R1
(E	+b)+c	D
(E+	b)+c	D
(E+b	)+c	R3
(E+T	)+c	R2

Figure: Analyse de (a+b)+c

LR(k)
méthode décalage-réduction sans rebroussement *(backtracking)* la plus générale connue.
le but est de **repérer** le pivot
inconvénient : trop de travail pour être mise en œuvre manuellement sur la grammaire d'un vrai langage de programmation

Modèle d'un analyseur LR



Actions

configuration de l'analyseur

$$(s_0\ s_1\ \ldots\ s_m\ ,\ a_i\ a_{i+1}\ \ldots\ a_n\ \$)$$

Action[s_m, a_i] = décaler s

$$(s_0\ s_1\ \ldots\ s_m\ s\ ,\ a_{i+1}\ \ldots\ a_n\ \$)$$

Action[s_m, a_i] = réduire par A $\rightarrow\beta$

$$(s_0\ s_1\ \ldots\ s_{m-r}\ s,\ a_i\ a_{i+1}\ \ldots\ a_n\ \$)$$

- r = | β |
- s = Successeur[s_{m-r}, A]

Algorithme LR

```
soit a le premier symbole de l'entrée
empiler s0
accepter ← faux
erreur ← faux
répéter
    soit s l'état en sommet de pile
    si Action[s,a] = décaler s' alors
        empiler s'
        soit a le symbole d'entrée suivant
    sinon si Action[s,a] = réduire par A → β alors
        dépiler |β| états
        soit s' l'état en sommet de pile
        empiler Successeur[s',A]
    sinon si Action[s,a] = accepter alors accepter ← vrai
    sinon erreur ← vrai
jusqu'à erreur ou accepter
```

Une grammaire

1 $E \rightarrow E + T$
2 $E \rightarrow T$
3 $T \rightarrow T * F$
4 $T \rightarrow F$
5 $F \rightarrow (E)$
6 $F \rightarrow \mathbf{id}$

Table d'analyse pour la grammaire

État	Action						Successeur		
	id	+	*	(	)	$	E	T	F
0	d5			d4			1	2	3
1		d6				acc			
2		r2	d7		r2	r2			
3		r4	r4		r4	r4			
4	d5			d4			8	2	3
5		r6	r6		r6	r6			
6	d5			d4				9	3
7	d5			d4					10
8		d6			d11				
9		r1	d7		r1	r1			
10		r3	d3		r3	r3			
11		r5	r5		r5	r5			

Table d'analyse pour la grammaire

1	$E \rightarrow E + T$
2	$E \rightarrow T$
3	$T \rightarrow T * F$
4	$T \rightarrow F$
5	$F \rightarrow (E)$
6	$F \rightarrow \mathbf{id}$

État	Action						Successeur		
	id	+	*	(	)	$	E	T	F
0	d5			d4			1	2	3
1		d6				acc			
2		r2	d7		r2	r2			
3		r4	r4		r4	r4			
4	d5			d4			8	2	3
5		r6	r6		r6	r6			
6	d5			d4				9	3
7	d5			d4					10
8		d6			d11				
9		r1	d7		r1	r1			
10		r3	d3		r3	r3			
11		r5	r5		r5	r5			

Pile états	symboles (pour info)	Entrée	Action
0		id*id+id $	d5
0 5	id	*id+id $	r6
0 3	F	*id+id $	r4
0 2	T	*id+id $	d7
0 2 7	T*	id+id $	d5
0 2 7 5	T*id	+id $	r6
0 2 7 10	T*F	+id $	r3
0 2	T	+id $	r2
0 1	E	+id $	d6
0 1 6	E +	id $	d5
0 1 6 5	E + id	$	r6
0 1 6 3	E + F	$	r4
0 1 6 9	E + T	$	r1
0 1	E	$	acc

Figure: Analyse de id*id+id

Grammaire LR

- Une grammaire LR est une grammaire pour laquelle il est possible de construire la table d'analyse.
- Pour qu'une grammaire soit LL(k) il faut pouvoir reconnaître l'usage d'une production à la vue des k premiers symboles dérivés de sa partie droite.
- Pour qu'une grammaire soit LR(k) il faut pouvoir reconnaître le pivot (partie droite de production) en ayant vu tout ce qui est dérivé de cette partie droite et à la vue des k prochains symboles de l'entrée.

Grammaire LR

- Une grammaire LR est une grammaire pour laquelle il est possible de construire la table d'analyse.
- Pour qu'une grammaire soit LL(k) il faut pouvoir reconnaître l'usage d'une production à la vue des k premiers symboles dérivés de sa partie droite.
- Pour qu'une grammaire soit LR(k) il faut pouvoir reconnaître le pivot (partie droite de production) en ayant vu tout ce qui est dérivé de cette partie droite et à la vue des k prochains symboles de l'entrée.
- donc LL(k) .... LR(k)

Items LR(0)

item d'une grammaire G = production de G avec un point repérant une position dans la partie droite.

- A -> XYZ fournit 4 items :
  - A -> •XYZ
  - A -> X•YZ
  - A -> XY•Z
  - A -> XYZ•
- A -> ε fournit A -> •

Opération *Fermeture*

I un ensemble d'items pour une grammaire G

Fermeture(I) est l'ensemble d'items construit à partir de I par les deux règles :

1. Initialement, placer chaque item de I dans *Fermeture*(I).
2. Si $A \to \alpha \bullet B\beta$ est dans *Fermeture*(I),
   pour chaque production $B \to \gamma$ de G,
   ajouter l'item $B \to \bullet\gamma$ à *Fermeture*(I) (si pas déjà présent).

   Appliquer cette règle jusqu'à ce que plus aucun item ne puisse être ajouté à *Fermeture*(I).

Opération *Transition*

Transition(I,X)

- I un ensemble d'items
- X symbole de la grammaire

Transition(I,X) est la fermeture de l'ensemble de tous les items $A \rightarrow \alpha X \bullet \beta$ tels que $A \rightarrow \alpha \bullet X\beta$ appartienne à I.

Construction des ensembles d'items

```
C ← { Fermeture({E'→ • E}) }
répéter
    pour chaque ensemble d'items I de C et
    pour chaque symbole X de la grammaire tel que
                Transition(I,X) soit non vide
                et non encore dans C faire
        ajouter Transition(I,X) à C
jusqu'à ce que plus aucun nouvel ensemble d'items
       ne puisse être ajouté à C
```

$$
\begin{array}{ll}
 & E' \rightarrow E \\
1,2 & E \rightarrow E + T \mid T \\
3,4 & T \rightarrow T * F \mid F \\
5,6 & F \rightarrow (E) \mid \textbf{id}
\end{array}
$$

C $\leftarrow$ { *Fermeture*({E'$\rightarrow$ • E}) }
répéter
    pour chaque ensemble d'items I de C et
    pour chaque symbole X de la grammaire tel que
                Transition(I,X) soit non vide
                et non encore dans C **faire**
        ajouter *Transition*(I,X) à C
jusqu'à ce que plus aucun nouvel ensemble d'items
        ne puisse être ajouté à C

Construction des tables d'analyse SLR

1. Construire C = $\{I_0, I_1, ..., I_n\}$, la collection des ensembles d'items LR(0) pour G'.
2. L'état i est construit à partir de I_i. Les actions d'analyse pour l'état i sont déterminées comme suit :
   1. Si $A \to \alpha \bullet a\beta$ est dans I_i et *Transition(I_i,a)* = I_j, remplir *Action[i,a]* avec *"décaler j"* (a est un terminal).
   2. Si $A \to \alpha\bullet$ est dans I_i, remplir *Action[i,a]* avec *"réduire par* $A \to \alpha$*"* pour tous les SUIV(A) (A ne doit pas être E').
   3. si $E' \to E\bullet$ est dans I_i, remplir *Action[i,$]* avec *"accepter"*.

   Si les règles précédentes engendrent des actions conflictuelles, la grammaire n'est pas SLR(1).
3. si *Transition*(I_i,A) = I_j alors *Successeur*[i,A] = j.
4. Toutes les entrées non définies sont positionnées à *"erreur"*.

	E' $\rightarrow$ E
1,2	E $\rightarrow$ E + T | T
3,4	T $\rightarrow$ T * F | F
5,6	F $\rightarrow$ (E) | **id**

1. Construire C = $\{I_0, I_1, ..., I_n\}$, la collection des ensembles d'items LR(0) pour G'.
2. L'état i est construit à partir de I_i. Les actions d'analyse pour l'état i sont déterminées comme suit :
   1. Si $A \rightarrow \alpha \bullet a\beta$ est dans I_i et *Transition*(I_i,*a*) = I_j, remplir *Action[i,a]* avec *"décaler j"* (a est un terminal).
   2. Si $A \rightarrow \alpha\bullet$ est dans I_i, remplir *Action[i,a]* avec *"réduire par* $A \rightarrow \alpha$*"* pour tous les SUIV(A) (A ne doit pas être E').
   3. si $E' \rightarrow E\bullet$ est dans I_i, remplir *Action[i,$]* avec *"accepter"*.
3. si *Transition*(I_i,A) = I_j alors *Successeur*[i,A] = j.

État	Action						Successeur		
	id	+	*	(	)	$	E	T	F
0									
1									
2									
3									
4									
5									
6									
7									
8									
9									
10									
11									

Comment ça marche ?

Les principales idées...

- proto-phrases droites, préfixes viables
- on a construit l'**automate LR(0)** qui reconnait les **préfixes viables** de la grammaire.
- notion d'**items valides** pour un préfixe viable
- L'ensemble des items atteint depuis l'état initial le long du chemin étiqueté γ dans l'automate LR(0) de la grammaire *est l'ensemble des items valides pour le préfixe viable* γ.

Comment ça marche ?

magie noire ...

Outils

Évidemment, tout ceci peut (doit !) être automatisé

outil classique : générateur d'analyseur LALR(1)

- yacc et bison en C/C++
- cup en Java
- PLY en Python
- GPPG en C#
- Yacc en Go
- Racc en Ruby
- ocamlyacc en OCaml
- Happy en Haskell
- Yecc en Erlang
- *insert your favorite language here*

Postparation

1. Terminez la première analyse par décalage réduction (transp. 47).
2. Dessinez l'arbre de dérivation correspondant.
3. Terminez l'automate LR(0) (solution sur chamilo, trouver l'erreur !)
4. Terminez la construction de la table SLR(1) (solution transp. 54, trouver l'erreur !)
5. Quel symbole représente chaque état de l'analyseur (transp. 54) ?

Bilan

1. Structure syntaxique
2. Analyse syntaxique
3. Analyse descendante
   - Analyseur prédictif
   - Conflits LL(1)
   - Analyseur prédictif itératif
4. Analyse ascendante
   - Principes
   - Analyse par décalage-réduction
   - Analyseurs LR
   - Grammaires LR
   - Construction des tables d'analyse SLR