Compilation

Analyse descendante

SI4 — 2018-2019

Erick Gallesio

Introduction

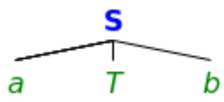
Principe de l'analyse:

À partir de l'axiome de la grammaire, *construire l'arbre d'analyse* permettant d'obtenir la phrase à analyser.

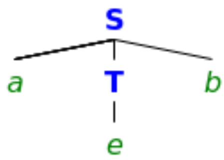
```
S → aTb | b                          (r1, r2)
T → cd | c | e                       (r3, r4, r5)
```

Analyse de la phrase P = `aeb`:

1. Le premier caractère de P permet de choisir $\mathbf{r_1}$.



2. P = a`e`b et on est sur **T** dans l'arbre. On choisit $\mathbf{r_5}$.



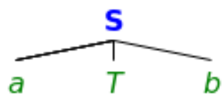
3. P = ae`b`. On est sur `b` dans le mot (et dans l'arbre) ⇒ **SUCCÈS**

Retours arrières

L'analyse descendante peut impliquer des retours en arrière:

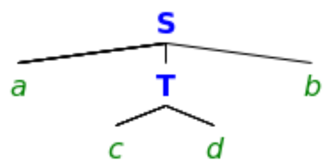
```
S → aTb | b                    (r1, r2)
T → cd | c | e                 (r3, r4, r5)
```

Analyse de la phrase P = `acb`:



1. Le premier caractère de P permet de choisir $\mathbf{r_1}$.
2. Maintenant P = a `cb` (on a le choix entre $\mathbf{r_3}$ et $\mathbf{r_4}$)

    - *choix de* $\mathbf{r_3}$

      Ici, P = ac `b` *mais problème (**d** dans l'arbre mais **b** à analyser)*
      *⇒ **RETOUR***

    - *choix de* $\mathbf{r_4}$ *…. ⇒ **SUCCÈS***

Analyse descendante

Pour analyser la grammaire précédente, il faut:

- avoir un analyseur capable de faire des retours en arrière.
- pouvoir voir 2 caractères en même temps:
  - *si* `c` *est suivi de* `d` *prendre* $\boldsymbol{r_3}$
  - *sinon prendre* $\boldsymbol{r_4}$

Dans le cas général, cela peut être plus compliqué (2 caractères peuvent ne pas suffire):

```
S → aSb | aSc | d
```

Exemple:

S → aSc → aaSbc → aaaSbbc → aaaaScbbc → aaaadcbbc

Ici, quand on lit le premier **a**, pour connaître la règle choisie, il faut regarder le **dernier** caractère du mot à analyser.

Analyseur descendant ND (1 / 4)

On peut construire facilement un analyseur descendant **non déterministe**:

Soit

```
E → T + E | T
T → F * T | F
F → (E) | int
```

- Chaque non terminal de la grammaire est implémentée par une fonction
- Les lexèmes sont dans un tableau et *next* pointe le prochain lexème.
- Si on a une alternative, on décompose en sous fonctions.
- Les fonctions renvoient un booléen indiquant si l'analyse a réussi ou pas.
- La fonction `verify(token)`:
    - *vérifie que next est bien égal à token et*
    - *avance next*

Analyseur descendant ND (2 / 4)

Pour la règle `E → T + E | T`, on obtient:

```
bool E1() { return T() && verify(PLUS) && E(); }
bool E2() { return T(); }
bool E() {
  token *save = next;
  return (next = save, E1 ()) ||  // Affectation inutile (pour la symétrie)
         (next = save, E2 ());
}
```

avec

```
bool verify(token tok) {
  return *next++ == tok;
}
```

- l'expression `int+int` renvoie vrai
- l'expression `(int+-int` renvoie faux
- *MAIS* l'expression `(int)*` renvoie vrai!!

Analyseur descendant ND (3 / 4)

- L'analyseur précédent renvoie vrai pour l'expression `(int)*` qui est fausse!
- Le problème ici est que l'analyseur
  - *reconnaît* `(int)` *comme étant une dérivation de* ***E****, et*
  - *s'arrête sur cette expression correcte*
  - *ne regarde pas ce qui arrive après une expression correcte*
  - *renvoie vrai sur* `int*int` *(mais en n'ayant analysé que* `int` *en fait).*
- $\Longrightarrow$ **Il faut vérifier que l'analyseur arrive à la fin de la phrase à analyser.**

On ajoute une règle:

```
S → E $
```

- S devient l'axiome
- $ est un lexème spécial qui dénote la fin de la phrase

Analyseur descendant ND (4 / 4)

Code complet de l'analyseur

```
// ---------- Start symbol
bool S() { return E() && verify(END); }

// ---------- F Non Terminal
bool F1() { return verify(OPEN) && E() && verify(CLOSE); }
bool F2() { return verify(INT); }
bool F()  { token *save = next; return (next=save, F1()) || (next=save, F2());}

// ---------- T Non Terminal
bool T1() { return F() && verify(MULT) && T(); }
bool T2() { return F(); }
bool T()  { token *save = next; return (next = save, T1()) || (next = save, T2());}

// ---------- E Non Terminal
bool E1() { return T() && verify(PLUS) && E(); }
bool E2() { return T(); }
bool E()  { token *save = next; return (next = save, E1()) || (next = save, E2());}
```

Pour démarrer l'analyse, il suffit d'appeler `S()`

Analyseur descendant ND: Problèmes (1/2)

Récursivité à gauche:

Avec cette méthode d'analyse, on ne peut pas avoir de récursivité à gauche.

En effet `E → E + T | T` ⇒

```
bool E1() { E() && verify(PLUS) && T(); }
bool E2() { T(); }
bool E()  { token *save = next; return (next = save, E1()) ... }
```

et donc E() $\Longrightarrow$ E1() $\Longrightarrow$ E() $\Longrightarrow$ E1 $\Longrightarrow$ …

Une récursivité gauche *immédiate* peut être éliminée facilement:

```
S → Sa | b   devient    S → bS'
                        S' → aS' | ε
```

Mais la récursivité peut ne pas être immédiate (plus difficile).

Analyseur descendant ND: Problèmes (2/2)

L'analyse d'une règle du type

```
INSTR → if ( EXPR ) then INSTR
      | if ( EXPR ) then INSTR else INSTR
```

peuvent provoquer un retour arrière de k lexèmes (k non borné) ⇒ analyse inefficace.

Factorisation à gauche:

> Elle consiste à introduire un nouveau terminal après le plus long préfixe commun d'une alternative.
> Ce nouveau non terminal produit ensuite une alternative avec les suffixes respectifs.

```
INSTR → if ( EXPR ) then INSTR SUITE
SUITE → else INSTR | ε
```

Analyseur à table

Analyseur ND n'est pas praticable sur de gros langages / grosses entrées.

On voudrait construire un analyseur tel que: étant donnés

- le terminal **a** que l'on vient de lire et
- le non terminal **A** en cours de dérivation,

on puisse connaître de façon sûre la règle à appliquer.

On va pour cela construire une table d'analyse. Pour cela, on aura besoin de:

- notion d'annulables
- notion de premiers
- notion de suivants

Notion d'annulables

On appelle *Annulables*, l'ensemble des non terminaux de la grammaire qui peuvent se dériver en la chaîne vide (ε).

Annulables = $\{ X \in N \mid X \overset{*}{\rightarrow} \varepsilon \}$

Construction de *Annulables*:

1. Si $X \rightarrow \varepsilon$, alors $X \in$ *Annulables*
2. Si $X \rightarrow Y_1\ Y_2 \ldots Y_n$ et que tous les non terminaux de Y_i sont annulables,
   alors $X \in$ *Annulables*.

Calcul de PREMIER (1 / 3)

La fonction PREMIER est une fonction associée à une grammaire G.

Si **α** est une chaîne de symboles terminaux ou non terminaux, PREMIER(α) est l'ensemble de tous les **terminaux** qui peuvent commencer une dérivation de **α**.

On a donc $x \in$ PREMIER(α) si:

- $x \in T$ et
- $\alpha \overset{*}{\rightarrow} x\beta_1\beta_2\ldots\beta_n$ (avec $\beta_i \in T \cup N$)

Note:

Si $\alpha \overset{*}{\rightarrow} \varepsilon$, alors $\varepsilon \in$ PREMIER(α)

Calcul de PREMIER (2 / 3)

Algorithme de construction des ensembles PREMIER:

Pour calculer PREMIER(X), répéter tant qu'on ne peut plus ajouter de terminal (ou ε) à aucun ensemble de PREMIERs.

1. si $X \in T$ alors PREMIER(X) = { X }
2. si $X \rightarrow \varepsilon$ alors ajouter ε à PREMIER(X)
3. si $X \in N$ et $X \rightarrow Y_1\ Y_2 \ldots Y_n$
    1. *ajouter les éléments de PREMIER(Y_1) (sauf ε) à PREMIER(X)*
    2. *si $\exists\, j\ (j \in \{2, \ldots, n\})$ où $\forall\, i\ (i \in \{1, \ldots, j\text{-}1\})$ on a $\varepsilon \in$ PREMIER(Y_j)*
        - ajouter PREMIER(Y_j) (sauf ε) à PREMIER(X)
    3. *Si $\forall\, i\ (i \in \{1, \ldots, n\})$ $\varepsilon \in$ PREMIER(i)*
        - ajouter ε à PREMIER(X)

Calcul de PREMIER (3 / 3)

```
S  → E $
E  → TE'
E' → +TE' | -TE' | ε
T  → FT'
T' → *FT' | /FT' | ε
F  → (E) | int
```

En appliquant l'algorithme précédent, on obtient:

```
PREMIER(S)  = PREMIER(S) = PREMIER(T) = PREMIER(F) = { '(', int }
PREMIER(E') = { '+', '-', ε }
PREMIER(T') = { '*', '/', ε }
```

Calcul de SUIVANT (1 / 3)

La fonction SUIVANT est une fonction associée à une grammaire G.

Pour un non terminal A,

- SUIVANT(A) est l'ensemble des terminaux x qui peuvent apparaître immédiatement à droite de A dans une dérivation: $S \overset{*}{\rightarrow} \alpha A x \beta$

Note:

Si A peut se trouver complètement à droite (càd $S \overset{*}{\rightarrow} \alpha A$), alors $\$ \in$ SUIVANT(A)

Calcul de SUIVANT (2 / 3)

Algorithme de construction des ensembles SUIVANT:

1. Placer $ dans SUIVANT(S), où
    - *S est l'axiome de la grammaire et*
    - *$ est le marqueur de fin d'entrée.*
2. Pour chaque production $A \to \alpha B \beta$ où $B \in N$,
   ajouter PREMIER(β) à SUIVANT(B) (sauf ε)
3. Pour chaque production $A \to \alpha B$,
   ajouter SUIVANT(A) à SUIVANT(B)
4. Pour chaque production $A \to \alpha B \beta$ où $\varepsilon \in$ PREMIER(β),
   ajouter SUIVANT(A) à SUIVANT(B)

Itérer sur 3 et 4 jusqu'à ce qu'on ajoute plus rien dans les ensembles suivants.

Calcul de SUIVANT (3 / 3)

```
S  → E$
E  → TE'
E' → +TE' | -TE' | ε
T  → FT'
T' → *FT' | /FT' | ε
F  → (E) | int
```

En appliquant l'algorithme précédent, on obtient:

```
SUIVANT(S)  = { '$' }
SUIVANT(E)  = { ')', '$' }
SUIVANT(E') = SUIVANT(E) = { ')', '$' }
SUIVANT(T)  = PREMIER(E') ∪ SUIVANT(E) (sans ε) = { ')', '$', '+', '-' }
SUIVANT(T') = SUIVANT(T)  = { ')', '$', '+', '-' }
SUIVANT(F)  = PREMIER(T') (sans ε) ∪ SUIVANT(T) = { '*', '/', ')', '$', '+', '-' }
```

Table d'analyse (1 / 2)

A l'aide des PREMIERs et des SUIVANTs, on peut construire une table d'analyse.

Cette table est un tableau **T** à deux dimensions qui permet de déterminer la production à appliquer pour un non terminal donné et le symbole terminal courant.

Construction de la table d'analyse

Soit une production A -> α

- $\forall$ x $\in$ PREMIER(α) - {ε}, ajouter la règle A → α à T[A,x]
- si ε $\in$ PREMIER(α), $\forall$ x $\in$ SUIVANT(A), ajouter la règle A → α à T[A,x]

Chaque case vide de T correspond à une erreur.

Table d’analyse (2 / 2)

	int	+	-	*	/	(	)	$
E	E → TE’					E → TE’		
E’		E’ → +TE’	E’ → -TE’				E ’→ ε	E’ → ε
T	T → FT’					T → FT’		
T’		T’ → ε	T’ → ε	T’ → *FT’	T’ → /FT’		T’ → ε	T’ → ε
F	F → int					F → (E)		

avec **PREMIERs et SUIVANTs:**

```
PREMIER(E) = { '(', int }        SUIVANT(E)  = { ')', '$' }
PREMIER(E')= { '+', '-', ε }     SUIVANT(E') = { ')', '$' }
PREMIER(T) = { '(', int}         SUIVANT(T)  = { ')', '$', '+', '-' }
PREMIER(T')= { '*', '/', ε }     SUIVANT(T') = { ')', '$', '+', '-' }
PREMIER(F) = { '(', int }        SUIVANT(F)  = { '*', '/', ')', '$', '+', '-' }
```

Analyse avec une table

- On utilise une pile contenant initialement `$S`
- En entrée on a un mot $m = m_1 m_2 \ldots m_n$ terminé par $
- On pointe le premier caractère du mot m (càd m_1)

Algorithme:

Soit X le sommet de la pile et m_i le caractère courant de m;

- si X est un terminal:
  - *si X = $ alors si m_i = $ alors ACCEPTER sinon ERREUR*
  - *si X = m_i alors enlever X de la pile et avancer sur l'entrée sinon ERREUR*
- si X est un non terminal
  - *si $T[X, m_i] = X \rightarrow Y_1 \ldots Y_n$*
    - dépiler X et empiler $Y_n \ldots Y_1$ (càd à l'envers)
    - indiquer en sortie la production $X \rightarrow Y_1 \ldots Y_n$
  - *si $T[X, m_i]$ est vide ERREUR*

Exemple d’analyse

Soit la phrase `(int)` à analyser:

Pile	Entrée	Sortie
$E	(int)$	E → TE’
$E’T	(int)$	T → FT’
$E’T’F	(int)$	F → (E)
$E’T’)E(	(int)$	
$E’T’)E	int)$	E → TE’
$E’T’)E’T	int)$	T → FT’
$E’T’)E’T’F	int)$	F → int
$E’T’)E’T’int	int)$	
$E’T’)E’T’	)$	T’ → ε
$E’T’)E’	)$	E’ → ε
$E’T’)	)$	
$E’T’	$	T’→ ε
$E’	$	E’→ ε
$	$	ACCEPTER

	int	+	-	*	/	(	)	$
E	E → TE’					E → TE’		
E’		E’ → +TE’	E’ → -TE’				E ’→ ε	E’ → ε
T	T → FT’					T → FT’		
T’		T’ → ε	T’ → ε	T’ → *FT’	T’ → /FT’		T’ → ε	T’ → ε
F	F → int					F → (E)		

Grammaire LL(1)

Une grammaire pour laquelle la table d'analyse n'a aucune case contenant plus d'une production est une **grammaire LL(1)**

L:

pour **L***eft to Right Scanning* car on parcourt le texte de la gauche vers la droite

L:

pour **L***eftmost derivations* car on utilise des dérivations gauches

(1):

car un **(1)** seul lexème d'avance (*lookahed*) est utilisé pour prendre une décision.

Grammaire LL(1)?

Soit la grammaire

```
S → aAb
A → cd|c
```

```
PREMIER(S) = { a }    SUIVANT(S) = { $ }
PREMIER(A) = { c }    SUIVANT(A) = { b }
```

	a	**b**	**c**	**d**	**$**
S	S → aAb				
A			A → cd A → c		

Pour T[A, c], on a deux règles ⇒ la grammaire n'est pas LL(1)

Par contre, si on a 2 lexèmes d'avance, on peut choisir (en fait cette grammaire est LL(2))

Construction d'un analyseur déterministe (1 /3)

La table d'analyse LL(1) peut aussi être utilisée pour construire un analyseur *à la main*.

On prend la grammaire simplifiée d'expressions suivante:

```
E  → TE'
E' → +TE' | ε
T  → FT'
T' → *FT' | ε
F  → (E) | int
```

	int	**+**	*****	**(**	**)**	**$**
E	E → TE'			E → TE'		
E'		E' → +TE'			E '→ ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → int			F → (E)		

Construction d'un analyseur déterministe (2 /3)

On modifie un peu la fonction `verify`:

```
bool verify(token tok) {
  if (*next != tok) {
    fprintf(stderr, "Invalid token '%c'\n", *next);
    return false;
  }
  next++;
  return true;
}
```

- on écrit une fonction par non terminal
- la table d'analyse nous donne les cas à traiter

Remarque:

La détection d'erreur sera plus précise puisque la table nous indique ce que nous attendons.

Construction d’un analyseur déterministe (3 /3)

	int	+	*	(	)	$
E	E → TE’			E → TE’		
E’		E’ → +TE’			E ’→ ε	E’ → ε

```
bool E() {
  switch (*next) {
    case INT:
    case OPEN:   return T() && Ep();
    default:     fprintf(stderr, "Expected INT or OPEN\n"); return false;
  }
}

bool Ep() {
  switch (*next) {
    case PLUS:   return verify(PLUS) && T() && Ep();
    case CLOSE:
    case END:    return true;
    default: fprintf(stderr, "Expected PLUS or CLOSE\n"); return false;
  }
}
```