Leçon 923 Analyse lexicale et syntaxique - Applications 06/06/2013

I Introduction à la compilation (voir Figure 1)

<u>Définition 1</u> Un compilateur est un programme qui lit le code d'un autre programme (en langage de programmation) et le traduit en langage machine (langage cible). Il peut également détecter des erreurs dans le programme source.

<u>Remarque</u>: La compilation s'approche par certains aspects de la conversion de fichiers. Certains outils seront réutilisés ailleurs.

1) Analyse lexicale

L'analyseur lexical regroupe les caractères en séquences appelées lexèmes, et pour chaque lexème, transmet une unité lexicale à l'analyseur syntaxique. Il supprime les éléments inutiles.

<u>Exemple 2</u>: en langage Caml,

let reponse = 42; (* pour n'importe quelle question *)

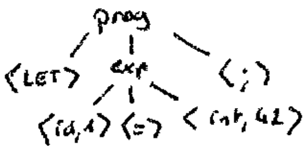
<LET> <id, 1> <=> <int, 42> <;>

position dans la table des symboles

2) Analyse syntaxique

L'analyseur syntaxique repère la structure du programme : il construit l'arbre de dérivation correspondant.

<u>Exemple 3</u>



3) Analyse sémantique

L'analyseur sémantique effectue des vérifications : les types, les variables non déclarées, ...

⚠ Si on parle d'assembleur ⇒ savoir donner des exemples !
Autres langages cibles ?
Compile-t-on toujours un programme pour l'utiliser ?
Erreurs de conception !

II Analyse lexicale

<u>Définition 4</u> : - Une unité lexicale est constituée d'un nom et d'une valeur d'attribut. Le nom représente le type de l'unité : mot clé, identificateur, opérateur, entier, ...

- Un motif est une expression régulière décrivant la forme que peut prendre une occurrence d'une unité lexicale.

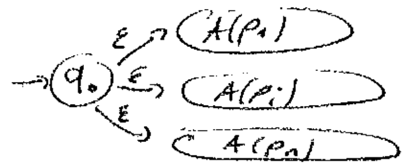
- Un lexème est une séquence de caractères dans le programme source reconnu par un motif, donc correspondant à une unité lexicale.

<u>Exemple 5</u>:

unité lexicale	int	float
motif	$(- + \varepsilon)$ chiffre$^+$	$(- + \varepsilon)$(chiffre$^+$$(. + \varepsilon)$ + . chiffre) chiffre* $(\varepsilon + E(- + \varepsilon)$ chiffre$^+)$
lexèmes	2, −27	.4, −1.27E12, 10E−14

<u>Construction de l'analyseur lexical</u> :

- On associe un automate à chaque motif p_i : $A(p_i)$
- On combine ces automates avec des ε-transitions :



L'analyse lexicale fait une lecture du code en suivant plusieurs chemins en parallèle. (L'automate est non déterministe)

<u>Gestion des conflits</u> :

- Déterminisation de l'automate → coûteux (voir fig 2)
- Choix du plus long préfixe
- ordres de priorité sur les sorties : les mots clés sont prioritaires par rapport aux identificateurs, ...

…rque : _ Certains outils pour l'écriture de programme font de l'analyse lexicale à mesure que le texte est tapé : emacs …
on voit alors bien l'utilisation d'automate (le → let → lettonie)
id ↑ LET ↑ id ↑

- Certains outils font une analyse lexicale très simplifiée : sur certaines calculatrices programmables, les mots clés sont accessibles via un menu.

Outil pour la construction de l'analyseur lexical : LEX / FLEX.

ifférence entre EX et LEX ?

LEX (dans sa version plus récente FLEX) est un constructeur d'analyseur lexicaux, un compilateur de compilateur en somme.
Il crée, à partir des motifs et des unités lexicales données en entrée, ainsi que des règles de priorité, un analyseur lexical.

III Analyse syntaxique

1) Cadre d'étude

ifférence entre …s-cont et …-cont !

Les langages de programmation sont généralement descriptibles par des grammaires algébriques hors contexte, G.
L'objet de l'analyseur syntaxique est de vérifier que la suite d'unités lexicales appartienne bien au langage $L(G)$ et de créer si tel est le cas l'arbre de dérivation correspondant, qui sera ensuite transformé en arbre de calcul.

1ère Idée : Algorithme de Cocke, Younger et Kasami
on récrit la grammaire sous forme normale de Chomsky, puis on effectue une analyse utilisant la programmation dynamique. C'est peu efficace ($O(n^3)$) et non utilisé en pratique

DVP 1

2ème Idée : Analyse descendante : on part du symbole initial de G et on reconstitue les étapes pour arriver à la chaine ⟹ Analyse LL

3ème Idée Analyse ascendante : on lit la chaine et on remonte les étapes jusqu'à arriver au symbole initial de G ⟹ Analyse LR

Les 2ème et 3ème idées sont utilisées en pratique.

2) Analyse descendante

Les noeuds de l'arbre de dérivation sont construits selon un parcours préfixe.

Exemple : Pour la grammaire G : $E \longrightarrow E+T \mid T$ d'axiome E,
$T \longrightarrow T\times F \mid F$ Terminaux $\{+, \times, id, (,)\}$
$F \longrightarrow (E) \mid id$

Pour la formule id+id, on a l'arbre de dérivation construit sur la Figure 3_a

Le problème est le choix de la règle de dérivation à appliquer

Définition 6 : Une grammaire est LL(k) s'il est possible de faire un analyseur syntaxique descendant qui lit k symboles d'entrée à l'avance.

Nous allons étudier les analyseurs LL(1). Quelques outils sont nécessaires à la prédiction de la règle à choisir. On se place dans le cadre d'une grammaire $G = (A, V, P, S)$

Définition 7 - Pour $\alpha \in (A\cup V)^*$, Premier $(\alpha) = \{a \in A \mid \exists u \in A^*, \alpha \xrightarrow{*} au\}$
- Pour $X \in V$, Suivant $(X) = \{a \in A \cup \{\$\} \mid \exists \alpha, \beta \in (A\cup V)^* \ S \xrightarrow{*} \alpha X a \beta\}$
où \$ est un marqueur de fin ($\beta = \varepsilon$).

Remarque : On peut calculer Premier récursivement, et Suivant à l'aide de Premier, en appliquant les règles suivantes jusqu'à un point fixe.

1) Si X est un terminal, Premier$(X) = \{X\}$.
2) Si $X \to \varepsilon$, $\varepsilon \in$ Premier(X)
3) Si $X \to Y_1 \ldots Y_k$, et si $\varepsilon \in$ Premier$(Y_1) \cap \ldots \cap$ Premier(Y_i) alors Premier$(Y_{i+1}) \subset$ Premier(X)

1) \$ $\in$ Suivant(S)
2) Si $A \longrightarrow \alpha B \beta$, Premier$(\beta) \setminus \{\varepsilon\} \subset$ Suivant(B)
3) Si $A \longrightarrow \alpha B$ ou $A \longrightarrow \alpha B \beta$ et $\varepsilon \in$ Premier(β), alors Suivant$(A) \subset$ Suivant(B)

Propriété 8 Une grammaire G est $LL(1)$ si et seulement si, pour toute paire de productions distinctes $A \rightarrow \alpha \mid \beta$ de G, on a :

1) $Premier(\alpha) \cap Premier(\beta) = \emptyset$
2) Si $\varepsilon \in Premier(\alpha)$, alors $Premier(\beta) \cap Suivant(A) = \emptyset$
   Si $\varepsilon \in Premier(\beta)$, alors $Premier(\alpha) \cap Suivant(A) = \emptyset$

Dans le cas d'une grammaire $LL(1)$, on peut donc construire un tableau : Table d'analyse prédictive qui contient les règles de production à appliquer à X en lisant a :

Table d'analyse prédictive (G)

```
Pour chaque règle X → α
  | Pour a ∈ Premier(α)
  |   | M(a, X) := X → α
  | Si ε ∈ Premier(α)
  |   | Pour chaque b ∈ Suivant X
  |   |   M(b, X) := X → α
```

Pour une grammaire $LL(1)$, on aura une seule règle par case et les cases sans règles sont des échecs.

Application 9 G n'est pas $LL(1)$. On la remplace par

G' : $E \rightarrow TE'$ d'axiome E.
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid id$

DVP 2

On calcule premiers et suivants, et on crée la table d'analyse associée.

3) Analyse ascendante

On part des feuilles de l'arbre et on essaie de remonter jusqu'à la racine.

Exemple 10 Figure 3.b

L'analyse ascendante suit un processus de lecture / réduction ; on lit des termes de l'entrée, on les réduit, on combine avec ce qu'on avait lu avant. La structure de donnée qui apparaît naturellement est une pile.

Définition 11 Une grammaire est $LR(k)$ s'il est possible de faire un analyseur syntaxique ascendant qui lit k symboles d'entrée à l'avance.

Exemple d'analyse $LR(0)$:

On construit l'automate à pile de l'analyse ascendante.
Il est déterministe si et seulement si G est $LR(0)$

DVP 3

4) Analyse syntaxique en pratique

Comme pour l'analyse lexicale, on a vu des méthodes assez systématiques d'analyse syntaxique. Des méthodes plus générales existent, pour d'autres classes de grammaires (même ambiguës) et il y a également des constructeurs d'analyseurs syntaxiques : Yacc par exemple. Il est utilisable sur le même principe que Lex.

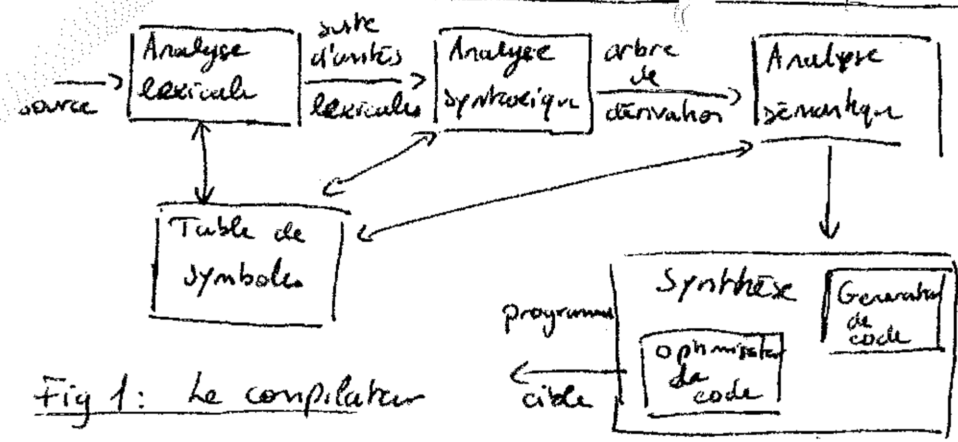


Fig 1 : Le compilateur

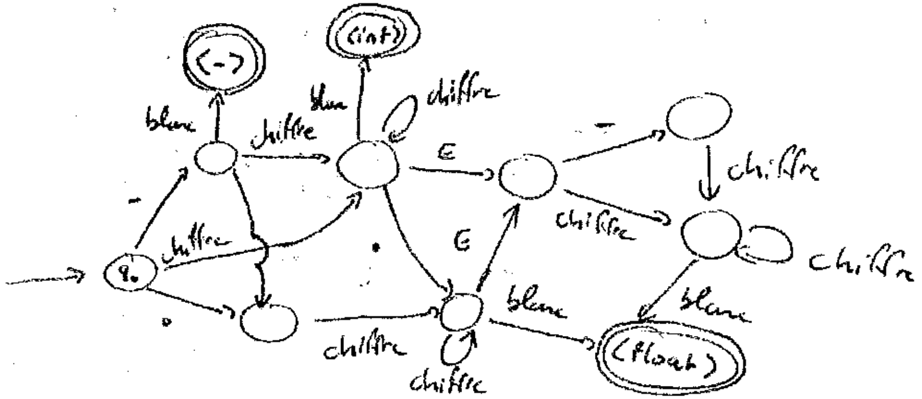
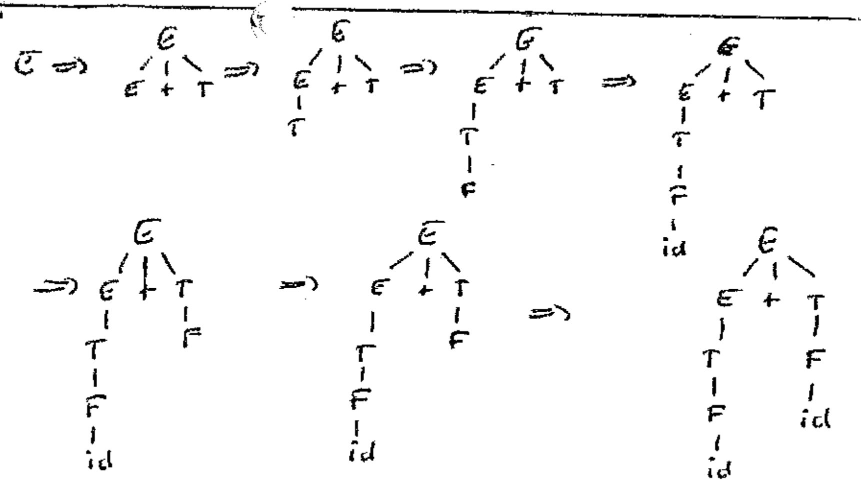
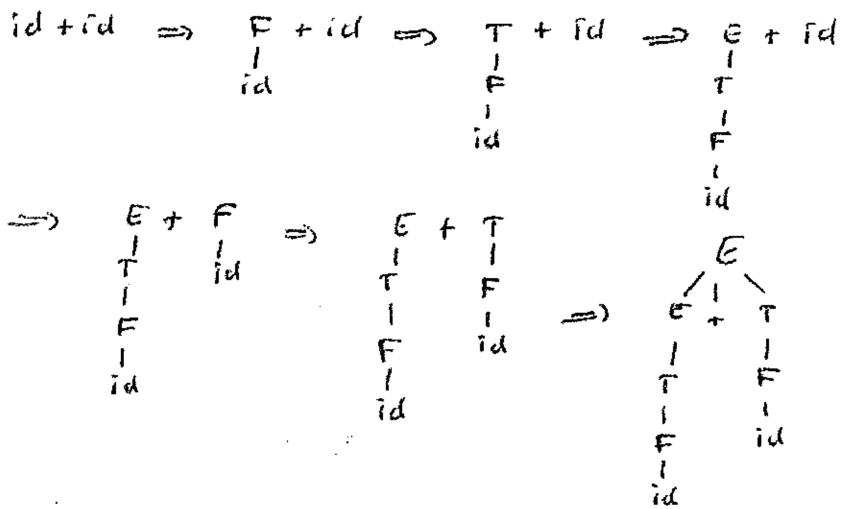


Fig 2 : Automate pour la reconnaissance des entiers et des flottants en langage de calculatrice (et -)



a - analyse descendante



b - analyse ascendante

Figure 3 - analyse lexicale de id+id, par la grammaire G

Algorithme de Cocke, Younger et Kasami

Référence : Carton

Leçon : 906, 907, 910, 923

Plan : I) Mise d'une grammaire sous forme normale de Chomsky
II) Réalisation d'un algorithme en utilisant la programmation dynamique.
III) Complexités ...

I)

Définition : Forme normale de Chomsky

Une grammaire est sous forme normale de Chomsky ssi toutes ses règles sont de la forme $T \to UV$ ou $T \to a$.

Théorème : existence de FN de Chomsky

Quelle que soit $G = (A, T, P, S_0)$ il existe G' sous forme de Chomsky telle que $L_{G'} = L_G \setminus \{\varepsilon\}$.

démonstration (en deux temps)

- Pour $a \in A$, on ajoute $V_a \in T$. On définit σ une substitution telle que $\sigma|_V = Id$, $\sigma(a) = V_a$ pour tout $a \in A$

On pose alors $P' = \{V_a \to a, a \in A\} \cup \{S \to \sigma(w), S \to w \in P, w \neq \varepsilon\}$

$T' = T \cup \{V_a, a \in A\}$.

$G' = (A, T', P', S_0)$ a toutes ses règles de la forme

$S \to S_1 \dots S_n$ ou $S \to a$ et $L_{G'} = L_G \setminus \{\varepsilon\}$

- Il reste à éliminer sans modifier le langage les règles $S \to S_1 \dots S_n$ $(n \neq 2)$

(i) $n = 1$: $S \to S_1$:

Pour chaque règle de la forme $S \to S_1$
pour chaque règle avec S dans le membre droit
On ajoute une nouvelle règle où l'on remplace S du membre droit par S_1
On supprime les règles $S \to S$.

$n > 2$ On note $P_1 \dots P_m$ les productions de la forme $S \longrightarrow S_1 \dots S_n$, $n > 2$.

On remplace chacune de ces productions P_k ($1 \leq k \leq m$)

$$S \to S_1 S_2^k \quad , \quad S_i^k \to S_i S_{i+1}^k \ , \ 1 \leq i < n-1 \quad , \quad S_{n-1}^k \to S_{n-1} S_n$$

Alors G'' est sous forme normale de Chomsky et est equivalente à G' donc à G. $\blacksquare$

II) On a alors une grammaire $G = (A, T, P, S_0)$ sous forme normale de Chomsky.

On veut resoudre le problème du mot pour G.

$w = w_1 \dots w_n \in A^*$. $\quad w[i,j] = w_i \dots w_j$

On note $E_{ij} = \{ S \in T, \ w[i,j] \in L_G(S) \}$

(but) • $w \in L_G \iff S_0 \in E_{1,n}$

(initialisation) • Si $i = j$ $\quad w[i,j] = w_i$ donc $E_{ii} = \{ S, \ S \to w_i \in P \}$

(hérédité) • Si $i < j$, $\quad w[i,j] \in L_G(S)$ ssi $\exists k \in [\![i, j-1]\!]$,

$S_1 \in E_{i,k}$, $S_2 \in E_{k+1,j}$ et $S \to S_1 S_2 \in P$.

L'algorithme de programmation dynamique s'en déduit.

<u>CYK</u> (w, G) $\qquad$ (G est sous forme de Chomsky)

```
∀(i,j) ∈ [[1,n]]²,
    initialiser E_{i,j} ← ∅
∀ i ∈ [[1,n]], ∀ S → a ∈ P
    Si w_i = a
        E_{ii} = {S} ∪ E_{ii}
Pour d de 1 à n-1
    Pour i de 1 à n-d
        j = i+d
        Appliquer la récurrence pour trouver E_{ij}
Retourner S_0 ∈ E_{1,n}
```

III) Complexités

- Pour la mise sous forme de Chomsky, on peut grandement augmenter le nombre de productions, mais si on prend en compte la taille des productions, c'est en fait seulement linéaire. $\Rightarrow O(n)$
- CYK comporte 3 boucles imbriquées de taille n (une boucle cachée dans "appliquer la récurrence" et une boucle de taille $O(|G|)$ cachée également. $\Rightarrow O(n^3 |G|)$

Construction d'un analyseur syntaxique

Référence : Dragon (Aho, Lam, Sethi, Ullman, "Compilateurs")

Leçon 907, 910, 923

Pour rappel :

$$G = (\{+, *, (,), id\}, \{E, T, F\}, P, E)$$

$$P: \quad E \to E+T \mid T$$
$$T \to T*F \mid F$$
$$F \to (E) \mid id$$

Cette grammaire n'est pas LL(1) (transitions linéaires gauches)

On la remplace par

$$E \to TE'$$
$$E' \to +TE' \mid \varepsilon$$
$$T \to FT'$$
$$T' \to *FT' \mid \varepsilon$$
$$F \to (E) \mid id$$

On va calculer les ensembles Premier et Suivant (cf Def 7)

	+	*	(	)	id	F	T'	T	E'	E
Premiers	+	*	(	)	id	(, id	*, ε	(, id	+, ε	(, id
Suivants	—	—	—	—	—	), +, *, $	+ $)	+,), $	), $	), $

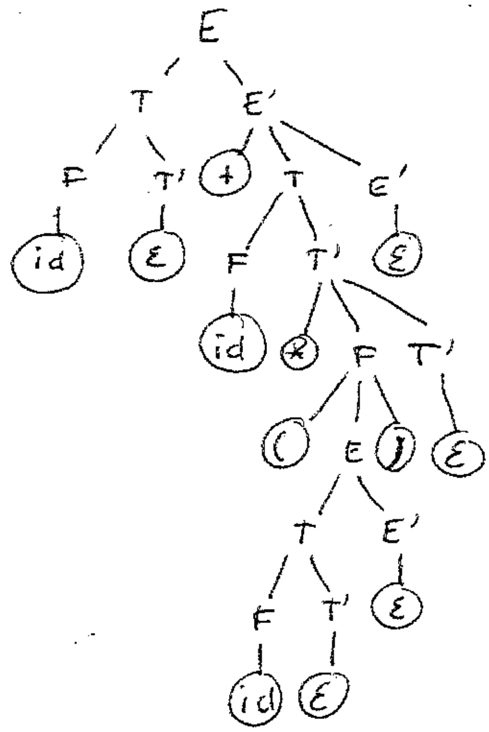
Table d'analyse :

	+	*	(	)	id	$
E			$E \to TE'$		$E \to TE'$	
E'	$E' \to +TE'$			$E' \to \varepsilon$		$E' \to \varepsilon$
T			$T \to FT'$		$T \to FT'$	
T'	$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$		$T' \to \varepsilon$
F			$F \to (E)$		$F \to id$	

Analyse d'une expression : id + id * (id)

lexème	variable	regle
id	E	E → TE'
	T, E'	T → FT'
	F, T', E'	F → id
+	T', E'	T' → ε
	E'	E' → + T E'
id	T, E'	T → FT'
	FT' E'	F → id
*	T' E'	T' → * FT'
(	FT' E'	F → (E)
id	E T' E'	E → TE'
	TE' T'E'	T → FT'
	FT'E'T'E'	F → id
)	T' E' T' E'	T' → ε
	E' T' E'	E' → ε
	T' E'	T' → ε
	E'	E' → ε

Construction de l'arbre : on applique les règles dans l'ordre, sur la feuille la plus à gauche



- Une ~~langage~~ grammaire LL(1) régulièrement utilisée ? LISP.

- LL(1) ⇒? LL(0) ? LISP