UNIVERSITÉ TOULOUSE III PAUL SABATIER

Rédigé avec LaTeX
Version du 10 octobre 2014

Traduction des langages

M1 Informatique – Développement Logiciel
Semestre 7

Cours donné par Christine Maurel
Rédigé par Antoine de Roquemaurel

2014

Avant-propos

— 7 séances de cours, 7 séances de TD ⇒ Rapide
— MCC : 1CT = 100% [a]
— Plus de cours/TD ⇒ Cours magistraux.
— Moyenne S7 doit être >= 10 + note UE >= 6

a. 22 Novembre

Table des matières

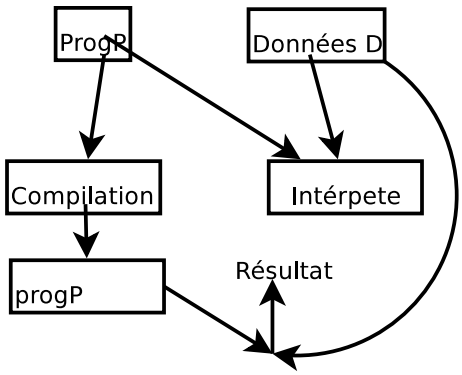
1 Introduction

La traduction des langaes peut être assimilée à de la « compilation ». C'est à dire comprendre pourquoi un programmae dans un langage de programmation est compris la machine où que les erreurs sont détectés.

1.1 Intéprétation ou Compilation

Une intérprétation utilise un intérpréteur et calcul lors de l'execution du programme.

Une compilation utilise un compilateur et traduit le programme. Aucune execution n'est nécessaire.



	Avantages	Inconvénients
Interpréteur	— Convivial — Mise au point rapide	— Moins efficace
Compilateur	— Efficacité — Optimisation possible	— Plus lourd

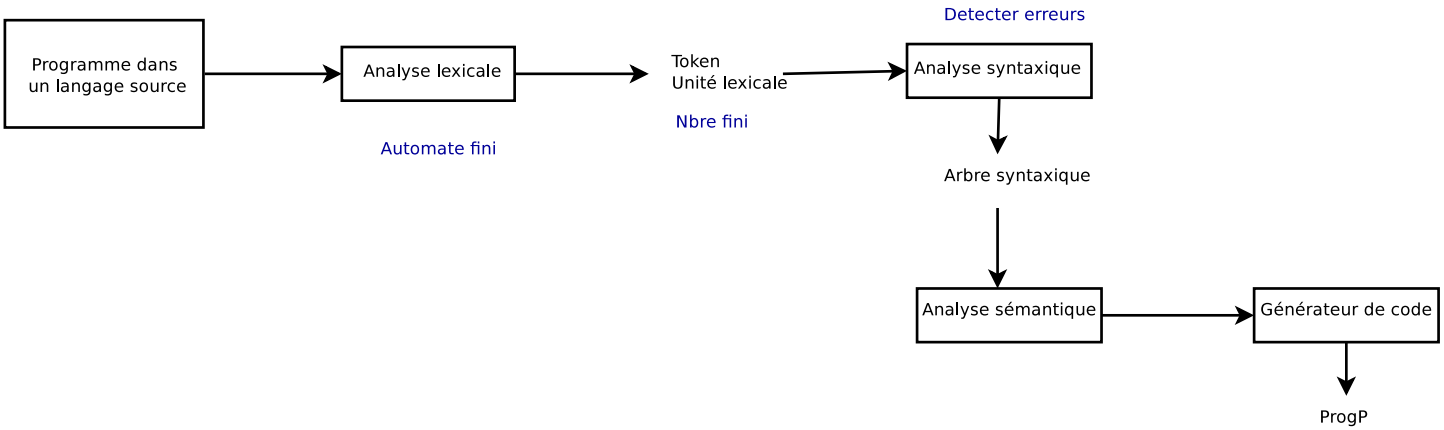


FIGURE 1.1 – Phases de compilation

2 Analyse lexicale

Un analyseur lexical doit découper un texte source en *tokens* , l'analyseur lexicale peut aussi être appelé scanner. L'analyseur lexical ne fonctionne pas tout seul, il est en général guidé par un analyseur syntaxique.

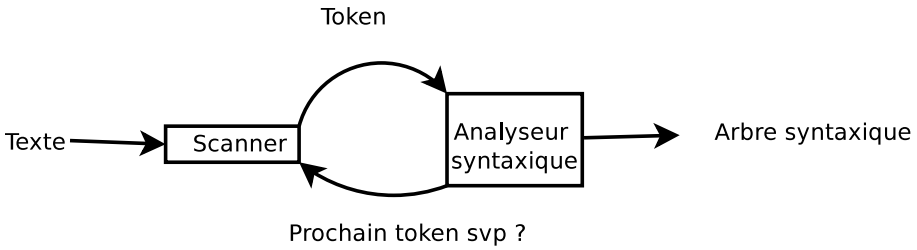


FIGURE 2.1 – Diagramme de traduction

2.1 Token

— Identificateur
— Mots clés
— Constantes numériques
— Opérateurs arithmétiques
— Opérateurs de comparaison
— Séparateur
— Commentaires
— Séparateurs

2.2 Identificateurs

— Commence par une lettre
— Suivi d'une suite éventuellement vide de lettres, de chiffres (et de caractères spéciaux)

Nombres entiers signés 2014, +2014, -2014

Alphabet $X = \{a, \cdots, z, \cdots 0, \cdots, 9, (+), (-)\}$

Automate fini qui reconnait identificateurs et nombres

$$\begin{aligned} L_0 &= l(L+c)^* + cc^* + (+)cc^* + (-)cc^* \\ L_0 &= \end{aligned}$$

2.3 Fonctionnement

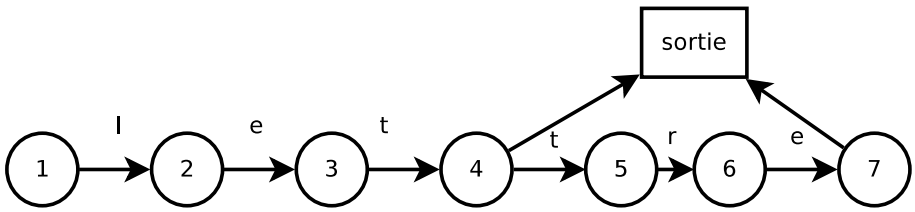
L'analyseur lexicale lit le texte caractère par caractère, découpe et reconnait des tokens(lexèmes) exprimé avec une regex.

Il existe des outils générateurs de scanner.

2.3.1 Les problèmes posés par l'analyse lexicale

2.3.1.1 Reconnaissance

Si on a `let` en mot clé et on a un identificateur `lettre`. On donne la priorité à l'unité syntaxique la plus longue, cf figure 2.3.1.1



2.3.1.2 Sur langage

Éventualité de faire un automate plus petit qui reconnait L' tel que $L \subseteq L'$ [1] avec des actions sémantiques plus « importantes ».

2.3.1.3 Le recul

Si on a $<$, on sait que l'unité syntaxique c'est $<$. Mais si on tombe sur $<=$ on a une autre unité syntaxique ; $\leq$.

Ce n'est pas réellement un problème car on a vu qu'on privilégie l'Unité Syntaxique(US) la plus longue. Si on tombe sur < 1 , il faut remettre 1 dans le flot d'entrée

1. C'est-à-dire un sur langage de L

2.3.2 La table des symboles

La table des symboles, appelé TDS, est un endroit où ranger les *tokens* rencontrés avec toutes les informations associées. Cela permet de calcul le « hashcode» pour la gestion des synonymes.

3 Analyse syntaxique

Objectif Voir si les *tokens* trouvés par l'AL forment une « phrase correcte» ou non par rapport à la grammaire du langage.

L'analyseur syntaxique est une grammaire reconnue par un automate à pile.

3.1 Analyse descendante

Reconnaître(ou pas) un mot $u \in X^*$ sur le ruban, on part de S et on a la pile.

À un instant donné, on a travaillé et on a reconnu un préfixe ω de u, $\omega \in X^*$. Pour ça on a mis $A\alpha$ dans la pile, $A \in N$

A est le sommet de pile et $\alpha \in (N \cup X)^*$

4 Génération de code

On veut traduire les instructions du langage de haut niveau L_1 dans un langage intermédiaire plus proche du langage cible. On a 3 langages :

- Pour L_1 c'est un langage impératif composé de l'affectation, et toutes les structures de contrôle(`if`, `else`, `case`, `switch`, ...) et des boucles (`for`, `while`, `repeat`, ...)
- Pour L_2 (langage intermédiaire) on prend un langage de quadruplets. (choix)
- Pour σ, on va utiliser un langage impératif style ADA, Pascal Traduction σ tq $\forall$ programme P , $\forall$ donnée $PD \equiv \sigma(P)D$

4.1 Langage intermédiaire des quadruplets

Nous avons un langage cible intermédiaire, qui ne ressemble pas vraiment à l'assembleur.
Les opérations se font directement en mémoire, pas de registres. Un quadruplet est une instruction à 4 champs dont 3 adresses mémoire : opération, opérande1, opérande2, résultat.

On peut faire l'affectation, les opérations arithmétiques, les branchements conditionnels ou inconditionnels.

Affectation `(:=, d, nil, e)`

Opération Arithmétique `(+, a, b, c)`

Branchemnet inconditionnel `(goto, nil, nil @)`

```
1 @i    +, a, b, t1 -- t1 := a + b
  @i+1  *, t1, c, t2 -- t2 := c * 1
3 @i+2  :=, t2, nil, d -- d := t2
  @i+3  >?, a, b, alpha1 -- si a > b aller en alph1 sinon faire suivant
```

Listing 4.1 – Exemple de quadriplets

4.2 Actions sémantiques couplées à l'analyseur descendant

L'analyseur syntaxique c'est le chef d'orchestre. Il est descendant et $LL(1)$

Il s'occupe de l'analyse lexicale(AL) et de faire les actions sémantiques.

- Il va engendrer un programme équivalent en quadruplets.

  > Il n'y a pas d'exécution !

- Pour ça on va avoir besoin d'informations à mémoriser ou à modifier.
- Les procédures de descente récursive vont avoir besoin de paramètres en entrée ou en sortie et de variables locales.

Les expressions arithmétiques définies par la grammaire suivante :

$$\begin{aligned} E &\rightarrow T\{+T\}* \\ T &\rightarrow F\{*F\}* \\ F &\rightarrow ident|(E) \end{aligned}$$

Notons que le symbole * sert ici pour le langage et pour définir l'opérateur de multiplication.

```
Procedure E is
begin
  T;
  while NEXTS = '+' loop
    SKIP('+');
    T;
    -- * => engendre un quadruplet boite a outils
    GEN(QUAD: String);
    -- engendre le prochain quadruplet (A,B,C,D)
    GEN("+, ?, ?, ?")
  endloop
end -- E
```

Il faut ajouter des arguments aux procédures E et T : $E \rightarrow T\{+T\}*$

```
Procedure E (out r :String) is u, t :String;
begin
  T(r);
  while NEXTS = '+' loop
    SKIP('+');
    T(t); --2e operande
    u := NEWTEMP; -- Permet de créer une variable temporaire
    GEN("+","^r^", "^t^", "^u"); -- ^ est la concaténation
    r := u;
  endloop
end //E
```

A Rappels

A.1 Grammaire

Une grammaire est fait pour raconter de quelle manières les mots du langages sont construit. On part de l'axiome S et on applique les règles de productions, ou réécriture.

Une grammaire $G =< N, X, P, S >$ avec :

N L'ensemble des non terminaux. Majuscules. $\{A, S, B\}$

X Alphabet, ensemble des terminaux. Minuscules $\{a, b, c\}$

P Règles de productions. À gauche on a un seul non terminal $P\{A \to \alpha, A \in N, \alpha \in (N \cup X)\}$

S $S \in N$ Axiome

$$\begin{aligned} G_1 &= < N, X, P, S > \\ N &= \{S\} \\ X &= \{a, b\} \\ P &= \{S \to abS, S \to a\} \end{aligned}$$

$$\begin{aligned} G_2 &= < N, X, P, S > \\ N &= \{S, A\} \\ X &= \{a, b, c\} \\ P &= \{S \to aAc \\ &\quad\ A \to bbA \\ &\quad\ A \to b\} \end{aligned}$$

$\omega \in X^*$ est un mot engendré par G, $\Rightarrow \omega \in L(G)$, avec $L(G)$ qui est un langage engendré par G.

Avec $G_1 : S \Rightarrow abS \Rightarrow ababS \Rightarrow ababa \in L(G_1)$

A.1.1 Théorèmes

A.1.1.1 Théorème d'Arden

P règle de production G, système d'équation de langages, c'est à dire résoudre L(G).

$X = r_1X + r_2 \Rightarrow X = r_1r_2$ est solution. Si $\lambda \notin r_1$ alors la solution est unique.

$$G_1 \rightsquigarrow S = \underbrace{ab}_{r_1} S + \underbrace{a}_{r_2}$$

$$G_2 \rightsquigarrow \begin{cases} S = aAc \\ A = \underbrace{bb}_{r_1} A + \underbrace{b}_{r_2} \end{cases}$$

A.1.1.2 Théorème d'Arden bis

$$X = Xr_1 + r_2 \Rightarrow X = r_2r_1^*$$

A.1.1.3 Théorème A^nB^n

$$X = YXZ + T \Rightarrow X = Y^nTZ^n$$

A.2 Reconnaître un langage avec un automate à pile

Un automate à pile est défini par $< Q, X, q_0, \Gamma, Z_0, F$

Q Ensemble d'état

X Alphabet

$q_0 \in Q$ Etat initial

Γ Alphabet de pile

Z_0 Fond de pile $Z_0 \in \Gamma$

F Ensemble d'états finales $F \subseteq Q$

σ Fonction de transition

$$
\begin{aligned}
S &\rightarrow aAc && \text{(A.1)}\\
A &\rightarrow bbA && \text{(A.2)}\\
A &\rightarrow b && \text{(A.3)}\\
N &= S, && \text{(A.4)}\\
X &= a, b, c && \text{(A.5)}\\
\sigma(q, \lambda, S) &= (q, aAc) && \text{(A.6)}\\
\sigma(q, \lambda, A) &= (q, bbA) && \text{(A.7)}\\
\sigma(q, \lambda, A) &= (q, b) && \text{(A.8)}\\
\sigma(q, a, a) &= (q, \lambda) && \text{(A.9)}\\
\sigma(q, b, b) &= (q, \lambda) && \text{(A.10)}\\
\sigma(q, c, c) &= (q, \lambda) && \text{(A.11)}
\end{aligned}
$$

Analyse de la séquence abbbbbc

Ruban	**Pile**	**Règle**
$\lambda abbbbbc$	S	A.6
$abbbbbc$	aAc	A.9
$\lambda bbbbbc$	Ac	A.7
$\lambda bbbbbc$	$bbAc$	A.10
$bbbbc$	bAc	A.10
$bbbc$	Ac	A.7
$bbbc$	$bbAc$	A.10
bbc	bAc	A.10
λbc	Ac	A.8
bc	bc	A.10
c	c	A.11
mot lu	pile vide	

L'analyse doit se faire en une seule lecture du ruban et de façon déterministe.

Pour cela, on regarde k symboles sur le ruban pour pouvoir décider de façon unique de la règle à appliquer. Cette analyse efficace est appelée analyse k-prédictive.

On peut écrire un algo déterministe pour la reconnaissance de $\omega \in L(G_2)$

```
Utiliser A.6;
Utiliser A.9 pour dépiler "a"
while il y a "bb" sur le ruban, do:
Utiliser A.7;
Utiliser deux fois A.10 pour dépiler les 2 "b"
end
```

B Table des figures