4TIN603U – Compilation – Licence 3 – 2020-2021

Lionel Clément

1er janvier 2021

Table des matières

Avertissement

Ces notes ne remplacent absolument pas le cours mais sont un complément pour travailler. Elles dépassent parfois le cours en apportant d'autres informations. Elles sont parfois dépassées en ne mentionnant pas des éléments qui ont pu être dits à l'oral.

Bibliographie

— *Modern Compiler Implementation in Java* 2nd Edition, 1998, Andrew W. Appel, Cambridge University Press
— *Compilers : Principles, Techniques, and Tools*, Addison-Wesley, 1986, Aho, Sethi, Ullman
— Très bonne traduction en francais : *Compilateurs : principes, techniques et outils*, Aho, Sethi, Ullman, InterÉditions, 1989. Dunod, 2000. 2e edition, Traducteurs : Deschamp, Lorho, Sagot, Thomasset

1 Introduction

Un compilateur est un logiciel de traduction d'un document depuis un langage **source** vers un langage **cible**. D'ordinaire le langage source est un langage de programmation évolué (Java, C, C++, etc.) qui permet à un développeur d'implémenter un programme en utilisant des notions relativement abstraites et en s'affranchissant de contingences liées au matériel sur lequel le logiciel doit tourner. Le langage cible est un code machine prévu pour programmer un ordinateur en particulier.

Un compilateur contient plusieurs modules que nous étudierons tout-à-tour :

— analyseur lexical
— analyseur syntaxique
— analyseur sémantique
— générateur de code intermédiaire
— optimiseur de code
— générateur de code

Le compilateur gère une ou plusieurs tables de symboles qui donnent au développeur la possibilité de déclarer des variables comme bon lui semble, mais aussi des fonctions, des procédures, les types, etc.

Il détecte et signale un ensemble d'erreurs à chaque niveau d'analyse, comme l'utilisation d'une variable non déclarée, une incompatibilité de type, l'utilisation d'une procédure avec de mauvais paramètres, etc. Et enfin produit le code final, soit en corrigeant les erreurs, soit lorsque le programmeur a corrigé son programme.

Le compilateur fait partie d'un ensemble d'outils permettant l'implémentation de logiciels. Nous distinguerons le compilateur à proprement parlé des autres programmes utilisés en amont ou en aval du processus de compilation dans son ensemble : Éditeur, Préprocesseur, Assembleur, Lieur, Chargeur.

Sources → Préprocesseur → Compilateur → Assembleur → Lieur → Chargeur

FIGURE 1 – Contexte du compilateur

Les phases et le résultat peuvent être différents d'un compilateur à un autre :

— Interprétation immédiate plutôt que compilation proprement dite : Postscript, Shell, HTML
— Production de code portable (bytecode) pour une machine virtuelle (`P-Code`, `Class Java`), code dédié à une machine en particulier (`Postcript`, code machine pour tel processeur)
— Langages sources plus ou moins structurés. L'assembleur par exemple est très peu structuré, alors que le langage Java l'est assez
— Optimisation du code plus ou moins poussée
— Analyse et correction éventuelle des erreurs plus ou moins poussée

Premier exemple illustrant les étapes de compilation :

Soit le programme C donnant le plus petit multiple commun à deux nombres :

```
int pgcd(int a, int b) {
   while (b != a) {
      if (a > b)
         a = a - b;
      else {
         /* Echanger a et b */
        int tmp;
        tmp = a;
        a = b;
        b = tmp;
      }
   return a;
}
```

1. Analyse lexicale :

   Repérage des commentaires, des mots réservés (*keywords*), et des identificateurs.

   Le code source pourrait être transformé en quelque chose comme ceci :

   KEYWORD(**int**) IDENTIF(**pgcd**) LPAR KEYWORD(**int**) IDENTIF(**a**) COMMA KEYWORD(**int**) IDENTIF(**b**) RPAR LCURVE KEYWORD(**while**) LPAR IDENTIF(**b**) DIFF IDENTIF(**a**) RPAR LCURVE KEYWORD(**if**) LPAR IDENTIF(**a**) GT IDENTIF(**b**) RPAR IDENTIF(**a**) AFF IDENTIF(**a**) MINUS IDENTIF(**b**) SEMI KEYWORD(**else**) LCURVE COMMENT KEYWORD(**int**) IDENTIF(**tmp**) SEMI IDENTIF(**tmp**) AFF IDENTIF(**a**) SEMI IDENTIF(**a**) AFF IDENTIF(**b**) SEMI IDENTIF(**b**) AFF IDENTIF(**tmp**) SEMI RCURVE KEYWORD(**return**) IDENTIF(**a**) SEMI RCURVE

2. Analyse syntaxique :

   Le code est transformé en une représentation arborescente comme illustré figure 2.

3. Analyse des types :

   Chaque variable et la fonction sont associées aux types suivants :

   pgcd : $int \times int \rightarrow int$

   a : int

   b : int

   tmp : int

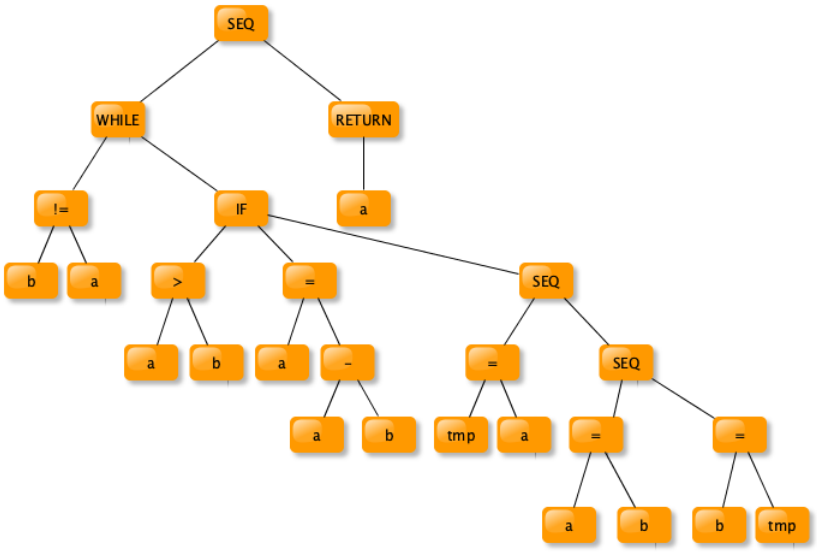

FIGURE 2 – **arbre de syntaxe abstraite**

4. Production du code assembleur qui pourrait ressembler à ceci selon le processeur visé :

```
WHILE1:
SET R1 DIFF a b
JUMP_IF R1 WHILE3
JUMP WHILE4
WHILE3:
JUMP_IF GT a b IF1
JUMP IF2
IF1:
MINUS R2 A B
SET A R2
JUMP IF3
IF2:
SET TMP A
SET A B
SET B TMP
IF3:
JUMP WHILE1:
WHILE4:
RETURN A
```

2 Analyse lexicale (*tokenization*)

L'analyse lexicale est traitée par un outil qui s'appelle un analyseur lexical, ou encore *scanner*, ou encore *tokenizer*. Nous voyons dans cette section comment développer un tel outil.

2.1 Expression régulière et langage rationnel

Quelques rappels

Soit un alphabet Σ, une expression régulière est :

1. ϵ (élément neutre), qui dénote l'ensemble qui contient la chaîne vide (à ne pas confondre avec $\emptyset$).
   $\sigma(\epsilon) = \{\epsilon\}$ [1]
2. a (lettre) dans Σ, qui dénote l'ensemble $\{a\}$.
   $\sigma(a) = \{a\}$

Étant donné deux expressions régulières R et S

1. (concaténation) RS, qui dénote l'ensemble des chaînes obtenues par concaténation d'une chaîne dans R et d'une chaîne dans S.
   $\sigma(RS) = \{xy / x \in \sigma(R) \wedge y \in \sigma(S)\}$
2. (union) $R + S$, qui dénote l'union des ensembles R et S.
   $\sigma(R + S) = R \cup S$
3. (étoile de Klenne) R^*, qui dénote la cloture par concaténation de chaînes dans R.
   $\sigma(R^*) = \bigcup_{n \in \mathbb{N}} R^n (R^i = \overbrace{R...R}^{i})$

Sucre syntaxique

$R^+ = RR^*$

$R|S = R + S$

$R? = (R + \epsilon)$

$R\{i, j\} = R^i + R^{i+1} + \cdots + R^j$

$R\{i\} = R^i$

. un caractère sauf `\n`

[^] un caractère Unicode (donc sur éventuellement plus d'un octet) sauf les *new line*

[*liste*] un caractère contenu dans la liste. La liste peut contenir des intervalles notés `a-z`

[^*liste*] un caractère non contenu dans la liste

Contexte d'une expression régulière

^R Le mot reconnu par R doit se trouver en début de ligne (début de fichier ou suite immédiate de `\n`).

R\$ Le mot reconnu par R précède immédiatement `\n` ou la fin de fichier.

R/S Le mot reconnu par R précède immédiatement un mot reconnu par S.

1. Nous notons $\sigma(R)$ l'ensemble dénoté par l'expression régulière R

Compléments utiles

`[:jletter:]` Un caractère qui peut être la première lettre d'un identificateur Java

`[:jletterdigit:]` Un caractère qui peut être une partie d'un identificateur Java

`[:letter:]` Une lettre

`[:digit:]` Un chiffre

`[:uppercase:]` Un caractère capitale

`[:lowercase:]` Un caractère bas de casse

`\d` Un chiffre

`\D` Tout caractère qui n'est pas un chiffre

`\s` Un espace (tabulation, espace, etc.)

`\S` Tout caractère qui n'est pas un espace

`\w` Chiffre, lettre, symboles de ponctuation

`\W` Tout caractère sauf `\w`

Négation La complémentation ne fait pas partie des opérations définissant les expressions régulières. En revanche, il est possible d'utiliser l'opération de complémentation des ensembles sur les langages définis. Et si les langages rationnels sont clos par complémentation ; le langage obtenu peut être de taille exponentiel. Il faut donc utiliser ces négations avec parcimonie.

`!`R Complément de $\sigma(R)$ (doit être exceptionnellement utilisé)

`~`R Toute chaine qui se termine par la première occurrence du texte dénoté par R. Par exemple `"/*"~"*/"` reconnait les commentaires du langage C.

2.2 Grammaire régulière, Automate à nombre fini d'états

Quelques rappels encore

À toute expression régulière R correspond un automate à nombre fini d'états A qui reconnait le langage $\sigma(R)$.

Ceci permet de construire un automatisme (un algorithme) qui reconnait le langage dénoté par une expression régulière.

À tout automate à nombre fini d'états A correspond un automate déterministe et minimal A' qui reconnait le même langage.

Cette dernière affirmation nous permet de construire un automatisme déterministe pour reconnaitre un langage dénoté par une expression régulière. C'est-à-dire un automatisme de complexité linéaire en fonction de la longueur de l'*input*.

2.3 JFlex

Cette dernière propriété est exploitée par les logiciels générateurs d'analyseurs lexicaux (`lex`, `flex`, `javacc`, `ocamllex`, `ply`, etc.), dont `jflex` fait partie.

`JFlex` est un logiciel écrit en Java qui contient principalement deux choses :

1. Un programme qui permet de traduire des expressions régulières en automate à nombre fini d'états.

2. Un programme qui permet de parcourir un automate à nombre fini d'états.

Le résultat de l'exécution de `JFlex` est une classe `Java`.

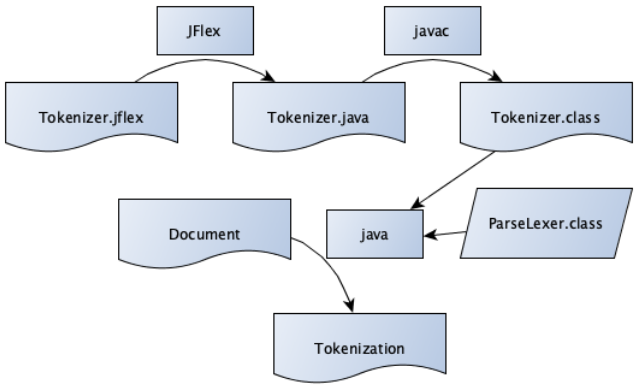

FIGURE 3 – **Schéma de l'utilisation de JFlex**

L'utilisateur a le loisir d'ajouter du code `java` qui sera exécuté :

1. Avant le parcours de l'automate
2. Après
3. Alors qu'un état final est reconnu pendant l'analyse de l'*input*.

Il lui est aussi possible de manipuler explicitement un automate dans lequel sera inséré l'automate des expressions régulières par composition (cf. plus loin).

2.4 Entités lexicales (token)

Qu'est-il possible d'analyser grâce à un programme écrit avec `JFlex` ?

Nous l'avons vu, `JFlex` est conçu pour transformer des expressions régulières en automate, et pour associer du code `Java` aux états finaux de cet automate. Il est donc naturel de dire que les entités qui seront reconnues par un programme écrit grâce à `JFlex` seront les mots d'un langage rationnel.

Cependant, dans la mesure où il est possible d'inclure au programme tout code `java`, les limites sont celles imposées par la programmation impérative et non celle des expressions régulières.

Il est donc techniquement possible de construire tout type de programmes – dont des compilateurs entiers – grâce à `JFlex`, mais comme nous le verrons dans la suite, l'outil n'est pas vraiment adapté à cet usage et d'autres logiciels seront utilisés en complément de `JFlex`.

Nous retiendrons que les `tokens`, ou mots reconnus par un *tokenizer*, sont des mots d'un **langage rationnel**.

Nous savons par ailleurs que le langage $a^n b^n$ n'est pas rationnel. Par conséquent les langages d'expressions bien parenthésées (expressions logiques, arithmétiques, algébriques, etc), ou structurées (langages de de types complexes, de structures de contrôle, etc.) ne **seront pas** des tokens et ne seront pas définis grâce à `JFlex`. L'utilisation de `JFlex` pour reconnaître de tels langages relève du bricolage informatique. Sauf dans les cas restreints où le langage non rationnel tient de

la simplicité (scripts de configuration, préprocesseur, etc.), nous utiliserons toujours un analyseur syntaxique en complément de l'analyseur lexical et laisserons à celui-ci la seule charge de traiter les *tokens*.

Démonstration d'un programme simple `JFlex` complet

2.5 États de l'automate

`JFlex` permet de déclarer des états de l'automate en plus de l'état initial et de programmer des sauts d'un état à un autre.

Cela n'est pas fait pour réaliser un automatisme et d'utiliser `JFlex` comme outil de programmation immédiat, mais pour donner des contextes complémentaires aux mêmes expressions régulières.

Exemples d'utilisations

1. Commentaires.
   Si les commentaires ne doivent pas simplement être ignorés, mais analysés pour ce qu'ils sont, il est évident que les mots clefs qui s'y trouvent ne sont pas ceux du langage ordinaire. Il est alors opportun de réaliser un état `COMMENT` qui rend explicite le contexte d'analyse du commentaire.
2. Chaînes de caractères.
   Les chaînes de caractères contiennent des caractères dit « échappés » (saut de ligne, espace, etc), ainsi que des variables non nommées permettant des substitutions (`%s`, `%d`, etc). Ces caractères doivent être analysés, alors que les autres éléments du langage (mots clefs, littéraux, etc) ignorés. Là encore un état `STRING` mérite d'être créé pour distinguer ce contexte.

Démonstration d'un programme `JFlex` avec états
Utilisation d'une variable pour compter un simple enchâssement
Utilisation d'une pile pour réaliser les inclusions de fichiers
Démonstration des limites de la solution (expression)