Chapitre 6

Le projet

6.1 Description du projet

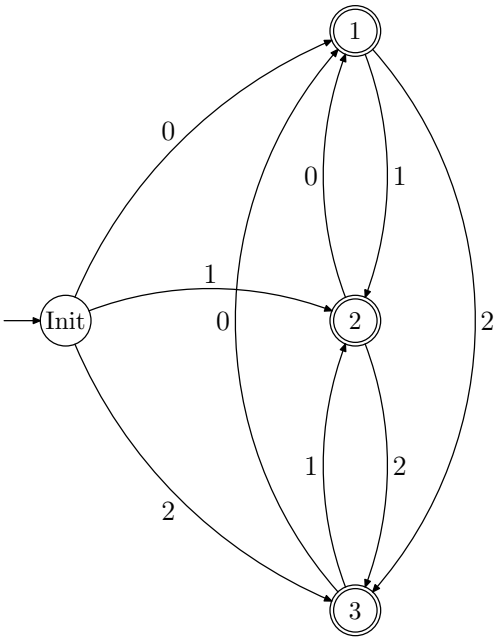
On veut écrire un *compilateur d'automate* qui
- lit un fichier contenant la description de l'automate
- produit en *machine virtuelle* un code capable de reconnaître ou pas tout mot qui lui sera fourni
- permet d'exécuter les machines virtuelles écrites

On veut aussi pouvoir tracer le comportement de l'automate (mode `debug`).

Exemple 6.1 – Automate sans pile

Ainsi l'automate de la figure 6.1, de la présente page sera représenté par le texte 6.1, page suivante.

Figure 6.1 – Automate sans pile à compiler

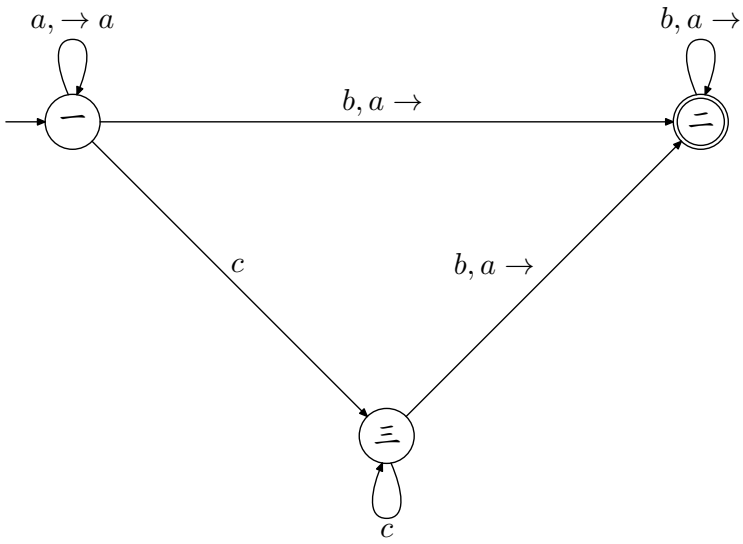


Texte 6.1 – Automate sans pile

```
Automate(0)={
// Cet automate n'a pas de pile
// Il correspond à l'automate du cours 5, page 5
    etats=["1","2", "3","Init"] // Le 0 est en dernier, c'est plus clair
// Chaque état est repéré par son numéro dans la liste etats
// Cette liste commence à l'indice 0
    initial =3 // L'état Init
// final est une liste, même s'il n'y a qu'un état final
    final= [0,1, 2]
    transitions = [(3→0,`0`),(3 →1,`1`),(3→ 2,`2`),
        (0 → 1,`1`),(0 → 2,`2`),(1 → 0,`0`),(1 → 2,`2`),
        (2 → 0,`0`),(2 → 1,`1`)]
}
```

On s'autorisera aussi des automates (déterministes) à une pile (voir la figure 6.2, de la présente page et le texte 6.2, de la présente page) et à deux piles (voir la figure 6.3, page ci-contre et le texte 6.3, page suivante).

Figure 6.2 – Automate à une pile à compiler

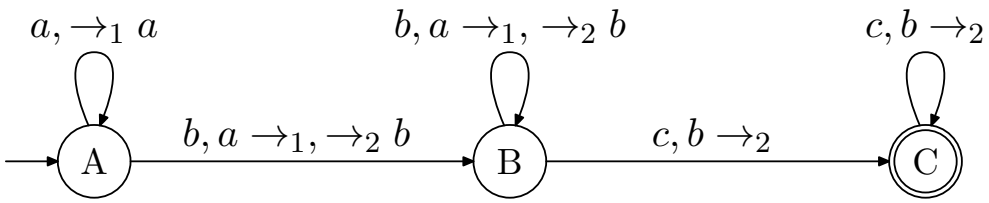


Texte 6.2 – Automate à une pile

```
/* Cet automate a une pile, il permet de programmer toutes les grammaires BNF et
de simuler la reconnaissance des langages algébriques*/
Automate(1) ={
/* Cet automate a une pile
 Il correspond à l'automate reconnaissant le langage
 a^n.c^p.b^n, où n>0 */
    etats =["一","二","三"] // Les noms peuvent être différents de numéros
// Chaque état est repéré par son numéro dans la liste etats
    initial= 0
// final est une liste, même s'il n'y a qu'un état final
```

```
    final =[1]
    transitions=[(0→ 0, `a`,(→, `a`)),
        (0 → 1,`b`, (`a`,→)),
        (2 → 1,`b`, (`a`,→)),
        (0 →2,`c`,()), // Pas d'action de pile
        (2 → 2, `c`), // Pas d'action de pile
        (1 → 1,`b`,(`a`,→))]
}
```

Figure 6.3 – Automate à deux piles à compiler



Texte 6.3 – Automate à deux piles

```
/* Cet automate a deux piles */

Automate(2) = {
/* Cet automate a deux piles
// Il correspond à l'automate reconnaissant le langage
// a^n.b^n.c^n où n>0 (cours 10, page 18) */
    etats = [`A`,`B`, `C`]
    // On pourrait mettre aussi [`a`, `b`,`c`]
// Chaque état est repéré par son numéro dans la liste etats
    initial = 0
// final est une liste, même s'il n'y a qu'un état final
    final= [2]
    transitions=[(0 → 0, `a`, (→,`a`)), // Pas d'action sur la pile 2
        // On pourrait aussi écrire (0 → 0, `a`, (→,`a`),())
        (0 → 1, `b`, (`a`, →),(→, `b`)),
        (1 → 1, `b`, (`a`, →), (→,`b`)),
        (1 → 2, `c`, (), (`b`, → )), // Pas d'action sur la pile 1
        (2 → 2, `c`,(), (`b`, →))]
}
```

Le mode `debug` permet de suivre les changements d'états des automates. Voici par exemple, à la session 6.1, de la présente page, l'exécution du troisième automate qui a été compilé dans une machine virtuelle notée `VM`.

Terminal 6.1 – Exemple d'exécution d'un automate compilé

```
root# Executeur VM
Donner le mot d'entrée : abc
Le mot abc est accepté !

root# Executeur -debug VM
```

```
Donner le mot d'entrée : abc
  -> État : A  Pile 1 : Vide  Pile 2 : Vide
a -> État : A  Pile 1 : a     Pile 2 : Vide
b -> État : B  Pile 1 : Vide  Pile 2 : b
c -> État : C  Pile 1 : Vide  Pile 2 : Vide
Le mot abc est accepté !

root# Executeur VM
Donner le mot d'entrée : abbc
Le mot abbc est refusé !

root# Executeur -debug VM
Donner le mot d'entrée : abbc
  -> État : A  Pile 1 : Vide  Pile 2 : Vide
a -> État : A  Pile 1 : a     Pile 2 : Vide
b -> État : B  Pile 1 : Vide  Pile 2 : b
b -> Erreur : Pile 1 vide !
Le mot abbc est refusé !

root# Executeur VM
Donner le mot d'entrée : aaabbc
Le mot aaabbc est refusé ! Pile 1 et Pile 2 non vides

root# Executeur -debug VM
Donner le mot d'entrée : aaabbc
  -> État : A  Pile 1 : Vide  Pile 2 : Vide
a -> État : A  Pile 1 : a     Pile 2 : Vide
a -> État : A  Pile 1 : aa    Pile 2 : Vide
a -> État : A  Pile 1 : aaa   Pile 2 : Vide
b -> État : B  Pile 1 : aa    Pile 2 : b
b -> État : B  Pile 1 : a     Pile 2 : bb
c -> État : C  Pile 1 : a     Pile 2 : b
Le mot aaabbc est refusé ! Pile 1 et Pile 2 non vides
```

6.2 Analyse lexicale

On peut constater sur les exemples donnés dans les textes 6.1, page 86, 6.2, page 86 et 6.3, page précédente les traits lexicaux suivants :

- il y a des commentaires de ligne, commençant par `//`
- il y a des commentaires sur plusieurs lignes, commençant par `/*` et finissant par `*/`
- il y a des espaces autorisées dans les descriptions (autour de `=` par exemple)
- des caractères particuliers sont présents dans la syntaxe (`→` et `` ` ``)
- dans les noms d'états, on s'autorise les mots français et chinois (attention ! ces mots sont utilisés dans le mode `debug`)

Remarque 6.1

On a besoin, pour les lexèmes, de leur type et de leur valeur, on pourra donc définir les lexèmes de la manière décrite au code ~~`lexemec`~~non référencé `C` 6.1, de la présente page.

Code non référencé `C` 6.1 – Type de lexème

```c
typedef enum TYPE
{
  Mot_clef,
```

```
  Nombre,
  Chaine,
  Caractere
} TYPE;

typedef struct lexeme
{
  TYPE type;
  char * valeur;
} lexeme;
```

6.3 Analyse syntaxique

Dans ce projet, l'analyse syntaxique est assez simple.

6.4 Analyse sémantique

Les vérifications faites sur l'arbre syntaxique seront, par exemple

1. vérification des différents types ;
2. vérification que les noms des états utilisés dans les transitions sont bien déclarés auparavant ;
3. vérification que les transitions ont des syntaxes cohérentes avec le nombre de piles ;
4. vérification que l'automate est bien déterministe ;
5. etc.

6.5 Modèle d'exécution

6.5.1 Compilation en machine virtuelle

La compilation consistera à créer deux fichiers

1. un fichier `TS.txt` contenant la table des symboles (on y trouvera entre autres, le nom de chaque état et son *adresse* dans la machine virtuelle) ;
2. un fichier `VM` (*virtual machine* ou machine virtuelle) qui contiendra un tableau d'entiers qui sera donc notre code compilé, on appellera *adresse d'un état* l'indice, dans `VM` où commence la définition de l'état.

▶ *Initialisation* : À l'adresse 0, on notera le *nombre de piles*, à l'adresse 1, *l'adresse* de l'état initial et à l'adresse 2, le *nombre des états d'acceptation*, suivi dans les cases suivantes de `VM` des adresses des états d'acceptation.

▶ *Automate sans pile* : chaque état sera représenté par :

$$n_0,\ \underbrace{c_1, k_1, \ldots\ldots, c_{n_0}, k_{n_0}}_{n_0 \text{ couples de la forme } (c,k)}$$

où n_0 est le nombre de transitions partant de l'état, c le caractère déclenchant la transition et k *l'adresse de* l'état activé par la transition. Voir l'exemple 6.2, page suivante.

▶ *Automate à une pile* : chaque état sera représenté par :

$$n_0, \underbrace{c_1, k_1, p_1, a_1 \ldots\ldots, c_{n_0}, k_{n_0}, p_{n_0}, a_{n_0}}_{n_0 \text{ quadruplet de la forme } (c,k,p,a)}$$

où n_0 est le nombre de transitions partant de l'état, c le caractère déclenchant la transition, k l'adresse de l'état activé par la transition, p le caractère à mettre ou à enlever de la pile ~~(0 si on ne fait rien)~~, $a \in \{-1, 0, 1\}$ l'action à faire sur la pile (-1 : on enlève, 0 : on ne fait rien, 1 : on ajoute le caractère sur la pile).

▶ *Automate à deux piles* : chaque état sera représenté par :

$$n_0, \underbrace{c_1, k_1, p_1, a_1, q_1, b_1 \ldots\ldots, c_0, k_{n_0}, p_{n_0}, a_{n_0}, q_{n_0}, b_{n_0}}_{n_0 \text{ 6-uplet de la forme } (c,k,p,a,q,b)}$$

où n_0 est le nombre de transitions partant de l'état, c le caractère déclenchant la transition, k l'adresse de l'état activé par la transition, p le caractère à mettre ou à enlever de la première pile ~~(0 si on ne fait rien)~~, $a \in \{-1, 0, 1\}$ l'action à faire sur la première pile (-1 : on enlève, 0 : on ne fait rien, 1 : on ajoute le caractère sur la pile), q le caractère à mettre ou à enlever de la deuxième pile (0 si on ne fait rien) et b l'action à faire sur la deuxième pile, toujours dans $\{-1, 0, 1\}$.

Exemple 6.2 – Codage d'un état dans un automate sans pile

Supposons que l'état 12 s'appelle "Ici" et qu'il soit codé dans la VM à partir de l'indice 30. (L'état 15 est à l'adresse x, l'état 18 à l'adresse y et l'état 5 à l'adresse z)

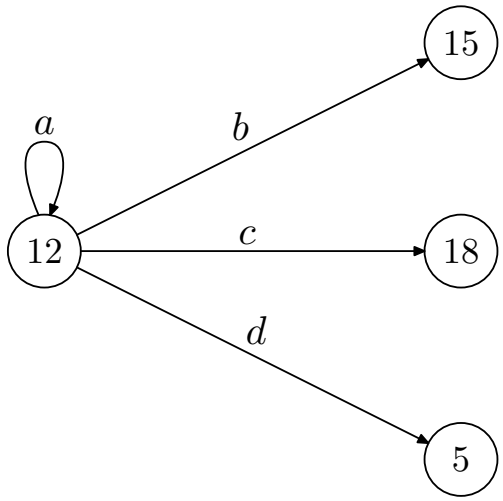


Table des symboles : Nom = "Ici", adresse = 30

VM	29	30	31	32	33	34	35	36	37	38	39
	···	4	97	30	98	x	99	y	100	z	···

6.5.2 Compilation et exécution

Le compilateur (~~`compile_automate`~~`Compilateur`) prendra en entrée un fichier `.txt` contenant la description de l'automate à compiler (voir les descriptions dans les textes 6.1, page 86, 6.2, page 86 et 6.3, page 87) et produira deux fichiers : la table des symboles notée `TS.txt` et la machine virtuelle notée `VM`. Il sera construit en respectant les étapes : analyse lexicale, analyse syntaxique, analyse sémantique, compilation.

L'exécuteur de la machine virtuelle devra fonctionner comme montré à la session 6.1, page 87.

6.6 Contraintes techniques

1. Les seules bibliothèques `C` autorisées sont données par le code non référencé `C` 6.2, de la présente page.
2. Pour chaque étape devront être fournis les fichiers
   — *analyse lexicale* `analyse_lexicale.c`, et un fichier de test `test_AL.c`
   — *analyse syntaxique* `analyse_syntaxique.c`, et un fichier de test `test_ASy.c`
   — *analyse sémantique* `analyse_semantique.c`, et un fichier de test `test_ASe.c`
   — *compilation* `compile_automate.c`, et un fichier de test `test_compil.c`
   — *assemblage* `Compilateur.c`
   — *exécution* ~~`runtime.c`~~`Executeur.c`
3. un rapport (la qualité de ce rapport est très importante !) décrivant les choix effectués, les problèmes rencontrés, les solutions apportées à ces problèmes, ainsi que des exemples de tests de chaque étape sera fourni en `.pdf`
4. le tout sera livré sur `Moodle` avant le 10 janvier, 23h59 sous la forme d'une archive .zip

Code non référencé `C` 6.2 – Bibliothèques autorisées

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <regex.h>
```

Remarque importante 6.2

Les fichiers doivent pouvoir être compilés sans erreur ni warning par le compilateur `linux` (`gcc`). Attention à bien vérifier que c'est le cas, certains étudiants ont, l'année dernière, livré des codes qui ne se compilaient pas sous `Linux` et ont perdu beaucoup de points !

Toute tricherie sera sévèrement punie.

— La qualité du travail sera appréciée fortement, même si, à la fin, cela ne marche pas tout le temps (mettre en ce cas les problèmes rencontrés et les solutions essayées dans le rapport).
— Un projet raisonnablement travaillé et respectant les consignes aura une bonne note.

Je rappelle que

$$\text{projet} = 80\% \text{ et attitude} = 20\%$$