上海交通大学巴黎卓越工程师学院
Ecole d'Ingénieurs Paris SJTU

Théorie des langages de programmation

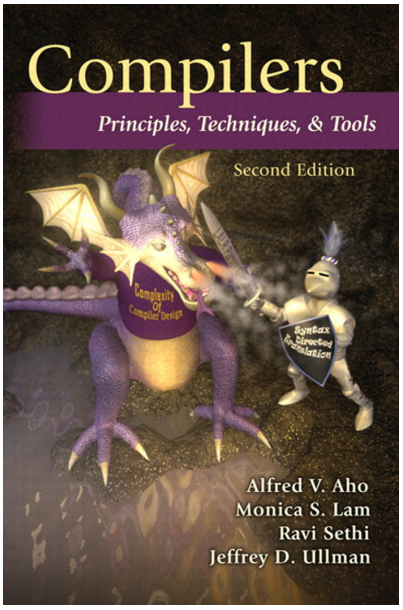
Alain Chillès (祁冲)

Préambule

L'objet de ce cours est de comprendre les outils fondamentaux pour écrire un compilateur ou toute autre traduction d'un langage compréhensible par un humain en un langage compréhensible par un ordinateur. Les choix effectués pour ce cours sont

1. programmer en langage `C` de base ;
2. présenter toutes les phases nécessaires à la conception et à la réalisation d'un compilateur ;
3. pour chaque phase, proposer des outils élémentaires pour les réaliser (outils de réflexion et non outils logiciels) ;
4. chaque année, il sera demandé de réaliser un mini-compilateur dans le cadre du projet.

Ce cours est loin d'être exhaustif ! On trouvera tous les compléments utiles dans le livre [1].



Livre de référence

Alain Chillès

Table des matières

Liste des codes Bash

Liste des codes C

Liste des codes dans divers langages

Liste des figures

Chapitre 1

Historique et particularités

1.1 De quoi parle-t-on ?

Définition 1.1 – Langage de programmation

Un *langage de programmation* est un langage (on verra une définition plus formelle ultérieurement) utilisé pour obtenir une exécution par l'ordinateur d'une tâche souhaitée.

Exemple 1.1

On utilise à SPEIT les langages `C`, `C++`, `Python`, `Markdown`, LaTeX 2_ε...

Remarque 1.1

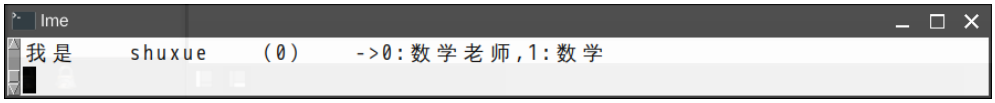
Ce cours s'intéresse aux moyens mis en œuvre pour, à partir d'un texte en langage de programmation (*code*), obtenir le résultat escompté. Est-ce bien utile ? Oui, pour

1. les futurs développeurs de compilateurs ;
2. tout besoin de traduction d'un langage A vers un langage B (activité hélas fréquente) ;
3. comprendre les mécanismes intervenant dans l'utilisation d'une machine programmable.

Exemple 1.2 – Exemple personnel

Aucun logiciel ne permettait sur mon système bizarre de taper du chinois, or cela m'était indispensable ! Voir la figure 1.1, de la présente page.

Figure 1.1 – 我是数学老师



Définition 1.2 – Constituants d'un langage

Le vocabulaire du langage autorisé s'appelle le *lexique*, les règles de grammaire du langage s'appelle la *syntaxe*, le sens des phrases du langage s'appelle la *sémantique*.

Exemple 1.3 – Exemple français

1. La phrase « Paris est fsjkhmqskh. » contient des mots qui ne sont pas dans le lexique (on parle d'*erreur lexicale*).
2. La phrase « Paris est pomme. » est correcte d'un point de vue lexical, mais contient une erreur de grammaire (on parle d'*erreur syntaxique*).
3. La phrase « Paris est une pomme. » est correcte d'un point de vue lexical et grammatical, mais n'a pas de sens autre que poétique (on parle d'*erreur sémantique*).

Remarque importante 1.2

Les règles lexicales et syntaxiques de chaque langage peuvent être très différentes. On peut voir la différence entre le langage `R` et le langage `Python` dans les sessions 1.1, de la présente page et 1.2, de la présente page.

Terminal 1.1 – `a<-2` en `R`

```
root# R --quiet
> a=3
> a<-2
> a
[1] 2
```

Terminal 1.2 – `a<-2` en `Python`

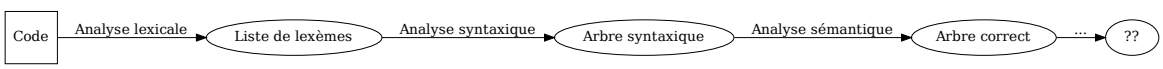
```
root# python3 -q
>>> a=3
>>> a<-2
False
>>> a
3
```

Définition 1.3 – Analyses

La *compilation* doit généralement (sauf dans les cas très simples) se diviser en plusieurs étapes (voir la figure 1.2, de la présente page) :

1. *l'analyse lexicale* : à partir du code fourni, l'analyse lexicale consiste à découper le code en une liste ou un tableau de *lexèmes* (mots du lexique), ou à générer des erreurs lexicales ;
2. *l'analyse syntaxique* : à partir de la liste ou du tableau de lexèmes, l'analyse syntaxique consiste à générer un *arbre syntaxique* correspondant ou à générer des erreurs syntaxiques (certains compilateurs peuvent éventuellement corriger certaines de ces erreurs) ;
3. *l'analyse sémantique* : à partir de l'arbre syntaxique correct, l'analyse sémantique consiste à générer (voire en corrigeant) un arbre syntaxique qui a un sens ;
4. ... le reste sera étudié après.

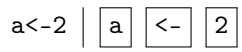
Figure 1.2 – Début de la compilation



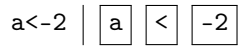
Remarque 1.3

Ainsi, pour reprendre les exemples proposés à la remarque 1.2, page ci-contre, l'analyse lexicale produit

1. R :



2. Python



Il nous faut donc des outils capables de décrire ces différences d'analyse !

1.2 Exemples d'erreurs

1.2.1 Erreurs lexicales

Code R 1.1

```r
2b <- 3
```

```
Erreur : unexpected symbol in "2b"
```

Code C 1.2

```c
int 2i;
```

```
code.c: In function ‘main’:
code.c:3:9: error: expected identifier
or ‘(’ before numeric constant
     int 2i;
         ^
Compilation échouée.
```

Code Matlab 1.3

```
2b=1
```

```
2b=1
|
Error: Unexpected MATLAB expression.
```

Code Python 1.4

```python
2b = 1
```

```
>>> 2b = 1
File "<stdin>", line 1
2b = 1
^
SyntaxError: invalid syntax
```

Remarque 1.4

Certains langages peuvent être plus laxistes et permettre des indicateurs inhabituels (voir la session 1.1, de la présente page), cela peut aller jusqu'à redéfinir des valeurs élémentaires (voir la session 1.2, de la présente page) !

Code Forth 1.1 – Exemple de syntaxe laxiste

```
variable 2b
4 2b !
2b @ 2b @ + .
```

```
variable 2b  ok
4 2b !  ok
2b @ 2b @ + . 8  ok
```

Code Forth 1.2 – Exemple de syntaxe trop laxiste

```
1 1 + .          \ Calcul usuel : 1+1=2
2 constant 1     \ Définit une constante 1, valant 2
1 1 + .          \ Maintenant : 1+1=4 !
```

```
1 1 + . 2  ok
2 constant 1  ok
1 1 + . 4  ok
```

1.2.2 Erreurs syntaxiques

Code R 1.5

```
b=(1+2;
```

```
Erreur : ';' inattendu(e) in "b=(1+2;"
```

Code C 1.6

```
i=(1+2;
```

```
code.c:4:11: error: expected ‘)’
before ‘;’ token
     i=(1+2;
           ^
Compilation échouée.
```

Code Matlab 1.7

```
b=(1+2;
```

```
b=(1+2;
      |
Error: Unbalanced or unexpected parenthesis or bracket.
```

Code Python 1.8

```
b = (1+2;
```

```
>>> b = (1+2;
File "<stdin>", line 1
b = (1+2;
        ^
SyntaxError: invalid syntax
```

Remarque 1.5

Certains logiciels peuvent corriger des erreurs syntaxiques. Voir la session TEX 1.3, de la présente page.

Terminal 1.3 – Correction d'une erreur syntaxique

```
root# cat essai.tex
$a
\end

root# pdftex -interaction=nonstopmode essai.tex
This is pdfTeX, Version 3.14159265-2.6-1.40.21 (TeX Live 2020) (preloaded format=pdftex)
 restricted \write18 enabled.
entering extended mode
(./essai.tex
! Missing $ inserted.
<inserted text>
                $
<to be read again>
                   \end
1.2 \end

[1{/usr/local/texlive/2020/texmf-var/fonts/map/pdftex/updmap/pdftex.map}] )
(see the transcript file for additional information)</usr/local/texlive/2020/te
xmf-dist/fonts/type1/public/amsfonts/cm/cmmi10.pfb></usr/local/texlive/2020/tex
mf-dist/fonts/type1/public/amsfonts/cm/cmr10.pfb>
Output written on essai.pdf (1 page, 15461 bytes).
Transcript written on essai.log.
```

1.2.3 Erreurs sémantiques

Code C 1.9

```
*p,*q;
p+q;
```

```
code.c:4:6: error: invalid operands
to binary + (have ‘int *’ and ‘int *’)
     p+q;
      ^
Compilation échouée.
```

Code Matlab 1.10

```
A=1:3;
A*A
```

```
Error using  *
Inner matrix dimensions must agree.
```

Remarque 1.6

Certains logiciels peuvent corriger des erreurs sémantiques. Voir la session C 1.1, de la présente page et les résultats (1.4, de la présente page).

Cela peut être dangereux !

Code C 1.1 – Erreur sémantique mal corrigée

```c
#include <stdio.h>

int main()
{
        int j;
        float *f;
        j=1;
        f=&j;
        printf("La valeur dans f est %f\n",*f);
        return(0);
}
```

Terminal 1.4 – Correction d'une erreur sémantique

```
root# gcc essai.c
essai.c: In function ‘main’:
essai.c:8:3: warning: assignment to ‘float *’ from incompatible pointer type ‘int *’
↪  [-Wincompatible-pointer-types]
    8 |  f=&j;
      |   ^

root# ./a.out
La valeur dans f est 0.000000
```

Remarque 1.7

Ils peuvent aussi *ne pas* corriger des erreurs sémantiques. Voir la session C 1.2, de la présente page et les résultats 1.5, page suivante et 1.6, page ci-contre.

Code C 1.2 – Erreur sémantique non corrigée

```c
#include <stdio.h>

int main()
{
        float *f;
        printf("La valeur dans f est %f\n",*f);
        return(0);
```

```
8  }
```

Terminal 1.5 – Non correction d'une erreur sémantique

```
root# gcc essai.c
root# ./a.out
Segmentation fault
```

Terminal 1.6 – Bonne utilisation des options de compilation

```
root# gcc -Wall essai.c
essai.c: In function ‘main’:
essai.c:6:37: warning: ‘f’ is used uninitialized in this function [-Wuninitialized]
    6 |  printf("La valeur dans f est %f\n",*f);
      |                                     ^~
```

Toujours demander le maximum de warning *!*

Un bon code ne doit pas produire de *warning* au moment de la compilation

Exercice(s) 1.1

Corrections

1.1.1 Dans les codes C suivant, préciser l'erreur et dire si elle est lexicale, syntaxique ou sémantique.

Code C 1.1

```c
int main()
{
        int i=1;
        printf("i=%d",i);
        return(0);
}
```

Une correction est proposée page 55.

Code C 1.2

```c
int main()
{
        int Chillès=1;
        return(2*Chillès);
}
```

Une correction est proposée page 55.

Code C 1.3

```c
#include <stdio.h>

int main()
{
        int i=1,j,k;
        k=i+j;
        printf("k=%d\n",k);
        return(0);
}
```

Une correction est proposée page 56.

Code C 1.4

```c
int main()
{
        int i=1;
        if (i==1)
                return(0)
        else
                return(1);
}
```

Une correction est proposée page 56.

Code C 1.5

```c
#include <stdio.h>

int main()
{
        int i=1;
        if ((i=1))
                printf("i=%d\n",0);
        else
                printf("i=%d\n",1);
        return(0);
}
```

Une correction est proposée page 56.

1.3 Un peu d'histoire

Trois grands langages précurseurs (les dates sont approximatives et dépendent beaucoup des sources)

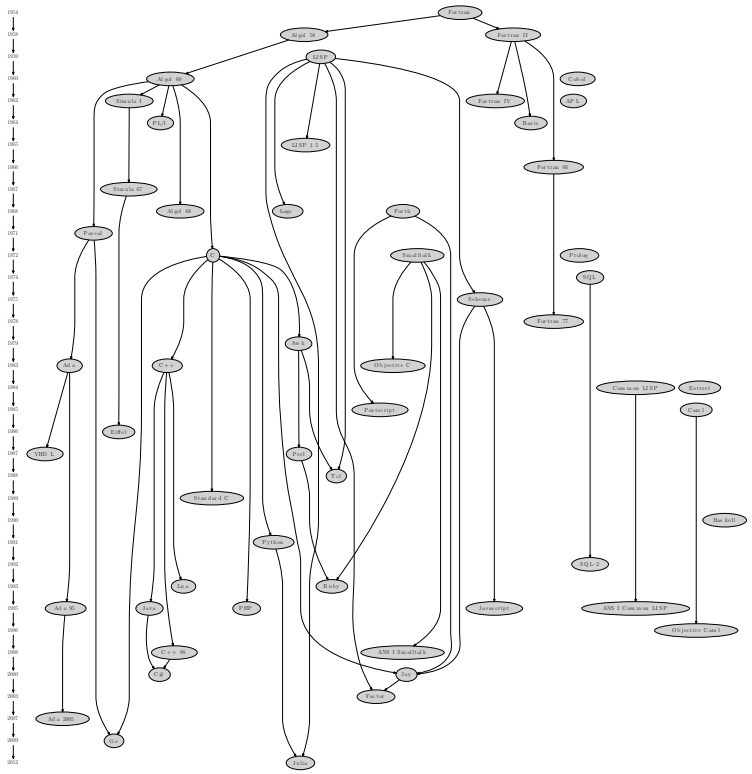
1. `Fortran` (1954 création), (1957 première version utilisable) : calculs numériques
2. `LISP` (1959) : calculs symboliques
3. `Cobol` (1960) : bases de données

Les langages novateurs :

1. `Simula` (1963-1967) : début de la POO (Programmation Orientée Objets)
2. `Forth` (1968-1969) : machine virtuelle, multi-utilisateur, semi-compilation
3. `Smalltalk` (1969-1972) : POO aboutie
4. `Prolog` (1970-1972) : déclaratif, programmation logique
5. `SQL` (1974-1977) : déclaratif, destiné aux bases de données
6. TEX (1977-1986) : langage avec balises (*tags*)

Les descendants : `C` (1972), `C++` (1983), `Postscript` (1983), `VHDL` (1987), `Python` (1991), `Java` (1995), `C#` (2002), etc. Voir les figures 1.3, de la présente page et 1.4, page suivante.

Figure 1.3 – Historique simplifié des langages



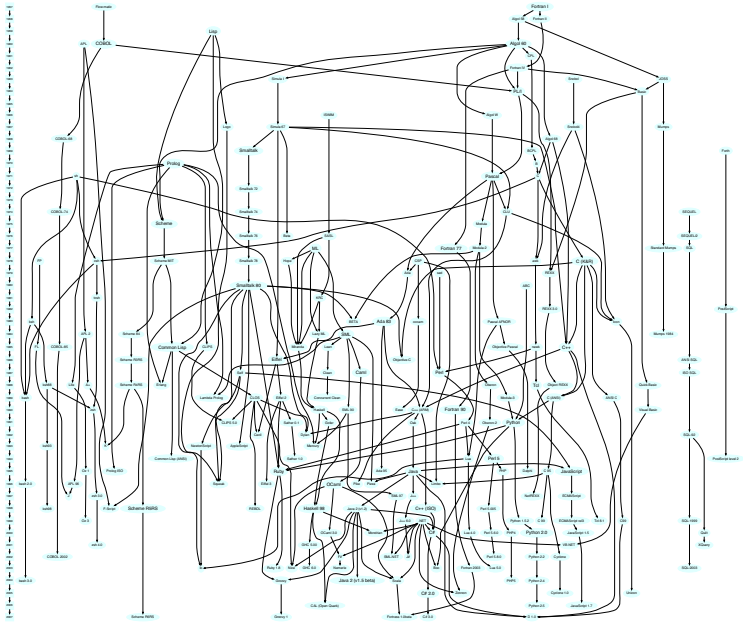
Définition 1.4 – Propriétés des langages – manière d'écrire

On distingue les langages *impératifs* (on donne une suite d'ordres), *fonctionnels* (on compose, comme en mathématiques, des fonctions) et *déclaratifs* (on décrit ce qu'on veut).

Définition 1.5 – Propriétés des langages – sémantique

On distingue les langages *séquentiels* (les instructions se déroulent dans un ordre précis), *événementiels* (le logiciel réagit aux événements extérieurs) et *parallèles ou concurrentiels* (les instructions se déroulent sur plusieurs machines ou processeurs et pour plusieurs utilisateurs)

Figure 1.4 – Historique détaillé des langages



Remarque 1.8

Pour les langages parallèles, on peut avoir des langages *synchrones* ou *asynchrones*.

Définition 1.6 – Propriétés des langages – fonctionnement interne

On distingue les langages *interprétés* (chaque instruction est immédiatement exécutée), *compilés* (le programme est transformé en un langage dédié à la machine hôte), *compilés en machine virtuelle* (le programme est transformé en un sous-langage très simple ; il faut alors un *runtime* pour l'exécuter)

Exercice(s) 1.2

1.2.1 Donner les caractéristiques des langages suivants : `C`, `C++`, `Python`, `Markdown`, LaTeX 2_ε.

Quelque soit le langage de programmation, on a toujours besoin, à partir du *code* d'effectuer les tâches décrites précédemment :

1. analyse lexicale
2. analyse syntaxique
3. analyse sémantique
4. modèle d'exécution

C'est l'objet de ce cours !

Chapitre 2

Analyse lexicale

L'objectif est le suivant : étant donné un texte (*code*), le séparer en *lexèmes* (mots du langage). Pour cela, nous allons disposer de deux outils très différents

1. les expressions régulières (ou rationnelles), avec la méthode de reconnaissance de motifs (*pattern matching*)
2. les automates finis, ou le texte est lu caractère par caractère.

Exemple 2.1

Soit le texte proposé dans le cadre 2.1, de la présente page. Comment trouver les dates ? Les adresses mél ? Les mots commençant par une majuscule ? Les lignes vides ? Les lignes se terminant par des espaces ? etc.

Texte 2.1 – À lire

```
Paris, le 23 septembre 2021

Bonjour Alain,
tu trouveras l'information que tu souhaites sur http://www.sjtu.edu.cn.

Tu pourras me contacter à l'adresse alain.chilles@sjtu.edu.cn

Ne te trompe pas comme la dernière fois !
Ce n'est pas alain.chilles@sjtu ni alain.chilles@sjtu.cn..

Nous travaillerons à la conversion des codes wxMaxima en Python (Sympy). Pour 90%
d'entre eux, il n'y aura pas de problème particulier...

Bonne journée :-) $$^
```

2.1 Expressions régulières (ou rationnelles)

Remarque 2.1

1. Plusieurs « normes » : `POSIX`, `PCRE`...
2. Supportées par plusieurs langages informatiques (`Java`, `Python`, `R`, `Matlab`, etc.)
3. Supportées par quelques outils (`grep`, `sed`, beaucoup d'éditeurs, etc.)
4. Utilisables en `C` avec la commande `#include <regex.h>`

Nous nous limiterons à un sous-ensemble de la norme `POSIX` étendue.

2.1.1 Utilisation de l'outil `grep`

`grep` est un outil `Linux` qui existe dans les autres systèmes (peut-être faut-il l'installer ?). Les options qui vont nous intéresser sont (voir la session 2.1, de la présente page)

- l'option `-E` qui permet la recherche d'un motif dans un fichier (c'est la norme `POSIX` Étendue)
- l'option `-n` qui permet d'avoir les numéros des lignes où on trouve les motifs demandés
- l'option `-o` qui permet de ne produire que le motif et non la ligne qui le contient

La syntaxe générale de `grep` est `grep -E<options> <expression régulière> <nom du fichier>`

L'objet de ce paragraphe est d'apprendre à écrire des expressions régulières.

Terminal 2.1 – Recherche de motif avec `grep` : les options

```
root# grep -E "Alain" TexteALire.txt
Bonjour Alain,

root# grep -En "Alain" TexteALire.txt
3:Bonjour Alain,

root# grep -Eo "Alain" TexteALire.txt
Alain

root# grep -Eno "Alain" TexteALire.txt
3:Alain
```

Définition 2.1 – Expressions régulières

Nous appelerons *expression régulière* (ou *expression rationnelle*) toute chaîne de caractères permettant de décrire un motif en satisfaisant la « norme » `POSIX` étendue. Un résumé de la syntaxe est fourni dans le tableau suivant.

Syntaxe des expressions régulières

Caractère	Type du caractère	Interprétation	Exemple	Voir
`c`	non spécial	`c`	Alain	2.1
\	spécial	caractère d'échappement	\\	2.2
`^`	spécial	1. en dehors d'un crochet : début de ligne 2. dans un crochet : tout sauf	1. `^$` 2. `[^a-z]`	2.3
$	spécial	en dehors d'un crochet : fin de ligne	`\.$`	2.4
.	spécial	n'importe quel caractère	Bon..	2.5

(|)	spécial	ou (disjonction)	`(a|b)`	2.6
?	spécial	itérateur : 0 ou 1	`a?`	2.7
*	spécial	itérateur : un nombre quelconque	`ala*`	2.8
+	spécial	itérateur : au moins un	`[$]+`	2.9
[]	spécial	disjonction de tout ce qui est entre crochets ; permet de faire des intervalles, des complémentaires et des échappements	1. `[aeiouy]` (les voyelles) 2. `[a-z]` (les minuscules) 3. `[^a-z]` (tout sauf les minuscules)	2.3 2.4 2.7
{}	spécial	itérateur : on peut préciser le minimum et le maximum	`{2,4}` `{,4}` `{2,}` `{3}`	2.10
\b	spécial	bord de mot	`\b[A-Z]`	2.11

Terminal 2.2 – Expression régulière : caractère d'échappement

```
root# grep -Eno "\^" TexteALire.txt
13:^
```

Terminal 2.3 – Expression régulière : caractère ^

```
root# grep -Eno "[^a-zA-Z0-9 ,.:'$\@/-\!()]" TexteALire.txt
11:%
13:-
13:^

root# grep -Eno "[^a-zA-Z0-9 ,.:'$\@/-!()]" TexteALire.txt $
bash: !: event not found
```

Remarque 2.2

Notons que les caractères `!` et `@` sont des caractères spéciaux... pour `bash`.

Terminal 2.4 – Expression régulière : caractère $

```
root# grep -Eno "  $" TexteALire.txt
6:
7:

root# grep -Eno "[$]" TexteALire.txt
13:$
13:$
```

Terminal 2.5 – Expression régulière : caractère . (point)

```
root# grep -Eno "Bon.." TexteALire.txt
3:Bonjo
13:Bonne

root# grep -Eno ":.) ." TexteALire.txt
13::-) $ $
```

Terminal 2.6 – Expression régulière : ou

```
root# grep -Eno "(s|p)ou" TexteALire.txt
4:sou
6:pou
```

Remarque 2.3

Les parenthèses servent aussi à grouper une expression régulière devant un itérateur (voir la session 2.10, de la présente page).

Terminal 2.7 – Expression régulière : itérateur ?

```
root# grep -Eno "[.][.][.]?" TexteALire.txt
9:..
12:...
```

Terminal 2.8 – Expression régulière : itérateur *

```
root# grep -Eno ":/*" TexteALire.txt
4::/
14::
```

Terminal 2.9 – Expression régulière : itérateur +

```
root# grep -Eno ":/+" TexteALire.txt
4::/
```

Terminal 2.10 – Expression régulière : itérateur {,}

```
root# grep -Eno "/{1,2}" TexteALire.txt
4:/

root# grep -Eno " {2,}" TexteALire.txt
6:
7:

root# grep -Eno " {3,}" TexteALire.txt
7:

root# grep -Eno "^([^t]*t.*){3,}" TexteALire.txt
4:tu trouveras l'information que tu souhaites sur http://www.sjtu.edu.cn.
6:Tu pourras me contacter à l'adresse alain.chilles@sjtu.edu.cn
9:Ce n'est pas alain.chilles@sjtu ni alain.chilles@sjtu.cn..

root# grep -Eno "^([^t]*t.*){3}" TexteALire.txt
```

```
4:tu trouveras l'information que tu souhaites sur http://www.sjtu.edu.cn.
6:Tu pourras me contacter à l'adresse alain.chilles@sjtu.edu.cn
9:Ce n'est pas alain.chilles@sjtu ni alain.chilles@sjtu.cn..

root# grep -Eno "^([^t]*t[^t]*){3}" TexteALire.txt
4:tu trouveras l'information que
6:Tu pourras me contacter à l'adresse alain.chilles@sjtu.edu.cn
9:Ce n'est pas alain.chilles@sjtu ni alain.chilles@sjtu.cn..
```

Remarque 2.4

On peut préciser le nombre d'itération de la manière suivante :

$$\begin{aligned} \{x\} &= \text{exactement } x \\ \{x,\} &= \text{au moins } x \\ \{,x\} &= \text{au plus } x \\ \{x,y\} &= \text{au moins } x \text{ et au plus } y \end{aligned}$$

Terminal 2.11 – Expression régulière : limite d'un mot

```
root# grep -Eno "\b[A-Z][a-z]*" TexteALire.txt
1:Paris
3:Bonjour
3:Alain
6:Tu
8:Ne
9:Ce
11:Nous
11:Python
11:Sympy
11:Pour
14:Bonne
```

Exercice(s) 2.1

Corrections

2.1.1 Que veulent dire les expressions rationnelles suivantes ? Et qu'obtient-on sur le fichier `TexteALire.txt` ?

```
^[A-Z]
␣$
^$
([^o]+o){2,}
[-+]?[0-9]*\.?[0-9]*
```

2.1.2 Produire une expression rationnelle permettant (avec l'outil `grep` ou sous un éditeur) de trouver les motifs suivants :

- Les mots contenant une majuscule.
- Les lignes qui ne sont pas vides et qui ne se terminent pas par un caractère blanc
- Les lignes vides et celles qui ne se terminent pas par un blanc
- Les dates.
- Les adresses mél.
- Les adresses Internet.
- Les nombres de type `float`.
- Les mots de 9 lettres minuscules.
- Faire l'exercice 3.3.5, page 125 de [1]. (Énoncé ci-après.)

Exercise 3.3.5: *Write regular expressions to match the following words (if possible, of course!):*

(a) All strings of lowercase letters that contains the five vowels in order.

(b) All strings of lowercase letters in which the letters are in ascending lexicographic order.

(c) Comments, consisting of a string surrounded by `/*` *and* `*/`*, whithout an intervening* `*/`*, unless it is inside double-quotes (*`"`*).*

(d) All strings of digits with no repeated digits. Hint: *Try this problem first with a few digits, such as* $\{0, 1, 2\}$*.*

(e) All strings of digits with at most one repeated digits.

(f) All strings of a's and b's with an even number of a's and an odd number of b's.

(g) All strings of a's and b's that do not contain the substring abb.

(h) All strings of a's and b's that do not contain the subsequence abb.

2.1.2 Utilisation de la bibliothèque `regex`

Rappel 2.1 Lecture sécurisée

En `C`, la fonction `scanf` n'est pas sécurisée. Heureusement, on peut utiliser un format de lecture qui protège la lecture. Voir le session 2.1, de la présente page.

Code `C` 2.1 – Lecture sécurisée

```c
#include <stdio.h>

int main()
{
    char s[100];
    printf("-> ");
    scanf("%5[^\n]", s);
    printf("J'ai lu : %s",s);
    return(0);
}
```

Terminal 2.12 – Lecture sécurisée

```
root# gcc -Wall essai.c

root# ./a.out
-> Formidable !
J'ai lu : Formi
```

On sécurisera systématiquement les utilisations de `scanf` *!*
Ne pas le faire est très dangereux !

Rappel 2.2 Écriture d'une sous-chaîne

On a souvent aussi besoin d'écrire une partie d'une chaîne de caractère *sans pour autant* extraire la sous-chaîne. Les options de la fonction `printf` permettent de faire ça simplement. Voir la session 2.2, de la présente page.

Code C 2.2 – Écriture d'une sous-chaîne

```c
#include <stdio.h>

int main()
{
    char s[] = "abcdefghijklmn";
    printf("%.*s\n", 7, s+4);
    return(0);
}
```

Terminal 2.13 – Écriture d'une sous-chaîne

```
root# gcc -Wall essai.c

root# ./a.out
efghijk
```

Notation 2.1 – Fonctions de la bibliothèque `regex`

La bibliothèque `regex` contient (entre autres) les définitions des objets suivants

1. Le type d'un expression rationnelle `regex_t`
2. Le type du résultat `regmatch_t` (avec 2 champs `rm_so` – début du filtrage et `rm_eo` – fin du filtrage)
3. Une fonction de compilation d'une expression rationnelle (et qui en vérifie donc la validité) `regcomp`
4. Une fonction d'exécution du filtrage d'une chaîne de caractères donnée par une expression rationnelle `regexec`
5. Des options pour préciser la norme choisie et son comportement `REG_EXTENDED` qui donne accès à la norme `POSIX` étendue et `REG_NEWLINE` qui fixe le comportement par rapport aux sauts de ligne (le saut de ligne est alors un caractère reconnue par l'expression régulière).
6. Une gestion permettant de comprendre les erreurs dans la compilation d'une expression rationnelle `regerror`
7. Une fonction de libération de la mémoire `regfree`

Texte 2.2 – Partie de `regex.h`

```
extern int regcomp (regex_t *__restrict __preg, const char *__restrict __pattern, int
    ↪  __cflags);
extern int regexec (const regex_t *__restrict __preg, const char *__restrict __string,
    ↪  size_t __nmatch, regmatch_t __pmatch[__restrict_arr], int __eflags);
extern size_t regerror (int __errcode, const regex_t *__restrict __preg, char
    ↪  *__restrict __errbuf, size_t __errbuf_size);
extern void regfree (regex_t *__preg);
```

Remarque 2.5

L'utilisation de la bibliothèque s'effectue de la manière suivante

1. compilation de l'expression régulière (voir le code 2.3, de la présente page) ;
2. exécution de l'expression régulière compilée sur le texte (voir le code 2.4, de la présente page) ;
3. lecture du ou des résultats (voir le code 2.5, de la présente page).

Un exemple d'exécution est donnée à la session 2.14, de la présente page.

Code C 2.3 – Compilation de l'expression régulière

```c
// L'expression régulière a été définie dans la chaîne expr
        regex_t myregexp;
        if (regcomp(&myregexp, expr, REG_EXTENDED|REG_NEWLINE))
            {
                printf("Mauvaise expression rationnelle : %s", expr);
                regfree(&myregexp);  // Toujours nettoyer la mémoire avant de sortir
                return(2);           // Code d'erreur
            };
```

Code C 2.4 – Exécution de l'expression régulière compilé

```c
// p est un pointeur sur le début du texte à lire (qui va changer au cours de la lecture)
// pmatch est un tableau contenant les résultats (de type regmatch_t)
                res = regexec(&myregexp, p, 1, pmatch, 0);
```

Code C 2.5 – Lecture des résultats du *pattern matching*

```c
// debut et fin sont les indices du texte où est l'expression reconnue, à partir de l'indice p
                debut = pmatch[0].rm_so;
                fin = pmatch[0].rm_eo;
```

Terminal 2.14 – Exemple d'utilisation de `regex`

```
root# gcc -Wall essai.c

root# ./a.out
Donner l'expression régulière : [a-zéè祁冲]+
Donner le texte à reconnaître :  Ce cours est donné par Alain Chillès (祁冲 en chinois)
e (2, 2)
cours (4, 8)
est (10, 12)
donné (14, 19)
par (21, 23)
lain (26, 29)
hillès (32, 38)
祁冲 (41, 46)
en (48, 49)
chinois (51, 57)
```

Remarque importante 2.6

Les caractères accentués occidentaux sont de longueur 2, les caractères chinois sont de longueur 3.

Exercice(s) 2.2

Corrections

2.2.1 Compléter le code pour reproduire le comportement donné à la session 2.14, page ci-contre.

2.2.2 Réécrire un `grep` prenant les options `-n` et `-o`.

2.2 Automates finis

2.2.1 Notion de langage

Définition 2.2 – Langage

1. On appelle *alphabet* un ensemble *fini* de lettres et de symboles.
2. On appelle *mot* ou *chaîne* une succession *finie* d'éléments de l'alphabet.
3. On appelle *langage* tout ensemble (fini ou infini) constitué de mots.

Exemple 2.2

1. *Alphabet binaire* : $\{0, 1\}$.
2. *Alphabet français* : {a, …, z, A, …, Z, é, É, è, È, ç, Ç, à, À, â, Â, ä, Ä, ê, Ê, ë, Ë, ŷ, Ŷ, ÿ, Ÿ, û, Û, ü, Ü, î, Î, ï, Ï, ô, Ô, ö, Ö, œ, Œ, æ, Æ, ù, Ù…}
3. *Langage des mots sur 64 bits* : reconnu par `(0|1){64}`

Notation 2.2 – Opérations sur les langages

Si L et M sont deux langages.

— *Union* : $L \cup M$

— *Concaténation* : LM, où

$$LM = \{st,\ s \in L,\ t \in M\}$$

Notation $L^0 = \{\varepsilon\}$ (où ε est le mot vide) et

$$\forall k \in \mathbb{N},\ L^{k+1} = LL^k = L^kL$$

— *Fermeture de Kleene de* L : L^*, où

$$L^* = \bigcup_{k \in \mathbb{N}} L^k$$

— *Fermeture positive de* L : L^+, où

$$L^+ = \bigcup_{k \in \mathbb{N}^*} L^k = LL^* = L^*L$$

Remarque importante 2.7

Si on note Σ l'alphabet utilisé, on a alors

1. l'ensemble des mots possibles est Σ^* ;

2. un langage L construit sur l'alphabet Σ est donc un *sous-ensemble* de Σ^* ($L \subset \Sigma^*$).

Exemple 2.3

Si L l'ensemble des lettres est reconnu par `[a-zA-Z]` et C l'ensemble des chiffres est reconnu par `[0-9]`, alors

- $L \cup C$ est reconnu par `[a-zA-Z0-9]`.
- LC est reconnu par `[a-zA-Z][0-9]`.
- L^* est reconnu par `[a-zA-Z]*`
- C^+ est reconnu par `[0-9]+`.

Définition 2.3 – Langage rationnel

Tout langage reconnaissable par une expression régulière (ou rationnelle) est appelé *langage rationnel*.

Exemple 2.4 – Langages rationnels ? ou pas ?

- L'ensemble des mots de k éléments de l'alphabet est un langage rationnel.
- L'ensemble des lexèmes d'un langage informatique est un langage rationnel.
- Si l'alphabet est $\{a, b\}$, le langage
$$\{a^n b^n,\ n \in \mathbb{N}\}$$
n'est pas un langage rationnel !
- L'ensemble des expressions *bien parenthésées* (autant de parenthèses ouvrantes que fermantes, et dans le bon ordre) *n'est pas* un langage rationnel !

Exercice(s) 2.3

Corrections

2.3.1 **(Exercice 3.3.2, page 125 de [1])** Décrire les langages reconnus par les expressions rationnelles suivantes :

(a) `a(a|b)*a`

(b) `(a?b*)*`

(c) `(a|b)*a(a|b)(a|b)`

(d) `a*ba*ba*ba*`

(e) `(aa|bb)*((ab|ba)(aa|bb)*(ab|ba)(aa|bb)*)*`

2.3.2 **(Exercice 3.3.5, pages 125-126 de [1])** Les langages suivants sont-ils rationnels ? Si oui, proposer une expression rationnelle pour les reconnaître.

(a) Une chaîne de caractères constituée de minuscules et qui contient les 5 voyelles dans l'ordre. ($a \to e \to i \to o \to u$)

(b) Une chaîne de caractères constituée de minuscules dont les lettres vont strictement croissantes. (Comme dans `bip`, car $b < i < p$)

(c) Un commentaire en langage `C`, il commence par `/*` et finit par `*/`, il ne doit pas y avoir un `*/` à l'intérieur, à moins d'être compris entre deux guillemets `"`.

(d) Une chaîne de caractères constituée de chiffres, dont deux chiffres consécutifs sont différents. (2016 est bon, mais 2005 ne l'est pas)

(e) Une chaîne de caractères constituée de 'a' et de 'b' et contenant un nombre impair de 'a' et un nombre pair de 'b'.

(f) Une chaîne de caractères constituée de 'a' et de 'b' et qui ne contient pas la sous-chaîne "abb".

2.2.2 Représentation graphique

Définition 2.4 – Automates finis

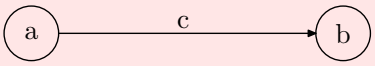
Étant donné un alphabet Σ sur lequel on à construit un langage L, on appelle *automate* reconnaissant les mots du langage L, un ensemble constitué d'*états* et de *transitions* comportant

— Un *état initial*, noté : 

— Des *états d'acceptation*, notés : 

— Des *états intermédiaires*, notés : 

Un mot du langage est consommé caractère par caractère et permet à travers les transitions de changer d'état. Au départ, l'automate est dans l'état initial. À la fin, si l'automate est dans un état d'acceptation, le mot est reconnu, sinon le mot n'est pas dans le langage. Une transition déclenchée par le caractère c est notée



Exemple 2.5 – Exemple de fonctionnement

Lecture de la chaîne de caractère *abbab* (qu'on lit de gauche à droite) par l'automate de la figure 2.1, page suivante.

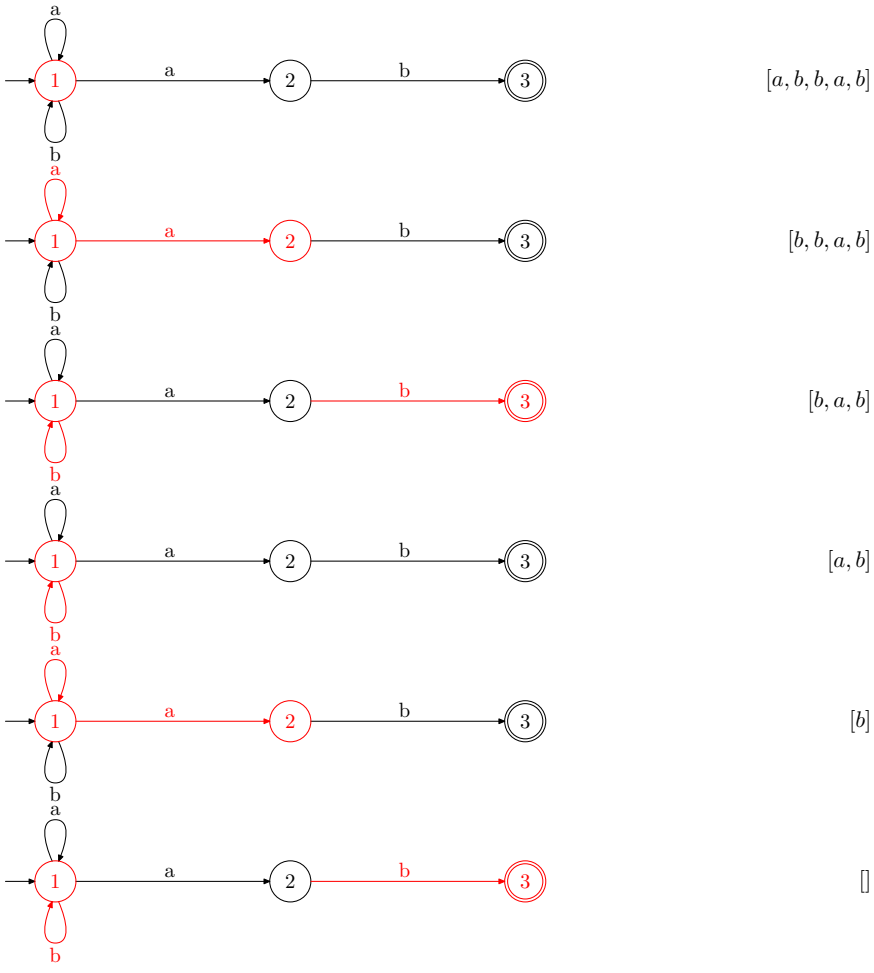
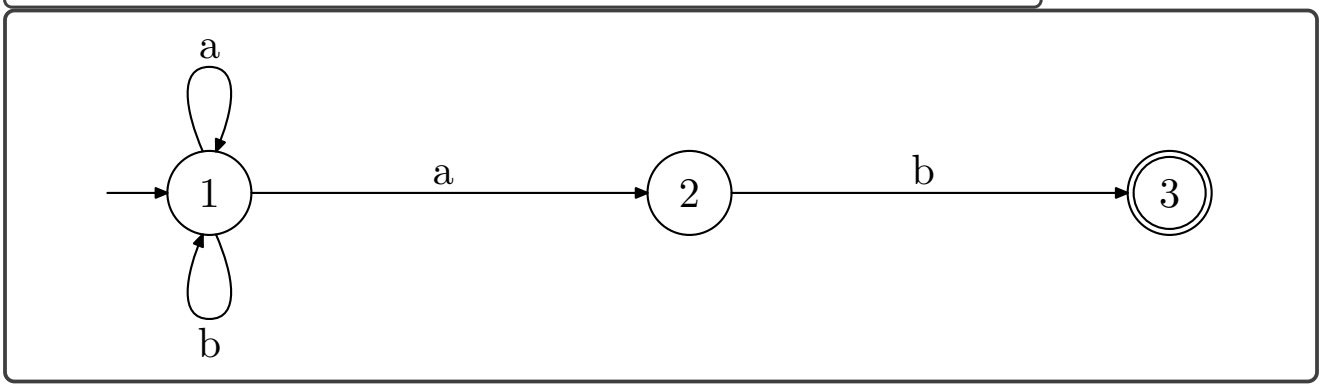
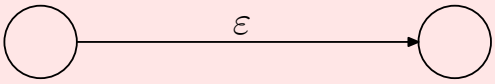


Figure 2.1 – Automate non déterministe équivalent à l'expression régulière `(a|b)*ab`



Définition 2.5 – ε-transition

On appelle *ε-transition* une transition qui se déclenche sans consommation d'un caractère du flux d'entrée. (Elle se déclenche tout le temps). Elle est notée :



Définition 2.6 – Automates déterministes et non-déterministes

On appelle *automate fini non déterministe (AFN)* tout automate possédant au moins un état d'acceptation et peut-être des ε-transitions.
On appelle *automate fini déterministe (AFD)* tout AFN ne possédant pas d'ε-transition et tel qu'au plus une transition puisse être déclenchée à tout moment pour un caractère donné.

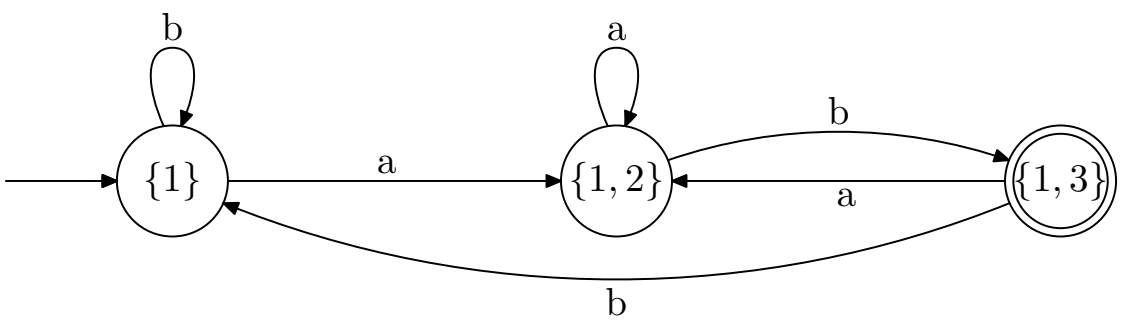
Théorème 2.1 – Équivalence des AFN et des AFD

Tout AFN peut-être transformé en un AFD équivalent (qui reconnaît le même langage).

Démonstration

À chaque instant l'AFN a un certain nombre d'états actifs, ceci nous définit un AFD où chaque état représente l'ensemble des états actifs en même temps de l'AFN.

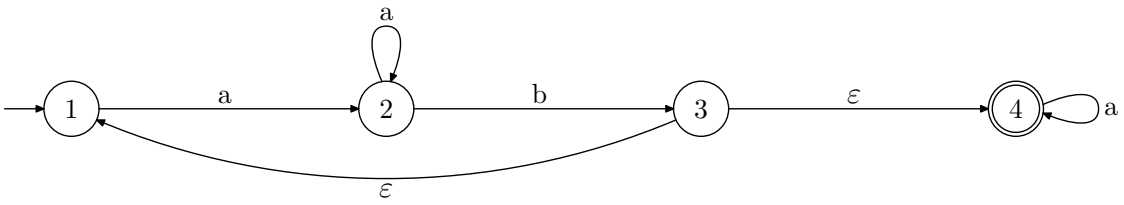
Exemple 2.6 – AFD équivalent à l'AFN de figure 2.1, de la présente page



Exercice(s) 2.4

Correction

2.4.1 Transformer l'AFN suivant en AFD. Quel langage reconnaît cet automate ?



2.3 Expressions régulières ↔ automates

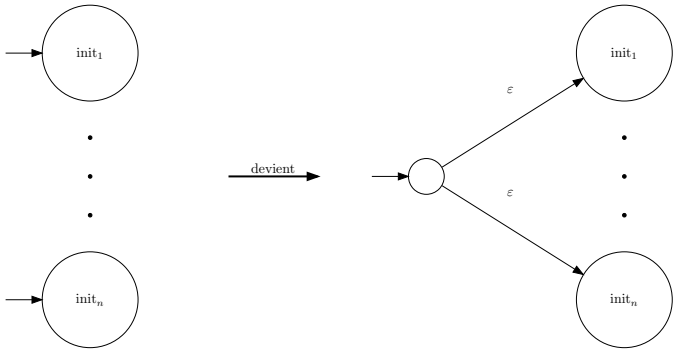
Suivant les situations, il sera utile d'avoir la représentation du langage traité (dans ce cours, celui que nous cherchons à compiler et dont nous devons faire l'analyse lexicale) dans un format plutôt qu'un autre. Il est donc *indispensable* de savoir passer d'un format à un autre.

2.3.1 Quelques remarques sur les automates

Propriété 2.1 – États initiaux

On peut se limiter à un seul état initial ! Voir la figure 2.2, de la présente page.

Figure 2.2 – Un seul état initial



Propriété 2.2 – États d'acceptation

On peut se limiter à un seul état d'acceptation ! Voir la figure 2.3, page suivante.

En conséquence, nous représenterons un automate sous la forme d'un rectangle, avec un état initial et un état d'acceptation, comme sur la figure 2.4, page suivante. Nous omettrons les ε sur les transitions (toute transition sans label sera une ε-transition !)

Figure 2.3 – Un seul état d'acceptation

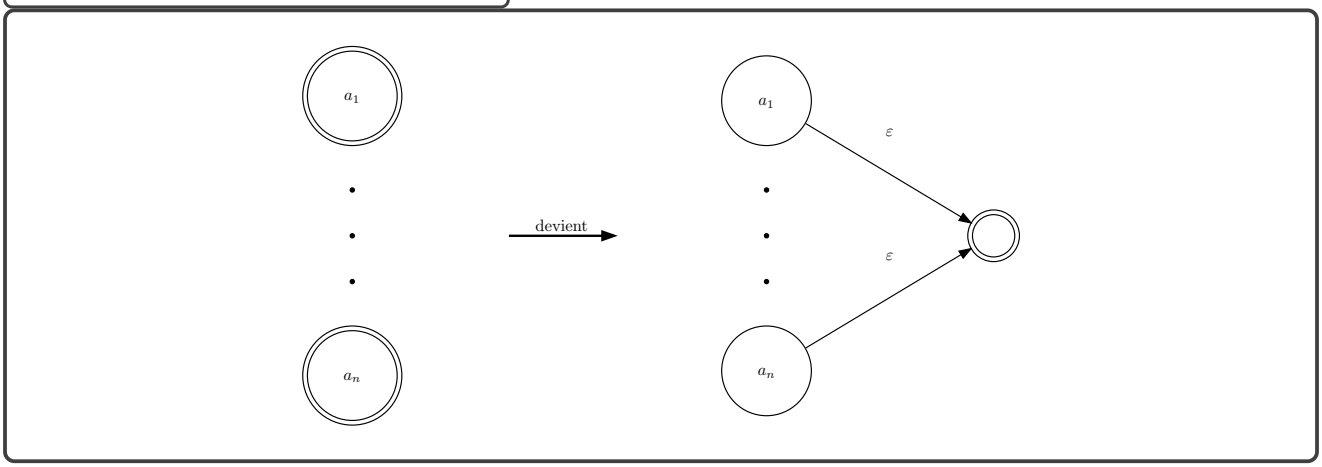
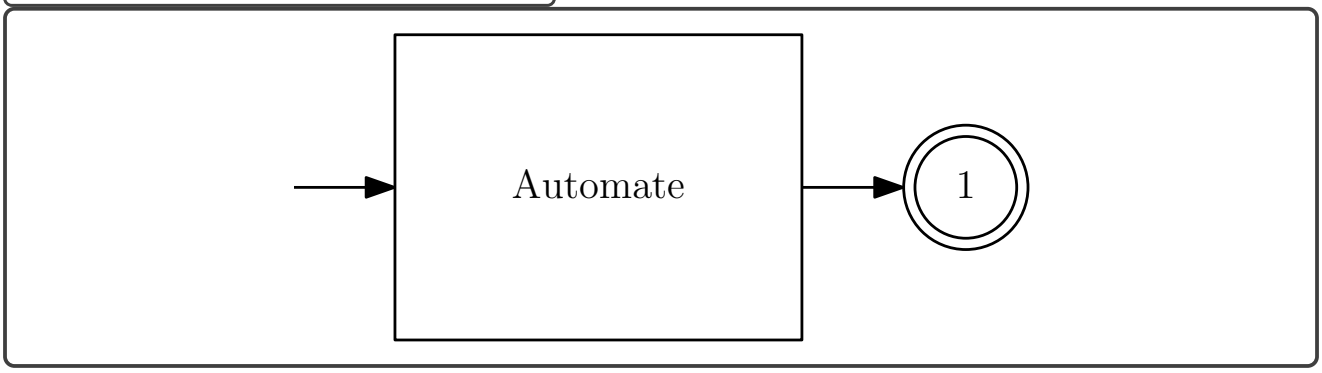


Figure 2.4 – Représentation d'un automate



2.3.2 Expressions régulières → automates

Pour produire un automate à partir d'une expression régulière (ou d'un langage rationnel), il suffit de suivre les étapes suivantes :

1. pour chaque sous-expression, produire un automate équivalent ;
2. assembler les automates avec les opérations des langages (union : figure 2.5, page ci-contre, concaténation : figure 2.6, page suivante, fermeture de Kleene : figure 2.7, page ci-contre, fermeture positive : figure 2.8, page 40) ; on obtient un AFN ;
3. transformer l'AFN en AFD.

Exercice(s) 2.5

2.5.1 Appliquer les algorithmes donnés pour trouver un AFD qui reconnaît le même langage que les expressions rationnelles suivantes :
- `a*ba*ba*ba*`
- `([-+]?[0-9]+\.?[0-9]*|[-+]?\.[0-9]+)`

Figure 2.5 – Union de deux automates

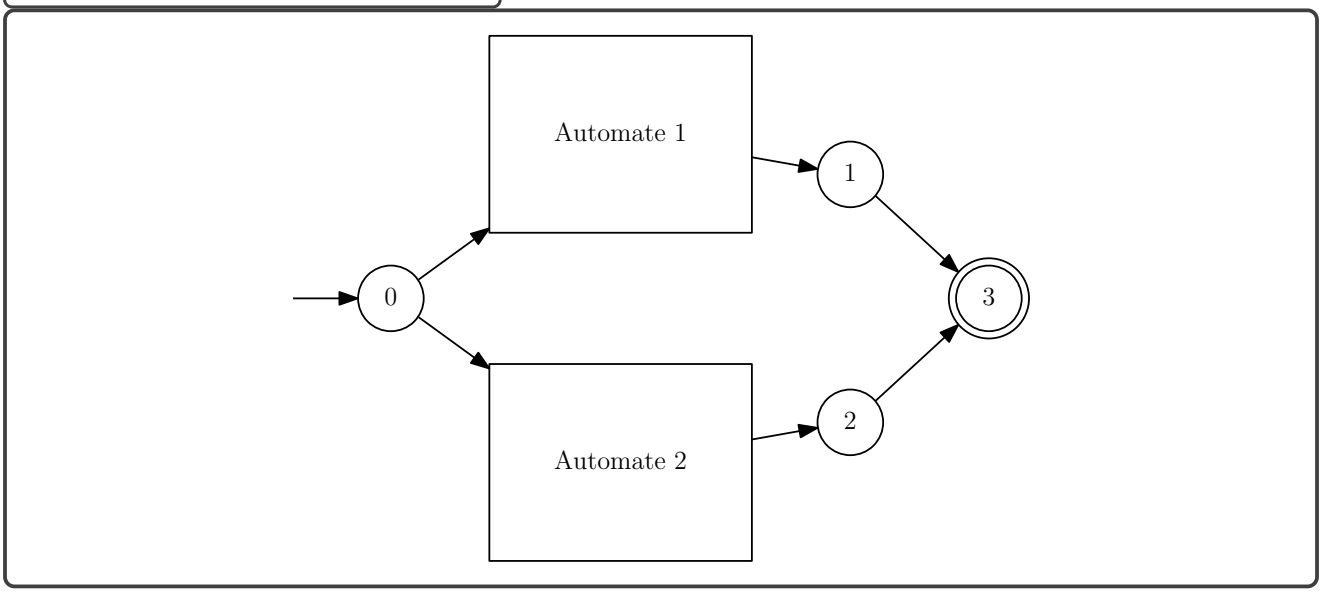
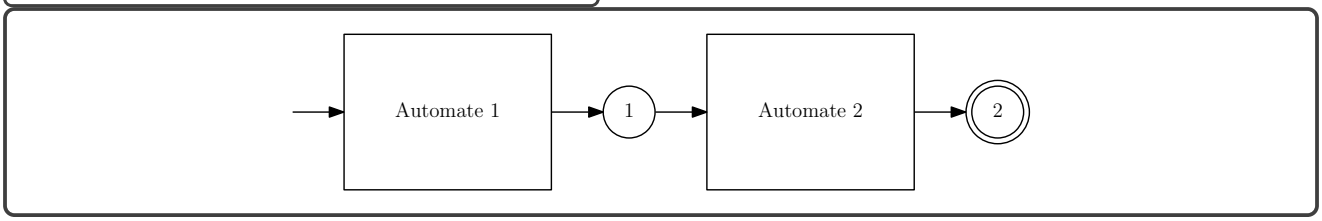


Figure 2.6 – Concaténation de deux automates



2.3.3 Automates → expressions régulières

Trois méthodes :

1. réduire pas à pas le nombre d'états (raisonnement itératif) ;
2. utiliser l'algorithme de Gauss – Arden (raisonnement ensembliste) ;
3. utiliser l'algorithme de Mac Naughton – Yamada (raisonnement récursif).

Figure 2.7 – Fermeture de Kleene d'un automate

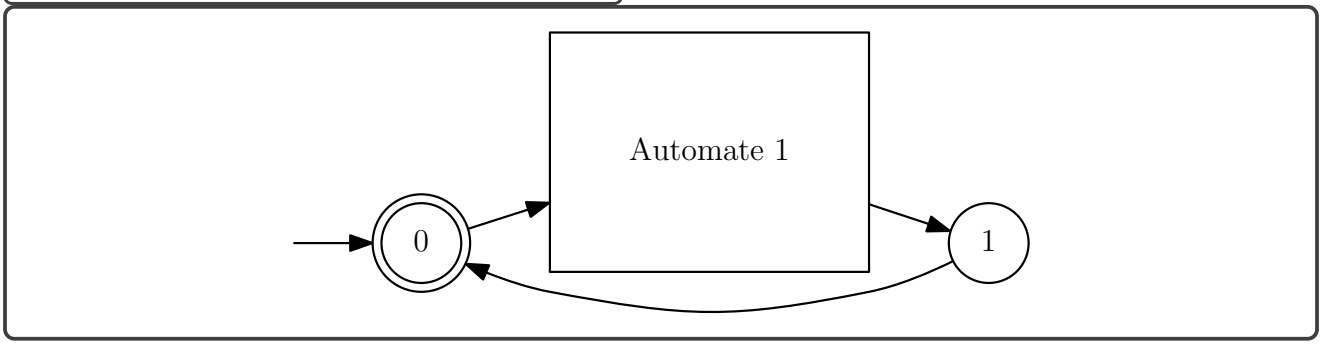
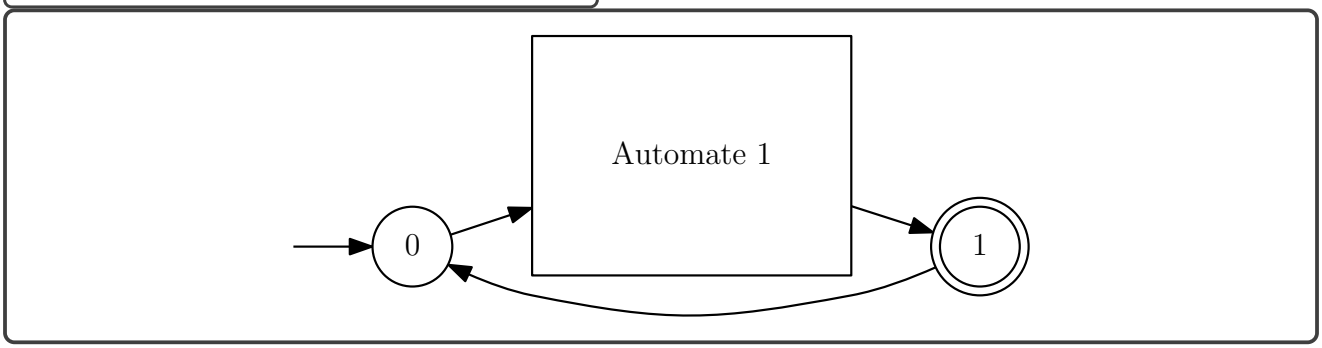


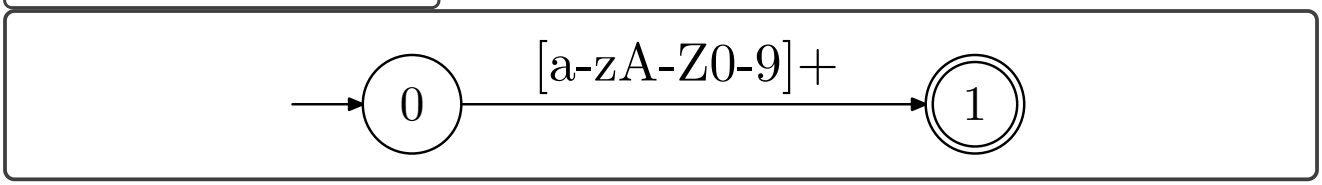
Figure 2.8 – Fermeture positive d'un automate



Notation 2.3 – Automate généralisé

On appelle *automate généralisé* un automate fini (AFN ou AFD) où les transitions peuvent être déclenchées par une expression régulière. Voir la figure 2.9, de la présente page.

Figure 2.9 – Automate généralisé



2.3.3.1 Méthode de suppression progressive des états

Si on veut supprimer l'état e, sur les transitions entre les états e_i et e_j, on écrit les expressions régulières traduisant :

1. on va de e_i à e_j *sans* passer par e (ancienne expression régulière présente sur la transition) ;

   ou

2. on va de e_i à e_j *en passant* par e (concaténation des anciennes expressions régulières de e_i à e et de e à e_j) (ne pas oublier les potentiels boucles sur l'état e).

Exemple 2.7 – Suppression progressive des états

Partant de l'automate de la figure 2.10, page suivante. On suit alors les étapes suivantes :

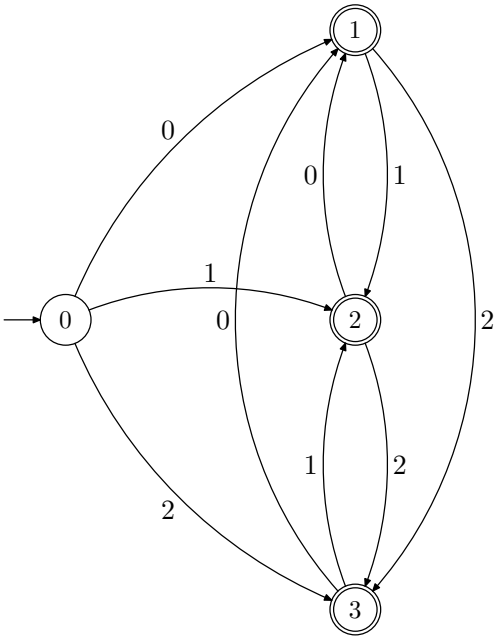
1. normalisation – un seul état initial, un seul état final (voir la figure 2.11, page 42) ;
2. suppression de l'état 3 (voir la figure 2.12, page 43) ;
3. suppression de l'état 2 (voir la figure 2.13, page 43) ;
4. suppression de l'état 1 et obtention de l'expression régulière cherchée (voir la figure 2.14, page 43). [a]

a. *Le résultat n'est pas garanti ! Comprendre la méthode !*

2.3.3.2 Méthode de Gauss – Arden

La méthode s'appuie sur la résolution d'un système linéaire (d'où le nom de Gauss) formulé dans le monde des ensembles. C'est le théorème d'Arden (voir le théorème 2.2, page ci-contre).

Figure 2.10 – Situation initiale de l'automate



Théorème 2.2 – Arden

Soit A et B deux langages tels que $\varepsilon \notin A$, alors le seul langage X qui vérifie

$$X = A.X \cup B$$

est

$$X = A^*.B$$

Démonstration

À faire vous-même !

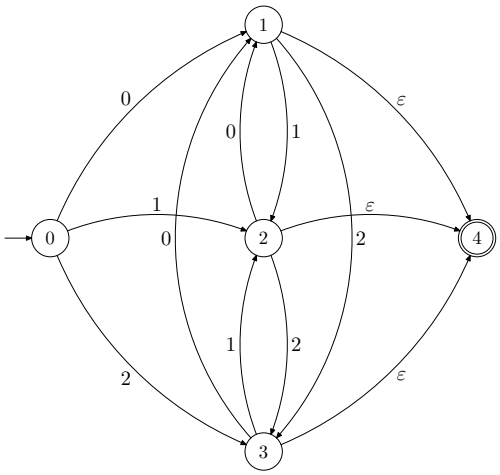
Système d'équations obtenu de la manière suivante :

1. chaque état correspond à une variable, qui est un nom de langage (L_i est le langage défini par l'ensemble des mots reconnus *à partir* de l'état e_i) ;
2. on décrit le langage L_i par une équation du genre

$$\begin{cases} L_i = \bigcup_j \{a_{i,j}\}.L_j & \text{si} \quad e_i \xrightarrow{a_{i,j}} e_j \\ L_i = \left(\bigcup_j \{a_{i,j}\}.L_j\right) \cup \{\epsilon\} & \text{si} \quad (e_i) \xrightarrow{a_{i,j}} e_j \end{cases}$$

3. on résout le système obtenu, avec un algorithme de Gauss et le théorème d'Arden ;
4. *le résultat obtenu pour l'état initial correspond à l'expression rationnelle cherchée.*

Figure 2.11 – Normalisation de l'automate



Exemple 2.8 – Utilisation du théorème d'Arden

Reprenons le cas traité dans l'exemple 2.7, page 40.
Le système obtenu est :

$$\begin{aligned} L_0 &= \{0\}.L_1 \cup \{1\}.L_2 \cup \{2\}.L_3 \\ L_1 &= \{1\}.L_2 \cup \{2\}.L_3 \cup \{\varepsilon\} \\ L_2 &= \{0\}.L_1 \cup \{2\}.L_3 \cup \{\varepsilon\} \\ L_3 &= \{0\}.L_1 \cup \{1\}.L_2 \cup \{\varepsilon\}. \end{aligned}$$

Première étape

En reportant la deuxième équation dans la troisième et la quatrième, on obtient :

$$\begin{aligned} L_0 &= \{0\}.L_1 \cup \{1\}.L_2 \cup \{2\}.L_3 \\ L_1 &= \{0\}.L_2 \cup \{2\}.L_3 \cup \{\varepsilon\} \\ L_2 &= \{01\}.L_2 \cup \{02\}.L_3 \cup \{0\} \cup \{2\}.L_3 \cup \{\varepsilon\} \\ L_3 &= \{01\}.L_2 \cup \{02\}.L_3 \cup \{0\} \cup \{1\}.L_2 \cup \{\varepsilon\}. \end{aligned}$$

La troisième équation se résout avec le théorème d'Arden, on obtient :

$$L_2 = \{01\}^*.(\{02\}.L_3 \cup \{0\} \cup \{2\}.L_3 \cup \{\varepsilon\})$$

Deuxième étape

En reportant le résultat dans la quatrième, on obtient :

$$\begin{aligned} L_0 &= \{0\}.L_1 \cup \{1\}.L_2 \cup \{2\}.L_3 \\ L_1 &= \{0\}.L_2 \cup \{2\}.L_3 \cup \{\varepsilon\} \\ L_2 &= \{01\}^*.(\{02\}.L_3 \cup \{0\} \cup \{2\}.L_3 \cup \{\varepsilon\}) \\ L_3 &= \{1,01\}.\Big(\{01\}^*.(\{02\}.L_3 \cup \{0\} \cup \{2\}.L_3 \cup \{\varepsilon\})\Big) \\ &\quad \cup \{02\}.L_3 \cup \{0\} \cup \{\varepsilon\}. \end{aligned}$$

La quatrième équation se résout avec le théorème d'Arden, on obtient :

$$L_3 = (\{1,01\}.\{01\}^*.\{02\} \cup \{1,01\}.\{01\}^*.\{2\} \cup \{02\})^*.$$

Figure 2.12 – Suppression de l'état 3

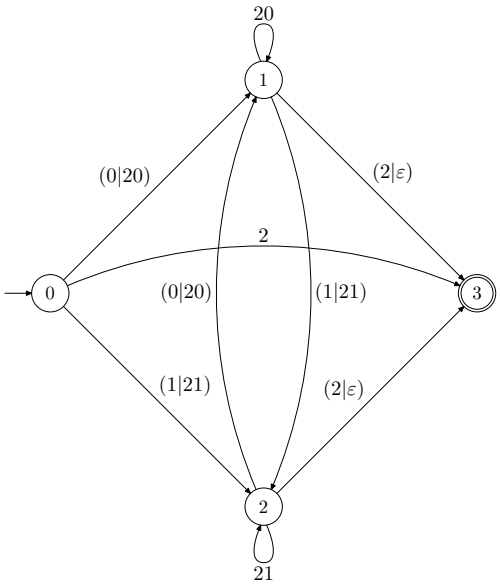
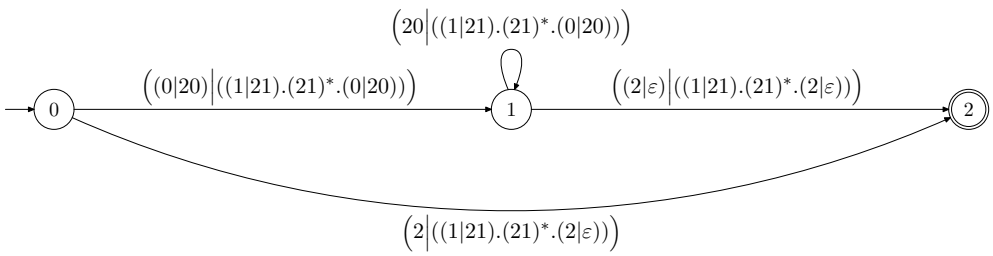


Figure 2.13 – Suppression de l'état 2



$$(\{1,01\}.\{01\}^*.\{0\} \cup \{0,01\}.\{01\}^* \cup \{0\} \cup \{\varepsilon\})$$

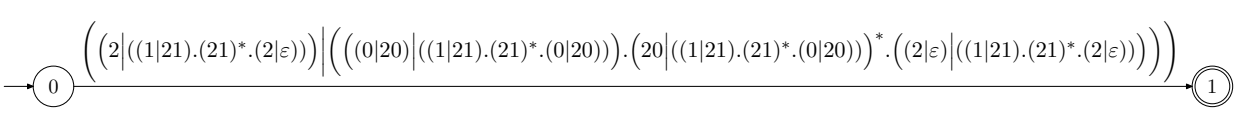
Et on continue...

Il suffit pour conclure de

1. reporter la valeur de L_3 dans L_2 ;
2. reporter les valeurs de L_2 et L_3 obtenues dans L_1 ;
3. reporter les valeurs de L_1, L_2 et L_3 obtenues dans L_0.

Bon courage !

Figure 2.14 – Suppression de l'état 1 et résultat final



2.3.3.3 Algorithme de Mac Naughton – Yamada

1. on numérote les états de 1 à n ;
2. si $(q, q') \in [\![1, n]\!]^2$, on note pour $k \in [\![1, n]\!]$

$$L^k_{q,q'} = \{\text{chemins de } q \text{ à } q' \text{ passant par les états dans } [\![1, k]\!]\}$$

on a alors, si on note I l'ensemble des états initiaux et F l'ensemble des états d'*acceptation*, le langage reconnu cherché vaut :

$$L = \bigcup_{(i,j)\in I\times F} L^n_{i,j}$$

et, on a clairement pour $(q, q') \in [\![1, n]\!]^2$

$$L^0_{q,q'} = \begin{cases} \left\{ a \in A,\ q \xrightarrow{\ a\ } q' \right\} & \text{si } q \neq q' \\ \left\{ a \in A,\ q \xrightarrow{\ a\ } q' \right\}^* & \text{si } q = q' \end{cases}$$

et on utilise la relation de récurrence immédiate, pour $k \in [\![0, n-1]\!]$ et $(q, q') \in [\![1, n]\!]^2$:

$$L^{k+1}_{q,q'} = L^k_{q,q'} \cup \left(L^k_{q,k+1} . \left(L^k_{k+1,k+1}\right)^* . L^k_{k+1,q'} \right)$$

Cela exprime :

« Pour aller de l'état q à l'état q', en n'utilisant que les $k+1$ premiers états, on peut, soit aller de q à q' en n'utilisant que les k premiers états, soit aller de l'état q à l'état $k+1$, en n'utilisant que les k premiers états, boucler sur l'état $k+1$ en n'utilisant que les k premiers états, puis aller de l'état $k+1$ à l'état q' en n'utilisant que les k premiers états. »

Exemple 2.9 – Algorithme de Mac Naughton – Yamada

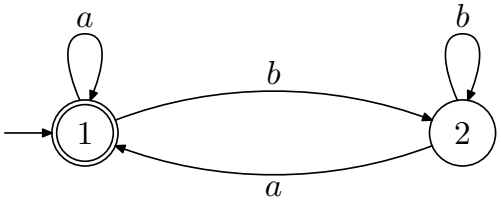
Reprenons le cas traité dans l'exemple 2.7, page 40.
On obtient :

1. $L^0_{1,2} = L^0_{3,2} = L^0_{4,2} = \{0\}$, $L^0_{1,3} = L^0_{2,3} = L^0_{4,3} = \{1\}$, $L^0_{1,4} = L^0_{2,4} = L^0_{3,4} = \{2\}$, $L^0_{1,1} = L^0_{2,2} = L^0_{3,3} = L^0_{4,4} = \{\varepsilon\}$ et les autres sont $\emptyset$.
2. puis, on applique l'algorithme...
3. celui qui nous intéresse est

$$\bigcup_{j=2}^{4} L_{1,j}$$

Exercice(s) 2.6

2.6.1 On considère l'automate (très simple) suivant :



Utiliser les trois algorithmes du cours pour trouver une expression rationnelle décrivant le langage reconnu par cet automate. *Qu'en concluez-vous ?*

2.6.2 Trouver une expression rationnelle pour le langage L en procédant de la manière suivante :

(a) écrire un AFD

(b) en déduire l'expression rationnelle

où L est l'ensemble des mots construits sur l'alphabet binaire tels que, transcrits en nombre entier, ils sont divisibles par 3. Par exemple, le mot [a]

$$\overline{110011}_{(2)} = 51 \text{ est divisible par } 3$$

a. On fera bien attention à l'ordre de lecture : le mot $\overline{1100}_{(2)}$ peut être lu sous les formes

$$[1,1,0,0] \text{ ou } [0,0,1,1]$$

Chapitre 7

Corrections des exercices

7.1 Exemples d'erreurs

Terminal 7.1 – Correction pour le code 1.1, page 21

```
root# gcc -Wall exo1.c
exo1.c: In function ‘main’:
exo1.c:4:2: warning: implicit declaration of function ‘printf’ [-Wimplicit-function-declaration]
    4 |  printf("i=%d",i);
      |  ^~~~~~
exo1.c:4:2: warning: incompatible implicit declaration of built-in function ‘printf’
exo1.c:1:1: note: include ‘<stdio.h>’ or provide a declaration of ‘printf’
  +++ |+#include <stdio.h>
    1 | int main()
```

Remarque 7.1

C'est donc une erreur sémantique : la fonction `printf` n'est pas définie.

L'existence d'un objet est vérifiée au moment de l'analyse sémantique.

Terminal 7.2 – Correction pour le code 1.2, page 21

```
root# gcc -Wall exo2.c
exo2.c: In function ‘main’:
exo2.c:3:11: error: stray ‘\303’ in program
    3 |  int ChillÃ¨s=1;
      |           ^
exo2.c:3:12: error: stray ‘\250’ in program
    3 |  int ChillÃ¨s=1;
      |            ^
exo2.c:3:13: error: expected ‘=’, ‘,’, ‘;’, ‘asm’ or ‘__attribute__’ before ‘s’
    3 |  int Chillès=1;
      |              ^
exo2.c:3:13: error: ‘s’ undeclared (first use in this function)
exo2.c:3:13: note: each undeclared identifier is reported only once for each function it appears
↪  in
exo2.c:4:16: error: stray ‘\303’ in program
    4 |  return(2*ChillÃ¨s);
      |                ^
exo2.c:4:17: error: stray ‘\250’ in program
    4 |  return(2*ChillÃ¨s);
```

```
      |                 ^
exo2.c:4:11: error: ‘Chill’ undeclared (first use in this function)
    4 |  return(2*Chillès);
      |           ^~~~~
exo2.c:4:16: error: expected ‘)’ before ‘s’
    4 |  return(2*ChillÃ¨s);
      |        ~       ^ ~
      |                )
```

Remarque 7.2

C’est clairement une erreur lexicale : le caractère `’è’` n’est pas autorisé dans les identificateurs de `C`.

Terminal 7.3 – Correction pour le code 1.3, page 22

```
root# gcc -Wall exo3.c
exo3.c: In function ‘main’:
exo3.c:6:3: warning: ‘j’ is used uninitialized in this function [-Wuninitialize ]
    6 |  k=i+j;
      |  ~~~~~
```

Remarque 7.3

C’est encore une erreur sémantique, l’écriture est correcte, mais utiliser j dans une somme n’a pas de sens !

Terminal 7.4 – Correction pour le code 1.4, page 22

```
root# gcc -Wall exo4.c
exo4.c: In function ‘main’:
exo4.c:5:12: error: expected ‘;’ before ‘else’
    5 |   return(0)
      |            ^
      |            ;
    6 |  else
      |  ~~~~
```

Remarque 7.4

Erreur syntaxique ! Il manque un `;`.

Terminal 7.5 – Correction pour le code 1.5, page 22

```
root# gcc -Wall exo5.c
```

Remarque importante 7.5

Il n’y pas d’erreur à la compilation ! Pourtant, ce code est sans intérêt, car la condition `i=1` (ou `i=2`) est *toujours vraie* ! Donc, la conditionnelle ne sert à rien !

 C’est une faute très courante *qui n’est pas visible* à la compilation. Bien sûr, il faudrait écrire la condition sous la forme `i==1`, ou `i!=1`.

7.2 Expressions régulières

Exercice 1 : Voir la session 7.6, de la présente page.

(a) L'expression régulière `^[A-Z]` filtre les majuscules des lignes qui commencent par une majuscule.

(b) L'expression régulière `␣$` filtre les fins de ligne des lignes qui se terminent par une espace.

(c) L'expression régulière `^$` filtre les lignes vides.

(d) L'expression régulière `([^o]+o){2,}` filtre les lignes qui contiennent au moins deux o séparés par autre chose et ne commençant pas la ligne (on utilisera aussi le texte 7.1, de la présente page). Le filtrage s'arrête au dernier o trouvé.

(e) L'expression régulière `[-+]?[0-9]*\.?[0-9]*` semble vouloir filtrer les nombres entiers et flottants (trouvé sur Internet), mais cela ne fonctionne pas !

Exercice 2 : Voir la session 7.7, page 59.

(a) Mots constitués de lettres, ne contenant qu'une majuscule : `\b[a-z]*[A-Z][a-z]*\b`

(b) Les lignes qui ne sont pas vides et qui ne se terminent pas par un caractère blanc : `^.*[^␣]$`

(c) Les lignes vides et celles qui ne se terminent pas par un blanc : `(^$|^.*[^␣]$)`

(d) Les dates :

```
([1-9]|(1|2)[0-9]|3(0|1))␣(janvier|février|mars|avril|mai|juin|juillet|août|septembre|octobre|novembre|décembre)␣20[0-9]{2}
```

(e) Les adresses mél (nous nous limiterons aux cas simples) : `[a-z]+(\.[a-z]+)?@[a-z]+(\.[a-z]+)+`

(f) Les adresses Internet (nous nous limiterons aussi à des cas simples) : `https?://[a-z]+(\.[a-z]+){2,}`

(g) Les nombres de type `float` en tenant compte des différents cas décrits en cours (testé sur le fichier 7.2, page suivante) `[+-]?([1-9][0-9]*\.[0-9]*|0\.[0-9]*|\.[0-9]+)`

Texte 7.1 – Où sont les *o* ?

```
1  oh ! un zoo, oh ! un igloo !
```

Terminal 7.6 – Essai d'expressions régulières diverses

```
root# grep -Eno "^[A-Z]" TexteALire.txt
1:P
3:B
6:T
8:N
9:C
11:N
14:B

root# grep -En "^[A-Z]" TexteALire.txt
1:Paris, le 23 septembre 2021
3:Bonjour Alain,
6:Tu pourras me contacter à l'adresse alain.chilles@sjtu.edu.cn
8:Ne te trompe pas comme la dernière fois !
9:Ce n'est pas alain.chilles@sjtu ni alain.chilles@sjtu.cn..
11:Nous travaillerons à la conversion des codes wxMaxima en Python (Sympy). Pour 90%
14:Bonne journée :-) $$^

root# grep -Eno " $" TexteALire.txt
3:
6:
7:

root# grep -En " $" TexteALire.txt
3:Bonjour Alain,
6:Tu pourras me contacter à l'adresse alain.chilles@sjtu.edu.cn
```

```
7:

root# grep -En "^$" TexteALire.txt
2:
5:
10:
13:

root# grep -Eno "([^o]+o){2,}" TexteALire.txt
3:Bonjo
4:tu trouveras l'information que tu so
6:Tu pourras me co
8:Ne te trompe pas comme la dernière fo
11:Nous travaillerons à la conversion des codes wxMaxima en Python (Sympy). Po
14:Bonne jo

root# grep -Eno "([^o]+o){2,}" Pb0.txt
1:, oh ! un iglo

root# grep -Eno "[-+]?[0-9]*\.?[0-9]*" TexteALire.txt
1:23
1:2021
4:.
4:.
4:.
4:.
6:.
6:.
6:.
9:.
9:.
9:.
9:.
9:.
11:.
11:90
12:.
12:.
12:.
14:-
```

Texte 7.2 – Fichier de nombres

```
1  1.23
2  -642.
3  0.256
4  -.23
5  0.0
6  .0
7  0.
8  .
9  156
10 -623
```

Terminal 7.7 – Expressions régulières de l'exercice 2.1, page 29

```
root# grep -Eno "\b[a-z]*[A-Z][a-z]*\b" TexteALire.txt
1:Paris
3:Bonjour
3:Alain
6:Tu
8:Ne
9:Ce
11:Nous
11:wxMaxima
11:Python
11:Sympy
11:Pour
14:Bonne

root# grep -Eno "^.*[^ ]$" TexteALire.txt
1:Paris, le 23 septembre 2021
4:tu trouveras l'information que tu souhaites sur http://www.sjtu.edu.cn.
8:Ne te trompe pas comme la dernière fois !
9:Ce n'est pas alain.chilles@sjtu ni alain.chilles@sjtu.cn..
11:Nous travaillerons à la conversion des codes wxMaxima en Python (Sympy). Pour 90%
12:d'entre eux, il n'y aura pas de problème particulier...
14:Bonne journée :-) $$^

root# grep -En "(^$|^.*[^ ]$)" TexteALire.txt
1:Paris, le 23 septembre 2021
2:
4:tu trouveras l'information que tu souhaites sur http://www.sjtu.edu.cn.
5:
8:Ne te trompe pas comme la dernière fois !
9:Ce n'est pas alain.chilles@sjtu ni alain.chilles@sjtu.cn..
10:
11:Nous travaillerons à la conversion des codes wxMaxima en Python (Sympy). Pour 90%
12:d'entre eux, il n'y aura pas de problème particulier...
13:
14:Bonne journée :-) $$^

root# grep -Eno "([1-9]|(1|2)[0-9]|3(0|1))
↪  (janvier|février|mars|avril|mai|juin|juillet|août|septembre|octobre|novembre|décembre)
↪  20[0-9]{2}" TexteALire.txt
1:23 septembre 2021

root# grep -Eno "[a-z]+(\.[a-z]+)?@[a-z]+(\.[a-z]+)+" TexteALire.txt
6:alain.chilles@sjtu.edu.cn
9:alain.chilles@sjtu.cn

root# grep -Eno "https?://[a-z]+(\.[a-z]+){2,}" TexteALire.txt
4:http://www.sjtu.edu.cn

root# grep -Eno "[+-]?([1-9][0-9]*\.[0-9]*|0\.[0-9]*|\.[0-9]+)" float.txt
1:1.23
2:-642.
3:0.256
4:-.23
5:0.0
6:.0
7:0.
```

7.3 Fichier `mygrep.c`

Ce code est décrit en *literate programming* : à partir de la description par abstractions (parties du code délimitées par \< et \>), le code final est généré automatiquement...

Code C 7.1 – Fichier `mygrep.c`

```
\<bibliotheques\>  // Chargement des bibliothèques

int main()
{
\<initialisation\>   /* Initialisation des données */
\<lectureUn\>   /* Lecture de l'expression régulière */
\<lectureDeux\>   /* Lecture de la chaîne à filtrer */
\<compilation\>   /* Compilation de l'expression régulière */
  do  /* Une boucle pour parcourir la chaîne */
  {
\<filtrage\>   /* Tentative de filtrage */
\<resultat\>   /* Lecture du résultat */
\<impression\>   /* Impression du résultat */
\<mise_a_jour\>   /* Mise à jour des variables */
  }
  while (res == 0);  /* Et on continue tant qu'il y a quelque chose à lire ! */
\<fin\>   /* Un peu de nettoyage... */
};
```

Code C 7.2 – Les bibliothèques (\<bibliotheques\>)

```
#include <stdio.h> // Pour avoir printf et scanf
#include <regex.h> // Pour faire des expressions régulières !
```

Code C 7.3 – Initialisation des données (\<initialisation\>)

```
    // expr est l'expression rationnelle
    // texte est le texte à lire
    // p est un pointeur sur le début à lire
    // myregexp est la forme compilée de l'expression régulière
    // pmatch contient les résultats du filtrage
    // res contient les résultats des filtrages successifs
    // debut désigne le début du motif filtré
    // fin désigne l'adresse juste après le motif
    // offset gère les décalages
    char expr[100], texte[100];
    char* p = texte;
    regex_t myregexp;
    regmatch_t pmatch[1];
    int res = 0, debut, fin, offset = 0;
```

Code C 7.4 – Lecture sécurisée de l'expression régulière (\<lectureUn\>)

```
22    // Noter la commande sécurisée du scanf
23    printf("Donner l'expression régulière : ");
24    scanf("%99[^\n]", expr);
25  \<verification\>
```

Code C 7.5 – Lecture sécurisée de la chaîne à filtrer (\<lecture2\>)

```
31    // On recommence avec le texte
32    printf("Donner le texte à reconnaître : ");
33    scanf("%99[^\n]", texte);
34  \<verification\>
```

Code C 7.6 – Lecture sécurisée (\<verification\>)

```
24    // Il faut savoir comment on s'est arrêté de lire
25    if (getchar() != '\n')
26    {
27      printf("Erreur,  ligne trop longue !");
28      return(1);
29    };
```

Code C 7.7 – Compilation de l'expression régulière (\<compilation\>)

```
40    if (regcomp(&myregexp, expr, REG_EXTENDED))
41    {
42      printf("Mauvaise expression rationnelle : %s", expr);
43      regfree(&myregexp);
44      return(2);
45    };
```

Code C 7.8 – Utilisation de l'expression régulière (\<filtrage\>)

```
48        // On filtre à partir de p
49        res = regexec(&myregexp, p, 1, pmatch, 0);
50        if (res) break; // regexec s'est planté ou n'a rien filtré
```

Code C 7.9 – Résultat du filtrage (\<resultat\>)

```
51        // On récupère les informations
52        debut = pmatch[0].rm_so;
53        fin = pmatch[0].rm_eo;
```

Code C 7.10 – Impression du motif filtré (\<impression\>)

```
54        // Noter le %.* qui permet d'afficher une sous-chaîne
55        printf("%.*s (%d, %d)\n", fin-debut, p+debut, debut+offset, fin+offset-1);
```

Code C 7.11 – Mise à jour des variables (\<mise_a_jour\>)

```
56          p += fin;
57          offset += fin;
```

Code C 7.12 – Et c'est la fin... (\<fin\>)

```
60      regfree(&myregexp);
61      return(0);
```

Finalement, le code re-construit est obtenu !

Code C 7.13 – Code final de `mygrep.c`

```
1   #include <stdio.h> // Pour avoir printf et scanf
2   #include <regex.h> // Pour faire des expressions régulières !
3    // Chargement des bibliothèques
4
5   int main()
6   {
7     // expr est l'expression rationnelle
8     // texte est le texte à lire
9     // p est un pointeur sur le début à lire
10    // myregexp est la forme compilée de l'expression régulière
11    // pmatch contient les résultats du filtrage
12    // res contient les résultats des filtrages successifs
13    // debut désigne le début du motif filtré
14    // fin désigne l'adresse juste après le motif
15    // offset gère les décalages
16    char expr[100], texte[100];
17    char* p = texte;
18    regex_t myregexp;
19    regmatch_t pmatch[1];
20    int res = 0, debut, fin, offset = 0;
21    /* Initialisation des données */
22    // Noter la commande sécurisée du scanf
23    printf("Donner l'expression régulière : ");
24    scanf("%99[^\n]", expr);
25    // Il faut savoir comment on s'est arrêté de lire
26    if (getchar() != '\n')
27    {
28      printf("Erreur,  ligne trop longue !");
29      return(1);
30    };  /* Lecture de l'expression régulière */
31    // On recommence avec le texte
32    printf("Donner le texte à reconnaître : ");
33    scanf("%99[^\n]", texte);
34    // Il faut savoir comment on s'est arrêté de lire
35    if (getchar() != '\n')
36    {
37      printf("Erreur,  ligne trop longue !");
38      return(1);
```

```
39    };  /* Lecture de la chaîne à filtrer */
40    if (regcomp(&myregexp, expr, REG_EXTENDED))
41    {
42      printf("Mauvaise expression rationnelle : %s", expr);
43      regfree(&myregexp);
44      return(2);
45    };  /* Compilation de l'expression régulière */
46    do  /* Une boucle pour parcourir la chaîne */
47    {
48        // On filtre à partir de p
49        res = regexec(&myregexp, p, 1, pmatch, 0);
50        if (res) break; // regexec s'est planté ou n'a rien filtré  /* Tentative de
          ↪  filtrage */
51        // On récupère les informations
52        debut = pmatch[0].rm_so;
53        fin = pmatch[0].rm_eo;  /* Lecture du résultat */
54        // Noter le %.* qui permet d'afficher une sous-chaîne
55        printf("%.*s (%d, %d)\n", fin-debut, p+debut, debut+offset, fin+offset-1);  /*
          ↪  Impression du résultat */
56        p += fin;
57        offset += fin;  /* Mise à jour des variables */
58    }
59    while (res == 0);  /* Et on continue tant qu'il y a quelque chose à lire ! */
60    regfree(&myregexp);
61    return(0);  /* Un peu de nettoyage... */
62  };
```

Il peut y avoir des problèmes avec les caractères accentués ! Avec `grep` *et* avec `regex.h`. Voir par exemple la session 7.8, de la présente page. `grep` reconnaît des caractères accentués alors qu'on ne le lui demande pas, et `regex` se trompe dans les comptes si on met un caractère accentué dans l'expression régulière (et imprime mal), car il lit le premier *byte* !

Terminal 7.8 – Problèmes avec les caractères accentués

```
root# cat toto.txt
  Ce cours est donné par Alain Chillès (祁冲 en chinois)

root# grep -Eno "[a-z]+" toto.txt
1:e
1:cours
1:est
1:donné    <--
1:par
1:lain
1:hillès  <--
1:en
1:chinois

root# ./mygrep
Donner l'expression régulière : [a-zè]+
Donner le texte à reconnaître :   Ce cours est donné par Alain Chillès (祁冲 en chinois)
e (3, 3)
cours (5, 9)
est (11, 13)
```

```
donnÃ (15, 19)  <--
par (22, 24)
lain (27, 30)
hillès (33, 39)
en (49, 50)
chinois (52, 58)

root# ./mygrep
Donner l'expression régulière : [a-zé]+
Donner le texte à reconnaître :   Ce cours est donné par Alain Chillès (祁冲 en chinois)
e (3, 3)
cours (5, 9)
est (11, 13)
donné (15, 20)
par (22, 24)
lain (27, 30)
hillÃ (33, 37)  <--
s (39, 39)
en (49, 50)
chinois (52, 58)

root# ./mygrep
Donner l'expression régulière : [a-zéè]+
Donner le texte à reconnaître :   Ce cours est donné par Alain Chillès (祁冲 en chinois)
e (3, 3)
cours (5, 9)
est (11, 13)
donné (15, 20)
par (22, 24)
lain (27, 30)
hillès (33, 39)
en (49, 50)
chinois (52, 58)

root# ./mygrep
Donner l'expression régulière : [a-zéè]+
Donner le texte à reconnaître :   Ce cours est donné par Alain Chillès (祁冲 en chinois). Ça va ?
e (3, 3)
cours (5, 9)
est (11, 13)
donné (15, 20)
par (22, 24)
lain (27, 30)
hillès (33, 39)
en (49, 50)
chinois (52, 58)
Ã (62, 62)  <--
a (64, 64)
va (66, 67)
```

7.4 Langages rationnels

7.4.1 Comprendre des expressions régulières

Nous utiliserons la notation de `Python` *pour désigner les éléments d'une chaîne de caractères.*

Correction de l'exercice 2.3.1, page 34

Expression régulière	Traduction ensembliste	Interprétation
`a(a|b)*a`	$\Sigma = \{a, b\},\ L = \{a\}\Sigma^*\{a\}$	$\{s \in \Sigma^*,\ \|s\| \geq 2,\ s[0] = a,\ s[-1] = a\}$
`(a?b*)*`	$\Sigma = \{a, b\},$ $L = (\{\varepsilon, a\}\{b\}^*)^*$	Σ^*
`(a|b)*a(a|b)(a|b)`	$\Sigma = \{a, b\},\ L = \Sigma^* a \Sigma^2$	$\{s \in \Sigma^*,\ \|s\| \geq 3,\ s[-3] = a\}$

7.5 Transformer un AFN en AFD

1. On commence par lister les ensembles d'états qui peuvent être actifs en même temps :

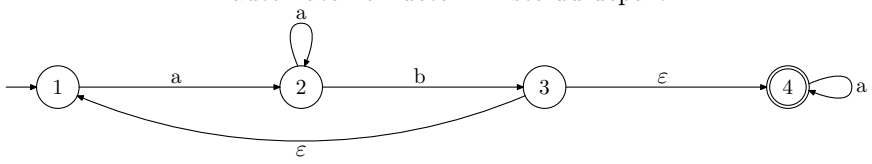
   $$\{1\},\ \{2\},\ \{1, 3, 4\} \text{ et } \{2, 4\}$$

2. On trace ensuite l'AFD en regardant ce qui se passe à partir de tous les états constituants le nouvel état. Cela donne le dessin 7.1, de la présente page.
3. Le langage reconnu peut être décrit par l'expression régulière suivante :

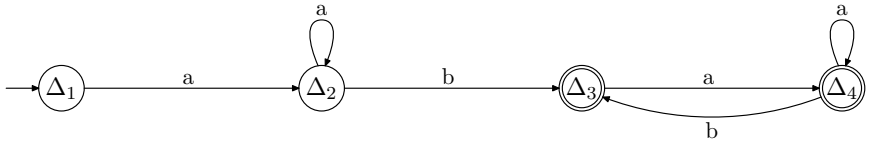
   `a+b(a+b?)*`

Figure 7.1 – Transformer un AFN en AFD

L'automate non déterministe du départ



devient l'automate déterministe équivalent (qui reconnaît le même langage)



$\Delta_1 = \{1\},\ \Delta_2 = \{2\},\ \Delta_3 = \{1, 3, 4\}$ et $\Delta_4 = \{2, 4\}$

Bibliographie

[1] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, *Compilers : Principles, Techniques, and Tools (2nd Edition)*, Addison Wesley, 2006.

Figure 7.2 – $\{a^n b^n c^n,\ n \in \mathbb{N}^*\}$