République **A**lgérienne **D**émocratique et **P**opulaire
Ministère de l'**E**nseignement **S**upérieur et de la **R**echerche **S**cientifique

UNIVERSITE **M**OULOUD **M**AMMERI DE **T**IZI-**O**UZOU

FACULTE DU **G**ENIE **E**LECTRIQUE ET D'INFORMATIQUE
DEPARTEMENT D'INFORMATIQUE

Mémoire de Fin d'Etudes de MASTER ACADEMIQUE

Domaine : **Mathématiques et Informatique**
Filière : **Informatique**
Spécialité : **Réseaux mobilité et systèmes embarqués**

Présenté par

Lilia Djermane

Thème

Conception et implémentation d'un générateur automatique d'automates d'états finis déterministes et leurs représentations graphiques, à partir d'expressions régulières

Mémoire soutenu publiquement le 11/07/2016. devant le jury composé de :

Président : M. Mohammed-Said HABET
Encadreur : M. Chabane HEMDANI
Examinateur : M. Samir SADOU
Examinateur : Mme Rebiha HADAOUI

Remerciement

Je tiens à adresser mes remerciements en premier à Mr C.Hemdani, enseignant à l'université de l'UMMTO. En tant que promoteur, il m'a guidé dans le travail, m'a aidé à trouver des solutions pour avancer et a été attentif et disponible.

Je remercie aussi mes amis et camarades qui m'ont été d'une grande aide et d'un grand soutien moral.

Et je remercie surtout ma famille pour leur patience, leur soutien inconditionnel et leur compréhension.

Je tiens aussi à présenter mes remerciements aux membres du jury, qui ont bien voulu accepter d'évaluer et de juger le travaille.

Table des matières

Table des figures

Liste des tableaux

Liste des algorithmes

Remerciements

Je tiens à adresser mes remerciements en premier à M. C. Hemdani, enseignant à l'université de l'UMMTO. En tant que promoteur, il m'a guidé dans le travail, m'a aidé à trouver des solutions pour avancer et a été attentif et disponible.

Je remercie aussi mes amis et camarades qui m'avoir apporter leur aide et leur soutien moral.

Je n'oublierai pas de remercier particulièrement ma famille pour sa patience, son soutien inconditionnel et sa compréhension.

Je tiens aussi à présenter mes remerciements aux membres du jury, qui ont bien voulu accepter d'évaluer et de juger mon travail.

Introduction

À l'origine, le langage est l'outil fondamental de l'expression orale, de la communication entre les hommes. Nombreux et d'une grande diversité, ces langages originels utilisés pour se comprendre sont appelés "naturels" car ils sont directs, sans intermédiaire. Ils se distinguent de ceux que l'homme a développés pour communiquer avec les machines. Mais quelque soit le domaine considéré, un langage est utilisé de deux façons différentes : Emission des mots, du point de vue du locuteur : cet aspect lié à la génération des mots du langage est décrit suivant la grammaire qui permet d'établir des règles précises à adopter, et la réception des mots : du point de vue de l'auditeur, la réception des mots d'un langage est liée à la reconnaissance des mots de ce langage.

Au XXe siècle, trois disciplines se sont particulièrement interrogées sur la notion de langage : la linguistique, les mathématiques et l'informatique. Elles ont établi une théorie des langages.

En informatique la théorie de langage s'intéresse à la reconnaissance d'ensemble de mots sur un vocabulaire donné, autrement dit elle décrit les langages formels qui sont des langages construits à partir d'un ensemble de symboles, lettres ou lexèmes et régis par des règles.

Les langages formels sont utilisés dans divers applications entre autres, la spécification de langages de programmation, la compilation, la recherche de motifs dans les éditeurs de texte dans les programmes dans une base de données et sur le web, compression de textes, codage et décodage, cryptographie, décodage du génome, calculabilité et aussi en linguistique, les sciences cognitives, etc.[1] et peuvent être analysés par des machines ayant des ressources de calcul limitées.

Pour participer à l'effort de développement d'outils logiciels facilitant la description et la reconnaissance de langages formels, on se propose dans ce projet de concevoir et d'implémenter un outil permettant de faciliter la spécification des langages réguliers et d'automatiser le processus de construction d'automates ainsi que leurs représentation graphiques sous formes d'images. En d'autres termes, il s'agit de concevoir et d'implémenter un générateur automatique d'automates d'états finis simples et déterministes à partir d'une expression régulière.

Le rapport sera scindé en trois chapitres.

Après une introduction générale, nous introduirons dans le premier chapitre les définitions des notions liées à ces langages ainsi que les outils permettant de les décrire.

Le second chapitre sera consacré à la présentation des différentes étapes liées à la conception de notre application.

Dans le troisième et dernier chapitre, nous traiterons de la réalisation de notre projet. Nous décrirons l'environnement de travail ainsi que les différents codes sources implémentés.

Enfin, nous terminerons par une conclusion générales et une perspective.

Chapitre 1

Langages formels et outils de manipulation

1.1 Introduction

Comme notre projet consiste à définir un langage de spécification de langages réguliers et de construire un générateur automatique d'automates d'états finis à partir d'expressions régulières, il est nécessaire avant d'entamer notre travail, de rappeler certaines notions relatives aux langages formels et aux outils de manipulation de ces derniers.

1.2 Langages formels

Avant de donner une définition d'un langage formel, rappelons d'abord quelques définitions d'objets qui le composent.

1.2.1 Notions de base

1.2.1.1 L'alphabet d'un langage [1]

L'alphabet d'un langage est l'ensemble non vide, dénombrable de symboles, lettres ou lexèmes, qui servent à construire les mots du langage ; on suppose souvent que cet alphabet est fini.

Exemple

Des exemples d'alphabets sont :
A_1= { • , ? }
A_2= { a, b, c, ..., z } ,
A_3= { if, $then$, $else$, id, nb, =, + }

1.2.1.2 Le mot

Un mot est une suite d'éléments d'un alphabet.

Exemple

— ••? est un mot sur l'alphabet A_1,
— if est un mot sur l'alphabet A_2,
— $if\ id = nb$ est un mot sur l'alphabet A_3.

1.2.1.3 Le mot vide ε

On appelle *mot vide*, l'unique mot de longueur 0 (à distinguer du caractère blanc). Il a surtout une utilité théorique et il est habituellement noté ε.

1.2.2 Définition d'un langage formel

Un langage peut être défini simplement comme un ensemble de mots. Les langages étant des ensembles, toutes les opérations ensemblistes « classiques » leur sont donc applicables. Elles sont utilisées pour fabriquer de nouveaux langages à partir de langages donnés.

Les opérations les plus importantes sur les langages formels sont l'*union*, l'*intersection*, la *concaténation* et la *fermeture de Kleene*.

Pour définir ces opérations, considérons trois langages arbitraires L, L_1 et L_2.

- L'*union* des deux langages L_1 et L_2, notée $L_1 \cup L_2$, est définie par : $L_1 \cup L_2 = \{u \in A^* \mid u \in L_1 \text{ ou } u \in L_2\}$. Cette opération est commutative et associative .
- L'*intersection* des deux langages L_1 et L_2, notée $L_1 \cap L_2$, est définie par : $L_1 \cap L_2 = \{u \in A^* \mid u \in L_1 \text{ et } u \in L_2\}$. Cette opération est commutative et associative
- La *concaténation* des deux langages L_1 et L_2 , notée L_1L_2, est définie par : $L_1L_2 = \{uv \in A^* \mid u \in L_1 \text{ et } v \in L_2\}$. Cette opération est associative mais pas commutative.
  Notons qu'on abrège habituellement le langage $LL.....L$ par L^n où n désigne le nombre d'occurrences de L dans la première expression.
- la *fermeture de Kleene* (l'étoile) du langage L, notée L^*, est définie par : $L^* = U_{i \geq 0} L^i$. L^* contient tous les mots qu'il est possible de construire en concaténant un nombre fini (éventuellement réduit à zéro) d'éléments du langage L.
- la *fermeture positive* de Kleene d'un langage L, notée L^+, est définie par : $L^+ = U_{i \geq 1} L^i$. L^+ contient tous les mots qu'il est possible de construire en concaténant un nombre fini (supérieur à zéro) d'éléments du langage L.

Un des intérêts de ces notations est qu'elles permettent d'exprimer de manière formelle (et compacte) des langages complexes, éventuellement infinis, à partir de langages plus simples.

Exemple

L'ensemble des suites de 0 et de 1 contenant la séquence 111 s'écrira $\{0, 1\}^*111\{0, 1\}^*$; la notation $\{0, 1\}^*$ permettant un nombre arbitraire de 0 et de 1 avant la séquence 111.

1.3 Les grammaires formelles

Une grammaire est un formalisme permettant de définir une syntaxe et donc un langage formel, c'est-à-dire un ensemble de mots admissibles sur un alphabet donné.

Autrement dit c'est un mécanisme qui permet d'engendrer un ensemble de phrases à partir d'un lexique donné. De façon formelle une grammaire est un quadruplet $G =< V_N, V_T, S, P >$

avec

- V_N, Un ensemble fini de symboles, appelés *symboles non-terminaux*, notés conventionnellement par des majuscules,

— V_T, ensemble fini de symboles, appelés *symboles terminaux* notés conventionnellement par des minuscules,
— un élément de V_N, appelé axiome, noté conventionnellement S,
— un ensemble P de règles de production de la forme $\alpha \longrightarrow \beta$ où α et β sont des séquences de symboles terminaux et non terminaux : $\alpha, \beta \in (V_N \cup V_T)^*$.

Appliquer une règle de production consiste à remplacer une occurrence du membre de gauche de cette règle par son membre de droite ; l'application successive de règles de productions s'appelle une *dérivation.*

Un langage $L(G)$ engendré par une grammaire G est l'ensemble de tout les mots dérivables par la grammaire à partir de S au moyen des règles de P.

Exemple

Soit la grammaire $G =< V_N, V_T, S, P >$ telle que :
— $V_N = \{S\}$,
— $V_T = \{a, b\}$,
— $P = \{S \longrightarrow aSb \,|\, \varepsilon\}$.

Un exemple de dérivation de mot est :
$S \Longrightarrow aSb \Longrightarrow aaSbb \Longrightarrow aaaSbbb \Longrightarrow aaabbb$.
Le langage engendré par cette grammaire est $L = \{\varepsilon, ab, aabb, aaabbb, \ldots\}$. C'est le langage des mots de la forme $a^n b^n$ (un certain nombre de a éventuellement 0, en vertu de la deuxième règle, suivis du même nombre de b).

1.3.1 Types de grammaires formelles

Les langages ont été classés en fonction du type de grammaire qui permet de les générer. Chomsky a défini quatre classes de langage, nommées de type 0 à type 3, et donc aussi quatre types de grammaire, qui engendrent ces langages :

Grammaires de type 0

Elles sont dites grammaires *générales* car sans aucune restriction. Les règles de production sont de la forme :

$$\alpha \longrightarrow \beta$$

avec α, $\beta \in (V_N \cup V_T)*$.

Grammaires de type 1

Ces grammaires sont dites grammaires *contextuelles* et les règles de production sont de la forme :

$$\alpha A \beta \longrightarrow \alpha \omega \beta$$

avec α, β, $\omega \in (V_N \cup V_T)^*$ et $w \neq \varepsilon$.

Toute règle comprend un non-terminal entouré de deux mots qui décrivent le contexte dans lequel la variable peut être remplacée. Elle est contextuelle, car le remplacement d'un élément non-terminal peut dépendre des éléments autour de lui : son contexte.

Les langages produits sont appelés *langages contextuels* ou *langages à contexte lié*.

Grammaires de type 2

Ce sont les grammaire dites *algébriques* ou *hors contexte*. Les règles de production sont de la forme :

$$A \longrightarrow \alpha$$

avec $\alpha \in (V_N \cup V_T)^*$

Ce sont des grammaires contextuelles où le contexte est vide, ce qui signifie que les symboles non terminaux sont traités indépendamment de la place où ils apparaissent.

Les langages engendrés par de tels grammaires sont appelés *langages hors contexte*.

Grammaires de type 3

Les grammaires de ce type sont dites régulières. Les grammaires régulières sont soit les grammaires *linéaires à gauche* soit les grammaires *linéaires à droite* :

— Dans les grammaires linéaires à gauche, les règles sont de la forme :

$$A \longrightarrow B\alpha \quad \text{ou} \quad A \longrightarrow \alpha$$

avec A, $B \in V_N$ et $\alpha \in V_T^*$.

— Dans les grammaires linéaires à gauche, les règles sont de la forme :

$$A \longrightarrow \alpha B \quad \text{ou} \quad A \longrightarrow \alpha$$

avec A, $B \in V_N$ et $\alpha \in V_T^*$.

Les grammaires régulières engendrent les *langages réguliers* ou *rationnels*.

On ne peut pas autoriser les deux types de règles simultanément dans une grammaire sans sortir de la classe des langages rationnels : on obtiendrait les grammaires linéaires qui constituent une classe intermédiaire entre le type 2 et le type 3.

Les règles d'une grammaire linéaire sont de la forme :

$$A \longrightarrow \alpha B \beta \quad \text{ou} \quad A \longrightarrow \alpha$$

avec A, $B \in V_N$ et $\alpha \in V_T^*$.

1.3.2 Langage engendré et dérivation

Le langage engendré par une grammaire G est l'ensemble de toutes les chaînes terminales (autrement dit, dans V_T^*) qui dérivent de l'axiome S. Il est noté $L(G)$. Pour montrer qu'un mot w est dans le langage L il faut donc exhiber une suite finie d'applications de règles de production partant de S et aboutissant à w, appelée une dérivation. On peut représenter une dérivation par :

une dérivation à gauche : Une dérivation à gauche est une dérivation qui s'applique à la première variable rencontrée dans la chaîne (c'est la variable qui est le plus "à gauche" possible).

une dérivation à droite : une dérivation à droite est la liste des règles employées lorsque l'on remplace systématiquement le non-terminal le plus à droite en premier.

On peut tracer un arbre de dérivation associé à toute suite de dérivations. Chaque nœud représente un exemplaire d'une variable réécrite, avec le numéro de la règle de grammaire appliquée.

Exemple

Soit la grammaire $G =< V_N, V_T, P, S >$ telle que $V_T = \{a, b\}$, $VN = \{A\}$, $P = \{S \longrightarrow abSA|\varepsilon\,,\ A \longrightarrow Aa|\varepsilon\}$. La dérivation du mot `ababa` s'effectue comme suit : S $\Longrightarrow$ abSA $\Longrightarrow$ abS $\Longrightarrow$ ababSA $\Longrightarrow$ ababSAa $\Longrightarrow$ ababAa $\Longrightarrow$ ababa

La figure 1.1 montre l'arbre de dérivation qui lui est associé.

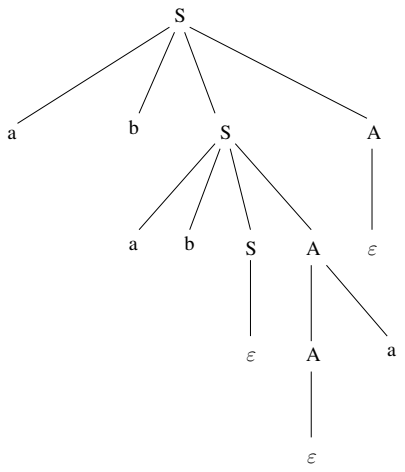


FIGURE 1.1 – Exemple d'arbre de dérivation.

En lisant les symboles qui figurent aux extrémités des branches, de gauche à droite, on obtient le mot `ababa`.

1.3.3 Classification des langages formels

Il existe une classification commune des langages formels, dite classification de *Chomsky*. Les types des différentes classes ainsi que leurs descriptions sont résumés dans le tableau 1.1.

Classe de langage	Description
0	Langages sans restriction.
1	Langages à contexte lié.
2	Langages à contexte libre.
3	Langages réguliers.

TABLE 1.1 – Classification de Chomsky.

L'ordre décroissant du type de langage, reflète la complexité de la structure.
À chaque chaque classe de langage i correspond type de grammaire i, et réciproquement.

1.3.3.1 Langages sans restriction

Comme ces langages n'ont aucune restriction, on ne peut automatiser leur analyse. Il ne peut pas exister de programme qui accepte en entrée n'importe quelle grammaire G du type 0 et un mot w et produise la réponse *« oui »*» si $w \in L(G)$ et *«non»* si ce n'est pas le cas. Ceci montre qu'avec de simples règles de production, on peut engendrer des ensembles de mots, hautement complexes.

1.3.3.2 Langages à contexte lié

Pour les langages à contexte liés, correspondant aux grammaires de type 1, l'analyse est plus facile car on peut énumérer toutes les mots de longueur 1, puis 2, puis 3, etc. Donc on peut toujours écrire un programme qui détermine si un mot appartient ou non au langage. Les langages contextuels sont rarement utilisés, à la fois en théorie et en pratique.

1.3.3.3 Langages à contexte libre

En théorie des langages formels, un langage algébrique ou langage non contextuel est un langage qui est engendré par une grammaire algébrique. De manière équivalente, un langage algébrique est un langage reconnu par automate à pile.

Les langages algébriques ont des applications importantes dans la description des langages informatiques en linguistique.

1.3.3.4 Les langages réguliers

Les langages réguliers sont dit aussi rationnels. Ce sont les langages engendrés par des grammaires régulières.

Les langages rationnels ont de très nombreuses applications, à la fois théoriques et pratiques. Ils sont très utilisés en informatique (par exemple en compilation) et en linguistique (par exemple pour décrire la morphologie d'une langue). Ils interviennent dans les traitements de texte ou dans des commandes spécifiques comme *grep* du système *Unix*.

1.4 Les expressions régulières

Hormis la notion de grammaire qui constitue un moyen de description d'un langage formel en général, nous disposons dans le cas particulier des langages réguliers d'un autre formalisme très puissant permettant de les décrire et de les manipuler : les expressions régulières.

Pour illustrer leurs utilisation, considérons l'exemple des identificateurs du langage C. Ils sont formés à partir de chiffres, de lettres et du souligné.

Si $lettre\_$ représente toute les lettres de l'alphabet ou le souligné, et $chiffres$ représente n'importe quel chiffre décimal, alors le langage des identificateurs du langage C peut être décrit par l'expression suivante :

$$lettres\_(lettres\_\,|\,chiffres)^*$$

La barre verticale représente l'union, les parenthèses sont utilisées pour regrouper les sous expressions, l'étoile signifie : zéro, une ou plusieurs occurrences de l'expression entre parenthèses et la juxtaposition de *lettres_* et les reste de l'expression représente une concaténation.

1.4.1 Expression régulière et langage régulier

Toute expression régulière dénote un langage régulier, et réciproquement, pour un langage régulier, il existe une expression régulière qui le dénote.[5]

Cela signifie que si R est un langage régulier, alors il existe une expression régulière E tel que $L(E) = R$.

1.4.2 Construction des expressions régulières

Les expressions régulières sont construites récursivement à partir d'expressions régulières plus petites. Chaque expression régulière r dénote un langage $L(r)$, qui est également défini récursivement à partir des langages dénotés par les sous-expressions régulières de r.[4]

Les règles qui définissent les expressions régulières sur un alphabet donné Σ et les langages que dénotent ces expressions sont :

1. ε est une expression régulière, et $L(r) = \{\varepsilon\}$, c.-à-d. le langage dont le seul membre est la chaîne vide.
2. Si a est un symbole appartenant à Σ, alors a est une expression régulière, et $L(r) = \{a\}$, c.-à-d. le langage comportant une seul chaîne de longueur un, avec a en première et unique position.

Il existe quatre façons de construire des expressions régulières plus grandes à partir de plus petite. Soient r et s des expressions régulières dénotant respectivement les langages $L(r)$ et $L(s)$:

1. $r|s$ est une expression régulière dénotant le langage $L(r) \cup L(s)$. On l'appelle l'*alternative* ou l'*union*.
2. rs est une expression régulière dénotant le langage $L(r)L(s)$. On l'appelle la *concaténation*.
3. r^* est une expression régulière dénotant le langage $(L(r))^*$. On l'appelle l' *itération*.
4. (r) une expression régulière dénotant le langage $L(r)$, autrement dit on peut ajouter des paires de parenthèses supplémentaire autour d'une expression sans changer le langage qu'elle dénote.

Des théorèmes algébriques et des conventions sur les opérateurs permettent d'éliminer des parenthèses superflues de certaines expressions régulières et indiquent que deux expressions régulières de forme différentes sont équivalentes :

— l'opérateur unaire $*$ a la plus grande priorité ;
— la concaténation a la deuxième priorité et est associative à gauche ;
— l'alternative a la plus faible priorité et est associative à gauche. L'alternative est commutative : $r|s = s|r$.

Exemple

Selon les conventions ci-dessus, $(a)|((b) * (c))$ est équivalente à $a|b * c$.

1.4.3 Notations supplémentaires

En pratique, on rencontre diverses notations qui peuvent toutes être exprimées à l'aide des constructions de base des expressions régulières. Autrement dit, l'emploi de ces notations n'augmente pas la classe des langages réguliers, mais permet simplement d'écrire des expressions régulières plus compactes.

— Le motif optionnel $p?$ défini comme $p|\varepsilon$.
— La répétition au moins une fois p^+ définie comme pp^*.
— Le point « . » qui est l'alternative de tous les caractères de l'alphabet du langage A. On a donc $\{.\} = A$.

Notons que Les opérateurs + et ? ont la même priorité et la même associativité que $*$.

Il existe également toute une variété de notations pour des ensembles de caractères. Ces motifs sont des abréviations de l'alternative, notées entre crochets [...]. Au sein de ces crochets, on trouve une suite des classes de caractères suivantes :

— Un caractère c se représente lui-même.
— Un intervalle $c_1 - c_2$ représente les caractères dont les codes sont compris au sens large entre les codes de c_1 et c_2.
— La notation [^...] se comprend comme le complémentaire des classes de caractères données.

1.4.4 Les définitions régulières

Pour simplifier les notations, on peut donner des noms à certaines expressions régulières et utiliser ces noms dans des expressions ultérieures ; ces noms deviennent comme des symboles. Cela évite d'avoir des expressions denses et difficiles à lire. [4]

Par exemple, on peut donner l'expression régulière des nombre non signés en utilisant la définition régulière suivante :

$chiffre = [0-9]$
$chiffres = chiffre^+$
$partie\_fractionnaire = (.chiffres)?$
$exposant = ((+|-)?chiffres))?$
$nombre = chiffres\ partie\_fractionnaire\ exposant$

1.4.5 Syntaxe d'écriture des expressions régulières

L'écriture de certaines expressions régulières peut être longue fastidieuse. Les métacaractère ont été mis au point afin de palier à cette difficulté, ce sont des caractères spéciaux : c'est à dire qu'ils ont une signification particulière dans la construction des motifs de recherche qui permettent d'exprimer des chaînes de caractères de manière compacte et facile.

Le tableau 1.2 énumère des métacaractères pouvant être utilisés et ayant chacun une correspondance bien précise.

Symbole	Correspondance	Exemple
\	Caractère d'échappement	[\.] contient un "."
^	Début de ligne	^b$ contient uniquement b
.	N'importe quel caractère	^.$ contient un seul caractère
$	Fin de ligne	er$ finit par "er"
|	Alternative	^(a|A) commence par a ou A
( )	Groupement	^((a)|(er)) commence par a ou er
-	Intervalle de caractères	^[a-d] commence par a,b,c ou d
[ ]	Ensemble de caractères	[0-9] contient un chiffre
[^]	Tout sauf un ensemble de caractères	^[^a] ne commence pas par a
+	1 fois ou plus	^(a)+ commence par un ou plusieurs a
?	0 ou 1 fois	^(a)? commence ou non par un a
*	0 fois ou plus	^(a)* peut ou non commencer par a
{x}	x fois exactement	a{2} deux fois "a"
{x,}	x fois au moins	a{2,} deux fois "a" au moins
{x, y}	x fois minimum, y maximum	a{2,4} deux, trois ou quatre fois "a"

TABLE 1.2 – Métacaractères et leurs correspondances.

1.4.6 Utilisation des expressions régulières

Initialement créées pour décrire des langages formels, les expressions rationnelles sont naturellement utilisées dans l'interprétation et la description des langages rationnels ; compilateurs et interprètes sont ainsi basés dessus.

La puissance des expressions rationnelles pour la simple manipulation du texte les rend incontournables dans les éditeurs de texte, notamment ceux en mode texte ne disposant pas d'interface graphique. Pour cette même raison, un grand nombre d'utilitaires Unix savent utiliser nativement les expressions rationnelles, les plus connus desquels étant *GNU grep* ou *GNU sed* qui, à la manière des éditeurs de texte, utilisent ces expressions pour parcourir de façon automatique un document à la recherche de morceaux de texte compatibles avec le motif de recherche, et éventuellement effectuer un ajout, une substitution ou une suppression.

Les expressions rationnelles ont vu un nouveau champ d'application avec le développement d'Internet, et la diffusion de code malveillant ou de messages pourriels.

À titre d'exemple, on pourrait représenter (de manière non exhaustive) des adresses email contenues dans un document textuel par l'expression régulière

$$\hat{}[a-zA-Z-]^{+}@[a-zA-Z-]^{+}\backslash.[a-zA-Z]\{2,6\}.$$

1.5 Les automates

1.5.1 Notions générales

Les automates sont des objets mathématiques qui permettent de modéliser un grand nombre de systèmes (informatiques). L'étude des automates a commencé vers la fin des années cinquante. Elle se base sur de nombreuses techniques (topologie, théorie des graphes, logique, algèbre, etc.).

Les automates sont utilisés dans plusieurs disciplines dont voici quelques exemples classiques :
— la vérification d'un circuit électronique
— la recherche d'occurrence dans un texte (moteur de recherches sur le web, etc.)
— la vérification de protocoles de communication,
— la compression de données,
— la compilation,
— la biologie (génomique).

En théorie des langages, les automates jouent le rôle d'*accepteurs* ou de *reconnaisseurs* de langages. À chaque type de grammaire (ou de langage), donc de langage dans la hiérarchie de Chomsky, correspond un type d'automate comme système reconnaisseur du langage. [5] Le tableau 1.3 identifie l'automate reconnaisseur pour chaque type de grammaire.

Type de grammaire	Automate reconnaisseur
0	Machine de Turing
1	Automate à bornes linéaires
2	Automate à pile
3	Automate d'états fini.

TABLE 1.3 – Automates pour les différents types de grammaires.

1.5.2 Automates d'états finis

Les automates finis sont des *machines abstraites* qui savent reconnaître l'appartenance ou la non-appartenance d'un mot à un langage régulier donné. Ces machines abstraites constituent un modèle théorique de référence.

Dans la pratique, nombreuses sont les applications qui implémentent la notion d'automates finis ou ses variantes (cela va du compilateur ... à la machine à café !).

Un automate fini est constitué d'états et de transitions. Son comportement est dirigé par un mot fourni en entrée : l'automate passe d'état en état, suivant les transitions, à la lecture de chaque lettre de l'entrée.

Il est dit *fini* car il possède un nombre fini d'états.

1.5.2.1 Définition formelle

Un automate fini est un quintuplet $A =< \Sigma, Q, \delta, q_0, F >$ où :
— Σ est un ensemble fini de symboles appelé alphabet.
— Q est un ensemble fini dont les éléments sont appelés états.
— δ est une relation de $Q \times \Sigma \times Q$ est l'ensemble des transitions de A.
— q_0 est un état de Q appelé état initial.
— F est un sous-ensemble de Q appelé ensemble des états finaux de A.

L'ensemble des transitions δ est une relation, c'est-à-dire un ensemble de triplets $< q, a, q' >$. Cet ensemble est nécessairement fini puisque Q et Σ le sont.

Un automate fini est fait de composantes qui sont toutes finies $< \Sigma, Q, \delta, F >$, d'où le qualificatif de fini[6].

1.5.2.2 Représentation d'un automate d'états fini

Un automate fini peut être vu comme un graphe orienté étiqueté : les états sont les sommets et les transitions sont les arêtes étiquetées. L'état initial est marqué par une flèche entrante ; un état final est, selon les auteurs, soit doublement cerclé, soit marqué d'une flèche sortante.

On peu éventuellement représenter un graphe en expliquant les cinq paramètres formels avec en particulier la définition précise de δ sur chaque élément du domaine lorsque δ existe pour un couple de donné.

Une autre représentation est aussi possible elle est matricielle avec sur une ligne quelconque de la matrice $q_j = \delta(q_i, x_j)$ où q_j est obtenu à partir de q_i en lisant x_j. [5]

Exemple

Soit l'automate d'alphabet $\{a, b\}$, de la figure 1.2.

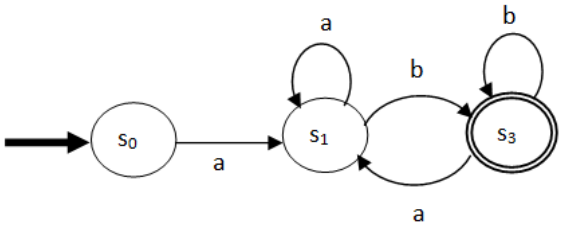


FIGURE 1.2 – Exemple d'automate d'états fini.

Les mots reconnus par cet automate sont : *ab*, *aab*, *abb*, *aaab*, *aabb*, *abab*, *abbb* ... et plus généralement tous les mots commençant par a et finissant par b.

Il existe des règles sur l'ensemble δ qui dénotent deux types d'automate :

— l'automate fini non déterministe (AFN),

— l'automate fini déterministe (AFD).

Formellement, les définitions des AFN et des AFD sont similaires : dans les deux cas, il faut préciser un alphabet, un ensemble d'états, une fonction de transition, un état initial et un ensemble d'états finaux. Elles diffèrent cependant dans la nature de la fonction de transition : dans un automate déterministe, la fonction de transition associe à tout couple constitué d'un état et d'un symbole, un unique état $q \in Q$ accessible en une transition. Dans un automate non déterministe, la fonction prend un couple constitué d'un état et d'un symbole ou du mot vide et lui associe un ensemble d'états $q \subseteq Q$ accessibles en une transition.

1.5.2.3 Automate fini non déterministe

Définition formelle

Un automate fini non-déterministe (AFN) est défini par un quintuplet $A =< \Sigma, Q, q_0, F, \delta >$, où :

— Σ est un ensemble fini de symboles (l'alphabet),

— Q est un ensemble fini d'états,

— q_0 est l'état initial,
— $F \subseteq Q$ est l'ensemble des états finaux,
— $\delta \subset Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ est une relation ; chaque triplet $< q, a, q' > \in \delta$ est une transition.[7]

Dans ce type d'automate, il peut y avoir plusieurs transitions avec la même lettre. Celles-ci peuvent être aussi spontanées, c.-à-d. la transition peut être étiquetée par le mot vide. Une transition spontanée permet de passer d'un état à l'autre d'un automate sans lire de symbole. Cela permet de programmer facilement des automates complexes.

Exemple

La figure 1.3 montre un automate fini non déterministe qui accepte tout les mots exprimés par l'expression régulière $aa^*|bb^*$.

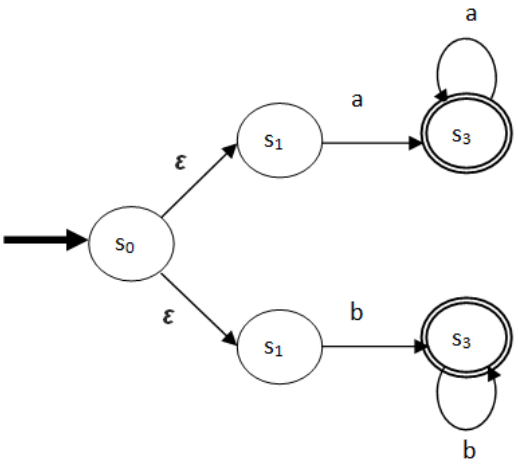


FIGURE 1.3 – Exemple d'automate d'états fini indéterministe.

Construction d'un automate fini non déterministe

Pour construire un automate reconnaissant un langage régulier, il est plus pratique d'exprimer ce langage sous forme d'une expression régulière, puis à partir de celle-ci construire l'automate correspondant.

Pour tout automate, il existe une expression régulière reconnaissant le même langage. On peut donc transformer un automate en expression régulière. [8]

Réciproquement, pour toute expression régulière, il existe un automate reconnaissant le même langage et peut donc transformer une expression régulière en automate.

Il existe plusieurs méthodes pour construire un automate fini non déterministe à partir d'une expression rationnelle :

1. **la méthode de Thompson**
   Elle a été utilisée par Ken Thompson dans l'implémentation de la commande *grep* du système Unix. On construit récursivement des automates pour les composants d'une expression. La forme particulière des automates permet de les combiner avec une grande facilité. L'automate obtenu est non déterministe ;

2. **la méthode de Glushkov**
   Cette méthode, attribuée à l'informaticien Glushkov, permet de construire un automate non déterministe de même taille (nombre d'états) que la taille (nombre de symboles) de l'expression rationnelle. Il a été observé que l'automate de Glushkov est le même que l'automate obtenu en supprimant les ε-transitions de l'automate de Thompson ;
3. **la méthode des quotients, résiduels ou dérivées, de Brzozowski**
   On forme les quotients (ou résiduels) successifs de l'expression. Il n'y en a qu'un nombre fini de différents, après application d'un certain nombre de règles de simplification qui sont l'associativité, la commutativité et l'idempotence de l'opération.

Aucune de ces méthodes ne donne directement l'automate déterministe d'un langage qui sera plus efficace. Donc pour avoir un automate fini déterministe il rendre faudra détérminiser l'AFN.

Détéminisation d'un automate fini non déterministe

Comme l'ordinateur est incapable de gérer un processus qui n'est pas déterministe, il est primordial de pouvoir déterminiser.

Le déterminisme est le fait de ne pas avoir le choix entre plusieurs exécutions. Pour un automate donné, c'est le fait que les transitions soient déterminées uniquement par le mot que l'on teste.

Or dans un AFN ce principe n'est pas respecté puisque une transition peut s'effectuer par plusieurs mots ou le mot vide : on peu prendre plusieurs chemin à partir d'un état, on aura donc un automate qui reconnaît plusieurs mots à la fois.

Il est possible, à partir d'un automate fini non déterministe A, de construire un automate fini déterministe A' reconnaissant le même langage.

Si $D =< Q_D, \Sigma, \delta_D, q_{0_D}, F_D >$ est l'AFD construit à partir de l'AFN $N =< Q_N, \Sigma, \delta_N, q_{0_N}, F_N >$ par construction des sous-ensembles d'états, alors $L(D) = L(N)$.[9] À partir de ce résultat, on peut conclure que :

— pour un mot donné, un automate déterministe n'a qu'une seul exécution possible,
— un AFD est un AFN particulier,
— pour chaque automate AFN, il existe un AFD équivalent. [5]

1.5.2.4 Automates fini déterministe

Définition formelle

Un automate fini déterministe (AFD) est défini par un quintuplet $A =< \Sigma, Q, q_0, F, \delta >$, où :

— Σ est un ensemble fini de symboles (l'alphabet),
— Q est un ensemble fini d'états,
— q_0 est l'état initial,
— $F \subseteq Q$ est l'ensemble des états finaux,
— $\delta \subset Q \times \Sigma \times Q$ est une relation ; chaque triplet $< q, a, q' > \in \delta$ est une transition.[7]

À la différence de l'automate fini non déterministe les transitions ne peuvent être étiquetées par le mot vide (il n y a pas de transition spontanée) et à partir d'un état, on ne peut accéder un autre état qu'avec un seul mot.

Construction d'un automate fini déterministe

Il existe un algorithme qui permet de transformer n'importe quel automate en automate déterministe équivalent. Cette algorithme sera présenté en détail dans le prochain chapitre.

Le principe est de supprimer les transitions spontanées et de déterminer les états accessibles depuis un ensemble d'états par les transitions selon un symbole donné de l'alphabet.

1.6 Conclusion

Dans ce chapitre, nous avons présenté les différentes notions inhérentes aux langages formels. Parmi les différentes classes de langages, nous avons attaché un plus grand intérêt aux langages réguliers puisqu'ils sont au centre de notre projet.

En effet, après avoir vu ce qu'est un langage régulier et décrit le fonctionnement des outils qui permettent de le manipuler et de le définir, à savoir la grammaire régulière, les expressions régulières et les automates finis, le chapitre qui va suivre aura pour contenu la conception d'une application qui, entre autres, illustrera la relation entre ces différents concepts.

Ce projet qui mettra en jeu ces différents concepts, a comme objectif principal, rappelons-le, l'implémentation d'un générateur automatique d'automates finis déterministes à partir d'expressions régulières.

Chapitre 2

Conception

2.1 Introduction

Après avoir présenté les concepts majeurs relatifs aux langages formels, on peu affiner la formulation de l'objectif de notre projet qui consiste à mettre au point un système permettant d'analyser et de reconnaître des expressions régulières données en entrée et de générer l'automate fini déterministe correspondant. Autrement dit ce système doit indiquer si une expression régulière appartient ou non au langage prédéfini, et si c'est le cas générer l'automate « reconnaisseur » de cette expression. Il faut donc définir au préalable un langage d'expressions régulières, qui doit être assez riche pour exprimer les langages réguliers de manière exhaustive.

Dans ce chapitre nous allons présenter les différentes étapes suivies afin de concevoir un procédé de réalisation, mais avant on propose de décrire les différentes procédures qui le composent et qui :

- mettre au point un analyseur lexico-syntaxique et sémantique qui génère l'arbre syntaxique,
- construire un générateur d'AFN qui fera correspondre à chaque arbre obtenu un automate fini non déterministe,
- construire un générateur de DFA qui rendra l'AFN obtenu déterministe, et
- enfin construire un générateur de graphe pour obtenir une représentation de l'AFN sous forme d'une image graphique.

L'organigramme de la figure 2.1 illustre l'enchaînement des procédures et la relation entre elles ainsi que le résultat de chacune d'elle.

Une expression régulière lue en entrée doit subir une analyse lexico-syntaxique et sémantique, dont le résultat est sa représentation sous forme d'un l'arbre syntaxique abstrait. Ensuite, c'est à partir de cet arbre que le générateur d'AFN construit l'AFN reconnaissant l'expression d'entrée. Le générateur d'AFD se charge de rendre l'AFN obtenu déterministe donc de générer l'AFD équivalent. Enfin, ce dernier sera traduit en fichier image graphique et enregistré sur disque.

La conception de l'application requiert donc la conception :

- d'un langage de spécifications régulières,
- de trois analyseurs lexical, syntaxique et sémantique pour ce langage,
- d'un générateur d'AFN à partir d'une expression régulière,
- d'un générateur d'AFD équivalent à un AFN donné,
- d'un générateur d'images graphiques à partir de la représentation interne d'un AFD.

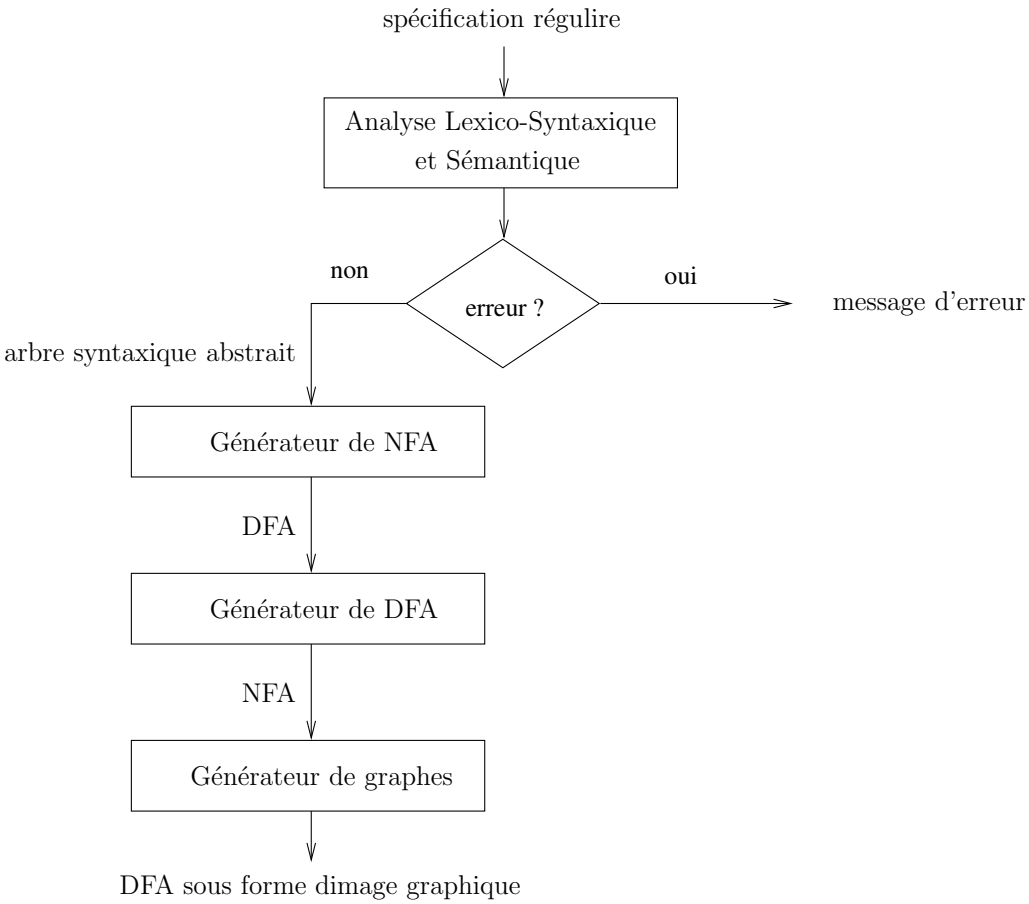


FIGURE 2.1 – Organigramme du système.

2.2 Conception d'un langage de spécifications régulières

2.2.1 Le langage de spécifications régulières

Notre conception d'un langage de spécifications régulières est guidée par le soucis de permettre à l'utilisateur l'écriture d'expressions compactes et lisibles.

Nous devons donc permettre l'utilisation :

— des différentes abréviations, et

— de définitions de notions utilisables dans des expressions ultérieurs.

En somme, l'utilisateur qui aura à fournir une expression régulière (celle pour laquelle il voudrait obtenir l'AFD) dans laquelle il pourra utiliser la quasi-totalité des abréviations courantes, pourra également la faire précéder d'une définition régulière et d'utiliser les notions définies dans son expression. Une spécification régulière sera alors une suite de déclarations, suivie d'une expression.

Exemple

Une spécification régulière de nombres décimaux non signés serait :

```
chiffre=[0-9] ;
```

```
chiffres={chiffre}+ ;

{chiffres}
```

Comme le langage des expressions régulières est lui même un langage algébrique, la grammaire de type 2, en notation *BNF* (Backus & Naur Form [11]), qui engendre notre langage, est donnée à la figure 2.2.

$$
\begin{array}{lcl}
< description > & ::= & < declaration >< expression > \\
 & & \\
< declaration > & ::= & < declaration > NOTION = < expression > ; \\
 & | & \varepsilon \\
< expression > & ::= & < expression > < expression > \\
 & | & < expression > '|' < expression > \\
 & | & < expression > {}^{*} \\
 & | & < expression > {}^{+} \\
 & | & < expression > ? \\
 & | & < expression > \{\ num\ \} \\
 & | & < expression > \{\ num\ ,\ num\ \} \\
 & | & < expression > \{\ num\ ,\ \} \\
 & | & < expression > \{\ ,\ num\ \} \\
 & | & (\ < expression >\ ) \\
 & | & [\ < intervalle >\ ] \\
 & | & CAR\_SIMPLE \\
 & | & CAR\_PREDEF \\
 & | & CAR\_SPECIAL \\
 & | & VIDE \\
 & | & NOTION \\
< intervalle > & ::= & < intervalle >< intervalle > \\
 & | & INTER \\
 & | & CAR\_SIMPLE \\
 & | & CAR\_PREDEF \\
 & | & CAR\_SPECIAL
\end{array}
$$

FIGURE 2.2 – Grammaire engendrant le langage.

Dans cette grammaire, les non-terminaux sont en lettres minuscules et écrits (comme l'impose la notation BNF) entre < et >. Les terminaux, quand ils ne sont pas réduits à un seul caractère, sont désignés par des mots en lettres majuscules, représentant des unités lexicales.

Notons que les expressions régulières ne sont pas limitées aux expressions de base, mais autorisent toutes les abréviations couramment utilisées.

Les caractères uniques sont répartis en trois classes :

— les *caractères simples* tels que a, 4, <, :, ... écrits tels quels,

— les *caractères prédéfinis* tels que +, −, [, }, ... écrits en les faisant précéder du caractère \. À titre d'exemple, \+ désigne le caractère + et non l'opérateur d'itération positive de Kleene. En général, $\backslash X$ désigne le caractère X,

— les *caractères spéciaux* tels que $\backslash n$, $\backslash t$, ... Ici, $\backslash n$ ne désigne pas n mais le caractère *newline* de code ASCII 10.

$INTER$ désigne une plage de caractères de la forme $c_1 - c_2$, c-à-d, un ensemble de tous les caractères dont les codes ASCII sont compris entre ceux des caractères c_1 et c_2.

Nous voudrions également permettre aux utilisateurs d'insérer des commentaires dans leurs spécifications. Nous considérons alors, que toute séquence de caractères commençant par @ et se terminant en fin de ligne est un commentaire.

2.2.2 Conception des analyseurs lexical, syntaxique et sémantique

Après avoir défini la grammaire qui engendre notre langage des spécifications régulières, la seconde étape consiste à en élaborer des analyseurs lexical, syntaxique et sémantique basés sur cette grammaire.

Ce sont deux analyseurs qui travailleront en collaboration en ayant pour objectif de reconnaître si un mot en entrée est correct, et de produire un arbre syntaxique abstrait qui représente la structure syntaxique abstraite de cette entrée, autrement dit, le but de cette analyse est de passer de la syntaxe concrète à la syntaxe abstraite. [11]

2.2.2.1 L'analyseur lexical

Le rôle de l'analyse lexicale est de transformer une suite de caractères en une suite de mots, dits *lexèmes* et de produire en sortie des unités lexicales (ou *tokens*) qui seront utilisées dans l'analyse syntaxique. L'analyseur lexical est défini par un ensemble d'expressions régulières qui définissent les séquences de caractères possibles, utilisées pour former des tokens.

Le tableau 2.1 énumère les différents tokens, leurs modèles, ainsi que leurs attributs. [4]

Dans le tableau 2.1,

— la plage $[\backslash 200 - \backslash 377]$ dans le modèle du token CAR_SIMPLE désigne tous les caractères de codes ASCII octaux compris entre 200 (128 en décimal) et 377 (255 en décimal),

— le caractère c qui apparaît dans le modèle du token $INTER$ désigne tout caractère simple, prédefini ou spécial. On a donc :
$c = [!"\%\&'/A - Za - z0 - 9 :< - > \_ `\sim \backslash 200 - \backslash 377] \mid \backslash[-\#\backslash + \{\} *; , = |()[]?@] \mid \backslash[ntrvfba]$

Outre la reconnaissance des unités lexicales, les analyseurs lexicaux assurent certaines tâches comme le saut des blancs et des commentaires, *la gestion des fichiers* et la gestion des erreurs. Le rôle assigné à un analyseur lexical est en quelque sortes de simplifier l'écriture d'un analyseur syntaxique. Les unités lexicales produites par l'analyseur lexical, sont transmises à l'analyseur syntaxique afin d'être traitées par celui-ci.

Token	Modèle	Attribut
PV	;	
$EGAL$	$=$	
OR	|	
$ITER$	$*$	
$ITERP$	$+$	
OPT	?	
ACG	{	
ACD	}	
NUM	$\#[0-9]^+$	*valeur entière*
VIR	,	
$PARG$	(	
$PARD$	)	
CRG	[	
CRD	]	
$VIDE$	$\pounds$	
CAR_SIMPLE	$[!"\%\&'/A-Za-z0-9:<->\_`\sim \backslash 200-\backslash 377]$	*caractère*
CAR_PREDEF	$\backslash[-\#\backslash+\{\}*;,=\|()[]?@]$	*caractère*
$CAR\_SPECIAL$	$\backslash[ntrvfba]$	*caractère*
$INTER$	$c-c$	$<$ *caractère*, *caractère* $>$
$NOTION$	$\{[A-Za-z\_][A-Za-z\_0-9]*\}$	*nom*

TABLE 2.1 – Description des tokens.

2.2.2.2 L'analyseur syntaxique

Le but de l'analyse syntaxique est de déterminer si une spécification régulière lue en entrée, peut être engendrée par la grammaire décrivant notre langage. En d'autres termes, il s'agit de tenter de construire la dérivation permettant de l'engendrer.

Dans le cas où la spécification n'est pas dérivable, l'analyse syntaxique affichera un message d'erreur et en donnera un diagnostic.

Développer un analyseur syntaxique à la main ne serait pas un meilleur choix. Le code pourrait être complexe, répétitif et difficile à maintenir. Le risque d'erreur est important.

Nous préférons alors spécifier ce que doit faire l'analyseur et le construire automatiquement à partir de la grammaire grâce au générateur YACC produisant des analyseurs syntaxiques basés sur la méthode d'analyse ascendante dite $LALR(1)$.[4]

2.2.2.3 L'analyseur sémantique

L'analyse sémantique permet de faire un contrôle pour s'assurer que l'assemblage des constituants des expressions a un sens.[11, 4]

Certaines expressions peuvent être syntaxiquement correctes mais ne pas avoir de sens par rapport à la sémantique de notre langage. À titre d'exemple, l'expression régulière $a\{5, 2\}$ est syntaxiquement correcte mais sémantiquement incorrecte.

Les différents contrôles que devra assurer notre analyseur sémantique sont :

— les contrôles de bornes pour les itérations comportant une limite inférieure et supérieur, et pour les plages de caractères,
— maintenir une table des notions indiquant l'expression régulière constituant la valeur de chacune d'elle,
— vérifier qu'une notion référencée est préalablement définie (donc présente dans la table), et
— empêcher la redéfinition d'une notion.

En outre, l'analyseur sémantique aura comme tache, de produire la représentation abstraite de l'expression régulière fournie par l'utilisateur. Dans un arbre syntaxique abstrait, il y a deux types de nœuds :
— les nœuds terminaux étiquetés par un caractère quelconque (simple, prédéfini ou spécial) ou par de token VIDE, représentant le mot vide et transcrit par le caractère £ disponible sur les claviers,
— les nœuds intérieurs étiquetés par les opérateurs tels que $*$, ?, |, ...

C'est la structure de cet arbre qui va guider par la suite la génération de l'AFN en utilisant la méthode de Tompson.

L'exemple suivant illustre le passage d'une expression régulière à l'arbre syntaxique équivalent.

Exemple

Soit l'expression régulière $(ab)^*$; L'arbre syntaxique que devra produire l'analyseur sémantique est donné à la figure 2.3.

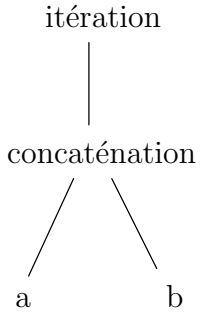


FIGURE 2.3 – Représentation abstraire de l'expression régulière $(ab)^*$.

2.2.3 Conception du génération d'automates d'états finis indéterministes

Un automate est une procédure effective (un algorithme) reconnaisseur de langage, c-à-d, permettant de déterminer si un mot donné appartient à un langage.

À la classe des langages réguliers correspond une classe particulière d'automates : la classe des automates d'états finis.

Il existe de nombreuses stratégies pour construire un automate fini reconnaissant le langage dénoté par une expression régulière. La méthode retenue et que nous allons présenter ici est l'algorithme de construction de *Thompson*, une variante simple, efficace et surtout facile à implanter, ce qui n'est pas le cas de toutes les méthodes.

L'algorithme est dirigé par la syntaxe, c'est-à-dire qu'il utilise la structure syntaxique de l'expression régulière pour guider le processus de construction. Il est récursif sur l'arbre syntaxique de l'expression régulière. Cet algorithme existe avec différentes variantes, notamment sur la manière de traiter la concaténation.

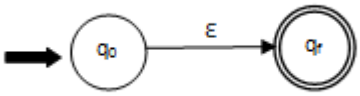
Propriété de l'AFN construit avec la méthode de Thompson

L'AFN construit avec la méthode de Tompson se caractérise par :
— un unique état initial q_0 ;
— un unique état final q_f ;
— exactement deux fois plus d'états que de symboles dans l'expression régulière (sans compter les parenthèses ni la concaténation implicite).

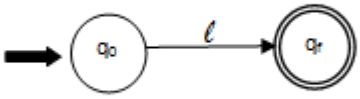
Fonctionnement de la méthode de Thompson

L'algorithme de Thompson décompose l'expression régulière en ses composants élémentaires, puis applique les constructions de Thompson de ces composants pour construire l'automate correspondant.

1. Automate pour le mot vide ε

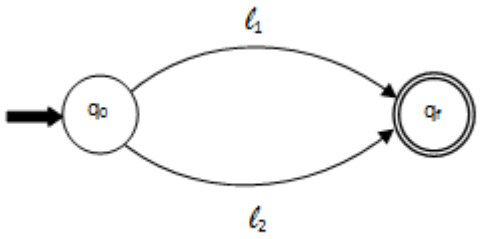


2. Automate pour un littéral l

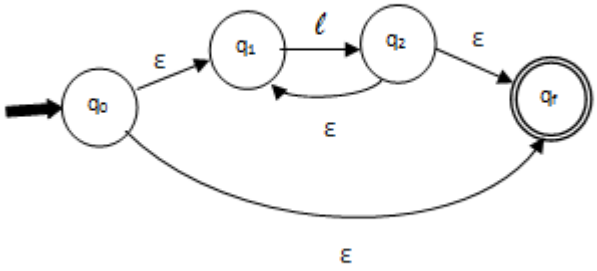


3. Automate pour la concaténation $l_1 l_2$

4. Automate pour l'alternative $l_1 \,|\, l_2$



5. Automate pour l'itération l^*



Exemple

L'AFN construit par la méthode de *Tompson* pour l'expression régulière (a|b)* est donné par la figure 2.4.

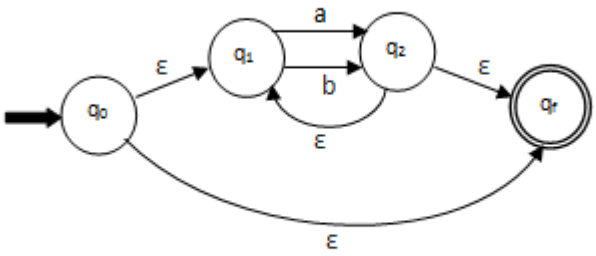


FIGURE 2.4 – AFN pour l'expression $(a|b)^*$.

L'algorithme de construction

L'algorithme 2.1 constitue notre algorithme d'implémentation de la méthode de Thompson. Nous le concevons comme une fonction récursive appelée $nfa()$, vu qu'il doit considérer de manière récursive les différents constituants d'une expression régulière.

Algorithme 2.1 Construction d'un AFN à partir d'une expression régulière.

```
nfa(er,ei,ef){
  selon(er){
            cas epsilon : creer_transition_epsilon(ei,ef);
            cas caractere (c) : creer_transition_caratere(ei,ef,c);
            cas concatenation (l1, l2): { e=creer_etat();
                                          nfa(l1, ei, e);
                                          nfa(l2, e, ef);
                                        }
            cas alternative (l1, l2): {  nfa(l1, ei, ef) ;
                                         nfa(exp_d, ei, ef);
                                      }
            cas itération (l) : { e1=creer_etat() ;
                                  e2=creer_etat();
                                  creer_transition_epsilon(ei,ef);
                                  creer_transition_epsilon(ei,e1);
                                  creer_transition_epsilon(e2,ef);
                                  creer_transition_epsilon(e2,e1);
                                  nfa(l, e1, e2);
                                }
  }
}
```

L'algorithme 2.1 parcourt en profondeur et en commençant par la racine, les différents nœuds de l'arbre abstrait représentant une expression régulière, et construit les différents états et transitions d'AFN tel que indiqué par la méthode de Tompson.

La fonction prend comme arguments :
— l'expression régulière (`er`) pour laquelle nous souhaitons construire l'AFN,
— deux états (`ei` et `ef`) de l'AFN entre lesquels appliquer les règles de Thompson.

Notre fonction `nfa()` utilise trois autres fonctions :
— `creer_etat()` permettant de créer un nouvel état et de l'ajouter à ceux de l'AFN,
— `creer_transition_epsilon(ei, ef)` permettant de créer une transition spontanée entre les états `ei` et `ef`,
— `creer_transition_caratere (c, ei, ef)` permettant de créer entre les états `ei` et `ef`, une transition sur le caractère `c`.

Pour utiliser l'algorithme 2.1, il suffit de :
— créer un état initial `e0` et un état final `ef`,

— d'appeler la fonction `nfa()` en lui fournissant comme arguments :
  — la racine de l'arbre syntaxique abstrait construit par l'analyseur sémantique,
  — les états `e0` et `ef`.

2.2.4 Conception du génération d'automates d'états finis déterministes

La *déterminisation* permet de passer d'un automate non-déterministe, construit à l'étape précédente, à son équivalent déterministe. Cette transformation se base sur la définition d'une relation d'équivalence sur les états du DFA.

La méthode consiste à définir des transitions sur des ensembles d'états et non des états. Chaque ensemble d'états de l'AFN correspond à un état de l'automate déterministe (l'AFD). Un état est final si un des états d'origines le composant est final.

Le principe est que si dans un automate indéterministe, il existe deux chemins (q0, qn) et (q0, qm) décrivant le même mot, alors ces deux chemins appartiennent à la même classe d'équivalence. Ils peuvent donc être regroupés en un seul chemin pour ce mot. Cette constatation est à la base de la méthode déterminisation.

Algorithme 2.2 Construction d'un AFD équivalent à un AFN.

```
dfa(AFN){
   litteral=littéraux(AFN);     // litteral=pointeur vers la liste
                                // des littéraux de l'AFN

   ed0=epsilon_transition(en0);  // en0=pointeur vers l'état 0 de
                                 // l'AFN
                                 // ed0=pointeur vers l'état 0 de
                                 // l'AFD
   ed=ed0;
   tantque(ed){
     tantque(litteral){
       l=transition(ed, litteral);  // l=liste des etats de l'AFN
                                    // vers lesquels transite ed
                                    // avec litteral

       el=epsilon_transition(l);  // el=liste des
                                  // epsilon-transitions des
                                  //éléments de la liste l
       si( ! existe(el))
          ajouter_etat(el);
       litteral=litteral->suivant;
     }
     ed=ed->suivant;
    }
}
```

Pour mettre en oeuvre cette méthode de détéminisation, nous proposons l'algorithme 2.2 présenté comme une fonction.

L'algorithme 2.2 construit tout d'abord l'état initial de l'AFD à partir de la liste des *epsilon − transitions* (états atteignables uniquement par transitions spontanées) de l'état 0 de l'AFN. Puis à partir de cette liste, pour chaque littéral, on forme la liste des états cibles avec la fonction `transition()`, puis la liste des epsilon-transitions de celle-ci avec la fonction `epsilon_transition ()`. Si la liste des epsilon-transitions existe déjà, on garde le littéral courant et l'état courant de l'AFD, sinon on crée un nouvel état de l'AFD en lui associant cette liste, puis on passe à l'état suivant.

Le processus est réitéré jusqu'à ce que toutes les listes construites de cette manière se répètent et que tous les états de l'AFD ait été traités.

Pour obtenir un automate d'états fini déterministe, il suffit de faire appel à la fonction `dfa()` et lui fournir comme argument l'AFN construit à l'étape précédente.

2.2.5 Conception du générateur d'images graphiques

Une fonction d'affichage permet d'afficher les états et leurs transitions sous forme de ligne de texte. Mais afin de mieux visualiser le résultat, on représentera l'AFD sous forme d'une image graphique.

Or la représentation graphique des graphes est un problème algorithmique ardu. La conception d'un programme offrant une telle fonctionnalité est une tâche de longue haleine qui requiert de fortes compétences en mathématiques et algorithmique. Ainsi, il est utile de recourir à un programme externe à qui on délègue la génération des graphes. Il existe en effet l'application *Graphviz* qui permet de générer des graphes sous divers format.

Les programmes de mise en page Graphviz prennent des descriptions de graphiques dans un langage de texte simple, et les transforme en diagrammes dans des formats utiles, tels que des images et SVG pour les pages web, PDF ou Postscript pour l'inclusion dans d'autres documents, ou pour afficher dans un navigateur graphique interactif.

Nous représenterons l'AFD à la manière conventionnelle, à savoir :

— les états seront représentés par des cercles s'ils sont non finaux et par des double cercles sinon. Avec une flèche qui pointe vers l'état initial.

— les transitions seront sont représentées par des arcs ciblés qui partent d'un état à un autre vers lequel il transite. Ces arcs portent des étiquettes représentant le caractère de transition.

L'outil Graphviz permet de définir une telle description au moyen du langage *dot*. La génération d'une image graphique d'un AFD consiste à transcrire la structure de ce dernier en langage dot.

Pour automatiser la génération de l'image graphique d'un AFD, nous proposons l'algorithme 2.3, décrit comme une fonction.

Algorithme 2.3 Générateur de l'image graphique d'un AFD.

```
gen_image_AFD (AFD, fichier_dot){  //  fichier_dot est le nom du
                                   //  fichier de sortie

   ouvrir(fichier_dot);  // ouvrir le ficher de sortie en écriture

   etat=AFD;             // etat pointe sur l'état 0 de l'AFD

   tantque(etat){
     si etat(etat=ETAT_FINAL)
        ecrire(fichier_dot,
                  < description d'état final dans le langage dot>);
     si non
        ecrire(fichier_dot,
           < description de l'état non final dans le langage dot>);
     lt=transitions(etat);
     tantque(lt){
        ecrire(fichier_dot,
              < description de la transition dans le langage dot>);
        lt=lt->suivant;
     }
       etat=etat->suivant;
   }
}
```

La fonction `gen_image_AFD()` prend deux paramètres : un pointeur de la structure de l'AFD et le non du fichier de sortie. Elle produit en sortie un fichier *dot* qui contient la description en langage dot des états et des transitions de l'AFD.

2.3 Conclusion

Après avoir décrit la conception des différents modules de notre projet, ainsi que les différent concepts qui entre en jeu, nous passerons dans le chapitre suivant à la pratique où seront présentés la réalisation de l'application et sa mise en œuvre.

Nous présenterons particulièrement, l'implémentation des différents algorithmes présentés précédemment, ainsi que les résultats obtenus pour chaque étape.

Cette étape est la partie pratique qui consolidera et confirmera ce qui a était avancé en théorie.

Chapitre 3

Réalisation

3.1 Introduction

Après avoir étudié tous les aspects de notre projet et décrit sa conception, la présente étape consiste à réaliser ce projet. En effet nous allons consacrer ce chapitre à la programmation et à la présentation des différents outils qui ont permis de réaliser l'application.

Pour chaque étape, nous réaliserons l'implémentation du programme qui permettra de répondre à la tache qui y est définie et, puisque les étapes sont dépendantes, nous présenterons à la fin de chaque tache accomplie le résultat obtenu.

Mais, commençons d'abord par présenter l'environnement et les outils qui ont permis cette réalisation.

3.2 Environnement et outils de programmation

Premièrement, il faut choisir un environnement de travail ; dans notre cas le choix s'est porté sur l'environnement Linux et précisément sur la distribution communautaire Kali Linux qui est dérivée de la distribution majeure Debian.

Les différents logiciels dont il faut disposer pour les besoins de la réalisation sont : un éditeur de texte pour écrire du code, un compilateur, un générateur d'analyseurs lexicaux, un générateur d'analyseurs syntaxiques, un générateur de graphes, et enfin un logiciel d'affichage d'images.

Or cette distribution dispose déjà par défaut de certain de ces logiciels comme l'éditeur de texte gedit et le compilateur gcc. Pour les autres il faudra les installer, mais avant, on les présentera d'abord et on expliquera leurs fonctionnements.

3.2.1 Le générateur d'analyseurs lexicaux flex

3.2.1.1 Présentation et Installation

flex est une version libre du générateur d'analyseurs lexicaux Lex ; c'est un outil pour générer des analyseurs lexicaux : programmes qui reconnaissent des motifs lexicaux dans du texte.[11]

Avant de lancer l'installation de flex dans notre environnement Linux, il faut d'abord mettre à jour le système avec les deux commandes :

```
apt-get update
apt-get upgrade
```

Ensuite, il suffit juste de lancer la commande :

```
apt-get install flex
```

Touts les paquets nécessaires à cet outil seront installés.

3.2.1.2 Fonctionnement

flex lit les fichiers d'entrée donnés en arguments, ou bien son entrée standard si aucun fichier n'est donné, pour obtenir la description de l'analyseur à générer.

La description est une liste de paires d'expressions rationnelles et de code C, appelées règles. En sortie, flex génère un fichier source en langage C, appelé `lex.yy.c`, qui définit une routine `yylex()`. Ce fichier est compilé afin de produire un programme exécutable. Quand l'exécutable est lancé, il analyse son entrée afin d'y trouver des occurrences correspondant aux précédentes expressions régulières. À chaque expression trouvée, il exécute le code C correspondant.

3.2.1.3 Structure d'une spécification flex

L'écriture d'une spécification flex est soumise à des règles tel que décrit par la figure 3.1.

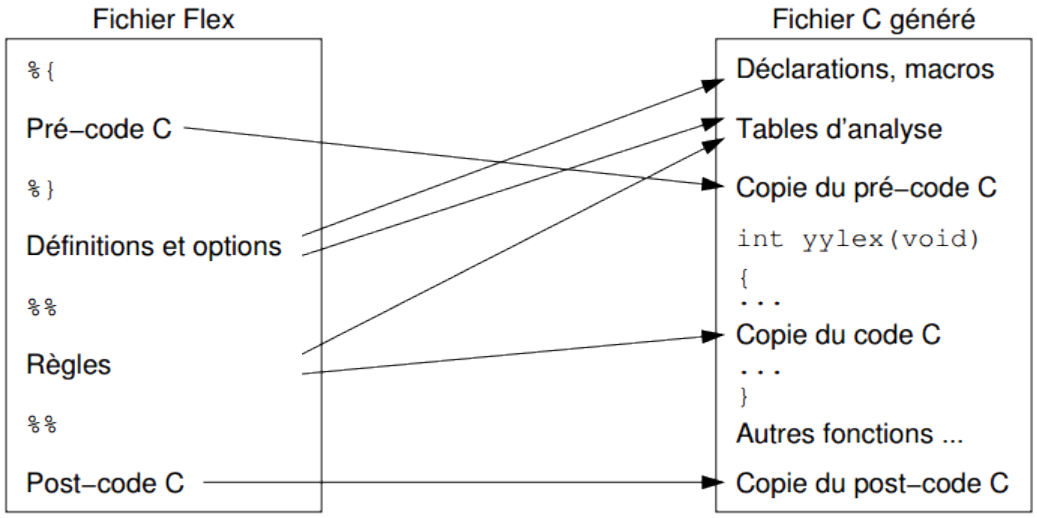


FIGURE 3.1 – Structure d'une spécification flex.

Section déclarations

Cette section comprend des déclarations de variables, des constantes littérales et des définitions régulières.

— Pré-code C : c'est du code C qui sera recopié dans le généré,
— Options : ce sont des paramètres de fonctionnement,
— Définitions : ce sont des définitions régulières.

Section règles

Les règles sont de la forme

```
motif action
```

Un *motif* est une expression régulière et l'*action* est normalement un fragment de programme en C qui décrit quelle action l'analyseur lexical devrait réaliser quand un lexème concorde avec le motif.

La chaîne qui correspond au modèle est placée dans une variable prédéfinie de type `char * ` nommée `yytext`.

Les actions sont normalement placées entre accolades de style C (`{}`).

Section fonctions auxiliaires

Elle contient des fonctions en C qui pourraient être utilisées dans les actions.

3.2.2 Le générateur d'analyseurs syntaxiques yacc

3.2.2.1 Présentation et installation

yacc est un générateur d'analyseurs syntaxiques basés sur la méthode LALR(1). Conçu à l'origine pour Unix et pour le langage C, son utilisation s'est étendue sur d'autres systèmes (Windows) et pour d'autres langages (JAVA).

yacc est l'acronyme de *Yet another compiler-compiler* (et un compilateur de compilateurs de plus), reflétant la popularité des constructeurs d'analyseurs au début des années 70. La première version de yacc fut conçue par Stephen Curtis Johnson en 1975 et fut développée aux laboratoires Bell (ensemble avec C et Unix). Les dernières versions connues de ce générateur sont : *bison* (version GNU de yacc) et *byacc* (Berkeley yacc).

Pour les besoins de notre application, on a choisit de travailler avec la dernière version de *byacc*. La commande qui permet d'installer cet outil est :

```
apt-get install byacc
```

3.2.2.2 Fonctionnement

On prépare tout d'abord un fichier, par exemple `parser.y`, qui contient une spécification pour yacc. Lors de la compilation avec la commande `byacc`, cette dernière produit un programme C appelé `y.tab.c`, constituant l'implémentation d'un analyseur syntaxique basé sur la méthode LALR(1), complétée par d'éventuelles routines C de l'utilisateur.

En compilant `y.tab.c`, on obtient l'exécutable `a.out` désiré qui effectue la traduction spécifiée par le programme yacc original.

3.2.2.3 Structure d'un spécification yacc

Une spécification yacc a la même structure qu'une spécification flex. Comme montrée sur la figure 3.2, elle est aussi scindée en trois sections.

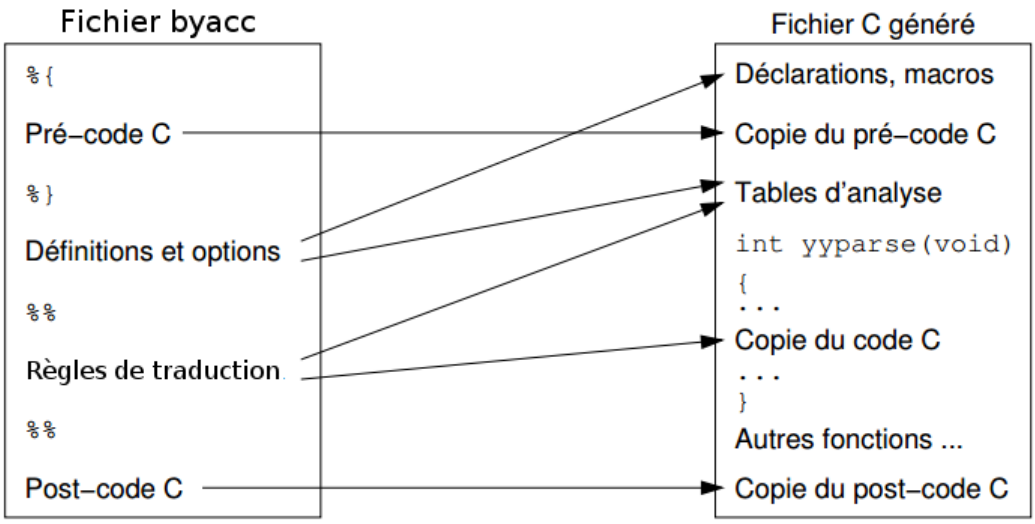


FIGURE 3.2 – Structure d'une spécification yacc.

Section déclarations

Cette sections contient deux parties optionnelles. La première contient des déclarations C ordinaires, délimitées par `%{` et `%}`. On inclut ici les déclarations des variables temporaires utilisées par les règles de traduction ou des fonctions utilisées dans la deuxième et troisième section.

La seconde partie contient des déclarations d'unités lexicales de la grammaire. Ces unités lexicales peuvent être utilisées dans la deuxième et la troisième partie.

Section règles de traduction

Cette partie se situe après le premier marqueur `%%`. Dans cette partie, on énonce les règles de traduction. Chaque règle est formée d'une production de la grammaire et de son action sémantique associée. Un ensemble de règles de production de la forme

$< partie\ gauche > \longrightarrow < alt1 > | < alt2 > | ... | < altn >$

s'écrirait en yacc

```
<partie gauche> : <alt1 >   {action sémantique 1}
                | <alt2>    {action sémantique 2}
                |  ...
                | <altn>    {action sémantique n}
;
```

A. Les règles de production

Dans une production yacc, un caractère simple entre apostrophes comme `'c'` est considéré comme désignant le symbole terminal `c`, et les chaînes de lettres et de chiffres non entourées d'apostrophes et non déclarées comme des unités lexicales, sont interprétées comme des non-terminaux.

Les différentes alternatives d'une même partie gauche peuvent être séparées par une barre verticale (`|`). Chaque production yacc se termine par un point-virgule (`;`). Les règles de traduction sont donc constituées d'une partie gauche, d'une série d'alternatives et d'actions sémantiques qui leur sont associées. La première des parties gauches est considérée comme l'axiome.

B. Les actions sémantiques

Une action sémantique est une séquence d'instructions en C. Dans une action sémantique, le symbole `$$` référence la valeur de l'attribut associé au non-terminal de la partie gauche. `$i` référence la valeur associée au $i^{ème}$ symbole de la grammaire (terminal ou non-terminal) en partie droite.

Section des routines annexes

La troisième partie des spécifications yacc contient des routines écrites en C, qui aident à la traduction. Un analyseur lexical nommé `yylex()` doit être fourni. D'autres fonctions comme les routines de récupération d'erreur peuvent être ajoutées si nécessaire.

L'analyseur lexical `yylex()` produit des couples formés d'une unité lexicale et de la valeur de l'attribut associé. Si une unité lexicale telle que `CHIFFRE` est retournée, celle-ci doit être déclarée dans la première section de la spécification yacc. La valeur de l'attribut associée à une unité lexicale est communiquée à l'analyseur syntaxique par l'intermédiaire de la variable *yylval* définie dans la partie déclarations de la spécification yacc.

3.2.3 Utilisation conjointe de Lex et de yacc

L'outil flex a été conçu pour produire des analyseurs lexicaux qui peuvent être utilisés avec des analyseurs syntaxiques construit avec yacc. La bibliothèque flex fournit une fonction `yylex()`, nom imposé par yacc pour son analyseur lexical.

Si nous utilisons flex pour produire l'analyseur lexical, nous remplaçons la routine `yylex()` dans la troisième partie de la spécification yacc par la clause

```
#include "lex.yy.c"
```

et nous spécifions alors chaque action flex de façon qu'elle retourne un terminal connu de yacc.

En utilisant la clause `#include "lex.yy.c"`, la fonction `yylex()` a accès aux noms que yacc a donnés aux unités lexicales puisque le fichier de sortie de flex est compilé comme étant une partie du fichier de sortie `y.tab.c`

3.2.4 Le générateur de graphe GraphViz

Graphviz est le diminutif de *Graph Visualization Software*, c'est un logiciel de visualisation de graphes. Il permet de représenter des données structurées sous la forme de diagrammes et de graphes. C'est un logiciel à diffusion libre disponible pour différentes distributions Linux ainsi que pour Windows ou MacOS X.[12]

En fait, Graphviz est un ensemble de programmes correspondant à l'implémentation de différents algorithmes. Il utilise des fichiers textes dans un format simple (DOT) pour décrire les graphes.

Les diagrammes peuvent être exportés sous différents formats d'images (FIG, postscript, GIF, JPEG, PNG,...).

3.3 Réalisation de l'application

Après l'installation et la préparation de l'environnement de travail et des outils nécessaires, on va pouvoir passer à la réalisation.

L'application est composée de plusieurs modules, dont chacun permet de réaliser une procédure, c'est pourquoi il faudra automatiser le processus de compilation en développant un fichier `Makefile`.

Le processus de compilation de tous les modules sera décrit dans ce fichier (Makefile) unique et la compilation, lancée avec la commande, `make` se chargera :

— d'effectuer une compilation séparée des différents fichiers source, et

— de lier les fichiers objets des différents modules afin de produire un seul ficher exécutable.

Maintenant que nous disposons de tout les outils nécessaires, nous allons entamer la réalisation de notre application. Pour ce faire, nous allons suivre le même enchaînement que pour la conception.

3.3.1 Implémentation des analyseurs lexical, syntaxique et sémantique

Le rôle principal assigné aux trois analyseurs est :

— retourner les unités lexicales,

— vérifier la conformité syntaxique et sémantique, et

— construire l'arbre syntaxique abstrait.

Grâce aux outils flex et yacc, nous allons pouvoir générer les trois analyseurs. Il suffit pour cela d'écrire des spécifications lex et yacc adéquates.

3.3.1.1 Implémentation de l'analyseur lexical

Cette étape consiste à écrire la spécification flex dans un fichier `lexer.l` qu'on aura créé, et d'après la syntaxe décrite précédemment, `lexer.l` sera constitué de trois parties.

La partie déclarations

— **Le pré-code**
  On déclare la fonction `void erreur_lexicale()` dont la définition sera écrite dans la partie *post_ code*.

— **Les définitions régulières**
  Dans cette partie, on doit déclarer les définitions régulières dont on aura besoin dans les règles de production. Ces définitions permettront de facilité l'écriture des motifs.

```
ident          [A-Za-z_][A-Za-z_0-9]*
notion         \{{ident}\}
nombre         #[0-9]+
car_simple     [!"%&'/A-Za-z0-9:<->_`~\200-\276\278-\377]
car_predef     \\[ #-\\+{ }*;,\.=|()\[ \] ^?@]
car_special    \\[ntrvfba]
caractere      {car_simple}|{car_predef}|{car_special}
plage          {caractere}\-{caractere}
blancs         [ \n\t]+
```

Rappelons juste que nous avons trois types de caractères : simples, prédéfinis et spéciaux.

— **Option**
On insère l'option

```
%option yylineo
```

pour indiquer à flex de maintenir la variable `yylineno` indiquant le numéro de la ligne en cours d'analyse ; elle est mise à jour à chaque lecture d'un caractère de fin de ligne.

La partie règles

C'est la partie oùu on fait correspondre à chaque motif une action qui consiste à renvoyer l'unité lexicale correspondante et placer la valeur associée dans une variable globale, si le lexème lu est reconnu.

— Pour les motifs simples, on retourne simplement l'unité lexicale correspondante, par exemple :

```
;          return PV;
=          return EGAL;
\|         return OR;
\*         return ITER;
...
```

— Pour les motifs qui peuvent prendre plusieurs valeurs, on place la valeur d'attribut correspondante dans `yylval` et on retourne l'unité lexicale correspondante. Notons que `yytext` contient toujours le dernier lexème reconnu. Il s'agit d'un tableau de caractère de longueur `yyleng`.

```
{nombre}                  { yylval.val=atoi(yytext+1);
                            return NUM;
                          }
{car_simple}              { yylval.car=yytext[0];
                            return CAR_SIMPLE;
                          }
{car_predef}              { yylval.car=yytext[1];
                            return CAR_PREDEF;
                          }
{car_special}             {switch(yytext[1]){
                           case 'n': yylval.car='\n'; break;
                           case 't': yylval.car='\t'; break;
```

```
        case 'r': yylval.car='\r'; break;
        case 'v': yylval.car='\v'; break;
        case 'f': yylval.car='\f'; break;
        case 'b': yylval.car='\b'; break;
        case 'a': yylval.car='\a';
        }
        return CAR_SPECIAL;
      }
...
```

Les fonctions auxiliaires

Dans cette partie, on définit les fonctions suivantes :

— `yywrap()` qui indique qu'il n'y a aucun fichier à traiter lorsque elle revoie une valeur différente de 0. Elle est appelée par l'analyseur lexical quand il atteint la fin du fichier en entrée.

```
int yywrap(){
  return 1;
}
```

— `erreur_lexicale()` qui est la fonction appelée dans le cas où le caractère en entrée n'est pas reconnu par l'analyseur.

L'analyseur produit par le générateur *flex* réalisera les fonctions suivantes :

— lire le texte d'entrée, caractère par caractère, de la gauche vers la droite et isoler les mots (lexèmes) et leur classes (unités lexicales).
— éliminer les blancs (espaces, tabulations, fin de lignes) et les commentaires,
— détecter d'éventuelles erreurs lexicales et les signaler.

3.3.1.2 Implémentation des analyseurs syntaxique et sémantique

Cela revient à écrire une spécification yacc dans laquelle on décrit un schéma de traduction du langage de spécifications régulières défini au chapitre conception. Selon la syntaxe décrite précédemment, la spécification yacc comprendra les trois parties suivantes.

La partie déclarations

La première partie du fichier `parser.y` contient :

— le pré-code C placé entre %{ et %}
On inclut le fichier qui contient les prototypes des fonctions dont on aura besoin durant la partie suivante et on déclare une autre fonction et variable.

```
%{
#include "tab_notions.h"
void erreur_semantique(const char *);
struct SER * erg;
%}
```

— La déclaration de toutes les unités lexicales grâce à la directive `%token`.

```
%token PV EGAL OR ITER ITERP OPT ACG ACD NUM VIR PARG PARD CRG
%token CRD NON CAR_SIMPLE CAR_PREDEF CAR_SPECIAL VIDE DEF INTER
```

— La déclaration du type de tous les attributs de tous les symboles avec la directive `%union`.

```
%union { char * notion;
         char car;
         struct {char car1;
                 char car2;
         } plage;
         unsigned  int val;
         struct SER * exp;
}
```

— L'assignation de types aux attributs des différents symboles.

```
%type <exp> expr intervalle
%type <plage> INTER
%type <car> CAR_SIMPLE CAR_PREDEF CAR_SPECIAL
%type <notion> NOTION
%type <val> NUM
```

— L'ajout d'informations au sujet de la priorité et l'associativité des opérateurs. Ces informations servent à résoudre les conflits d'analyse.

```
%left OR
%nonassoc PARG CRG CAR_SIMPLE CAR_PREDEF CAR_SPECIAL VIDE
%nonassoc NOTION INTER
%left CONCAT
%nonassoc ACG
%nonassoc ITER ITERP OPT
```

Les directives `%left`, `%right` et `%nonassoc` déclarent, respectivement, les symboles qui les suivent, associatifs à gauche, associatifs à droite et non associatifs.
La priorité est en fonction de l'ordre de déclaration, les moins prioritaires en premier. Une précédence peut être localement modifiée par `%prec` lors de l'écriture des règles.

— La définition de l'axiome de la grammaire.

```
%start desc
```

La partie règles

Nous retrouverons ici les règles de traduction de la grammaire, chaque règle étant composée d'une production suivie le plus souvent par l'action sémantique qui correspond.

L'action sémantique consiste en du code qui permet soit de construire un arbre syntaxique pour l'expression reconnue, la racine de cet arbre représente le type de l'opération et les nœuds sont des expressions ou des terminaux ; soit de retourner des messages d'erreur si les expressions n'ont

pas un sens par rapport à la sémantique du langage. La transcription du schéma de traduction en règles yacc sera présentée par fragments, selon la notion générée.

A. Traitement des descriptions

```
desc:  decl expr          { erg=$2; }
```

L'action sémantique permet de récupérer la représentation abstraite de l'expression (`expr`) dans une variable de nom `erg` (expression régulière globale) déclarée dans la première partie de la spécification.

B. Traitement des déclarations

Les deux règles qui traitent les déclarations sont :

```
decl:  decl NOTION EGAL expr PV
          {
            if (er=chercher_tn($2))
               erreur_semantique2("Redéfinition de la notion",$2);
            else
               inserer_tn($2,$4);
          }
    |     { }
```

La fonction `chercher_tn()` permet de vérifier si la notion donnée en argument existe déjà dans la *table des notions* (communément appelée table des symboles). Si c'est le cas, un message d'erreur sera renvoyé indiquant qu'il y a redéfinition de la notion c-à-d. qu'elle existe déjà ; sinon la fonction `inserer_tn()` se chargera d'insérer cette notion avec l'expression régulière quelle désigne, dans la table des notions.

C. Traitement des expressions

L'action consiste à construire pour à chaque type d'expression régulière, l'arbre syntaxique correspondant, et cela en faisant appel à des fonctions qui permettent de créer la structure d'arbre syntaxique pour chaque type d'expression.

L'arbre syntaxique d'une expression est composé du nœud racine et des nœuds intermédiaires qui représentent les opérateurs, ainsi que des nœuds terminaux (des feuilles) occupés soit par des caractères ou des le token `VIDE` désignant le mot vide (transcrit par `£`).

Un arbre syntaxique est représenté en mémoire sous forme d'un enregistrement de type `struct SER` qui contient deux champs :

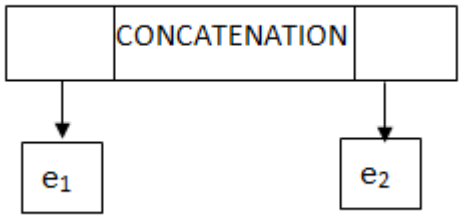
- un champ `type` qui indique le type de l'opération, et
- un pointeur `exp` sur l'une des données suivantes :
  - un caractère,
  - un pointeur vers une structure pouvant représenter
    - une concaténation (struct CONCAT),

— une alternative (struct ALT), ou
— une itération (struct SER).

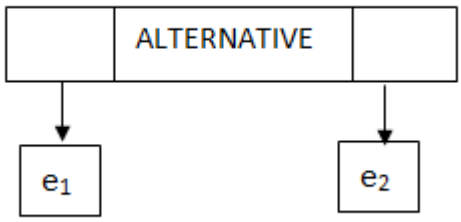
```
enum TYPE {EPSILON,CARACTERE,CONCATENATION,ALTERNATIVE,
                                                    ITERATION};

union UER {
        char ecar;
        struct CONCAT * econc;
        struct ALT * ealt;
        struct SER * eiter;
};

struct SER {
    enum TYPE type;
    union UER * exp;
};
```

Chaque fonction appelée dans les actions permet de construire la structure arborescente souhaitée.

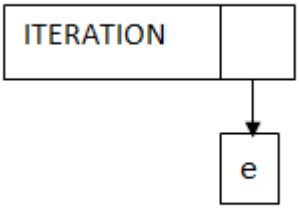
— La fonction *struct SER * make_ concat(struct SER * e1, struct SER * e2)*
Cette fonction prend en paramètre deux expressions et retourne la structure de l'arbre syntaxique pour la concaténation, illustrée par le schéma suivant :



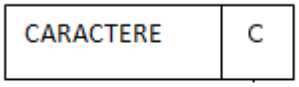
— La fonction *struct SER * make_ alt (struct SER * e1, struct SER * e2)*
Cette fonction prend en paramètres deux expressions et retourne la structure de l'arbre syntaxique pour l'alternative, illustrée par le schéma suivant :



— La fonction *struct SER * make_ iter (struct SER * e)*
Cette fonction prend en paramètre une expression et retourne la structure de l'arbre syntaxique pour l'itération, illustrée par le schéma suivant :



— La fonction *struct SER * make_ car(char c)*
Cette fonction prend en paramètre un caractère et retourne la structure de l'arbre syntaxique pour le caractère, illustrée par le schéma suivant :



— La fonction *make_ epsilon()*
Cette fonction sans paramètre retourne la structure de l'arbre syntaxique pour le mot vide, illustrée par le schéma suivant :

Grâce à ces fonctions, on peut construire l'arbre syntaxique correspondant aux expressions qui appartiennent au langage.

Le listing suivant montre l'implémentation en yacc de la traduction de quelques expressions régulières.

```
expr: expr expr     %prec CONCAT { $$=make_concat($1,$2); }
    | expr OR expr  { $$=make_alt($1,$3); }
    | expr ITER     { $$=make_iter($1); }
    | expr ITERP    { $$=make_concat($1,make_iter($1)); }
    | expr OPT      { $$=make_alt($1,epsilon); }
   ...
    | CAR_SPECIAL   { $$=make_car($1); }
    | VIDE          { $$=epsilon; }
    | NOTION        { if (er=chercher_tn($1))
                        $$=er;
                         else
                        erreur_semantique2($1,"notion non définie");
                    }
;;
```

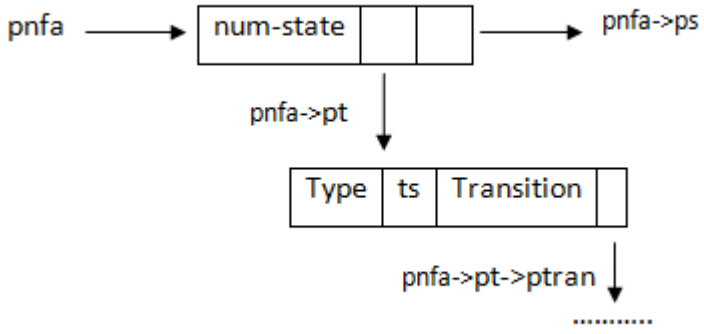


FIGURE 3.3 – Structure pour représenter un état de l'AFN.

La partie routines annexes

Cette partie comporte :

— L'inclusion du fichier `lexer.c` :

```
#include "lexer.c"
```

Cette directive permet à l'analyseur syntaxique de faire appel à la fonction `yylex()` qui se trouve dans le fichier `lexer.c`.

— La définition de la routine de gestion d'erreurs syntaxiques `int yyerror(char * msg)` L'analyseur signale l'erreur en appelant la fonction de rapport d'erreur yyerror fournie chaque fois qu'une erreur de syntaxe est rencontrée.

— La définition des routines de gestion d'erreurs sémantiques qui ont été appelées dans la partie précédente. Il s'agit de `void erreur_semantique(const char * msg)` et `void erreur_semantique2(const char * msg1, const char * msg2)`.
Ces fonctions permettent d'afficher le message d'erreur passé en argument et d'indiquer la ligne où se trouve l'erreur grâce à variable `yylineo` qui maintient le numéro de la ligne en cours.

3.3.2 Implémentation du générateur d'automates finis indéterministes

Maintenant qu'on peut générer la structure de l'arbre syntaxique d'une quelconque expression régulière, on peut procéder à l'étape suivante qui est la génération d'automates finis non déterministes, en implémentant l'algorithme reposant sur la méthode de Thompson présenté durant la conception.

3.3.2.1 Représentation d'un AFN

Chaque état d'un automate se présentera en mémoire sous forme d'un enregistrement qui contient quatre champs. Cette structure sera référencée avec un pointeur pour permettre l'accès et la manipulation de ses champs.

La figure 3.3 donne une représentation de la structure d'un état de l'AFN.

Sur la figure 3.3, `pnfa` est un pointeur vers l'état initial de l'AFN. Chaque structure représentant un état contient les champs :

— `num_state` indiquant le numéro de l'état,
— `ps`, un pointeur vers l'état suivant,
— `pt`, une liste des transitions dont l'état représenté par la structure est l'état de départ. Chaque élément de cette liste est une structure contenant les champs :
  — `type` qui décrit le type de la transition (epsilon/avec caractère),
  — `ts` indiquant l'état cible,
  — `transition` qui indique le littérale de transition (si elle n'est pas spontanée),
  — `ptran`, un pointeur vers la transition suivante.

3.3.2.2 Implémentation de l'algorithme de génération

Pour pouvoir manipuler des structures représentant des AFN, nous avons été amenés à définir plusieurs fonctions parmi lesquels la plus importante est :

*struct state *nfa(struct SER * ex, struct state * si, struct state * sf) ;*

C'est la fonction principale à laquelle on fera appel dans la fonction principale *main()* de notre application. Elle a trois arguments :

— `struct SER* ex` est un pointeur sur une expression régulière pour laquelle on désire construire un AFN ;
— `struct state* si` et `struct state* sf` sont des pointeurs sur deux états de l'AFN, entre lesquels nous devons construire la portion de l'AFN pour l'expression `ex`.

La définition complète de cette fonction (essentielle) est présentée dans le listing suivant.

```
struct state *nfa(struct SER* ex,struct state* si,struct state* sf)
{
   struct state *sk,*sl,*sj;

   switch(ex->type){
       case EPSILON:         tran_eps(si,sf);
                             break;
       case CARACTERE:       tran_car(ex->exp->ecar,si,sf);
                             break;
       case  CONCATENATION: sk=create_state();
                             nfa (ex->exp->econc->left,si,sk);
                             nfa (ex->exp->econc->right,sk,sf);
                             break;
       case ALTERNATIVE:     nfa (ex->exp->ealt->left,si,sf);
                             nfa (ex->exp->ealt->right,si,sf);
                             break;
       case ITERATION:       sl=create_state();
                             sj=create_state();
                             tran_eps(si,sl);
                             tran_eps(sf,si);
```

```
                                tran_eps(sj,sf);
                                tran_eps(sl,sj);
                                nfa(ex->exp->eiter,sl,sj);
    }
}
```

Cette fonction teste `ex->type` (type de l'expression régulière) et pour chaque valeur, elle réalise l'action appropriée et ce, en faisant appel d'autres fonctions :

- *struct state* create_state()* pour créer un nouvel état ;
- *void tran_eps(struct state *s1, struct state *s2)* pour ajouter une epsilon-transition de l'état *s1* vers *s2 ;*
- *void tran_car(char c, struct state *s1, struct state *s2)* pour ajouter une transition sur le caractère c, de l'état *s1* vers *s2.*

Après que la compilation du programme implémenté ait réussit, plusieurs expressions ont été testées. Les AFN obtenus après exécution du programme correspondent à ceux trouvés manuellement et nous pouvons donc passer à l'étape suivante qui consiste à implémenter le générateur d'AFD.

3.3.3 Génération d'un automate fini déterministe

Pour réaliser le générateur d'AFD, nous allons implémenter l'algorithme présenté dans la partie conception.

3.3.3.1 Représentation d'un AFD

L'AFD sera représenté en mémoire sous forme d'une liste dont chaque élément est un enregistrement qui représente un état de l'AFD et qui possède quatre champs. La figure 3.4 donne une représentation de la structure d'un état de l'AFD.

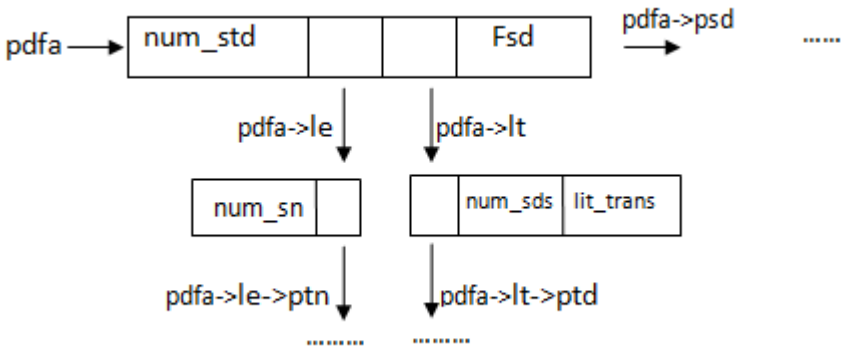


FIGURE 3.4 – Structure pour représenter un état de l'AFD.

Sur la figure 3.4, `pdfa` est un pointeur vers le premier état de l'AFD. Chaque structure représentant un état contient les champs :

— `num_std` indiquant le numéro de l'état,
— `psd`, un pointeur vers l'état suivant,
— `fsd`, un entier qui indique si l'état est final,
— `le`, un pointeur vers la liste des états de l'AFN que contient cette état de l'AFD. Chaque élément de cette liste est une structure contenant les champs :
  — `num_sn`, le numéro de l'état de l'AFN,
  — `ptn`, un pointeur vers l'état suivant,
— `lt`, une liste des transitions dont l'état représenté par la structure est l'état de départ. Chaque élément de cette liste est une structure contenant les champs :
  — `num_sds` indiquant le numéro de l'état cible,
  — `lit_trans` qui indique le littérale de transition.

3.3.3.2 Implémentation de l'algorithme de génération

Pour pouvoir manipuler des structures représentant des AFD, nous avons été amenés à développer plusieurs fonctions dont la principale est :

struct dfa_ state dfa(struct state *s) ;*

Cette fonction prend comme argument un AFN et retourne un pointeur vers la structure de l'AFD équivalent. Elle réalise son traitement en faisant appel à d'autres fonctions :
— *struct list *list_ literals(struct state*s, struct list *l)* qui permet de retourner la liste de tout les littéraux des transitions de l'AFN,
— *struct list_ state* l_ trans(char litt, struct dfa_ state *ppdfa, struct state *nfa)* qui retourne la liste des états de l'AFN vers lesquels transitent une liste d'états de ce même AFN sur un littérale `litt,`
— *struct list_ etat*eps_ tran(struct list_ etat *eds, unsigned ns, struct list_ etat *eds_ suivant, struct state *s)* permettant de calculer la liste des epsilon-successeurs d'une liste d'états,
— *bool compar_ list(struct list_ etat *l, unsigned * variable)* qui fait la comparaison entre deux listes d'états et retourne un booléen,
— *struct dfa_ state* fs_ dfa(struct dfa_ state *)* qui indique si un état est final ou non en attribuant soit un 1 si c'est le cas ou un 0 dans le cas contraire, au champ `fsd` de l'état de l'AFD.

Comme les résultats obtenus lors de l'exécution du programme avec différentes expressions régulières correspondent à ceux trouvés manuellement avec ces mêmes expressions régulières, nous pouvons affirmer que le programme implémenté génère effectivement les AFDs correspondant aux expressions lues en entrée.

3.3.4 Représentation graphique

L'implémentation de la fonction de génération d'une représentation d'un AFD sous forme d'une image graphique est assurée la fonction :

*void gen_ image_ dfa(struct dfam_ state *dfam, char *nom_ fichierI_ dfa) ;*

dont la définition complète est donnée dans le listing suivant.

```
void gen_image_dfa(struct dfam_state *dfam, char *nom_fichierI_dfa){
   struct dfam_state *etat_dfam = dfam;

   fichierI_dfam = fopen(nom_fichierI_nfam, "w");
   if (!fichierI_dfam )
      gen_erreur_dfam("Impossible de creer le fichier",
                                                  nom_fichierI_nfam);
   fprintf(fichierI_dfam, "digraph AFD {\n");
   while (etat_dfam) {
      if (etat_dfam->type_etat!=1)
         fprintf(fichierI_dfam, "s%d [shape=circle];\n",
                                                 etat_dfam->num_stdm);
      else
         fprintf(fichierI_dfam,
        "\ts%d [shape=circle,peripheries=2];\n",etat_dfam->num_stdm);
      gen_image_trans_nfa(etat_dfam, fichierI_dfam);
      etat_dfam = etat_dfam->psdm;
    }
    fprintf(fichierI_dfam,"start [shape=none,label=\"start\",
                                                    style=invis]");
    fprintf(fichierI_dfam,"start -> s0 [arrowhead=vee,arrowsize=1.5,
                                                  style=\"bold\"]");
    fprintf(fichierI_dfam, "}\n");
    fclose(fichierI_dfam);
}
```

gen_ image_ dfa() est est appelée dans dans la fonction principale *main()* de l'applidation. Elle parcourt les états de l'AFD, et pour chacun d'eux, elle écrit la description en langage dot dans le fichier de sortie.

Comme le langage dot ne possède pas de formalisme qui permet de représenter la flèche indiquant l'état initial, nous avons du ajouter un état invisible *start* et un arc sans étiquette à partir de cet état vers l'état initial de l'AFD.

Pour représenter les transitions, *gen_ image_ dfa()* fait appel à la fonction *gen_ image_ trans_ dfa()* dont la définition est donnée par listing suivant.

```
void gen_image_trans_dfa(struct dfam_state *etat_dfam,
                                        FILE * fichier_image_dfam){
   struct list_trandm *tran_dfam  = etat_dfam->ltdm;
   while (tran_dfam) {
      fprintf(fichier_image_dfam, "s%d -> s%d [ arrowhead=vee,
                    label=", etat_dfam->num_stdm, tran_dfam->etd);
      fprintf(fichier_image_dfam, "\" %c \"", tran_dfam->lit_tran);
      fprintf(fichier_image_dfam, " ]\n", tran_dfam->etd);
      tran_dfam = tran_dfam->tsuiv;
   }
}
```

Cette fonction prend en paramètres un pointeur vers la liste des transitions et le fichier de sortie dans lequel elle retourne la description des transitions en langage *dot*.

Cette étape marque la fin de la réalisation de notre application. Cependant, une étape de test est importante pour s'assurer de la conformité des résultats obtenus après la réalisation avec ceux escomptés.

3.4 Tests et résultats

Exemple1

Considérons l'expression régulière $(ab)^*$. Après avoir soumis cette expression au générateur, nous avons obtenu l'AFD dont la représentation graphique est donnée à la figure 3.5.

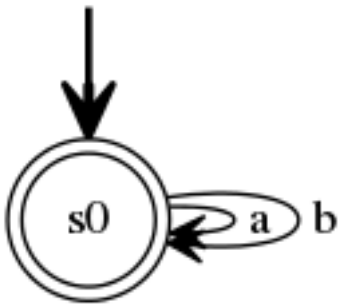


FIGURE 3.5 – Premier exemple d'AFD construit par l'outil.

Exemple 2

Considérons dans cet exemple le langage dont l'alphabet est l'ensemble des nombres entiers non signé. Une description pour ce langage serait :

```
{chiffre}=[01]
{chiffre}+
```

Après avoir soumis cette spécification au générateur, nous avons obtenu l'AFD dont la représentation graphique est montrée à la figure 3.6.

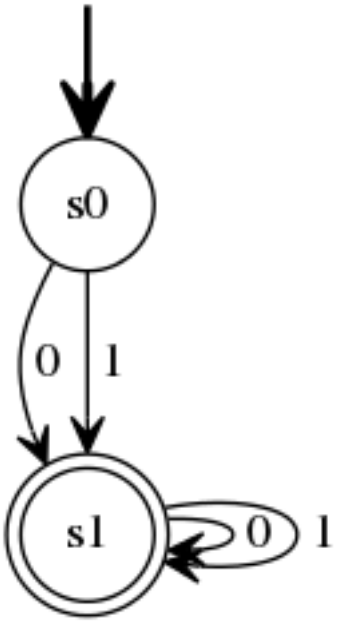


FIGURE 3.6 – Deuxième exemple d'AFD construit par l'outil.

Nous avons effectué plusieurs autres tests et comparé les résultats obtenus grâce à l'application, avec ceux obtenus en appliquant à la main ou avec un outil similaire qu'on peut trouver sur le site `www.hakingoff.fr`. On retrouve effectivement une similitude entre ces résultats, ce qui nous amène à affirmer que notre outil fonctionne correctement.

3.5 Conclusion

Cette partie du projet a été particulièrement délicate et a pris la majeure partie du temps consacré à l'accomplissement de ce projet. Cela reflète bien son importance dans la mesure où nous avons mis en pratique tout ce qui a été cité dans la théorie et en tenant compte des imprévus et des difficultés rencontrées durant cette réalisation.

Nous avons obtenu à la fin des résultats concluants qui montrent qu'on a pu atteindre l'objectif fixé au départ à savoir la réalisation d'un générateurs d'AFD à partir d'une expression régulière.

Conclusion générale

La réalisation de ce projet nous a été très enrichissante dans la mesure où elle nous à permis d'approfondir nos connaissances en théorie des langages formels. Elle nous à permis aussi de mesurer la puissance des expressions régulières et des grammaires formelles dans la description et des automates finis dans la reconnaissance de ces langages qui sont à la base de tous les langages informatiques. Ce projet nous a aidé à adopter une vision conceptuelle dans l'organisation des données lors de l'implémentation des différents algorithmes, en nous confrontant à l'automatisation du traitement, de l'analyse et de la reconnaissance.

Ce projet nous a permit également de maîtriser d'avantage le système GNU/Linux ainsi qu'une multitude d'outils tels que ceux de la chaîne de compilation *gcc*, les outils *flex*, *yacc*, *make* et *graphviz*, très utilisés par les chercheurs et développeurs à travers le monde.

Notre travail ouvre plusieurs perspectives notamment l'implémentation de la procédure de minimisation qui permettra d'optimiser les ressources en espace mémoire et en temps d'exécution. Le générateur qui en résulterait fera correspondre à chaque DFA un DFA minimal ayant moins d'états et transitions et reconnaissant le même langage.

Bibliographie

[1] *Automates et Langages : Cours et TD*
S. Julia, 2015-2016,
http://deptinfo.unice.fr/∼julia/AL

[2] *L'outil byacc, code source et documentation*
http://byacc.sourceforge.net

[3] *Cours sur les langages rationnels*,
P. Amsili, 2008,
http://utilisateurs.linguist.univ-paris-diderot.fr/~amsili/Ens08\
/pdf/poly-li324-1.pdf

[4] *Compilers : Principles, Technics and tools*,
Aho, Sethi and Ullman, InterEditions, 1987.

[5] *Compilation : cours et exercices*,
H. Drias, OPU, 1992.

[6] *Notes de cours sur les automates*,
F. Barthélemy, 2015,
http://deptinfo.cnam.fr/∼barthe/NFP108/poly-automates.pdf

[7] *Théorie des langages, Notes de Cours*
F. Yvon et A. Demaille, 2016,
www.lrde.epita.fr/people/akim/thl/lecture-notes/theorie-des-langages-1.pdf

[8] *Propriétés des langages réguliers*,
D. Nouvel, 2015,
www.damien.nouvels.net/cours/automates/4_ProprietesDesLanguagesReguliers.pdf

[9] *Analyse lexicale*,
L. maranget, 2012,
http://pauillac.inria.fr/∼maranget/X/compil/poly/lexical.html

[10] *Cours de compilation*,
Rahmani, 2015,
http://www.fsr.ac.ma/cours/informatique/rahmani

[11] *L'outil flex, code source et documentation*,
http://flex.sourceforge.net

[12] *graphviz : logiciel et documentation*
`http://graphviz.org`