Théorie des langages

& outils pour la compilation

ESGI – 3 AL 2016 / 2017

Frédéric Baudoin

fbaudoin@myges.fr

Introduction

La compilation est une transformation qui consiste à rendre un programme exécutable. Fondamentalement, il s'agit d'une traduction : le texte initial (le programme) est transformé en un texte compréhensible par la machine (dans un formalisme de plus bas niveau).

La sortie d'un compilateur est de nature très variable : un programme exécutable par un processeur physique (Pentium, G7), un fichier de code pour une machine virtuelle (JVM), un code abstrait destiné à être recompilé, …

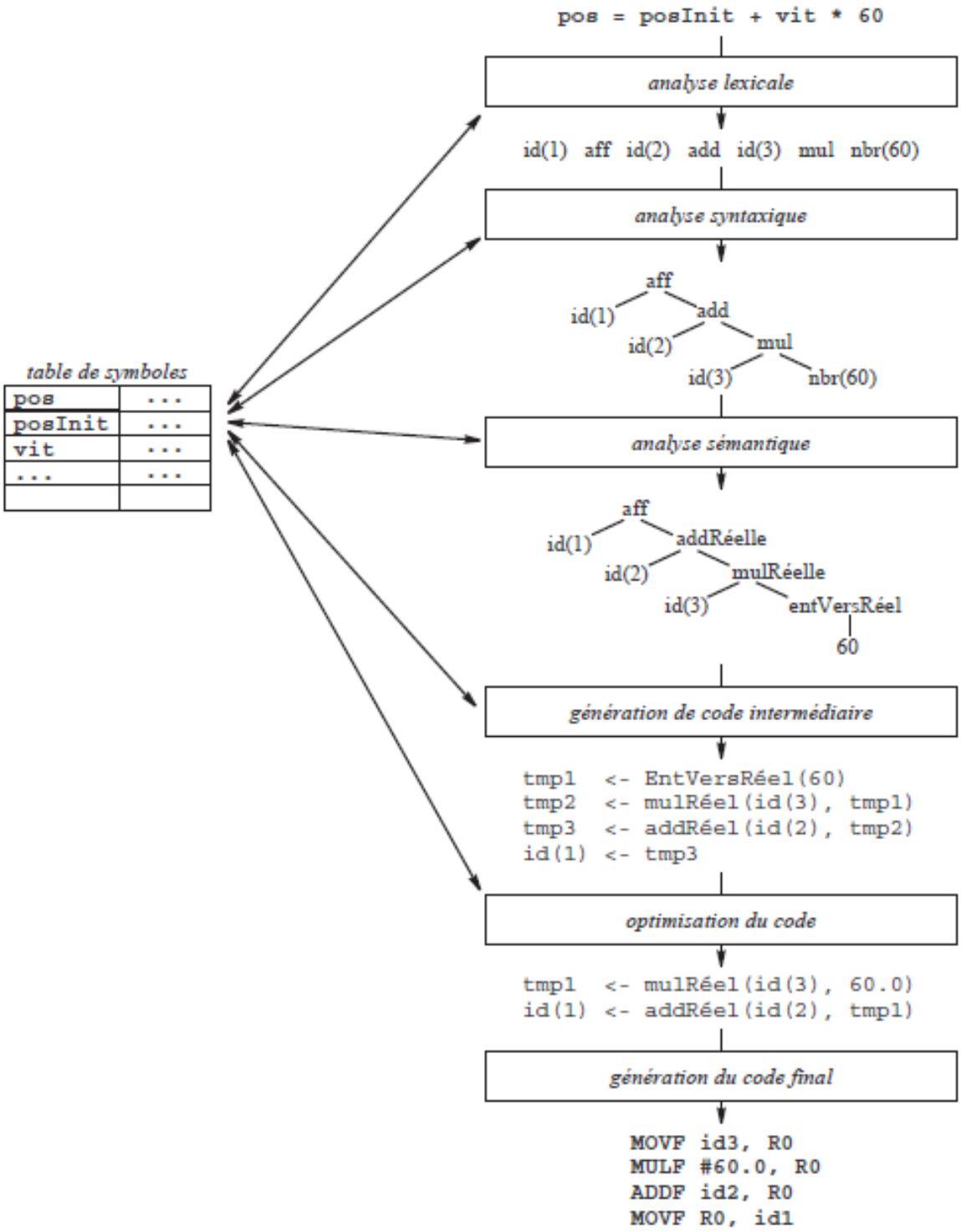
Le travail du compilateur se fait usuellement en plusieurs phases :

1. **Analyse lexicale**
   Les caractères isolés du texte initial sont regroupés en unités lexicales appartenant au langage. La tâche de l'analyse lexicale est déléguée à celle de l'analyse lexicale en fournissant un mot lors de chaque appel.
2. **Analyse syntaxique**
   La bonne formation structurelle de la suite d'unités lexicale est vérifiée relativement à la grammaire du langage.
3. **Analyse sémantique**
   Les propriétés sémantiques à vérifier : la bonne déclaration des identificateurs, la vérification des types dans les opérandes relativement aux opérateurs, le besoin de conversions, le type et le nombre des arguments dans les fonctions, …
4. **Génération de code intermédiaire**
   un peu comme avec les machines abstraites, pour mutualiser les dernières étapes de la compilation avec des compilateurs d'autres langages source.
5. **Optimisation de code**
   Consiste à transformer le code pour une exécution plus rapide.
6. **Génération du code final**
   Nécessite la connaissance de la machine cible (réelle, virtuelle, ou abstraite), et notamment de ses possibilités en matières de registre, de pile, …

Dans ce cours, notre étude se focalisera sur les 2 premières phases : l'analyse lexicale et l'analyse syntaxique. Au cours du projet de fin d'année, vous vous confronterez (de façon élémentaire) aux autres phases : il s'agira de définir votre propre langage de programmation, et évidemment, d'en concevoir un interpréteur permettant de l'opérationnaliser.

1. **Analyse lexicale.** Après avoir présenté les notions de bases sur les langages formels, on s'intéressera aux automates finis, une représentation « opérationnelle » des expressions régulières qui facilite significativement la modélisation du test d'appartenance d'une chaine, ou l'énumération des chaines couvertes par une expression régulière. Ensuite, nous étudierons les différentes tâches mises en jeu dans un analyseur lexical, ainsi que l'utilitaire *lex*, un outil permettant de générer des analyseurs lexicaux.

2. **Analyse syntaxique**. Après avoir présenté les concepts de base des grammaires formelles (arbre de dérivation, automates à piles), on s'intéressera à leur capacité à modéliser de manière avantageuse les langages et à produire une analyse des chaines d'entrée (les programmes dans le cas des langages informatiques). Ensuite, on verra les principes de l'analyse descendante LR(k) et le fonctionnement de *yacc*, un générateur d'analyseur syntaxique.



I. Alphabets et langages

a. Généralités

Alphabet

∑ : alphabet des symboles (éléments de base du langage) avec $|\sum| < \infty$

Chaînes

Une chaîne est une suite finie de symboles.

Notation :

|P| = 7 : la chaîne P contient 7 éléments

$\sum^n = l'ensembledes\ chaines\ de\ taille\ n$

$\sum^0 = \{\varepsilon\}\ (chaîne\ vide)$

$$\sum^* = \bigcup_{n \geq 0} \sum^n$$

$$\sum^+ = \bigcup_{n > 0} \sum^n$$

Langage

Un langage L sur ∑ est une partie de $\sum^*$ = certaines séquences de symboles de ∑

En pratique, les langages comprennent souvent un nombre **infini** de chaines

Exemples de langages

Exemple 0 : langage abstrait

$\sum = \{a, b, c\}$

$\Sigma^* = l'ensemble\ des\ séquences\ de\ symboles\ \in \{a, b, c\}$

$L = \{mots\ de\ \sum^*\ commençant\ par\ a\} \subset \Sigma^*$

Exemple 1 : Calculatrice

∑ = {0,1,2,…,9,+,-,x,/}

L : ensemble des expressions arithmétiques syntaxiquement correctes

Exemple 2 : cas du langage naturel

∑ : dictionnaire + verbes conjugués + féminins + pluriels + espaces + ponctuation

L : ensemble des phrases (ou des textes) syntaxiquement bien formées

Exemple 3 : cas des langages de programmation – version 1

∑ : les identifiants, les chiffres, les opérateurs arithmétiques, les mots clefs réservés du langage

L : ensembles des programmes compilables

Exemple 4: cas des langages de programmation – version 2

∑ : les caractères

L : ensembles des programmes compilables

Concaténation entre chaines

Soit u et v deux chaînes, u.v = chaîne concaténée.

Opérations sur les languages

a. Opérateurs des ensembles usuels : union, intersection, complémentaire $A \cup B$, $A \cap B, \bar{A}$

   Leur utilisation est licite et bien définie car les langages désignent des ensembles

b. Concaténation entre languages

   $L1.L2 = \{m.v, m \in L1, v \in L2\}$
   $L^2 = L.L$

   $L^0 = \{\varepsilon\}$

   $L^{n-1}.L = L^n$

c. Etoile de Kleene :

   $$L^* = \bigcup_{n \geq 0} L^n$$

   $$L^* = L^*.L \cup L$$

Exemple : Si L et C désigne 2 langages comportant des chaine de longueur 1 (des caractères)
$C = \{0, 1, \dots, 9\}$ *et* $L = \{\mathrm{A}, \dots, \mathrm{Z}, \mathrm{a}, \dots, \mathrm{z}\}$

- $L \cup C$ désigne l'ensemble des chiffres et des lettres
- LC désigne l'ensemble des chaines formées d'une lettre suivie d'un chiffre
- L^4 désigne l'ensemble des chaines composées de 4 lettres
- L^* désigne l'ensemble des chaines faites d'un nombre quelconque de lettres. ε en fait partie.
- $L(L \cup C)^*$ désigne l'ensemble des chaines formées de lettre et de chiffres commençant par une lettre.
- Et $L(\ L \cup C^*\ )$?

Exercice : Décrire les langages suivant sur l'alphabet $\sum = \{0, 1, \dots, 9\}$ à l'aide des opérateurs décrits ci-avant

$L1 = \{n° \ de \ portable\} =$

$L2 = \{nombre \ entiers \ sans \ \ 0 \ inutiles\} =$

$L3 = \{nombre \ décimaux \ sans \ \ 0 \ inutiles\} \ sur \ l'alphabet \ \{0, 1, \dots, 9, "."\}$

2. Automates finis et expressions régulières

a. Automates finis

Définition

$$\boldsymbol{A = (V_t, E, E_0, F, \partial)}$$

$V_t \ \rightarrow Alphabet \ terminal$[1]
$E \ \rightarrow \ Etat$
$E_0 \ \rightarrow Etats \ initiaux$
$F \ \rightarrow Etats \ finaux$
$\partial \ \rightarrow Ensemble \ des \ transitions$

Un automate fini peut être représenté par un graphe orienté et étiqueté par des éléments de V_t.

Remarque : il peut y en avoir un nombre quelconque d'état initiaux et finaux, et au minimum un de chaque.

Chemin dans un automate fini

Un chemin est une suite de flèches consécutives dans le graphe. Comme chaque flèche est étiquetée par un symbole terminal, on peut associer un mot à chaque chemin. Un chemin est dit *réussi* si et seulement s'il va d'un état initial à un état final de l'automate.

[1] $\boldsymbol{V_t}$ **renvoie à** $\sum$

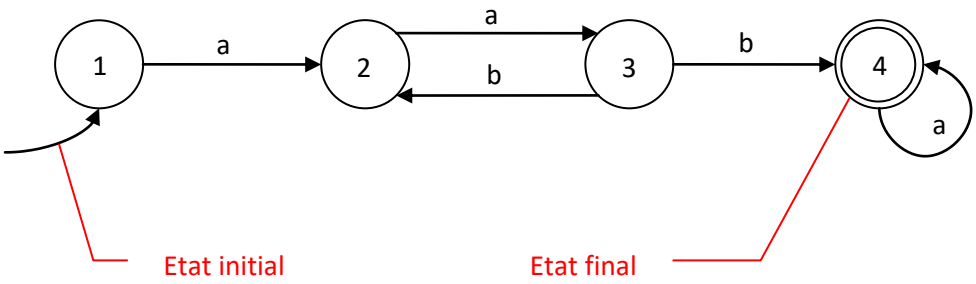
Langage reconnu par un automate fini :

On note L(A) le langage engendré par un automate A

$$\boldsymbol{L(A) = \{\omega \in V_t^*, \text{tel que } e \xrightarrow{\omega} f \text{ avec } e \in E_0 \text{ et } f \in F\}}$$

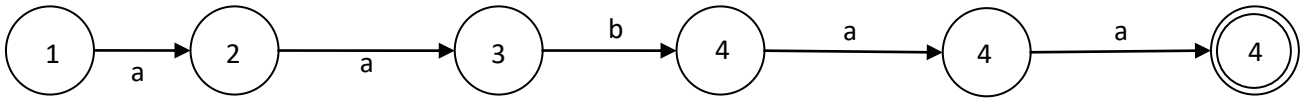
Autrement dit, un mot w est reconnu par un automate A s'il existe un **chemin réussi** correspondant.

Exemple



Ici, $A = (\{a, b\},\ \{1,2,3,4\},\ \{1\},\ \{4\},\ \{1a2,2a3,3b2,3b4,4a4\})$

Par exemple, $\omega = a^2ba^2 \in L(A)$ car il s'agit d'un chemin réussi de l'automate. On a bien $1 \in E_0, 4 \in F$. Même si c'est peu courant, on peut représenter le chemin de a²ba² comme :



Exercice :

1. Décrire 2 ou 3 autres mots de L(A)
2. Décrire $L(A)$ à l'aide des opérateurs ∪ et *
3. Que devient $L(A)$ si on rajoute l'état 2 à l'ensemble des états initiaux ?

Propriétés des automates finis

Automates déterministes

Un automate est déterministe si, pour chaque état, il n'existe pas deux transitions sortantes portant la même étiquette. Les automates déterministes ne peuvent comporter qu'un seul état initial.

Automates complets

Un automate est complet s'il pour chaque état, il existe une transition sortante pour chaque symbole de l'alphabet terminal. Dans un tel automate, les progressions (liées à un mot) ne sont jamais bloquées.

Automate émondé

Un état est utile s'il existe au moins un chemin réussi qui l'emprunte. Un automate est émondé si tous ses états sont utiles.

b. Exercices

A. Déterminer les automates qui engendrent les langages suivants :

$L_1 = \{a^n, n \geq 1\}$

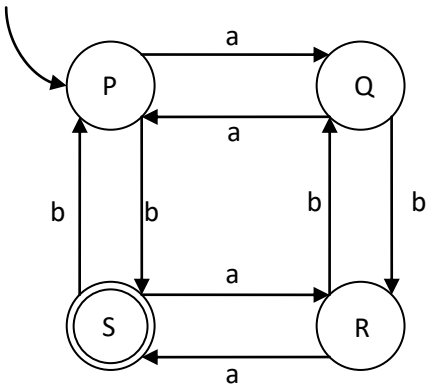
$L_2 = \{a^{2k+1}, k \geq 0\}$

$L_3 = \{a^n b^m, \quad n \geq 1, m \geq 1\}$
$L_4 = \{a^n b^m, \quad n \geq 1, m \geq 0\}$

$L_5 = \{a^n b^m, \quad n \geq 0, m \geq 1\}$
$L_6 = l'ensembledes\ séquences\ (finies)\ de\ a\ et\ de\ b$
$L_7 = les\ mots\ sur\ a\ et\ b\ où\ b\ est\ immédiatement\ suivi\ d'un\ a$

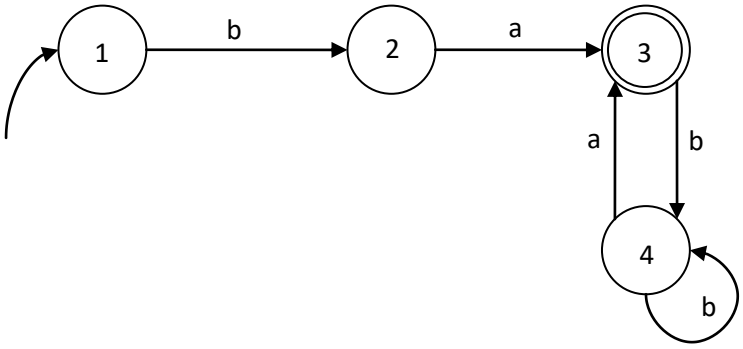
$L_8 = \{\omega \in \{a,b\}^*, |\omega|_a \% 3 = 0\}$ où $|\omega|_a$ désigne le nombre de a dans ω
$L_9 = \{(ab)^n b^m, \quad n \geq 1, m \geq 0\}$
$L_{10} = \{((ab)^n b^m)^k, \quad k > 0,\ n \geq 1, m \geq 0\}$ (avec et sans transition vide)

B. Donner le langage reconnu par l'automate suivant :



C. Automate du langage complémentaire

Exercice : Soit un automate A et son langage engendré L(A). Déterminer l'automate A' qui engendre le langage complémentaire $L^{'} = (V_t{}^*) \setminus L(A)$



Indice : Si $\omega \notin L(A)$, on a 2 possibilités :

- Soit il existe bien un chemin correspondant à w mais il aboutit à un état non-final
- Soit w ne correspond à aucun chemin (il y a des transitions impossibles)

c. Méthode de déterminisation des automates finis :

Définition : a-successeurs

Soit un automate fini non déterministe $\boldsymbol{A = (V_t, E, E_0, F, \partial)}$

On note l'ensemble des successeurs d'un état s de $\boldsymbol{E}$ par un symbole a de $\boldsymbol{V_t}$

$$\boldsymbol{Succ(s, a) = \{p \in E, (s \xrightarrow{a} p) \in \partial\}}$$

On peut de même définir les successeurs relatifs à un ensemble d'états **M** par un symbole a de $\boldsymbol{V_t}$

$$\boldsymbol{Succ(M, a) = \bigcup_{s \in M} Succ(s, a)}$$

Déterminisation d'un automate fini non déterministe sans ε-transition

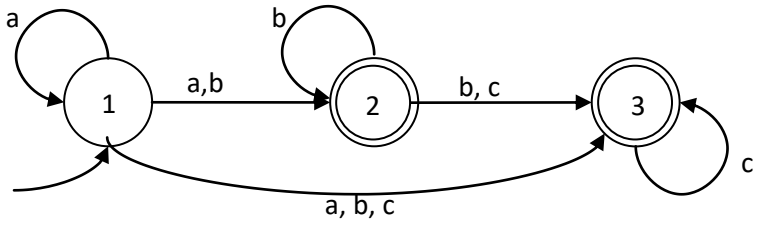
Soit un automate fini non déterministe $\boldsymbol{A} = (\boldsymbol{V_t}, \boldsymbol{E}, \boldsymbol{E_0}, \boldsymbol{F}, \boldsymbol{\partial})$ (***A*** est supposé sans ε −transitions)

On peut construire un automate fini déterministe A' qui reconnait le même langage $\boldsymbol{A'} = (\boldsymbol{V'_t}, \boldsymbol{E'}, \boldsymbol{E'_0}, \boldsymbol{F'}, \boldsymbol{\partial'})$ tel que **:**

$$\boldsymbol{A} = (\boldsymbol{V_t}, \boldsymbol{E}, \boldsymbol{E_0}, \boldsymbol{F}, \boldsymbol{\partial})$$

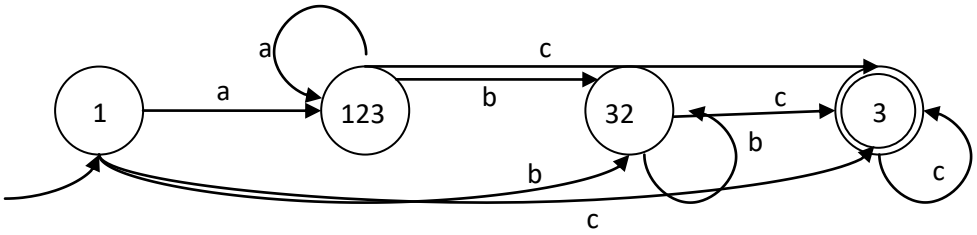
- $\boldsymbol{V'_t} = \boldsymbol{V_t}$ le même alphabet puisque on veut le même langage
- $\boldsymbol{E'} = \wp(\boldsymbol{E})$ l'ensemble des parties de E
- $\boldsymbol{\partial'}$est telle que $\boldsymbol{\partial'}(\boldsymbol{M}, \boldsymbol{a}) = \boldsymbol{Succ}(\boldsymbol{M}, \boldsymbol{a})$
- $\boldsymbol{E'_0} = \{\boldsymbol{E_0}\}$
- $\boldsymbol{F'} = \{\boldsymbol{M} \in \boldsymbol{E'}, \boldsymbol{E'} \cap \boldsymbol{F} \neq \emptyset\}$ *M est final s'il contient au moins un état final de A*

Exemple :



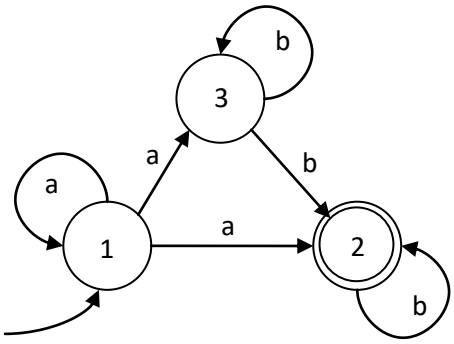
Pour la construction, on pourra utiliser un tableau des successeurs comme ci après. Attention, pour alléger l'écriture, on écrira 123 à la place de {1, 2, 3} pour désigner l'ensemble des états 1, 2 et 3

Succ	**a**	**b**	**c**
1	123	23	3
123	123	23	3
23	-	23	3
3	-	-	3

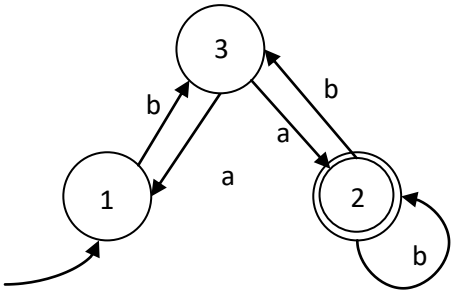


d. Exercices

1. Déterminiser les automates suivant :



2.



3. Que se passe t il *concrètement* si on applique la méthode de déterminisation à un automate déjà déterministe ?

Définition : ε –fermeture

Pour un état s, on définit son ε – fermeture comme une partie $\boldsymbol{Eps(s)}$ de $\boldsymbol{E}$ définit comme :

$$\boldsymbol{Eps(s)} = \{\boldsymbol{q} \in \boldsymbol{E}, \quad \boldsymbol{s} \xrightarrow{\varepsilon^*} \boldsymbol{q}\}$$

En d'autres termes, est l'ensemble de tous les états de *E* que l'on peut atteindre à partir de s en effectuant 0, 1, ou plusieurs ε-transition. Comme pour Succ, l'opération *Eps* peut être étendue aux ensembles :

$$\boldsymbol{Eps(M)} = \bigcup_{\boldsymbol{s \in M}} \boldsymbol{Eps(s)}$$

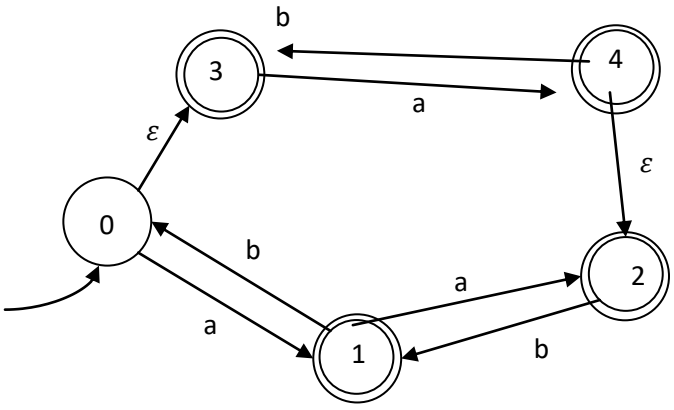
Déterminisation d'un automate fini non déterministe avec ε-transition

On peut construire un automate fini déterministe A' qui reconnait le même langage $\boldsymbol{A'} = (\boldsymbol{V'_t}, \boldsymbol{E'}, \boldsymbol{E'_0}, \boldsymbol{F'}, \boldsymbol{\partial'})$ tel que **:**

$$\boldsymbol{A} = (\boldsymbol{V_t}, \boldsymbol{E}, \boldsymbol{E_0}, \boldsymbol{F}, \boldsymbol{\partial})$$

- $\boldsymbol{V'_t} = \boldsymbol{V_t}$ le même alphabet puisque on veut le même langage
- $\boldsymbol{E'} = \wp(\boldsymbol{E})$ l'ensemble des parties de E
- $\boldsymbol{\partial'}$est telle que $\boldsymbol{\partial'}(\boldsymbol{M}, \boldsymbol{a}) = \boldsymbol{Eps}(\boldsymbol{Succ}(\boldsymbol{M}, \boldsymbol{a}))$
- $\boldsymbol{E'_0} = \boldsymbol{Eps}(\{\boldsymbol{E_0}\})$
- $\boldsymbol{F'} = \{\boldsymbol{M} \in \boldsymbol{E'}, \boldsymbol{E'} \cap \boldsymbol{F} \neq \emptyset\}$ *M est final s'il contient au moins un état final de A*

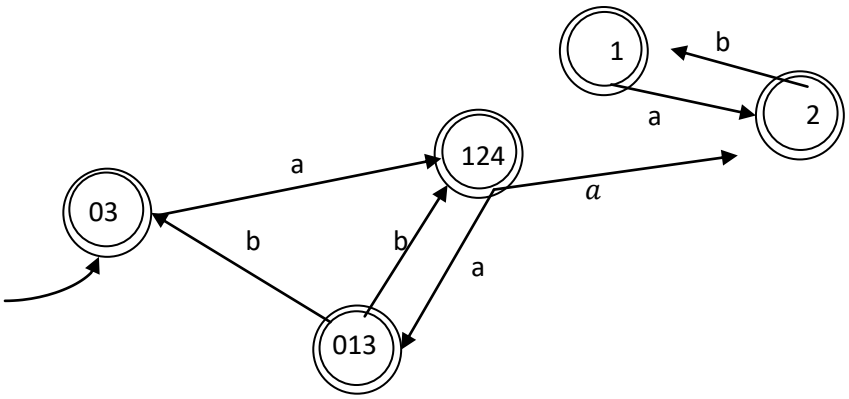
Exemple : Soit l'automate non déterministe avec ε-transition suivant :



On obtient la table suivante :

Eps(Succ	a	b
03	124	-
124	2	013
2	-	1
013	124	03
1	2	03

On obtient donc l'automate déterministe sans ε-transition suivant



e. Expressions régulières

L'objet de cette section est de montrer que les expressions régulières et les automates permettent de représenter les mêmes langages. Les automates finis constituent une représentation opérationnelle de certains langages. Il peut être utile d'en avoir une représentation déclarative (langage de requête, …) : c'est le principal intérêt des expressions régulières.

Soit V_t un alphabet et *ER* l'ensemble des expressions régulières sur V_t.

Définition syntaxique :

$\varepsilon \in ER$
$a \in V_t \implies a \in ER$
e et $e' \in ER \implies e|e' \in ER$
e et $e' \in ER \implies e.e' \in ER$
$e \in ER \implies e^* \in ER$

Remarque :

- $e|e'$ se note également $e + e'$
- $e.e'$ se note également ee'

Définition sémantique :

On note L l'application qui associe à une expression régulière e, l'ensemble L(e) des chaines d'entrée qu'elle couvre :

$$L: \begin{matrix} ER \to P(V_t{}^*) \\ e \to L(e) \end{matrix}$$

$L(\varepsilon) = \{\varepsilon\}$
$L(a) = \{a\}$
$L(e|e') = L(e) \cup L(e')$
$L(e\,.\,e') = L(e)\,.\,L(e')$

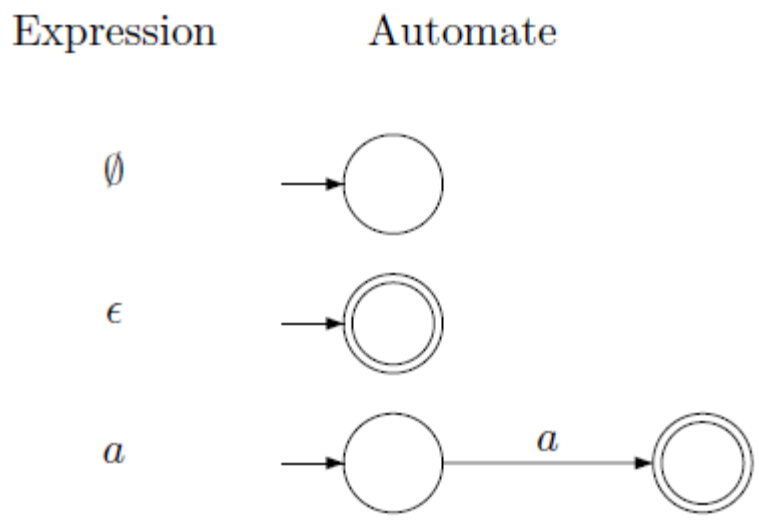
$$L(e^*) = L(e)^*$$

Priorités des opérateurs : priorité(*) > priorité(concaténation) > priorité(|)

Notations abrégés :
Soit x une expression régulière désignant un langage L,
$L(e^+) = L^*L$
$L(e?) = L \cup \{\varepsilon\}$
$e_1|e_2|\dots|e_n = [e_1 e_2 \dots e_n]$
$[e_1 e_2 \dots e_n] = [e_1 - e_n]$ $(réservé\ au\ cas\ des\ caractères)$

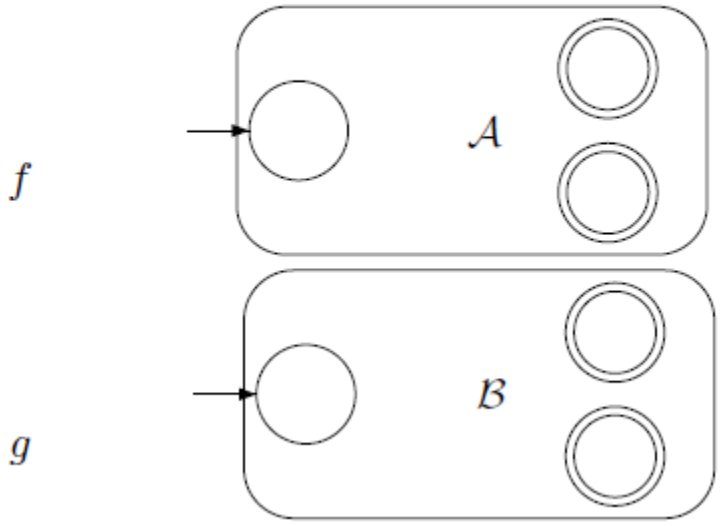
Expressions régulières → Automates
Problème : Etant donnée une expression régulière f, construire un automate A qui accepte le même langage dénoté par cette expression

Construction : cas de base



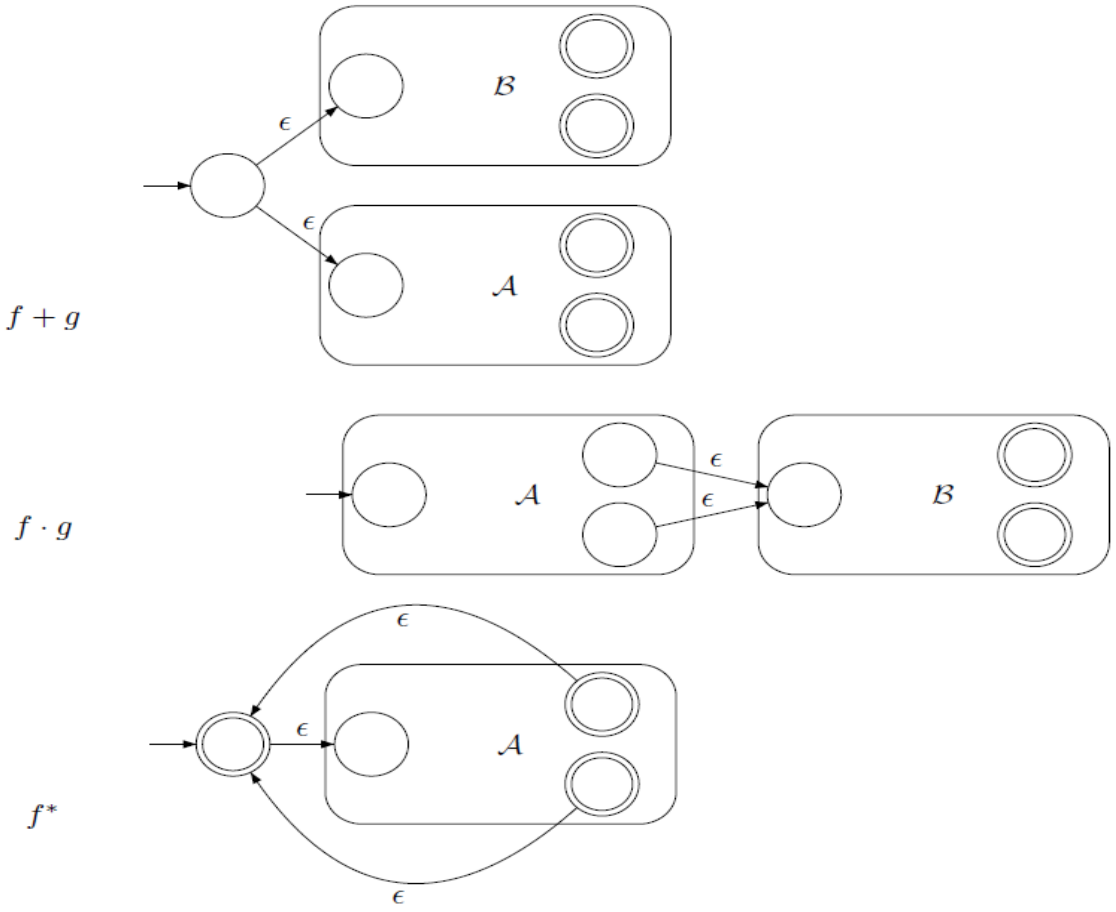
Construction : cas inductif
On suppose qu'on a déjà construit 2 automates A et B acceptant les langages des expressions f et g.

Expression Automate



On construit à partir de A et B les automates pour $f + g, f.g, f^*$

Expression Automate



Automates → Expressions régulières

Problème : étant donné un automate $\boldsymbol{A = (V_t, E, E_0, F, \partial)}$ **,** on veut construire une expression régulière f qui dénote le même langage.

La construction, non détaillée ici, comprend plusieurs étapes :

a. Passer de l'automate à un système d'équations sur les langages. On fait correspondre une équation par état, les inconnues désignent le langage accepté *à partir* de l'état correspondant et on obtient un système de n équations (n étant le nombre d'états de l'automate).
b. La résolution d'une équation se fait avec le lemme d'Arden. Si L et M désignent 2 langages, et X un langage inconnu, l'équation $X = LX + M$ a une solution unique $X_0 = L^*M$
c. On résout enfin le système par la méthode de Gauss

f. Les expressions régulières en pratique

Les expressions régulières sont de très puissants outils de manipulation de textes et de données. Elles constituent ainsi un bagage précieux de l'informaticien, quelque soit sa spécialisation (réseau, développement, système …). Par ailleurs, elles cachent des problématiques complexes, dont l'étude est importante : certaines (prestigieuses !) universités outre-Atlantique vont jusqu'à y consacrer une part significative des enseignements en 1er ou 2ème cycle : un module d'un an environ.

Finalement, ce n'est pas que dans la conception des compilateurs que les expressions régulières montrent leur force, même si elles permettent, avec des outils comme LEX par exemple, de générer très rapidement des parsers fiables et facilement modifiables, et ce, dans la plupart des langages de programmation, ce qui n'est pas rien.

Dans cette annexe du cours de théorie des langages, on verra des exemples de problèmes qu'on résout avec des expressions régulières, le champ de leur utilisation concrète, ainsi que la manière de décliner dans les principaux langages de programmation (Java, Pearl, JavaScript, Python, PHP, …) ces fameuses « regex »[2]

Références bibliographiques

- *Les expressions régulières par l'exemple*, de Vincent Fourmond (9.90 €). Un petit livre très clair, possiblement à acquérir. Les exemples de cette annexe en proviennent.
- *Maîtrise des expressions régulières*, de Jeffrey E.F.Friedl (Auteur), Laurent Dami (Traduction) (40€). Un gros livre détaillé, plutôt à emprunter, très précis sur le fonctionnement des moteurs de regex, et les méthodes d'écriture optimisée dans ce sens.

Web

Assez précis, en anglais : http://www.regular-expressions.info

Pour tester vos regex online : par exemple : http://regex101.com/

[2] En français, on parle *d'expressions rationnelles*, du fait de leur correspondance avec les *langages rationnels (ou langages réguliers)*, qui sont les langages de type 3 dans la hiérarchie de Chomsky. Mais on trouve aussi, comme ici, la dénomination *expressions régulières*, issue de la traduction directe de la forme anglaise *regular expressions*, abrégées en *regex*

Définition

Une expression régulière est en informatique une chaîne de caractères que l'on appelle parfois un **motif** et qui décrit un **ensemble de chaînes de caractères** possibles selon une syntaxe précise.

Par exemple, à l'expression régulière ''abc(d|e)'' correspondent les chaines de caractères « abcd » et « abce »

Exemples d'utilisation

Développement :

- Recherche de chaines dans les éditeurs de texte (GREP, …) Par exemple, pour trouver des fonctions/variables dans du code. Ceci est particulièrement utile dans le cas d'un développement sur une machine distante (sans IDE).
- Check de formulaires (formats de nombres, dates, email, …) et extraction d'information
- En paramètre dans certaines fonctions de manipulation de chaine dans les librairies des langages usuels
- Remplacement ou nettoyage de chaine. par exemple, pour passer une chaine saisie par l'utilisateur en paramètre d'une fonction

Réseau et système:

- Analyse de fichiers de logs (par exemple, compter le nombre de connections valides (response code 200) sur des pages montrées entre telle heure et telle heure, …)

Navigation web :

- Requêtes avancées dans les moteurs de recherche

Note : on adoptera les codes de présentation du livre de Fourmond :

- on notera les expressions régulières entre barre oblique : /expression/
- les chaines de caractères entre ''guillemets américains '', qu'on ne tape pratiquement jamais en tant que programmeur
- les éléments des expressions régulières, qu'on écrit dans les programmes, entre « guillemets français »
- les espaces seront notées avec le symbole ␣

Exemple introductif : reconnaitre une adresse email

Le joker « . »

Il représente n'importe quel caractère

Le plus « + »

Il s'applique au caractère qui le précède et signifie au moins une fois.

Exemple : /.+@.+/ signifie une chaine d'au moins un caractère, suivie d'un arobase, suivi d'une autre chaine d'au moins un caractère.

Le point « \.»

Il faut *protéger* le point pour le différencier du joker. On utilise pour cela l'antislash, ce qui donne « \. »

Exemple : /.+@.+\..+/ signifie une chaine d'au moins un caractère, suivie d'un arobase, suivi d'une autre chaine d'au moins un caractère, d'un point, et enfin d'une autre chaine de caractère d'au moins un caractère.

La construction [^abc]

Cela signifie tous les caractères sauf a, b , et c (par exemple)

Exemple : /[^@]+@[^@]+\.[^@]+/ signifie une chaine d'au moins un caractère qui ne comprend aucun arobase, suivie d'un arobase, suivi d'une autre chaine d'au moins un caractère qui ne comprend aucun arobase, d'un point, et enfin d'une autre chaine de caractère d'au moins un caractère qui ne comprend aucun arobase.

Début et fin de ligne : « ^ » et « $ »

Pour s'assurer que la chaine (la ligne) ne contient pas d'autre caractère, on peut placer « ^ » pour indiquer le début d'une ligne, et « $ » pour la fin. On s'assure ainsi que la chaine entière correspond.

Exemple : /^[^@]+@[^@]+\.[^@]+$/

Nous avons ici une première définition du format d'une adresse email :

Une chaine **sans autre caractère** que :

1. une chaine d'au moins un caractère qui ne comprend aucun arobase,
2. suivie d'un arobase,
3. suivi d'une autre chaine d'au moins un caractère qui ne comprend aucun arobase,
4. suivie d'un point,
5. suivi d'une autre chaine de caractère d'au moins un caractère qui ne comprend aucun arobase.

Les expressions régulières en général

Les expressions régulières permettent de faire 3 types d'opérations sur les chaines :

1. la correspondance (matching), qui consiste à déterminer si une chaine répond bien à un certain « format », la notion de *format* coïncidant avec celle de *pattern*, de *motif*, et finalement avec la regex qui le définit.
2. le remplacement dans une chaine, ou la substitution
3. l'extraction d'information d'une chaine

La suite du document est un résumé de la partie de « Les expressions régulières par l'exemple », de Vincent Fourmond, qui traite de la correspondance

Les expressions régulières pour la correspondance (matching)

On parlera de correspondance entre une chaine et une regex quand la chaine répond aux spécifications de la regex. Mais on pourra aussi parler de correspondance quand seulement une partie de la chaine correspond à la regex. Dans ce cas, on soulignera cette partie.

Par exemple, pour une regex r =/[^@]+@[^@]+\.[^@]+/

'' abc@def.ghi '' correspond entièrement à la regex r

'' @@@abc@def.ghi@jkl'' match avec la regex r

Si on cherche *une* correspondance, on choisira celle qui commence le plus tôt dans la chaine d'entrée.

Si on cherche *des* correspondances, on choisira celles qui ne se chevauchent pas : celle qui commence le plus tôt, puis celle qui commence le plus tôt *après la fin de la première*, et ainsi de suite …

Les caractères usuels et le joker.

Les caractères représentent avant tout eux même : /chaine/ match avec '' chaine '' ou '' une chaine ''

Classe de caractères : Une classe de caractères désigne un ensemble de caractères. Elle peut se définir de plusieurs manières :

- avec la liste ou les plages de caractères en faisant partie « [abcdefgXRZSTU]» ou « [a-gXZR-U]» pour les minuscules de a à g, les majuscules entre R et U, le X ou le Z
- avec la liste ou les plages de caractères n'en faisant pas partie. « [^abcdefgXRZSTU]» ou « [^a-gXZR-U]» pour tout caractère sauf les minuscules de a à g, les majuscules entre R et U, le X et le Z

Joker : Suivant les environnements, le saut de ligne est, ou pas, inclus dans les caractères couverts par le joker.

Exercice : A quelle regex correspond n'importe quel nombre à 2 chiffres.

Exercice : A quelle regex correspond les nombres entre 10 et 49

L'échappement

Lorsqu'on souhaite utiliser un caractère spécial en tant que tel, il faut le faire précéder d'un caractère dit *d'échappement* : « \ ». Par exemple, « \. » pour le point, ou « \/ »pour le slash

Exercice : A quelle regex correspond les dates du type ''14.10.2004''

Alternatives : Fromage|Dessert ?

Pour exprimer un chois entre plusieurs possibilités, on utilise le caractère « | », ce qui correspond au OU logique.

Par exemple,voici la regex des nombres entiers formés de 1, 2 ou 3 chiffres :

/[0-9]| [0-9] [0-9]| [0-9] [0-9] [0-9]/

Exercice : A quelle regex correspond les nombres réels de la forme ''1.2'', ''3.'' ou ''.4'' dans lesquels il n'y a pas plus d'un chiffre de part de d'autre du point.

Le groupage : Jean-(Paul|Pierre)

La regex /creme brulée|crème au caramel|creme au chocolat/ est équivalente à /creme (brulée| au (caramel| chocolat))/

On peut ainsi réécrire la regex des nombres entiers formés de 1, 2 ou 3 chiffres :

/[0-9](| [0-9](| [0-9]))/

Zero, un, beaucoup

Les quantificateurs ont été crées pour faciliter la reconnaissance d'éléments dont on ne connait pas la longueur. Ces constructions se placent immédiatement après l'élément que l'on veut répéter.

Au moins une fois « + » : Répète l'élément qui le précède au moins une fois

Exercice : A quelle regex correspond l'ensemble des fractions, comme ''43/455'' ou ''3/56'' ?

Exercice : A quelle regex correspond les mots composées, non accentués, comme ''porte-clefs'' ?

Exercice : A quelle regex correspond les expressions additives de nombres entiers positifs, comme ''1+2'' ou '' 34+3445+5277+7355+0'' ?

Combien de fois exactement ?

Dans certains cas, il peut y avoir plusieurs possibilités pour le nombre exact de répétitions que contiennent les correspondances. En effet, si on cherche une correspondance de /<.+>/ dans ''avide'', obtient on ''avide'' ou ''avide'' ? La réponse est la deuxième. On dit que les quantificateurs sont **avides** : les correspondances d'un élément suivi d'un quantificateur sont les plus grandes possibles *sans gêner le reste de la regex*. Par exemple, /.+/ match avec ''Je mangerai''

Exercice : pour chercher les passages entre parenthèses dans un texte, peut on toujours utiliser /\(.+\)/ ?

Quand 2 quantificateurs sont en compétition dans une même regex (par exemple /.+a.+m/ dans ''Je mangerai demain''), la priorité est donnée à celui le plus à gauche, avant de passer au suivant, et ainsi de suite… Le groupe « .+a »match avec ''Je mangerai demain'' et non pas ''Je mangerai demain'', sans quoi on ne peut plus trouver de correspondance pour le groupe « .+m ». Le groupe « .+m » match alors avec ''Je mangerai demain''

Exercice : précisez le matching de chaque groupe de /.+[0-9]+.+/ dans ''Je viendrai à 15 heures''.

Exercice : Modifier la regex /\(.+\)/ de manière à ce qu'elle ne match pas avec les parenthèses imbriquées ou en série

Au plus une fois « ? » : Répète l'élément qui le précède 0 ou une fois

Exercice : quelle regex match avec ''mail'', ''email'', et ''e_mail '' au singulier et au pluriel ?

Zero ou plus « * » : Répète l'élément qui le précède 0 ou plusieurs fois.

Entre n et m fois « {n,m} » : Répète l'élément qui le précède entre n et m fois.

Au moins n fois « {n,} » : Répète l'élément qui le précède au moins n fois.

Début et fin de ligne « ^ » et « $ »

Exercice : trouver la regex pour les chaines représentant complètement un nombre impair

Raccourcis

/\d/ un chiffre

/\s/ un espace quelconque

/\w/ un caractère alphanumérique ou ''_''

Par exemple, la regex des nombres entiers relatifs : /(+|-)?\s*\d+/

Ces raccourcis doivent être préférés au classes du type /[a-zA-Z]/ car ils constituent un habile moyen de prendre en compte les caractères accentués.

Des raccourcis pour les complémentaires de ces classes sont aussi disponibles :

/\D/ tout sauf un chiffre

/\S/ tout sauf un espace

/\W/ tout sauf un caractère alphanumérique ou ''_''

Par exemple, la regex des suites de caractères séparés par des espaces : /(\S+\s+)+/

Exercice : Proposez une 2ème version pour la regex des adresse électroniques : sans espace avant et après l'(unique)arobase, un point et un mot de 2 à 4 lettres.

Non avides

On peut annuler l'avidité des quantificateurs avec « ? ». Les correspondances d'un élément non avide sont les plus petites possibles sans gêner le reste de la regex. Par exemple, la regex /a+ ?/ dans ''aaab'' sont ''aaab''. En revanche, celle avec /a+ ?b/ est ''aaab''

Par exemple, pour rechercher les textes mis en gras dans un doc html, la première idée serait /.* <\/b>/. Mais on obtient des correspondances trop étendues : '' gras pas gras gras ''. Il faut donc utiliser un quantificateur non avide /.* ?<\/b>/.

Exercice : Ecrire une regex dont les correspondances commencent en début de chaine et s'arrêtent à la première occurrence de ''prochaine''.

Les quantificateurs non avides sont particulièrement utiles pour analyser les codes de programmes. Par exemple, si on cherche à extraire toutes les chaines de caractères d'un programme en C, une première idée serait d'utiliser /''[^'']*''/

Mais cela pose des problèmes pour les chaines de caractères qui contiennent un guillemet américain (échappé) : \ ''. par exemple,

const char * chaine = ''guillemets:\'', ', etc''

La solution consiste donc à chercher une chaine formées de n'importe quels caractères et qui se termine aux premiers guillemets non précédés d'un caractère d'échappement :

/''.* ?[^\\]''/

Exercice : cette dernière version ne correspond pas aux chaines vides ''''. Comment y remédier ?

Ignorer la casse

Avec le modificateur i. par exemple /a/i match avec ''a'' et ''A''

Extraire de l'information : numérotation des groupes

Les captures et les remplacements : $1, $2, …

Bien utiliser les regex dans un programme

1. Les tests simples
   Par exemple, if(adresse =~/regex de l'adresse email valide/)
   then (utiliser (adresse))
   else print(erreur)
2. Utilisation des groupes capturés
3. Toutes les correspondances : cf analyse lexicale
4. Utilisation des remplacement

Les lignes

Il est naturel de travailler ligne par ligne. Pour cela, « . » ne comprend pas le saut de ligne pour éviter que /.*/ ne s'étende sur plusieurs lignes. Pour changer ce comportement, on peut utiliser le modificateur s :

Ainsi, ''a\nb'' correspond à /^.*$/s mais pas à /^.*$/

On peut de même modifier le comportement de « ^ » et de « $ » avec le modificateur m pour les faire correpondre, non pas au début et à la fin de la chaine, mais au début et à la fin d'une ligne.

L'international (la portabilité)

Les classe de type POSIX et mais surtout UNICODE avec « \p{class} », où class peut être n'importe quelle classe de la base de données de caractères unicode. Par exemple, « \p{L} » désigne les lettres.

Les regex dans vos langages

Perl, Javascript, PHP, C#, Python, C++, Emacs, C, Grep et Sed..

Un exemple avec Java :

Les outils sont repartis dans 2 classes : java.util.regex.Pattern une regex, et java.util.regex.Matcher l'objet qui permet d'effectuer des recherches dans une chaine.

```
Pattern re = Pattern.compile(''(\\d+)'') ; // (notez le doublement des \)

Matcher match = re.matcher(chaine); // chaine est la chaine où l'on souhaite effectuer des recherché

if(match.find()) {

        System.out.print(''match ok '') ;

}
```

On peut également recupérer les groups captures avec match.group(0), …

Exercice supplémentaire : Proposez votre dernière version pour la regex des adresse email (avant de pâlir devant les spécifications de rfc3696) en tenant compte des spécifications suivantes :

Exemples d'adresses valides :

- `Abc@example.com`
- `Abc.123@example.com`
- `user+mailbox/department=shipping@example.com`
- `!#$%&'*+-/=?^_`.{|}~@example.com`
- `"Abc@def"@example.com`
- `"Dany Bloggs"@example.com`
- `"Joe.\\Blow"@example.com`
- `Loïc.Accentué@voilà.fr`[3]

Exemples d'adresses non valides :

- `Abc.example.com`
  - Le caractère `@` manque.
- `Abc.@example.com`
  - Le caractère `.` n'est pas à l'intérieur de la partie locale.
- `Abc..123@example.com`
  - Le caractère `.` apparaît deux fois de suite.

Exercice supplémentaire : Analyse de log sur un serveur web [3] :

```
64.242.88.10 - - [07/Mar/2004:16:05:49 -0800] "GET
/twiki/bin/edit/Main/Double_bounce_sender?topicparent=Main.C
onfigurationVariables HTTP/1.1" 401 12846
64.242.88.10 - - [07/Mar/2004:16:06:51 -0800] "GET
/twiki/bin/rdiff/TWiki/NewUserTemplate?rev1=1.3&rev2=1.2
HTTP/1.1" 200 4523
64.242.88.10 - - [07/Mar/2004:16:10:02 -0800] "GET
/mailman/listinfo/hsdivision HTTP/1.1" 200 6291
64.242.88.10 - - [07/Mar/2004:16:11:58 -0800] "GET
/twiki/bin/view/TWiki/WikiSyntax HTTP/1.1" 200 7352
```

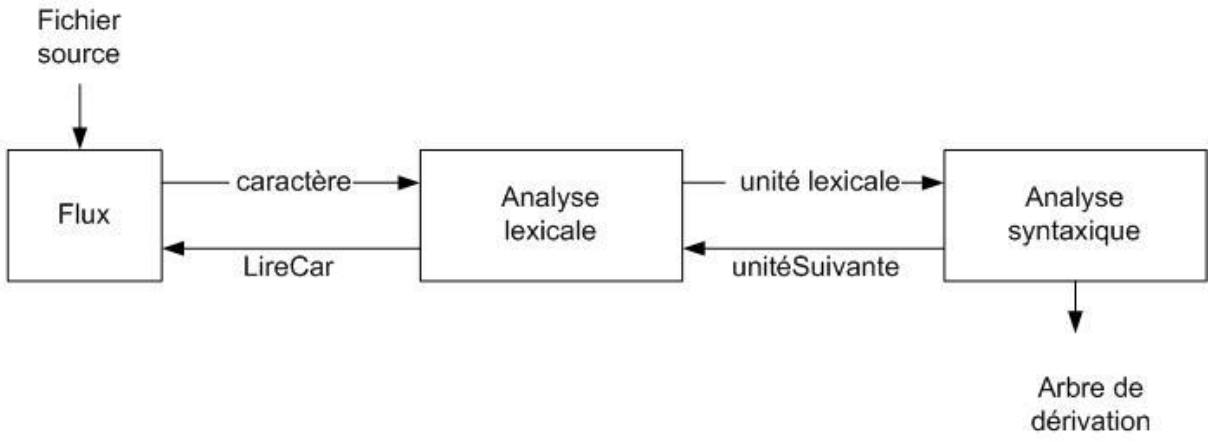
Les champs dans l'ordre:
- IP/hostname du serveur qui log
- [tiret = vide] --> identifiant du client si dispo
- [tiret = vide] --> Username de l'utilisateur si loggé
- date + heure
- "METHOD (GET/POST) url protocol"
- response code (200 = OK, 404 = NOT FOUND ... http://en.wikipedia.org/wiki/List_of_HTTP_status_codes)
- Taille de la page retournée en octets

Quelle regex (avec extractions et remplacements) permet de savoir combien de pages on été montrées (response code 200) entre telle heure et telle heure.

[3] à cette adresse, un fichier d'exemple de logs de serveur web.
http://www.monitorware.com/en/logsamples/apache.php

3. L'analyse lexicale

Relations entre l'analyse lexicale et l'analyse syntaxique. Ces deux phases fonctionnent de manière entremêlées :



L'analyse lexicale est la première phase de la compilation. Dans le texte source, qui se présente comme un flot de caractères, l'analyse lexicale reconnaît des *unités lexicales*, qui sont les mots avec lesquels les phrases sont formées (les symboles terminaux avec lesquels les mots du langage sont formées), et les présente à la phase suivante, l'analyse syntaxique.
Les principales sortes d'unités lexicales qu'on trouve dans les langages de programmation courants sont :

- les caractères spéciaux simples : +, =, etc.
- les caractères spéciaux doubles : <=, ++, etc.
- les mots-clés : if, while, etc.
- les constantes littérales : 123, -5, etc.
- et les identificateurs : i, j, vitesse_du_vent, etc.

A propos d'une unité lexicale reconnue dans le texte source on doit distinguer quatre notions importantes :

- l'*unité lexicale*, représentée généralement par un code conventionnel (PLUS, EGAL, INFEGAL, PLUSPLUS, …)
- le lexème, qui est la chaine de caractères correspondante ("+", "=", "<=", "++", "if", "while", "123", "-5", "i" et "vitesse_du_vent"
- éventuellement un attribut qui dépend de l'unité lexicale en question et qui la complète. Pour un nombre, il s'agit de sa valeur. Pour un identificateur, il s'agit d'un renvoie à une table des symboles, où sont placés tous les identificateurs rencontrés.
- le modèle, qui sert à spécifier l'unité lexicale (une sorte de pattern spécifiant tous les lexèmes qui concordent avec l'unité lexicale)

a. Définitions régulières

La spécification des modèles des unités lexicales se fait par l'intermédiaire d'expressions régulières. On les allège en introduisant des définitions régulières, permettant de donner un nom à certaines expressions en vue de leur réutilisation. On écrit donc :

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

…

$d_n \rightarrow r_n$

où chaque d_i désigne un nom et chaque r_i est une expression régulière sur $\Sigma \cup \{d_1, \dots, d_{i-1}\}$

Par exemple,

$lettre \rightarrow [a-zA-Z]$

$chiffre \rightarrow [0-9]$

$nombre$[4] $\rightarrow [1-9]^{+} chiffre^{*}(.chiffre^{*}[1-9]^{+})?\,([Ee][-+]?\,chiffre^{+})?$

$identificateur \rightarrow lettre(chiffre|lettre)^{*}$

b. Reconnaissance des unités lexicales

Après avoir spécifié les unités lexicales, il faut un programme qui les reconnait dans le fichier source. Un tel programme s'appelle un *analyseur lexical.*

Le principal client d'un analyseur lexical est l'analyseur syntaxique :

- L'analyseur lexical acquiert les caractères issus du fichier source un à un
- L'analyseur syntaxique n'a pas accès au fichier source, il n'acquiert ses données que par la fonction *unitéSuivante()*
- les seules données partagées entre l'analyseur lexical et l'analyseur syntaxique sont :
  - les codes des unités lexicales
  - les attributs de l'unité lexicale reconnue
  - la table des symboles[5]

La codification des unités lexicales se trouve généralement dans un fichier d'en tête comportant une série de définition comme

```
#define IDENTIF  1
#define NOMBRE   2
#define SI       3
#define ALORS    4
#define…
```

c. Ecrire un analyseur lexical : 3 alternatives

[4] attention au caractère « . » qui joue un rôle particulier de méta symbole (voir formalisme étendu des expressions régulières dans la partie consacrée à Lex.)

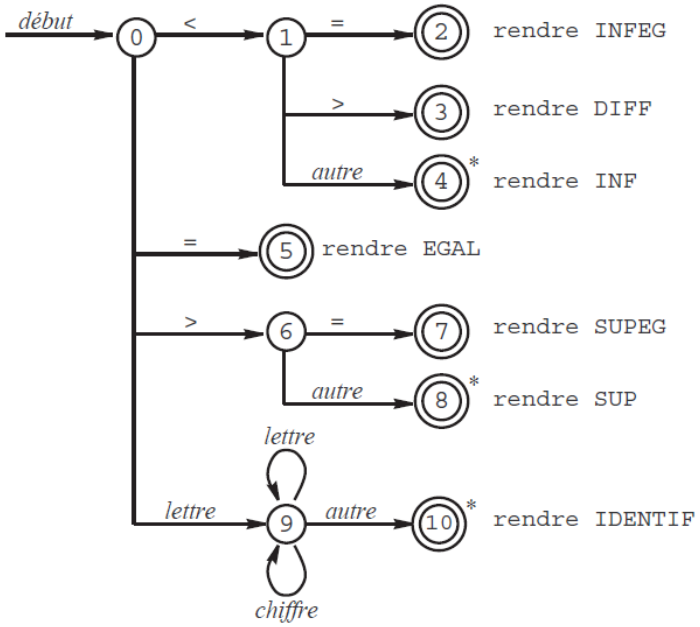
[5] La table précise le nom de l'identifiant, la mémoire réservée, le type, la portée et, pour les sous programmes, le nombre et le type des paramètres, le mode de transmission de chacun, le type du résultat.

Codage en dur des diagrammes de transition

On peut recourir aux automates à états finis, en adaptant un peu la formalisation du problème. Bien sur, on code l'automate en faisant correspondre les états finaux avec les expressions régulières des différentes unités lexicales. Mais à la différence du cadre classique du problème de reconnaissance via un automate, on ne teste pas un mot prédéfini mais le début d'un flux de caractères : il manque l'information qui indique la fin du mot.

De fait, on aura recourt à un artifice qui consiste à distinguer certains états finaux (alors marqués d'une étoile), pour indiquer que la reconnaissance s'est faite au prix d'un caractère au-delà de la fin du lexème.

Par exemple, voici un diagramme traduisant la reconnaissance des unités lexicales INFEG, DIFF, INF, EGAL, SUPEG, SUP et IDENTIF



En C, on obtient :

```
int uniteSuivante(void){
char c ; c=lireCar() ;                          //état 0
if(c=='<'){
       c=lireCar() ;                            //état 1
       if(c=='=') return INFEG;                 //état 2
       else  if(c=='>') return DIFF;            //état 3
       else { deLireCar();return INF;}          //état 4
}
else if (c=='=') return EGAL;                   //état 5
else if (c=='>'){
       c=lireCar() ;                            //état 6
       if(c=='=') return SUPEG;                 //état 7
       else { deLireCar();return SUP;}          //état 8
}
else if(estLettre(c)){
       lonLex=0;                                //état 9
       lexeme[lonLex++]=c;
       c=lireCar() ;
       while(estLettre(c)||estChiffre(c)){
              lexeme[lonLex++]=c;
              c=lireCar() ;
       }
       deLireCar(c) ;                           //état 10
       return IDENTIF ;
}
}
```

La fonction DeLire(car c) a pour effet d'ajouter à l'entrée le caractère c.

Pour la reconnaissance des mots réservés (par exemple, « if » qui répond à l'expression régulière des identificateurs), on a deux solutions. Soit, on les incorpore au diagramme de transition, soit on modifie l'unité lexicale des Identificateurs en identificateursOuMotsReservés, que l'on distingue ultérieurement en se basant sur un index des mots réservés.

Automates finis[6]

On code un automate à l'aide d'une fonction *transit(e1,c)=e2* qui signifie qu'il existe un arc étiqueté par un caractère c de l'état e1 à l'état e2. Dans notre exemple, la fonction transit peut être représentée par le tableau suivant (on a ajouté le fait que les blancs (espace, saut de ligne ou tabulation devant une unité lexicale ne changent rien).

TRANSIT	' '	'\t'	'\n'	'<'	'='	'>'	Lettre	Chiffre	autre
0	0	0	0	1	5	6	9	erreur	erreur
1	4*	4*	4*	4*	2	3	4*	4*	4*
6	8*	8*	8*	8*	7	8*	8*	8*	8*
9	10*	10*	10*	10*	10*	10*	9	9	10*

Par rapport à la première solution, l'utilisation d'une table de transition permet de faire l'essentiel du travail en répétant l'instruction *etat=transit[etat][lireCar()]*. En quelque sorte, les connaissances

[6] ici, on voit l'intérêt de disposer d'automates déterministes sans epsilon transition

procédurales sont changées en connaissances déclaratives. L'inconvénient est qu'on ne peut pas tricher, et qu'une telle table est parfois fastidieuse et délicate à établir.

On obtient le programme suivant :

```
#define NBR_ETATS …
#define NBR_CAR=256

int transit[NBR_ETATS][ NBR_CAR] ;
int final[NBR_ETATS+1] ;

int uniteSuivante(void){
        char c ;
        int etat = etatInitial ;
        while(!final[etat]){
                c=lireCar() ;
                etat=transit[etat][c] ;
        }
        if(final[etat]<0) delireCar(c);
        return abs(final[etat])-1;
}
```

On a ajouté un état supplémentaire, ayant le numéro NBR_ETATS, qui correspond à la mise en erreur de l'analyseur lexical, et une unité lexicale *erreur* pour signaler cela.

Le tableau final, indexé par les états, est défini comme :

- final[e]=0 si e n'est pas un état final (vu comme un booléen, finale[e] est faux)
- final[e]=U+1 si e est final, sans étoile, associé à l'unité lexicale U(vu comme un booléen, finale[e] est vrai puisque les unités lexicales sont numérotées à partir de 0)
- final[e]=-(U+1) si e est final, étoilé, associé à l'unité lexicale U (vu comme un booléen, finale[e] est aussi vrai)

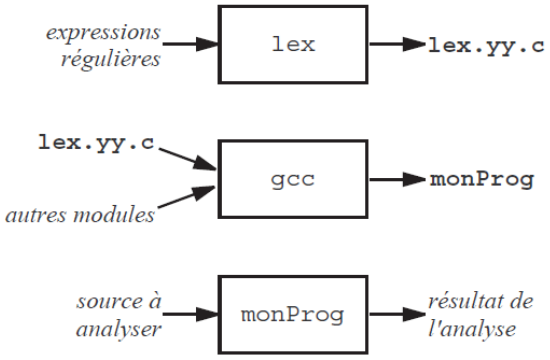
L'initialisation des tableaux transit et final doit ressembler à :

```
#define INFEG   2
#define DIFF    3
#define INF     4
#define EGAL    5
#define SUPEG   7
#define SUP     8
#define IDENTIF 10
#define ERRREUR 11
```

```
void initialiser(void){
int i,j;
for(i=0;i<NBR_ETATS;i++)final[i]=0;

final[2]=INFEG+1;
final[3]=DIFF+1;
final[4]=-(INF+1);
final[5]=EGAL+1;
final[7]=SUPEG+1;
final[8]=-(SUP+1);
final[10]=-(IDENTIF+1);
final[NBR_ETATS]= ERRREUR+1;

for(i=0;i<NBR_ETATS;i++)
        for(j=0;j<NBR_CAR;j++)
                transit[i][j]= NBR_ETATS;
}
transit[0][' ']=0;
transit[0]['\t']=0;
transit[0]['\n']=0;

transit[0]['<']=1;
transit[0]['=']=5;
transit[0]['>']=6;

for(j='A';j<='Z';j++) transit[0][j]=9;
for(j='a';j<='z';j++) transit[0][j]=9;

for(j=0;j<NBR_CAR;j++) transit[1][j]=4;
transit[1]['=']=2;
transit[1]['>']=3;

for(j=0;j<NBR_CAR;j++) transit[6][j]=8;
transit[6]['=']=7;

for(j=0;j<NBR_CAR;j++) transit[9][j]=10;

for(j='A';j<='Z';j++) transit[9][j]=9;
for(j='a';j<='z';j++) transit[9][j]=9;
for(j='0';j<='9';j++) transit[9][j]=9;
```

4. Lex[7], un générateur d'analyseurs lexicaux

Le programme Lex construit automatiquement la table de transition de l'analyseur lexical. il prend en entrée un ensemble d'expressions régulières, et produit en sortie le texte source d'un programme en C (*lex.yy.c*) qui, une fois compilé, est l'analyseur lexical correspondant au langage défini par les expressions régulières. Le fichier source *lex.yy.c* contient une fonction *int yytext(void)* qui est l'exacte homologue de notre fonction *int uniteSuivante(void)*



Structure d'un fichier source pour lex

Le fichier lex.yy.c comporte 3 sortes d'ingrédients :

1. des tables dont les valeurs sont établies à partir des définitions régulières.
2. des morceaux de code invariable, et notamment la boucle *etat=transit[etat][caractère]*
3. des morceaux de code en C trouvés dans le fichier lex, et recopiés tels quels

Un fichier lex doit avoir un nom se terminant par .l. Sa structure est la suivante :

```
%{
Partie 1 : déclarations pour le compilateur C
%}
Partie 2 : définitions régulières
%%
Partie 3 : règles
%%
Partie 4 : fonctions C supplémentaires
```

La partie 1 se compose de déclarations qui seront simplement recopiées au début du fichier produit. On y trouve souvent une directive #include qui produit l'inclusion du fichier d'en tête contenant les définitions des codes des unités lexicales. On peut également y trouver des variables globales, évoquées dans plusieurs fonctions du programme (en particulier pour la communication entre la fonction *main* et la fonction *yylex*). Cette partie et les symboles %{ et %} qui l'encadrent peuvent être omis.

La partie 4 se compose de fonctions C qui seront simplement recopiées à la fin du fichier produit. Cette partie peut être absente également (les symboles ´ %% qui la séparent de la troisième partie peuvent alors être omis.

La partie 2 comprend les définitions régulières, qui sont de la forme

[7] Flex, JFlex,Ocamllex, JavaCC

identificateur expressionRégulière

où *identificateur* est écrit au début de la ligne et séparé de *expressionRégulière* par des blancs. Les identificateurs ainsi définis peuvent être utilisés dans les règles et les définitions suivantes ; il faut les encadrer par des accolades.

lettre	[A-Za-z]
chiffre	[0-9]
alphanum	{lettre}|{chiffre}

La partie 3 comprend les règles, qui sont de la forme

expressionRégulière {action}

où *expressionRégulière* est écrit au début de ligne. *action* est un morceau de code C, qui sera recopie tel quel, au bon endroit, dans la fonction *yylex*. A la fin de la reconnaissance d'une unité lexicale, la chaîne reconnue est la valeur de la variable *yytext* de type char *. La variable *yylen* indique la longueur de l'unité lexicale contenue dans *yytext*.

```
if                          {return SI;}
then                        {return ALORS;}
{lettre}{alphanum}*         {return IDENTIF;}
 (+|-)?[0-9]+               {yylval = atoi(yytext); return NOMBRE;}
```

Dans l'exemple ci-dessus, les actions étant de la forme « return unité », leur signification est claire : quand une chaine du texte source est reconnue, la fonction *yylex* se termine en rendant comme résultat l'unité lexicale reconnue. Il faudra de nouveau appeler cette fonction pour que l'analyse du texte source reprenne.

```
%{
#define INT 1
#define ID  2
#define OP  3
%}
  int val;
```

$$mot \rightarrow [a-zA-Z]+$$
$$nombre \rightarrow [0-9]+$$

```
%%
              int strlg;

{nombre}      {val=atoi(yytext); return INT;}

[a-z]{mot}    {val=dico(yytext); return ID;}

[-+/*]        {val=yytext[0];    return OP;}

.|\n          ;
```

```
#define EOF  0
main()
{
int token;
while ((token = yylex()) != EOF) {
  switch (token) {
    case INT : ...
    ...
    }
  }
exit(0);
}
```

```
#define INT 1
#define ID  2
#define OP  3
  int val;
```

```
Déclarations lex
char *yytext;/*pointeur vers le lexème sélectionné*/
int yyleng; /*longueur du lexème sélectionné*/
FILE* yyin ;
FILE* yyout ;
int yyval; /*voir communication avec yacc*/
Initialisations Lex
Table 'transition' de l'automate
```

```
yylex()
{
 int strlg;
  while (1) {
    while (etat=transition[etat][c]) {
      if (final(etat)) {
        dernier_etat_final=etat;
        lgfinale = nbcalu;
        }
      c=input();  nbcalu++;
      }
    ...  /* : se recaler sur dernier etat final*/
    switch (etat) {
    case 1:
      {val=atoi(yytext); return INT;} break;
```

```
    case 2:
      {val=dico(yytext); return ID;} break;
    case 3:
      {val=yytext[0];    return OP;} break;
    case 4:
      ; break;
    case 5:
    ... /* default actions */
    }
  }
}
```

```
#define EOF  0
main()
{
int token;
while ((token = yylex()) != EOF) {
  switch (token) {
    case INT : ...
    ...
    }
  }
exit(0);
}
```

Ordre d'application des règles :

- lex essaye les règles dans leur ordre d'apparition dans le fichier.

- La règle reconnaissant la séquence la plus longue est appliquée

  a {printf("1");}
  aa {printf("2");}
  l'entrée aaa provoquera la sortie 21
- Lorsque deux règles reconnaissent la séquence la plus longue, la première est appliquée

  aa {printf("1");}
  aa {printf("2");}
  l'entrée aa provoquera la sortie 1
- Les caractères qui ne sont reconnus par aucune règle sont recopiés sur la sortie standard. ´

  aa {printf("1");}
  l'entrée aaa provoquera la sortie 1a

La fonction yywrap(void)

La fonction *yywrap* est appelée lorsque l'analyseur rencontre la fin du fichier à analyser (la fin du flux *yyin)*. Outre d'éventuelles actions utiles dans telle ou telle application particulière, cette fonction doit rendre une valeur non nulle pour indiquer que le flot d'entré est définitivement épuisé, ou bien ouvrir un autre flot d'entrée.

Version minimaliste de *yywrap :* { return 1;}

Ecrire des programmes en lex

On dispose tout d'abord de variables globales définies par Lex :

char *yytext;/*pointeur vers le lexème sélectionné*/

int yyleng; /*longueur du lexème sélectionné*/

FILE* yyin ; Si le pointeur est nul, la chaine d'entrée devra être saisie au clavier. Sinon, il s'agit d'un fichier texte.

FILE* yyout ; idem.

1. Programmes lex sans le concept de token

On spécifie dans les actions le code à appliquer. La fonction yylex est appelée une unique fois (généralement dans le main) et se termine quand toute la chaîne d'entrée est consommée, en appliquant à chaque unité lexicale reconnue les actions spécifiées.

2. Programmes lex avec des tokens

Les programmes s'écrivent autrement en utilisant le concept de 'token' : les règles reconnaissent des expressions, et les actions renvoient une indication ('token'). C'est ensuite la fonction (le main généralement) qui appelle l'analyseur (yylex()) qui se charge de faire 'avancer' un automate qui peut-être très complexe.

Avec ce format d'utilisation de 'lex', les règles s'écrivent plus simplement, et on sépare l'analyseur lexical de l'automate de calcul. Les tokens sont définis comme étant des constantes, qui sont renvoyées par les actions à la fin de l'évaluation d'une chaîne (instructions de la forme « return unité »); la fonction *yylex* se termine en rendant comme résultat l'unité lexicale reconnue. Il faudra de nouveau appeler cette fonction pour que l'analyse du texte source reprenne.

Remarques

Echo du texte analysé

```
[A-Za-z][A-Za-z0-9]*    { printf("%s", yytext); return IDENTIF; }
```

est équivalent à

```
[A-Za-z][A-Za-z0-9]*    { ECHO; return IDENTIF; }
```

On peut changer ce comportement par défaut en donnant une valeur µa la variable *yyin*, avant le premier appel de *yylex* ; par exemple :yyin = fopen(argv[1], "r") ;

Expressions régulières étendues

Metacaractère	définition
.	Tout caractère sauf nouvelle ligne
\n	Caractère de nouvelle ligne
e*	0 ou plusieurs copies de l'expression régulière e
e+	1ou plusieurs copies de l'expression régulière e
e?	0 ou 1 copie de l'expression régulière e
^	Début de ligne
$	Fin de ligne
a|b ou [ab]	a ou b
[^ab]	Tout caractère sauf a et b
(ab)+	Une ou plusieurs copies de ab (groupé)

"a+b"	Le litteral "a+b"
e{n}	n copies de l'expression régulière e
e{n,m}	Entre n et m copies de l'expression régulière e
e{n,}	N copies ou plus de l'expression régulière e
"." ou \.	Le point (" et \ pour les caractères spéciaux)
"*" ou *	L'étoile
e1/e2	Reconnaît l'expression régulière e1 si elle est suivie de e2
[x1 – x2]	N'importe quel caractère dont le code ascii se trouve entre celui de x1 et x2

Expression	**Lexèmes concordants**
abc	**abc**
abc*	**ab abc abcc abccc ...**
abc+	**abc abcc abccc ...**
a(bc)+	**abc abcbc abcbcbc ...**
a(bc)?	**a abc**
[abc] ou a|b|c	Un parmi: **a**, **b**, **c**
[ab]{n}	n répétitions de a ou b
[a-z]	N'importe quelle lettre au code ASCII entre a et z
[a\-z]	Un parmi : **a**, **-**, **z**
[-az]	Un parmi : **-**, **a**, **z**
[A-Za-z0-9]+	1 ou plusieurs caractères alphanumériques
[\t\n]+	Blanc (espace, tabulation, ou saut de ligne)
[^ab]	Tout sauf : **a**, **b**
[a^b]	Un parmi : **a**, **^**, **b**
[a|b]	Un parmi : **a**, **|**, **b**
a/b	reconnaît un a s'il est suivi par un b
^abc	reconnaît abc si placés en début de ligne
abc$	reconnaît abc si placés en fin de ligne

Le parenthèsage sert à induire l'ordre des opérations. La priorité des opérateurs est : priorité(*)>priorité(concaténation)>priorité(|)

Les états

Certaine règles peuvent être associées à des états. Elles ne s'appliquent que lorsque l'analyseur se trouve dans l'état spécifié.

```
%s COMM /* on définit l'état COMM */
%%
"/*" BEGIN COMM;      /* on rentre dans l'état COMM */
<COMM>. ;             /* on ignore tout ce qui se trouve*/
<COMM>\n ;            /* dans les commentaires */
<COMM>"*/" BEGIN INITIAL /* on revient à l'état standard */
```

Les règles qui ne sont pas associées à un état s'appliquent dans tous les états. Dans l'exemple précédent, toutes les règles qui ne sont pas liées à l'état COMM peuvent s'appliquer aussi dans l'état COMM. Pour éviter ce comportement, on utilise des états exclusifs: une règle qui n'est liée a aucun état ne s'applique pas dans un état exclusif:

```
%x COMM /* l'état COMM est exclusif */
%%
"/*" BEGIN COMM;       /* on rentre dans l'état COMM */
<COMM>. ;              /* on ignore tout ce qui se trouve */
<COMM>\n ;             /* dans les commentaires */
<COMM>"*/" BEGIN INITIAL /* on revient à l'état standard */
```

TP lex[8]

1. Installation de flex (lex) et bison (yacc) sous windows
   a. télécharger le setup de Flex http://gnuwin32.sourceforge.net/packages/flex.htm et veiller à ce que les chemins des répertoires d'installation, ainsi que le répertoire de travail ne comportent pas d'espace
   b. idem pour Bison : http://gnuwin32.sourceforge.net/packages/bison.htm
   c. ajouter flex, yacc et gcc à la variable d'environnement PATH (sans espace entre les différents chemins)

2. créer les fichiers suivants (attention aux guillemets) et compiler les 2 exemples suivants

Exemple 1 : avec le main dans le fichier lex, sans le concept de token

Fichier mystere.l

```
%{
  int chiffreOK=0;
%}
chiffre   [0-9]
lettre    [a-z]|[A-Z]
%%
{lettre}+          {printf("%s", yytext);}
{chiffre}+         { chiffreOK=1;}
 %%
#include <stdio.h>
int main () {
  yyin = fopen("in.txt", "r") ;
  yylex() ;
  printf("presence de chiffre : %d", chiffreOK );
  return 0;
}
int yywrap(void) {
return 1;
}
```

CompileEtRun.bat

```
flex mystere.l
gcc lex.yy.c -o monprog
monprog
pause
```

Fichier d'entrée in.txt

[8] Si vous ne disposez pas de GCC sous windows, télécharger MinGW : http://sourceforge.net/projects/mingw/files/

```
lknjfdAloili34,mlk   loip
```

Exemple 2 : avec plusieurs fichiers et en utilisant le concept de token

Dans cet exemple, le traitement ne se fait plus dans les actions associées à la reconnaissance des unités lexicales mais au sein d'un algorithme extérieur (jouant le rôle de l'automate)

Fichier[9] *unitesLexicales.h*

```
#define IDENTIF         256
#define NOMBRE          257
#define SI              258
#define ALORS           259
#define SINON           260
#define TANTQUE         261
#define FAIRE           262
#define RENDRE          263
#define EGAL            264
#define DIFF            265
#define INFEG           266
#define SUPEG           267

extern int valNombre;
extern char valIdentif[];
extern FILE * yyin;
```

[9] Pour les opérateurs simples on décide que tout caractère non reconnu par une autre expression régulière est une unité lexicale, et qu'elle est représentée par son propre code ASCII. Pour éviter des collisions entre ces codes ASCII et les unités lexicales nommées, on donne à ces dernières des valeurs supérieures à 255.

Fichier *analex.l*

```
%{
#include "unitesLexicales.h"
#include <string.h>
%}
chiffre [0-9]
lettre [A-Za-z]
%%
[" "\t\n]                       { ECHO; /* rien */ }
{chiffre}+                      { ECHO; valNombre = atoi(yytext); return NOMBRE; };
si                              { ECHO; return SI; }
alors                           { ECHO; return ALORS; }
sinon                           { ECHO; return SINON; }
tantque                         { ECHO; return TANTQUE; }
fairerendre                     { ECHO; return FAIRE; }
{lettre}({lettre}|{chiffre})*   {ECHO; strcpy(valIdentif, yytext); return IDENTIF; }
"=="                            { ECHO; return EGAL; }
"!="                            { ECHO; return DIFF; }
"<="                            { ECHO; return INFEG; }
">="                            { ECHO; return SUPEG; }
.                               { ECHO; return yytext[0]; }
%%

int valNombre;
char valIdentif[256];

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

Fichier principal.c

```
#include <stdio.h>
#include "unitesLexicales.h"

int main(void) {
int unite;
yyin = fopen("essai.txt", "r") ;
 do {
        unite = yylex();
        printf(" (unite: %d", unite);
        if (unite == NOMBRE)
        printf(" val: %d", valNombre);
        else if (unite == IDENTIF)
        printf(" '%s'", valIdentif);
        printf(")\n");
} while (unite != 0);
fclose(yyin);
return 0;
}
```

Fichier essai.txt

```
si x == 123 alors y = 0;
```

CompileEtRun.bat (Instructions de compilation)

```
flex analex.l
gcc lex.yy.c principal.c -o monprog
monprog
pause
```

Sortie attendue :

```
si (unite: 258)
x (unite: 256 'x')
== (unite: 264)
123 (unite: 257 val: 123)
alors (unite: 259)
y (unite: 256 'y')
= (unite: 61)
0 (unite: 257 0)
; (unite: 59)
```

Remarque : Pour obtenir la sortie dans un fichier (à la place de la sortie standard), on ajoute dans le main l'instruction *yyout= fopen("resultat.txt", "w")* ; et dans *unitesLexicales.h* la ligne *extern FILE * yyout;*

Exercices d'application

Application simple : Expressions régulières et reconnaissance des unités lexicales

1. Écrire un programme lex qui n'imprime que les commentaires d'un programme, sans utiliser les états de lex. Ceux-ci sont compris entre { } (donc accolades exclues).
2. Ecrire un programme lex qui compte le nombre de caractères, de mots et de lignes d'un fichier.
3. Écrire un programme lex qui numérote les lignes d'un fichier (hormis les lignes vides). Attention à ne pas avoir un numéro en trop après la dernière ligne.
4. Écrire un programme lex qui retire les commentaires /* …. */ d'un programme en C, sans utiliser les états de lex. Essayer votre programme avec un fichier comprenant un commentaire de taille > 16K. Conclure sur l'intérêt de l'utilisation des états.
5. Écrire un programme lex qui met en majuscule la première lettre des mots précédés d'un point.
6. URL
   Version 1 : Ecrivez un programme en flex qui fournit toutes les adresses (absolues) contenues dans une page html dont on fournit aussi l'URL de départ :

   Version 2 : Idem en classant adresses hypertexte, images, script, css ...

Intégration de l'analyseur : interaction avec le main plus complexes

7. Écrire un filtre qui transforme un texte en remplaçant le mot compilateur par beurk si la ligne début par a, par schtroumpf si la ligne débute par b et par youpi! si la ligne débute par c.
8. Ecrire un programme qui vérifie le balancement des parenthèses d'un programme d'un langage quelconque (C, Pascal, LISP, sh, ...): le programme doit sortir une erreur si à un moment donné le nb de parenthèses fermantes est supérieur au nombre de parenthèses ouvrantes, ou si à la fin du fichier d'entrée il manque des parenthèses fermantes.
9. Réaliser en lex un évaluateur d'expression postfixée (une expression dans laquelle les opérateurs sont placés après les opérandes, comme dans les calculettes HP). Par exemple,

   la notation infixe «(3+4)*5» s'écrit en postfixée «3 4 + 5 *»

   la notation infixe «3-4*5» s'écrit en postfixé «3 4 5 * -»

   Pour calculer, l'évaluateur utilisera une pile d'opérandes. Cette pile sera modélisée, soit comme une liste chainée, soit comme un (grand tableau) muni d'un entier qui mémorise la position du haut de pile.

   L'évaluateur réalisera les actions suivantes à la lecture d'un

   - entier : empilement de la valeur dans la pile des opérandes

- opérateur : dépilement de 2 opérandes, réalisation de l'opération, empilement du résultat.
- fin de ligne : impression du résultat ou message d'erreur si la pile contient un nb d'éléments différent de 1
- fin de fichier : terminaison du programme

5. Analyse syntaxique

L'analyse syntaxique reçoit une suite de token de l'analyseur lexical, détermine si cette suite est bien formée et produit certains traitements en sortie. Ces traitements peuvent être la génération de code dans le cas d'un compilateur, ou l'interprétation de code pour un exécuteur. Dans ces deux cas, les traitements ultérieurs peuvent prendre pour point de départ *l'arbre de syntaxe(ou arbre de dérivation)* du fichier source.

a. Langages et grammaires

Grammaires

Définition : une grammaire G = $< V_T, V_n, S \in V_n, P >$ où :

V_T (ou $\sum$) :	Alphabet terminal
V_n :	Alphabet non terminal (ou alphabet des variables)
$S \in V_n$:	Symbole de Start (démarrage)
P :	Ensembles de règles de production

Les règles doivent comporter au moins un non terminal à gauche :

P : $\{x \to y\}$ avec :

$x \in (\sum \cup V_n)^*.V_n{}^+.(\sum \cup V_n)^*$

$y \in (\sum \cup V_n)^*$

Dérivation immédiate

Soient $u\ et\ v \in (\sum \cup V_n)^*$

On note $u \to v$ si et seulement si $\exists p, s \quad tels\ que\ u = p.x.s$ et $v = p.y.s$ et $x \to y \in P$

Dérivation

On note $u \underset{G}{\Rightarrow} v$ (ou simplement $u \Rightarrow v$) si et seulement si il existe n dérivations immédiates à l'aide de règles de la grammaire G telles que $u \to \cdots \to \cdots \to v$

L'opérateur de dérivation correspond à la fermeture transitive de l'opérateur de dérivation immédiate.

Langage engendré par une grammaire

On note L(G) le langage engendré par une grammaire G. L(G) est défini comme :

$$L(\mathrm{G}) = \{\mathrm{w} \in \sum^*,\ S \underset{G}{\Rightarrow} w\}$$

Exemple :

Soit G une grammaire définie comme :

∑ ou V_T = {a, b} Alphabet terminal
$V_n = S, A$ Alphabet non terminal (variables)
$S \in V$: Start (démarrage)
P : Règles de production

$S \rightarrow abA \mid \varepsilon$
$A \rightarrow Aa \mid \varepsilon$

Les dérivations successives sont de la forme $S \rightarrow abSA \rightarrow abA \rightarrow abAa \rightarrow abAaa \rightarrow abaa$

Exercice : Expliquer pourquoi le langage engendré est $ab.a^* \quad \cup \{\varepsilon\} = L(G)$

Arbre de dérivation d'un mot

Considérons la grammaire suivante G=$< \{a, b\}, \{S, A, B\}, S, règles >$ avec les règles suivantes :

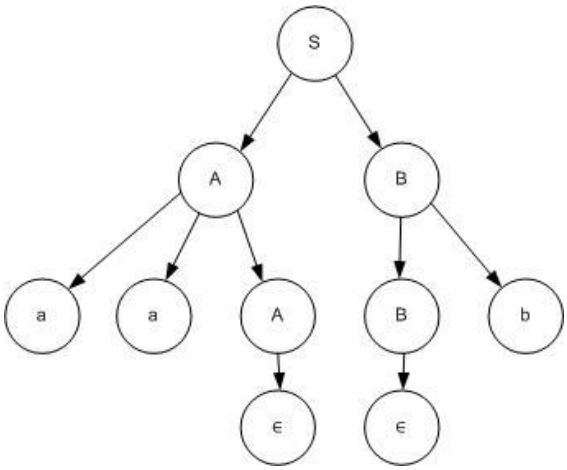
N°	Règle
1	S→AB
2	A→ aaA
3	A→ ε
4	B→Bb
5	B→ε

Ainsi que le mot w = aab.

On peut considérer,

- soit la dérivation gauche (leftmost : S→AB→aaAB→aaB→aab),
- soit la dérivation droite (rightmost : S→AB→Ab→aaAb→aab)

Si on représente une dérivation avec un arbre, les 2 dérivations donnent lieu au même arbre : l'ordre d'application des règles ne change pas l'arbre résultant.



Grammaires ambiguës

Une grammaire est dite *ambigüe* s'il existe un mot du langage engendré qui peut être obtenu par 2 arbres de dérivations différents.

Grammaires équivalentes

Deux grammaires sont dites

- *faiblement équivalentes* si elles engendrent le même langage
- *fortement équivalentes* si elles produisent le même langage et les mêmes dérivations

Exercices :

Exercice 1.

Trouver les grammaires (règles de production) qui engendrent les langages suivants :

$L_1 = \{ a^n,\ n \geq 0\}$

$L_2 = \{ a^n,\ n > 0\}$

$L_3 = \{ les\ mots\ sur\{a,b\}\ qui\ comportent\ un\ seul\ a\ au\ maximum\ \} = \{w \in \{a,b\}^*, |w|_a < 2\}$

$L_4 = \{ a^n b^m,\ n,m \geq 0\}$

$L_5 = \{ a^n b^n,\ n \geq 0\ \}$

$L_6 = \{Palindromes\ sur \sum = \{a,b\}\ \}$

$L_7 = \{ a^n b^n c^n,\ n > 0\}$

$L_8 = \{ a^n b^m c^{m+n},\ n > 0\}$
$L_9 = \{ a^n b^n c^n d^n,\ n > 0\}$

Exercice 2.

Déterminer les langages engendrés par les grammaires suivantes

G_1 :

$S \rightarrow aSbb \mid \varepsilon$

G_2 :

$S \rightarrow aB|bA$

$A \rightarrow a|aS|bAA$

B$\rightarrow b|bS|aBB$

G_3 :

$S \rightarrow aSb|bSa|SS|\epsilon$

Exercice 3 Langage de parenthèses

Soit $L \quad = le\ langage\ de\ Dyck = \{\ séquences\ de\ parenthèses\ bien\ équilibrées\ \}$

Trouver une grammaire G qui engendre L. G est elle ambigüe ?

Si oui, trouver une grammaire G' faiblement équivalente à G mais non ambigüe.

Exercice 4 Expressions arithmétiques

- Ecrire la grammaire des expressions arithmétique bien formées.
  $(1 + 34)/(5 * 9) \quad \in L(G)$
  $+25 - (\quad \notin L(G)$

- Incorporez le moins unaire : $-1 + 25/(4 + 2 * 3) \ \in L(G)$

Exercice 4 Signatures des fonctins en Java

Ecrire la grammaire des signatures de fonction en java.

Exemple :

void toto()

char tata(float f, int[][] z)
int toutou(int c2ERA)

-

-

Exercice 5 Commandes SQL

On souhaite écrire une grammaire chargée de vérifier la syntaxe des 2 commandes SQL suivantes (*select* et *insert*) avec des spécifications simplifiées. On ne s'intéressera pas à l'aspect sémantique dans les commandes (par exemple, savoir si un identificateur est un nom de table ou de champ, ou vérifier l'harmonisation des types dans l'expression des contraintes)

Commande *select* : (select … from …) OU (select … from … where …)

une partie obligatoire

entre *select* et *from* : *, un identificateur, ou une liste d'identificateurs séparés par des virgules

juste après le *from* : un identificateur, ou une liste d'identificateurs séparés par des virgules

une partie optionnelle :

après le *where* : une expression booléenne non ambigüe, c'est-à-dire des égalités, des non-égalités ou des comparaisons entre des termes qui sont des littéraux ou des identificateurs. On peut combiner les expressions booléennes avec les opérateurs logiques classiques (AND ou OR). Des parenthèses sont nécessaires (car `(true or true) and false` ne vaut pas `true or (true and false)`

Les littéraux sont soit des nombres, soit la référence null, soit des chaine de caractères entre apostrophes.

Les identificateurs (noms de champ, noms de tables, …) sont des suites de lettres ou de chiffres commençant obligatoirement par une lettre.

La commande est clôturée par un point-virgule

Commande *insert into* : (insert into …(…) values (…)) OU (insert into … values (…))

entre le *into* et le *values* : on a un identificateur, suivi ou non d'une liste d'identificateurs entre parenthèse séparés par une virgule.

après le *values* : on a une liste de littéraux entre parenthèses séparés par des virgules

La commande est clôturée par un point-virgule

Exemples de commandes SQL bien formées :

```
select abc from table1, table2 ;
select abc, def from table1, table2 ;
select * from table1, table2;
select abc, def from table1 where abc==der+1;
select abc, def from table1 where 1==2 AND (toto!=null OR
toto<'tata');
```

Et aussi avec le select imbriqué :

```
select abc, def from select abc, def from table1, table2 ;
;
insert into table1 values ('test', 5);
insert into table1 (champ1, champ2) values ('test', 5);
insert into table1 (champ1, champ2, champ3) values ('test', 5);
insert into table1 (champ1, champ2) values ('43', 5, null);
```

Exemples d'expressions SQL mal formées :

```
select abc, from table1, table2 ;
malformation : pas de virgule si un seul champ (abc)

select 1abc, def from table1, table2 ;
malformation : pas d'identificateur commençant par un chiffre

insert into (champ1, champ2) values ('test',  2);
malformation : pas d'identificateur avant les parenthèse entre le
into et le values
```

On rappelle que l'alphabet terminal se compose de tous les symboles présents dans les commandes. Les mots clefs réservés (select, insert, into, values, AND, OR, null, …) seront considérés comme des symboles élémentaires de l'alphabet terminal.

On aura donc

V_t={0,1,2,3,4,5,6,7,8,9,a, … z, A, .., Z,(,), ;, « , », AND, OR, =, !, *, select, insert, into, values, null}

Pour les éléments de l'alphabet non terminal, on choisira les conventions de nommage suivantes :

- L pour lettre
- C pour chiffre
- A pour caractère alphanumérique (une lettre ou un chiffre)
- ID pour les identificateurs
- LID pour liste d'identificateurs
- LITT pour littéraux
- LLITT pour liste de littéraux
- TERM pour les termes (littéraux ou identificateurs)
- OPR pour opérateur relationnel
- OPB pour opérateur booléen (OU et ET), pas de NON
- EXB pour expression booléenne
- S pour le symbole de départ

Ecrire la grammaire des 2 commandes SQL Select et Insert

Le problème de reconnaissance

Soient une grammaire G et un mot ω. Le problème de reconnaissance consiste à déterminer si $\omega \in L(G)$ ou si $\omega \notin L(G)$

Donc, s'il existe un chemin de dérivation tel que $S \Longrightarrow^{(*)} \omega$.

Formulation intuitive du problème de reconnaissance :

L'idée est que la reconnaissance dépend de la forme des règles de G :

Soit G une grammaire et L(G) le langage induit. Si on modifie une règle de G en allongeant la partie gauche de la règle, la taille de L(G) aura tendance à réduire. On a l'intuition que si la partie droite des règles augmente, L(G) aura tendance à augmenter en taille. La taille des langages induits, et accessoirement leur régularité, dépend donc de la forme des règles utilisées dans la grammaire. Evidemment, l'effet de l'allongement d'un règle n'est pas le même si les symboles ajoutés sont des terminaux ou des non terminaux. Noam Chomsky a été le premier (en 1956) à s'atteler à la catégorisation des formes des règles, dans l'effet que cette forme a sur la structure du langage induit. Le point original de son travail vient du fait qu'il catégorise la complexité d'un langage[10] par la complexité de la « machine » permettant de résoudre le problème de reconnaissance. Notons que son travail suit celui d'Alan Turing qui a mis en relation la solution algorithmique d'un problème et les machines de calculs permettant de les mettre en œuvre (1936), introduisant ainsi la notion de calculabilité.

[10] Notons que la plupart des langages contiennent un nombre infini de mots et que ce nombre est en bijection avec l'ensemble des entiers (il est énumérable). Du point de vue mathématique, il s'agit donc des « mêmes infinis ». il est donc nécessaire de trouver un critère de complexité qui dépasse la seule évaluation du nombre de mots induits dans le langage.

Classification des grammaires : Hiérarchie de Chomsky

Type 0 : Grammaires générales

Les règles sont de la forme :

$\gamma \rightarrow \beta$ où $\gamma, \beta \in (V_n \cup V_t)^* \; avec \; \gamma \; contient \; au \; moins \; un \; non \; terminal$

Les reconnaisseurs sont les Machines de Turing. La non appartenance d'un mot n'est pas prouvable (on ne peut pas prouver que la machine ne va pas s'arrêter : on ne sait pas combien de temps attendre avant de pouvoir conclure).
Classe de langages : **langages récursivement énumérables.**

Type 1 : Grammaires contextuelles

Les règles sont de la forme :

$\gamma A \beta \rightarrow \gamma \alpha \beta$ où $A \in V_n \; et \; \alpha, \gamma, \beta \in (V_n \cup V_t)^* \; avec \; \gamma \neq \varepsilon$

Remarque terminologique : $\gamma \ldots \beta \;\; est \; vu \; comme \; le \; contexte \; d'application \; de \; la \; règle \; A \rightarrow \alpha$

Les reconnaisseurs sont les automates linéairement bornés (= une Machines de Turing avec une mémoire linéairement borné). La non appartenance d'un mot n'est pas prouvable, elle est indécidable.
Classe de langages : **langages contextuels**

Type 2 : Grammaire hors contexte (ou grammaire algébrique)

Les règles sont de la forme :

$A \rightarrow \omega$ où $A \in V_n \; et \; \omega \in (V_n \cup V_t)^*$

Remarque terminologique : le non terminal A est remplacé indépendamment de son contexte, c'est-à-dire de sa place dans le mot.

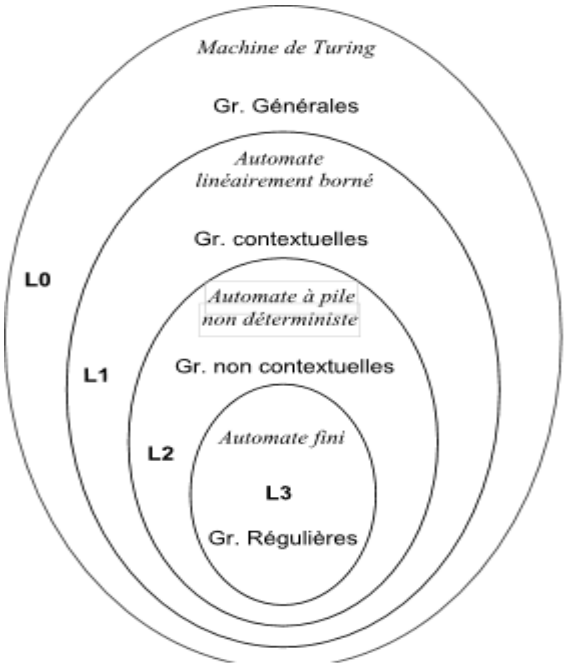
Les reconnaisseurs sont des automates à piles. L'appartenance et le non –appartenance sont directement lisibles sur l'état d'aboutissement du mot : final ou non final. Le problème est décidable.
Classe de langages : **langages non contextuels ou langages algébriques**

Types 3 : Grammaires régulières

Les règles sont de la forme :

$A \rightarrow aB$ où $A, B \in V_n \; et \; a \in V_t$
$A \rightarrow a$

Les reconnaisseurs sont des automates d'états finis. L'appartenance et le non –appartenance sont directement lisibles sur l'état d'aboutissement du mot : final ou non final. Le problème est décidable.
Classe de langages : **langages réguliers ou langages rationnels**


Machine de Turing
Gr. Générales
Automate
linéairement borné
Gr. contextuelles
L0
Automate à pile
non déterministe
L1
Gr. non contextuelles
Automate fini
L2
L3
Gr. Régulières

Un exemple de grammaire de langage naturel

On cherche à vérifier si la syntaxe de la phrase suivante est conforme à la grammaire de la langue française.

$$\mathbf{P} = le\ \ vieux\ chat\ attrape\ le\ petit\ rat$$

On dispose de l'alphabet de la langue française (un dictionnaire augmenté des verbes conjugués, des formes plurielles et féminines des noms et adjectifs) :

$$\sum = \{le, vieux, petit, chat, rat, attrape\}$$

On dispose des règles de grammaire suivante :

1. PHRASE → GROUPESUJET VERBE GROUPECOMPLÉMENTOBJET
2. GROUPESUJET → GROUPENOMINAL
3. GROUPECOMPLÉMENTOBJET → GROUPENOMINAL
4. GROUPENOMINAL → ARTICLE NOM ADJECTIF | ARTICLE ADJECTIF NOM

} Règles syntaxiques

5. ARTICLE→ le
6. VERBE → $attrape$
7. NOM → $chat|\ rat$ [11]
8. ADJECTIF → $vieux|\ petit$

} Règles lexicales

Opérations de récriture à l'aide d'une grammaire

N° de la règle	*Résultat*
-	PHRASE
1	GROUPESUJET VERBE GROUPECOMPLÉMENTOBJET
2	GROUPENOMINAL VERBE GROUPECOMPLÉMENTOBJET
2	GROUPENOMINAL VERBE GROUPENOMINAL
4b	ARTICLE ADJECTIF NOM VERBE GROUPENOMINAL
4b	ARTICLE ADJECTIF NOM VERBE ARTICLE ADJECTIF NOM
5	le ADJECTIF NOM VERBE ARTICLE ADJECTIF NOM
…	…
…	$le\ \ vieux$ NOM $attrape\ le$ ADJECTIF NOM
7a	$le\ \ vieux\ chat\ attrape\ le$ ADJECTIF NOM
8b	$le\ \ vieux\ chat\ attrape\ le\ petit$ NOM
7b	$le\ \ vieux\ chat\ attrape\ le\ petit\ rat$

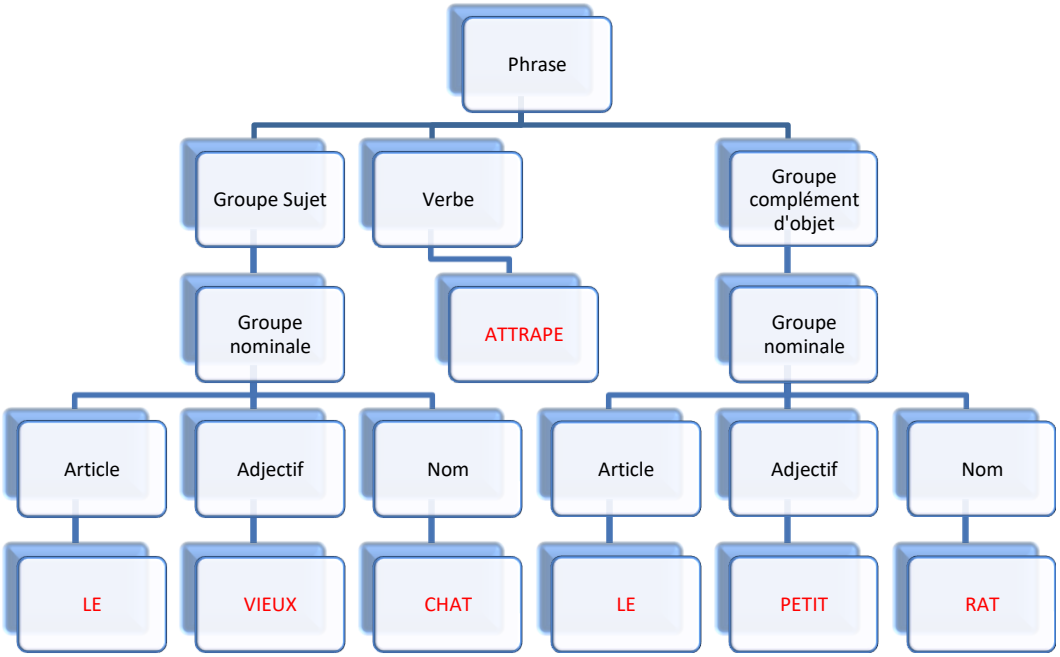
Résultat de l'analyse syntaxique :

On arrive bien à obtenir notre phrase P en réecrivant le symbole de départ PHRASE . On dit que P dérive de PHRASE, OU QUE P est obtenu par dérivation à partir de PHRASE. Ceci signifie que P est une phrase conforme à la grammaire, P fait partie des phrases bien formées relativement à la grammaire.

[11] Équivalent à NOM → $chat\ \ ou\ \ $ NOM $\rightarrow\ rat$

Il faut bien comprendre que l'opération de base dans la programmation à partir de grammaires est l'opération de réécriture (vs l'instanciation pour la programmation classique)

On peut représenter cette dérivation à l'aide d'un arbre. La phrase à analyser coïncide avec la frontière de l'arbre :



Il faut bien comprendre que l'arbre syntaxique d'une phrase a 2 rôles :

1. L'arbre assure la bonne formation de celle-ci
2. L'arbre de dérivation est le support idéal pour des traitements ultérieurs : calculs, traduction, …

-

Automates à piles et langages hors-contexte

Automates à piles

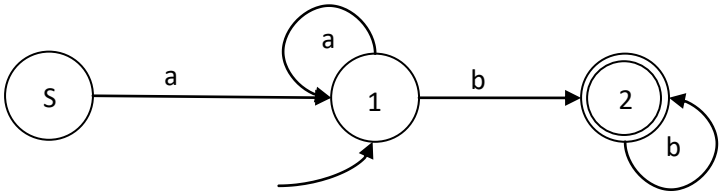
Intuitivement, on pressent que $\boldsymbol{L = \{a^n b^n, n \geq 1\}}$ ne peut pas être engendré par un automate fini[12] : il faut pouvoir enregistrer le nombre de a pour pouvoir produire autant de b. Il faut donc une forme de mémoire.
C'est le rôle des automates à pile. On rajoute des symboles de pile (Γ). On précise les opérations d'empilement/dépilement dans les transitions :

[Etat de départ, symbole à dépiler] symbole [Etat d'arrivé, symbole à empiler]

Exemple :

$\boldsymbol{L = \{a^n b^n, n \geq 1\}}$

Les symboles de pile sont quelconques (des jetons par exemple, notés u), seul leur nombre nous intéresse.



[S, ε]a[1,u]
[1, ε]a[1,u]
[1,u]a[2, ε]
[2,u]a[2, ε]

Critère d'appartenance d'un mot :

Pour qu'un mot appartienne au langage ($\omega \in L$), il faut :

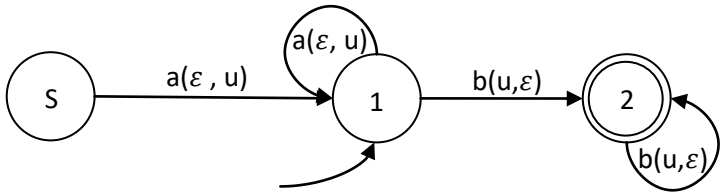
1. Atteindre l'état final
2. Avoir la pile vide
3. La conjonction des deux conditions précédentes.

Cas d'échec :

$a^3 b^2 \rightarrow la\ pile\ n'est\ pas\ vide$
$a^2 b^3 \rightarrow la\ lecture\ est\ bloquée\ (pas\ assez\ en\ pile\ pour\ atteindre\ l'état final)$

Autre représentation:

[12] cf lemme de l'étoile

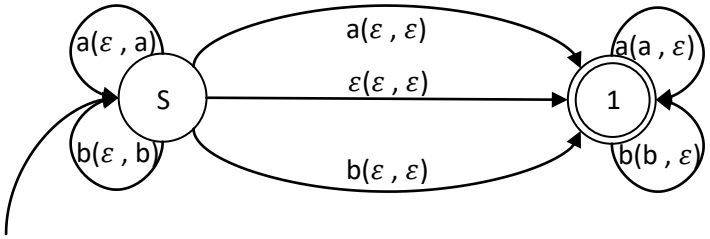


	a	b	ε	Accepté
S	1, ε,u			
1	1, ε,u	2, u, ε		
2		2, u, ε		Oui

Exercice

Soit L le langage des palindromes sur $\sum = \{a, b\}$.

Trouver l'automate à pile qui engendre L (ici, l'alphabet de la pile est le même que celui du langage).



Transformation des grammaires hors contexte

Suppression des récursivités gauche dans les grammaires hors contexte

on remplace

$\boldsymbol{A \rightarrow A\beta_1 |\ A\beta_2 |\ A\beta_3 |\ A\beta_4 |\ \alpha_1 | \alpha_2}$

par

$\boldsymbol{A \rightarrow\ \alpha_1 A' | \alpha_2 A''}$

$\boldsymbol{A' \rightarrow \epsilon | \beta_1 A' |\ \beta_2 A' |\ \beta_3 A' |\ \beta_4 A'}$

Factorisation à gauche des grammaires hors contexte

on remplace

$X \rightarrow ab|abbX|abbbX$

par:

$\boldsymbol{X \rightarrow abY}$

$\boldsymbol{Y \rightarrow \epsilon | bX | bbX}$

puis à nouveau factorisation de b

Exercice :

Soit G la grammaire non ambiguë des expressions arithmétiques

$$\boldsymbol{E \rightarrow E + T | T}$$
$$\boldsymbol{T \rightarrow T * F | F}$$
$$\boldsymbol{F \rightarrow id | (E)}$$

Supprimer recursivités gauches et factoriser à gauche la grammaire G

Analyse syntaxique ascendante

Un analyseur applique une procédure systématique permettant d'analyser une chaine d'entrée. On se place dans le cas où l'analyseur est associé à un analyseur lexical : les éléments de la chaine d'entrée sont des unités lexicales.

En termes de processus, l'analyse syntaxique se définit comme :

Entrée :

- une grammaire hors contexte G
- une chaine d'entrée w

Sortie :

- réussite : w est accepté par G. Dans ce cas, l'analyseur fournit l'arbre d'analyse associé à w
- échec : w n'est pas accepté, il ne fait pas partie du langage engendré par la grammaire G

Il est impératif de concevoir l'analyse syntaxique comme la (tentative de) construction de l'arbre d'analyse de la chaine d'entrée.

Réductions

Dans les analyseurs ascendants, la reconnaissance de la chaine d'entrée se fait de bas en haut : des feuilles de l'arbre à sa racine. On cherche à savoir s'il est possible de *réduire* la chaine d'entrée (w) au symbole de départ (S), par une application judicieuse des règles de la grammaire. A chaque étape de la réduction, si une sous chaine correspond à la partie droite d'une règle, elle est remplacée par le non terminal qui constitue la partie gauche de la règle. Les décisions clés à prendre au cours de l'analyse descendante concernent le moment où il faut réduire, et la production à utiliser pour réduire.

Par exemple, pour la grammaire G des expressions de variables (id) :

$$E \rightarrow E + T \mid T$$

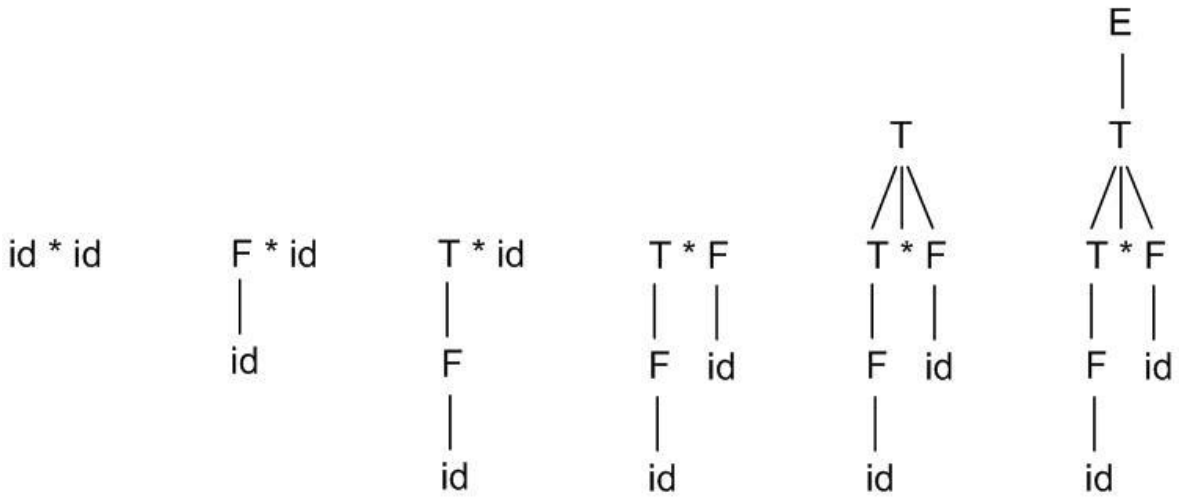
$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid \boldsymbol{id}$$

Les étapes de la réduction d'une chaine d'entrée $w = \boldsymbol{id} * \boldsymbol{id}$:

$$\boldsymbol{id} * \boldsymbol{id}, \quad \boldsymbol{F} * \boldsymbol{id}, \quad \boldsymbol{T} * \boldsymbol{id}, \quad \boldsymbol{T} * \boldsymbol{F}, \quad \boldsymbol{T}, \quad \boldsymbol{E}$$

Chaque étape correspond à une phase de la construction de l'arbre de syntaxe de la chaine w.



1. la première réduction réduit le ***id*** le plus à gauche et le transforme en F
2. La seconde réduit F en T.
3. ici on a le choix :
    a. soit réduire T en E
    b. soit réduire le second ***id*** en F. C'est cette option qui est choisie car c'est la seule qui conduise à l'analyse de la chaine w : sinon, on reste bloqué avec *E*id*
4. Ensuite T*F en T
5. Et enfin T en E

Les étapes d'une réduction correspondent aux étapes d'une dérivation droite de la chaine d'entrée mais dans l'ordre inverse :

$$E \rightarrow T \rightarrow T * F \rightarrow T * id \rightarrow id * id$$

Analyseurs syntaxiques par décalage-réduction

Principe :

- une pile stocke des symboles grammaticaux (des proto phrases)
- un tampon d'entrée stocke la partie de la chaine restant à analyser.

Etat initial de l'analyse : Au départ, la pile est vide et et la chaine w est sur l'entrée :

Pile	Entrée
$	$id_1 * id_2$$

Etapes intermédiaires : au cours d'un parcours gauche-droite de la chaine d'entrée, l'analyseur *décale* des symboles terminaux de la la chaine d'entrée vers la pile, jusqu'à être en mesure de *réduire* le haut de la pile en la partie gauche de la règle appropriée. Ce cycle est itéré.

Etat final de l'analyse : l'entrée est vide et la pile est réduite au symbole de départ. L'analyseur annonce le succès de l'analyse.

Pile	Entrée
$S	$

Exemple :

Pile	Entrée	Actions
$	$id_1 * id_2$$	décaler
$$id_1$	$* id_2$$	Réduire par $F \rightarrow id$
$$F$	$* id_2$$	Réduire par $T \rightarrow F$
$$T$	$* id_2$$	décaler
$$T*$	$id_2$$	décaler
$$T* id_2$	$	Réduire par $F \rightarrow id$
$$T* F$	$	Réduire par $T \rightarrow T * F$
$$T$	$	Réduire par $E \rightarrow T$
$$E$	$	Analyse réussie

Symbole de pré-vision : La majorité des analyseurs sont LR(k). Cela signifie que le choix de l'analyseur syntaxique (par exemple le fait de décider s'il faut réduire ou décaler à l'étape 4) est basé sur les k prochains symboles de la chaine d'entrée. Attention, il ne s'agit pas d'un conflit car une seule des possibilités conduit à l'analyse de la chaine. Dans ce cours, on ne va considérer que les grammaires LR(1), c'est-à-dire où un seul symbole de pré-vision est considéré (ici ce symbole vaut *).

Exercice : analyse par décalage-réduction – 6pts

Soit G la grammaire suivante qui définit les expressions arithmétiques restreintes aux nombres formés des chiffres de 1 à 4, à l'addition et à la multiplication.

S →S+S | S*S | N

N→ NC | C

C → 1 | 2 | 3 | 4

Les symboles sont catégorisés selon :

- symboles terminaux (tokens) : 1, 2, 3, 4, +, *
- symboles non terminaux : S, N et C

Soit w la chaine d'entrée à analyser : w = 12+3*4

1. Donner 2 arbres dérivation pour le mot w ; Pour chaque arbre, on précisera les étapes d'une analyse ascendante par décalage réduction de la chaine w (pour les réductions, on précisera juste le numéro de la règle utilisée parmi les 9 règles de la grammaire, numérotées dans l'ordre : par exemple, la règle S→S*S est la règle numéro 2)
2. Lequel des 2 arbres vous semble il le plus approprié pour de futurs traitements (comme calculer la valeur de l'expression par exemple)

Arbre A :

Pile	Entrée	Actions
$	$12 + 3 * 4$$	

Arbre B :

Pile	Entrée	Actions
$	$12 + 3 * 4$$	

Conflits pendant une analyse par décalage réduction

Conflit n°1 : le conflit décalage réduction

Par exemple, dans le cas des règles :

$$inst \rightarrow \boldsymbol{si}\ expr\ \boldsymbol{alors}\ inst \ \mid\ \boldsymbol{si}\ expr\ \boldsymbol{alors}\ inst\ \boldsymbol{sinon}\ instr \ \mid\ autre \ldots$$

Dans la configuration suivante :

Pile	Entrée
\$ … $\boldsymbol{si}\ expr\ \boldsymbol{alors}\ inst$	$\boldsymbol{sinon}$ ….\$

On ne peut pas dire s'il faut

- réduire en inst puis rechercher un autre alors pour décaler le sinon
- décaler le sinon et rechercher une inst à droite pour réduire en inst.

Conflit n°2 : le conflit réduction/réduction

Ce conflit advient dans le cas où plusieurs règle la même une partie droite, lorsque celle-ci apparait au sommet de la pile :

$$accesTableau \rightarrow identificateur \quad '('\ listeExpr')'$$
$$appelFonction \rightarrow identificateur \quad '('\ listeExpr\ ')'$$

Utiliser des grammaires ambiguës

Grammaire ambigüe des expressions arithmétiques : $E \rightarrow E + E \mid E * E \mid (E) \mid \boldsymbol{id}$

Grammaire non ambigüe des expressions arithmétiques :

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (\ E\ ) \mid \boldsymbol{id}$$

Cette dernière grammaire donne la priorité à * par rapport à + (par ex. 2+3*4=2+(3*4)) et rend les 2 opérateurs associatifs à gauche (par ex. 2+2+2=(2+2)+2) (le vérifier !)

Si l'on souhaite utiliser des grammaires ambiguës (plus concises que leurs homologues non ambigus), il faut pouvoir spécifier dans tous les cas des règles de désambigüisation qui n'autorisent qu'un seul arbre de syntaxe. Dans la section sur Yacc, on verra comment procéder à de tels spécifications.

6. Yacc, un générateur d'analyseurs syntaxiques (Yet another Compiler Compiler)

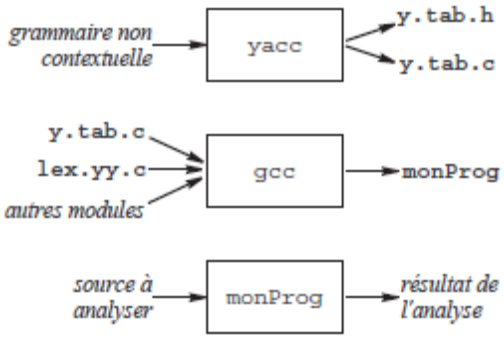


Schéma d'un fichier yacc

```
%{

Partie 1 : déclarations pour le compilateur C

%}

Partie 2 : déclarations pour yacc

%%

Partie 3 : schémas de traduction

%%

Partie 4 : fonctions C supplémentaires
```

Les parties 1 et 4 sont simplement recopiées dans le fichier produit, respectivement a la fin et au début de ce dernier. Chacune des deux parties peut être absente.

1. Partie 1 : Déclarations C
   C'est l'endroit adéquate pour déclarer des variables temporaires utilisées par les règles de traduction ou les procédures des partie 2 et 3, ainsi que les déclarations c ordinaires
   `#include <ctype.c>`
2. Partie 2 : Déclaration yacc
   On trouve la déclaration des unités lexicales de la grammaire (tokens)
   `%token nombre`
3. Partie 3 : règles de traduction
   Elles sont de la forme
   `<partie gauche> : <partie droite 1> {<Action sémantique 1>}`

|<partie droite 2> {<Action sémantique 2>}
|<partie droite 3> {<Action sémantique 3>}
;

Un caractère unique entre apostrophe 'c' est considéré comme le symbole terminal c, ainsi que comme l'entier qui est le code de l'unité lexicale représentée par ce caractère. Pour cette raison, le code des tokens introduits dans lex commencent à 256. La partie gauche de la première production est considérée comme le symbole de départ.
Une action sémantique yacc est une séquence d'instructions C. Dans une action sémantique, le symbole $$ désigne la valeur associée au non terminal de la partie gauche, alors que $i dénote la valeur associée au $i^{ème}$ symbole (terminal ou non) de la partie droite de la règle. L'action sémantique est exécutée au moment de la réduction est effectuée. L'action sémantique par défaut est {$$=$1 ;}

4. Routine C
   L'analyseur lexical lex produit des unités lexicales constituées d'un nom (un entier) et de sa valeur associée. Si un token de code NOMBRE est retourné par l'analyseur lexical, l'attribut doit contenir la valeur du nombre. Cette valeur est communiquée à yacc via la variable yyval. Le type par défaut de yyval est entier mais il est possible de la modifier (en ajoutant #define YYSTYPE double pour avoir yyval de type double par exemple).

Utiliser Yacc avec des grammaires ambigues[13]

1. Règle de résolution des conflits : comportement par défaut

Conflit réduction/réduction : c'est la réduction listée en premier qui est choisie

Conflit réduction/décalage : c'est le décalage qui est choisi

2. Règle de résolution des conflits : mécanisme général

Dans la partie 2, on peut définir des associativités et des priorités.

```
%left '+' '-'
%left '*' '/'
%right '^'
%nonassoc '<'
```

Cela signifie :

- les terminaux '+' '-' '*' '/' sont associatifs à gauche.
- '+' '-' ont même priorité, mais celle ci est moindre que celle de '*' '/' car ces derniers sont listés après
- le terminal '^' est associatif à droite
- '<' est non associatif (on ne peut pas combiner 2 occurrences de suite)

Avec ces associativités et ces priorités, les conflits sont résolus :

[13] Invoquer yacc avec l'option –v permet de produire un fichier y.output qui récence les conflits rencontrés durant l'analyse.

Conflit réduction/décalage :

- on associe à chaque règle une *priorité qui est celle de son terminal le plus à droite.*
- pour résoudre le conflit, on compare les priorités de la règle, et du terminal de pré-vision
    - si la priorité de la règle est supérieure, on réduit. Si elle est inférieure, on décale.
    - si la priorité de la règle est égale,
        - on réduit si le terminal est associatif à gauche
        - on décale sinon.

Dans les cas où le terminal le plus à droite d'une règle ne fournit pas la bonne priorité, on peut forcer une priorité en ajoutant à une production l'étiquette %prec <terminal> . La priorité et l'associativité est alors la même que celle de <terminal>.

TP YACC

1. Installation
   Téléchargez *Bison* ou *winBison*[14] et lancer la compilation en exécutant calc.bat dans le dossier calcV00
   Commentaires :
   - calc.bat : les renommages sont superflus
   - global.h permet de préciser le type la variable yyval (en définissant YYSTYPE) et de la partager
   - calc.l
     - %option noyywrap permet d'éviter d'écrire la forme standard de la fonction yywrap (cf lex)
     - à la différence de l'utilisation de lex vue en tp, les actions dans les règles de lex contiennent des instructions return. C'est l'approche de lex avec token : les appels de yylex() sont gérés à l'extérieur. Quand on a renvoyé un token, on attend la prochaine sollicitation (ici par yacc) pour envoyer le suivant. Les token sont codés sous la forme d'entiers (cf calc.h)
   - calc.y
     - les printf servent à suivre la construction de l'arbre
     - main et yyerror dans leur forme minimale

2. CalcV00
   - Expliquez la présence de conflits
   - Expliquez les résultats obtenus à partir des expressions suivantes :
     - 2*3+4
     - 4+2*3
   - Essayez d'ajouter une priorité et une associativité aux tokens + - * et /
   - Comment traiter le moins unaire ?
   - Réponse : CalcV01
   - Améliorer la calculette de manière à pouvoir calculer une séquence d'expressions (sans avoir à relancer calc.exe)
   - Réponse : CalcV02
3. CalcV10
   On souhaite maintenant prendre en compte les chaines de caractères (pour les noms de variable) dans la chaine d'entrée : sans effet sur le calcul et admises uniquement en préfixe d'un nombre : 2 + toto14 +1 doit afficher 17, tout comme 2 + 14 +1, mais 2 + 14toto +1 renvoie une erreur
   Commentaires de :
   - plus besoin de global.h.
   - calc.l

[14] dans ce cas, changer le nom de l'exécutable yacc dans calc.bat

    - on a ajouté un token nommé VARIABLE pour les chaines de caractère de l'alphabet
- calc.y
    - remarquez la directive union pour le type de YYSTYPE et le nommage des champs
    - et le typage des token et des non terminaux (avec %type) en fonction du champ de l'union

4. Exercice

Modifier le fichier calc.y de Calc10 de manière à ce qu'il se comporte comme suit :

```
user: 3 * (4 + 5)
calc: 27
user: x = 3 * (4 + 5) + 1
calc: x vaut 28
user: y = 5
calc  y vaut 5
user: a = x+2*y
calc: a vaut 38
…
```

V1 : On se limitera aux noms de variable à une lettre. Les valeurs des variables seront stockées dans un tableau de 26 doubles

- Réponse : CalcV11

V2 : en prenant en compte les chaines de caractères pour les variables (implémenter une liste chainée)

Exercice sur les conflits décalage/réduction et le système de gestion des priorités de yacc :

Exercice 2.(extrait de CC 2015 2016) Yacc – 14pts

On décide d'utiliser la grammaire ambigue, en utilisant le système de priorités de Yacc pour gérer les conflits. (cf code à la fin du sujet)

1. Donner la sortie du programme généré sur l'entrée w=`2a3b5`

2. Donner la sortie du programme généré sur l'entrée w=`3a3a3`

3. [Version modifiée] On modifie les priorités des tokens A et B, avec le code suivant :

```
%left A
%left B
```

Donner la sortie du programme généré sur l'entrée w=`2a3b5`

4. [Version modifiée] On modifie les priorités des tokens A et B, avec le code suivant :

```
%right A
%left B
```

Donner la sortie du programme généré sur l'entrée w=`2a3b5`

Fichier LEX

```
%option noyywrap
%{
#include "global.h"
#include "calc.h"
#include <stdlib.h>
%}
blancs    [ \t]+
chiffre   [0-9]
entier    {chiffre}+
exposant  [eE][+-]?{entier}
reel    {entier}("."{entier})?{exposant}?
%%
{blancs}  { /* On ignore */ }
{reel}    {
      yylval=atof(yytext);
      return(NOMBRE);
    }
"a"   return(A);
"b"   return(B);
"\n"  return(FIN);
```

Fichier YACC

```
%{
#include "global.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
%}

%token  A B NOMBRE
%token  FIN

%left B
%left A

%start Input
%%
Input:
    FIN
  | Expression FIN    { printf("Resultat : %f\n",$1); }
  ;
Expression:
    NOMBRE      { $$=$1; printf("Nombre %f\n",$1);}
  | Expression A Expression  { $$=$1-$3*$1-$3;printf("%f a %f =
%f\n",$1,$3,$$); }
  | Expression B Expression { $$=$1*$1-$3; printf("%f b %f =
%f\n",$1,$3,$$);}
  ;
%%
int yyerror(char *s) {
  printf("%s\n",s);
}
int main(void) {
  yyparse();
}
```

TP yacc 2

Analyse syntaxique = Construction d'arbres de syntaxe

Dans notre premier interpréteur d'expressions arithmétiques (calcV0), les traitements se faisait à la volée : les opérations étaient coordonnées par l'analyse syntaxique et le résultat des calculs étaient véhiculées par les valeurs associées aux symboles terminaux et non terminaux (avec les $).

Une deuxième approche, plus puissante, consiste à confier à l'analyse syntaxique de yacc la construction de l'arbre de syntaxe de la chaine d'entrée, et de confier les traitements à l'évaluation de celui-ci dans une 2eme phase distincte.

Analyse par construction d'un arbre (de syntaxe abstraite)

- Phase 1 : construction de l'arbre de syntaxe abstrait lors de la construction de l'arbre de dérivation (syntaxe concrète)
- Phase 2 : évaluation de l'arbre de syntaxe abstrait.

Terminologie :

On appelle *arbre de syntaxe abstrait* (cf celui de l'exercice 1) celui à la base de l'évaluation (donc du calcul de l'expression dans le cas des expressions arithmétiques), car il est indépendant de tout langage. Cet arbre de syntaxe abstrait est plus compact que le précédent et contient des informations sur la suite des actions effectuées par un programme. Chaque noeud interne de cet arbre possède une étiquette qui désigne une opération à exécuter. Il s'obtient par des transformations simples à partir de l'arbre de dérivation. Par exemple, si on utilisait des expressions postfixées (ex : 3 2 + 5 *), l'arbre de syntaxe concret serait différent, mais pas l'arbre de syntaxe abstrait.

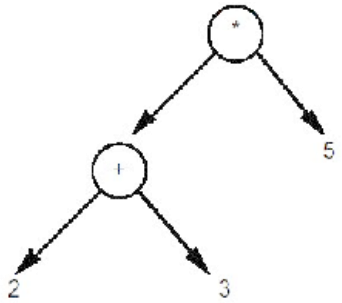
En pratique dans un programme lex/yacc:

1. LEX : on confie à lex la création des nœuds pour chaque terminal. La fonction *createNode* proposée ci-dessous peut faire l'affaire dans les parties droites des règles de lex. la pramètre de type du nœud sera déterminé en fonction du token reconnu par lex (ignorer pour l'instant la partie var). Le nœud de l'arbre qui correspond au token lui est associé via la variable yyval, (cf la directive union en conséquence).
2. YACC- règles : Dans les actions sémantiques, yacc relie les nœuds grâce à la fonction NodeChildren. On pourra se convaincre qu'une arité de 2 est suffisante moyennant quelques bricolages.

3. YACC-main :
    a. yyparse() construit l'arbre
    b. on lance l'évaluation de l'arbre résultant après le yyparse ()

Exercice 1 (algo) : Arbres binaires et calcul d'expressions numérique

1. Ecrire une structure d'arbre binaire adaptée au calcul d'expressions arithmétiques. Montrer que la structure *node* proposée ci après convient pour représenter les nœuds de l'arbre.
2. Ecrire une fonction qui affiche l'arbre sur la console (les fils devront être décalés par rapport au père)
3. Ecrire la fonction d'évaluation de l'arbre
4. Construire à la main l'arbre de l'expression 2+3*4 à l'aide des fonctions *createNode* et *NodeChildren*
5. Testez vos fonctions d'évaluation et d'affichage sur l'arbre l'expression 5*(3+2) ci-dessous :




```
typedef struct Node {
        enum NodeType type;
        union {
                double val;
                char* var;
                struct Node** children;
        } ;
} Node;


typedef union YYSTYPE
{
        struct Node *node;
} YYSTYPE;

extern YYSTYPE yylval;



Node* createNode(int type) {
        Node* newnode = (Node *) malloc(sizeof(Node));
        newnode->type = type;
        newnode->children = NULL;
        return newnode;
}
```

```
Node* nodeChildren(Node* father, Node *child1, Node *child2) {
        father->children = (Node **) malloc(sizeof(Node*) * 2);
        father->children[0] = child1;
        father->children[1] = child2;
        return father;
}
enum NodeType {
        NTNUM   = 201,
        NTVAR   = 202,
         NTPLUS  = 321,
         NTMULT  = 323
};


void printGraphRec(Node *node, int decalage){
TODO
}
double evalRec(Node *node){
TODO
}
```

Exercice 2 : Arbres de syntaxe et calcul d'expressions numérique avec yacc

1. incorporez vos fonctions dans un programme yacc et écrire les parties droites des règles appropriées.
2. comme pour la version sans arbre de calc, ajouter une mémoire et des noms de variables (on appréciera ici le dernier champ *var* de la directive union pour la structure Node

Commentaires de la correction :

input : *2+1 ;*

suite de token : NUM PLUS NUM

Dans l'analyse ascendante ci-dessous, sont notés entre parenthèse

- les attributs des token (via yylval)
- les attributs des non terminaux

Pile	Entrée	Actions d'analyse
$	NUM (NTNUM 2.0) PLUS NUM(NTNUM 1.0) $	decalage
NUM (NTNUM 2.0)	PLUS NUM $	réduction
Expr[15] (NTNUM 2.0)	PLUS NUM $	
Expr (NTNUM 2.0) PLUS NUM (NTNUM 1.0) $	$	2 décalages
Expr (NTNUM 2.0) PLUS Expr (NTNUM 1.0) $	$	réduction
Expr(NTPLUS, relié à ses 2	$	Réduction par inst → Expr

[15] ici, on voit l'intérêt du typage des non terminaux avec l'instruction %type <node> Expr

fils : (NTNUM 2.0) et (NTNUM 1.0))		
Inst(NTPLUS, relié à ses 2 fils : (NTNUM 2.0) et (NTNUM 1.0))	$	Réduction par LINST → INST
InstList(NTINTLIST avec filsGauche = NTPLUS, relié à ses 2 fils : (NTNUM 2.0) et (NTNUM 1.0) et filsDroit = NTEMPTY)	$	Réduction par line → LINST Exec : Affichage puis évaluation de l'arbre NTINTLIST avec filsGauche = NTPLUS, relié à ses 2 fils : (NTNUM 2.0) et (NTNUM 1.0) et filsDroit = NTEMPTY)
Line	$	

Affichage :

NTINTLIST
    NTPLUS
        NTNUM : 2.0
        NTNUM : 1.0
    NTEMPTY

Evaluation :

eval(NTINTLIST) appelle

evalInst(NTINTLIST) qui appelle

evalExpr(NTPLUS) qui vaut evalExpr(NTNUM : 2.0)+evalExpr(NTNUM : 1.0)=2.0+1.0=3.0

3.0 est affiché sur la console

eval(NTEMPTY) qui n'appelle rien

Exercice 3 : simulation d'output

Remplir le même tableau avec l'input suivant à partir de la correction de la gestion de l'affectation :
1+2 ;
x=4 ;

Exercice 4 : AFFICHAGE

Ajouter l'instruction d'affichage. Pour le distinguer de l'affichage de la valeur d'une expression, on pourra préciser « CONSOLE » avant l'affichage

1. Penser à l'arbre abstrait et à la fonction d'évaluation associée à l'instruction AFFICHE
2. Compléter le LEX (ajouter le schéma de détection du token PRINT)
3. Compléter le Tree.h (ajouter le NTPRINT)
4. Compléter le YACC (nouveau token, règles, actions sémantiques)

Exercice 5 : Expressions booléennes

Attention, le type booléen n'est pas explicite ici. On pourra se baser sur le type double et sur la nullité des valeurs pour simuler le type booléen (comme en C). On pourra tester ces expressions avec un AFFICHE :
Par exemple AFFICHE(2+2==5) doit afficher sur le console 0 .0

Exercice : idem avec le SiAlors, le SiAlorsSinon, et le TantQue

7. Projet mini langage

Informations pratiques :

- Le projet est à réaliser en binôme

Sujet :

L'objectif du projet est de concevoir un interpréteur pour un mini langage. Il est obligatoire de baser l'interprétation sur un arbre de syntaxe abstrait construit au cours de l'analyse syntaxique du « programme » donné en entrée.

Le projet doit vous permettre d'acquérir les compétences suivantes :

- générer des analyseurs lexicaux et syntaxiques avec lex et yacc
- construire un arbre de syntaxe abstrait à partir d'un arbre concret
- interpréter la chaine d'entrée (le programme) en évaluant l'arbre de syntaxe abstrait correspondant.

Spécifications de la version minimale (10/20) :

1. Votre interpréteur devra gérer les noms de variables à plusieurs caractères.
2. Prendre en compte le type flottant
3. Gérer les instructions suivantes :
    a. affectation
    b. affichage d'expressions numériques (pouvant contenir des variables numériques)
    c. instructions conditionnelles : implémenter le si-alors-sinon/si-alors
    d. structures itératives : implémenter le while
    e. Affichage de l'arbre de syntaxe et de la table des variables

Améliorations possibles (entre autre):

1. Gestion des erreurs (variable non initialisée, …)
2. Gestion des boucles *pour*
3. Gestion de la saisie clavier
4. Gestion du type chaine de caractères (et extension d'autant de l'instruction d'affichage)
5. Gérer les fonctions
6. Gérer la déclaration explicite des variables
7. Gérer la portée des variables et les fonctions récursives
8. Les pointeurs
9. Les tableaux

Exemple de code (la syntaxe du langage ainsi que les mots clefs peuvent être librement choisis):

```
12;
AFFICHER(12);
AFFICHER(12 + 8);
x = 0;
SI x == 0 ALORS AFFICHER(1); SINON AFFICHER(0); FIN;
fm1 = 1;
fm2 = 1;
ftmp = 0;
TANTQUE x < 20 FAIRE x = x + 1; ftmp = fm1 + fm2; fm2 = fm1; fm1 = ftmp; AFFICHER(ftmp);
FIN;
```

Exemple de sortie :

```
12.000000
12.000000
20.000000
x <- 0.000000
1.000000
 - Fait -
fm1 <- 1.000000
fm2 <- 1.000000
tmp <- 0.000000
2.000000
3.000000
5.000000
8.000000
13.000000
21.000000
34.000000
55.000000
89.000000
144.000000
233.000000
377.000000
610.000000
987.000000
1597.000000
2584.000000
4181.000000
6765.000000
10946.000000
17711.000000
 - Fait –
```