Principe d'Interprétation des Langages

Christine Paulin
Christine.Paulin@universite-paris-saclay.fr

Département Informatique, Faculté des Sciences d'Orsay, Université Paris-Saclay

Licence Informatique - 2ème année

2022–23

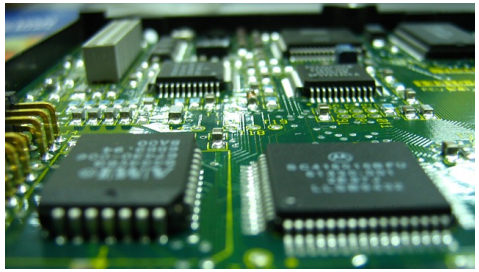
Introduction à l'interprétation des langages

Interprétation des langages

S'il vous plait, dessine-moi un mouton!

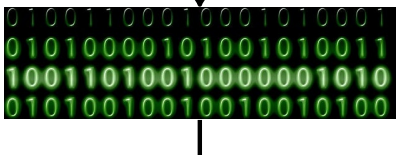
modélisation algorithmique

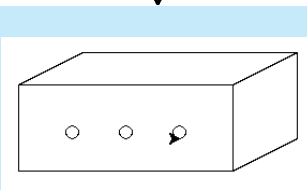
données
calcul



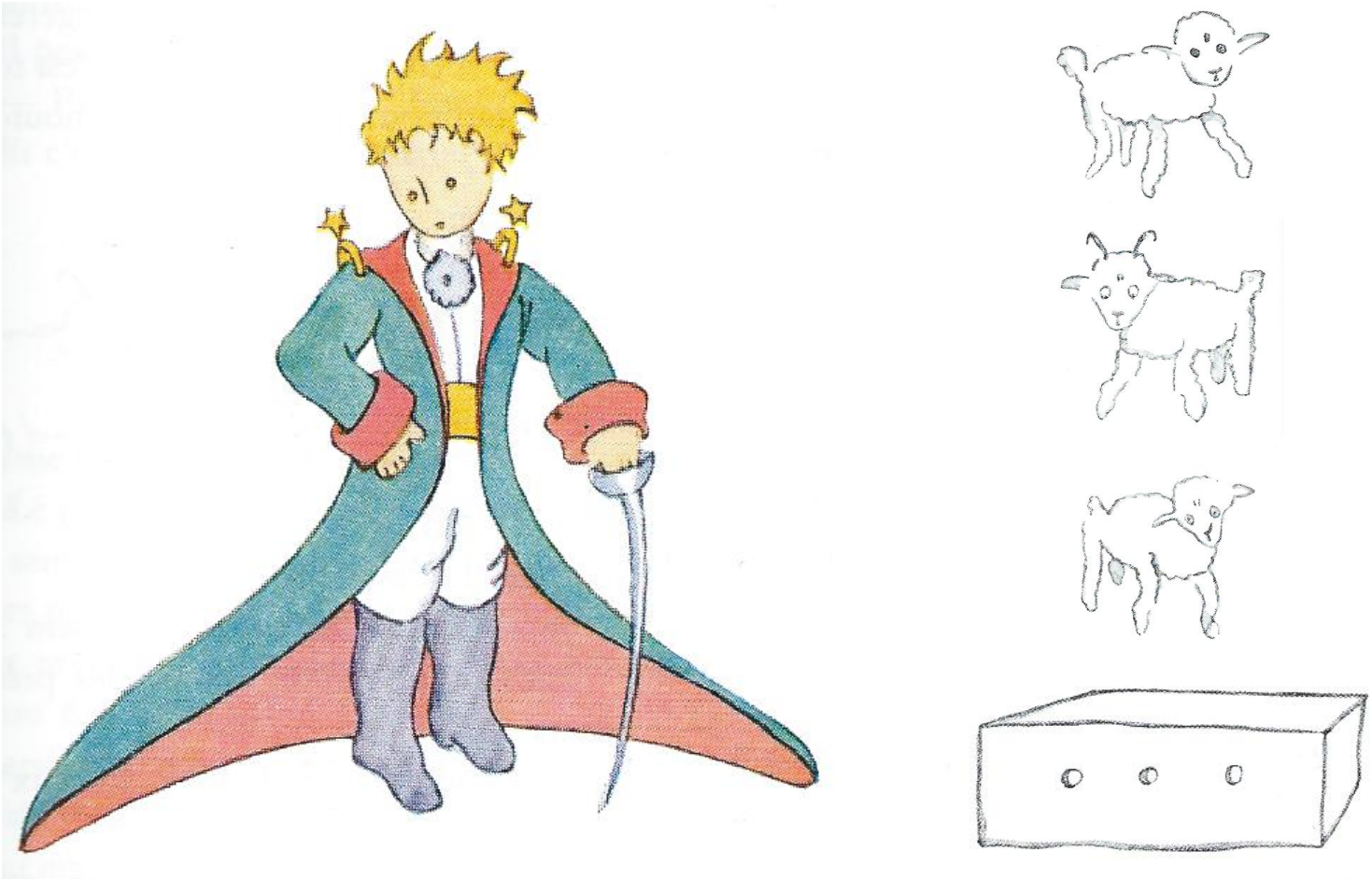
```
 1  from turtle import *
 2  def trou (x,y,r):
 3      up()
 4      goto(x,y)
 5      down()
 6      circle(r)
 7  def mouton (h,L,l,a,t) :
 8      goto(0,h)
 9      down()
10      goto(0,0)
11      goto(L,0)
12      goto(L,h)
13      goto(0,h)
14      goto(l,h+a)
15      goto(L+l,h+a)
16      goto(L,h)
17      goto(L,0)
18      goto(L+l,a)
19      goto(L+l,h+a)
20      trou(L//4,t,6)
21      trou(2*L//4,t,6)
22      trou(3*L//4,t,6)
23      done()
24  mouton(80,200,60,30,30)
```

```
18          108 LOAD_GLOBAL              0 (goto)
            110 LOAD_FAST                1 (L)
            112 LOAD_FAST                2 (l)
            114 BINARY_ADD
            116 LOAD_FAST                3 (a)
            118 CALL_FUNCTION            2
            120 POP_TOP

19          122 LOAD_GLOBAL              0 (goto)
            124 LOAD_FAST                1 (L)
            126 LOAD_FAST                2 (l)
            128 BINARY_ADD
            130 LOAD_FAST                0 (h)
            132 LOAD_FAST                3 (a)
            134 BINARY_ADD
            136 CALL_FUNCTION            2
            138 POP_TOP

20          140 LOAD_GLOBAL              2 (trou)
            142 LOAD_FAST                1 (L)
            144 LOAD_CONST               2 (4)
            146 BINARY_FLOOR_DIVIDE
            148 LOAD_FAST                4 (t)
            150 LOAD_CONST               3 (6)
            152 CALL_FUNCTION            3
            154 POP_TOP

21          156 LOAD_GLOBAL              2 (trou)
            158 LOAD_CONST               4 (2)
            160 LOAD_FAST                1 (L)
            162 BINARY_MULTIPLY
            164 LOAD_CONST               2 (4)
            166 BINARY_FLOOR_DIVIDE
            168 LOAD_FAST                4 (t)
            170 LOAD_CONST               3 (6)
            172 CALL_FUNCTION            3
            174 POP_TOP
```





Référence



Langages de programmation

- Intermédiaires entre l'humain et la machine
  - exprimer (le plus fidèlement possible) ce que l'humain "souhaite"
  - être "compris" de manière mécanisée par la machine
- Faciliter la programmation
  - détecter des erreurs
  - organiser et factoriser les traitements
  - réutiliser du code (bibliothèques)
  - effectuer la maintenance et les évolutions
- Faciliter le déploiement sur des architectures variées

Du code source au résultat de l'exécution

- Transformations successives du code source
- Plusieurs langages intermédiaires
- Préservation du "sens" du programme (sémantique)
- Chaque étape réalise une tâche particulière
- Compilateur/Interpréteur : des programmes qui manipulent d'autres programmes
  - Compilateur : le compilateur transforme le programme source en un code binaire que la machine exécute sur différentes entrées
  - Interpréteur : l'interpréteur est un programme général qui s'exécute sur la machine, il prend en argument le programme et l'entrée et calcule le résultat
  - Méthodes mixtes (Ocaml, java, python. . .) : le compilateur engendre un code intermédiaire qui est interprété (machine virtuelle)

Problématique

- Nombreux problèmes d'efficacité :
  - efficacité des transformations
  - efficacité de l'exécution
- Résultats théoriques
  - langages formels (reconnaissance)
  - sémantique des langages de programmation (typage, modèle d'exécution)
  - optimisation de programmes (allocation de la mémoire)

1–Introduction

Objectifs du cours

- Etre de meilleurs programmeurs :
  - mieux comprendre le point de vue de l'ordinateur
  - mieux appréhender différents langages de programmation
- S'initier à la théorie des langages formels
  (langages réguliers, langages algébriques)
- Connaître de nouveaux paradigmes de programmation : expressions régulières, grammaires
- Comprendre les rudiments de la sémantique des langages de programmation

Orientation un peu différente par rapport aux années précédentes
+langage de programmation -langage formel

Eléments pratiques

Site

https://ecampus.paris-saclay.fr/course/view.php?id=97138

TD/TP

- 8 semaines de TD, 3 semaines de TP (semaines 4, 5, 9)
- dernière séance TP noté (a priori le 25 avril)

Evaluation

- partiel (semaine du 6 mars) : 30%
- TP noté (25 avril) : 10%
- examen final (entre le 9 et le 19 mai) : 60%

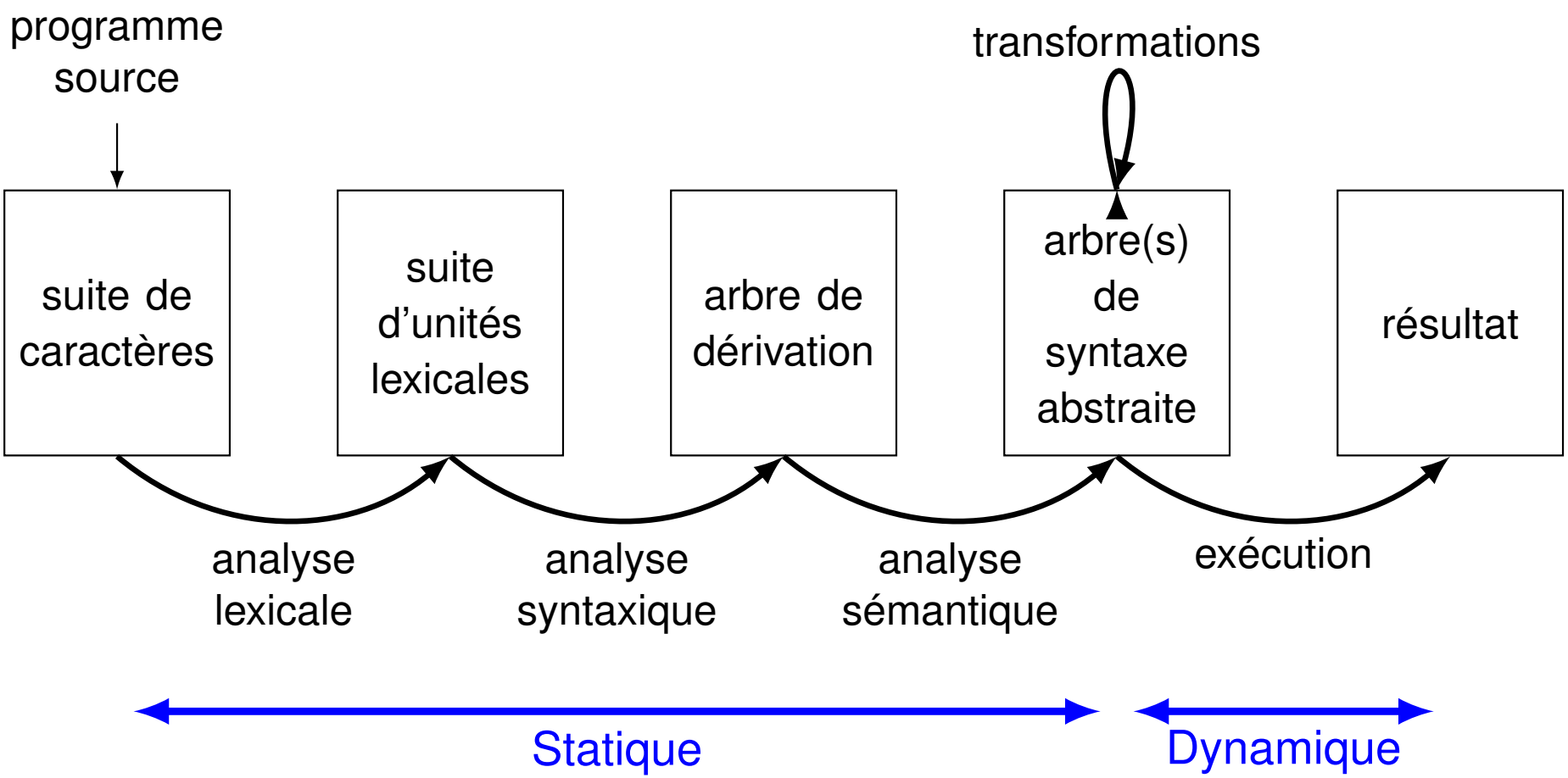
Une feuille A4 (recto-verso) manuscrite autorisée au partiel et à l'examen

Plan du cours

1. Introduction à l'interprétation des langages
2. Analyse lexicale : expressions régulières, automates finis, langages réguliers
3. Analyse syntaxique : grammaires et langages algébriques
4. Exécution : sémantiques statique et opérationnelle

1–Introduction

Etapes


programme source
transformations
suite de caractères
suite d'unités lexicales
arbre de dérivation
arbre(s) de syntaxe abstraite
résultat
analyse lexicale
analyse syntaxique
analyse sémantique
exécution
Statique
Dynamique

Exemple : mini-python (source J.-C. Filliâtre)

Un fichier mini-Python a la structure suivante :

```
# définition(s) de fonctions
def fibaux(a, b, k):
    if k == 0:
        return a
    else:
        return fibaux(b, a+b, k-1)

def fib(n):
    return fibaux(0, 1, n)

# une ou plusieurs instructions
print("quelques valeurs :")
for n in [0, 1, 11, 42]:
    print(fib(n))
```

Spécification mini-python

Un fichier mini-Python est composé d'une liste optionnelle de *déclarations de fonctions*, suivie d'une liste d'*instructions*.

- Les instructions sont :
  - l'affectation simple ($x = e$) ou dans un tableau $t[e_1] = e_2$,
  - la conditionnelle (**if**/**else**),
  - la boucle (**for**/**in**),
  - l'affichage d'une expression avec **print**,
  - le renvoi d'une valeur avec **return**
  - ou l'évaluation d'une expression.
- Les expressions sont :
  - une constante (booléen, entier ou chaîne de caractères),
  - l'accès à une variable,
  - la construction d'une liste (notation $[e_1, e_2, \ldots, e_n]$),
  - l'accès à un élément de liste (notation $e[i]$),
  - l'appel d'une fonction (notation $f(e_1, \ldots, e_n)$),
  - une des opérations arithmétiques `+`, `-`, `*` ou `//`, un opérateur de comparaison `==`, `!=`, `<`, `<=`, `>`, `>=` ou un opérateur booléen **and**, **or** et **not** associé à un ou deux expressions (notations `unop` e et e_1 `binop` e_2).

Spécification de langages de programmation

On pourra regarder quelques sites donnant les spécifications complètes de langages de programmation.
On remarquera la complexité d'une définition précise même des objets les plus simples (identifiants, chaînes de caractères etc...)

- Java :
  - lexical : `https://docs.oracle.com/javase/specs/jls/se19/html/jls-3.html`
  - expression :
    `https://docs.oracle.com/javase/specs/jls/se19/html/jls-15.html`
- Python :
  - lexical : `https://docs.python.org/fr/3/reference/lexical_analysis.html`
  - expressions : `https://docs.python.org/fr/3/reference/expressions.html`
- Ocaml :
  - lexical : `https://v2.ocaml.org/releases/5.0/htmlman/lex.html`
  - expressions : `https://v2.ocaml.org/releases/5.0/htmlman/expr.html`

Analyse lexicale

Rôle

- Transformer le programme source vu comme une suite de caractères, en une suite d'objets (appelés unités lexicales, ou lexèmes ou token) significatifs pour le calcul :
  - entités numériques ou textuelles : entiers, flottants, chaînes de caractères (`hello` versus `"hello"`)
  - identifiants
  - mots clés (suite de caractères participant à la structure du programme, non utilisables comme identifiant)
  - symboles spéciaux (opérations numériques, crochets, séparateurs de liste...)
- Agréger dans une même catégorie les éléments de même nature d'un point de vue syntaxique (identifiant, nombre) en gardant la valeur en réserve pour le reste du calcul.
- Supprimer les éléments inutiles au calcul
  - espaces non significatifs (mais servent à séparer les unités lexicales `+=` versus `+ =`)
  - commentaires

Analyse lexicale

Erreurs détectées

- suite de caractères ne correspondant à aucune unité lexicale :
  - caractère non accepté dans un identifiant,
  - chaîne de caractère non fermée,
  - symbole inconnu. . .

Outils

- Les unités lexicales sont décrites par des expressions régulières
- L'analyse du texte se fait en utilisant des automates finis (associés à des actions de calcul à chaque étape de reconnaissance)

Analyse lexicale : exemple mini-python

- Unités lexicales (en nombre fini, possiblement associées à des valeurs):
  - commentaire : commence par le synbole `#` et se termine au retour à la ligne
  - mots-clés : **def if else return print for in and or not**
  - opérations arithmétiques : `PLUS MINUS TIMES DIV MOD`
  - comparaisons : `CMP` (une seule unité, associée à l'opérateur de comparaison)
  - symboles associés à l'indentation `BEGIN END NL EOF`
  - autres symboles spéciaux : `'(' ')' '[' ']' ',' '=' ':'`
  - constantes : `CST` (associé à une valeur entière, flottante ou chaîne de caractères)
  - identifiants : `ID` associé à une représentation de l'identifiant (chaîne)

Résultat

```
# définitions de fonctions
def fibaux(a, b, k):
    if k == 0:
        return a
    else:
        return fibaux(b, a+b, k-1)

def fib(n):
    return fibaux(0, 1, n)

# une ou plusieurs instructions
print("quelques valeurs :")
for n in [0, 1, 11, 42]:
    print(fib(n))
```

```
NL
def ID '(' ID ',' ID ',' ID ')' ':' NL
BEGIN if ID CMP CST ':' NL
BEGIN return ID NL
END else ':' NL
BEGIN return ID '(' ID ',' ID PLUS ID ',' ID MINUS CST ')' NL

END END def ID '(' ID ')' ':' NL
BEGIN return ID '(' CST ',' CST ',' ID ')' NL


END PRINT '(' CST ')' NL
for ID in '[' CST ',' CST ',' CST ',' CST ']' ':' NL
BEGIN PRINT '(' ID '(' ID ')' ')' NL
END NL EOF
```

Analyse syntaxique

Rôle

- Analogie : l'analyse lexicale a transformé la suite de *caractères* en une suite de *mots*, l'analyse syntaxique construit à partir de la suite de mots, des *phrases* auxquelles il va être possible de donner un sens.
- Expliciter la structure du programme
  - expressions à calculer
  - enchaînement des instructions (conditionnelles, boucles)
  - déclarations et utilisation de fonctions
- Lever les ambigüités (choix)
- Transformer la suite de lexèmes en un arbre qui reflète la structure logique du programme

Erreurs détectées

- suite d'unités lexicales ne correspondant pas à une construction de programme correcte

Outils

- La structure du programme est décrite par une grammaire
- L'analyse du texte se fait en utilisant des *automates à pile* ou des fonctions d'analyse ad-hoc

Analyse syntaxique : exemple

Expliciter la structure d'un programme correct

- zéro, une ou plusieurs définitions de fonctions
  - une définition de fonction est formée ainsi (en séquence)
    - mot-clé **def**,
    - nom de la fonction (identifiant),
    - parenthèse ouvrante `'('`,
    - suite de zéro, un ou plusieurs paramètres (identifiants) séparés par des virgules,
    - parenthèse fermante `')'`,
    - symbole `':'`,
    - une "*suite d'instructions*" qui va être soit une instruction élémentaire suivie d'un retour à la ligne (`NL`) soit un retour à la ligne suivi d'un bloc d'instructions indenté, c'est-à-dire entre un `BEGIN` et un `END`.
- au moins une instruction

Analyse syntaxique : instruction et expression

- instruction élémentaire
  - instructions **print**/**return**
  - affectation (variable ou tableau)
  - expression
- instruction générale
  - instruction élémentaire suivie d'un retour à la ligne
  - conditionnelle (avec ou sans else) :
    - mot clé **if** suivi d'une expression, suivi du symbole `':'` suivi d'une *suite d'instructions*
    - suivi optionnellement du mot clé **else** suivi du symbole `':'` suivi d'une *suite d'instructions*
  - boucle for : mot clé **for** suivi d'un identifiant, suivi du mot clé **in** suivi d'une expression suivi du symbole `':'` suivi d'une *suite d'instructions*
- expressions
  - constantes, identifiants, accès tableau
  - opérateur unaire suivi d'une expression
  - opérateur binaire entre deux expressions
  - appel de fonction
  - liste d'expressions (entre crochets carrés)
  - expression parenthésée

Grammaire associée

- Reconnaître qu'une suite de lexèmes correspond à un programme bien formé
- Introduire des entités intermédiaires (symboles) pour décrire différentes catégories syntaxiques
  - entités *principales* : *prog*, *deffun*, *instr*, *expr*,
  - entités *auxiliaires* :
    - *unop*, *binop* : opérateurs unaires (**not**, `MINUS`) et opérateurs binaires
    - *linstr* : liste d'instructions (juxtaposition)
    - *simpleinstr* : instruction élémentaire
    - *suiteinstr* : instruction élémentaire suivie d'un retour à la ligne ou suite d'instructions dans un bloc
    - *lident* : liste possiblement vide d'identifiants séparés par des virgules
    - *lexpr* : liste possiblement vide d'expressions séparées par des virgules

Grammaire associée

- Décrire par des *règles* (récursives) comment les entités sont construites à l'aide des lexèmes
  - *deffun* ≔ **def** `ID '(' lident ')' ':'` *suiteinstr*
  - Il y a en général plusieurs règles pour la même entité
    - *expr* ≔ `ID`
    - *expr* ≔ `CST`
    - *expr* ≔ *unop expr*
    - *expr* ≔ *expr binop expr*
    - *expr* ≔ `'(' expr ')'`
    - *expr* ≔ `ID '(' lexpr ')'`
    - *expr* ≔ *expr* `'['` *expr* `']'`
    - *expr* ≔ `'['` *lexpr* `']'`
    - ...

  On utilise une notation générale pour couvrir l'ensemble des cas en les séparant par des barres verticales

expr ≔ `ID` | `CST` | *unop expr* | *expr binop expr* | `'(' expr ')'`
| `ID '(' lexpr ')'` | *expr* `'['` *expr* `']'` | `'['` *lexpr* `']'`

Résultat

La suite de lexèmes est acceptée comme une entité valide si on peut construire un arbre de dérivation dont la racine est le nom de l'entité, dont chaque nœud interne correspond à une règle et dont les feuilles lues de gauche à droite correspondent à la suite de lexèmes.

On considère juste une instruction et la suite d'unités lexicales associées

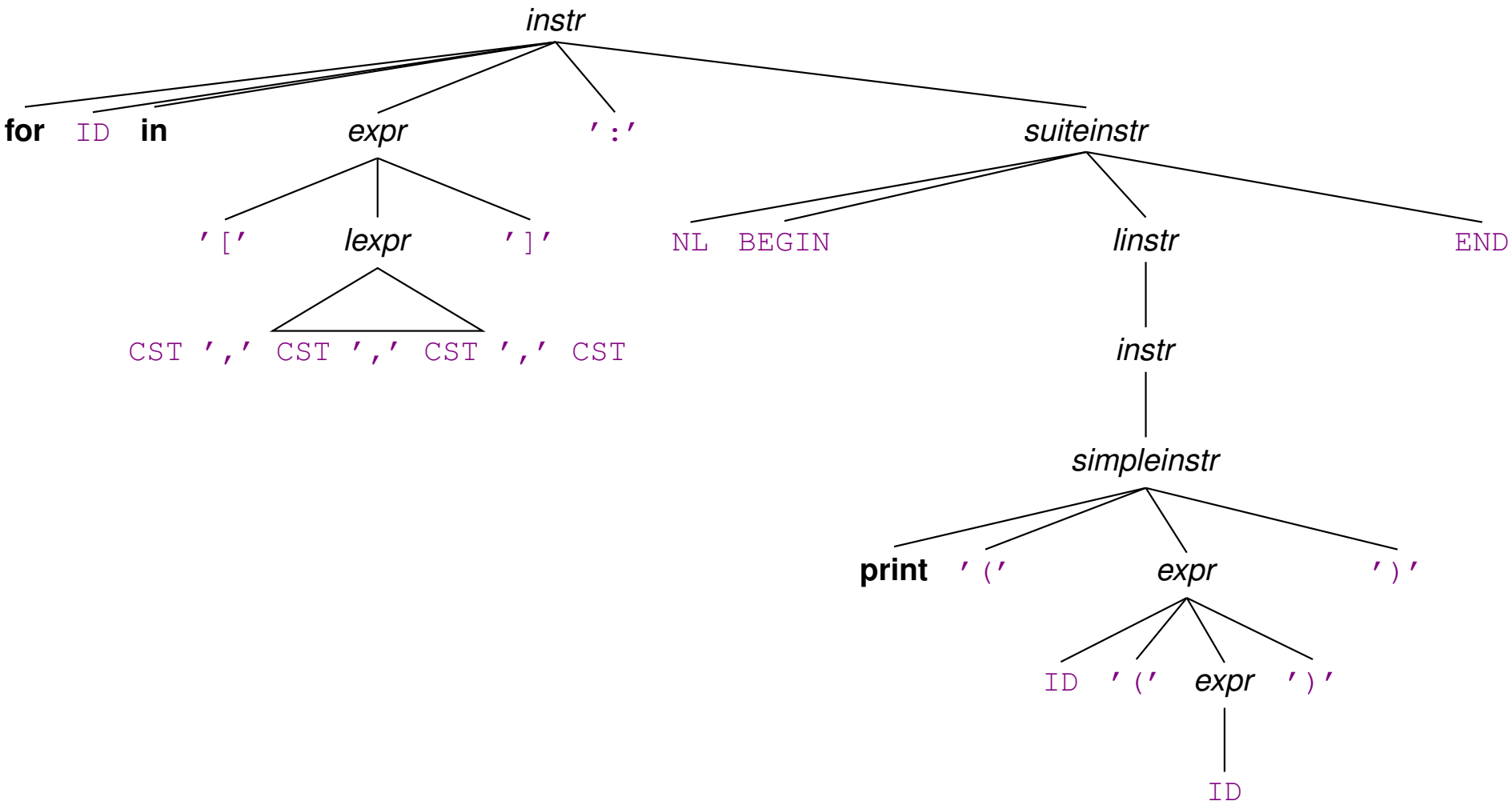
```
for n in [0, 1, 11, 42]:
    print(fib(n))
```

for ID **in** '[' CST ',' CST ',' CST ',' CST ']' ':' NL
BEGIN **print** '(' ID '(' ID ')' ')' NL END

Les règles supplémentaires utiles concernant les instructions sont

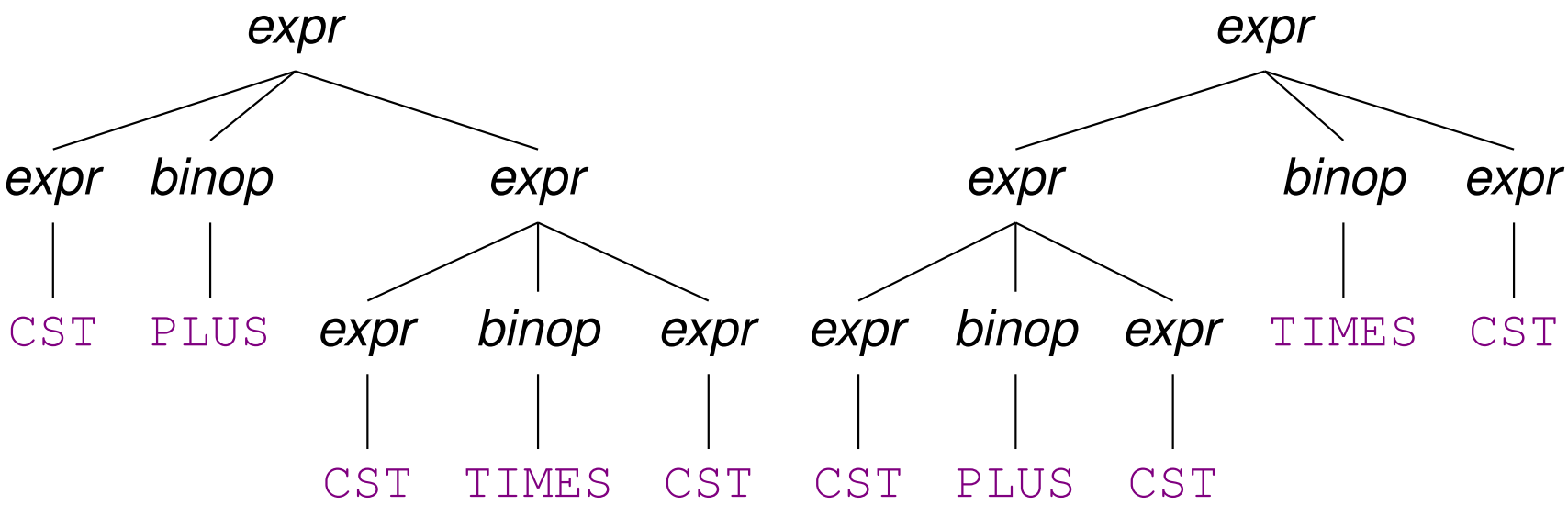
- *instr* ≔ **for** `ID` **in** *expr* `':'` *suiteinstr*
- *suiteinstr* ≔ *simplinstr* `NL` | `NL BEGIN` *linstr* `END`
- *simplinstr* ≔ **print** `'('` *expr* `')'`

Arbre de dérivation



Ambigüité syntaxique

- Certaines suites de mots peuvent s'interpréter syntaxiquement de plusieurs manières
- Le choix de l'interprétation peut avoir une incidence sur le calcul
- Exemple : `CST PLUS CST TIMES CST` conduit à deux arbres de dérivation différents :



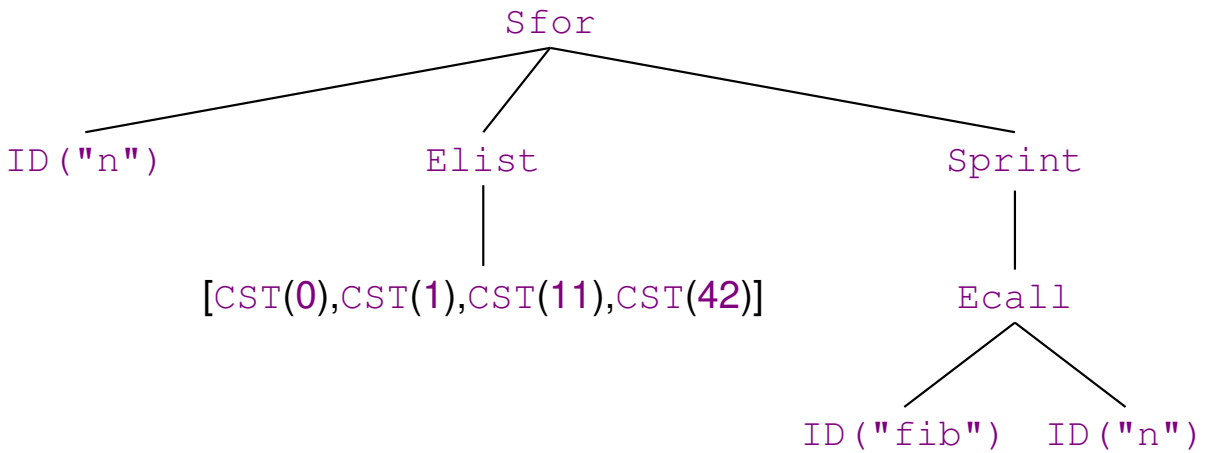
- Le langage contraint l'interprétation
- Modification de la grammaire ou indication de précédence (priorité entre les opérateurs)

Questions sur l'analyse syntaxique et les choix des langages de programmation

1. Que se passerait-il si on autorisait un caractère comme `'-'` ou encore un espace à apparaître dans un identificateur ?
2. Pourquoi les "mots" qui apparaissent dans un programme ne peuvent-ils pas faire partie d'un dictionnaire comme celui de la langue française ?
3. Comment l'analyseur interprète-t-il la suite de caractères `ifelse` ? Pourquoi ?
4. On dit souvent que les *espaces* sont *non significatifs*, qu'est-ce que cela signifie et quel est leur rôle ? Est-ce vrai dans tous les langages ?
5. Pourquoi traiter les mots-clés différemment des autres identifiants ?
6. On a une seule unité lexicale pour les symboles de comparaison mais des unités différentes pour les symboles arithmétiques, qu'est-ce qui peut expliquer cette différence de traitement ?

Arbre de syntaxe abstraite

- L'arbre de dérivation contient des informations sur la manière dont le programme a été analysé syntaxiquement
  - justifie que le programme répond à la spécification syntaxique du langage
  - informations pas toujours pertinentes pour le calcul.
- En pratique on se sert de la dérivation pour reconstruire un arbre plus *abstrait* qui se concentre sur la structure logique du programme
  - Chaque noeud de l'arbre correspond à une construction du programme
  - Utilisation de listes *natives* (ou autre construction) du langage de modélisation
  - Intégration des valeurs des unités lexicales (identifiants, constantes, opérateurs. . .)
- Exemple



Analyse sémantique

- Analyse de portée : faire le lien entre l'utilisation d'un identifiant et sa "définition"
  - déclaration de variable, paramètre
  - appel à des fonctions externes
- Repérer des erreurs liées au typage (opérations appliquées à des objets de nature incompatible)
- Désambigüer des opérations en tenant compte du type des arguments (surcharge)

Cette étape achève la partie d'analyse du programme, son sens est explicité, on a rejeté tous les programmes "mal écrits".
On démarre alors une partie synthèse qui transforme le programme en vue de calculer efficacement le résultat.

Transformations et exécution

- Les compilateurs passent par différents langages intermédiaires pour aboutir à du code assembleur et du code binaire
  - un enjeu important est le positionnement des données et valeurs intermédiaires dans la mémoire (registres, pile, tas. . .)
  - Il y a une phase d'*édition de liens* pour associer différents codes compilés
  - Le programme final s'exécute dans un environnement qui peut contenir d'autres processus comme le *Garbage collector* (*Ramasse-miettes* ou *Glaneur de Cellules* (GC) en français) qui se charge de libérer la mémoire non utilisée
- On peut transformer le code vers un code de *machine virtuelle*, souvent une machine à pile, qui peut ensuite être interprété simplement.
- L'interpréteur est un programme (écrit dans le langage de notre choix) qui prend en argument la syntaxe abstraite du programme et de son entrée, qui analyse la structure du programme et de l'entrée et calcule le résultat attendu en respectant sur la sémantique du programme.

A savoir

- Les différentes étapes de l'interprétation d'un langage
- Le positionnement des erreurs en fonction des langages par rapport à ses étapes
- Comprendre la description d'une unité lexicale et celle d'une règle de syntaxe (comme on les trouve dans les manuels de description des langages)

1–Introduction

Introduction

- La première étape de l'interprétation du langage est de traduire la suite de caractères en un arbre de syntaxe abstraite qui va correspondre à un calcul exécutable par un ordinateur
- Pour que ce que veut l'utilisateur et ce qu'exécute l'ordinateur corresponde, il faut des règles claires de construction du programme
- Il faut également que cette étape d'analyse se fasse rapidement (algorithmes efficaces)
- La théorie des langages formels est une abstraction de cette problématique
  - description synthétique des langages
  - algorithmes pour reconnaître qu'un mot donné appartient bien au langage considéré

Vocabulaire

- Alphabet : ensemble (fini) des caractères possibles
  - analyse lexicale : l'alphabet est l'ensemble des caractères usuels (ascii, unicode...)
  - analyse syntaxique : l'alphabet est l'ensemble des unités lexicales
- Mot : suite finie de caractères de l'alphabet (notée par juxtaposition des caractères)
  - analyse lexicale : la suite de caractères correspondant à l'unité lexicale
  - analyse syntaxique : la suite d'unités lexicales correspondant à la construction de programme
- Le nombre de caractères dans un mot s'appelle la longueur du mot
- Il existe un mot sans aucun caractère (longueur 0), appelé mot vide et noté ϵ (epsilon).
- Tout caractère peut être vu comme un mot de longueur 1
  - Le langage Ocaml différencie le caractère `'a'` du mot formé d'un seul caractère "*a*" mais ce n'est pas le cas de python.

Langage

- Un langage est défini comme n'importe quel ensemble de mots.
- Un langage peut être fini ou infini
- Un langage est un ensemble dénombrable : union dénombrable d'ensembles finis (les mots de longueur n)
- En interprétation des langages :
  - analyse lexicale : chaque unité lexicale correspond à un langage (ensemble des identifiants, ensemble des constantes numériques, ...)
  - analyse syntaxique : l'ensemble des programmes syntaxiquement bien formés est vu comme un langage dont l'alphabet est formé des différentes unités lexicales (mots).

Exemples

- On prend un alphabet avec deux caractères $\{a, b\}$.
- a, b, aa, ab, bb, ba, $bbbabbbabb$ sont des mots sur cet alphabet
- $\{aa, ab, bb, ba\}$ est un langage (fini) que l'on peut caractériser comme l'ensemble des mots de longueur 2 sur cet alphabet.
- L'ensemble des mots de longueur inférieure ou égale à 2 sur cet alphabet se compose de 7 mots $\{\epsilon, a, b, aa, ab, bb, ba\}$.
- L'ensemble des mots ne contenant que le caractère a est un langage (infini) : $\{\epsilon, a, aa, aaa, \ldots\}$.
- L'ensemble des mots qui contient autant de a que de b est un langage (infini).
- L'ensemble vide ($\emptyset$) est un langage qui ne contient aucun mot, il se distingue de l'ensemble $\{\epsilon\}$ qui contient exactement un mot (le mot vide)

Représentation des mots

Soit A un alphabet (ensemble fini) dont les éléments sont des caractères

- un mot est une *suite finie* de caractères
  - un mot peut se modéliser par un entier naturel $n \in \mathbb{N}$ qui est la longueur du mot et une application de $[1 \ldots n]$ dans A qui associe à un entier k tel que $1 \leq k \leq n$ la k-ème lettre du mot
  - d'un point de vue informatique, un mot est un tableau de caractères (mais indicé à partir de 1)
  - si w est un mot, on note $|w|$ la longueur du mot et pour k tel que $1 \leq k \leq |w|$, on note $w[k]$ la k-ème lettre du mot w.
    - le mot $aabb$ est de longueur 4, on a $aabb[1] = aabb[2] = a$ et $aabb[3] = aabb[4] = b$
  - si on se donne n caractères $a_1, \ldots, a_n \in A$ alors on peut former un mot de longueur n tel que la k-ème lettre de ce mot est a_k. On note cette suite comme une juxtaposition des caractères qui la composent : $a_1 \ldots a_n$.
    On utilisera si nécessaire des espaces pour séparer les caractères entre eux.

Opérations sur les mots

- sous-mot : w est un sous-mot de m si w peut s'obtenir en effaçant des lettres de m.
  On a donc $|w| \leq |m|$ et il existe une application f strictement croissante de $[1 \ldots |w|]$ dans $[1 \ldots |m|]$ telle que $|w|[k] = |m|[f(k)]$ pour tout $k \in [1 \ldots |w|]$, on a donc $w = m[f(1)]m[f(2)] \ldots m[f(|w|)]$
  - Soit le mot *abab*, les mots ϵ, *a*, *b*, *aa*, *ab*, *bb*, *ba*, *aab*, *abb*, *aba*, *bab*, *abab* sont des sous-mots de *abab*. Par contre les mots *bba*, *aabb* ne sont pas des sous-mots.
- miroir : l'opération miroir consiste à réarranger les caractères du mot de droite à gauche. Le mot miroir de w est un mot de même longueur, il est noté w^R et pour tout $k \in [1 \ldots |w|]$, on a $w^R[k] = w[|w| - k + 1]$
  - le mot miroir de *abab* est le mot *baba*
  - appliquer deux fois l'opération miroir revient au mot d'origine : le mot miroir du mot miroir de w est le mot w
  - un mot qui est égal à son mot miroir s'appele un *palindrome*

Opérations sur les mots : concaténation

Les mots se mettent bout à bout pour former de nouveaux mots.

- concaténation : Si w_1 et w_2 sont deux mots, on forme un nouveau mot correspondant à la concaténation de w_1 et w_2 et noté par simple juxtaposition $w_1 w_2$. On a $|w_1 w_2| = |w_1| + |w_2|$ et pour tout $k \in [1 \dots |w_1| + |w_2|]$ on a $(w_1 w_2)[k] = w_1[k]$ si $k \leq |w_1|$ et $(w_1 w_2)[k] = w_2[k - |w_1|]$ si $|w_1| < k \leq |w_2|$.
  - La concaténation des mots *ab* et *abba* est le mot *ababba*.
- l'absence de signe pour noter la concaténation peut se rapprocher de l'usage de ne pas noter la multiplication ($2x + yz$).
  Dans les langages de programmation, la concaténation de chaînes de caractères utilise un symbole spécifique ($+$ en python ou Java, $\hat{}$ en Ocaml. . .)
- l'opération de concaténation est *associative* $(w_1 w_2)w_3 = w_1(w_2 w_3)$, son *élément neutre* est le mot vide $w\epsilon = \epsilon w = w$,
  elle n'est pas *commutative* ($ab \neq ba$)

Opérations sur les mots (préfixe, suffixe)

- préfixe : le mot u est un préfixe du mot w si et seulement si il existe un mot v tel que $w = uv$ (le mot u est un début de w)
  - le mot ab est un préfixe du mot $abba$ mais n'est pas un préfixe du mot $aabb$
  - le mot vide ϵ est un préfixe de n'importe quel mot
- suffixe : le mot u est un suffixe du mot w si et seulement si il existe un mot v tel que $w = vu$ (le mot u est une fin de w)
  - le mot ba est un suffixe du mot $abba$ mais n'est pas un préfixe du mot $aabb$
  - le mot vide ϵ est un suffixe de n'importe quel mot

Opérations sur les mots (puissance)

Étant donné un mot w, on peut itérer l'opération de concaténation un nombre quelconque de fois, cela amène à définir une opération de puissance qui étant donné un entier $n \in \mathbb{N}$ renvoie un mot qui correspond à n copies du mot w qui est noté w^n.

- la puissance 0 d'un mot est définie comme le mot vide
- le mot w à la puissance $n+1$ est défini récursivement comme la concaténation du mot w à la puissance n et du mot w.

On a donc

$$w^0 = \epsilon \qquad\qquad w^{n+1} = w^n w$$

On vérifie que la puissance 1 d'un mot est le mot lui-même : $w^1 = w$

Exemples

$$(ba)^2 = baba \qquad\qquad (ba)^3 = bababa$$

Opérations sur les langages : opérations ensemblistes

Les langages sont des ensembles de mots finis ou infinis qui peuvent se combiner pour former de nouveaux langages.
Les constructions usuelles d'ensembles s'appliquent aux langages.

- union : $w \in (L_1 \cup L_2)$ ssi $w \in L_1$ ou $w \in L_2$
- intersection : $w \in (L_1 \cap L_2)$ ssi $w \in L_1$ et $w \in L_2$
- différence : $w \in (L_1 \setminus L_2)$ ssi $w \in L_1$ et $w \notin L_2$
- complémentaire : $w \in \complement L_1$ ssi $w \notin L_1$

Exemples sur l'alphabet $\{a, b\}$

- L'union de l'ensemble des mots qui commencent par un a et de l'ensemble des mots qui commencent par un b est l'ensemble des mots non vides.
- L'intersection de l'ensemble des mots qui commencent par un a et de l'ensemble des mots qui commencent par un b est l'ensemble vide.
- Le complémentaire de l'ensemble des mots qui commencent par un a est formé du mot vide et de l'ensemble des mots qui commencent par b.

Opérations sur les langages : concaténation

La concaténation de deux langages L_1 et L_2 s'obtient en prenant tous les mots formés par concaténation d'un mot $w_1 \in L_1$ et d'un mot $w_2 \in L_2$.

$$w \in (L_1 L_2) \quad \text{ssi} \quad \text{il existe } w_1 \in L_1 \text{ et } w_2 \in L_2 \text{ telsque } w = w_1 w_2$$

- Si L_1 est l'ensemble des mots qui commencent par un a et L_2 l'ensemble des mots qui commencent par un b alors $L_1 L_2$ est l'ensemble de tous les mots qui commencent par un a et contiennent au moins un b.

Opérations sur les langages : puissance

La puissance n-ème d'un langage L s'obtient en itérant n fois l'opération de concaténation.

$$L^0 = \{\epsilon\} \qquad L^{n+1} = L^n L$$

- La puissance 0 d'un langage contient exactement un mot à savoir le mot vide.
- $L^1 = L$.
- $w \in L^n$ ssi il existe $w_1 w_2 \dots w_n \in L$ tels que $w = w_1 \dots w_n$.
- Si $w \in L$ alors $w^n \in L^n$ mais il y a en général beaucoup plus de mots dans L^n que les puissances des mots de L.
  Par exemple si $L = \{a, b\}$ (langage formé de deux mots, chacun réduit à un caractère) alors L^n contient tous les mots de longueur n et pas juste a^n et b^n.

Etoile de Kleene

L'étoile de Kleene d'un langage L consiste à considérer l'union de toutes les puissances possibles du langage.

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

- $\epsilon \in L^*$
- $L \subseteq L^*$
- Si le langage L n'est pas vide ou réduit au mot vide, alors le langage L^* contient des mots de longueur arbitrairement grande et est donc infini.
- Soit A l'alphabet. A peut se voir comme un langage (ensemble des mots formés d'un seul caractère).
  On montre facilement que A^n est l'ensemble des mots sur l'alphabet A de longueur n et A^* est donc l'*ensemble de tous les mots sur l'alphabet* A.

Problème de la reconnaissance

Etant donné un langage L peut-on construire un *algorithme* qui étant donné un mot w répond vrai ou faux suivant si $w \in L$?

- Si L est un langage fini donné *en extension* par une énumération des mots qui le composent ($L = \{w_1, \ldots, w_n\}$) alors répondre à la question $w \in L$ se fait (naïvement) en testant si $w = w_1$ et sinon si $w = w_2$ etc
- Les langages infinis sont décrits de manière *intensionnelle* par exemple par une propriété logique ($L = \{w \in A^* \mid P(w)\}$).
  Exemple : mots de longueur paire, mots qui contiennent la lettre a...
  Tester si $w \in L$ se ramène à tester si la propriété $P(w)$ est vérifiée.
- Certaines propriétés ne peuvent être testées par aucun algorithme (on dit qu'elles sont indécidables).
  C'est le cas de la question de savoir si un programme quelconque *termine*. Si on considère le langage T qui contient tous les programmes qui terminent alors il n'y a pas d'algorithme qui teste si un mot $w \in T$.

Reconnaissance effective

- On va donc caractériser certaines classes de langages pour lesquels on va non seulement avoir la décidabilité de l'appartenance mais également des *algorithmes efficaces* pour tester cette appartenance.
- Cela va être le cas des langages réguliers dont la reconnaissance se fait avec des automates finis déterministes (en temps linéaire sur la longueur du mot)
- Les langages réguliers étant trop restreints (ils ne permettent pas par exemple de tester si une expression est bien parenthésée), on étend cette classe en la classe des langages algébriques décrits par des grammaires (algébriques) qui est aussi décidable.

A savoir

- Le vocabulaire de la théorie des langages formels.
- Les opérations principales sur les mots et les langages (en particulier concaténation, puissance).
- Justifier qu'un mot est un élément d'un langage donné.

Analyse lexicale–langage régulier

Crédits

- Transparents Alain Denise
  - extraits de polycopiés de Sophie Gire, de Michel Termier et de Jean Privat
  - extraits du cours Claude Marché et Ralf Treinen

Introduction

Objectif de l'analyse lexicale et syntaxique

Entrée : suite de caractères
Sortie : arbre de syntaxe abstraite représentant le programme sous-jacent
Structure intermédiaire : suite d'unités lexicales (lexèmes, tokens)
sert d'alphaber d'entrée à l'analyse syntaxique
Prolématique :

- décrire les langages concernés (qu'est-ce qu'un identificateur, une constante entière ou flottante, une chaîne de caractères. . .)
- construire un analyseur efficace (est-ce que l'entrée appartient au langage, dans quelle catégorie ?)
- engendrer automatiquement l'analyseur à partir de la description

Outils théoriques : expressions régulières, langages réguliers
Outils logiciels : outils de la famille Lex, décliné suivant les langages supports

2–Analyse lexicale

Expressions régulières : introduction

- des caractères spéciaux pour représenter aisément des ensembles de mots sur un alphabet A
- exemple shell (cf cours intro à l'info)

Motifs glob : exemples

On suppose que le répertoire courant contient les fichiers :

```
ficher1.txt   fichier2.txt   PERSO.txt   une_image.jpeg
```

- `ls *.txt`

```
ficher1.txt   fichier2.txt   PERSO.txt
```

(n'importe quel nom qui termine par `.txt`)

- `ls *[0-9]*`

```
ficher1.txt   fichier2.txt
```

(n'importe quel nom de fichier qui contient un chifffre)

- `ls [a-z]*`

```
ficher1.txt   fichier2.txt   une_image.jpeg
```

(n'importe quel nom de fichier qui commence par une minuscule)

Motifs shell

- Même base mais syntaxe des motifs différente (`*`, `?`)
- syntaxe des expressions régulières GLOB

Règles d'expansion: `*` n'importe quelle chaîne
`?` n'importe quel caractère
`[ab12…]` un caractère dans la liste
`[^ab12…]` un caractère absent de liste
`[a-z]` un caractère dans l'intervalle
`[^a-z]` un caractère absent de l'intervalle

`?(`m_1`|…|`m_n`)` `@(`m_1`|…|`m_n`)` `*(`m_1`|…|`m_n`)` `+(`m_1`|…|`m_n`)`
k motifs parmi les m_i :

Expressions régulières : définition

- On se place sur un alphabet A (ensemble fini de caractères)
- On cherche à décrire un langage A, c'est à dire un ensemble de mots ($L \subseteq A^*$)
- On définit des *notations* pour décrire ce langage
  - ϵ : décrit le langage réduit au mot vide $\{\epsilon\}$
  - $w \in A^*$: un mot sur A, décrit le langage réduit au mot w, ie. $\{w\}$
  - si e, e_1, e_2 sont déjà des expressions régulières qui correspondent aux langages L, L_1, L_2 alors on peut former de nouvelles expressions régulières
    - union : $e_1 \mid e_2$ correspond au langage $L_1 \cup L_2$
    - concaténation : $e_1 e_2$ correspond au langage $L_1 L_2$
    - étoile : e^* correspond au langage L^* ($\bigcup_{n \in \mathbb{N}} L^n$)
  - des parenthèses (e) permettent de lever les ambigüités sans changer l'interprétation : on écrira $(e_1 \mid e_2)^*$ ou $e_1 \mid (e_2^*)$
- Les caractères spéciaux $\epsilon, |, *, (,)$ sont différents des caractères de l'alphabet.
- les expressions régulières sont appelées de manière équivalente expressions rationnelles

Exemple : constantes entières

- alphabet des caractères ASCII
- entiers non signés (plusieurs tentatives) :
  - $(0|1|2|3|4|5|6|7|8|9)^*$ contient le mot vide !
  - $(0|1|2|3|4|5|6|7|8|9)((0|1|2|3|4|5|6|7|8|9)^*)$ des zéros inutiles au début
  - $(1|2|3|4|5|6|7|8|9)((0|1|2|3|4|5|6|7|8|9)^*)$ manque 0
  - $0 \mid ((1|2|3|4|5|6|7|8|9)((0|1|2|3|4|5|6|7|8|9)^*))$

Exercice

Enumérer le mots de longueur inférieure ou égale à 3 dans les langages représentés par les expressions régulières suivantes

1. $(a|b|c|d)^*$
2. $(a|b)^*(c|d)$
3. $(a|b)^*(c|d)^*$
4. $(a^*|c)^*$

Solution

Exercice

Donner des expressions régulières pour chacun des langages suivants sur l'alphabet $\{a, b, c\}$

1. ensemble des mots qui commencent par abc
2. ensemble des mots qui ne contiennent pas de b
3. ensemble des mots qui commencent par abc ou ac
4. intersection des deux langages précédents

Solution

Exercice

Pour chacun des mots (colonnes) et des expressions régulières (lignes), déterminer si le mot appartient au langage défini par l'expression :

	ϵ	$abba$	a^{10}	b^7	$aaabbb$	$abbbabbabbababababbababa$
$aa\|(b^*)$						
$(aa\|b)^*$						
$(a^*)^*$						
$(a^*b^*)^*$						
$(a\|b)^*$						
$(aab(a^*\|b^*)baa)$ $\|\ ba(a^*)$						

Solution

Langage régulier

Definition (Langage régulier)

- Un langage (non vide) est dit régulier (ou de manière équivalente rationnel) s'il existe une expression régulière qui le décrit.
- Par convention l'ensemble vide de mots $\emptyset$ est un langage régulier.

Il existe des langages non réguliers : $\{a^n b^n | n \in \mathbb{N}\}$
(on ne peut pas "compter")

Expressions régulières étendues

Pour faciliter la description des langages, on introduit des notations supplémentaires.
Soit e une expression régulière (étendue) qui correspond au langage L

- e^+ décrit le langage LL^* ($\bigcup_{n>0} L^n$)
- $e?$ décrit le langage $L \cup \{\epsilon\}$
- $[a_1 \dots a_n]$ avec $a_i \in A$ des caractères, décrit le langage $\{a_1, \dots, a_n\}$
- $[\hat{}\, a_1 \dots a_n]$ avec $a_i \in A$ des caractères, décrit le langage $\{b | b \in A \setminus \{a_1, \dots, a_n\}\}$
- si a et b sont deux caractères ASCII représentant des chiffres ou bien deux caractères ASCII représentant des caractères alphabétiques majuscules ou minuscules, on peut écrire entre les crochets $a - b$ pour représenter l'ensemble des caractères dont le code ASCII est compris entre celui de a et celui de b.

Exemples d'expressions régulières étendues

- $[0-9]+$: suite non vide de chiffres
- $''[\hat{}''] *''$ suite de caractères délimitée par des guillements $''$
- $[a-z\,A-Z]([a-z\,A-Z\,0-9]+)$ identifiant ne contenant que des caractères alphanumériques et commençant par un caractère alphabétique

Propriété

- Les expressions régulières étendues décrivent des langages réguliers
- Expressions régulières équivalentes
  - $e+$ correspond à l'expression régulière $ee*$
  - $e?$ correspond à l'expression régulière $e|\epsilon$
  - $[a_1 \ldots a_n]$ avec $a_i \in A$ des caractères correspond à l'expression régulière $a_1|\ldots|a_n$
  - $[\hat{}a_1 \ldots a_n]$ avec $a_i \in A$ des caractères correspond à l'expression régulière $b_1|\ldots|b_m$ avec $\{b_1, \ldots, b_m\} = A \setminus \{a_1, \ldots, a_n\}$ (l'alphabet est fini)

A savoir

- le langage des expressions régulières étendues
- le lien entre une expression régulière et le langage associé
  - la définition d'un langage régulier
  - trouver le langage associé à une expression régulière
  - construire l'expression régulière correspondant à un langage

2–Analyse lexicale

Analyse : introduction

- On a caractérisé une classe de langages dont les éléments sont décrits simplement par une expression régulière.
- On s'intéresse maintenant à des techniques de reconnaissance.
- Introduction de la notion générale d'automate fini.
- les automates sont utilisés dans de nombreux contextes
  - modélisation de systèmes interactifs
  - procédures de décision

Automate fini : introduction

- Un ensemble fini d'états dont un état initial et plusieurs états finaux
- Des transitions entre les états déclenchées par une *entrée* externe
- Est-ce qu'une suite d'entrées peut faire passer de l'état initial à un état final ?
- Variante :
  - Version déterministe : pour chaque entrée et chaque état, un seul état cible possible
  - Version avec ϵ-transitions (transitions silencieuses) : le système peut changer d'état de manière *spontanée* (sans lire d'entrée)

Automates : extensions

- automate à pile : le système dispose d'une mémoire qui lui permet de stocker des informations sur les transitions prises et de les utiliser pour choisir la transition à effectuer (grammaires et langages algébriques) permet de compter
- machine de Turing : possibilité de lecture, écriture sur un ruban infini (tous les langages reconnaissables sur machine)
- automates temporisés (transition par passage du temps)
- automates d'arbres
- ...

Automate fini : définition

Un automate fini est formé de cinq composants présentés comme un quintuplet (A, Q, q_0, F, δ)

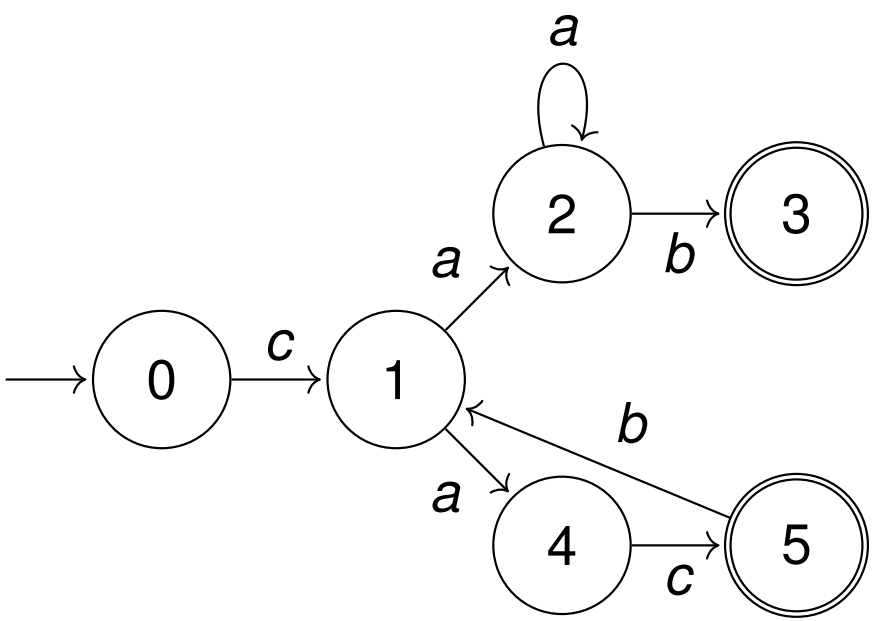
- A alphabet, ensemble fini de caractères (entrées)
- Q est un ensemble *fini* appelé ensemble des états
- $q_0 \in Q$ est l'état initial
- $F \subseteq Q$ est l'ensemble des état terminaux (on dit aussi état final)
- δ (delta) est la fonction de transition.
  Elle prend en argument un état $q \in Q$ et un élément de $A \cup \{\epsilon\}$ (entrée dans l'alphabet $a \in A$ ou transition silencieuse) et renvoie un ensemble d'états qui correspond à tous les états auxquels on peut arriver à la suite de q pour l'entrée a ou de manière silencieuse.
  Il est possible que cet ensemble soit vide.
- alternativement, on peut voir δ comme une relation, sous ensemble de $Q \times (A \cup \{\epsilon\}) \times Q$ (les transitions possibles entre les états).
  transition : $(q_1, a, q_2) \in \delta$ ssi $q_2 \in \delta(q_1, a)$

Automate fini : exemple

$$A = \{a, b, c\} \quad Q = \{0, 1, 2, 3, 4, 5\} \quad q_0 = 0 \quad F = \{3, 5\}$$

Transitions

$$\delta(0, c) = \{1\} \quad \delta(1, a) = \{2, 4\} \quad \delta(2, a) = \{2\}$$
$$\delta(2, b) = \{3\} \quad \delta(4, c) = \{5\} \quad \delta(5, b) = \{1\}$$
$$\delta(n, x) = \emptyset \quad \text{sinon}$$



Automate fini : représentations

- Plusieurs manières de représenter graphiquement les états terminaux (flèche sortante, croix, . . .)
- Représentation de la fonction de transition δ

	a	b	c
0			1
1	2, 4		
2	2		3
3			
4			5
5		1	

Automate fini : représentations

- Représentation relationnelle

$$\delta = \{(0, c, 1), (1, a, 2), (1, a, 4), (2, a, 2), (2, c, 3), (4, c, 5), (5, c, 1)\}$$

- Chaque triplet (q, x, q') avec $q, q' \in Q$ des états et $x \in Q$ un caractère et $q' \in \delta(q, x)$ est appelé une transition de l'automate.
- En tableau

	0	1	2	3	4	5
0		c				
1			a		a	
2			a	b		
3						
4					c	
5		b				

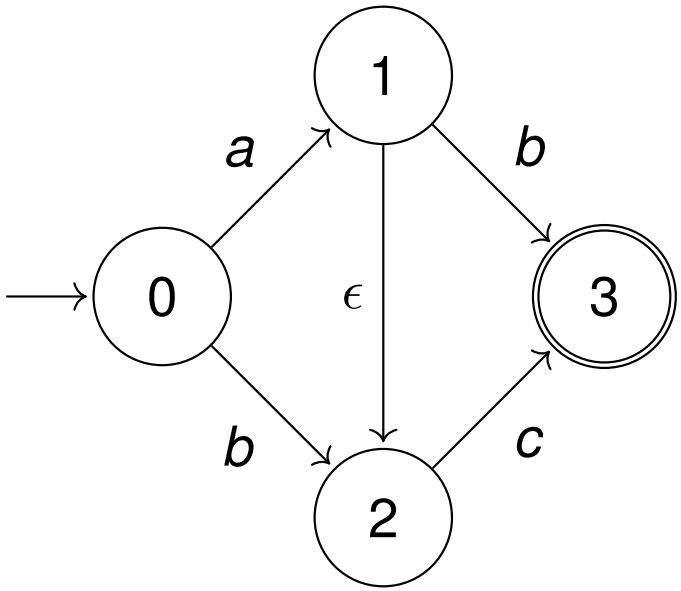
- Plusieurs entrées possibles entre deux états (représentées par une seule flèche étiquetée par les entrées)

Automates avec ϵ-transition

$$A = \{a, b, c\} \quad Q = \{0, 1, 2, 3\} \quad q_0 = 0 \quad F = \{3\}$$

Transitions δ

	a	b	c	ϵ
0	1	2		
1		3		2
2			3	
3				



- Un automate *avec ϵ-transition* est dit asynchrone (réagit même sans entrée)
- Un automate *sans ϵ-transition* est dit synchrone

Automates et langages

- Chaque automate définit un langage, celui des mots *reconnus* entre l'état initial et un état final.
- Définitions préalables des notions de chemin et de trace

Chemin dans un automate

- Automate $\mathcal{A} = (A, Q, q_0, F, \delta)$
- Transition : $\{(q, x, q') \in Q \times (A \cup \{\epsilon\}) \times Q | q' \in \delta(q, x)\}$
- Chemin : suite de transitions *consécutives* (l'état d'arrivée d'une transition est l'état de départ de la suivante)

$$(q_1, x_1, q_2)(q_2, x_2, q_3) \dots (q_n, x_n, q_{n+1})$$

- Le chemin mène du sommet q_1 (origine) au sommet q_{n+1} (destination)
- Le chemin a pour longueur n
- On note δ^* l'ensemble de toutes les suites de transitions associées à la fonction δ

Trace d'un chemin

- A chaque chemin C on peut associer un mot $tr(C)$ de A^* en concaténant les caractères qui apparaissent dans les transitions
- Au chemin $(q_1, x_1, q_2)(q_2, x_2, q_3) \dots (q_n, x_n, q_{n+1})$, on associe le mot $x_1 x_2 \dots x_n$
- Ce mot a pour longueur au plus n (moins en cas de ϵ-transition)

Langage reconnu par un automate

- Un mot $u \in A^*$ est reconnu (on dit aussi accepté) par un automate $\mathcal{A}$ s'il existe un chemin C d'origine l'état initial, dont la destination est un état final et dont la trace est le mot u.
- Le langage reconnu par l'automate $\mathcal{A}$ est l'ensemble des mots reconnus par $\mathcal{A}$. Il est noté $L(\mathcal{A})$
- Un langage L est dit reconnaissable s'il existe un automate fini $\mathcal{A}$ tel que $L = L(\mathcal{A})$
- Théorème de Kleene : les langages reconnaissables sont exactement les langages réguliers
  - Description par une expression régulière (étendue)
  - Reconnaissance par un automate fini

Exercice

Soit l'alphabet $A = \{a, b, c\}$
Donner un automate fini pour chacun des langages suivants

1. Mots qui commencent par abc
2. Mots qui ne contiennent pas de b Solution
3. Mots qui commencent par abc ou ac
4. Mots contenant au moins trois occurrences successives de a Solution
5. Mots de longueur paire Solution

Automate fini déterministe

- Un automate fini est déterministe ssi
  - il est synchrone (pas de ϵ-transition)
  - la relation de transition est fonctionnelle : au plus un état dans $\delta(q, x)$
    - On a alors une fonction de transition (partielle) $\delta : Q \times A \to Q$
- on peut toujours rendre un automate déterministe mais cela peut faire grossir exponentiellement le nombre des états (voir suite du cours)

Exercice

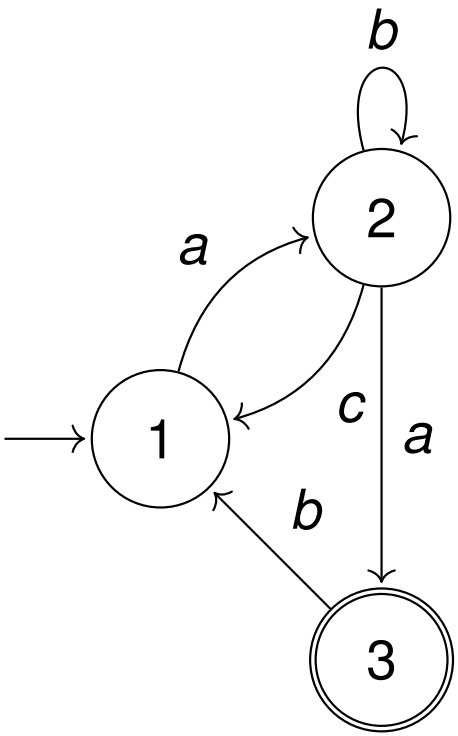
- construire un automate fini déterministe qui reconnait tous les mots dans lesquels apparait le mot *abc* :
- un tel mot s'écrit $u\mathbf{abc}v$
- on dit que *abc* est un facteur du mot
  (à ne pas confondre avec un sous-mot qui peut effacer des lettres intermédiaires)

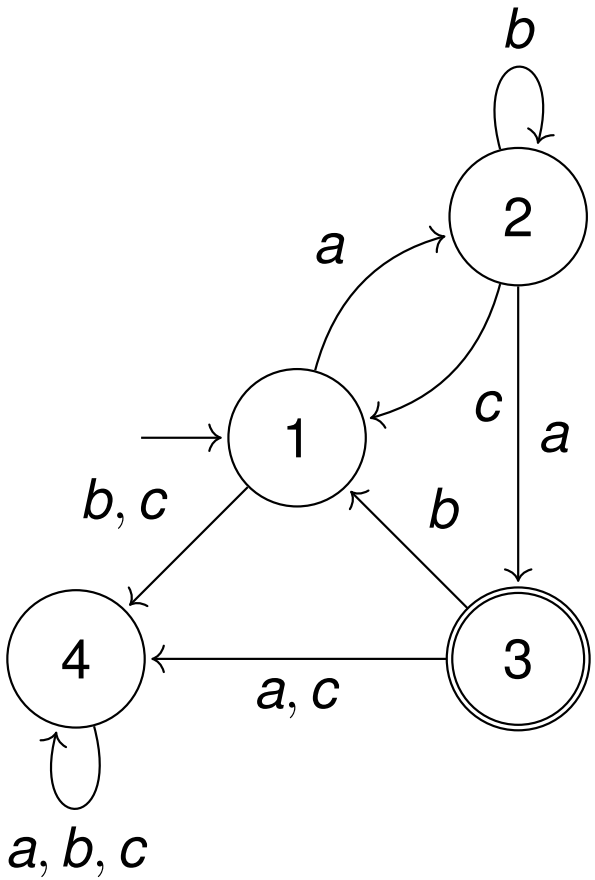
Solution

Automate fini déterministe complet

- Un automate fini déterministe est complet ssi la fonction de transition δ est *totale*
- il y a une transition possible pour chaque état et chaque caractère
- il est facile de rendre un automate déterministe complet
  - ajouter un état *puits* : $\bot$
  - compléter la fonction de transition partielle $\delta \in Q \times A \to Q$ en une fonction totale $\delta \in (Q \cup \{\bot\}) \times A \to Q \cup \{\bot\}$
    - $\delta'(q, x) = \delta(q, x)$ si $\delta(q, x)$ est défini
    - $\delta'(q, x) = \bot$ si $\delta(q, x)$ non défini ou $q = \bot$
- aucune transition ne sort de l'état $\bot$ vers un état de Q et $\bot$ n'est pas un état final : aucun chemin de l'état initial à un état final ne passe par $\bot$
- le langage reconnu n'est pas changé.

Automate fini déterministe complet : exemple





Automate fini déterministe complet : applications

- avoir une fonction de transition totale est plus facile à gérer
- construction directe de l'automate qui reconnaît le complémentaire du langage : on inverse les états terminaux et les états non terminaux.
- Soit $\mathcal{A} = (A, Q, q_0, F, \delta)$ un automate déterministe complet qui reconnait le langage L, alors $\mathcal{A} = (A, Q, q_0, Q \setminus F, \delta)$ est un automate déterministe complet qui reconnaît le langage $A^* \setminus L$.

A savoir

- La définition d'un automate fini et le vocabulaire associé (état initial, terminal, transition)
- Les différentes catégories d'automates : synchrone/asynchrone, déterministe ou non, complet ou non
- Lien entre le langage reconnu par l'automate et l'automate
  - les automates finis reconnaissent exactement les langages réguliers
  - construire un automate pour un langage donné
  - reconnaître qu'un mot appartient au langage accepté par un automate

Correction

1. $(a|b|c|d)^*$: Tous les mots sur l'alphabet $\{a, b, c, d\}$

   ϵ, a, b, c, d, aa, ab, ac, ad, ba, bb, bc, bd, ca, cb, cc, cd, da, db, dc, dd,
   aaa, aab, aac, aad, aba, abb, abc, abd, aca, acb, acc, acd, ada, adb, adc, add,
   baa, bab, bac, bad, bba, bbb, bbc, bbd, bca, bcb, bcc, bcd, bda, bdb, bdc, bdd,
   caa, cab, cac, cad, cba, cbb, cbc, cbd, cca, ccb, ccc, ccd, cda, cdb, cdc, cdd,
   daa, dab, dac, dad, dba, dbb, dbc, dbd, dca, dcb, dcc, dcd, dda, ddb, ddc, ddd

2. $(a|b)^*(c|d)$: Mots formés de a et b qui se terminent par un c ou un d

   c, d, ac, ad, bc, bd, aac, aad, abc, abd, bac, bad, bbc, bbd

Correction

1. $(a|b)^*(c|d)^*$: Mots formés d'une suite (possiblement vide) de a et de b puis d'une suite (possiblement vide) de c et de d

   $\epsilon, a, b, c, d, aa, ab, ac, ad, ba, bb, bc, bd, cc, cd, , dc, dd,$
   $aaa, aab, aac, aad, aba, abb, abc, abd, acc, acd, adc, add,$
   $baa, bab, bac, bad, bba, bbb, bbc, bbd, bcc, bcd, bdc, bdd,$
   $ccc, ccd, cdc, cdd, dcc, dcd, ddc, ddd$

2. $(a^*|c)*$: Mots formés juste de a et de c

   $\epsilon, a, c, aa, ac, ca, cc, aaa, aac, aca, acc, caa, cac, cca, ccc$

Retour

Correction

Alphabet $\{a, b, c\}$

- ensemble des mots qui commencent par abc :

$$(abc)(a|b|c)^*$$

- ensemble des mots qui ne contiennent pas de b :

$$(a|c)^*$$

- ensemble des mots qui commencent par abc ou ac:

$$a(b|\epsilon)c(a|b|c)^*$$

- intersection des deux langages précédents:

$$ac(a|c)^*$$

Retour

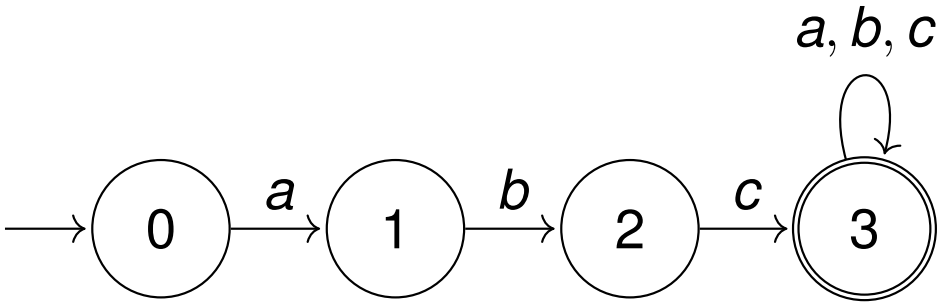
Solution

	ϵ	$abba$	a^{10}	b^7	$aaabbb$	$abbbabbabbababababbababa$
$aa\|(b^*)$	O	N	N	O	N	N
$(aa\|b)^*$	O	N	O	O	N	N
$(a^*)^*$	O	N	O	N	N	N
$(a^*b^*)^*$	O	O	O	O	O	O
$(a\|b)^*$	O	O	O	O	O	O
$(aab(a^*\|b^*)baa)$ $\|\ ba(a^*)$	N	N	N	N	N	N

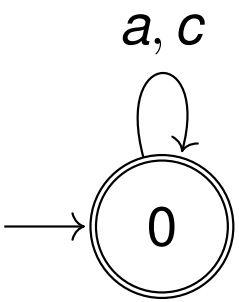
Retour

Solution

Mots qui commencent par *abc*



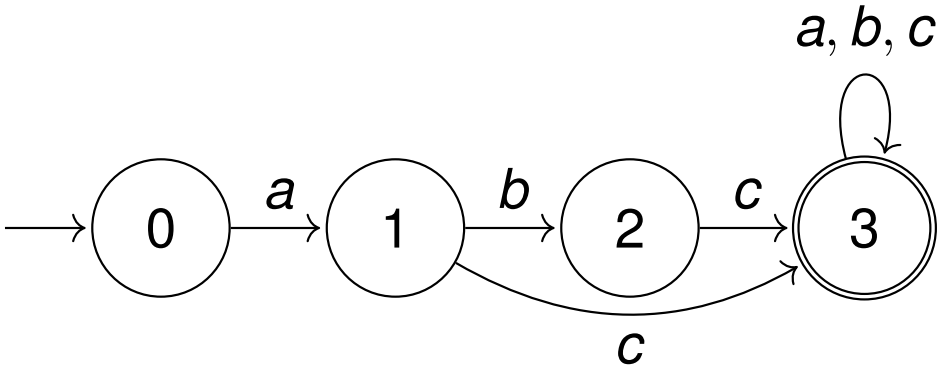
Mots qui ne contiennent pas de *b*



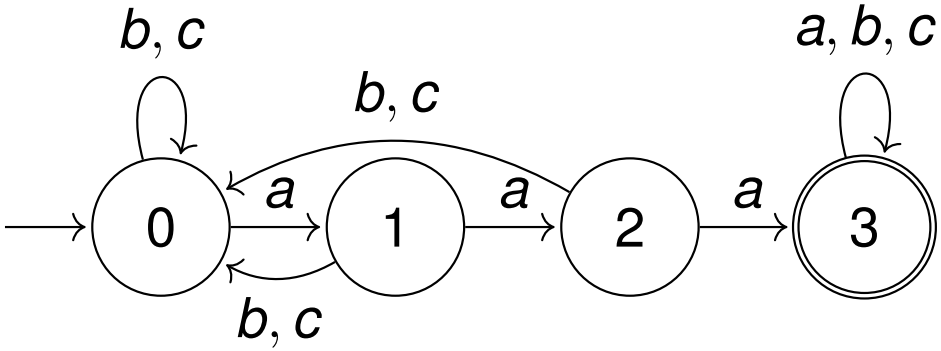
Retour

Solution

Mots qui commencent par *abc* ou *ac*



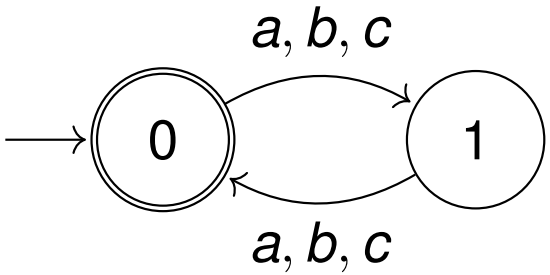
Mots contenant au moins trois occurrences successives de *a*



Retour

Solution

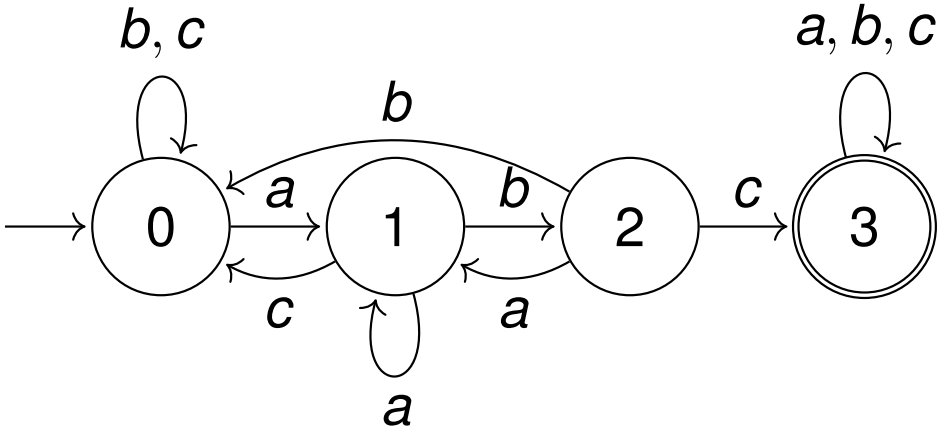
Mots de longueur paire



Retour

Solution

tous les mots dans lesquels apparait le mot *abc* :



Retour

Analyse lexicale : générateurs d'analyseur (*lex)

Rappel

- Les expressions régulières sont des *notations* pour représenter des langages réguliers
- Les automates finis permettent de *reconnaître* qu'un mot appartient à un langage régulier

Analyse lexicale des langages de programmation

- Plusieurs unités lexicales (mots-clés, symboles, identificateurs, constantes entières, flottantes, chaîne de caractères. . .) dans une suite de caractères (le programme en entrée).
  - Chaque unité lexicale appartient à un langage décrit par une expression régulière : e_i
  - L'unité est (en général) associée à une classe (comme la classe des identifiants ou celle des constantes ou celle des symboles de comparaison)
  - S'il y a plusieurs mots possibles, on associe à la classe une valeur (l'entier, le flottant, la chaîne. . .)
  - Certaines parties du texte sont ignorées, elles servent de séparateur (blancs, commentaires. . .)
- Le texte est découpé en unités successives
- Langage global reconnu : $(e_1 \mid \ldots \mid e_n)^*$
- L'analyseur produit un mot sur l'alphabet des classes d'unités lexicales

Générateur d'analyseur

Objectif

- Reconnaître dans une suite de caractères des mots tels que décrits par des expressions régulières
- En fonction de l'expression régulière et du mot reconnu, effectuer une *action*

Fonctionnalité

- Engendrer un automate (déterministe) à partir d'une expression régulière
- Fonction de lecture d'une suite de caractères et de reconnaissance successive des mots

Générateur d'analyseur : Approche

- Langage spécifique pour décrire l'analyseur (expression régulière/action)
- Outil spécifique qui engendre le code de l'analyseur
- Insertion naturelle du code engendré dans le langage de programmation cible
- Application :
  - analyse lexicale des langages de programmation
  - mais aussi : coloriage, extraction d'information, transformation de textes, etc

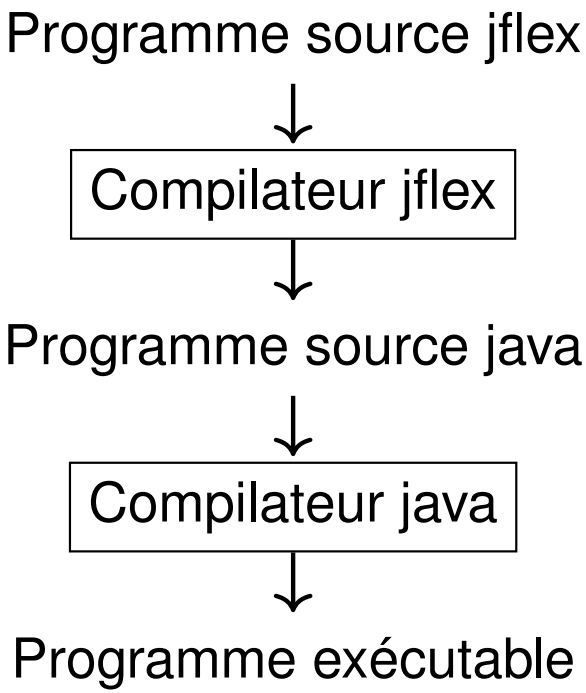
Programmes *lex

- Langage de description avec ses règles lexicales, syntaxiques, sémantique
- Modèle d'exécution ad-hoc
  - Traiter une suite de caractères (ASCII, unicode) en entrée
  - Effectuer des actions en fonction de *motifs* reconnus, décrits par des expressions régulières
- Généraliste
- Optimisé pour le traitement efficace de grandes données
- Facilités de description des expressions régulières, de repérage dans le texte initial (ligne, colonne)
- Règles opérationnelles
  - choix de reconnaître la suite de caractères la plus longue possible
  - priorités entre deux motifs de même longueur
- Exploitation de l'automate
  - superposer plusieurs automates (reconnaissance spécifique des commentaires ou des chaînes de caractères)
  - mémoriser des informations via les actions (langages non réguliers)

Jflex

- Une approche ancienne (`lex` pour C, Bell labs, 1975)
- Ce cours : utilisation de `jflex` pour Java
  - Site `https://www.jflex.de/`
  - Un mémento (Télécom Sud-Paris) bien fait : `http://www-inf.telecom-sudparis.eu/cours/CSC4536/web/?page=Mementos/jflex-memento`
- D'autres analyseurs sur le même principe : `ocamllex`, `flex` ...

Schéma d'utilisation



description.jflex

```
> jflex description.jflex
```

Yylex.java

```
> javac Yylex.java
```

Yylex.class

```
> java Yylex fichier.txt
```

Format d'un fichier jflex

- Trois parties séparées par le symbole `%%` seul sur une ligne
- Commentaires à la Java (de `//` à la fin de ligne ou `/* ... */`)

```
/* Préambule : code utilisateur */
  // Déclarations préliminaires : import, package
  // Recopié tel quel dans la cible
%%
/* Déclarations et options */
  // directives "%..."
  // blocs de code auxiliaire "%{" ... "%}"
  // abbréviations d'expressions régulières : nom = regexp
%%
/* Règles lexicales */
  // Règle sous forme :
  // regexp { /* code java de l'action */ }
```

Exemple

- Extraire des adresses mail d'un fichier texte
- Spécification des adresses mail (approximation)
  - suite de caractères alpha-numériques, points et tirets, commence par un caractère alphabétique
  - symbole @
  - suite de caractères alpha-numériques, points et tirets, commence par un caractère alphabétique
  - suffixe final : point suivi d'une suite non vide de caractères alphabétique
- Impression des adresses trouvées, reste du texte ignoré

JFlex : exemple

```
%%
%include ../TP/Jflex.include
%standalone
%class Mail
%{
  void ECHOL(String cat)
     { System.out.println("["+ cat +":"+ yytext() +"]"); }
%}
AL = [a-zA-Z]
OK = {AL} | [0-9.-]
%%
{AL} {OK}* @ {AL} {OK}* \. {AL}+
          { ECHOL("MAIL");}
[^]       { }
```

JFlex : ligne à ligne

```
%%
```

Séparateur entre le préambule (code java import) et les déclarations Jflex

```
%include ../TP/Jflex.include
```

Directive `%include` pour partager un fichier de configuration

```
%standalone
```

Directive `%standalone` pour engendrer la fonction main (lecture de l'entrée dans un fichier)
Dans le mode standalone, les caractères qui ne sont pas reconnus sont renvoyés sur la sortie standard

```
%class Mail
```

Directive `%class Mail` fixe le nom de la classe engendrée
(par défaut `Yylex`)

JFlex : ligne à ligne

```
%{
  void ECHOL(String cat)
     { System.out.println("["+ cat +":"+ yytext() +"]"); }
%}
```

- Code Java entre `%{ ... %}` inséré dans la classe, utilisé dans les actions
- Le code utilise la fonction `yytext()` qui renvoie dans l'action la chaîne de caractère reconnue correspondant à l'expression régulière (type `String`)

```
AL = [a-zA-Z]
OK = {AL} | [0-9.-]
```

- abbréviations pour les expressions régulières de la forme `nom=regexp`
- le nom sera réutilisé dans d'autres expressions régulières sous la forme `{nom}` (ne pas oublier les accolades!)
- expression régulière, pas de cycles! (`PAR = ( {PAR} )`)

JFlex : ligne à ligne

```
%%
```

Séparateur entre la partie déclarations Jflex et les règles lexicales

```
{AL} {OK}* @ {AL} {OK}* \. {AL}+
        { ECHOL("MAIL");}
[^]     { }
```

- les règles de la forme `regexp { code }`
- les caractères `<>{}()[]|!~*+?"^$/.\,-` ont un sens particulier dans la syntaxe Jflex, leur utilisation dans les expressions régulières doit être échappée (ici `\.` désigne le point)
- l'expression régulière `[^]` est spécifique à Jflex, elle reconnaît n'importe quel caractère (les différentes formes de retour à la ligne comprises), utile en fin de spécification pour traiter ce qui n'aura pas été reconnu par les autres règles.
- possibilité d'action vide (l'entrée est consommée)

Compilation

```
>jflex mail.jflex

Reading "mail.jflex"
Including "../TP/Jflex.include"
Constructing NFA : 26 states in NFA
Converting NFA to DFA :
........
10 states before minimization, 8 states in minimized DFA
Old file "Mail.java" saved as "Mail.java~"
Writing code to "Mail.java"

>javac Mail.java
>java Mail test-mail.csv

[MAIL:christine.paulin@universite-paris-saclay.fr]
[MAIL:frederic.gruau@universite-paris-saclay.fr]
[MAIL:jeremy.labiche@universite-paris-saclay.fr]
[MAIL:alida.nana-sunji-kouanang@universite-paris-saclay.fr]
[MAIL:nathan.thomasset@universite-paris-saclay.fr]
[MAIL:mathias.loiseau@universite-paris-saclay.fr]
...
```

Expressions régulières Jflex

- Alphabet : dépend du système de caractères (ASCII, unicode. . .)
- Expressions régulières étendues
  - opérations de base : mot, concaténation, union, étoile et extensions
    - `a(bc)*a+b?|aaa`
  - ensemble de caractères :
    - `[1-5]` ou exclusif `[^a-z]`
    - caractère *universel* `.` tout caractère sauf les fins de ligne (`\r\n | [\r\n\u2028\u2029\u000B\u000C\u0085]`)
    - expression régulière pour les fins de ligne : `\R` (Jflex)
    - caractère quelconque : `[^]` (Jflex)
  - itérations bornées :
    - `r{N}` : répétition exactement N fois
    - `r{N,M}` : répétition n fois avec $N \leq n \leq M$

Expressions régulières contextuelles

- `^r` reconnaît *r* uniquement en début de ligne, le chapeau doit être le premier caractère de l'expression régulière, sinon c'est un simple caractère
  Exemples : `^\n` : ligne vide, `^[ \t]+$` une ligne blanche non vide (`^[ \t]*$` n'est pas autorisé car `[ \t]*` contient le mot vide),
- `r$` reconnaît *r* uniquement en fin de ligne, le chapeau doit être le dernier caractère de l'expression régulière, sinon c'est un simple caractère, le retour à la ligne suivant n'est pas consommé
- `r/s` reconnait *r* uniquement si suivi d'un mot de *s* mais ne retient que la partie correspondant à *r*, usage simultané de `/` et `$` interdit.

Expressions régulières Jflex

- Règles de priorité
  - les opérateurs postfixes `*`, `+` et `?` sont plus prioritaires que la concaténation elle-même plus prioritaire que l'union
    - `ab*|c+` correspond à `(a(b*))|(c+)`
  - des parenthèses pour des articulations différentes comme `((ab)*|c)+`
- Caractères réservés
  - `<>{}()[]|!~*+?"^$/.\,-`
  - à protéger dans une chaîne `"..."` ou avec le caractère d'échappement `\` comme dans `\.`
  - échappement non nécessaire dans certains contextes comme les classes de caractères (à l'exception du tiret) ou les caractères `^$` lorsqu'ils ne sont pas en début ou fin d'expression.
  - exemples : `"(.)"` `\.\.\.` `\".*\"` `[+*/-]` `[+*-/]`

Expressions régulières récapitulatif

e f	l'expression *e* suivie de l'expression *f*
e | *f*	l'expression *e* ou l'expression *f*
e?	0 ou 1 fois l'expression *e*
e⋆	0 ou *n* fois l'expression *e* (*n* quelconque)
e+	1 ou *n* fois l'expression *e*(*n* quelconque)
e{*n*}	*n* fois l'expression *e* (*n* quelconque)
e{*n*, *m* }	*k* fois l'expression *e* (*k* quelconque $n \leq k \leq m$)
(*e*)	l'expression *e*
{*D*}	l'expression correspondant à la définition *D*

e/*f*	l'expression *e* si suivie de l'expression *f*
^*e*	l'expression *e* en début de ligne
e$	l'expression *e* en fin de ligne

.	tout caractère sauf fin de ligne (\n \r...)
c	le caractère *c* non spécial (fonction du contexte)
c	le caractère *c* même spécial
"*abc*"	le mot *abc* même si caractères spéciaux
[*abc*]	les caractères *a* ou *b* ou *c*
[*a* − *c*]	les caractères de code entre *a* et *c*
[^*abc*]	les caractères sauf *a* et *b* et *c*

Interprétation des règles

- reconnaître un mot dans le langage associé à l'union des expressions régulières
- chaque état final est associé à la règle concernée
- un même état peut être final pour plusieurs règles :
  - *mode glouton* : le système choisit la règle qui correspond au mot le plus long reconnu et, en cas d'égalité, la *première* règle dans le fichier
- état du système :
  - position de début dans la suite de caractères d'entrée
  - position du caractère courant
  - état courant dans l'automate
  - possiblement dernier état final rencontré et position du caractère associé
- lorsque le prochain caractère lu ne permet pas d'avancer :
  - retour au dernier état final rencontré et au mot reconnu à ce stade
  - déclenchement de l'action correspondante
  - reprise de la lecture au caractère qui suit le mot reconnu à partir de l'état initial de l'automate
  - en mode `%standalone`, les caractères non reconnus sont juste transcrits sur la sortie standard (ajouter une *règle balai* pour les traiter explicitement)

Jflex : directives

```
%%
%include ../TP/Jflex.include

// int yyline, yycolumn; long yychar;
%line
%column
%char

// renommage de la classe engendrée : MaClasse.java défaut Yylex
// et fonction d'analyse : "int next_token()" défaut yylex
%class MaClasse
%int
%function next_token
```

Jflex : directives

```
%standalone // Programme autonome
%cup // Programme couplé avec CUP

%init{ /* code dans le constructeur : action initiale */
System.out.println("Analyse Lexicale Sample0 (type any text) :");
%init}
%eof{  /* code en action final */
System.out.println("Bye!");
%eof}
%caseless              /* confondre minuscules/majuscules   */
%state   ETAT, ETAT2 /* États inclusifs du super-automate */
%xstate  STATE       /* États exclusifs du super-automate */
```

Jflex : opérations prédéfinies

- `String yytext()` : la valeur du mot reconnu
- `int yylength()` : la longueur du mot reconnu
- `char yycharat(int pos)` : caractère de la chaîne reconnue
- `long yychar; int yycolumn, yyline` si les directives correspondants sont activées, donne la position courante dans la lecture (affichage des erreurs)

Actions

- La même action peut être partagée entre plusieurs expressions régulières avec la syntaxe

```
^a*     |
b*$     |
ab/a  { ECHO();}
```

traduit en

```
^a*   { ECHO();}
b*$   { ECHO();}
ab/a  { ECHO();}
```

Au delà des langages réguliers

- Grace aux actions, on peut se servir d'un outil comme Jflex pour reconnaître des langages qui ne sont pas réguliers.
- Exemple : compter les parenthèses ouvrantes et fermantes (commentaires imbriqués)

Exemple langage non-régulier

```
%%
%standalone
%int
%line
%column
%class Parens
%{
 int par = 0;
 int erreur ()  {
   System.out.println( "parenthèse fermante en trop ligne "
                       +(yyline+1) +" colonne "+yycolumn);
   return 1;}
%}
%eofval{
 if (par > 0) {System.out.println(par+" parenthèses ouvertes non fermées");
               return 1;}
 else  {System.out.println("OK"); return 0; }
%eofval}
%%
 \(       {par++;}
 \)       {if (par > 0)  {par--; } else { erreur (); }}
 [^()]    { }
```

Etats contextes

- Les expressions régulières prennent en compte le contexte de manière limitée (début/fin de ligne, mot suivant)
- Lors de l'analyse d'un texte, on peut vouloir utiliser des règles spécifiques sur une portion de texte
- En pratique on introduit plusieurs parcours possibles de l'automate, en fonction du *contexte*
- Règles étiquetées par l'état contexte si elles s'appliquent lorsqu'on est dans l'état considéré
- Fonction dans les actions pour tester/modifier l'état contexte

états-contexte : mise en œuvre

- déclaration par une directive

```
%state ETAT1
%xstate ETAT2
```

- état-contexte par défaut `YYINITIAL`
- étiquetage d'une ou plusieurs règles par un ou plusieurs états-contexte

```
<ETAT1> regexp {action}
<ETAT1,ETAT2>{
  regexp1   {action1}
  regexp2   {action2}
}
```

- fonctions dans les actions `yybegin(etat)` pour se placer dans le contexte `etat`, `yystate()` pour récupérer l'état courant
- les règles non étiquetées s'appliquent dans tous les états inclusifs (déclaration `%state`) mais pas aux états exclusifs (déclaration `%xstate`)
- Etat initial par défaut (inclusif) `YYINITIAL`

Exemple des chaînes de caractères

```
%%
%standalone
%class Chaines
%{
 StringBuffer string = new StringBuffer();
%}
%state STRING

%%
<YYINITIAL> \"          { string.setLength(0); yybegin(STRING); }

<STRING> {
      \"                { System.out.println(string.toString());
                          yybegin(YYINITIAL);}
      [^\n\r\"\\]+      { string.append( yytext() ); }
      \\t               { string.append('\t'); }
      \\n               { string.append('\n'); }
      \\r               { string.append('\r'); }
      \\\"              { string.append('\"'); }
      \\                { string.append('\\'); }
    }
[^]                     { }
```

Jflex : conclusion

- Des outils efficaces mais d'un usage assez technique
- Un bel exemple d'interprétation de langage !
- Grande sensibilité aux erreurs de syntaxe dans la description jflex
- Les erreurs dans le code Java inclus sont seulement détectées à la compilation du code Java engendré
- Des bibliothèques spécialisées pour des tâches de traitement de texte élémentaires
- Partage de la reconnaissance entre expression régulière et action
  - mots-clés : peut se faire par des expressions régulières spécifiques ou bien en utilisant l'expression des identifiants puis en reconnaissant les mots-clé via une table de hashage

A retenir

- Les principes généraux des générateurs d'analyseurs lexicaux
- Programmer à l'aide d'actions à déclencher sur des mots reconnus par des expressions régulières
- Approfondissement en TP (cette semaine et la semaine prochaine)

Analyse lexicale : génération d'automates

Rappel

- Les expressions régulières sont des *notations* pour représenter des langages dits réguliers
- Les automates finis permettent de *reconnaître* qu'un mot appartient à un langage qui est alors dit reconnaissable
- Les deux notions de langages coïncident
  - on va montrer que tout langage régulier est reconnaissable

Comment engendrer *automatiquement* un automate fini (déterministe) à partir d'une expression régulière ?

Opérations de base des expressions régulières

Alphabet A :

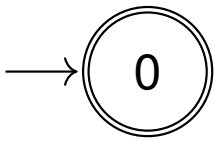
- mot vide : ϵ
- caractère : $a \in A$
- concaténation : $e_1\ e_2$
- union : $e_1 \mid e_2$
- étoile : e_1^*

Construction par récurrence sur la structure de l'expression régulière :

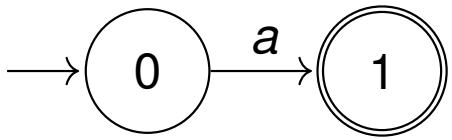
- On construit les automates pour les cas de base ϵ et $a \in A$
- On suppose construits des automates pour e_1 et e_2 qui reconnaissent les langages associés,
  on construit alors l'automate pour $e_1\ e_2$, $e_1 \mid e_2$ et e_1^*
- Restrictions (invariant à préserver):
  - un seul état final
  - pas de transition qui arrive sur l'état initial ou sort de l'état final

Cas de base

- mot vide
  - expression régulière ϵ
  - langage $\{\epsilon\}$
  - automate : $(\{0\}, 0, \emptyset, \{0\})$

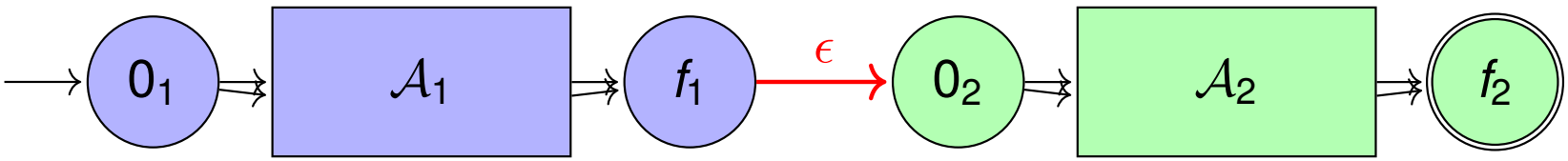


- caractère $a \in A$
  - expression régulière : a
  - langage $\{a\}$
  - automate : $(\{0, 1\}, 0, \{1\}, \{(0, a, 1)\})$



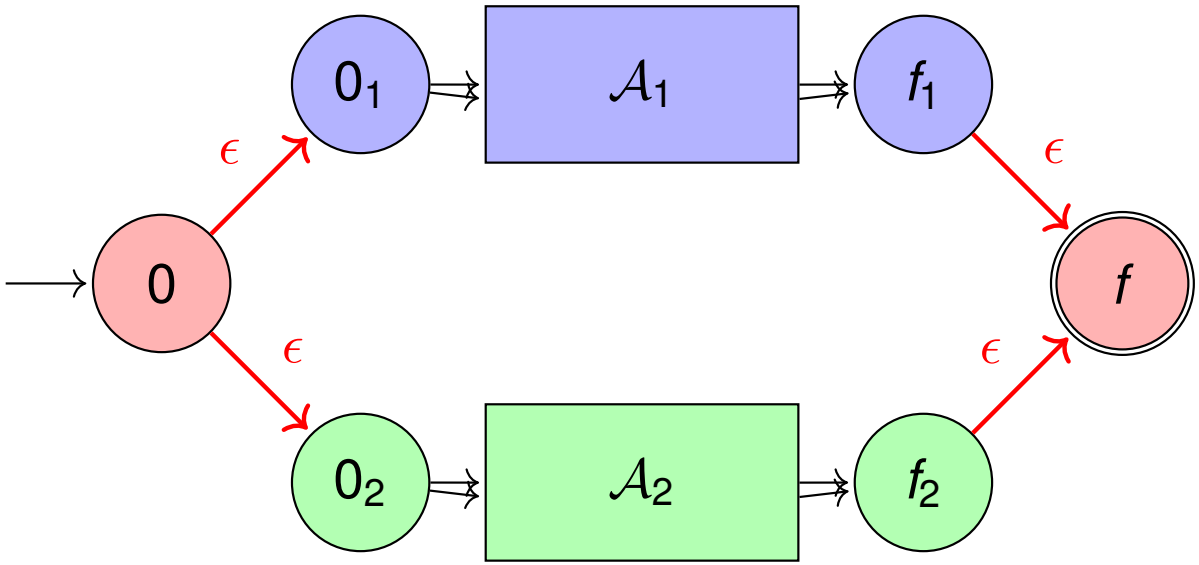
Concaténation

- Déjà construits :
  - expressions régulières e_i ($i = 1, 2$)
  - langages associés L_i ($i = 1, 2$)
  - automates associés $\mathcal{A}_i \stackrel{\text{def}}{=} (S_i, 0_i, \{f_i\}, T_i)$ ($i = 1, 2$) avec $S_1 \cap S_2 = \emptyset$
- expression régulière : $e_1\, e_2$
- langage $L_1 L_2$
- automate : $(S_1 \cup S_2, 0_1, \{f_2\}, T_1 \cup T_2 \cup \{(f_1, \epsilon, 0_2)\})$



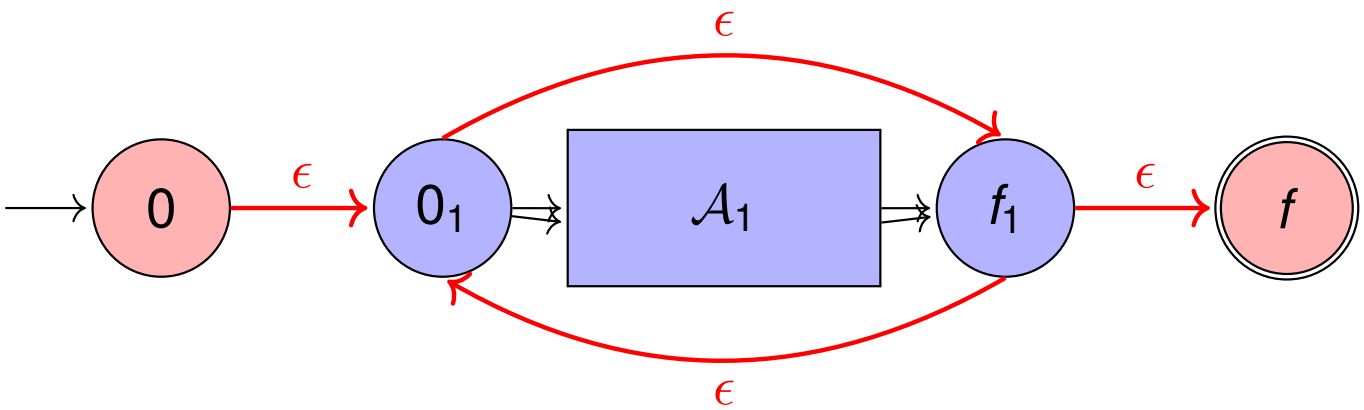
Union

- Déjà construits e_1, e_2 : (comme pour la concaténation)
- expression régulière : $e_1 \mid e_2$
- langage $L_1 \cup L_2$
- automate : $(S_1 \cup S_2 \cup \{0, f\}, 0, \{f\},$
  $T_1 \cup T_2 \cup \{(0, \epsilon, 0_i), (f_i, \epsilon, f) | i = 1, 2\})$



Etoile

- Déjà construit
  - expression régulière e_1
  - langage associé L_1
  - automate associé $\mathcal{A} \stackrel{\text{def}}{=} (S_1, 0_1, f_1, T_1)$
- expression régulière : e^*
- langage L^*
- automate : $\mathcal{A}_1 \stackrel{\text{def}}{=} (S \cup \{0, f\}, 0, \{f\}, T \cup \{(0, \epsilon, 0_1), (f_1, \epsilon, f), (0_1, \epsilon, f_1), (f_1, \epsilon, 0_1)\})$



Preuve

- Le fait que les automates construits reconnaissent au moins les mots du langage régulier associé est immédiat
- la propriété d'un seul état terminal et d'absence de transition qui arrive sur l'état initial ou sort de l'état final est préservé par chaque construction
- Il est un peu plus délicat de montrer que l'automate ne reconnait pas plus de mots
  - analyse des chemins qui partent de l'état initial jusqu'à l'état final
  - le fait qu'il n'y ait pas de transition qui sorte de l'état final ou entre dans l'état initial sert pour la construction étoile

2–Analyse lexicale

Déterminiser un automate

- Un automate déterministe permet de mener l'analyse d'un mot sans retour arrière.
- Pour tout langage régulier, il est possible de trouver un automate déterministe qui reconnait ce langage.
- Algorithme pour construire à partir d'un automate non-déterministe un automate déterministe qui reconnait le même langage
- Si l'automate initial a n états, l'automate déterministe correspondant peut avoir jusqu'à 2^n états.

Déterminiser un automate synchrone

- Soit $\mathcal{A} = (A, Q, q_0, F, \delta)$ un automate
- On construit un nouvel automate sur le même alphabet dont les états sont des sous-ensembles finis de Q (synthétise l'ensemble des états possibles en fonction de l'entrée).
  - Etat initial $Q_0 = \{q_0\}$
  - Transitions :

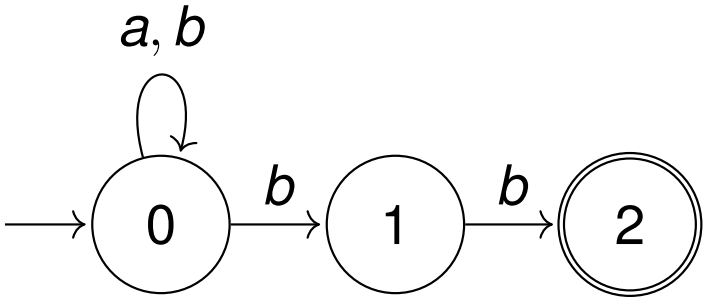
    $$\Delta(\{s_1, \ldots, s_j\}, x) = \delta(s_1, x) \cup \ldots \cup \delta(s_j, x)$$

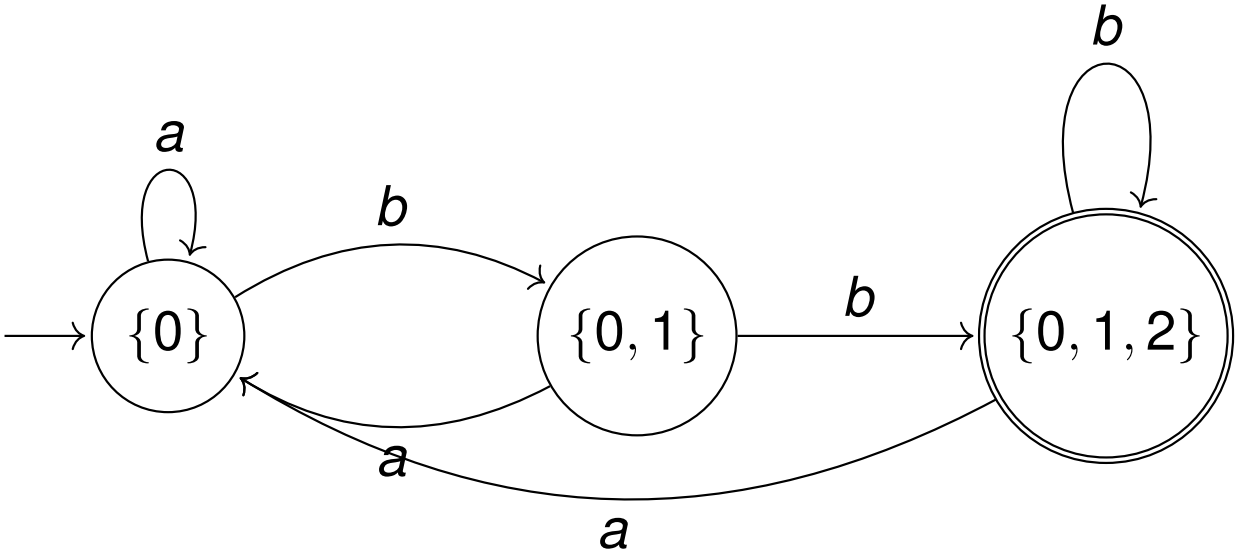
    On fait l'union des suites possibles des états de la source pour le caractère x
  - Etats terminaux : contiennent un état terminal de F
- En pratique on part de l'état initial, puis pour chaque entrée on calcule l'état atteignable, on traite ensuite de la même manière tous les états ainsi engendrés (au plus 2^n s'il y a n états dans l'automate initial).

Déterminisation d'un automate synchrone : exemple

	a	b
0	0	0, 1
1		2
2		

$q_0 = 0, F = \{2\}$





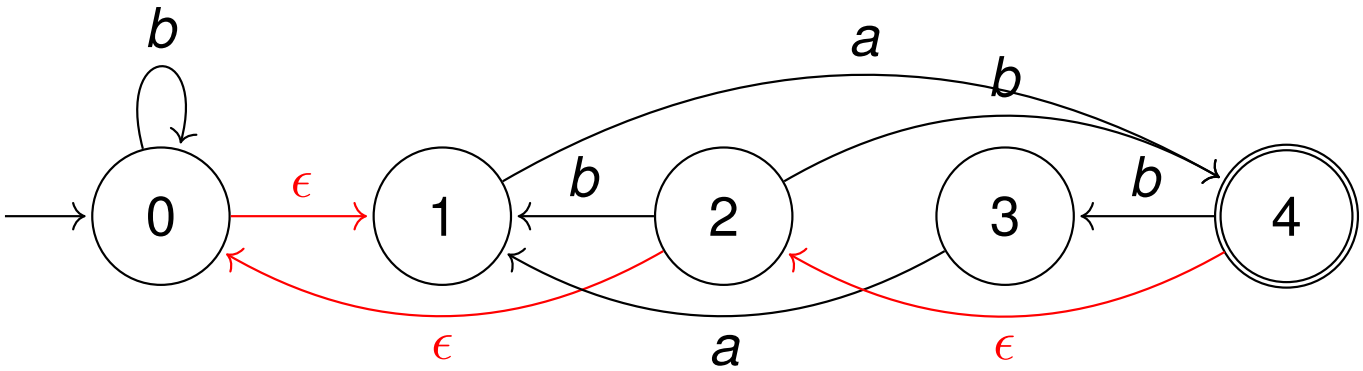
Cas d'un automate asynchrone

- Présence de ϵ-transition
- Possibilité de passer d'un état à l'autre de manière "silencieuse" (sans entrée)
- On associe à un état tous ceux atteignables par une ou plusieurs ϵ-transitions
- Définition de l'ϵ-cloture d'un état $q \in Q$: l'ensemble des états q' pour lesquels il existe un chemin de q à q' dont la trace est le mot vide ϵ

ϵ-cloture : exemple

	a	b	ϵ
0		0	1
1	4		
2		1, 4	0
3	1		
4		3	2

$q_0 = 0, F = \{4\}$



Etat	$\epsilon -$ cloture
0	$\{0, 1\}$
1	$\{1\}$
2	$\{0, 1, 2\}$
3	$\{3\}$
4	$\{0, 1, 2, 4\}$

Déterminiser un automate asynchrone

1. Calculer l'ϵ-cloture de chacun des états
2. Construire l'automate déterministe
   - Etat initial : l'ϵ-cloture de q_0
   - Transitions :

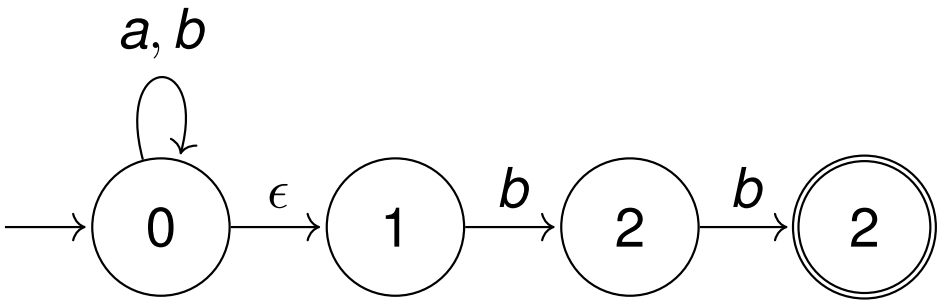
     $$\Delta(\{s_1, \ldots, s_j\}, x) = \epsilon - \text{cloture}(\delta(s_1, x) \cup \ldots \cup \delta(s_j, x))$$

     On fait l'union des ϵ-clotures des suites possibles des états de la source pour le caractère x
   - Etats terminaux : contiennent un état terminal de F

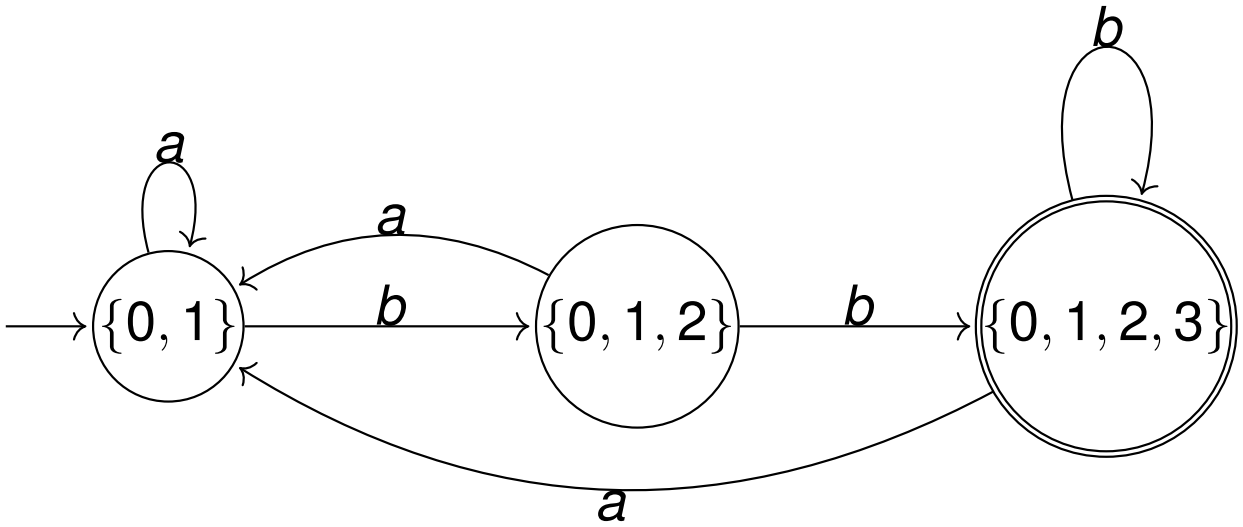
Déterminisation d'un automate asynchrone : exemple

	a	b	ϵ
0	0	0	1
1		2	
2		3	
3			

$q_0 = 0, F = \{3\}$

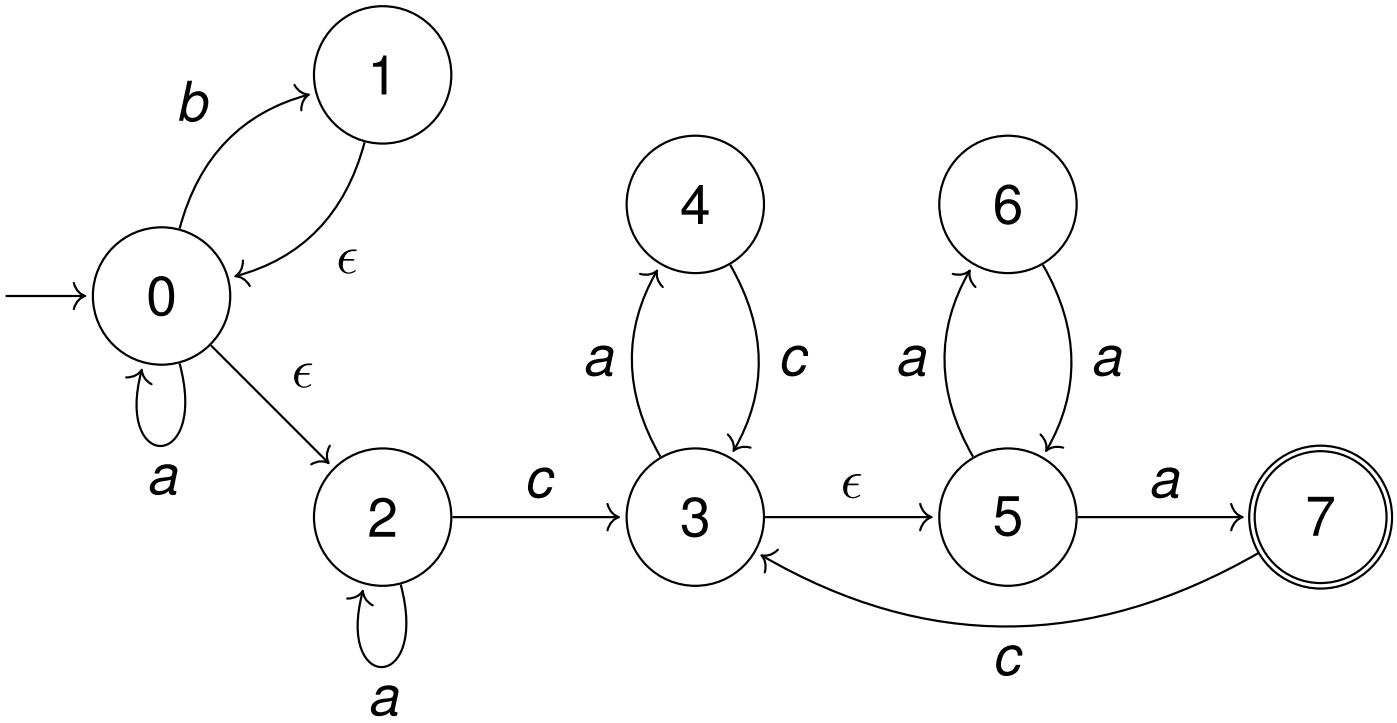


	$\epsilon-$clot
0	0, 1
1	1
2	2
3	3



Exercice

Déterminiser l'automate suivant :



Solution

Exercice

Soit $A = a, b$ un alphabet.
Construire un automate fini déterministe qui reconnaît le langage des mots sur A qui contiennent un nombre pair de lettres a et un nombre impair de lettres b.

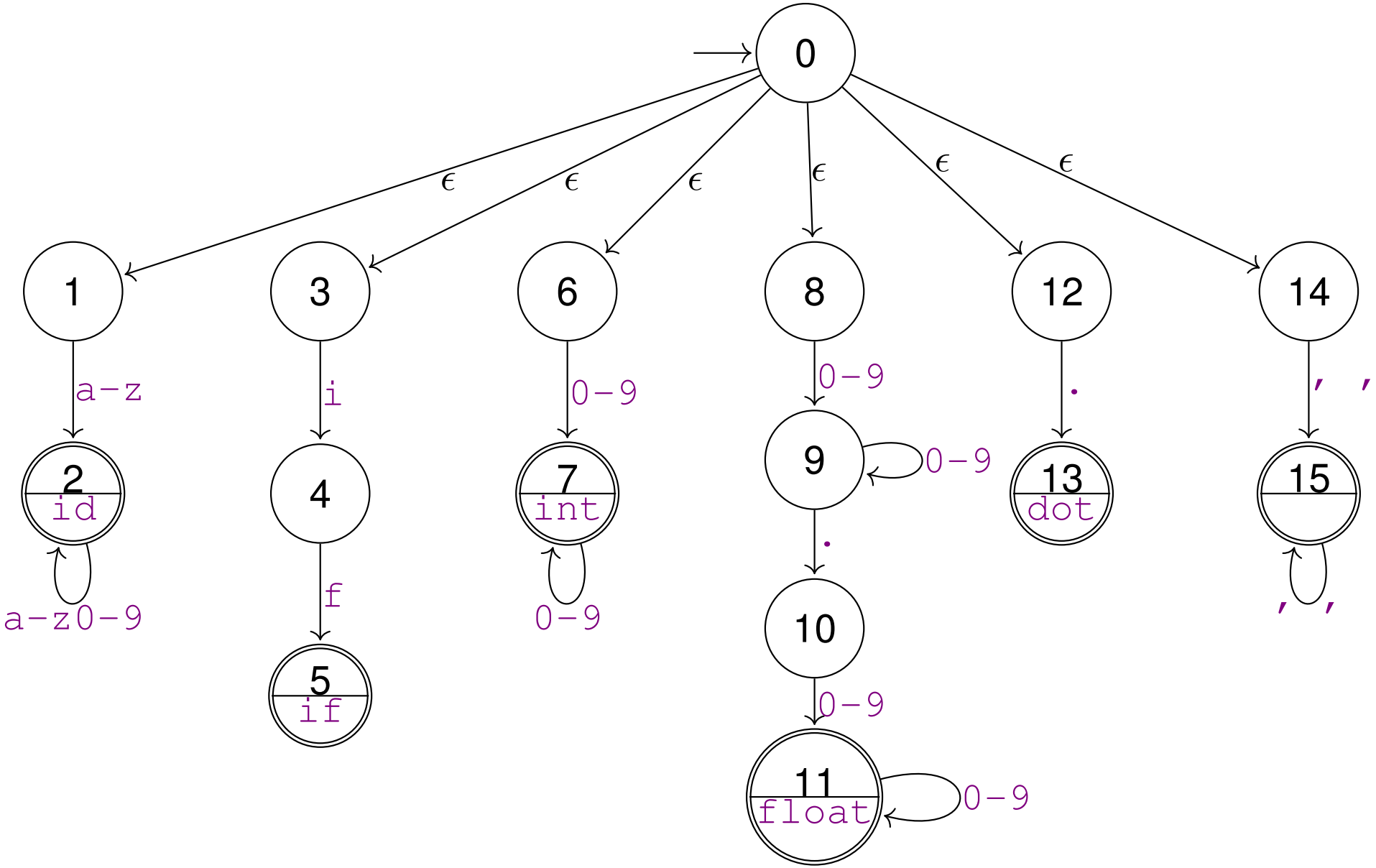
Solution

Retour sur l'analyse lexicale

```
letter = [a-z]
digit = [0-9]
id = letter ( letter | digit )*
if = "if"
int = ( digit )+
float = ( digit )+ "." ( digit )+
dot = "."
bl = (" ")+
```

- On construit un automate unique non-déterministe à partir des automates de chaque expression régulières
- Les états finaux sont distincts et produisent possiblement une sortie ici : `id`, `if`, `int`, `float`, `dot`, les blancs sont ignorés

Automate non déterministe associé



Analyse lexicale : classes de caractères et mots-clés

- Au niveau de l'alphabet d'entrée on peut considérer équivallents tous les caractères qui jouent le même rôle dans les actions (classes de caractères)
  - Par exemple $0 - 9$
  - L'alphabet devient alors celui des classes, cela diminue la taille de la table
- Les mots clés font partie du langage associé à l'expression régulière des identifiants
- Rendre déterministe un automate union des identifiants et des mots-clés multiplie le nombre d'états et le nombre de classes de caractères
- On utilise souvent une seule règle et une table de hachage qui contient les mots clés et les léxèmes associés
- Au niveau de l'action, on vérifie si l'identifiant reconnu est ou non dans la table et on renvoie le léxème approprié

A savoir

- Connaître l'équivalence entre langages réguliers et reconnaissables
- Construire l'automate associé à une expression régulière
- Calculer les ϵ-clotures des états d'un automate asynchrone
- Rendre un automate (synchrone ou asynchrone) déterministe

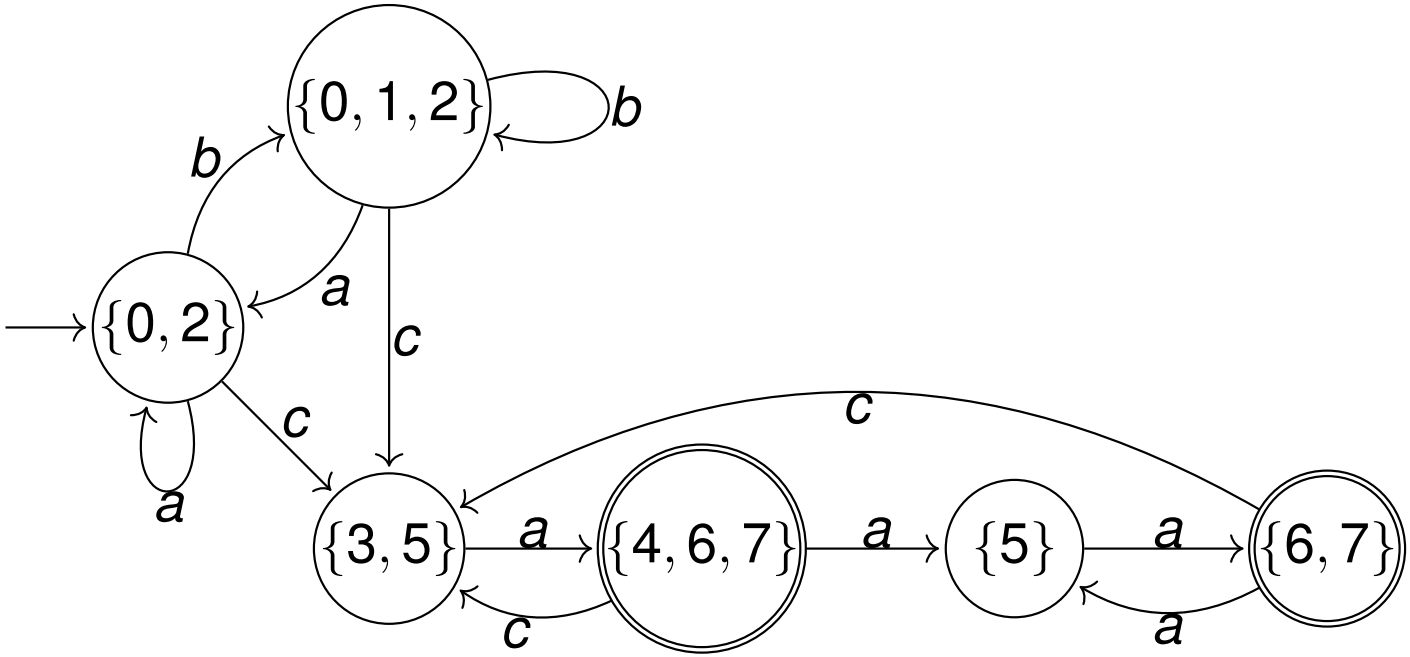
Solution

Calcul de l'ϵ-clotûre

0	0, 2
1	1, 0, 2
2	2
3	3, 5
4	4
5	5
6	6
7	7

1. Etat initial $\{0, 2\}$: transitions possibles a, b, c
   - a : état $\{0, 2\}$
   - b : état $\{0, 1, 2\}$
   - c : état $\{3, 5\}$
2. Etat $\{0, 1, 2\}$: mêmes transitions que $\{0, 2\}$
3. Etat $\{3, 5\}$: transition possible a vers $\{4, 6, 7\}$
4. Etat $\{4, 6, 7\}$: transitions possibles a, c
   - a : état $\{5\}$
   - c : état $\{3, 5\}$
5. Etat $\{5\}$: transition possible a vers $\{6, 7\}$
6. Etat $\{6, 7\}$: transitions possibles a, c
   - a : état $\{5\}$
   - c : état $\{3, 5\}$
7. Etats terminaux : $\{6, 7\}$ et $\{4, 6, 7\}$

Solution



Retour

Solution

Reconnaître qu'un mot contient un nombre pair de a et un nombre impair de b.

- Le nombre de a et de b dans un mot est soit pair soit impair
- 4 situations possibles, chacune correspondant à un état de l'automate
  1. $(0, 0)$: nombres pairs de a et de b (initial)
  2. $(1, 1)$: nombres impairs de a et de b
  3. $(0, 1)$: nombre pair de a et impair de b (final)
  4. $(1, 0)$: nombre pair de a et impair de b (final)

Retour

Analyse lexicale : compléments

Propriétés des langages réguliers

- L'union et la concaténation de deux langages réguliers est un langage régulier
- L'étoile d'un langage régulier est un langage régulier
- Le complémentaire d'un langage régulier est un langage régulier
- L'intersection de deux langages réguliers est un langage régulier

Exercices

- Montrer que le complémentaire d'un langage régulier est un langage régulier
- Montrer que l'intersection de deux langages réguliers est un langage régulier

Solution

Exemple

Soit L_1 l'ensemble des mots qui commencent par ab et L_2 l'ensemble des mots qui terminent par bb.

- Construire des automates déterministes pour reconnaître ces langages, leur complémentaire et leur intersection.

Solution

Décision

- Soit e une expression régulière, donner un algorithme pour déterminer si le langage engendré est vide.
- Soit deux langages réguliers L_1 et L_2, donner un algorithme pour déterminer si $L_1 \subseteq L_2$.

Solution

Equivalence entre automates

- Le même langage peut être décrit par différentes expressions régulières et différents automates.
- Est-il possible de trouver un automate *canonique* ?
- Peut-on reconnaître simplement l'équivalence entre deux automates ?

Equivalence entre états

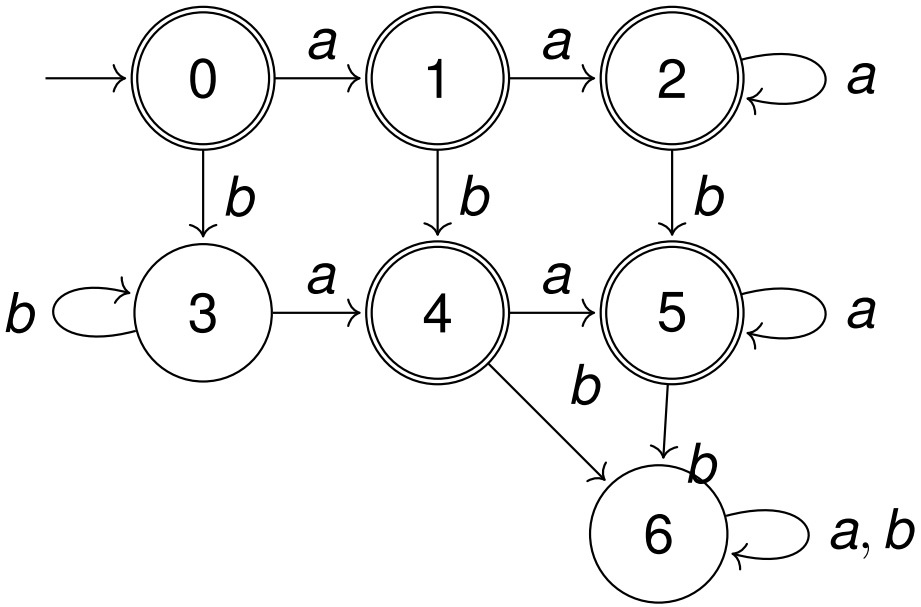
- On introduit une relation d'équivalence entre états.
- Deux états p et q sont équivalents, s'il est possible de reconnaître les mêmes mots à partir de chacun de ces états,
  c'est-à-dire que s'il existe un chemin de l'état p jusqu'à un état final dont la trace est le mot m alors il existe également un chemin de l'état q jusqu'à un état final (qui peut être différent) dont la trace est le même mot m.

Equivalence entre états : calcul

- Pour calculer les états équivalents, on calcule de manière itérative pour $k \in \mathbb{N}$ les états équivalents en k-étapes qui sont ceux à partir desquels on reconnait les mêmes mots de longueur inférieure ou égale à k
  - On part d'un automate fini, déterministe et complet : (A, Q, i, F, δ)
  - Initialement ($k = 0$) on a deux classes : les couples d'états finaux (reconnaissent le mot vide) et les couples d'états qui ne sont pas finaux et donc qui ne reconnaissent pas le mot vide.
  - On suppose que l'on sait reconnaître les états qui sont équivalents pour des mots de longueur inférieure ou égale à k ($p \equiv_k q$),
  - Pour tous les couples (p, q) tels que $p \equiv_k q$, on aura que $p \equiv_{k+1} q$ si et seulement si $\delta(p, a) \equiv_k \delta(q, a)$ pour toutes les lettres de l'alphabet $a \in A$,
  - A une certaine étape, on aura $p \equiv_k q$ si et seulement si $p \equiv_{k+1} q$ pour tout couple d'états et on en déduit $p \equiv_k q$ si et seulement si $p \equiv q$

Automate minimal

- Si à partir de deux états de l'automate on peut reconnaître les mêmes mots alors ces deux états peuvent être fusionnés.
- La relation $\equiv$ d'équivalence entre états est une relation d'équivalence
- On garde un seul état par classe d'équivalence



Solution

Compléments sur les langages réguliers

Lemme de l'étoile

- Comment déterminer si un langage donné L est rationnel ou pas ?
  - Rationnel : trouver une expression régulière ou un automate qui correspond au langage
  - Non rationnel : utiliser le lemme d'itération (ou lemme de l'étoile)

Proposition (Lemme de l'étoile)

Soit un langage L régulier, il existe un entier N qui ne dépend que de L, tel que tout mot $m \in L$ de longueur plus grande que N se décompose en $m = xyz$ avec

- $y \neq \epsilon$
- $|xy| \leq N$
- *pour tout entier naturel $n \in \mathbb{N}$ on a $m = xy^n z \in L$*

Preuve: *La preuve vient de l'existence d'une boucle dans tout chemin qui permet de reconnaître le mot m de longueur strictement supérieure au nombre d'états de l'automate.* □

Langages non réguliers

Proposition

Le langage $L = \{a^n b^n | n \in \mathbb{N}\}$ n'est pas régulier.

Preuve:

- *Si le langage était régulier, soit N la constante du lemme d'itération*
- *Soit $m = a^N b^N$ on a $m = xyz$ mais comme $|xy| \leq N$ on a $y = a^p$ avec $p > 0$*
- *On aurait $xy^2z = a^{N+p}b^N \in L$ ce qui est contradictoire.*

□

Le même argument permet de montrer que le langage formé des mots qui ont le même nombre de a et de b n'est pas régulier.

Compléments sur les langages réguliers

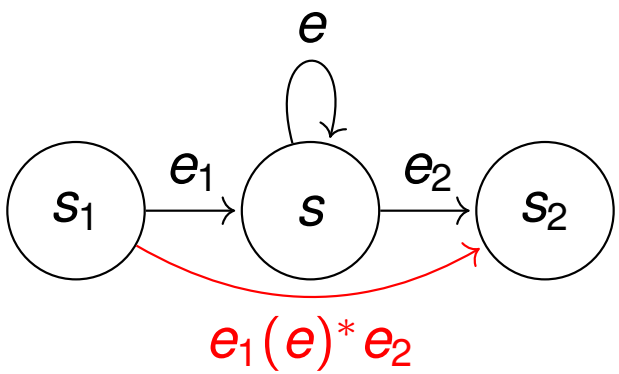
Associer une expression régulière à un automate fini

- Il est assez élémentaire de construire un automate (non-déterministe) à partir d'une expression régulière
- Comment faire l'opération inverse de reconstruction de l'expression régulière à partir de l'automate ?
  - Plusieurs algorithmes possibles

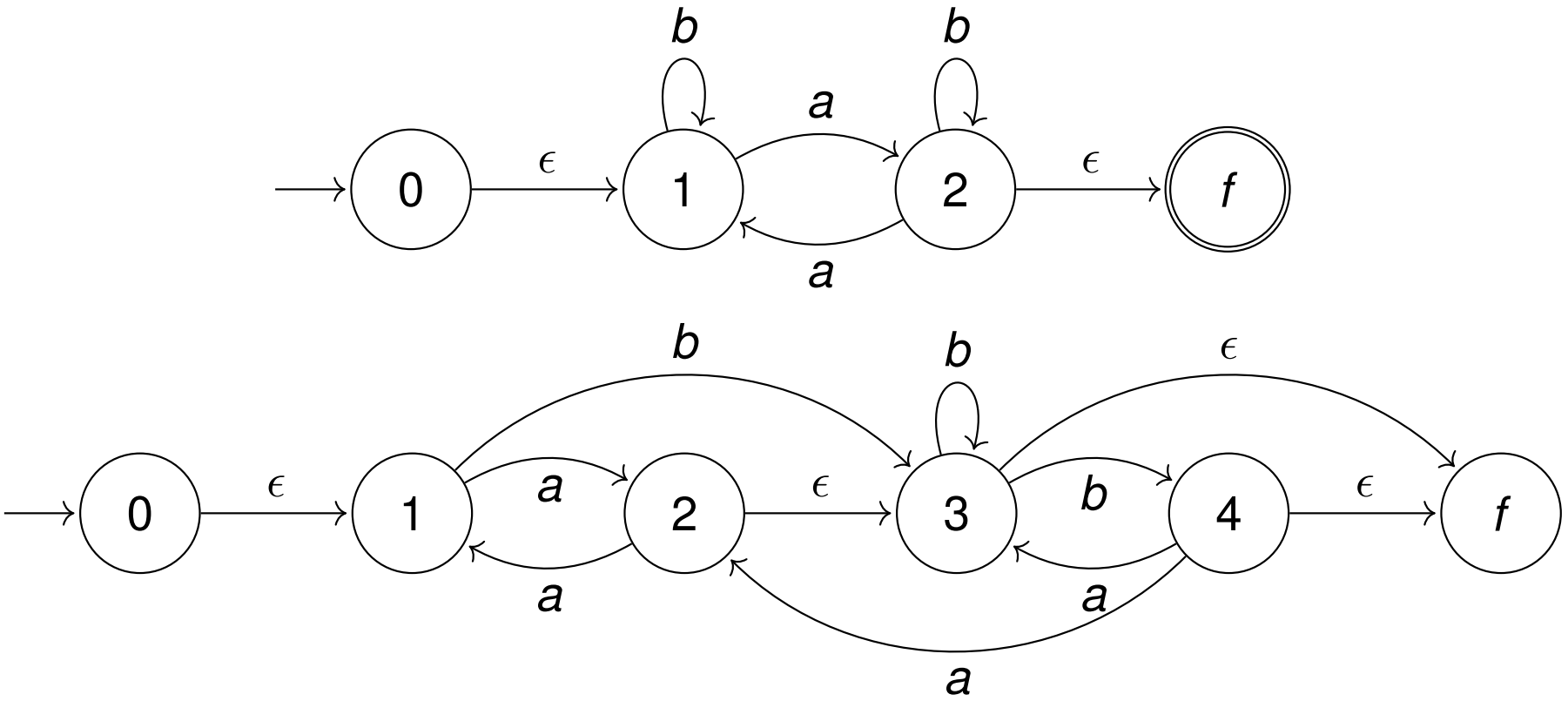
Elimination des états : idée

- On part d'un automate non-déterministe avec un seul état initial sans transition entrante et un seul état final sans transition sortante (on peut toujours ajouter un état initial i et un état final f avec des ϵ-transitions qui les relient à l'état initial et aux états terminaux de l'automate de départ.
- On va transformer le graphe en étiquetant les arètes non plus par des lettres de l'alphabet mais par des expressions régulières.
- Initialement les arètes $a \in A$ ou ϵ peuvent se voir comme des expressions régulières.
- A chaque étape, on supprime un état (qui n'est ni l'état initial, ni l'état final) jusqu'à ce qu'il ne reste plus que l'état initial et l'état final.
- Lorsqu'on supprime un état on met à jour les couples d'arêtes entrante et sortante sur cet état en tenant compte également d'une éventuelle boucle
- Lorsqu'il ne reste que l'état initial et l'état final, l'arète entre ces deux états est une expression régulière qui correspond au langage reconnu.

Suppression d'un état



Exemples



Solution

Compléments sur les langages réguliers

Solution complémentaire et intersection

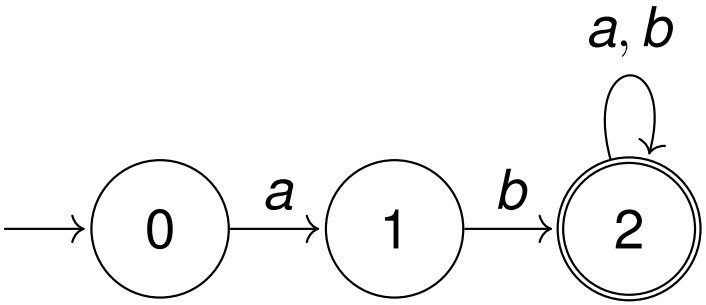
- Complémentaire
  1. Prendre un automate déterministe et le compléter avec des entrées pour toutes les entrées à l'aide d'un état puits
  2. Prendre comme états terminaux de l'automate du complémentaire tous les états qui ne sont pas terminaux dans l'automate initial (en particulier l'état puits)
- Intersection de deux langages reconnaissables
  1. Construire un automate déterministe pour chacun des langages $\mathcal{A}_i \stackrel{\text{def}}{=} (S_i, 0_i, F_i, T_i)$ $(i = 1, 2)$
  2. L'automate pour l'intersection a pour état des couples (s_1, s_2) avec $s_i \in S_i$
  3. Etat initial : $(0_1, 0_2)$
  4. Transition : $((s_1, s_2), a, (t_1, t_2))$ si et seulement si $(s_i, a, t_i) \in T_i$
  5. Etat final : (s_1, s_2) avec $s_i \in F_i$

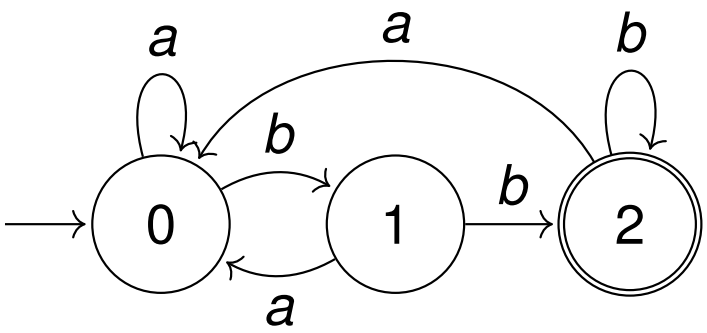
Retour

Exemple : complémentaire et intersection

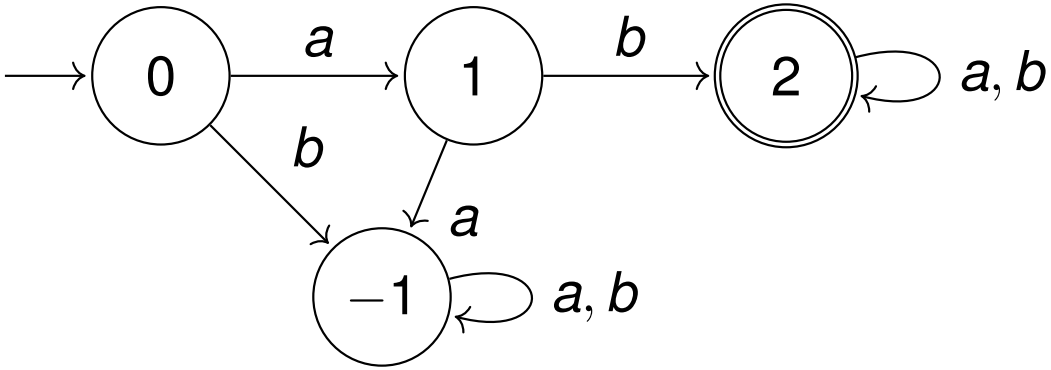
L_1 : mots qui commencent par ab et L_2 : mots qui terminent par bb.

1. Automates déterministes pour les deux langages



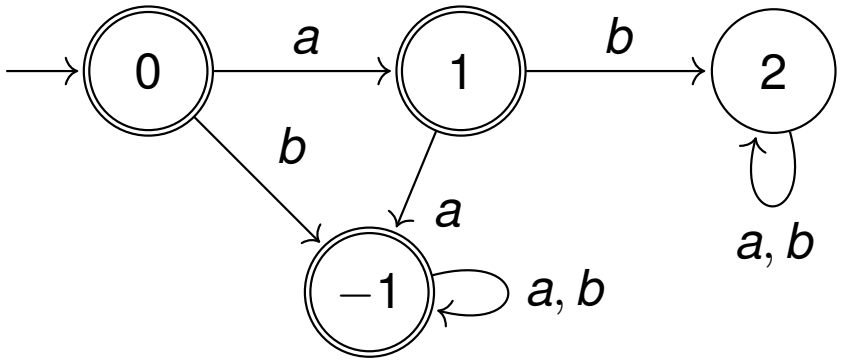


2. Automates complets associés (ajout état puits et transitions pour L_1, L_2 complet)

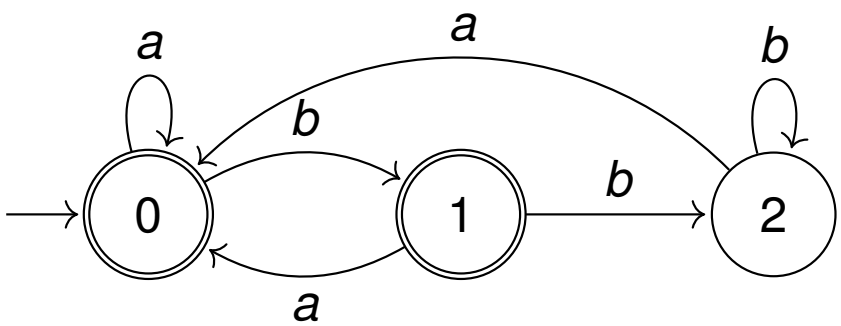


Exemple (suite)

3. Automates des langages complémentaires (inversion des états terminaux)
   - $A^* \setminus L_1$

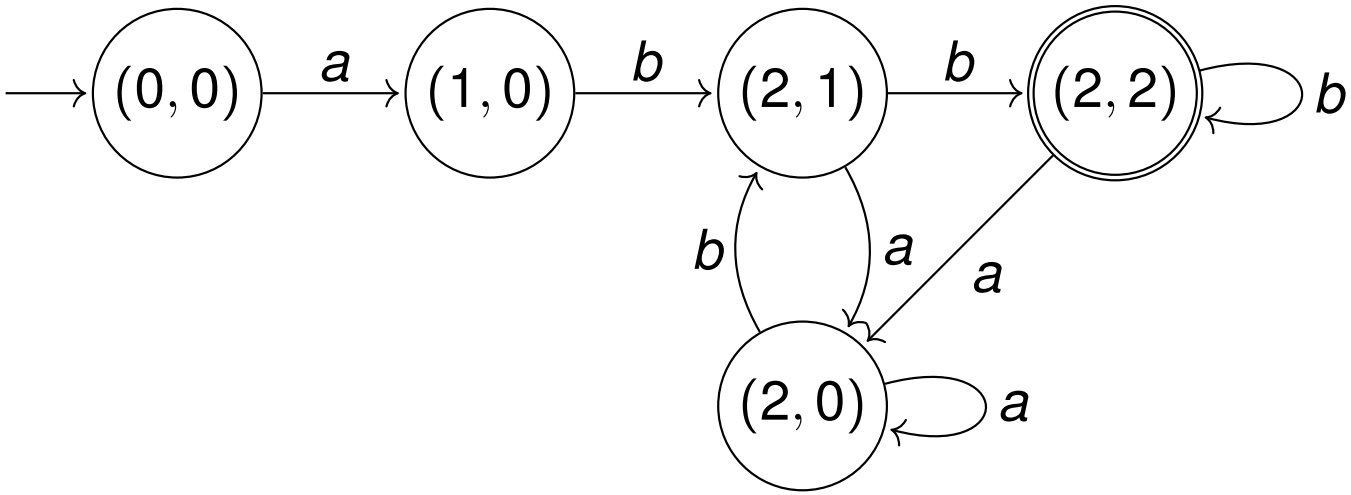


   - $A^* \setminus L_2$



Exemple (suite)

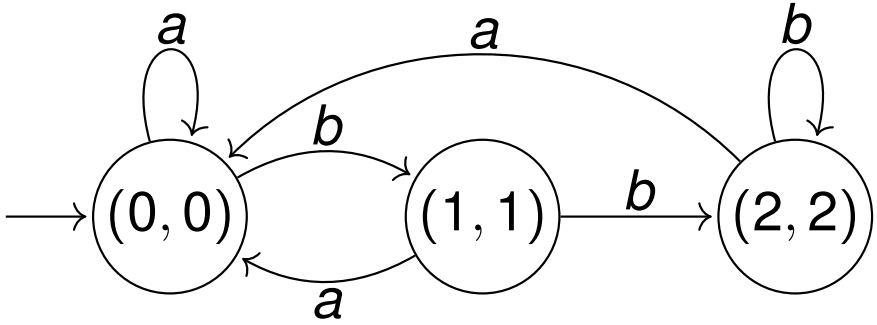
4. Automate produit pour reconnaître l'intersection $L_1 \cap L_2$



Retour

Décision

- Un automate reconnaît le langage vide s'il n'y a pas de chemin entre l'état initial et un état final.
  Exemple l'intersection du langage L_2 et de son complémentaire, les deux automates ont les mêmes transitions, les seuls états accessibles sont $(0,0)$ (état initial), $(1,1)$ et $(2,2)$ mais aucun n'est teminal.



- Soit deux langages réguliers L_1 et L_2,
  - $L_1 \subseteq L_2$ si et seulement si tous les éléments de L_1 sont dans L_2
  - si et seulement si il n'y a pas d'élément de L_1 qui n'est pas dans L_2
  - si et seulement si l'intersection de L_1 et du complémentaire de L_2 est vide.

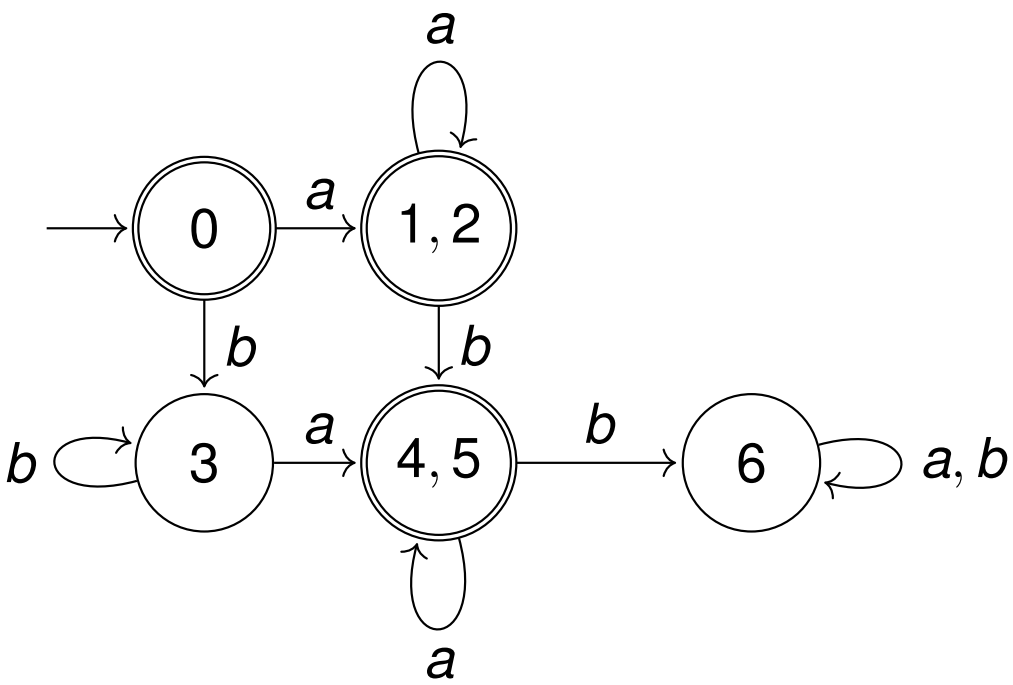
Retour

Solution : Automate minimal

- Equivalence

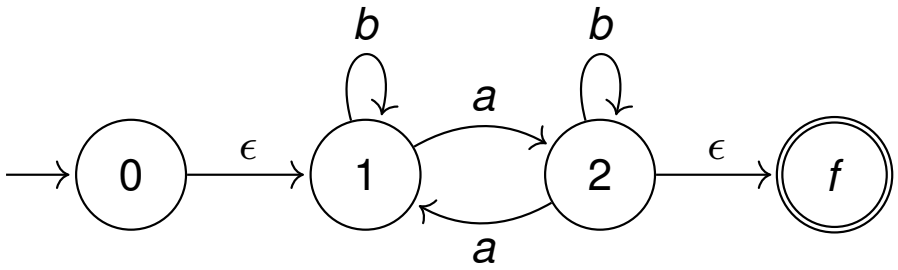
$\equiv_0$	$\{0,1,2,4,5\}\{3,6\}$	états terminaux
$\equiv_1$	$\{0,4,5\}\{1,2\}\{3\}\{6\}$	$0 \xrightarrow{b} 3, 1 \xrightarrow{b} 4 \ldots$
$\equiv_2$	$\{0\}\{4,5\}\{1,2\}\{3\}\{6\}$	$0 \xrightarrow{b} 3, 4 \xrightarrow{b} 6$
$\equiv_3$	$\{0\}\{4,5\}\{1,2\}\{3\}\{6\}$	

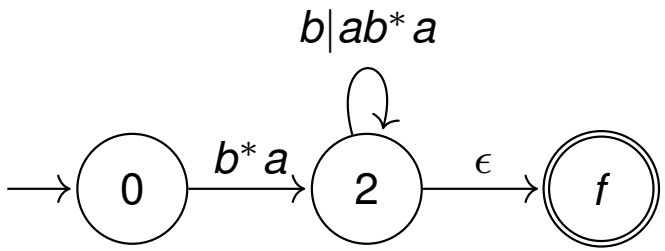
- Automate minimisé :

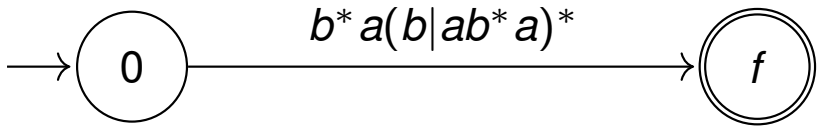


Retour

Expression régulière associée à un automate







Retour

Préparation en vue du partiel

Phases de l'interprétation des langages

- Phases d'analyse lexicale, syntaxique, sémantique et évaluation (entrée, sortie)
- Reconnaître le positionnement des erreurs

Vocabulaire et opérations sur les langages

- alphabet, caractère, mot, langage
- opérations sur les mots : concaténation, puissance
- opérations sur les langages : union, intersection, complémentaire, concaténation, puissance, étoile
- questions de cours, savoir dire si un mot appartient à un langage décrit par des opérations

Expressions régulières pour les unités lexicales

- Définition récursive des expressions régulières sur un alphabet A

$$\epsilon \quad a \text{ (si } a \in A) \quad e_1 \mid e_2 \quad e_1 e_2 \quad e^*$$

- Syntaxe pour les expressions régulières étendues
  - Classe de caractères : représente des ensembles de caractères inclusif `[a-z0-9]` ou exclusif (tout sauf) `[^ a-z0-9]`
  - Opérations étendues : e^+, $e?$
  - Précédences et parenthèses : (e)
- Langage associé à une expression régulière
- Stratégie de reconnaissance à partir d'expressions régulières dans un texte

Exemples de questions

- Savoir définir un concept par récurrence sur la structure de l'expression régulière : calculer `contient-vide`(e) `vrai` si et seulement si $\epsilon \in L(e)$

$$
\begin{array}{ll}
\texttt{contient-vide}(\epsilon) & = \texttt{vrai} \\
\texttt{contient-vide}(a) & = \texttt{faux} \\
\texttt{contient-vide}(e_1 \mid e_2) & = \texttt{contient-vide}(e_1) \text{ ou } \texttt{contient-vide}(e_2) \\
\texttt{contient-vide}(e_1 e_2) & = \texttt{contient-vide}(e_1) \text{ et } \texttt{contient-vide}(e_2) \\
\texttt{contient-vide}(e^*) & = \texttt{vrai}
\end{array}
$$

- Expression régulière correspondant à un langage donné :
  - nombre flottant en java (partiel) : suite de chiffres (entre 0 et 9), suivie du symbole `.` suivi d'une suite de chiffres (la suite avant le point ou celle après le point doit être non vide) suivi de manière optionnelle par un exposant qui est composé d'un indicateur (`e` ou `E`) d'un signe (`+` ou `-`) optionnel et d'une suite non vide de chiffres.

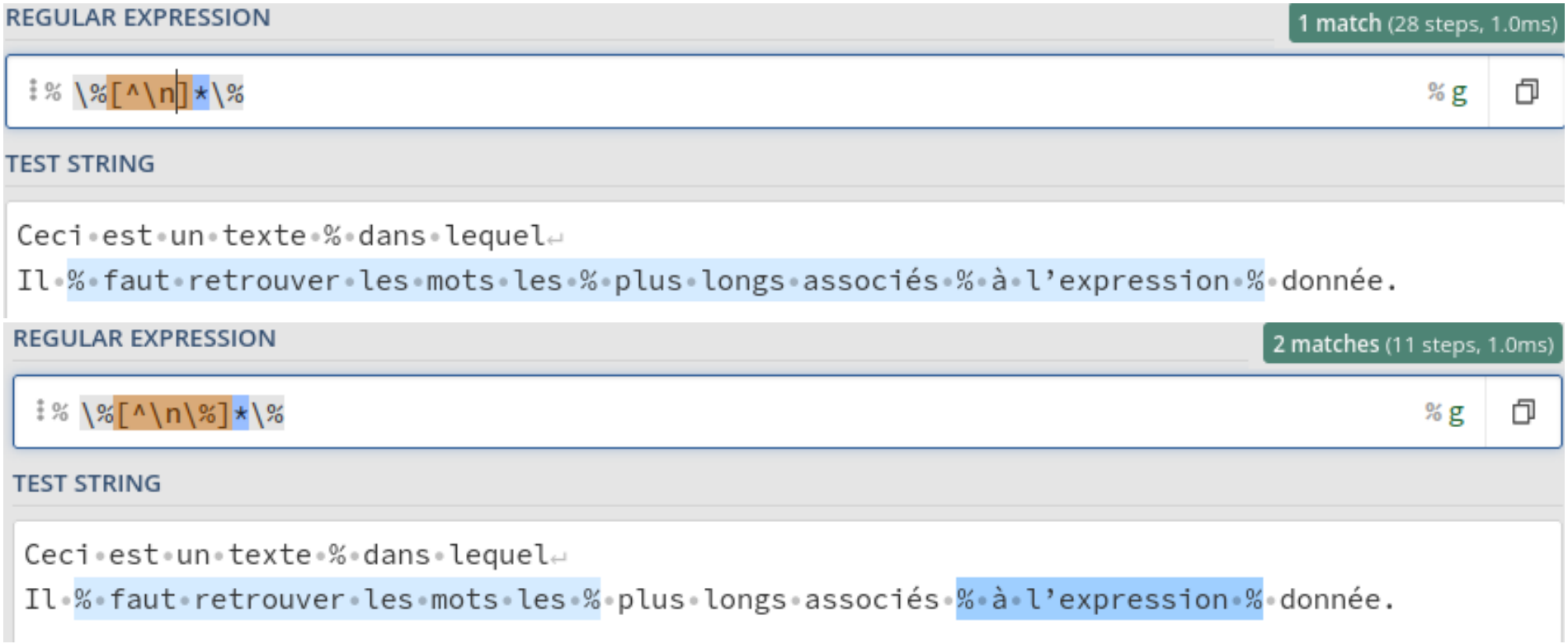
Exemples de questions (suite)

- Soit l'expression régulière `%[^\n]*%`,
  quelle(s) partie(s) du texte suivant est reconnue par un analyseur lexical comme correspondant à cette expression.

  Ceci est un texte % dans lequel
  Il % faut retrouver les mots les % plus longs associés % à l'expression % donnée.

- Même question avec l'expression régulière `%[^\n%]*%`,

Solution : reconnaissance lexicale



Automates finis

- Définition
- Vocabulaire : automates déterministes, complets, (a)synchrones
- Répondre à la question si un mot est reconnu ou non par un automate
- Construire un automate correspondant à un langage donné
- Opérations sur les automates (union, concaténation, étoile, complémentaire, intersection)
- Calculer des ϵ-clotures, rendre un automate déterministe
- Enoncé du théorème de Kleene : langages réguliers = langages reconnaissables, exemple de langage non régulier

Analyse syntaxique–Grammaire

Analyseurs syntaxiques

Où en est-on ?

- Objectif : aller du programme source (suite de caractères) au résultat de l'évaluation
- Etapes d'analyse : comprendre la suite de caractères comme un programme structuré qui respecte des règles de constructions
- Etapes de synthèse : transformer le programme pour calculer le résultat
  - compilation : transformation vers le langage machine puis exécution
  - interprétation : évaluation du programme sur son entrée

Etapes d'analyse

- partie lexicale : la suite de caractères est transformée en une suite d'unités lexicales (associées ou non à des valeurs)
  - constantes
  - identifiants (variables, noms de fonctions, modules...)
  - mots clés (identifiants spéciaux)
  - symboles pour des opérations (opérateurs arithmétiques, logiques, affectations, tableaux, listes, blocs...)
- les unités lexicales sont décrites par des expressions régulières et reconnues par des automates finis
- partie syntaxique : la suite d'unités lexicales est transformée en un arbre de dérivation qui suit les règles de grammaire et permet de construire récursivement un arbre de syntaxe abstraite qui représente la *structure logique* du programme.
- description par une grammaire (hors contexte), reconnaissance par un automate à pile.
- partie sémantique : transformer l'arbre de syntaxe abstraite pour préparer le calcul du résultat.

Analyse syntaxique : choix de la grammaire

- Il existe de nombreuses grammaires qui reconnaissent le même langage
- Du choix de la grammaire dépend
  - l'efficacité de la reconnaissance (trouver rapidement la suite de règles à appliquer)
  - la facilité de construction de l'arbre de syntaxe abstraite à partir de la dérivation
    - Les grammaires en forme normale de Chomsky par exemple sont très éloignées de la structure logique du programme
- Exemple pour ce cours : expressions arithmétiques avec addition, multiplication, constantes entières et expressions parenthésées

Grammaire expressions arithmétiques simples

- Unités lexicales : `CTE` (constantes entières), `LP` et `RP` (parenthèses gauche et droite) `PLUS` (addition) et `MULT` (multiplication)
- Grammaire intuitive :

$$E ::= \texttt{CTE} \mid \texttt{LP}\ E\ \texttt{RP} \mid E\ \texttt{PLUS}\ E \mid E\ \texttt{MULT}\ E$$

- Proche de l'arbre de syntaxe abstraite pour les expressions

```
type oper = Plus | Mult
type expr = Cte of int
          | Binop of oper * expr * expr
```

- Impropre à l'analyse car ambiguë : `CTE PLUS CTE MULT CTE` peut représenter deux arbres différents

Grammaire non ambiguë

- les ambiguïtés d'une grammaire d'un langage de programmation doivent être levées :
  - modifier la grammaire
  - expliciter des règles de priorité
- reformulation de la grammaire avec une priorité plus forte de la multiplication sur l'addition ($3 + 5 * 4 = 3 + (5 * 4)$):
  - une expression est l'addition d'un ou plusieurs termes multiplicatifs
  - un terme multiplicatif est la multiplication d'un ou plusieurs facteurs simples
  - un facteur simple est une constante ou une expression parenthésée
- plusieurs opérations de même nature : $(3 * 5 * 8)$: fixer l'ordre d'évaluation
  - résultat indifférent pour l'addition et la multiplication mais pas pour la soustraction ou la division
  - convention : associativité à gauche $(3 + 5 + 4 = (3 + 5) + 4)$

Grammaire expressions arithmétiques non-ambiguë

$$E \coloneqq \texttt{CTE} \mid \texttt{LP}\ E\ \texttt{RP} \mid E\ \texttt{PLUS}\ E \mid E\ \texttt{MULT}\ E$$

transformée en une grammaire non ambiguë :

$$E \coloneqq E\ \texttt{PLUS}\ T \mid T$$
$$T \coloneqq T\ \texttt{MULT}\ F \mid F$$
$$F \coloneqq \texttt{CTE} \mid \texttt{LP}\ E\ \texttt{RP}$$

Impression

Structure utile pour une fonction d'impression intelligente

```
let rec print_expr = function
| Binop (Plus, e1,e2) -> print_expr e1;
                         printf "+"; print_terme e2
| e   -> print_terme e
and print_terme = function
| Binop (Mult, e1,e2) -> print_terme e1;
                         printf "*"; print_facteur e2
| e   -> print_facteur e
and print_facteur = function
  Cte n -> printf "%d" n
| e -> printf "("; print_expr e;  printf ")"
```

Analyseurs

Analyse descendante

- Dérivation gauche (on cherche la règle qui s'applique au non terminal le plus à gauche dans la dérivation)
- Exemple pour la grammaire

$$
\begin{aligned}
E &\coloneqq E\ \texttt{PLUS}\ T \mid T \\
T &\coloneqq T\ \texttt{MULT}\ F \mid F \\
F &\coloneqq \texttt{CTE} \mid \texttt{LP}\ E\ \texttt{RP}
\end{aligned}
$$

et le mot `CTE PLUS CTE MULT CTE`

$$
\begin{aligned}
E &\to E\ \texttt{PLUS}\ T \to T\ \texttt{PLUS}\ T \to F\ \texttt{PLUS}\ T \to \texttt{CTE PLUS}\ T \\
&\to \texttt{CTE PLUS}\ T\ \texttt{MULT}\ F \to \texttt{CTE PLUS}\ F\ \texttt{MULT}\ F \\
&\to \texttt{CTE PLUS CTE MULT}\ F \to \texttt{CTE PLUS CTE PLUS CTE}
\end{aligned}
$$

Analyse LL(1)

- Il n'est utile d'appliquer une règme $X \to m$ que si la prochaine lettre à reconnaître appartient à l'ensemble $\texttt{premier}(m)$ ainsi que dans le cas où $m \to^* \epsilon$ lorsque cette première lettre appartient à l'ensemble $\texttt{suivant}(X)$
- La grammaire est dite LL(1) si le choix de la règle est unique pour un non-terminal et un caractère d'entrée donné.
- Notre grammaire des expressions arithmétiques non-ambiguë

$$E \coloneqq E\ \texttt{PLUS}\ T \mid T$$
$$T \coloneqq T\ \texttt{MULT}\ F \mid F$$
$$F \coloneqq \texttt{CTE} \mid \texttt{LP}\ E\ \texttt{RP}$$

  vérifie $\texttt{premier}(E) = \texttt{premier}(T) = \texttt{premier}(F) = \{\texttt{LP}, \texttt{CTE}\}$ il n'y a donc pas moyen de choisir laquelle des deux règles s'applique pour les non-terminaux E et T.
- La grammaire n'est pas LL(1)

Grammaire LL(1)

- On remplace les règles $E \coloneqq E\ \texttt{PLUS}\ T \mid T$ par

$$E \coloneqq T\ E'$$
$$E' \coloneqq \texttt{PLUS}\ E \mid \epsilon$$

  - Une seule règle pour E : pas d'ambiguité
  - Pour E' :
    - la règle $E' \coloneqq \texttt{PLUS}\ E$ est positionnée pour le caractère PLUS, $\texttt{premier}(\texttt{PLUS}\ E) = \{\texttt{PLUS}\}$
    - la règle $E' \coloneqq \epsilon$ est positionnée pour les caractères de $\texttt{suivant}(E') = \texttt{suivant}(E)$
- On applique la même technique de transformation pour T,

Grammaire LL(1) résultat

on obtient la grammaire suivante qui est LL(1)

$$
\begin{aligned}
S &\coloneqq E\sharp \\
E &\coloneqq T\ E' \\
E' &\coloneqq \texttt{PLUS}\ E \mid \epsilon \\
T &\coloneqq F\ T' \\
T' &\coloneqq \texttt{MULT}\ T \mid \epsilon \\
F &\coloneqq \texttt{CTE} \mid \texttt{LP}\ E\ \texttt{RP}
\end{aligned}
$$

	null	premier	suivant
E	faux	{CTE, LP}	{♯, RP}
E'	vrai	{PLUS}	{♯, RP}
T	faux	{CTE, LP}	{PLUS, ♯, RP}
T'	vrai	{MULT}	{PLUS, ♯, RP}
F	faux	{CTE, LP}	{MULT, PLUS, ♯, RP}

De la table d'analyse LL(1) à l'analyseur

```
let read_expr m =
  let (pop,next) = init_mot m in
  let rec readE () = readT(); readE'()
  and readE' () = let a = next() in
                  (match a with
                   | '+' -> (pop('+'); readE())
                   | ')'|'#' -> ()
                   | _ -> failwith "mot incorrect")
  and readT () = readF(); readT' ()
  and readT' () = let a = next() in
                  (match a with
                   | '*' -> (pop('*'); readT())
                   | ')'|'#'|'+' -> ()
                   | _ -> failwith "mot incorrect")
  and readF () = let a = next() in
                  (match a with
                   | '(' -> (pop('('); readE(); pop(')'))
                   | _ -> if '0'<=a && a <= '9' then pop(a)
                          else failwith "mot incorrect")
  in (readE (); if next()<>'#' then failwith "mot incorrect")
```

Programmation d'analyseur descendant

- Fonctions auxiliaires pour avancer dans l'entrée
  - recherche du caractère suivant (`next`)
  - reconnaissance d'un caractère *a* donné (`pop`(*a*), avance dans le mot)
- La pile des appels récursifs simule la pile de l'automate
- Les grammaires LL(1) s'éloignent de la structure naturelle des programmes

Analyseurs

Générateur d'analyseurs

- Famille `Yacc`
  - pour Java, on utilisera `CUP`
  - CUP : Based Constructor of Useful Parsers (C. Scott Ananian, Frank Flannery, Dan Wang, Andrew W. Appel and Michael Petter. Technische Universität München, Princeton, Georgia Institute of Technology)
  - `http://www2.cs.tum.edu/projects/cup`
- Analyse ascendante (des feuilles vers l'axiome, LALR(1), cf cours LF L3)
- Association d'actions aux règles (pour évaluer ou construire l'arbre de syntaxe abstraite)
- Grammaires "naturelles"
  - possiblement ambiguës mais gestion des conflits par des précédences

CUP : exemple

```
import java_cup.runtime.*;

terminal PLUS, MULT, LP, RP, NL ;
terminal Integer CTE ;
non terminal expr, terme, facteur, ligne ;

start with ligne;
```

Fichier cup : entête

- import de packages Java
- code à insérer dans la classe parser qui sera engendrée, pour la gestion d'erreur ou à utiliser dans les actions

```
action code {: ... :};
parser code {: ... :};
```

- code à exécuter à l'initialisation, pour récupérer le prochain lexème...

```
init with {: ... :};
scan with {: ... :};
```

- Ensemble des terminaux (possiblement associés à des valeurs)

```
terminal classname name1, name2, ...;
terminal name1, name2, ...;
```

- Ensemble des non-terminaux

```
non terminal classname name1, name2, ...;
non terminal name1, name2, ...;
```

- Symbole initial

```
start with N;
```

Mots réservés dans CUP

```
code, action, parser,
terminal, non, nonterminal,
init, scan, with, start,
precedence, left, right, nonassoc,
import, package
```

CUP : grammaire

```
ligne ::=
      expr  NL
;
expr ::=
        expr PLUS terme
      | terme
;
terme ::=
        terme MULT facteur
      | facteur
;
facteur ::=
        CTE
      | LP expr RP
;
```

Interactions entre analyseurs lexical et syntaxique

- C'est l'analyseur syntaxique qui définit les symboles terminaux
- La compilation de l'analyseur syntaxique se fait par la commande

```
java -jar /usr/share/java/java-cup-0.11b.jar
     -package calc Parser.cup
```

  Elle produit deux fichiers (noms modifiables par la directive `class` dans le fichier `cup`):
  - `sym.java` contient la déclaration des symboles terminaux à utiliser dans l'analyseur lexical
  - `parser.java` contient une fonction d'analyse syntaxique qui fait un appel à un analyseur lexical

Fichier `sym.java`

```
/** CUP generated class containing symbol constants. */
public class sym {
  /* terminals */
  public static final int NL = 6;
  public static final int MULT = 3;
  public static final int error = 1;
  public static final int PLUS = 2;
  public static final int LP = 4;
  public static final int CTE = 7;
  public static final int RP = 5;
  public static final int EOF = 0;
  public static final String[] terminalNames = new String[] {
  "EOF",
  "error",
  "PLUS",
  "MULT",
  "LP",
  "RP",
  "NL",
  "CTE"
  };
}
```

Couplage avec l'analyseur lexical

Entête du fichier *Lexer.jflex*

```
package calc;

import java_cup.runtime.*;
import java.util.*;
import static calc.sym.*;

%%
%class Lexer
%unicode
%cup
%line
%column
%yylexthrow Exception
```

Couplage avec l'analyseur lexical

Corps du fichier *Lexer.jflex*

```
NL = \R
BLANC = [ \t\f]
NOMBRE = [0-9]+
%%
{BLANC}    { /* ignore */ }
{NL}       { return new Symbol(NL); }
\+         { return new Symbol(PLUS); }
\*         { return new Symbol(MULT); }
\(         { return new Symbol(LP); }
\)         { return new Symbol(RP); }
{NOMBRE}   { return new Symbol(CTE,Integer.parseInt(yytext())); }
.   { throw new Exception
      (String.format ("Line %d, column %d: illegal character: '%s'\n",
                      yyline, yycolumn, yytext() )); }
```

Actions du lexer

- La bibliothèque `java_cup.runtime` définit une classe générique pour les symboles terminaux
- Un symbole a deux composantes : l'identifiant et la valeur
  - l'identifiant correspond à l'alphabet de la grammaire,
  - l'identifiant est codé par un entier (défini dans le fichier `sym` engendré par `cup`)
- La valeur est un objet java quelconque, avec une valeur `null` pour un terminal qui ne porte pas de valeur (mots-clés, ponctuation . . ...)
- Deux constructeurs : un seul argument (`int`) ou deux arguments (`int`,`Object`)
- La classe de l'objet renvoyé par le lexer doit correspondre à celui déclaré dans le fichier `cup`

Fichiers `parser.java`, `Main.java`

- `parser.java`

```
public class parser extends java_cup.runtime.lr_parser {...}
public parser(java_cup.runtime.Scanner s) {super(s);}
```

- `Main.java`

```
java.io.Reader reader = new java.io.FileReader(file);
Lexer lexer = new Lexer(reader);
parser parser = new parser(lexer);
  try { parser.parse();
      } catch (Exception e) {
        System.out.println("error: " + e.getMessage());
        System.exit(1);      }
```

Ajout d'actions

- Posibilité d'exécuter du code chaque fois qu'une règle est utilisée
- Code java associé à la fin de la règle `{:....:}`
- Possibilité de désigner la valeur d'un terminal en étiquetant le terminal
  - Etiquetage : `CTE : n`,
  - la variable *n* porte la valeur du terminal et est utilisé comme variable Java dans l'action associée
- Code auxilliaire pour les actions dans le préambule

```
parser code {:
void echo (String s) {System.out.print(s);}
void echo (Integer n) {System.out.print(n);}
              :}
```

Analyseur avec actions

```
terminal PLUS, MULT, LP, RP, NL ;
terminal Integer CTE ;
non terminal expr, terme, facteur, ligne ;

start with ligne;

ligne ::=
     expr  NL                {:echo("\n");:}
;
expr ::=
       expr PLUS terme       {:echo("␣+␣");:}
     | terme:e
;
terme ::=
       terme MULT facteur    {:echo("␣*␣");:}
     | facteur
;
facteur ::=
       CTE:n                 {:echo(n); echo("␣");:}
     | LP expr RP
;
```

Résultat

- Test
  - Entrée : `(20*((50+30)*40))*2`
  - Sortie : `20 50 30 + 40 * * 2 *`
- Besoin de controler l'ordre de construction des objets

Associations de valeurs aux non-terminaux

- La classe de la valeur est déclarée avec le terminal
- Les non-terminaux dans la pile viennent avec leur valeur
- Etiquetage des non-terminaux à droite des règles pour désigner la valeur calculée
- La valeur du non-terminal à gauche de la règle est calculée via l'action associée

$$\{: \texttt{RESULT} = \ldots :\}$$

Analyseur avec actions

```
terminal PLUS, MULT, LP, RP, NL;
terminal Integer CTE ;
non terminal String expr, terme, facteur ;
non terminal ligne ;

start with ligne;

ligne ::=
     expr:e  NL                    {: System.out.println(e);  :}
;
expr ::=
       expr:e1 PLUS terme:e2       {: RESULT = e1 + '+' + e2; :}
     | terme:e                     {: RESULT = e; :}
;
terme ::=
       terme:e1 MULT facteur:e2    {: RESULT = e1 + '*' + e2; :}
     | facteur:e                   {: RESULT = e; :}
;
facteur ::=
       CTE:n                       {: RESULT = String.valueOf(n); :}
     | LP expr:e RP                {: RESULT = '(' + e + ')'; :}
;
```

Evaluateur

```
terminal PLUS, MULT, LP, RP, NL;
terminal Integer CTE ;
non terminal Integer expr, terme, facteur ;
non terminal ligne ;

start with ligne;

ligne ::=
     expr:e  NL                    {: System.out.println(e);   :}
;
expr ::=
       expr:e1 PLUS terme:e2       {: RESULT = e1+e2; :}
     | terme:e                     {: RESULT = e; :}
;
terme ::=
       terme:e1 MULT facteur:e2    {: RESULT = e1*e2; :}
     | facteur:e                   {: RESULT = e; :}
;
facteur ::=
       CTE:n                       {: RESULT = n; :}
     | LP expr:e RP                {: RESULT = e; :}
;
```

Gestion des précédences

- `CUP` n'accepte pas des grammaires qui contiennent un trop grand de conflits entre les opérations à réaliser (en lisant un caractère en avance)
- `CUP` offre un mécanisme de résolution de ces conflits par des règles de précédence pour décider quelle règle appliquée prioritairement
- Cela se fait en regroupant les symboles terminaux par niveau de priorité et en les ordonnant de la priorité la plus faible jusqu'à la propriété la plus forte
- Les symboles sont déclarés associatifs à gauche ou à droite ou bien non associatifs (interdisant par exemple de reconnaitre $t == u == v$ comme une expression bien formée)
- Sur les exemples simples d'opérateurs binaires, le comportement obtenu correspond à l'intuition en terme de priorité et d'associativité
- Le fonctionnement général précis nécessite de comprendre les bases de l'analyse LR

Exemples

- Avec une priorité plus importante de $*$ sur $+$ on a que $x+y*z$ se parenthèse $x+(y*z)$ et $x*y+z$ se parenthèse $x*(y+z)$
- Avec une associativité gauche de $+$ sur $-$ de précédence égales $x-y+z$ se parenthèse $(x-y)+z$ et $x+y-z$ se parenthèse $(x+y)-z$
- déclaration `CUP`

```
terminal PLUS, MULT, LP, RP, EQ, NL, SUB, DIV, VA;
terminal Integer CTE ;
nonterminal Integer expr ;
nonterminal lignes, ligne ;

precedence left PLUS, SUB;
precedence left MULT, DIV;

expr ::=
        expr:e1 PLUS expr:e2      {: RESULT = e1+e2; :}
      | expr:e1 SUB expr:e2       {: RESULT = e1-e2; :}
      | expr:e1 MULT expr:e2      {: RESULT = e1*e2; :}
      | expr:e1 DIV expr:e2       {: RESULT = e1 / e2; :}
      | CTE:n                     {: RESULT = n; :}
      | VA                        {: RESULT = varA; :}
```

Analyseurs

Arbres de syntaxe abstraite

- L'évaluation au vol ne peut se faire que dans des cas simples
- Construction dans la grammaire d'une représentation arborescente du programme
- Traitement ensuite dans le langage de programmation (analyse sémantique, évaluation)
- AST/ASA (Abstract Syntax Tree/Arbre de Syntaxe Abstraite)
  voir cours OLA
  - en Java
    - Classe abstraite et sous classes concrètes
    - Interface et implémentation
  - en Ocaml
    - types algébriques avec constructeurs

Implantation des AST

```
package calc;

abstract class Ast {
    public abstract int eval();
}

enum Binop { PLUS, MULT }

class Expr extends Ast {
    final Binop op;
    final Ast f1, f2;
    public Expr(Binop op, Ast f1, Ast f2) {
        this.op = op; this.f1 = f1; this.f2 = f2; }
    public int eval(){switch (op) {
        case PLUS: return (f1.eval() + f2.eval());
        case MULT: return (f1.eval() * f2.eval());
        default: throw new IllegalArgumentException();
        }}}

class Cte extends Ast {
    final int val;
    public Cte(int n) {this.val = n; }
    public int eval() {return val; }
}
```

Construction des AST

```
terminal PLUS, MULT, LP, RP, NL;
terminal Integer CTE ;
non terminal Ast expr, terme, facteur ;
non terminal ligne ;

start with ligne;

ligne ::=
     expr:e  NL                      {: System.out.println(e.eval());   :}
;
expr ::=
       expr:e1 PLUS terme:e2         {: RESULT = new Expr(Binop.PLUS,e1,e2); :}
     | terme:e                       {: RESULT = e; :}
;
terme ::=
       terme:e1 MULT facteur:e2      {: RESULT = new Expr(Binop.MULT,e1,e2); :}
     | facteur:e                     {: RESULT = e; :}
;
facteur ::=
       CTE:n                         {: RESULT = new Cte(n); :}
     | LP expr:e RP                  {: RESULT = e; :}
;
```

Principe de choix des AST

- L'AST doit se construire simplement à partir de la structure de la grammaire
- Attention aux effets de bord dans les actions : l'ordre d'évaluation des actions peut être difficile à prévoir
- L'AST ne garde que ce qui est pertinent pour le calcul :
  - les espaces, commentaires sont éliminés dès l'analyse lexicale
  - ce qui sert à lever les ambiguités de l'écriture linéaire disparait
    - parenthèses
    - plusieurs non-terminaux différents construiront des objets de même nature (expressions, termes, facteurs)

Retour sur mini-python : AST

```
enum Unop { Uneg, Unot }
enum Binop {
  Badd , Bsub , Bmul , Bdiv , Bmod,
  Beq , Bneq , Blt , Ble , Bgt , Bge, // comparaison
  Band , Bor // paresseux
}
/* constantes littérales */
abstract class Constant {
    static final Cnone None = new Cnone();
}
class Cnone extends Constant { }
class Cbool extends Constant {
    final boolean b;
    Cbool(boolean b) { this.b = b; }
}
class Cstring extends Constant {
    final String s;
    Cstring(String s) { this.s = s; }
}
class Cint extends Constant {
    final int i;
    Cint(int i) { this.i = i; }
}
/* expressions */
abstract class Expr { }
class Ecst extends Expr {
    final Constant c;
    Ecst(Constant c) { this.c = c; }
}
```

Mini-python : AST expressions

```
class Ebinop extends Expr {
    final Binop op;
    final Expr e1, e2;
    Ebinop(Binop op, Expr e1, Expr e2) { this.op = op; this.e1 = e1; this.e2 = e2; }
}
class Eunop extends Expr {
    final Unop op;
    final Expr e;
    Eunop(Unop op, Expr e) { this.op = op; this.e = e; }
}
class Eident extends Expr {
    final String s;
    Eident(String s) { this.s = s; }
}
class Eget extends Expr {
    final Expr e1, e2;
    Eget(Expr e1, Expr e2) { this.e1 = e1; this.e2 = e2; }
}
class Ecall extends Expr {
    final String f;
    final LinkedList<Expr> l;
    Ecall(String f, LinkedList<Expr> l) { this.f = f; this.l = l; }
}
class Elist extends Expr {
    final LinkedList<Expr> l;
    Elist(LinkedList<Expr> l) { this.l = l; }
}
```

Mini-python : AST instructions

```
abstract class Stmt { }
class Sif extends Stmt {
    final Expr e;
    final Stmt s1, s2;
    Sif(Expr e, Stmt s1, Stmt s2) { this.e = e; this.s1 = s1; this.s2 = s2; }
}
class Sreturn extends Stmt {
    final Expr e;
    Sreturn(Expr e) { this.e = e; }
}
class Sassign extends Stmt {
    final String s;
    final Expr e;
    Sassign(String s, Expr e) { this.s = s;    this.e = e; }
}
class Sprint extends Stmt {
    final Expr e;
    Sprint(Expr e) { this.e = e; }
}
class Sblock extends Stmt {
    final LinkedList<Stmt> l;
    Sblock() { this.l = new LinkedList<Stmt>(); }
    Sblock(LinkedList<Stmt> l) { this.l = l; }
}
```

Mini-python : AST instructions (suite)

```
class Sfor extends Stmt {
    final String x;
    final Expr e;
    final Stmt s;
    Sfor(String x, Expr e, Stmt s) { this.x = x; this.e = e; this.s = s; }
}
class Seval extends Stmt {
    final Expr e;
    Seval(Expr e) { this.e = e; }
}
// e1[e2] = e3
class Sset extends Stmt {
    final Expr e1, e2, e3;
    Sset(Expr e1, Expr e2, Expr e3) { this.e1 = e1; this.e2 = e2; this.e3 = e3; }
}
```

Mini-python : AST programme

```
class Def {
    final String f;
    final LinkedList<String> l; // arguments formels
    final Stmt s;
    Def(String f, LinkedList<String> l, Stmt s) {
        this.f = f; this.l = l; this.s = s;
    }
}

class File {
    final LinkedList<Def> l;
    final Stmt s;
    File(LinkedList<Def> l, Stmt s) { this.l = l; this.s = s; }
}
```

AST mini-python en Ocaml

```
type ident = string
type unop =
  | Uneg (* -e *)
  | Unot (* not e *)
type binop =
  | Badd | Bsub | Bmul | Bdiv | Bmod     (* + - * / % *)
  | Beq | Bneq | Blt | Ble | Bgt | Bge   (* == != < <= > >= *)
  | Band | Bor (* && || *)
type constant =
  | Cnone
  | Cbool of bool
  | Cstring of string
  | Cint of int
type expr =
  | Ecst of constant
  | Eident of ident
  | Ebinop of binop * expr * expr
  | Eunop of unop * expr
  | Ecall of ident * expr list
  | Elist of expr list
  | Eget of expr * expr (* e1[e2] *)
```

AST mini-python en Ocaml

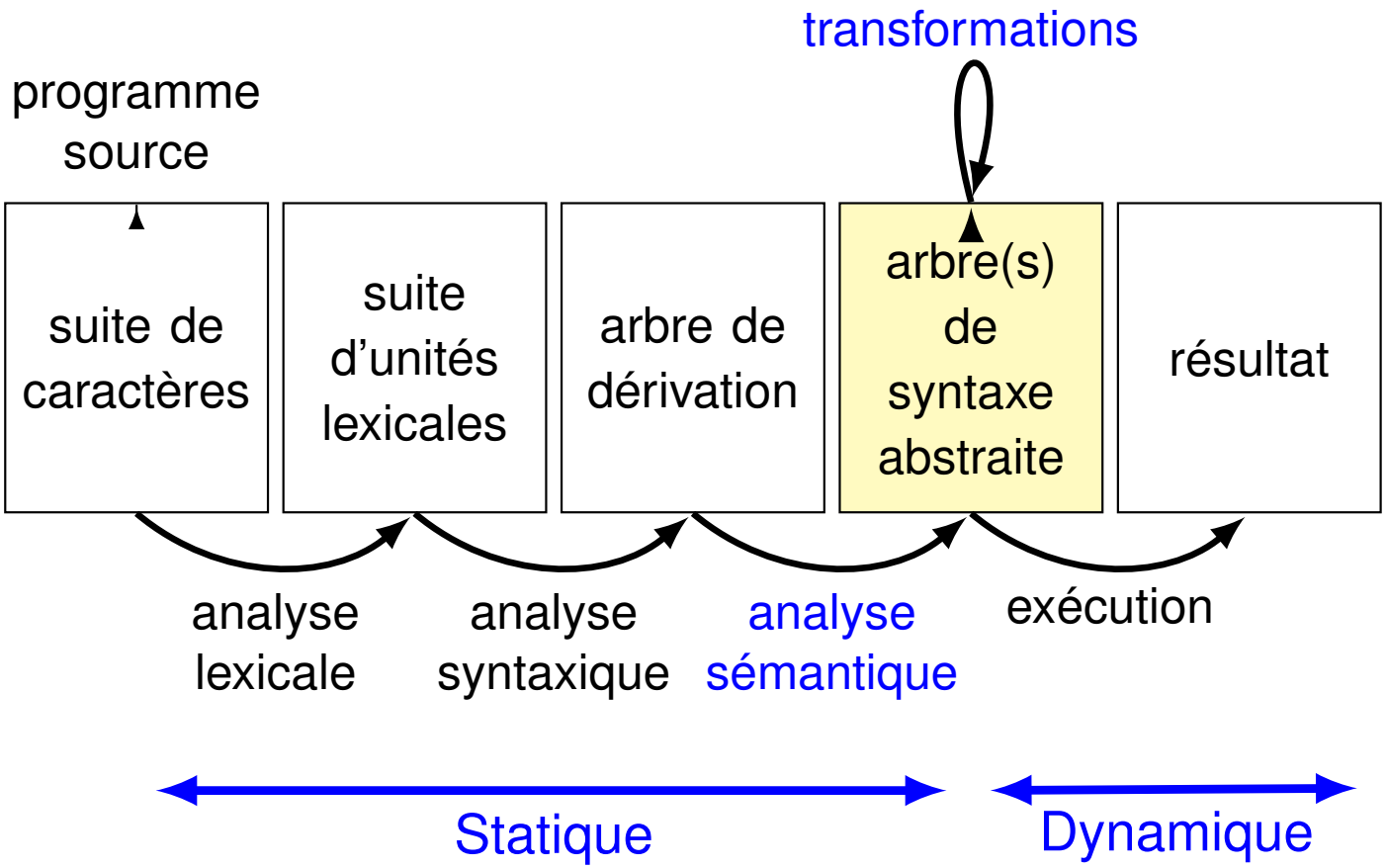
```
type stmt =
  | Sif of expr * stmt * stmt
  | Sreturn of expr
  | Sassign of ident * expr
  | Sprint of expr
  | Sblock of stmt list
  | Sfor of ident * expr * stmt
  | Seval of expr
  | Sset of expr * expr * expr (* e1[e2] = e3 *)
type def = ident * ident list * stmt
type file = def list * stmt
```

A savoir

- Comprendre une grammaire en format BNF
- Construire un arbre de dérivation
- Détecter les ambiguïtés et les résoudre en terme de précédence (expressions arithmétiques, if then else...)
- Ecrire un analyseur descendant avec des fonctions récursives
- Décrire une grammaire dans `CUP` (et au-delà utiliser les outils de la famille Yacc, voir TP)
- Proposer et implanter une structure d'arbre de syntaxe abstraite pertinente par rapport à un langage

Analyse sémantique

Rappel : étapes de l'interprétation



Analyse sémantique

- L'analyse syntaxique a reconnu un programme syntaxiquement bien formé
- Le programme est représenté sous forme d'arbre
- Ce qui n'a pas été vérifié à l'analyse syntaxique, à faire avant l'exécution
  - le bon usage des identifiants (ce qui est utilisé existe)
  - le bon usage des opérations :
    - appliquées au bon nombre d'arguments
    - appliqués à des arguments de bonne nature
- Préparer le choix de représentation en fonction de la nature des données

Rôle de l'analyse sémantique

Entrée : arbre de syntaxe abstraite *brut* issu de l'analyse syntaxique
Sortie :

- un arbre de syntaxe abstraite *décoré*: résultat de la *compréhension du programme*
- une *erreur* si le programme est *incorrect*

Outils théoriques : descriptions sémantiques.
Outils logiciels : programmes récursifs sur les arbres de syntaxe abstraite.

Analyse sémantique

Analyse de portée: principes

- Repérer les utilisations d'objets non déclarés.
- Relier les déclarations d'objets (variables, types, fonctions . . .) et leur utilisation.
- Une fois ce lien réalisé, le sauvegarder pour les autres phases.

Table des symboles

- Des informations doivent être collectées pour chaque identificateur introduit:
  - Nature de l'objet: variable, fonction, type; variable globale, locale, paramètre ...
  - Type d'une variable; signature d'une fonction.
  - Allocation en mémoire (emplacement, taille ...)
- Ces informations serviront à chaque utilisation de l'objet.
- Création d'un lien direct vers les informations pour chaque utilisation.

Arbres de syntaxe annotés

Plusieurs représentations possibles pour lier les utilisations et les déclarations :

- Chaque utilisation est un pointeur sur la déclaration.
- Chaque utilisation est un nom unique ou bien un numéro faisant référence à une *table annexe* qui contient les informations sur les déclarations.
  - Possibilité d'utiliser différentes tables suivant l'étape de compilation.

L'analyse de portée va permettre de construire cette nouvelle représentation de l'arbre.

Vérifier la portée

- **Entrée** : l'arbre de syntaxe abstraite issu de l'analyse syntaxique
- **Sortie** : un nouvel arbre de syntaxe abstraite avec des identificateurs uniques et une table qui stocke des informations sur les variables (globale ou locale).
- **Erreur** : si une variable a été utilisée sans être déclarée ou bien en dehors de son *scope* (portée).
- **Technique** : utiliser une liste d'associations intermédiaire entre les noms visibles et les déclarations de l'environnement identifiées par un nom unique.

Sous-ensemble arithmétique de Mini-Python

```
type binop = Badd | Bsub | Bmul | Bdiv
type constant = Cint of int
type expr =
  | Ecst of constant
  | Eident of ident
  | Ebinop of binop * expr * expr
  | Ecall of ident * expr list
type stmt =
  | Sassign of ident * expr
  | Sprint of expr
  | Sreturn of expr
  | Seval of expr
  | Sblock of stmt list
type def = ident * ident list * stmt
type file = def list * stmt
```

- L'analyse syntaxique construit un objet de type `file`.
- L'analyse de portée produira un nouvel objet de type `file` + un tableau identifiant chaque déclaration.

Exemple de programme

```
def f (x,y) :
  z = 2 * x + y
  return (z)
x = 4
x = f(x,5)
print (y * y)
```

Arbre issu de l'analyse syntaxique:

```
[("f",["x";"y"],
  Sblock [Sassign("z",Ebinop(Badd,Ebinop(Bmul,Ecst(Cint 2),Eident "x"),E
          Sreturn (Eident "z")])],
Sblock [
 Sassign("x", Ecst(Cint 4));
 Sassign("x", Ecall("f", [Eident "x";Ecst(Cint 5)]));
 Sprint(Eident "x") ]
```

Effet de l'analyse de portée

- Des identifiants uniques pour les objets.
- Une table des symboles qui conserve les informations sur les déclarations:
  - le nom original (impression),
  - la nature de la variable, sa "position",
  - le nombre de paramètres des fonctions ...
- Les règles de portée dépendent du langage
  - variables déclarées explicitement (Java, Ocaml) ou implicitement (Python)
  - gestion explicite (Ocaml) ou implicite (Java, Python) des fonctions récursives
- Décider des annotations à stocker (typage, génération de code)
  - variable locale ou globale
  - paramètre de fonction
  - fonction

Mini-python : après analyse de portée

Positions de *déclaration (allocation)*

```
[("f0",["x1";"y2"],
  Sblock [Sassign("z3",Ebinop(Badd,Ebinop(Bmul,Ecst(Cint 2),Eident "x?")
          Sreturn (Eident "z?")])],
Sblock [
 Sassign("x4", Ecst(Cint 4));
 Sassign("x4", Ecall("f?", [Eident "x?";Ecst(Cint 5)]));
 Sprint(Eident "x?") ]
```

f_0	Fun(2)	f
x_1	Param(1)	x
y_2	Param(2)	y
z_3	Local(1)	y
x_4	Global(1)	x

Mini-python : après analyse de portée

Positions d'utilisation (accès)

```
[("f0",["x1";"y2"],
  Sblock [Sassign("z3",Ebinop(Badd,Ebinop(Bmul,Ecst(Cint 2),Eident "x1")
          Sreturn (Eident "z3")])],
Sblock [
 Sassign("x4", Ecst(Cint 4));
 Sassign("x4", Ecall("f0", [Eident "x4";Ecst(Cint 5)]));
 Sprint(Eident "x4") ]
```

Règles de portées

- notion de programme bien formé dans un environnement où certains identifiants sont connus
- $\rho \vdash e$ **ok** dans l'environnement où les variables de ρ sont visibles, l'expression e respecte les règles de portées.
- Des règles pour chaque catégorie syntaxique (expression, instruction, programme...)
- Présentation sous forme de système d'inférence, composé de règles d'inférence

$$\frac{P_1 \ \ldots \ P_n}{P}$$

  - $P_1 \ \ldots \ P_n$ sont les prémisses de la règle (hypothèses)
  - P est la conclusion de la règle
  - Quelles que soient les valeurs des paramètres, si $P_1 \ \ldots \ P_n$ sont vérifiés alors P est vrai
  - Si P est vrai alors il existe (au moins) une règle qui s'applique (dont les prémisses sont également vraies)
  - Une preuve d'un cas particulier de P est donnée par un arbre dont chaque nœud interne correspond à une règle et dont toutes les feuilles correspondent à des règles sans hypothèse.

Règles pour mini-python

Expressions

$$\frac{}{\rho \vdash \texttt{Ecst}(\texttt{Cint}(n))\ \textbf{ok}} \qquad \frac{x \in \rho}{\rho \vdash \texttt{Eident}(x)\ \textbf{ok}} \qquad \frac{\rho \vdash e_1\ \textbf{ok} \quad \rho \vdash e_2\ \textbf{ok}}{\rho \vdash \texttt{Ebinop}(op, e_1, e_2)\ \textbf{ok}}$$

$$\frac{(\rho \vdash e_i\ \textbf{ok})_{i=1..n} \quad f \in \rho \cup \text{PRIM}}{\rho \vdash \texttt{Ecall}(f, [e_1; \ldots; e_n])\ \textbf{ok}}$$

Suite d'instructions ($\texttt{SE} = \texttt{Sreturn}, \texttt{Seval}, \texttt{Sprint}$)

$$\frac{}{\rho \vdash []\ \textbf{ok}} \qquad \frac{\rho \vdash e\ \textbf{ok} \quad \rho + x \vdash p\ \textbf{ok}}{\rho \vdash (\texttt{Sassign}(x, e) :: p)\ \textbf{ok}} \qquad \frac{\rho \vdash e\ \textbf{ok} \quad \rho \vdash p\ \textbf{ok}}{\rho \vdash (\texttt{SE}(e) :: p)\ \textbf{ok}}$$

$$\frac{\rho \vdash l@p\ \textbf{ok}}{\rho \vdash (\texttt{Sblock}(l) :: p)\ \textbf{ok}}$$

Règles pour mini-python (suite)

Programme

$$\frac{\rho \vdash [s]\ \mathbf{ok}}{\rho \vdash ([], s)\ \mathbf{ok}}$$

$$\frac{\rho + lx\ [+f] \vdash sf\ \mathbf{ok} \quad \rho + f \vdash (lf, s)\ \mathbf{ok}}{\rho \vdash ((f, lx, sf) :: lf, s)\ \mathbf{ok}}$$

Reconstruire l'ast décoré

- L'environnement est une suite de liaisons $x \rightsquigarrow n$ avec x le nom de la variable visible et n l'identifiant unique associé

Expressions

ne produisent pas de nouvelles déclarations : $\rho \vdash e \rightsquigarrow e'$

- e respecte les règles de portées dans l'environnement ρ
- e' ast correspondant avec des identifiants uniques

Reconstruire l'ast décoré : expressions

$\rho \vdash e \rightsquigarrow e'$

$$\frac{}{\rho \vdash \texttt{Ecst}(\texttt{Cint}(n)) \rightsquigarrow \texttt{Ecst}(\texttt{Cint}(n))} \qquad \frac{(x \rightsquigarrow n) \in \rho}{\rho \vdash \texttt{Eident}(x) \rightsquigarrow \texttt{Eident}(n)}$$

$$\frac{\rho \vdash e_1 \rightsquigarrow e_1' \quad \rho \vdash e_2 \rightsquigarrow e_2'}{\rho \vdash \texttt{Ebinop}(op, e_1, e_2) \rightsquigarrow \texttt{Ebinop}(op, e_1', e_2')}$$

$$\frac{(\rho \vdash e_i \rightsquigarrow e_i')_{i=1..n} \quad f \rightsquigarrow f' \in \rho \cup \textsf{PRIM}}{\rho \vdash \texttt{Ecall}(f, [e_1; \ldots; e_n]) \rightsquigarrow \texttt{Ecall}(f', [e_1'; \ldots; e_n'])}$$

Reconstruire l'ast décoré : Suite d'instructions

- $\rho \vdash ls \rightsquigarrow ls', d$
- d ensemble des déclarations: pour identifiant x, le nom dans le programme original et la nature: variable (locale, globale) ou paramètre.
- opération $\texttt{decl}(x, e, \rho)$ vérifie si x existe déjà dans l'environnement avec l'étiquette e sinon engendre un nouvel identifiant et l'ajoute, renvoie l'identifiant unique, la déclaration correspondante et le nouvel environnement.

($\texttt{SE} = \texttt{Sreturn}, \texttt{Seval}, \texttt{Sprint}$)

$$\frac{}{\rho \vdash [] \rightsquigarrow [], \emptyset} \qquad \frac{\rho \vdash e \rightsquigarrow e' \quad \rho \vdash p \rightsquigarrow (p', d)}{\rho \vdash (\texttt{SE}(e) :: p) \rightsquigarrow (\texttt{SE}(e') :: p', d)}$$

$$\frac{\rho \vdash e \rightsquigarrow e' \quad (n, d, \rho') = \texttt{decl}(x, \texttt{Var}, \rho) \quad \rho' \vdash p \rightsquigarrow p', d'}{\rho \vdash (\texttt{Sassign}(x, e) :: p) \rightsquigarrow (\texttt{Sassign}(n, e') :: p', \{d\} \cup d')}$$

$$\frac{\rho \vdash l@p \rightsquigarrow (l'@p', d)}{\rho \vdash (\texttt{Sblock}(l) :: p) \rightsquigarrow ((\texttt{Sblock}(l') :: p'), d)}$$

Reconstruire l'ast décoré : Programme

- `newpar`(lx) prend en argument une liste d'identifiant et renvoie la liste des noms uniques et la liste des déclarations de ces identifiant comme paramètre.
- `newf`(f) prend en argument un identifiant, renvoie le nom unique et la déclaration de l'identifiant comme fonction.

$$\frac{\rho \vdash [s] \rightsquigarrow [s'], d}{\rho \vdash ([], s) \rightsquigarrow (([], s'), d)}$$

$$(ln, ld) = \texttt{newpar}(lx) \quad (f', df) = \texttt{newf}(f)$$

$$\frac{\rho + (lx \rightsquigarrow ln) + [f \rightsquigarrow f'] \vdash sf \rightsquigarrow (sf', df') \quad (\rho + f \rightsquigarrow f') \vdash (lf, s) \rightsquigarrow (lf', s'), d}{\rho \vdash ((f, lx, sf) :: lf, s) \rightsquigarrow ((f', ln, sf') :: lf', s'), ld \cup \{df\} \cup df' \cup d)}$$

Mise en œuvre de l'analyse de portée

- Il faut bien distinguer:
  - La *table des symboles* qui contient l'ensemble des déclarations locales et globales incluses dans le programme et qui fait partie de la représentation du programme.
  - L'*environnement local* ρ qui conserve les variables visibles en un point de programme et n'est plus utilisé après l'analyse de portée.
- Les règles peuvent se lire de la conclusion vers les hypothèses (une seule règle pour chaque construction de programme)
- Construction récursive sur la structure du programme, des expressions
- Les déclarations peuvent se construire par effet de bord sur la table des symboles
- L'environnement se traite de manière fonctionnelle

Remarques sur l'analyse de portée

Les règles de portée sont spécifiques au langage

- En python, le statut des variables dépend si elles apparaissent dans le corps d'une fonction (variable locale, sauf usage d'une déclaration **global**) ou de manière globale
  - deux interprétations différentes de l'instruction `Eassign`
- En Java, les déclarations de fonctions dans une classe peuvent être mutuellement récursives, la visibilité des déclarations dépend de leur déclaration *private*, *public* . . .

Variantes

- On peut aussi avoir des constructeurs différents `GVar`, `LVar`, `PVar` pour distinguer les variables locales, globales et les paramètres.

- On peut introduire des *tables différentes* en fonction de la nature des déclarations (variables, fonctions, types, . . .) si l'utilisation permet de choisir la bonne table.

- L'analyse de portée peut aussi être faite *en même temps que le typage* (par exemple ici vérifier que le nombre d'arguments à l'appel d'une fonction correspond à la déclaration).

- On s'est placé dans le cadre où la portée est résolue de manière syntaxique. Dans le cas de surcharge, des informations de typage peuvent être nécessaires pour déterminer l'opération à appliquer.

Analyse sémantique

Le typage

Diviser l'espace des valeurs en collections: un type est une collection de valeurs partageant une même propriété (en particulier même représentation).

Exemple: représentation des flottants ou de structures chaînées.

Les types sont toujours plus ou moins vérifiés:

- soit à l'exécution : typage dynamique (Python)
- soit à la compilation : typage statique (Ocaml)
- soit à la fois à la compilation et à l'exécution (Java)

Intérêts du typage statique

- détection précoce d'erreurs;
- documentation, structuration (une première information sur le programme);
- élimination de certaines erreurs d'exécution; plus de sûreté;
- une manière générale de décrire des analyses de programmes (e.g., analyses d'exception, effets de bords, sécurité du code).

Typage statique vs dynamique

- le typage dynamique est plus précis
  (`if B then 1 else 2+"a"`)
  avec `B` qui s'évalue en `true`
- mais les erreurs de type dépendent des valeurs à l'exécution
- les langages typés *statiquement* peuvent être compilés plus efficacement (e.g., pas de test de représentation à l'exécution), facilite les optimisations (enregistrements, accès, représentations,...)
- il est possible d'utiliser une même représentation pour deux valeurs de types différents sans risque de confusion à l'exécution, e.g.:
  - représenter `true` et `false` par des entiers;
  - les constructeurs de valeur de Caml par des entiers consécutifs.

Propriété attendue du typage statique

le typage statique doit être *conservatif* :

- si le programme est bien typé statiquement alors il n'y a pas d'erreur de type à l'exécution :

$$\frac{e : \tau \quad \text{le calcul de } e \text{ termine sans erreur}}{e \rightsquigarrow v \in [\tau]}$$

- Contre exemple (union disjointe en C):

```
union test {
  int champ1;
  float champ2;
};
main() {union test s;
        s.champ2 = 3.3e15;
        printf("%d\ n", s.champ1);}
```

La confusion entier/adresse donne des erreurs *segmentation fault* à l'exécution

Remarques

- Trouver des systèmes de types les plus riches possibles (polymorphisme, généricité)
  - des opérations qui s'appliquent à des objets de différente nature
    - même code (représentation uniforme)
    - codes différents à déterminer de manière statique ou dynamique
- Des langages comme Java associent du typage statique et dynamique
  - le type dynamique d'un objet Java fait partie de sa représentation

Dans ce cours

- On étudie ici les principes du typage *statique*.
- Comment formuler les règles de typage ?
- Principes des algorithmes pour les mettre en œuvre.

Vocabulaire

Qu'est qu'un type?

- Un type est une expression d'un langage de types
- un type représente un *ensemble de valeurs* (pas un calcul)
- *Déclarer* un type = introduire un nouvel ensemble de valeurs
  Exemple : déclarations de **struct** ou **enum** ...
- *Définir* un type = associer un nom à une exp. de type déjà connue

Typer une expression

- Typer = associer un type à une expression
- Typage *fort* = toute expression a un type *principal*

Vérification vs Inférence

Vérification des types

le programmeur associe un type à chaque déclaration d'identificateur et le compilateur vérifie la correction (e.g, C, C++, Pascal, Ada,...)

```
int addition (int x1, int x2)
  int temp;
   temp = x1 + x2;
   return(temp);
```

Synthèse des types

les types sont devinés (on dira synthétisés ou inférés) par le compilateur par une analyse des contraintes d'utilisation des variables (e.g, Caml, SML,...)

```
let addition x1 x2 =
 let temp = x1 + x2 in temp
val addition : int -> int -> int = <fun>
```

Règles d'inférence

La vérification de types se fait par rapport à des *règles de typage*.

"le programme P est bien typé de type ty"

$$P : ty$$

Vérification

- *vérifier* que ty est un type valide pour P

Synthèse

- *trouver* un ty valide

Typer une expression:
parcourir l'expression en associant un type à chaque sous-expression

Règles de typage

- des axiomes:

$$P : ty$$

- des règles d'inférence

$$\frac{P_1 : ty_1 \ldots P_n : ty_n}{C(P_1, \ldots, P_n) : ty}$$

- typer = construire un arbre où les feuilles sont des axiomes et les noeuds, des règles d'inférence
- Un programme est bien typé lorsque l'on peut construire une telle déduction

Exemple

Expressions arithmético-logiques

Axiomes:

$$\texttt{Cint}(i) : \texttt{int} \quad \texttt{Cbool}(b) : \texttt{bool}$$

règles de déduction:

$$\frac{c : \tau}{\texttt{Ecte}(c) : \tau}$$

$$\frac{e_1 : \texttt{int} \quad e_2 : \texttt{int} \quad op \in \{\texttt{Badd}, \texttt{Bmul}, \texttt{Bdiv}, \texttt{Bsub}\}}{\texttt{Ebinop}(op, e_1, e_2) : \texttt{int}}$$

Exemples

(Ecriture simplifiée “naturelle” des arbres de syntaxe abstraite)

Expression 1 :

```
2 + (4 * 5)
```

$$\dfrac{2 : \mathit{int} \qquad \dfrac{4 : \mathit{int} \qquad 5 : \mathit{int}}{4 * 5 : \mathit{int}}}{2 + (4 * 5) : \mathit{int}}$$

Expression 2 :

```
2 + (true * 5)
```

$$\dfrac{2 : \mathit{int} \qquad \dfrac{\mathit{true} : \mathit{bool} \qquad 5 : \mathit{int}}{\mathit{true} * 5 : ?}}{2 + (\mathit{true} * 5) : ?}$$

Identificateurs

Question:
Que faire en présence d'identificateurs?

```
int x = 1;
int y;
y = 2 + (4 * x);
```

On introduit l'environnement de typage : ρ.

- contient des liaisons $(id, \texttt{type})$
- $\rho = \{(\texttt{y} : \texttt{int})(\texttt{x} : \texttt{int})\}$

Notations

"Dans l'environnement ρ, le programme P a le type ty"

$$\rho \vdash P : ty$$

L'environnement de typage

- ensemble d'associations $\{(x_1 : ty_1), ..., (x_n : ty_n)\}$
- L'analyse de portée faite préalablement assure l'unicité des identifiants.

Langage de types

- *Types de base* : `int`, `bool`, `char`, `string`,...
- *Types construits* :
  - constructeur de type (opérateur sur les types). e.g, `int[]`

```
type typ =  Tint | Tbool | Tchar | Tstring
          | Tarr of typ
```

On pourrait aussi avoir:

- des abréviations de type, des types définis par l'utilisateur...
- gestion de l'égalité, d'une table des symboles de types

Langage

Mini-langage fonctionnel :

```
type constant = Cint of int | Cbool of bool ...
type primop = Badd | Bsub | Bmul | Bdiv ...
type oper = Oprim of primop | Ouser of ident
type expr =
    | Ecst of constant
    | Eident of ident
    | Eoper of of oper * expr list
    | Eletin of ident * expr * expr
type stmt =
      Sassign of ident * expr
    | Sprint of expr
    | Sfun of ident * (ident * typ) list * expr
type file = instr list
```

Signature

On associe à une fonction une signature qui spécifie

- le nombre d'arguments attendus
- le type de chaque argument
- le type de retour

Plusieurs sortes d'opérateurs:

- opérateurs primitifs (arithmétiques) : la signature est fixée
- opérateurs génériques (accès dans un tableau) : la signature va dépendre du type des arguments
- opérateurs définis par l'utilisateur : la signature est donnée par le programme.

Une signature peut se représenter par le type suivant :

```
type sign = typ list * typ
```

Algorithme de typage - prérequis

Chaque expression a un type unique qui peut être inféré.
Une table associe à chaque déclaration son type.

```
val add_typ : ident -> typ -> unit
val find_typ : ident -> typ
```

De même pour les signatures des opérateurs:

```
val add_sign : ident -> sign -> unit
val find_sign : ident -> sign
```

Algorithme de typage

```
let rec type_expr = function
  Ectr c -> type_cte c
| Eident x -> find_typ x
| Eoper(op,le) -> let (lt,t) = find_sign op in
                  check_lexpr lt le; t
| Eletin(x,e1,e2) -> let t1 = type_expr e1 in
                     add_typ x t1; type_expr e2
and check_lexpr = function
  [],[] -> ()
| t::lt,e::le
        -> if eqt (type_expr e) t then check_lexpr lt le
           else raise TypeError
| _,_ -> raise TypeError
```

Algorithme de typage (2)

```
let rec type_prog = function
  [] -> ()
| Sassign(x,e)::p
   -> let t = type_expr e
      in add_typ x t; type_prog p
| Sprint e::p -> let _ = type_expr e in  type_prog p
| Sfun(f,lv,e)::p
   -> let lt = List.map (fun (x,t) -> add_typ x t;t) lv in
      let t = type_expr e
      in add_sign f (lt,t); type_prog p
```

Opérateurs génériques

Certains opérateurs n'ont pas une signature unique:

- conditionnelle **if** b **then** e_1 **else** e_2 a pour type τ si $b : \texttt{bool}$ et $e_1, e_2 : \tau$.
- accès dans un tableau $t[n] : \tau$ si t de type τ `array` et $n : \texttt{int}$.

On leur associe souvent des constructeurs explicites dans la syntaxe abstraite.

```
| Eif(b,e1,e2) ->
    let tb = type_expr b
    and t1 = type_expr e1 and t2 = type_expr e2
    in if eqt tb Tbool && eqt t1 t2 then t1
       else raise TypeError
| Eoper(op,le) -> let lt = List.map type_expr le in
                  find_and_check_sign op lt
```

Remarques

- Il est nécessaire de *comparer les types* suivant des règles qui dépendent du langage quand le système de type est complexe.
  Utilisation d'une fonction spécifique `eqt` et pas l'égalité structurelle.
- Notre algorithme utilise le fait que la *portée* a été préalablement résolue, sinon il faut introduire un environnement ρ qui contient les variables visibles.
- Il n'y a pas d'*annotations* de type dans les instructions **set** car les variables sont initialisées (leur type est calculé à partir du type de l'expression d'initialisation)
- Si les fonctions sont *récursives*, alors il faut aussi donner le type de retour attendu de la fonction

A savoir

- Le rôle de l'analyse de portée, d'une table des symboles, du typage
- Comprendre et manipuler des règles de portée ou de typage définies par un système d'inférence
- La différence entre typage statique et typage dynamique

Evaluation

Exécution

- Un programme décrit des *calculs*
  - appel aux *opérations* du processeur
  - utilisation de la *mémoire* de l'ordinateur (données, code)
  - utilisation des *entrées/sorties* de l'ordinateur (`input`/`print`)
- Un programme manipule des suites de 0, 1 (octets) utilisés pour représenter des valeurs
  - entier (plus ou moins long)
  - caractère (ASCII : 1 octet, utf8 : de 1 à 4 octets)
  - chaine de caractères, tableau : bloc qui va aussi contenir la taille
  - objets : bloc avec les valeurs des attributs ainsi que la classe dynamique
- Le code à exécuter est aussi une donnée
  - suite d'instructions
  - existence d'un pointeur de code : prochaine instruction à exécuter
  - possibilité de changer le cours de l'exécution
    - instruction conditionnelle : `if/then/else`
    - appel de fonction : `call/return`
    - exceptions

Règles sémantiques

- Décrire le comportement opérationnel de l'exécution d'un programme
- Référence à un *modèle abstrait* d'évaluation
- Ingrédients :
  - mémoire : données et code
  - entrées/sorties
  - résultat attendu
    - expression : valeur (+ effet de bord ?)
    - instruction : transformation de la mémoire, entrées sorties
    - gestion des erreurs

Règles sémantiques

- Evaluation à grand pas : du programme à la valeur finale (si elle existe)
- Exemple mini-python
  - mémoire (m) : lien entre les noms (uniques, de variables et des valeurs)
  - sortie (commande `print`) : suite de valeurs (o)
  - expression (e) : calcule une valeur entière,
    peut faire des appels de fonction qui font des effets de bord (modification de la mémoire, sortie)
  - instructions : valeur de retour (`null` si pas de retour) + des effets de bord,
  - on traitera une liste d'instructions (li)
  - jugements :

$$m \vdash e \rightsquigarrow (v, m', o) \qquad m \vdash li \rightsquigarrow (r, m', o)$$

Règles : expressions

Jugements : $m \vdash e \rightsquigarrow (v, m', o)$ $\quad$ $m \vdash [e_1; \ldots; e_n] \rightsquigarrow ([v_1; \ldots; v_n], m', o)$

$$\frac{}{m \vdash \texttt{Ecst}(\texttt{Cint}(n)) \rightsquigarrow (n, m, \epsilon)}$$

$$\frac{(n \rightsquigarrow v) \in m}{m \vdash \texttt{Eident}(n) \rightsquigarrow (v, m, \epsilon)}$$

Opération arithmétique

- Les sous-expressions peuvent faire des effets
- Ordre d'évaluation important
- Nécessité de stocker les valeurs intermédiaires

$$\frac{m \vdash e_1 \rightsquigarrow (v_1, m_1, o_1) \quad m_1 \vdash e_2 \rightsquigarrow (v_2, m_2, o_2)}{m \vdash \texttt{Ebinop}(op, e_1, e_2) \rightsquigarrow (v_1 \; op_{\mathbb{N}} \; v_2, m_2, o_1.o_2)}$$

Règles : appel de fonction

- évaluation et mémorisation des valeurs des arguments
- utilisation possibles de variables locales
- après évaluation du corps de la fonction, seules les variables globales sont conservées
- ici on demande qu'il y ait un retour d'une valeur

$$\frac{m \vdash le \rightsquigarrow (lv, m', o) \quad f = (lx, s) \quad m' + (lx \rightsquigarrow lv) \vdash s \rightsquigarrow (r, m'', o') \quad r \neq \texttt{null}}{m \vdash \texttt{Ecall}(f, le) \rightsquigarrow (r, m'' \mid \texttt{dom}(m), o.o')}$$

Règles : instructions

Jugement : $m \vdash li \rightsquigarrow (r, m', o)$ avec r une valeur entière ou la constante `null`

$$\frac{}{m \vdash [] \rightsquigarrow (\texttt{null}, m, \epsilon)}$$

$$\frac{m \vdash e \rightsquigarrow (v, m', o)}{m \vdash (\texttt{Sreturn}(e) :: p) \rightsquigarrow (v, m', o)} \qquad \frac{m \vdash e \rightsquigarrow (v, m', o) \quad m' \vdash p \rightsquigarrow (r, m'', o')}{m \vdash (\texttt{Seval}(e) :: p) \rightsquigarrow (r, m'', o.o')}$$

$$\frac{m \vdash e \rightsquigarrow (v, m', o) \quad m' \vdash p \rightsquigarrow (r, m'', o')}{m \vdash (\texttt{Sprint}(e) :: p) \rightsquigarrow (r, m'', o.v.o')}$$

$$\frac{m \vdash e \rightsquigarrow (v, m', o) \quad m' + \{x \rightsquigarrow v\} \vdash p \rightsquigarrow (r, m'', o')}{m \vdash (\texttt{Sassign}(x, e) :: p) \rightsquigarrow (r, m'', o.o')}$$

$$\frac{m \vdash l@p \rightsquigarrow (r, m', o)}{m \vdash (\texttt{Sblock}(l) :: p) \rightsquigarrow (r, m', o)}$$

Exécution

- Les règles sémantiques peuvent se traduire en un interpréteur
- Entrée : la mémoire, l'arbre de syntaxe abstraite du programme
- Retour : résultat de l'évaluation, transformation de la mémoire, sorties

Organisation de la mémoire

- codage des données
  - représentation des valeurs de base, des valeurs structurées (record, types définis, objets)
  - Python : tout est objet, pointeur sur une structure avec la classe, la valeur
  - autres langages : valeurs directes et pointeurs (explicites ou implicites)
- données globales, locales à une fonction, locales à une fonction englobante . . .
- politique de *rangement* :
  - emplacement fixe ou relatif
  - emplacement connu à la compilation ou bien dépendant de l'exécution
  - allocation/désallocation statique (C) ou dynamique (langages avec GC)
  - utilisation d'une pile et d'un tas
  - exploiter l'architecture de la machine : registres, mémoires persistentes. . .

Accès aux variables

- Le caractère global ou local des variables est déterminé à l'analyse de portée
- Mémoire globale
  - Valeurs accessibles à tout moment
  - Se comporte comme un tableau mis à jour au cours du calcul
- Mémoire locale
  - liée à l'appel de fonction : paramètres, variables locales (nombre déterminé à l'analyse sémantique)
  - espace libéré à la sortie de fonction
  - des blocs mémoire qui sont empilés à chaque appel, dépilés en sortie
- Définitions imbriquées de fonctions, fonctions en résultat :
  - une fonction peut accéder à ses variables locales mais aussi à celles de la fonction englobante

Appel de fonction

- `Ecall`(f, le) :
  - f fait référence à la définition de la fonction
    - liste des paramètres lx
    - corps de la fonction (dont déclaration de variables locales)
  - les paramètres sont *initialisés* avec les expressions le
  - le corps de la fonction est exécuté dans ce nouvel environnement
- Modes de passage des paramètres :
  - par *valeur* : le paramètre x est initialisé avec la valeur de l'expression e
    - les modifications faites à la variable x disparaissent à la sortie de la fonction
    - en Java, Python, Ocaml, les valeurs sont des pointeurs (vers le tas) ou des valeurs qui tiennent sur un mot (possiblement modifiable)
    - en C des données de plusieurs mots peuvent être copiés sur la pile lors de l'appel
    - les modifications faites à l'adresse de la variable dans le tas sont pérennes
  - par *référence* : lorsque e est une variable, c'est l'adresse de la variable qui est passé en argument (alias entre le paramètre et l'argument)

Appels imbriqués

- Chaque appel de fonction alloue de la mémoire
- Le corps d'une fonction peut appeler une autre fonction
- En cas de fonctions récursives : la chaîne d'appels n'est pas bornée a priori
  - force de la récursivité

```
let rec hanoi A B C n =
if n > 0 then begin hanoi A C B (n-1);
                    trans(A,n,B);
                    hanoi C B A (n-1)
              end
```

  - parfois inutilement complexe, risque de dépassement de la capacité de la pile

```
let rec fact n =
 if n > 1 then n * fact (n-1) else 1
```

Récursion terminale

```
let fact n =
let rec fact2 k n =
  if n > 1 then fact2 (n*k) (n-1) else k
in fact2 1 n
```

- l'appel récursif constitue la dernière instruction
- l'espace alloué pour l'appel initial est réutilisé (boucle)

```
let fact n
let rec fact0 () =
 if n > 1 then ((k,n)=(n*k,n-1); fact()) else k
 in (k,n)=(1,n); fact0()
```

Conclusion

Notions à retenir

- Notion de langage formel : ensemble de mots (suites de caractères)
- Différentes classes de langages :
  - langages réguliers :
    - décrits par une expression régulière
    - reconnaissable à l'aide d'un automate fini
    - exemple de langage non régulier : $\{a^n b^n \mid n \in \mathbb{N}\}$
  - langages algébriques :
    - décrits par une grammaire hors contexte (algébrique)
    - reconnaissable à l'aide d'un automate à pile
    - exemple de langage non algébrique : $\{a^n b^n c^n \mid n \in \mathbb{N}\}$
  - langages récursifs
    - appartenance au langage décidable (calculable)
    - description possible par des règles sémantiques
- règles de portées
- règles de typage (statique/dynamique)
- règles d'évaluation

Savoir faire

- Lire et écrire des spécifications de langage
  - expressions régulières
  - grammaires algébriques
  - règles sémantiques
- Reconnaître qu'un mot est dans un langage
  - construire un automate fini à partir du langage ou de l'expression régulière, simuler l'exécution
  - faire tourner l'algorithme *CYK*, reconnaître une grammaire ambigüe,
  - dire si une grammaire est *LL(1)* (calcul des premiers et suivants), simuler la reconnaissance par analyse descendante
  - spécifier le type des arbres de syntaxe abstraite pour un langage, implanter des règles sémantiques comme *fonction récursive* sur ces arbres
- Principe d'évaluation des expressions, instructions, appel de fonction
- Mettre en œuvre la chaîne d'interprétation d'un langage simple à l'aide des outils `jflex/CUP`

La suite

- TP noté 25 avril à 14h, 2 heures, 4 salles (vérifier votre salle)
- Modifications de fichiers `jflex/cup` pour réaliser différentes tâches
- Examen le mardi 16 mai matin (feuille A4 recto-verso **manuscripte**)
- Suite du programme en L3 :
  - cours de Langages Formels (minimisation d'automates, analyse ascendante, LALR(1),. . .)
  - magistère : cours de compilation