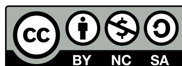
Chapitre 2 - Analyse lexicale

INF7641 Compilation

Jean Privat

Université du Québec à Montréal

INF7641 Compilation
v251



Plan

1. Analyses lexicale et syntaxique
2. Expressions régulières (rappel ?)
3. Automates finis
4. Analyseur lexical
5. Utilisation de SableCC

Analyses lexicale et syntaxique

Lecture de texte

- Objectif: lire un fichier texte « *informatique* »
- Entrée: un fichier texte (ASCII, UTF-8, etc.)
- Sortie: le contenu « *structurel* » du fichier
  Pour pouvoir le traiter simplement ensuite

Tâche informatique commune

- Pour nombreux fichiers textes informatiques structurés
- Langages de programmation (C), de description (HTML, CSS, JSON), de requête (SQL), de configuration (ad hoc), DSL, etc.

Comment faire ?

À la main

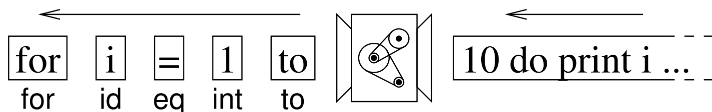
- Programmer un truc ad hoc à chaque fois
- Lire des caractères, comparer des chaînes, gérer les blancs, les mots clés, maintenir les états, backtracker, etc.
- … risque de blessures élevé !

C'est souvent la « même » chose…

- Peut-on résoudre le problème globalement, proprement et efficacement ?

Analyse lexicale

- Synonymes : lexer, scanner, tokenizer
- Donnée : une séquence de caractères
- Résultat : une séquence de jetons (lexèmes, tokens)
- Un jeton : un type (étiquette) + un texte (contenu) + une position (ligne & colonne)



Analyse syntaxique

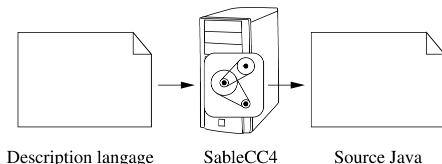
- Extraire la structure grammaticale des jetons
- Synonyme : parser, syntax analysis, syntactic analysis
- Une prochaine fois...

« Compilateur de compilateur »

- Vrai nom: générateur d'analyseur lexical et/ou syntaxique
- Entrée: la « description » d'un langage « informatique »
- Sortie: le code source d'un analyseur lexical et/ou syntaxique

Quelques exemples d'outils

- Lex (et Flex): lexical pour C/C++
- Yacc (et Bison): syntaxique pour C/C++
- ANTLR: les deux, pour plusieurs langages
- Tree-Sitter: les deux, pour plusieurs langages
- SableCC: les deux, pour Java



Problème résolu ?

- On a des résultats théoriques
- Et des outils pratiques
- Problème plus à la mode

Expressions régulières (rappel ?)

Expressions régulières

Inventés par Stephen Kleene en 1951

Outil pratique

- Identifier des sous-chaînes particulières à l'aide de motifs

En informatique

- Outils shells : grep, sed, awk
- Primitifs dans les langages dédiés au traitement de textes : Perl, Ruby, Tcl
- Souvent standard dans les autres langages de programmation
- Éditeurs de texte avancés : rechercher, remplacer
- Programmation : vérification et assainissement des données
- Règles de transformation : réécriture d'URL avec Apache
- Compilateurs et interpréteurs : analyse lexicale

Voir INF1070 pour un rappel

Langage formel

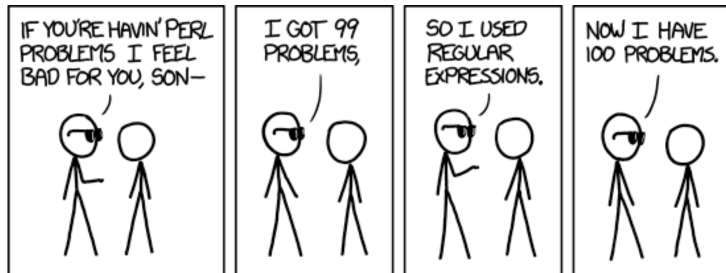
Définitions

- Un **langage** est un ensemble (fini ou pas) de chaînes de caractères (*mots*)
- Une **chaîne de caractères** est une séquence finie de caractères d'un alphabet
- Un **alphabet** est un ensemble fini de caractères

Exemples

- Les prénoms des étudiants du cours :
  « `Olivier` », « `Philippe` », etc.
- Les entiers littéraux décimaux : « `0` », « `1` », ..., « `42` », etc.
- Les identifiants de fonction C : « `toto` », « `fork` », etc.
- Le langage C ?
- Le français ?

Perl Problems



Source: xkcd, 2013

Expression et langage réguliers

Expression régulière

- Ou « expression rationnelle »
- Une façon de définir un langage
- En intension (ses propriétés)
- Et non en extension (la liste des mots du langage)

Exemple : les identifiants de fonction C

- Commencent par une lettre ou un souligné
- Éventuellement suivit d'un mélange de lettres, de chiffres et de soulignés

Langage régulier (définition)

- Langage définissable par une expression régulière
- Note : tous les langages ne sont pas réguliers

Concepts **de base** des expressions régulières

Éléments de base

- Caractères : `a`, `b`, `c`, etc.
- Chaîne vide : `''` (ou ε)
- Langage vide : $\emptyset$ (rarement utilisé)

Opérations de base

- Alternation : `a|b` = {`a`, `b`}
- Concaténation : `ab` = {`ab`}
- Fermeture de Kleene (étoile) : `a*` = { ε, `a`, `aa`, …}

Parenthèses (pour les priorités)

- `(a)` = `a` = { `a` }

Familles d'expressions régulières

- Celles habituellement utilisées par les informaticiens
- Chaque langage et bibliothèque a ses propres détails
- BRE, ERE, PCRE, etc.

Étendent les expressions régulières *de base*

- Échappement : `\`
- Groupes de caractères : `.`, `[...]`, `\d`, etc.
- Quantificateurs : `?`, `+`, `{...}`

Dépassent les expressions régulière *de base*

- Assertions : `^`, `$`, `\b`, etc.
- Groupe de capture et références arrières : `(...)` et `\1`
- Prévision (*lookahead*), rétrovision (*lookbehind*): `(?=...)`, `(?<=...)`, etc.
- Anti-avarice et possession: `...*?`, `...*+`, etc.
- Conditions, commentaires, code embarqué, etc.

Backslashes

\ — BACKSLASH
\\ — REAL BACKSLASH
\\\ — *REAL* REAL BACKSLASH
\\\\ — ACTUAL BACKSLASH, FOR REAL THIS TIME
\\\\\ — ELDER BACKSLASH
\\\\\\ — BACKSLASH WHICH ESCAPES THE SCREEN AND ENTERS YOUR BRAIN
\\\\\\\ — BACKSLASH SO REAL IT TRANSCENDS TIME AND SPACE
\\\\\\\\ — BACKSLASH TO END ALL OTHER TEXT
\\\\\\\\\\\... — THE TRUE NAME OF BA'AL, THE SOUL-EATER

Source: xkcd, 2016

Exercice 1

Écrire l'expression régulière des identifiants C

- Une lettre ou un souligné
- Éventuellement suivit d'un mélange de lettres, de chiffres et de soulignés

Exercice 1

Écrire l'expression régulière des identifiants C

- Une lettre ou un souligné
- Éventuellement suivit d'un mélange de lettres, de chiffres et de soulignés

Solution

- `[a-zA-Z_][a-zA-Z_0-9]*`

Exercice 2

Écrire l'expression régulière des commentaires C

- Exemple : `/* Ceci est un commentaire */`

Exercice 2

Écrire l'expression régulière des commentaires C

- Exemple : `/* Ceci est un commentaire */`

Pièges

- `/* Bla */ Bla */`
- `/*Bla*/ x=y; /*Bla*/`

Exercice 2

Écrire l'expression régulière des commentaires C

- Exemple : `/* Ceci est un commentaire */`

Pièges

- `/* Bla */ Bla */`
- `/*Bla*/ x=y; /*Bla*/`

Solutions (ignorez les espaces)

- `/* ([^*] | *+ [^*/])* *+/`
- `/* [^*]* (* ([^*/] [^*]*)?)* */`

Syntaxe SableCC 4

Syntaxe spécifique adaptée à l'écriture de *lexers*

- Joker: `Any` = { …, ‘a’, ‘b’, ‘c’, …}
- Classe: `'a'..'z'` = { ‘a’, …, ‘z’ }
- Chaîne: `'abcd'` = { ‘a’ ‘b’ ‘c’ ‘d’ ‘e’ }
- Quantificateurs : `'a'*`, `'a'?`, `'a'+`, `'a'^2`, `'a'^(2..3)`, `'a'^(3...)`

Syntaxe SableCC 4 : Opérations étendues

Opérations ensembliste

- Intersection : `('aa'|'bb') And ('a'+)` = { ‘aa’ }
- Soustraction : `('aa'|'bb')? Diff ('a'+)` = { ‘bb’ }

Soustraction sémantique

- `Any* - 'Jean'`

Plus court, plus long

- Commentaire C : `Shortest '/*' Any* '*/'`
- `Longest`

Listes et séparateurs

- Adresse IP : `(('0'..'9')+ Separator '.')^4`

Automates finis

Évaluation d'une expression régulière

Soit une expression régulière définissant un langage

- Une chaîne appartient-elle au langage ?
- Rechercher les sous-chaînes appartenant au langage ?

Questions non triviales

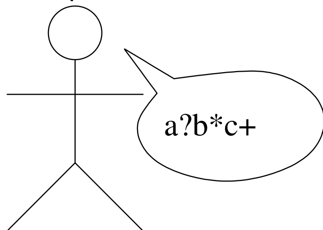
Évaluation d'une expression régulière

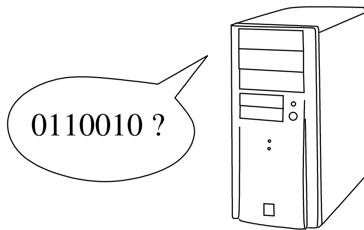
Soit une expression régulière définissant un langage

- Une chaîne appartient-elle au langage ?
- Rechercher les sous-chaînes appartenant au langage ?

Questions non triviales

- Même pour un ordinateur





Questions non triviales

Comment évaluer ?

- `a*ba*ba*`
- `(ab|b)*a?`

Questions non triviales

Comment évaluer ?

- `a*ba*ba*`
- `(ab|b)*a?`

Ces expressions régulières sont-elles équivalentes ?

C'est-à-dire, les langages sont-ils identiques ?

- `a+ba*|a*ba+`
- `a+ba+|a+b|ba+`

Questions non triviales

Comment évaluer ?

- `a*ba*ba*`
- `(ab|b)*a?`

Ces expressions régulières sont-elles équivalentes ?

C'est-à-dire, les langages sont-ils identiques ?

- `a+ba*|a*ba+`
- `a+ba+|a+b|ba+`

Si oui, laquelle est la plus rapide à évaluer?

- Est-ce quelque chose impacte les performances ?
- Si oui, quoi ?

Outils nécessaires

Structures de données

- Automates finis

Algorithmes

- Transformation d'automates
- Évaluation d'automates

Automate = Multi-Graphe

Transitions = arcs

- Orientés
- Étiquetés par un caractère de l'alphabet ou par ε (epsilon)

États = nœuds

- Un état de départ
- Un ensemble d'états d'acceptation (éventuellement vide)

Automate fini

- Nombre fini d'états (et de transitions)

Plus formel ?

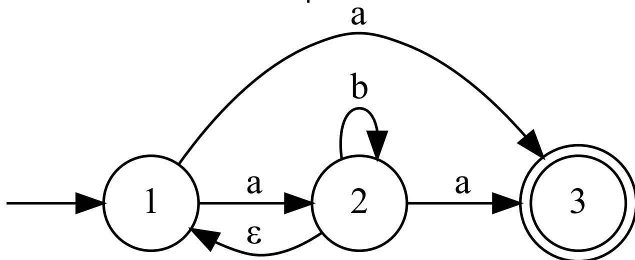
Un automate fini est un quintuplet $(Q, \Sigma, \delta, q_0, F)$ où

- Q est un ensemble d'état (nœuds)
- Σ est un ensemble de symboles (alphabet)
- $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ est l'ensemble des etiquettes
- $\delta \subseteq Q \times \Sigma_\varepsilon \times Q$ est l'ensemble des transitions (arcs)
- $q_0 \in Q$ est un état initial
- $F \subseteq Q$ est un ensemble d'états d'acceptation (finaux ou terminaux)

Automate fini non déterministe (NFA)

Règle : pas de règle

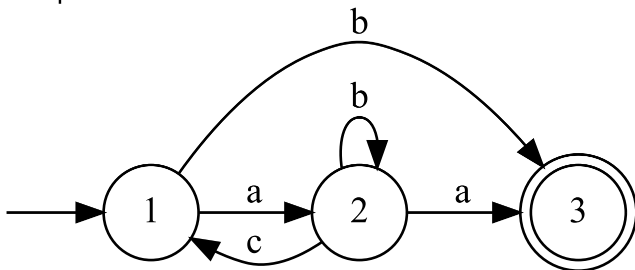
- Pas de restriction sur les étiquettes des transitions



Automate fini déterministe (DFA)

Règles

- Au plus une transition sortante pour une étiquette donnée
- Pas d'étiquette ε



Langages réguliers

NFA et DFA définissent un langage

- L'ensemble des chemins partant d'un état de départ vers un état d'acceptation

NFA, DFA et expression régulières

- Reconnaissent la même classe de langages :
  $\Rightarrow$ les langages réguliers
- Et ça c'est fort !

Évaluation d'automates

Soit un automate fini définissant un langage

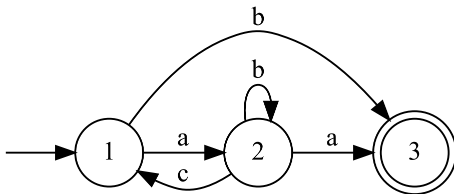
- Une chaîne appartient-elle au langage ?

Facile

- Il suffit de trouver un chemin
  du départ jusqu'à un état d'acceptation
- Pénible sur un NFA
- Mais très facile avec un DFA (algorithme linéaire)

Évaluation d'automates : Exercice

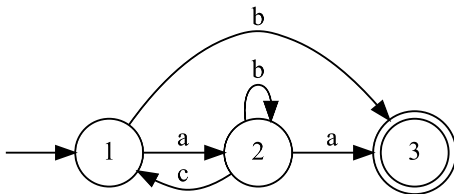
Soit le DFA



Quelles chaînes sont reconnues parmi

- `aa`
- `acabcb`
- `acc`
- `abbc`

Évaluation d'automates : Exercice
Soit le DFA



Quelles chaînes sont reconnues parmi

- `aa`
  $1 \to a \to 2 \to a \to 3 \to \$ \to OK$
- `acabcb`
  $1 \to a \to 2 \to c \to 1 \to a \to 2 \to b \to 2 \to c \to 1 \to b \to 3 \to \$ \to OK$
- `acc`
  $1 \to a \to 2 \to c \to 1 \to c \to$ PAS OK
- `abbc`
  $1 \to a \to 2 \to b \to 2 \to b \to 2 \to c \to 1 \to \$ \to$ PAS OK

Évaluation d'expression régulières

Trois étapes

- Transformation RE $\rightarrow$ NFA
- Transformation NFA $\rightarrow$ DFA
- Évaluation du DFA

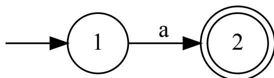
Efficace

- Si on pré-traite l'expression régulière

Transformation d'expression régulières → NFA

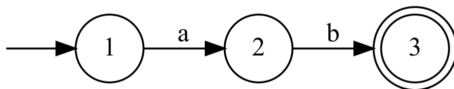
Atome (caractère ou ε)

- `a`



Concaténation

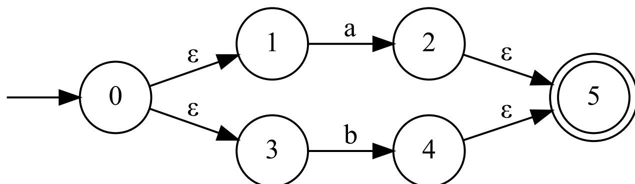
- `ab`



Transformation d'expression régulières → NFA

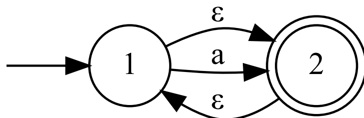
Alternation

- `a|b`



Étoile

- `a*`



RE $\rightarrow$ NFA : Exercice 1

Écrire le NFA de l'expression régulière

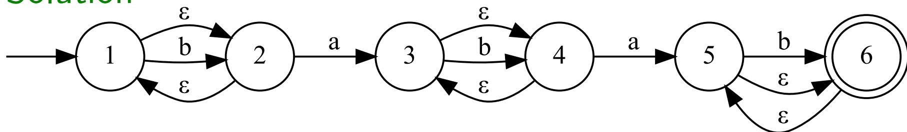
- `b*ab*ab*`

RE → NFA : Exercice 1

Écrire le NFA de l'expression régulière

- `b*ab*ab*`

Solution



RE → NFA : Exercice 2

Écrire le NFA de l'expression régulière

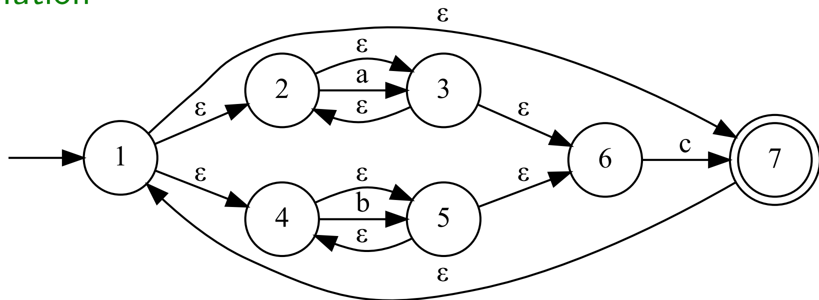
- `((a*|b*)c)*`

RE → NFA : Exercice 2

Écrire le NFA de l'expression régulière

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

Solution



RE $\rightarrow$ NFA : Exercice 3

Écrire le NFA de l'expression régulière

- `a(bc)?d`

RE → NFA : Exercice 3

Écrire le NFA de l'expression régulière

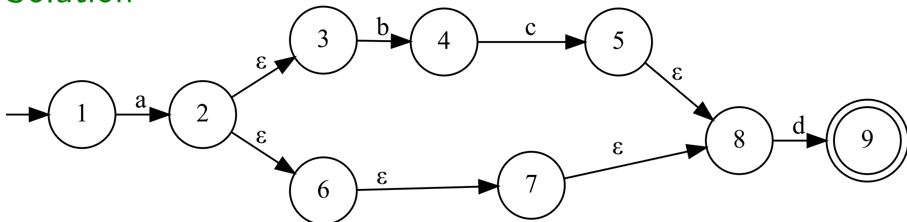
- `a(bc)?d` $\equiv$ `a(bc|)d`

RE → NFA : Exercice 3

Écrire le NFA de l'expression régulière

- `a(bc)?d` ≡ `a(bc|)d`

Solution



Transformation NFA $\rightarrow$ DFA

Idée

- Simuler en parallèle tous les chemins
  $\Rightarrow$ un état du DFA $\approx n$ états du NFA

Risque

- Au pire, DFA exponentiellement plus grand que NFA
- Mais suffisamment rare en pratique

Outils sur les NFA

ε-fermeture(E)

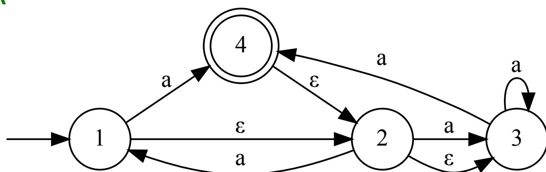
- L'ensemble des états atteignable par 0, 1, ou plusieurs transitions ε à partir d'un état de l'ensemble E

trans(E, c)

- L'ensemble des états atteignable par une seule transition c à partir d'un état de l'ensemble E

Outils sur les NFA : Exercice

Soit le NFA



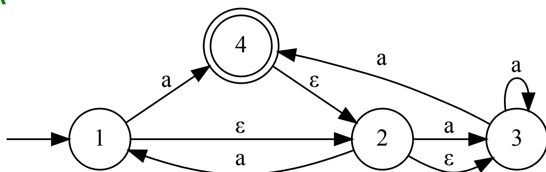
Déterminer $F = \varepsilon$-fermeture(E) et $T =$trans(E, a)

Pour les ensembles E suivants :

- $E = \{1\}$
- $E = \{1, 2\}$
- $E = \{3\}$
- $E = \{4\}$

Outils sur les NFA : Exercice

Soit le NFA



Déterminer $F = \varepsilon$-fermeture(E) et $T =$trans(E, a)

Pour les ensembles E suivants :

- $E = \{1\}$: $F = \{1, 2, 3\}$; $T = \{4\}$
- $E = \{1, 2\}$: $F = \{1, 2, 3\}$; $T = \{1, 3, 4\}$
- $E = \{3\}$: $F = \{3\}$; $T = \{3, 4\}$
- $E = \{4\}$: $F = \{2, 3, 4\}$; $T = \emptyset$

NFA $\rightarrow$ DFA : Algorithme

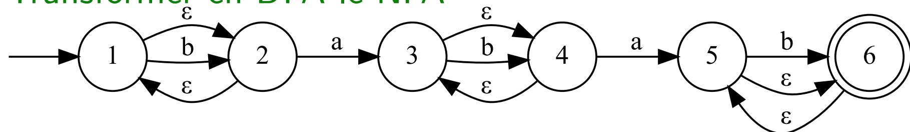
Données : Un NFA N

Résultat : Un DFA D définissant le même langage que N

```
E = ε−fermeture(depart(N));
ajouter E comme état de départ de D (sans le marquer);
tant que un état E de D est non marqué faire
    marquer E;
    pour chaque caractère c de l'alphabet faire
        F = ε−fermeture(trans(E, c));
        si F n'est pas un état de D alors
            ajouter l'état F à D (sans le marquer);
            si un élément de F est un état d'acceptation de N alors
              F est un état d'acceptation de D;
        fin
        ajouter la transition E -c-> F à D;
    fin
fin
```

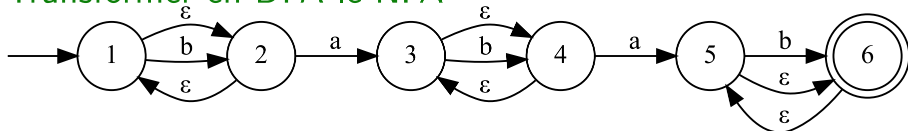
NFA → DFA : Exercice 1

Transformer en DFA le NFA

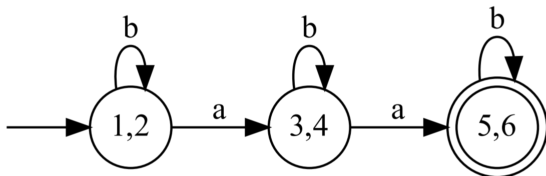


NFA → DFA : Exercice 1

Transformer en DFA le NFA

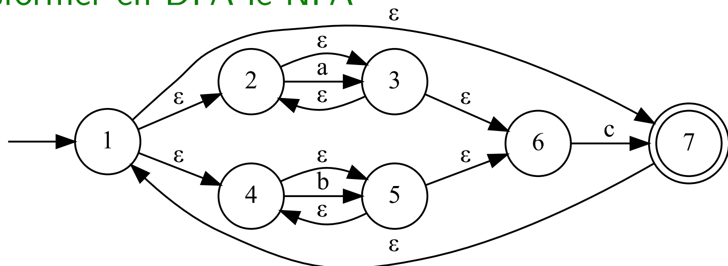


Solution



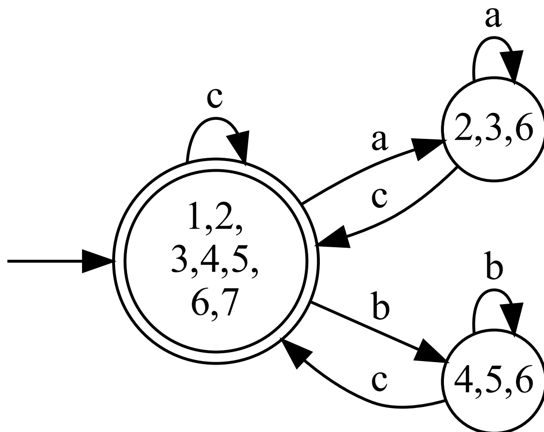
NFA → DFA : Exercice 2

Transformer en DFA le NFA



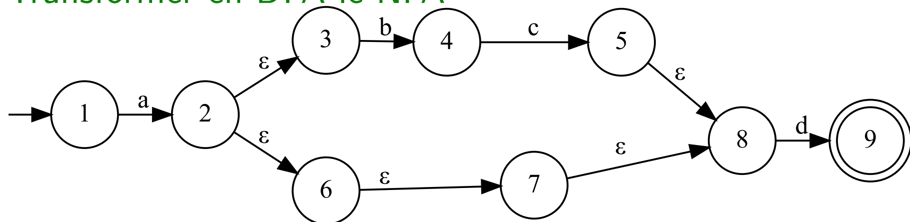
NFA → DFA : Exercice 2

Solution



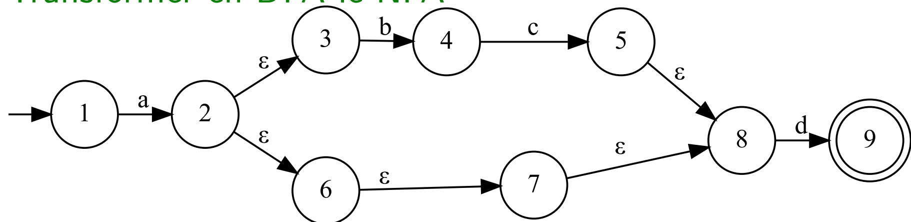
NFA → DFA : Exercice 3

Transformer en DFA le NFA

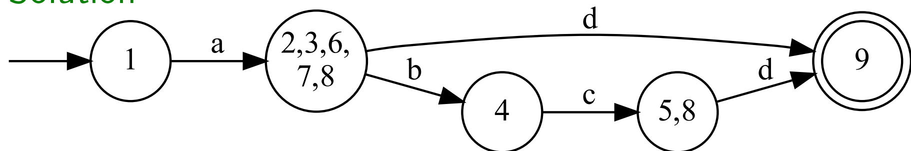


NFA → DFA : Exercice 3

Transformer en DFA le NFA



Solution



Autres algorithmes

Minimisation DFA

- La même chose avec moins d'états
- Théorème : DFA minimal unique
- Corollaire : Permet de déterminer l'équivalence d'expressions régulières

DFAisation paresseuse

- Construire et évaluer le DFA en même temps
- Utile si l'expression régulière n'a pas vocation à être réutilisée (exemple `grep`)

Transformation DFA → Expression régulière

- Prouve l'équivalent de la classe de langages réguliers
- Pas forcément très utile en pratique

Analyseur lexical

Description de langage

```
Language demo;
Lexer
  letter = 'a'..'z';
  digit = '0'..'9';
  identifier = letter (letter | digit)*;
  comma = ',';
  blank = (' ' | #9 | #10 | #13)+;
  if = 'if';
  else = 'else';
Token identifier, comma, if, else;
Ignored blank;
Parser
g=;
```

Sélection de jetons

Plusieurs séquençages en jetons sont souvent possibles

Séquence de caractères : « `ify` »

- 1 jeton : id « `ify` »
- 2 jetons : id « `if` », id « `y` »
- 2 jetons : if « `if` », id « `y` »
- 2 jetons : id « `i` », id « `fy` »
- 3 jetons : id « `i` », id « `f` », id « `y` »

Pas de place à l'ambiguïté

- Il faut des règles !

Règles de l'analyse lexicale

Règle 1

- Le jeton le plus long gagne toujours

Exercice : Traitez les séquences suivantes

- « `toto ,,i` »
- « `iiff iff i f` »

Règles de l'analyse lexicale

Règle 1

- Le jeton le plus long gagne toujours

Exercice : Traitez les séquences suivantes

- « `toto ,,i` »
  $\rightarrow$ 4 jetons : id « `toto` », comma « `,` », comma « `,` », id « `i` »
- « `iiff iff i f` »
  $\rightarrow$ 4 jetons : id « `iiff` », id « `iff` », id « `i` », id « `f` »

Règles de l'analyse lexicale

Règle 2

- Des jetons de même taille ne peuvent gagner ensemble

Exemple

- Séquence de caractères : « if »
- 1 jeton, deux choix : id « `if` » ou if « `if` »

Solutions

- Lex, SableCC3 : Ordre des déclarations importe
- SableCC4 : Priorité d'inclusion

Priorité lexicale d'inclusion en SableCC4

Règle de l'inclusion lexicale

- Une expression régulière strictement inclue dans une autre gagne la priorité
- Remarque : fait en général la bonne chose

Priorité lexicale d'inclusion en SableCC4

```
Grammar priorite_d_inclusion:
Lexer
  identifier = ('a'..'z')+;
  if = 'if';
  Token identifier, if;
  Ignored ' ', #9, #10, #13;
```

- *if* $\subset$ *identifier* donc *if* à la priorité sur *identifier* pour une chaîne de même taille

Exercice : Traitez les séquences suivantes

- « `i f` »
- « `if` »
- « `iffy` »

Priorité lexicale d'inclusion en SableCC4

```
Grammar priorite_d_inclusion:
Lexer
  identifier = ('a'..'z')+;
  if = 'if';
  Token identifier, if;
  Ignored ' ', #9, #10, #13;
```

- *if* $\subset$ *identifier* donc *if* à la priorité sur *identifier* pour une chaîne de même taille

Exercice : Traitez les séquences suivantes

- « `i f` » $\rightarrow$ 2 jetons : id «i», id «f»
- « `if` » $\rightarrow$ 1 jeton : if «if» (priorité d'inclusion)
- « `iffy` » $\rightarrow$ 1 jeton : id «iffy»

Déclarer les inclusions lexicales en SableCC4

Déclaration de priorités

- La directive `Priority` permet de déclarer des priorités
- Attention : on a en rarement besoin

Forcer la priorité lexicale en SableCC4

```
Grammar priorite_forcee;
Lexer
  letter = 'a'..'z';
  digit = '0'..'9';
  identifier = letter(letter|digit)*;
  hexinteger = (digit|'a'..'f')+;
  Token identifier, hexinteger;
  Priority identifier > identifier
```

Exercice : traitez les séquences suivantes

- « `z10` »
- « `00ff1` »
- « `fff` »
- « `00fg1` »

Forcer la priorité lexicale en SableCC4

```
Grammar priorite_forcee;
Lexer
  letter = 'a'..'z';
  digit = '0'..'9';
  identifier = letter(letter|digit)*;
  hexinteger = (digit|'a'..'f')+;
  Token identifier, hexinteger;
  Priority identifier > identifier
```

Exercice : traitez les séquences suivantes

- « `z10` » → 1 jeton : identifier « ‘z10‘ »
- « `00ff1` » → 1 jeton : hexinteger « ‘00ff1‘ »
- « `fff` » → 1 jeton : identifier « ‘fff‘ » (priorité déclarée)
- « `00fg1` » → 2 jetons : hexinteger « ‘00f‘ », identifier « ‘g1‘ »

Fonctionnement d'un analyseur lexical

Éléments de base

- Expressions régulières
- Automates

Particularité

- Ne plus reconnaître les mot d'un langage
  N'est plus un simple algorithme de décision oui/non
- Mais extraire des jetons d'une séquence de caractères
  Possiblement infinie

Un seul automate fini déterministe

Un automate fini déterministe unique

- Un automate pour les reconnaître tous
- Un automate pour les trouver
- Un automate pour les extraire tous et en jetons les séquencer

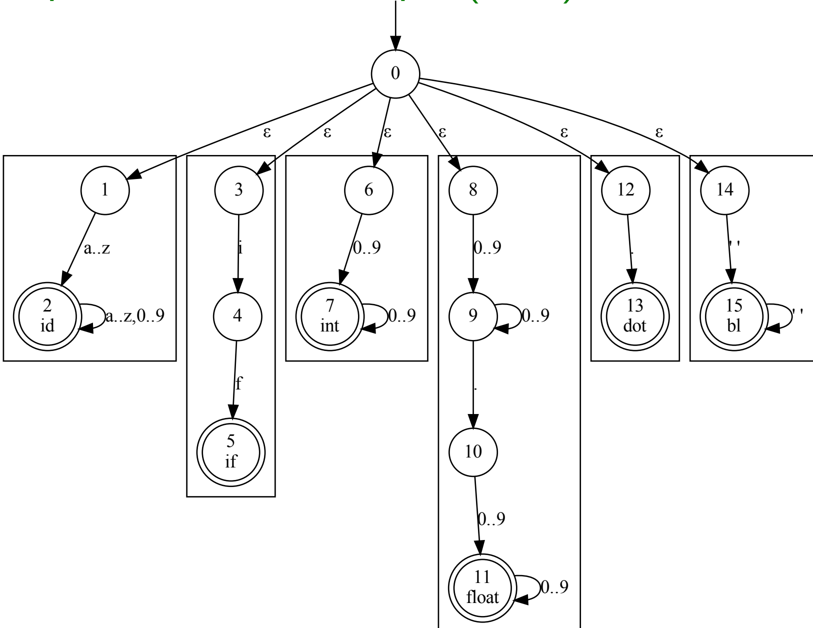
Forger l'automate unique

- Transformer l'expression régulière de chaque jeton en NFA
- Marquer les états d'acceptation des NFA par le jeton
- Regrouper tous les états de départ
- Transformer en DFA

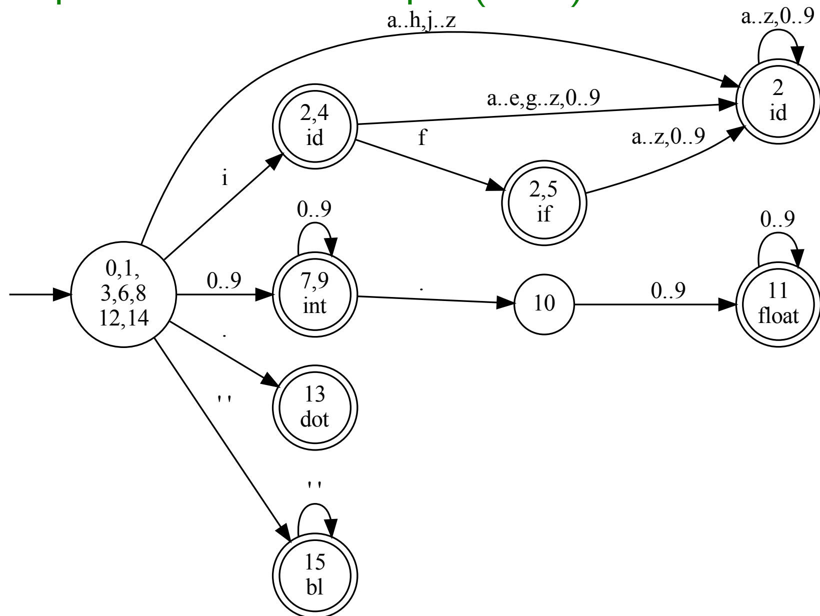
Exemple d'automate unique

```
Grammar automate:
Lexer
  letter = 'a'..'z';
  digit = '0'..'9';
  id = letter(letter|digit)*;
  if = 'if';
  int = (digit)+;
  float = (digit)+ '.' (digit)+;
  dot = '.';
  bl = ' ';
  Token id, if, int, float, dot;
  Ignored bl;
```

Exemple d'automate unique (NFA)



Exemple d'automate unique (DFA)



Gestion des priorités

Définition de conflit

- Un état d'acceptation du DFA unique est annoté de plusieurs jetons

Résolution de conflit

- Par priorité d'inclusion
- Par priorité déclarée
- Erreur sinon

Calcul des inclusions des tokens

- Déduit de l'inclusion des états d'acceptations

Principe d'extraction de jetons

Avancer

- Le plus loin possible dans l'automate
- En mémorisant le dernier état d'acceptation rencontré

Si avancer est impossible, alors

- Retourner le jeton du dernier état d'acceptation rencontré
- Repartir de l'état de départ de l'automate
- Commencer au caractère qui suit le jeton retourné (*rollback* possible)

Algorithme d'extraction de jetons

Données : Un DFA D, une séquence de caractères S
Résultat : Une séquence de jetons J
$debut = pos = 0$; $candidat =$ null; $E =$ départ(D);
Boucler
    $c =$ caractère numéro pos de S (ou EOF sinon);
    $pos++$;
    $E =$ successeur de E par la transition c (ou null sinon);
    si $E ==$ *null* **alors**
        si $candidat ==$ *null* **alors retourner** erreur lexicale;
        Ajouter $candidat$ à J;
        si $c ==$ *EOF* **alors retourner** J;
        $E =$ départ(D);
        $pos = debut =$ caractère après $candidat$; $candidat =$ null;
    sinon si E *accepte jeton* j **alors**
        $candidat =$ new Jeton(j, $debut$, pos-1);
    fin
fin

Algorithme d'extraction

Exercice : Traitez les séquences suivantes

- « `if cond iffy10 iffy 11 12.13 14 . 15` »
- « `1.2.3.4.5` »
- « `1a2b3c4` »
- « `1.a` »

Algorithme d'extraction

Exercice : Traitez les séquences suivantes

- « `if cond iffy10 iffy 11 12.13 14 . 15` » → 9 jetons :
  if « if », id « cond », id « iffy10 », id « iffy », int « 11 », float « 12.13 », int « 14 », dot « . », int « 15 »
- « `1.2.3.4.5` » → 5 jetons :
  float « 1.2 », dot « . », float « 3.4 », dot « . », int « 5 »
- « `1a2b3c4` » → 2 jetons :
  int « 1 », id « a2b3c4 »
- « `1.a` » → 3 jetons :
  int « 1 », dot « . », id « a »

Coût algorithmique

En pratique

- Linéaire en la taille de la séquence de caractères

Au pire

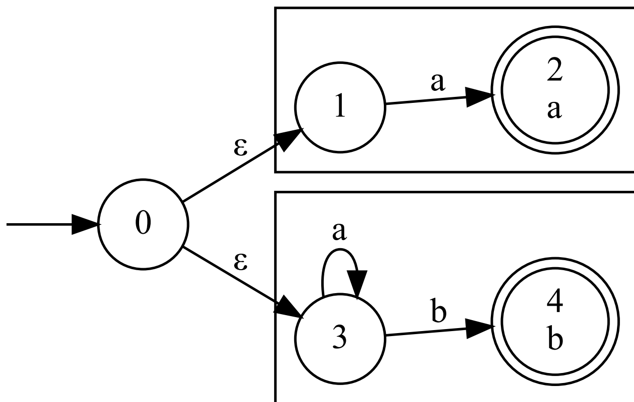
- Quadratique en la taille de la séquence de caractères (à cause du *rollback* possible)

Pire coût algorithmique

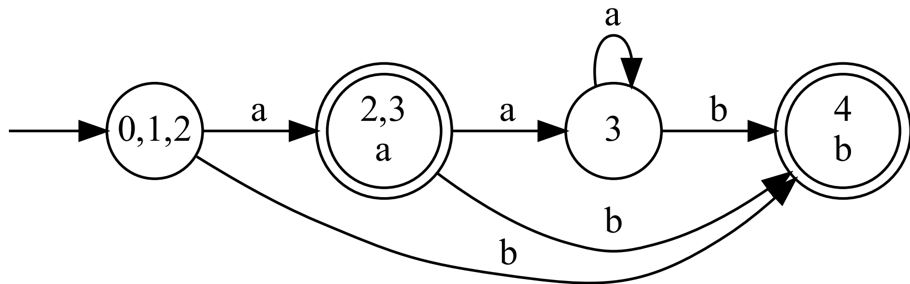
Exemple de langage du pire coût

```
Grammar pire_cout:
Lexer
  a = 'a';
  b = 'a'* 'b';
  Token a, b;
```

Pire coût algorithmique (NFA)



Pire coût algorithmique (DFA)



Pire coût algorithmique

Exercice : Traitez les séquences suivantes & comptez les tours de boucle

- « `aaab` »
- « `aaa` »
- « `aaaaa` »
- mille `a` suivis d'un `b` (« `aaa...ab` »)

- mille `a` (« `aaa..a` »)

Pire coût algorithmique

Exercice : Traitez les séquences suivantes & comptez les tours de boucle

- « `aaab` » $\rightarrow$ 1 jeton b « aaab » ; 5 tours de boucle
- « `aaa` » $\rightarrow$ 3 jetons a « a » ; 9 tours de boucle
- « `aaaaa` » $\rightarrow$ 5 jetons a « a » ; 21 tours de boucle
- mille a suivis d'un `b` (« `aaa...ab` »)
  $\rightarrow$ 1 gros jeton b ; 1002 tours de boucles
- mille a (« `aaa..a` »)
  $\rightarrow$ 1000 jetons a ; 501501 tours de boucles

Utilisation de SableCC

Utilisation du générateur d'analyseur SableCC4

Invocation de SableCC4

- `java -jar sablecc4.jar lang.sablecc -p pack.age`
- `lang.sablecc` est le fichier de description de langage
- `pack.age` est le package racine des classes Java générées
  les fichiers sont crées à partir du répertoire `pack/age`

Fichiers générées

- `pack/age/language_XXX/Lexer.java`
  La classe qui fait l'analyse lexicale
- `pack/age/language_XXX/Token.java`
  La classe racine des jetons
- Une classe `N...` par jeton.

API principale des classes générées

Classe `Lexer`

- `Lexer(Reader)`
  Initialise un nouvel analyseur syntaxique à partir d'une séquence de caractères.
- `Token next() throws LexerException, IOException`
  Retourne le jeton suivant
  ou un jeton de type spécial `End` à la fin de la séquence
  ou lève une exception

API principale des classes générées

Classe `Node`

- `enum Type`
  Un identifiant par type de nœud (`T_...`)
- `Type getType()`
  Le type du nœud

Classe `Token` (entends `Node`)

- `String getText()`
  Retourne la chaîne de caractère du jeton
- `int getLine()` et `int getPos()`
  Retournent la ligne et la colonne du jeton

Exemple d'utilisation d'un analyseur

```
public class Demo {
  public static void main(String[] args) {
    try {
      FileReader fr = new FileReader(args[0]);
      Lexer lexer = new Lexer(fr);
      Token t;
      do {
        t = lexer.next();
        System.out.println("["+t.getLine()+","+
          t.getPos()+"]"+t.getType()+" '"+
          t.getText()+"'");
      } while(t.getType() != Node.Type.TEnd);
    } catch(Exception e) {
        System.err.println(e.getMessage() + ".");
}}}
```