Langages et Compilation

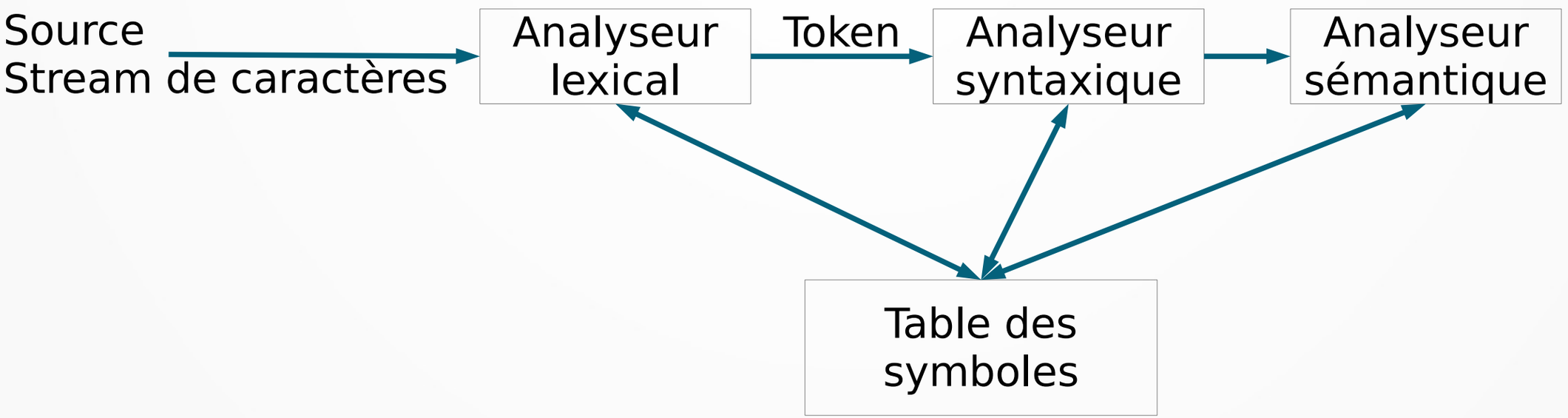
L2: Analyse lexicale

T. Goubier

L3A

2019/2020

L2: Analyse lexicale



L2: Analyse lexicale

- Mots clés
  - Analyseur lexical / lexer / scanner
  - Automates
  - Expressions régulières
- Objectif de l’analyse lexicale :
  - Découper une entrée texte (un flot, un stream) en un flot de tokens regroupant ces tokens

L2: Analyse lexicale

- Définitions
  - Token
    - Un nom (un type) et une valeur: un lexème
  - Pattern / Motif
    - Un motif décrivant tous les lexèmes possibles pour une classe de token
  - Lexème
    - Une chaîne de caractères

L2: Analyse lexicale

- Exemples

Token	Description / pattern / motif	Lexème
IF	les charactères i f	if
ELSE	les charactères e l s e	else
COMPARAISON	< ou > ou <= ou >= ou == ou !=	<=
IDENTIFIANT	une lettre suivie par une lettre ou un chiffre zéro ou plusieurs fois	pi score d2
NUMBER	Une constante numérique	6.02e23 42
LITERAL	tout sauf ", entouré par des " "	"core dumped"

L2: Analyse lexicale

- Particularités :
  - Les espaces sont / peuvent être des tokens
    - C'est en fonction des cas:
    - Certains langages donnent un sens aux espaces et tabulations:
      - Python (mais seulement en début de ligne)
    - La plupart, non
      - Le passé a montré que ce n'est pas une bonne idée...
    - Implémentation par des tokens ignorés (souvent)
      - Ils deviennent des séparateurs entre tokens

L2: Analyse lexicale

- Particularités :
  - Les espaces sont / peuvent être des tokens
    - C'est en fonction des cas:
    - Certains langages donnent un sens aux espaces et tabulations:
      - Python (mais seulement en début de ligne)
    - La plupart, non (C, C++, Java, etc...)
      - Le passé a montré que ce n'est pas une bonne idée...
    - Implémentation par des tokens ignorés (souvent)
      - Ils deviennent des séparateurs entre tokens

L2: Analyse lexicale

- Python:

The indentation levels of consecutive lines are used to generate INDENT and DEDENT tokens, using a stack, as follows.

Before the first line of the file is read, a single zero is pushed on the stack; this will never be popped off again. The numbers pushed on the stack will always be strictly increasing from bottom to top. At the beginning of each logical line, the line's indentation level is compared to the top of the stack. If it is equal, nothing happens. If it is larger, it is pushed on the stack, and one INDENT token is generated. If it is smaller, it must be one of the numbers occurring on the stack; all numbers on the stack that are larger are popped off, and for each number popped off a DEDENT token is generated. At the end of the file, a DEDENT token is generated for each number remaining on the stack that is larger than zero.

L2: Analyse lexicale

- Particularités :
  - Les retour à la ligne
    - Sont dépendant de la plateforme:
      - Unix / Linux: LF (linefeed)= \n
      - Windows: CR LF (carriage return + line feed) = \r\n
      - Mac: OSX: LF = \n, pre OSX: CR = \r
  - Peuvent être des tokens
    - Mais souvent pas partout / tout le temps
    - Retour à la ligne utilisé comme fin de statement (équivalent au ;)
      - Mais pas tout le temps (ex: Python)

L2: Analyse lexicale

- Python (encore!) :
  - Une ligne logique (finie par un token NEWLINE)
    - Peut être composée de plusieurs lignes physiques (sans token NEWLINE)

3 lignes
1 logique

```
if 1900 < year < 2100 and 1 <= month <= 12 \
   and 1 <= day <= 31 and 0 <= hour < 24 \
   and 0 <= minute < 60 and 0 <= second < 60:   # Looks like a valid date
        return 1
```

4 lignes
1 logique

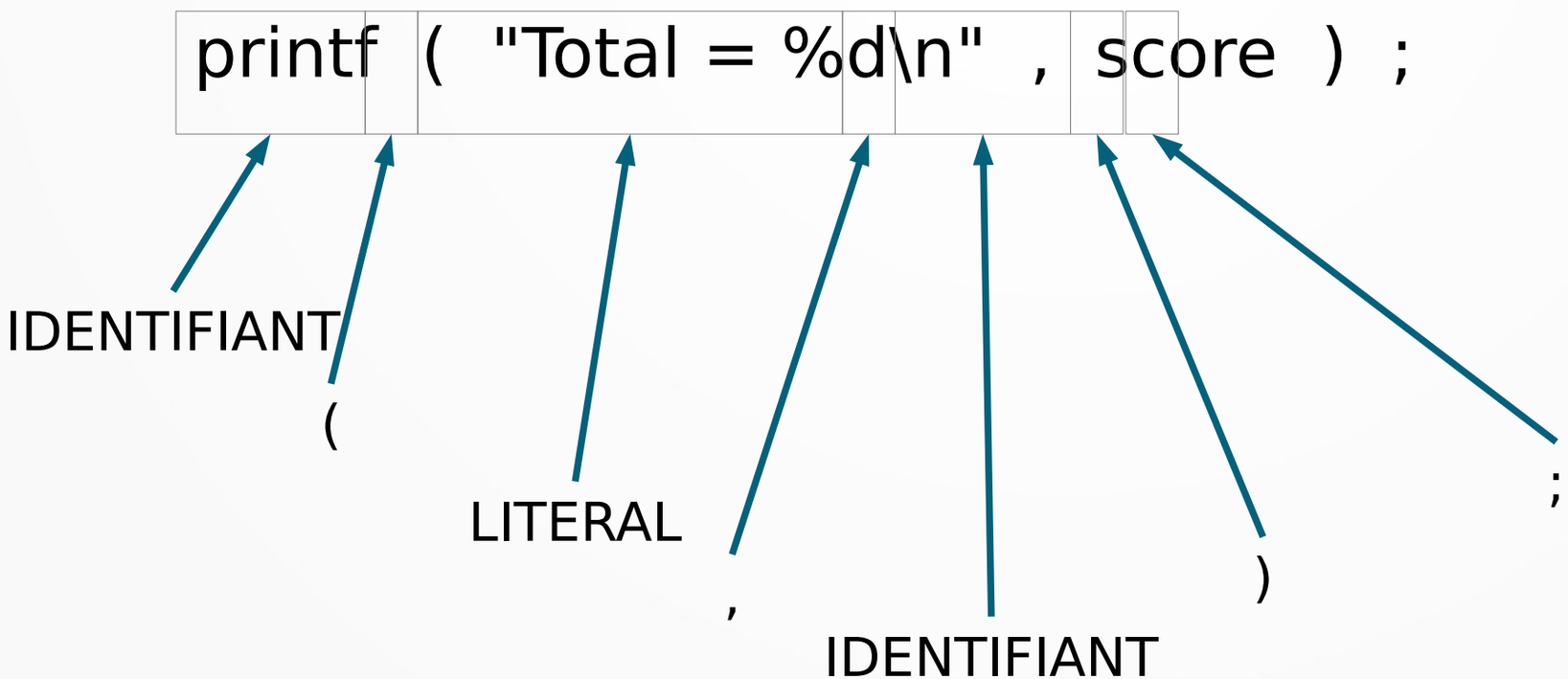
```
month_names = ['Januari', 'Februari', 'Maart',      # These are the
               'April',   'Mei',      'Juni',       # Dutch names
               'Juli',    'Augustus', 'September',  # for the months
               'Oktober', 'November', 'December']   # of the year
```

L2: Analyse lexicale

- Méthode (pour un langage de programmation)
  - un token pour chaque mot-clé du langage
    - Pattern = le mot en question, ex: pattern pour IF = if
  - Des tokens pour les opérateurs (+, -, *, ...) ou un seul
    - Un token OP avec pour pattern + ou * ou – ou / ...
  - Un token pour les identifiants (et un pour les types utilisateurs ?)
  - Un token ou plusieurs pour les constantes et litéraux
    - nombres, chaînes de charactères, ...
  - Des tokens pour chaque symbole de punctuation, parenthèses, etc...
    - Token {, avec comme pattern { (nommé d'après leur pattern, c'est plus simple)

L2: Analyse lexicale

- Exemples



L2: Analyse lexicale

- Un token est donc une structure, avec typiquement:
  - son type
  - sa position dans le flot de caractères
  - son lexème
  - des attributs supplémentaires
    - sa valeur (si c'est une constante numérique, on peut la calculer là)
    - son symbole (si c'est un identifiant, pour savoir s'il a déjà été défini)
    - d'autres peuvent être calculés: sa longueur (= celle du lexème)
    - d'autres peuvent être ajoutés (types additionels, numéro de ligne)

L2: Analyse lexicale

- Détails d'implémentation
  - On suppose que le preprocessing est déjà fait
    - Parce que le préprocessing est un espèce d'affreux bricolage aucquel personne ne veut toucher
  - L'entrée est bufferisée (sinon c'est très lent)
    - Mais on veut aussi ne pas avoir à gérer les buffers : que se passe-t-il si un lexème continue au-delà du buffer ?
  - Multilingue, Unicode:
    - C'est complexe: > 1 million de charactères différents
    - Nouvelle définition de choses comme les nombres et les espaces
      - Bien plus que ce que nous avions en ASCII

L2: Analyse lexicale

- Théorie des langages
  - L, M des ensembles de caractères, ε la chaîne vide
  - Décrire des mots possibles -> exprimer les patterns des tokens
  - Opérateurs
    - Union : L U M = { s | s dans L ou dans M }
    - Concaténation: L M = { st | s dans L et t dans M }
    - fermeture de Kleene: L * = { s | s dans L^i, i positif ou nul }
    - fermeture positive: L+ = { s | s dans L^i, i positif non nul }

L2: Analyse lexicale

- Exemple: L = { A, B, ..., Z, a, b, ..., z }, D = {0, 1, ..., 9 }
  - L U D : { A, ..., z, 0, ... , 9 }
  - LD : { A0, A1, ..., z9 }
  - L^4 : { AAAA, AAAB, ... , zzzz }
  - L* : { ε, A, ..., zA, ... }
  - L (L U D)* : { A, ... , d2, ... , pi , ... , score , ... } => identifiants
  - D+ : { 0, ... , 42, ... , 791 , ... } => nombres entiers positifs

L2: Analyse lexicale

- Un langage
  - un ensemble de mots potentiellement infini
    - identifiants, nombres
- Une expression régulière
  - Une expression qui décrit tous les mots d'un langage
    - Génération: sert à construire des mots
    - Vérification: permet de vérifier qu'un mot appartient à un langage
    - Calcul: possède un équivalent exécutable → un automate

L2: Analyse lexicale

- Expressions régulières
  - règles définissant des langages : r → L(r)
    - ε (chaîne vide) → L(ε)
    - Σ un alphabet (a dans Σ), L(a) = {a}
  - Construction
    - (r) → L(r)
    - (r) | (s) → L(r) U L(s)
    - (r)(s) → L(r)L(s)
    - (r)* → L(r)*
  - Précédence
    - (a) | ((b)*(c)) = a | b*c

L2: Analyse lexicale

- Extensions
  - + : r+ = r r*
  - ? : r? = r | ε
  - [ ] :
    - [abc] = a | b | c
    - [a-z] = a | b | ... | z
  - [^a] : tout sauf a
  - a{min, max} : a répété entre min et max fois
  - . : n’importe quoi

L2: Analyse lexicale

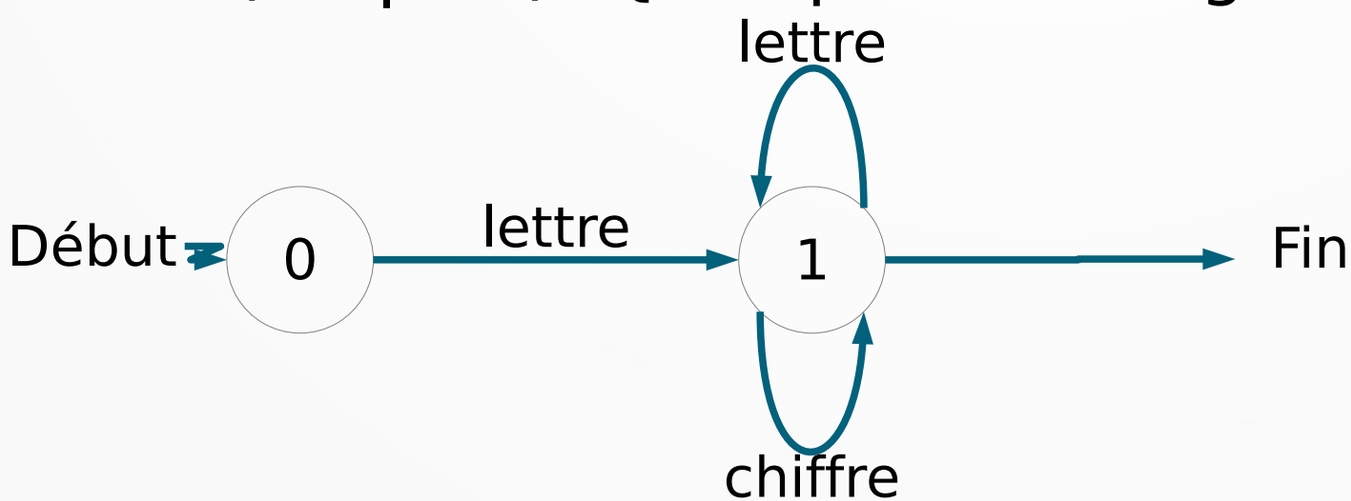
- Extensions (SmaCC : outil que nous allons utiliser)
  - \d Matches a digit, 0-9.
  - \D Matches anything that is not a digit.
  - \f Matches a form-feed character, "Character value: 12".
  - \n Matches a newline character, "Character value: 10".
  - \r Matches a carriage return character, "Character value: 13".
  - \s Matches any whitespace character, [\f\n\r\t\v].
  - \S Matches any non-whitespace character.
  - \t Matches a tab, "Character value: 9".
  - \v Matches a vertical tab, "Character value: 11"
  - \w Matches any letter, number or underscore, [A-Za-z0-9_].
  - \W Matches anything that is not a letter, number or underscore.
  - \xHexNumber Matches a ascii value or unicode code point

L2: Analyse lexicale

- Passage expression régulière → langage
- Décrire un motif pour un token avec une expression régulière
- Etape suivante :
  - Utiliser l’expression régulière pour reconnaître un mot du langage.
  - Utiliser l’équivalence expression régulière / automate à états finis

L2: Analyse lexicale

- Exemple d'une transcription:
  - L(L U D)* { l'ensembre des identifiants commençant par une lettre}
  - \w (\d | \w)* { l'expression régulière équivalente }



L2: Analyse lexicale

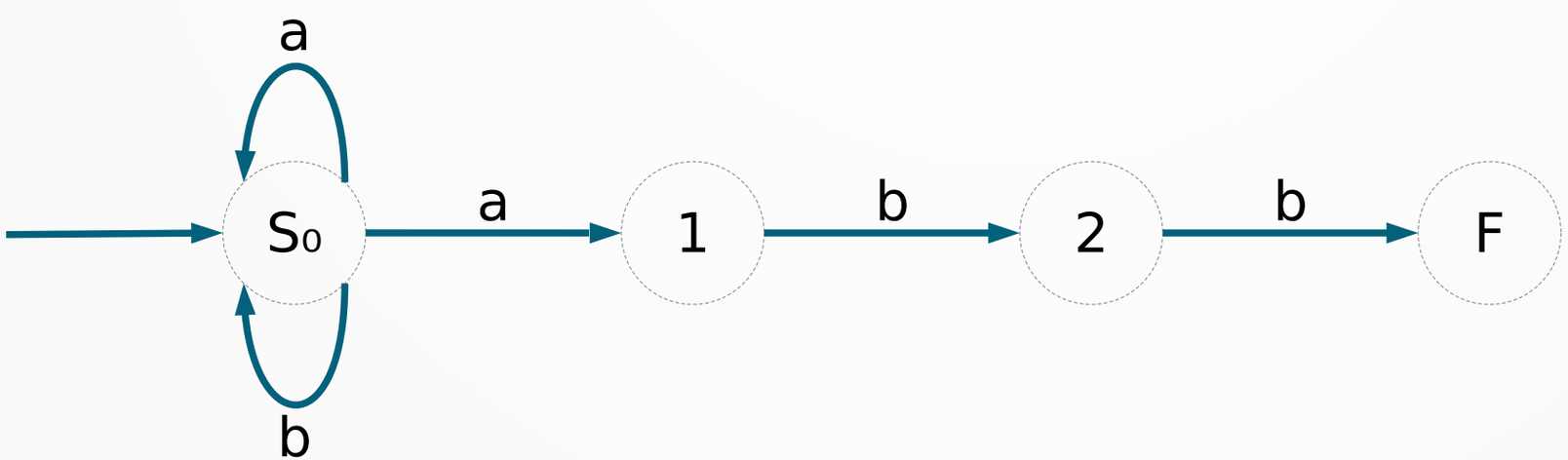
- Automates à états finis
  - Dit oui ou non sur une entrée
  - Peut être de deux types:
    - NFA (Non deterministic finite state automata)
    - DFA (deterministic finite state automata)
- Equivalence:
  - Automates ≡ expressions régulières ≡ langages réguliers

L2: Analyse lexicale

- Définition d'un NFA :
  - S : un ensemble d'états
  - Σ : un alphabet (un jeu de caractères).
    - Attention, ε n'appartient pas à Σ
  - T : fonction de transition S, Σ U { ε } → S
    - À un état courant et un caractère (ou le rien) associe un nouvel état
  - Début: S_0 dans S
  - Fin: F un ensemble d'états inclus dans S
    - Astuce : utile de pouvoir terminer en plusieurs états.
  - Interprétation: si à la fin du mot on est sur un état appartenant à F, alors le mot appartient au langage

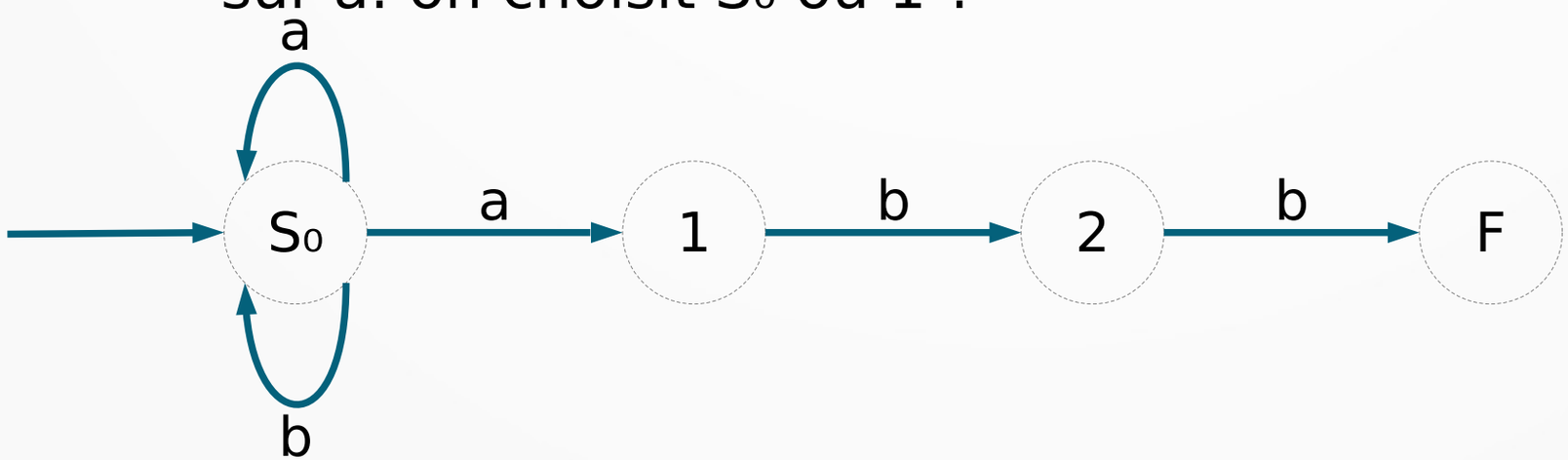
L2: Analyse Lexicale

- Exemple
  - (a|b)*abb



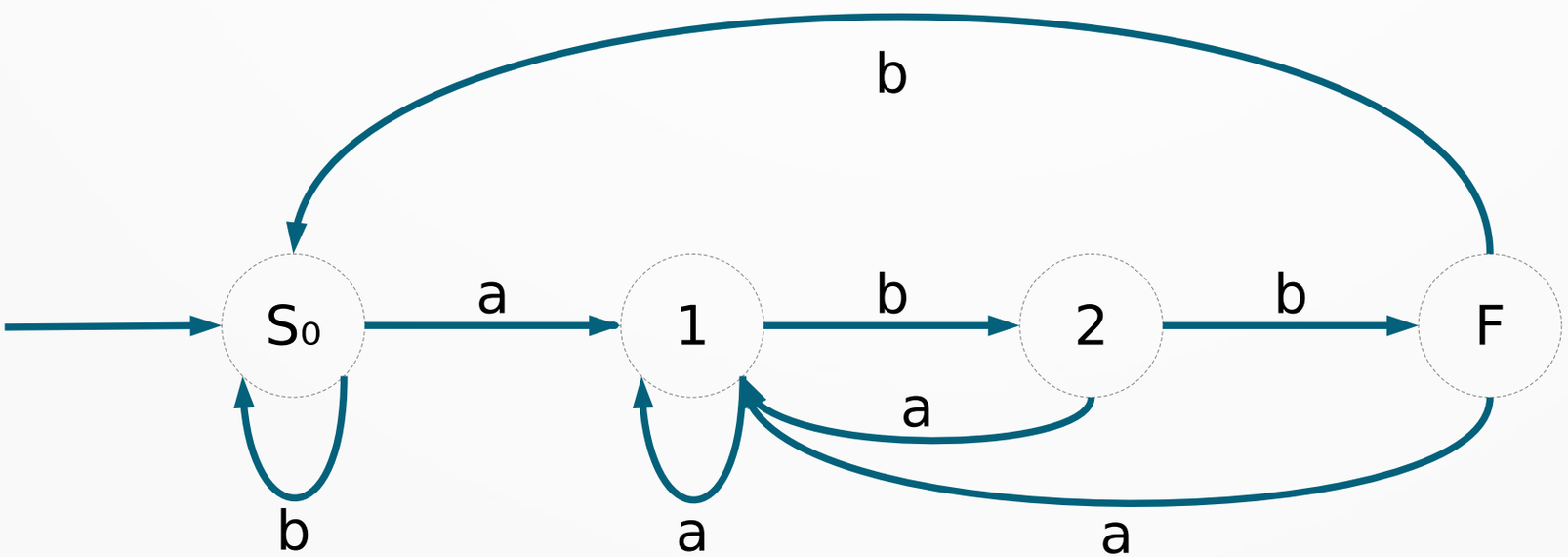
L2: Analyse Lexicale

- Implémentation
  - NFA : problème d'exécuter un automate non déterministe
  - Par exemple, sur cet automate:
    - sur a: on choisit S_0 ou 1 ?



L2: Analyse Lexicale

- Solution
  - Convertir le NFA en un DFA
  - Exécuter le DFA (c'est le programme)



L2: Analyse Lexicale

- Convertir un NFA en un DFA:
  - Méthode
    - Si le NFA après une entrée a_1, a_2, …, a_n peut être dans les états s_i, s_j, s_k, alors le DFA doit être dans un état $s' \equiv s_i, s_j, s_k$
- Pourquoi commencer par un NFA ?
  - Parce que dans un NFA, composer les regex est facile
  - Et qu'un analyseur lexical, c'est une composition de regex
    - Sous la forme pattern1 | pattern2 | pattern3

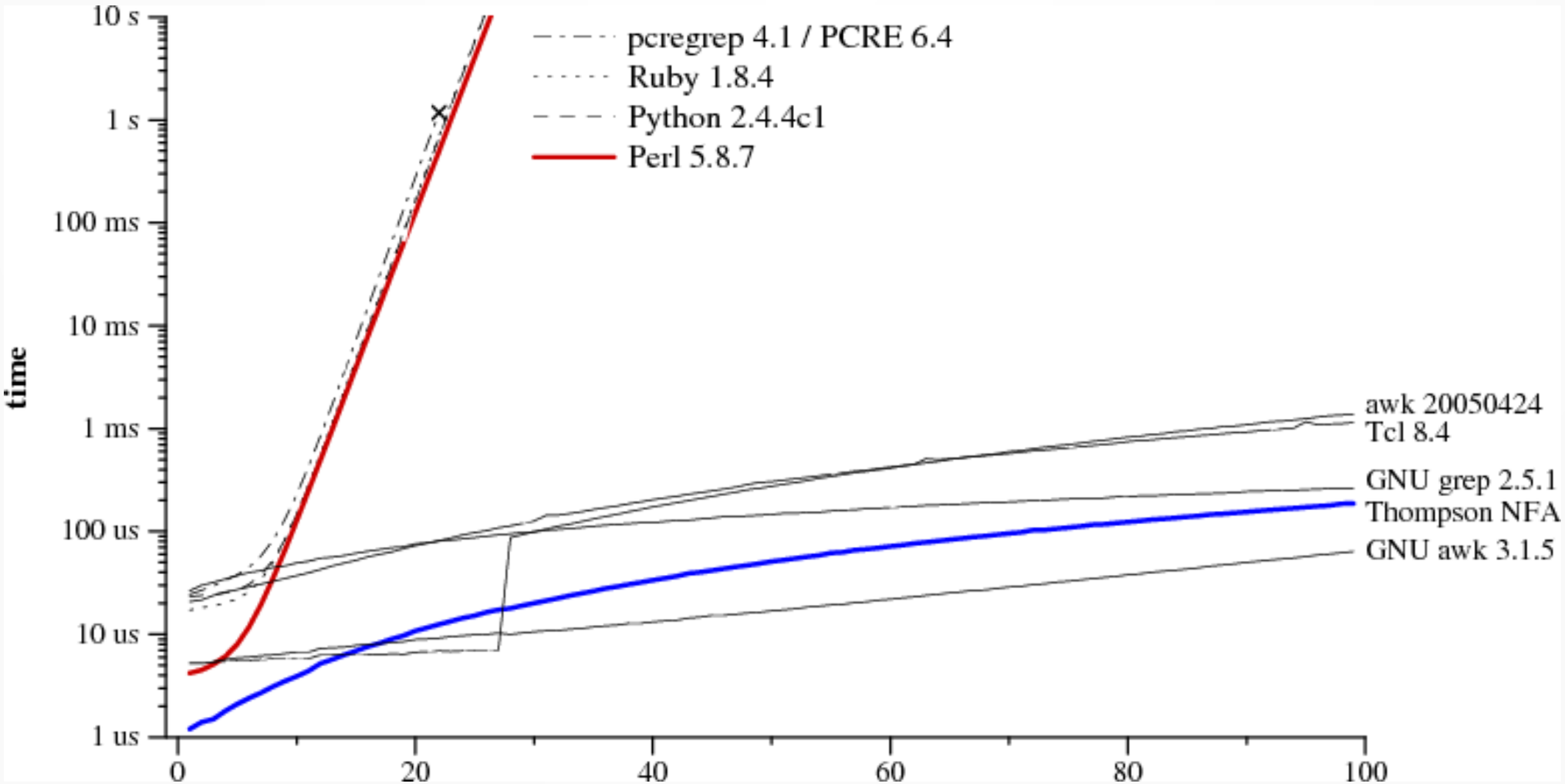
L2: Analyse Lexicale

- Détails d’implémentation
  - Un analyseur lexical est un | entre tous les motifs des tokens
    - Java: 108 types de tokens différents
  - Les états "fin" identifient le token
  - On peut ajouter des actions exécutées lorsque le token est reconnu
- Une implémentation simple (et rapide)
  - L’automate (le DFA) est une table (état / caractère entrant)
  - Le code complet < 20 lignes de code.

L2: Analyse Lexicale

- Différence analyseur lexical | librairie d’expressions régulières
  - Un analyseur lexical est rapide, pas de backtracking
  - Les librairies d’expressions régulières peuvent être lentes
    - Si vous utilisez la capture et les back references
      - extraction de sous-chaînes dans la pattern, et utilisation
      - (cat|dog)\1
    - Utilisation de backtracking pour beaucoup d’entre elles (Perl, Python)
      - Ou d’actions arbitraires (Java)
  - Conséquences:
    - Temps d’exécution exponentiels

L2: Analyse Lexicale



Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...), Russ Cox , rsc@swtch.com
January 2007