

Langages : interprétation et compilation

Pablo Rauzy

pr@up8.edu

pablo.rauzy.name/teaching/liec



UFR MITSIC / L3 informatique

Séance 1

Analyse lexicale

Analyse lexicale

- ▶ L'*analyse lexicale* est la première étape de la lecture d'un code source.
- ▶ La plupart des langages de programmation ont un code source textuel.
- ▶ C'est à dire qu'au niveau des données brutes, le code source est une chaîne de caractères.
- ▶ Le rôle de l'analyse lexicale est de découper cette chaîne *unités lexicales*.

- ▶ Une *unité lexicale* est en quelques sortes un “mot” du langage de programmation.
- ▶ Par la suite, l’analyse syntaxique vérifiera que ces “mots” sont utilisés pour former des phrases grammaticalement correctes.
- ▶ Après ça, l’analyse sémantique vérifiera que ces phrases ont effectivement un sens.

- Quand on parle d'*unité lexicale* on peut parler de quatre choses :
- un *lexème* : "+", "foo", "bar", "42", "if", "==" , "=", "true" ;
 - un *type* d'unité : Plus, Ident, Ident, Number, If, Equal, Assign, True ;
 - une *valeur* : soit une fonction "built-in", soit une référence à une table d'identifiants, soit directement une valeur (comme pour les nombres).
 - un *modèle* qui sert à spécifier l'unité lexicale : le lexème directement pour les mots clefs, "une suite de chiffre" pour les nombres entiers positifs, etc.

- ▶ On va donc avoir besoin d'au moins deux choses :
 1. spécifier les modèles,
 2. reconnaître ces modèles.

- ▶ Quel outil pourrait-on mettre en œuvre pour la première étape ?

- ▶ Une *expression régulière*, ou “*regex*”, est une notation qui décrit un ensemble de chaîne de caractères.
- ▶ On va s'en servir pour spécifier nos modèles de lexèmes.

- ▶ Un *alphabet* est un ensemble fini de symboles.
- ▶ On le note souvent Σ .
- ▶ Exemple : $\Sigma = \{a, b\}$.

- ▶ Un *mot* sur un alphabet Σ est une suite finie de symboles de Σ .
- ▶ On note ε le mot vide.
- ▶ Exemple : $\varepsilon, a, b, ab, aaaaa, ababababababab, bbba$.

- ▶ Un *langage* sur un alphabet Σ est un ensemble de mot sur Σ .
- ▶ On note \emptyset le langage vide (qui ne contient aucun mot).
- ▶ Exemple : $\emptyset, \{\varepsilon\}, \{a, aa, aaa, aaaa, aaaaa\}, \{a, b, ab, ba\}$.

- ▶ On peut utiliser des *opérations* sur les langages pour en construire de nouveaux.
- ▶ On distingue trois opérations :
 - le produit (ou concaténation),
 - la somme (ou l'union),
 - l'étoile de Kleene (ou fermeture de Kleene).

- ▶ Soit L et M deux langages sur un alphabet Σ .
- ▶ Le produit LM correspond au langage obtenu en concaténant deux à deux les éléments de L et M .
- ▶ Formellement, $LM = \{xy \mid x \in L \wedge y \in M\}$.

- ▶ Soit L et M deux langages sur un alphabet Σ .
- ▶ La somme $L \cup M$ correspond à l'union des deux langages L et M .
- ▶ Formellement, $L \cup M = \{x \mid x \in L \vee x \in M\}$.

- ▶ Soit L un langage sur un alphabet Σ .
- ▶ L'étoile L^* correspond à l'ensemble de concaténations possibles de mots de L .
- ▶ Formellement, $L^* = \{x_1 x_2 \cdots x_n \mid x_i \in L, n \geq 0 \in \mathbb{N}\}$.

1. Donnez l'alphabet et le langage des nombres écrits en binaire.
2. Soit Σ un alphabet et L le langage de tous les mots de un symbole.
 - Donnez le langage des mots d'exactly 3 symboles.
 - Donnez le langage des mots d'au moins 2 symboles.
 - Donnez le langage des mots de 2 ou 3 symboles.

- ▶ En pratique on utilise des notations similaires à celles des regexps dans les langages de programmation.
 - ϵ note le langage $\{\epsilon\}$,
 - Si $a \in \Sigma$, alors a note le langage $\{a\}$,
 - Si x et y sont deux regexps qui définissent respectivement les langages L et M , alors :
 - $(x)|(y)$ note le langage $L \cup M$,
 - $(x)(y)$ note le langage LM ,
 - $(x)^*$ note le langage L^* ,
 - (x) note le langage L , c'est à dire que les parenthèses ne changent rien.
 - L'ordre de priorité des opérateurs est $*$ puis concaténation puis $|$, et ils sont associatifs à gauche.

- Si x est une regexp qui définit le langage L , alors :
- $(x)^n$, raccourci pour $xx \cdots x$ (n fois), définit L^n ;
 - $(x)^+$, raccourci pour xx^* , définit LL^* ;
 - $(x)?$, raccourci pour $x|\varepsilon$, définit $L \cup \{\varepsilon\}$.

- Soit Σ l'ensemble des 26 lettres minuscules de l'alphabet latin. Donnez les expressions régulières les plus courtes possibles qui reconnaissent exactement :
1. "non", "noon", "nooon", "noooooon", etc.
 2. "oui", "ouioui", "ouiouioui", etc.
 3. "oui", "non", et "nan"
 4. "banane" et "patate".

- ▶ En théorie des langages formels, les langages réguliers sont ceux qu'on peut spécifier avec une expression régulière.
- ▶ Étant donné un alphabet Σ , l'ensemble des langages réguliers sur cet alphabet est le plus petit stable pour les opérations régulières et qui contient le langage vide \emptyset , les langages singleton du mot vide ε et de chaque symbole de Σ .

- Pouvez-vous donner un exemple de langage qui ne soit pas régulier ?

- ▶ Pouvez-vous donner un exemple de langage qui ne soit pas régulier ?
- ▶ Les *langages de Dyck* sont l'ensemble des mots bien parenthésés sur un alphabet composé de parenthèses ouvrantes et fermantes.
 - Soit l'alphabet $D = \{ (,) \}$.
 - $(())()$ est bien parenthésé, c'est un mot de Dyck.
 - $())(($ ne l'est pas.
- ▶ Ils ne sont pas réguliers : intuitivement on a besoin d'un "compteur" (d'une pile) pour les spécifier.
- ▶ Pourtant ce type de langage est assez important, en particulier pour la compilation, voyez-vous pourquoi ?

- ▶ Pouvez-vous donner un exemple de langage qui ne soit pas régulier ?
- ▶ Les *langages de Dyck* sont l'ensemble des mots bien parenthésés sur un alphabet composé de parenthèses ouvrantes et fermantes.
 - Soit l'alphabet $D = \{ (,) \}$.
 - $(())()$ est bien parenthésé, c'est un mot de Dyck.
 - $())(($ ne l'est pas.
- ▶ Ils ne sont pas réguliers : intuitivement on a besoin d'un "compteur" (d'une pile) pour les spécifier.
- ▶ Pourtant ce type de langage est assez important, en particulier pour la compilation, voyez-vous pourquoi ?
- ▶ On a besoin de vérifier la syntaxe des langages de programmation !
- ▶ Ce genre de problème se règlera donc lors de l'analyse syntaxique.

- ▶ La *hiérarchie de Chomsky* est une classification des *grammaires formelles* (et donc les langages formels engendrés par ces grammaires).
- ▶ Elle définit quatre types de grammaire :

- ▶ La *hiérarchie de Chomsky* est une classification des *grammaires formelles* (et donc les langages formels engendrés par ces grammaires).
- ▶ Elle définit quatre types de grammaire :
 - type 0, pour les langages récursivement énumérables

Machines de Turing

- ▶ La *hiérarchie de Chomsky* est une classification des *grammaires formelles* (et donc les langages formels engendrés par ces grammaires).
- ▶ Elle définit quatre types de grammaire :
 - type 0, pour les langages récursivement énumérables
 - type 1, pour les langages contextuels

Machines de Turing
Automates linéairement bornés

- ▶ La *hiérarchie de Chomsky* est une classification des *grammaires formelles* (et donc les langages formels engendrés par ces grammaires).
- ▶ Elle définit quatre types de grammaire :
 - type 0, pour les langages récursivement énumérables
 - type 1, pour les langages contextuels
 - type 2, pour les langages algébriques

Machines de Turing
Automates linéairement bornés
Automates à pile (non déterministes)

► La *hiérarchie de Chomsky* est une classification des *grammaires formelles* (et donc les langages formels engendrés par ces grammaires).

► Elle définit quatre types de grammaire :

- type 0, pour les langages récursivement énumérables
- type 1, pour les langages contextuels
- type 2, pour les langages algébriques
- type 3, pour les langages réguliers

Machines de Turing

Automates linéairement bornés

Automates à pile (non déterministes)

Automates finis

- ▶ La *hiérarchie de Chomsky* est une classification des *grammaires formelles* (et donc les langages formels engendrés par ces grammaires).
- ▶ Elle définit quatre types de grammaire :
 - type 0, pour les langages récursivement énumérables
 - type 1, pour les langages contextuels
 - type 2, pour les langages algébriques
 - type 3, pour les langages réguliers
- ▶ Ce sont ces derniers, les moins puissants, qui nous intéressent aujourd'hui.

Machines de Turing
Automates linéairement bornés
Automates à pile (non déterministes)
Automates finis

- ▶ Maintenant qu'on est capable de spécifier des modèles d'unités lexicales, on voudrait être capable de les reconnaître.
- ▶ Les conditions sont les suivantes :
 - on lit le code source un caractère à la fois,
 - on lit toujours autant qu'on peut (on est glouton),
 - on a le droit de regarder un caractère d'avance sans le consommer.

- ▶ Un travail préliminaire à l'implémentation de notre analyseur lexical est la réalisation de ce qu'on appelle un diagramme de transition.
- ▶ Le diagramme se présente sous forme de graphe :
 - les sommets correspondent à des états,
 - les arêtes (orientées) correspondent à des lectures de caractères, et sont étiquetées par ceux-ci,
 - un des états est l'état initial,
 - les états correspondant à la reconnaissance complète d'une unité lexicale sont finaux,
 - une étiquette spéciale \star correspond à un joker (qui lit n'importe quel caractère, auquel cas on a seulement regardé le caractère sans le consommer).

- ▶ Admettons qu'on ait un langage dans lequel on a :
 - les symboles =, +, -, *, /, ;, ==, <, >, <=, >=, et <> ;
 - des noms de variables (que des lettres) ;
 - des entiers.

- ▶ Dessinez un diagramme de transition pour les symboles.

- ▶ Regardons ensemble une implémentation “à la main” d’un analyseur lexical pour ce petit langage (fichier `hand.rkt`).

- Un *automate fini* est défini par :
- un ensemble fini d'états E ,
 - un alphabet Σ ,
 - une fonction de transition $tr : E \times \Sigma \rightarrow E$,
 - un état initial $s \in E$,
 - un ensemble d'états finaux $F \subseteq E$.

- ▶ Un *automate fini* est défini par :
 - un ensemble fini d'états E ,
 - un alphabet Σ ,
 - une fonction de transition $tr : E \times \Sigma \rightarrow E$,
 - un état initial $s \in E$,
 - un ensemble d'états finaux $F \subseteq E$.

- ▶ On peut représenter un automate sous forme de graphe :
 - chaque sommet correspond à un état,
 - il y a une arête de e_1 à e_2 ssi $\exists c \in \Sigma, tr(e_1, c) = e_2$.

- ▶ Un *automate fini* est défini par :
 - un ensemble fini d'états E ,
 - un alphabet Σ ,
 - une fonction de transition $tr : E \times \Sigma \rightarrow E$,
 - un état initial $s \in E$,
 - un ensemble d'états finaux $F \subseteq E$.
- ▶ On peut représenter un automate sous forme de graphe :
 - chaque sommet correspond à un état,
 - il y a une arête de e_1 à e_2 ssi $\exists c \in \Sigma, tr(e_1, c) = e_2$.
- ▶ Cela correspond exactement à nos diagrammes de transition !

- ▶ Il y a différentes façons d'implémenter un automate fini, plus ou moins efficace.
- ▶ On peut bien sûr l'implémenter comme on l'a fait précédemment, "en dur".
- ▶ On peut aussi choisir d'implémenter une fonction de transition générique qui travaille avec une structure représentant la table des transitions.
- ▶ Si cette fonction est très simple, l'exécution de l'automate sera plus efficace.
- ▶ Comme d'habitude en *complexité*, il y a un compromis entre *temps* et *espace*.

- ▶ Regardons ensemble une implémentation d'analyseur lexical sous forme d'automate (fichier `auto.rkt`).

- ▶ Comme vous avez pu le constater, construire la table de transition est long et fastidieux.
- ▶ L'outil `lex` a été créé pour ça en 1975 par Mike E. Lest et Eric Schmidt.
- ▶ Il s'agit à l'origine d'un outils propriétaire pour le langage C.
- ▶ Depuis, il y a une version libre développée par GNU (`flex`), et des implémentations ont été écrites pour virtuellement tous les autres langages.

- ▶ <https://docs.racket-lang.org/parser-tools/Lexers.html>
- ▶ Ou mieux : lancer `raco docs` puis allez voir la documentation des “parser-tools”.
- ▶ Regardons ensemble comment créer des lexers avec Racket (fichiers `lexer1.rkt` et `lexer.rkt`).

- ▶ Les outils de type `Lex` peuvent être utilisés pour écrire d'autres types de programme que des analyseurs lexicaux destinés aux compilateurs.
- ▶ Regardons ensemble comment utiliser le lexer de Racket pour écrire un programme de conversion automatique d'unité (fichier `conv.rkt`).