Analyses lexicale et syntaxique

Les étapes de la compilation

programme source

↓ analyse lexicale, analyse syntaxique

arbre de syntaxe abstraite

↓ analyses statiques (typage,. . .)

arbre décoré

↓ traduction**s**, optimisations

code exécutable (langage machine)

lecture du fichier source: on récupère un *flot de caractères*

Les deux étapes

- analyse lexicale

  $\boxed{\text{flot de caractères (source)}} \longrightarrow \boxed{\text{flot de } \textit{lexèmes}}$

  - lexème (*token*): "atome" du langage
  - typiquement:
    - mots-clefs (`let`, `begin`, `while`,...)
    - symboles réservés (`(`, `+`, `;;`, `;`,...)
    - identificateurs (`f`, `toto`, ...)
- ainsi `32*52+(let x = 5 in x*x)`

  → `INT(32), MULT, INT(52), ADD, LPAREN, LET, ID("x"), EGAL, INT(5), IN, ID("x"), MULT, ID("x"), RPAREN`

  (`INT` et `ID` ont un *attribut*, entier et chaîne de caractères respectivement)
- analyse syntaxique

  $\boxed{\text{flot de lexèmes}} \longrightarrow \boxed{\textit{arbre de syntaxe abstraite}}$

  →
  ```
  Add( Mult(Int(32), Int(52)),
       Let("x", Int(5), Mult(Var("x"), Var("x"))) )
  ```

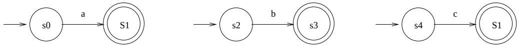
  - étape intermédiaire: arbre d'analyse syntaxique *(parse tree)*

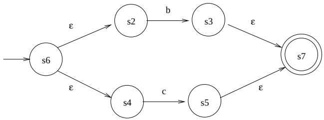
Analyse lexicale

chaque lexème est décrit par une *expression régulière*
principaux éléments (syntaxe de `ocamllex`):

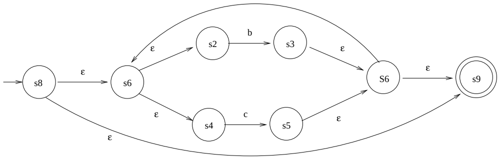
- caractère `'$'`, chaîne de caractères `"else"`
- intervalle `['0'-'9']` *(un chiffre)*
- disjonction `['\t' ' ']` *(tabulation ou espace)*
- juxtaposition `['A'-'Z']['a'-'z' 'A'-'Z']` *(mot de 2 lettres commençant par une majuscule)*
- répétitions: `+` signifie au moins 1, `*` zéro ou plus
  `['a'-'z']+['a'-'z' '0'-'9']*` *(ça commence par une lettre puis des lettres ou des chiffres)*

Expression régulière ↔ automate non déterministe

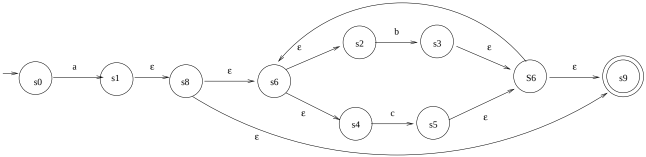




NFA pour b|c



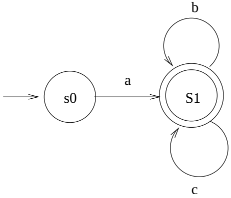
NFA pour (b|c)*



NFA pour a(b|c)*

Déterminisation, minimisation

- à partir de l'automate du transparent précédent, on dispose de procédures pour *déterminiser* l'automate (explosion du nombre d'états), puis le *minimiser*



- on aboutit à

- comment implémenter l'automate résultant?
  - une table (très creuse)

état	a	b	c	d	e
e1	-	e2	e3	-	-
e2	e4	-	-	-	-
e3	-	-	e3	-	-

  - éliminer les états: un plat de spaghetti, fait de `if` et de `goto`

`ocamllex`

- on met tout ça dans un fichier `lexer.mll`, en associant un lexème à chaque expression régulière
  - quand plusieurs règles s'appliquent, c'est celle qui donne le lexème le plus long qui gagne, et s'il y a encore ambiguïté c'est la première règle dans l'ordre où elles sont écrites
  - DÉMO `lexer.mll` *(lexer = analyseur lexical)*
- un outil comme `ocamllex` transforme une liste d'expressions régulières en un *automate* déterministe, représenté par sa table de transitions (fichier `lexer.ml`)
  - une transition: on reconnaît un caractère
  - état final: on a reconnu un lexème
  - `ocamllex`, mais aussi (surtout) `lex`, `flex`

Analyse syntaxique

- on change la structure: un flot (de lexèmes) devient un arbre
- la syntaxe est décrite par des *règles de grammaire*
  - les règles de grammaire font intervenir les lexèmes et des “variables” (les non terminaux)
  - exemple de grammaire:

    $$E \quad ::= \quad K \;\mid\; E + E \;\mid\; E * E \;\mid\; (E) \;\mid\; \texttt{let}\ Id = E\ \texttt{in}\ E$$

    - le $\mid$ se lit comme un “ou”
    - E: non terminal (il peut y en avoir plusieurs)
    - $K, \texttt{let}, Id, +, *, (,), \texttt{in}, =$: lexèmes
    - on parle de *grammaire hors contexte*
  - analyseur syntaxique (ou *parser*): applique les règles de grammaire pour *reconnaître* une suite de lexèmes

Reconnaître une séquence de lexèmes

- l'idée est de construire un *arbre de dérivation* permettant de reconnaître la suite de lexèmes
  - grammaire: $P \quad ::= \quad P + P \;\mid\; P * P \;\mid\; K \qquad K$ un entier
  - soit le flot `32, +, 26, *, 2`
  - on peut faire deux dérivations:

$$\begin{array}{lcl} P & \rightarrow & P + P \\ & \rightarrow & K_{32} + P \\ & \rightarrow & K_{32} + P * P \\ & \rightarrow\rightarrow & K_{32} + K_{26} * K_2 \end{array} \qquad \text{ou alors} \qquad \begin{array}{lcl} P & \rightarrow & P * P \\ & \rightarrow & P * K_2 \\ & \rightarrow & P + P * K_2 \\ & \rightarrow\rightarrow & K_{32} + K_{26} * K_2 \end{array}$$

    deux arbres différents: *ambiguïté*
  - aucune dérivation possible en revanche pour `9 * 1 + + 1`

Ce que fait le parser

$$E \quad ::= \quad E + E \;\mid\; E * E \;\mid\; (E) \;\mid\; a \;\mid\; b \;\mid\; c$$ a+b*c

pile	entrée	action
$\$$	$a+b*c\$$	shift
$\$a$	$+b*c\$$	reduce : $E \to a$
$\$E$	$+b*c\$$	shift
$\$E+$	$b*c\$$	shift
$\$E+b$	$*c\$$	reduce : $E \to b$
$\$E+E$	$*c\$$	shift **(très malin)**
$\$E+E*$	$c\$$	shift
$\$E+E*c$	$\$$	reduce : $E \to c$
$\$E+E*E$	$\$$	reduce : $E \to E*E$
$\$E+E$	$\$$	reduce : $E \to E+E$
$\$E$	$\$$	accept

- conflit ‘shift/reduce’: ambiguïté de la grammaire
- lever l’ambiguïté:
  - modifier la grammaire (p.ex. en ajoutant des non terminaux)
  - introduire des précédences *(ici le $*$ a une précédence plus grande que celle de la règle qu’il veut réduire, et donc on fait shift)*

`ocamlyacc`

- la version Caml de `yacc` (cf. aussi `bison`)
- générateur d'analyseurs syntaxiques
- on indique les lexèmes et les règles de grammaire
- + des artefacts, pour gérer les questions de priorité des règles, signaler des erreurs de syntaxe de manière informative, etc.
- $\boxed{\text{DÉMO}}$ `parser.mly`

Tout mettre ensemble

```
let lexbuf = Lexing.from_channel stdin
let parse () = Parser.main Lexer.token lexbuf
```

- on prend l'entrée standard (`stdin`, le clavier)
- on fait l'analyse lexicale, `lexbuf` est un flot de lexèmes
- l'analyse syntaxique renvoie un `Expr.expr` (cf. `parser.mly`)
  l'application `parse ()` renvoie la prochaine expression
  reconnue sur la saisie au clavier
- DÉMO `main.ml`
- cf. la doc de Caml (*Lexer and parser generators*)