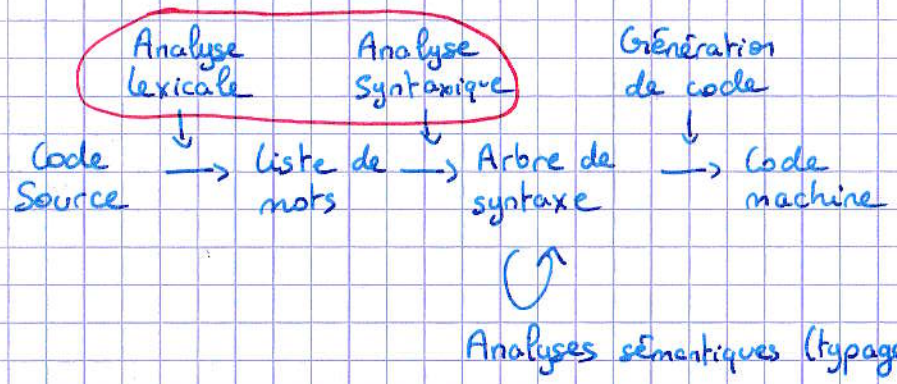
Leçon 25. Analyses lexicales et syntaxique. Applications.

Prérequis: automates, expressions régulières. OCaml.

I/ Motivation : compilation

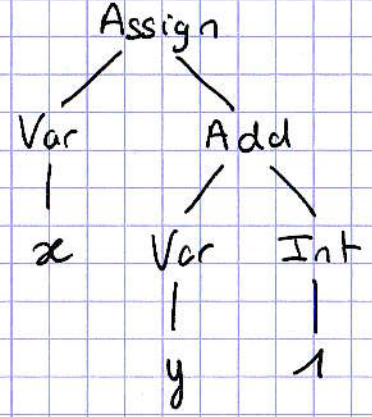


Exemple:

let x = y + 1;

-> LET VAR("x") EQUAL VAR("y") PLUS INT(1)

->



II/ Analyse lexicale

Objectif: reconnaître les mots du langage

1) Principe

Def: Les lexèmes sont les mots du langage
on les identifie par leur type, et éventuellement une valeur.

Ex: let -> LET, x -> VAR("x")

L'analyse repose sur la reconnaissance de motifs:
chaque lexème est décrit par une expression régulière, dont les mots sont reconnus par un automate.

Def: règle d'analyse lexicale
Etant donné A l'alphabet des caractères, les règles sont de la forme:

$$e \longrightarrow a$$

avec e une expression régulière sur A et a une action

NB: on peut simplement renvoyer le lexème, mais aussi renvoyer une autre information, modifier le texte...

Rq: une analyse lexicale gère une liste de règles, une pour chaque lexème. Si deux règles reconnaissent un mot, il y a ambiguïté. On considère donc les deux règles suivantes:
- on choisira la règle reconnaissant le mot le plus long
- en cas d'égalité, on suit l'ordre de priorité, prédéfini, des règles

2) Algorithme

Data : $(e_i \to a_i)_{i \in [\![1;n]\!]}$ la liste des règles, ordonnée
pour chaque e_i, A_i l'automate associé

lexeme_reconnu := -1
initialisation des A_i

Tant qu'il existe un A_i non bloqué

On fait lire le caractère suivant à tous les A_i
Si un A_i atteint l'état final
lexeme_reconnu := i
// si plusieurs A_i atteignent l'état final, on garde
// celui d'indice supérieur

// tous les automates sont bloqués
Si lexeme_reconnu == -1
erreur lexical, le texte ne correspond à aucun lexème

Sinon
// lexeme_reconnu correspond au lexeme le plus long
On applique a_i
on reprend à la suite de ce lexème

3) Applications

Coloration syntaxique, filtration des commentaires, reconnaissance de motif... -> Développement 1
Exemple ocamllex

II/ Analyse syntaxique

Objectif : • reconnaître si un texte est reconnu par la grammaire
• si oui, générer l'arbre de syntaxe abstraite

1) Contexte

Def : une grammaire est un ensemble de règles
$A \to a_1 \dots a_n$
$A \in N$: non-terminal, $a_i \in N \cup T$: non-terminal ou terminal

Ex : S -> let var = E
E -> var | int | E+E
(N = {S, E}, T = {let, var, =, int, +}

Def : une dérivation est l'application successive de règles

Ex : S -> let var = E
-> let var = E+E
-> let var = var+E
-> let var = var + int
=> let x=y+1 est reconnu par cette grammaire.

2) Analyse descendante

Principe : en partant de la première règle (l'axiome), retrouver les règles à appliquer pour générer le texte

NB : à ne pas confondre avec l'analyse ascendante, qui part du texte et cherche à revenir à l'axiome

Problème : si on a $S \to a|b$, comment savoir quelle règle appliquer ?

Solution : on regarde les k caractères suivants, c'est l'analyse $LL(k)$. Ici, on se contente de $LL(1)$

3) Analyse LL(1)

On utilise une pile pour mémoriser où on en est des règles appliquées.
S'il y a $c \in T$ au sommet de la pile, et $a \in T$ en tête du texte, et $a = c$, on avance dans la lecture (si $a \neq c$ c'est une erreur syntaxique)
Sinon, on regarde dans le tableau T qui indique pour chaque $X \in N$, et $c \in T$, s'il y a une règle $X \to \beta$, où β commence par c.
Quand la pile est vide, soit il ne reste plus que le symbole de fin (ici #) et c'est bon, soit on a une erreur syntaxique.

Construction du tableau

$$T(X, c) = \begin{cases} \beta \text{ si } X \to \beta \text{ et } c \in \text{premier}(\beta) \\ \gamma \text{ si } X \to \gamma, \text{null}(\gamma) \text{ et } c \in \text{suivant}(X) \end{cases}$$

avec $\text{null}(\alpha \in (N \cup T)^*) = \text{true}$ ssi $\alpha \to^* \varepsilon$
$\text{premier}(\alpha \in (N \cup T)^*) = \{a \in T \mid \exists w.\ \alpha \to^* aw\}$
$\text{suivant}(X \in N) = \{a \in T \mid \exists u, w.\ S \to^* uXaw\}$

NB : la grammaire est $LL(1)$ si pour chaque case il y a au plus une règle, i.e pas d'ambiguïté

Ex : Développement 2 : analyse $LL(1)$ pour un langage arithmétique

IV/ Applications

En tant que première étape de la compilation :
AST → code machine
analyse de typage de l'AST

Pour l'interprétation : code source → résultat

Pour la génération d'images :
LaTeX → PDF
svg → image

Pour l'affichage : HTML → affichage dans un navigateur

Ouvertures : Analyse ascendante, OCamlyacc
↳ Génération de code et optimisation
↳ Typage