Analyse Syntaxique descendante

- Rappels sur les grammaires
- Définition d'analyse descendante
- Un exemple simple en Ocaml
- Définition et construction de NULLABLE, FIRST, FOLLOW
- Construction de la table LL(1)
- Transformations de grammaires:
  - desambigüation par précédence d'opérateurs
  - derecursivisation gauche,
  - factorisation a gauche
- Un exemple complet

Rappels sur les grammaires: I

Introduite par Chomsky pour traiter le langage naturel, une grammaire est un quadruplet $G = (\Sigma, V_N, S, \mathcal{P})$, où:

Σ est un alphabet fini de “terminaux”,

V_N est un ensemble fini de symboles “non terminaux”, avec $\Sigma \cap V_N = \emptyset$
On écrit V pour $\Sigma \cup V_N$.

S est un non terminal distingue nomme le “symbole de départ” (“start symbol”)

$\mathcal{P}$ est une relation finie sur ${V_N}^+ \times V^*$ qui définit les “règles” (“productions”) de la grammaire.

Une règle $(\alpha, \beta) \in \mathcal{P}$ s’écrit aussi $\alpha \rightarrow \beta$.

Rappels sur les grammaires: II

On classifie les grammaires selon la forme de $\mathcal{P}$:

type 0 $\mathcal{P}$ quelconque

type 1 (contextuelles) pour toute règle $\alpha \to \beta$ on a $|\alpha| \leq |\beta|$.

type 2 (libres de contexte) règle de la forme $A \to \beta$.

type 3 (rationnelles) règle de la forme $A \to a\beta$ ou $A \to a$ avec a terminal.

On s'intéresse pour les langages de programmation a des grammaires de type 2.
N.B.: il existent d'autres systèmes de production, comme les L-systems utilisés en biologie.

Rappels sur les grammaires: III

Si $\alpha \to \beta$ est une règle, on peut alors l'appliquer a n'importe laquelle séquence $x\alpha y$ pour obtenir $x\beta y$ et on écrit alors

$$x\alpha y \Rightarrow x\beta y$$

On écrira $w \Rightarrow^* z$, s'il existent $w_1, w_2, \ldots w_n$, avec $n \geq 1$, t.q. $w = w_1 \Rightarrow w_2 \ldots \Rightarrow w_n = z$.

Le langage engendre par une grammaire G est

$$L(G) = \{w | S \Rightarrow^* w\}$$

Résultats fondamentaux:

Lemma 1.1 (Lemme de la pompe) *Soit L un langage libre de contexte. Il existe p et q dépendant de L tel que pour tout mot z de L dont la longueur $|z|$ excède p, il existe une décomposition $z = uvwxy$ avec $|vwx| \leq q, |vx| \neq 0$ et pour tout n, $uv^n wx^n y$ est dans L.*

Conséquences:

- $\{wcw|w \in (a|b)^*\}$ pas libre de contexte
  (ex: déclaration avant utilisation des identificateurs)
- $\{a^n b^m c^n d^m | n, m \geq 1\}$ pas libre de contexte
  (ex: correspondance entre paramètres formels et actuels d'une procédure)
- $\{a^n b^n c^n | n, m \geq 0\}$ pas libre de contexte
  (ex: ancienne technique pour indiquer les mots soulignes)

Arbre de dérivation, ambiguïtés

Définition 1.2 (Arbre de dérivation) *On peut représenter une dérivation $A \Rightarrow^* w$ (avec $w \in \Sigma^*$) comme un arbre ayant pour noeuds internes des non terminaux, pour feuilles des terminaux et des arcs reliant un noeud X a des fils $W_1, \ldots W_n$ si il y a une règle $X \to X_1 \ldots X_n$.*

Définition 1.3 (Grammaire ambiguë) *Une grammaire G est ambiguë s'il existe un mot $w \in L(G)$ avec au moins deux arbres de dérivation différents.*

Ex: le mot $1 + 1 + 1$ dans la grammaire

$$\begin{aligned} E &\to E + E \\ E &\to 1 \end{aligned}$$

De façon équivalente, une grammaire est ambiguë si elle permette deux *dérivations gauches* différentes pour un même mot w.

Backus Naur Form (BNF)

La “forme normale de Backus” est une extension du formalisme des grammaires libre de contexte qui n’ajoute pas de pouvoir expressif (on ne définit pas une classe plus large de langages), mais qui permet des définitions plus concises.

A la structure des règles on ajoute les notations suivantes:

étoile de Kleene : on peut écrire

$$A \to wx^*z$$

a la place de

$$A{\to}wBz \qquad B{\to}xB \qquad B{\to}x \qquad B{\to}\epsilon$$

option : on peut écrire

$$A \to w[x]z$$

a la place de

$$A \to wBz \qquad B \to x \qquad B \to \epsilon$$

cas : on peut écrire

$$A \to w_1 | w_2 \ldots | w_n$$

a la place de

$$\begin{aligned} A &\to w_1 \\ &\to \vdots \\ A &\to w_n \end{aligned}$$

Bien entendu, il faudra faire attention a ne pas mélanger les symboles BNF avec les terminaux de Σ. Dans l'énoncé du projet, par exemple, on a pris soin de distinguer tout symbole *s* de Σ en l'écrivant '*s*', comme pour les caractères de Ocaml. Mais s'il n'y a pas d' ambiguïté, on ne se fatiguera pas a faire la distinction.

Analyse descendante

Le problème de l'analyse est celui de prendre une grammaire G, et de construire un algorithme capable, pour toute chaîne w de terminaux, de décider si elle appartient ou non a $L(G)$. De plus, on demande de reconstruire un arbre de dérivation pour w dans G en cas de réponse affirmative.

Certains langages permettent de résoudre ce problème a l'aide d'une méthode fort simple, l'analyse dite *descendante*.

Cette méthode cherche a construire une dérivation gauche directement a partir du symbole initial, ou de façon équivalente, cherche a construire un arbre de dérivation a partir de la racine avec une exploration en profondeur d'abord de gauche a droite.

Pour cela, l'analyseur doit pouvoir décider, en s'aidant éventuellement avec les prochains k symboles en entrée, quelle est la prochaine production dans la dérivation gauche que l'on reconstruit.

Vue la “distance” entre le terminal que l'on voit en entrée et la production a choisir, il n'y a pas trop de chance pour que la méthode fonctionne (en particulier toute grammaire ambiguë posera des problèmes), mais si on réussit, alors le langage de la grammaire est dans la classe $LL(k)$ (langages analysables en parcourant l'entrée de gauche (L) a droite et en reconstituant une dérivation gauche (L)).

Exemple OcamlLex

Voyons un exemple de grammaire $LL(1)$ et un analyseur construit a la main en Ocaml:

$$\begin{array}{rcl} S & \rightarrow & E\ eof \\ E & \rightarrow & int\ E' \\ E' & \rightarrow & +\ int\ E' | \epsilon \end{array}$$

```
exception Parse_Error;;
type token = Int | PLUS | Eof;;
let descente gettoken =
 let tok = ref (gettoken()) in let advance () = tok := gettoken() in
 let eat t = if (!tok = t) then advance() else raise Parse_Error in
 let check t = if (!tok = t) then () else raise Parse_Error in
 let rec ntS () = (ntE(); check(Eof))
 and ntE () = match !tok with  Int -> (eat(Int); ntE'())
                         |  _   -> raise Parse_Error
 and ntE' () = match !tok with PLUS -> (eat(PLUS);eat(Int); ntE'())
                         |  _      -> () (* transition vide *)
in ntS();;

(* un lexeur bidon vite fait *)
let gettoken_of l = let data = ref l in
    let f () = let t = List.hd !data in data:= List.tl !data; t in f;;

descente (gettoken_of [Int;PLUS;Int;PLUS;Int;Eof]);;
```

Analyseurs $LL(1)$: choisir la production en regardant le prochain token

Pour réaliser un analyseur LL(1), il s'agit de remplir une table de la forme

	t_1	$\cdots$	t_m
X_1		$\cdots$	
$\vdots$		$\cdots$	
X_n		$\cdots$	

en mettant une règle de production $X_i \to \gamma$ dans la case (X_i, t_k) si le fait de voir le token t_k indique qu'on peut appliquer la règle de production $X_i \to \gamma$. S'il n'y a pas de cases avec plus d'une entrée une fois le remplissage fini, on aura réussi et on pourra implémenter l'analyseur.

NULLABLE, FIRST, FOLLOW

Question: comment remplir les cases?
Réponse: pour cela on a besoin de calculer pour les symboles de la grammaire trois ensembles:

NULLABLE(X) vrai ssi X peut produire la chaîne vide (ϵ)

FIRST(X) ensemble de symboles terminaux qui peuvent paraître en première position dans une chaîne γ dérivable à partir de X

FOLLOW(X) ensemble de symboles terminaux t qui peuvent paraître immédiatement après X dans une dérivation (i.e. il existe une dérivation contenant Xt, ce qui peut arriver aussi si elle contient $XYZt$ et Y et Z sont nullable).

Définition *formelle* de NULLABLE, FIRST, FOLLOW

On peut voir ces ensembles comme des relations, et alors

- Nullable est la plus petite relation Nu t.q.

$$Nu = Nu \cup \{(X, vrai) | X \rightarrow \in \mathcal{P}\}$$

$$Nu = Nu \cup \{(X, vrai) | X \rightarrow Y_1 \dots Y_n \in \mathcal{P}, \text{ avec } (Y_i, vrai) \in Nu\}$$

- First est la plus petite relation Fi telle que

$$Fi = Fi \cup \{(a, a) | a \in \Sigma\}$$

$$Fi = Fi \cup \{(X, a) | \begin{array}{l} X \rightarrow Y_1 \dots Y_n \in \mathcal{P}, \text{ et existe } i \text{ avec} \\ Y_1, \dots Y_{i-1} \text{ nullable et } (Y_i, a) \in Fi \end{array} \}$$

Définition *formelle* de NULLABLE, FIRST, FOLLOW

Enfin, Follow est la plus petite relation Fo telle que

$$Fo \;=\; Fo \cup \{(Y,a) | \begin{array}{l} X \rightarrow Y_1 \ldots Y_i Y Y_{i+1} \ldots Y_n \in \mathcal{P}, \\ \text{avec } Y_{i+1}, \ldots, Y_n \text{ nullable et } (X,a) \in Fo \end{array} \}$$

$$Fo \;=\; Fo \cup \{(Y,a) | \begin{array}{l} X \rightarrow Y_1 \ldots Y_i Y Y_{i+1} \ldots Y_j Y_{j+1} \ldots Y_n \in \mathcal{P}, \\ \text{avec } Y_{i+1}, \ldots, Y_j \text{ nullable et } (Y_{j+1},a) \in First \end{array} \}$$

On peut considérer ces équations comme définissant ces ensembles récursivement, et on peut calculer la plus petite solution par itération jusqu'au plus petit point fixe en partant de la relation vide.

N.B.: ce qui fait que ce point fixe existe toujours et est atteignable de cette sorte est la continuité (par rapport a une topologie bien choisie) des opérations ensemblistes utilisées dans la définition.

Construction de la table $LL(1)$

Une fois calcules Nullable(X), First(X) et Follow(X) pour tout symbole, il est facile de calculer aussi Nullable et First sur une séquence de symboles:

Nullable($X_1 \ldots X_n$**)** est vrai ssi Nullable(X_i) est vrai pour $1 \leq i \leq n$;

First($X_1 \ldots X_k$**)** est First(X_1) si X_1 n'est pas nullable

First($X_1 \ldots X_k$**)** est $First(X_1) \cup First(X_2 \ldots X_k)$ si X_1 est nullable

Et on peut finalement remplir notre table

	t_1	$\cdots$	t_m
X_1		$\cdots$	
$\vdots$		$\cdots$	
X_n		$\cdots$	

de la façon suivante, on analyse toute production $X_i \rightarrow \gamma$ et

- on met $X_i \rightarrow \gamma$ dans la case (X_i, t) si $t \in First(\gamma)$
- on met $X_i \rightarrow \gamma$ dans la case (X_i, t) si $t \in Follow(X_i)$ et $Nullable(\gamma)$

Si la table, après remplissage, n'a pas d'entrées multiples, alors la grammaire est analysable par l'analyseur et le langage est dans la classe $LL(1)$.

Transformations préliminaires: élimination de l'ambiguïté

L'existence d'une grammaire non ambiguë pour un langage donne L n'est pas décidable. Cependant, dans les cas les plus fréquents, on sait éliminer l'ambiguïté assez facilement:

Exemple: on peut réécrire la grammaire (ambiguë)

$$E \rightarrow E + E | E * E | (E) | int | id$$

en imposant une priorité sur les opérateurs et une associativité, comme:

$$\begin{array}{lcl} E & \rightarrow & E + T | T \\ T & \rightarrow & T * F | F \\ F & \rightarrow & (E) | int | id \end{array}$$

qui reconnaît le même langage, mais n'est pas ambiguë.

Transformations préliminaires: derecursivisation

Une grammaire recursive a gauche pose toujours de problèmes pour $LL(1)$: si on a deux productions $E \rightarrow E\alpha$ et $E \rightarrow \beta$, alors il y aura toujours une double entrée dans les cases (E, t) pour $t \in First(\beta)$.

Mais on sait éliminer la récursion a gauche de toute grammaire: toute production

$$X \rightarrow X\gamma_1 | \ldots | X\gamma_n | \alpha_1 | \ldots | \alpha_m$$

peut être remplacée par les productions (équivalentes, si X' est un symbole nouveau) :

$$\begin{array}{rcl} X & \rightarrow & \alpha_1 X' | \ldots | \alpha_m X' \\ X' & \rightarrow & \gamma_1 X' | \ldots | \gamma_n X' | \epsilon \end{array}$$

Transformations préliminaires: factorisation gauche

Un autre exemple de problème pour $LL(1)$ apparaît si on a deux productions $E \rightarrow \alpha E'$ et $E \rightarrow \alpha E''$ qui ont le même préfixe α: l'analyseur ne saura pas non plus dans ce cas faire le bon choix.

On sait améliorer la situation en "factorisant" a gauche le préfixe commun: toute production

$$X \rightarrow \alpha\beta_1|\ldots|\alpha\beta_k|\gamma$$

peut être remplacée par les productions (équivalentes, si X' est un symbole nouveau) :

$$\begin{array}{rcl} X & \rightarrow & \alpha X'|\gamma \\ X' & \rightarrow & \beta_1|\ldots|\beta_k \end{array}$$

Transformations préliminaires: factorisation gauche, II

Voyons un exemple bien connu:

$$X \rightarrow if\ E\ then\ S\ else\ S | if\ E\ then\ S$$

devient

$$\begin{aligned} X &\rightarrow if\ E\ then\ S\ X' \\ X' &\rightarrow else\ S | \epsilon \end{aligned}$$

N.B.: ce n'est pas LL(1) non plus, mais on sait mieux gérer ce deuxième cas que le premier (ex. en donnant priorité au cas "else")

Un exemple complet

Le langage des expressions arithmétiques

On veut analyser

$$\begin{array}{rcl} S & \rightarrow & Eeof \\ E & \rightarrow & E+E|E*E|(E)|int|id \end{array}$$

Cette grammaire est ambiguë, donc nous allons appliquer notre technique de desambiguation.

Un exemple complet: desambiguation

En imposant que * lie plus que + et qu'on associe a gauche, on obtient la grammaire non ambiguë:

$$\begin{array}{rcl} S & \rightarrow & Eeof \\ E & \rightarrow & E+T|T \\ T & \rightarrow & T*F|F \\ F & \rightarrow & (E)|int|id \end{array}$$

Mais cette grammaire est recursive a gauche, donc ...

Un exemple complet: derecursivisation

En derecursivisant, on arrive a:

$$
\begin{array}{rcc}
S & \rightarrow & Eeof \\
E & \rightarrow & TE' \\
E' & \rightarrow & +TE'|\epsilon \\
T & \rightarrow & FT' \\
T' & \rightarrow & *FT'|\epsilon \\
F & \rightarrow & (E)|int|id
\end{array}
$$

On va maintenant calculer Nullable, First et Follow dans l'ordre

Calcul de Nullable

$Iteration$	S	E	E'	T	T'	F
0	$false$	$false$	$false$	$false$	$false$	$false$
1	$false$	$false$	$true$	$false$	$true$	$false$
$2(fini)$	$false$	$false$	$true$	$false$	$true$	$false$

Calcul de First

On sait que :

	S	E	E'	T	T'	F
$Nullable$	$false$	$false$	$true$	$false$	$true$	$false$

On calcule First:

$Iteration$	S	E	E'	T	T'	F
0						
1			$+$		$*$	$(, int, id$
2			$+$	$(, int, id$	$*$	$(, int, id$
3		$(, int, id$	$+$	$(, int, id$	$*$	$(, int, id$
4	$(, int, id$	$(, int, id$	$+$	$(, int, id$	$*$	$(, int, id$
$5 = 4$	$(, int, id$	$(, int, id$	$+$	$(, int, id$	$*$	$(, int, id$

Calcul de Follow

On sait que :

	S	E	E'	T	T'	F
$Nullable$	$false$	$false$	$true$	$false$	$true$	$false$
$First$	$(, int, id$	$(, int, id$	$+$	$(, int, id$	$*$	$(, int, id$

On calcule Follow:

$Iteration$	S	E	E'	T	T'	F
0						
1		$eof,)$		$+$		$*$
2		$eof,)$	$eof,)$	$+, eof,)$	$+$	$*, +, eof,)$
3		$eof,)$	$eof,)$	$+, eof,)$	$+, eof,)$	$*, +, eof,)$
$4 = 3$		$eof,)$	$eof,)$	$+, eof,)$	$+, eof,)$	$*, +, eof,)$

La table de l'automate

On peut finalement construire la table de l'automate: On sait que :

	S	E	E'	T	T'	F
$Nullable$	$false$	$false$	$true$	$false$	$true$	$false$
$First$	$(, int, id$	$(, int, id$	$+$	$(, int, id$	$*$	$(, int, id$
$Follow$		$eof,)$	$eof,)$	$+, eof,)$	$+, eof,)$	$*, +, eof,)$

On construit alors:

	$+$	$*$	int	$($	$)$	eof
S			$S \to Eeof$	$S \to Eeof$		
E			$E \to TE'$	$E \to TE'$		
E'	$E' \to +TE'$				$E' \to$	$E' \to$
T			$T \to FT'$	$T \to FT'$		
T'	$T' \to$	$T' \to *FT'$			$T' \to$	$T' \to$
F			$F \to int$	$F \to (E)$		

Enfin, notre programme d'analyse en Ocaml

```
exception Parse_Error;;
type token = Int | PLUS | TIMES | LPAR | RPAR | Eof;;
let descente gettoken =
 let tok = ref (gettoken()) in
 let advance () = tok := gettoken() in
 let eat t = if (!tok = t) then advance() else raise Parse_Error in
 let check t = if (!tok = t) then () else raise Parse_Error
 in
 let rec ntS () = match !tok with
                            Int  -> (ntE(); check(Eof))
                          | LPAR -> (ntE(); check(Eof))
                          | _    -> raise Parse_Error
 and ntE () = match !tok with
                            Int  -> (ntT(); ntE'())
                          | LPAR -> (ntT(); ntE'())
                          | _    -> raise Parse_Error
 and ntE' () = match !tok with
                            PLUS -> (eat(PLUS); ntT(); ntE'())
                          | RPAR -> () (* transition vide *)
                          | Eof  -> () (* transition vide *)
                          | _    -> raise Parse_Error
 and ntT () = match !tok with
                            Int  -> (ntF();ntT'())
                          | LPAR -> (ntF();ntT'())
                          | _    -> raise Parse_Error
 and ntT' () = match !tok with
                            TIMES -> (eat(TIMES); ntF(); ntT'())
                          | PLUS  -> () (* transition vide *)
```

```
                         |  RPAR  -> () (* transition vide *)
                         |  Eof   -> () (* transition vide *)
                         |  _     -> raise Parse_Error
 and ntF () = match !tok with
                            Int  -> (eat(Int))
                         |  LPAR -> (eat(LPAR);ntE();eat(RPAR))
                         |  _    -> raise Parse_Error
in ntS();;
```

Enfin, notre programme d'analyse en Ocaml, avec traitement d'erreurs

```
(* on assume que l'on nous passe un analyseur lexical gettoken *)
exception Parse_Error;;
type token = Int | PLUS | TIMES | LPAR | RPAR | Eof;;
let descente gettoken =
 let position = ref 1 and tok = ref (gettoken()) in
 let advance () = tok := gettoken(); position := !position+1 in
 let eat t = if (!tok = t) then advance() else raise Parse_Error in
 let check t = if (!tok = t) then () else raise Parse_Error
 in
 let perror s = print_string ("Expecting "^s^" at position: ");
                print_int !position; print_newline(); raise Parse_Error in
 let rec ntS () = match !tok with
                  Int  -> (ntE(); check(Eof))
                | LPAR -> (ntE(); check(Eof))
                | _    -> perror "int,("
 and ntE () = match !tok with
                  Int  -> (ntT(); ntE'())
                | LPAR -> (ntT(); ntE'())
                | _    -> perror "int,("
 and ntE' () = match !tok with
                  PLUS -> (eat(PLUS); ntT(); ntE'())
                | RPAR -> () (* transition vide *)
                | Eof  -> () (* transition vide *)
                | _    -> perror "+,),eof"
 and ntT () = match !tok with
                  Int  -> (ntF();ntT'())
                | LPAR -> (ntF();ntT'())
                | _    -> perror "int,("
```

```
 and ntT' () = match !tok with
                   TIMES -> (eat(TIMES); ntF(); ntT'())
                 | PLUS  -> () (* transition vide *)
                 | RPAR  -> () (* transition vide *)
                 | Eof   -> () (* transition vide *)
                 | _     -> perror "+,*,),eof"
 and ntF () = match !tok with
                   Int  -> (eat(Int))
                 | LPAR -> (eat(LPAR);ntE();eat(RPAR))
                 | _    -> perror "int,("
in ntS();;
```