Chapitre 5 Théorie des grammaires et du parsing

Panorama de la suite du cours

Grammaires hors-contextes (GHC) :

- équivalent à la notion de langage algébrique.
- donne un procédé de calcul, par *réécriture*, plus "souple" que calcul de $\bigcup_{i\in\mathbb{N}} F^i(\emptyset)$.
  NB : algos efficaces pour décider si $w \in L$.
- analyse LL(1) des GHC : algo efficace d'analyse syntaxique qui généralise celui vu sur la notation préfixe.

Objectif du cours GHC LL(1) attribuées = méta-language pour générer automatiquement des interpréteurs.

Plan de la section 5.1

Introduction

On introduit ici réécriture sur les mots = un "*modèle de calcul*" à la base des paradigmes modernes de programmation des analyseurs syntaxiques !

Réécriture de mots = réécriture sur structure de liste.
Généralisé en :

- réécriture sur des arbres (codés par des termes avec variables).
- réécriture sur des graphes.

$\rightsquigarrow$ à la base *des paradigmes de programmation par règles & contraintes* !

Nombreuses applications en "*intelligence artificielle*" : langage de requête dans bases de données, bio-informatique, démonstration mathématique automatisée, etc.

Définition (Thue 1914)

Un **système de réécriture de mots** R est un couple $(\mathcal{V}, \mathcal{R})$ tq :

- $\mathcal{V}$ ensemble *fini* de symboles, appelé **alphabet** (ou *vocabulaire*) ;
- et, $\mathcal{R}$ sous-ensemble *fini* de $\mathcal{V}^* \times \mathcal{V}^*$.
  Un couple $(l, r) \in \mathcal{R}$ s'appelle une **règle de réécriture** (ou *production*) et se note plutôt "$l \to r$".

On note $w \Rightarrow_R w'$ ssi il existe u, v de $\mathcal{V}^*$ et $l \to r \in \mathcal{R}$ tels que $w = u\ l\ v$ et $w' = u\ r\ v$.

On note $w \Rightarrow_R^* w'$ (c-à-d. "w **se réécrit en** w'" ou "w' **dérive de** w") ssi il existe une suite $(u_i)_{i \in 0..n}$ telle que $u_0 = w$ et $u_n = w'$ et pour tout i de $1..n$, $u_{i-1} \Rightarrow_R u_i$.

Exemples

Exo 5.1 Soit R_1 le système de réécriture pour $\mathcal{V}_1 \stackrel{def}{=} \{b, c, C\}$ et $\mathcal{R}_1$ formé des 2 règles :

$$C\ b \quad \rightarrow \quad b\ C \tag{1}$$
$$C\ c \quad \rightarrow \quad c\ c \tag{2}$$

Calculer l'ensemble des mots dérivables depuis "$C\ b^5\ c$".

...

Exo 5.2 Soit R_2 le système de réécriture pour $\mathcal{V}_2 \stackrel{def}{=} \mathcal{V}_1 \uplus \{a\}$ et $\mathcal{R}_2$ formé des règles de $\mathcal{R}_1$ et de :

$$a\ b \quad \rightarrow \quad a\ a\ b\ b\ C \tag{3}$$

Calculer l'ensemble des mots dérivables depuis "$a\ b\ c$".

...

Réécriture comme calcul non-déterministe

Redex d'un mot $w \stackrel{def}{=}$ un triplet (u, l, v) tel que $w = u\ l\ v$ et l membre gauche d'une règle de $\mathcal{R}$.

Calcul de "$w_1 \Rightarrow_R^* w_2$" = calcul avec 2 choix *non-déterministes* à chaque étape de réécriture :

1. choix du redex.
2. choix de la règle à appliquer (un même l peut être membre gauche de plusieurs règles).

Semi-algorithme (c-à-d qui ne termine pas forcément) pour savoir si "$w_1 \Rightarrow_R^* w_2$" avec R, w_1 et w_2 fixés : *exploration en largeur* de l'ensemble des mots dérivables depuis w_1 en s'arrêtant dès qu'on rencontre w_2.

$\rightsquigarrow$ Technique standard pour effectuer un calcul non-déterministe par un (semi-)algo déterministe !
(Applicable si ensemble des choix possibles à chaque étape est fini).

Stratégies de réécriture

Stratégie $\stackrel{def}{=}$ calcul éventuellement non-déterministe de sélection "redex+règle" à chaque étape de réécriture.
En gros, stratégie = sous-ensemble de $\Rightarrow_R^*$.

Stratégie complète vis-à-vis d'un "problème" sur $\Rightarrow_R^*$ (R fixé) $\stackrel{def}{=}$ pour ce "problème" il est équivalent de remplacer $\Rightarrow_R^*$ par la dite stratégie.

Intérêt : réduire l'espace de l'exploration en largeur !

En conclusion : si stratégie complète et déterministe, on peut espérer avoir un algo efficace pour résoudre ce "problème" !

Exemples de stratégies

On appelle **dérivation-gauche** (resp. **dérivation-droite**) la stratégie consistant à sélectionner un redex le plus à *gauche* (resp. à droite).

Exo 5.3 Dans le cas de R_2, ensemble des mots dérivables depuis "*a b c*" par chacune d'elle ?

...

Exo 5.4 Pour chacune de ces 2 stratégies, donner condition suffisante sur R garantissant qu'elle soit **déterministe** (c-à-d avec au plus un choix à chaque étape de réécriture).
Est-ce le cas pour R_2 ?

...

Exemple d'une stratégie complète et déterministe

Problème Étant donné $w \in \{a, b, c\}^*$, on cherche à savoir si

$$a\ b\ c \Rightarrow_{R_2}^* w$$

Exo 5.5 Montrer que la dérivation-droite est une stratégie déterministe et complète pour ce problème.
Est-ce le cas pour la dérivation-gauche ?

Plan de la section 5.1

Présentation

- Théorie générale des *grammaires génératives* proposées N. Chomsky en 1956 en *linguistique* (étude du langage humain).

- Résultats fondamentaux pour l'informatique, établissant connexions entre classes de langages (*ensembles de mots*) et classes d'automates (*algorithmes*).

Définitions des grammaires génératives

Une grammaire G est un 4-uplet $(\mathcal{V}_T, \mathcal{V}_N, S, \mathcal{R})$ avec :

- $\mathcal{V}_T$ alphabet *fini* des **terminaux** (alphabet du langage qu'on veut *engendrer* à l'aide de G).
- $\mathcal{V}_N$ alphabet *fini* des **non-terminaux** (symboles "auxiliaires" utilisés dans étapes intermédiaires de réécriture).
- $S \in \mathcal{V}_N$ est **l'axiome** ("*start symbol*" en anglais).
- $\mathcal{R}$ un ensemble de règles tq $(\mathcal{V}, \mathcal{R})$ système de réécriture avec $\mathcal{V} \stackrel{def}{=} \mathcal{V}_T \uplus \mathcal{V}_N$. On appelle aussi G ce système de réécriture.

Le langage L_G **engendré** (ou *reconnu*) par G est

$$L_G \stackrel{def}{=} \{w \in \mathcal{V}_T^* \ / \ S \Rightarrow_G^* w\}$$

Exo 5.6 Étendre R_2 en une grammaire appelée G_3 tq $L_{G_3} = \{a^n \ b^n \ c^n \ / \ n \in \mathbb{N}\}$.

Grammaires hors-contextes (dites aussi "de type 2")

Une grammaire est dite **hors-contexte** ssi toutes les règles sont de la forme :

$$A \to \alpha \text{ avec } A \text{ dans } \mathcal{V}_N \text{ et } \alpha \in \mathcal{V}^*$$

Exo 5.7† Montrer que la GHC ci-dessous engendre $\{a^n b^n \mid n \in \mathbb{N}\}$.
On pose $\mathcal{V}_T \stackrel{def}{=} \{a, b\}$ et $\mathcal{V}_N \stackrel{def}{=} \{S\}$, avec les règles :

$$S \to a\ S\ b \qquad\qquad S \to \epsilon$$

Thm Un langage hors-contexte ssi il est algébrique.
Exo 5.8† Donner la BNF sur $\mathcal{V}_T$ associée à la GHC ci-dessus.

Grammaires de type 3

Une grammaire est dite de type 3 ssi elle est linéaire à gauche ou linéaire à droite.

Une grammaire est linéaire à droite (resp à gauche) ssi toutes les règles sont de la forme :

- $A \to w$
- ou, $A \to w\ B$ (resp. $A \to B\ w$)

avec A et B dans $\mathcal{V}_N$ et w dans $\mathcal{V}_T^*$.
NB : une grammaire de type 3 est donc aussi de type 2 !

Thm Un langage est engendré par une grammaire linéaire à droite (resp à gauche) ssi c'est un langage régulier.
Preuve : déjà fait au chapitre 2 sur les langages algébriques.

Autres types de grammaires

type 0 cas général (aucune restriction)

type 1 introduites par Chomsky, mais peu utilisées en pratique ? contient les GHCs.

LL GHCs pour lesquelles une certaine sous-stratégie de la dérivation-gauche est complète et déterministe (cf. détails plus tard)

LR GHCs pour lesquelles on a une stratégie déterministe et complète dans le système de réécriture *inverse*. Plus générales que grammaires LL, mais aussi plus complexes à analyser.

...

Classification de Chomsky (1956) / Knuth† (1965)

Langages	Grammaires génératives	Automates reconnaisseurs	Exemple caractéristique
quelconques	aucune	aucun	ens. des procédures Ada à 1 paramètre qui terminent sur une entrée A et qui bouclent sur une entrée B (A,B fixés).
récursivement énumérables	type 0	machines de Turing (semi-algorithme !)	ens. des pg. Ada sans entrée qui terminent
contextuels	type 1	automates linéairement bornés	$\{ a^n b^n c^n \mid n \in \mathbb{N} \}$
hors-contextes (ou algébriques)	type 2	automates à piles	ens. des palindromes $\{w \in \{a,b\}^* \mid w = \bar{w}\}$
hors-contextes déterministes	LR(k)†	automates à piles déterministes	$\{ a^n b^n \mid n \in \mathbb{N} \}$
réguliers	type 3	automates finis	$a.(b+c)^*$

horizontalement équivalence entre classes de langages/grammaires/automates

verticalement (de bas en haut) inclusion *stricte* cf. exemple caractéristique

Chapitre 5 Théorie des grammaires et du parsing

Plan de la section 5.2

Introduction

"LL(1)" pour "Left-to-right reading, Leftmost derivation, 1 lookahead".
$\rightsquigarrow$ Analyse syntaxique en dérivation-gauche limitée aux GHC où on a *stratégie complète et déterministe* en sélectionnant la *bonne règle* uniquement à partir du *premier* caractère du w en entrée.
NB De telles GHC sont dites LL(1).

Intérêts

- grammaires LL(1) non-ambiguës par construction.
- décidabilité de la vérif qu'une GHC est LL(1).
- la plupart des constructions usuelles des langages informatiques peuvent s'exprimer via grammaires LL(1).
- parsing efficace (sans backtracking) et facile à implémenter
  $\Rightarrow$ généralise automate fini déterministe + parsing en notation préfixe

Difficulté

Indécidable de trouver une forme LL(1) pour une GHC donnée.

Intuition de l'analyse LL(1) sur des exemples (1/2)

Exo On considère la BNF d'équation $S ::= b \mid a\ S\ a\ S$.

1) Compléter la procédure de parsing récursif `parse_S` ci-dessous :

```
typedef enum { a, b, END } token;
token next(); // analyseur lexical

void parse_S() { ... }
void parse() { // procédure principale du parsing
  parse_S(); if (next()!=END) { printf("ERROR");exit(1); }
}
```

2) Adapter ensuite votre code pour la version alternative

```
token current; // var. globale de pré-vision (look-ahead)

void parse_token(token expected) { // consommer "current"
  if (current != expected) { printf("ERROR");exit(1); }
  if (current != END) {current = next();}
}
void parse_S() { ... }
void parse() { // procédure principale du parsing
  current = next(); // init. "current" (sur 1er token)
  parse_S(); parse_token(END);
}
```

Intuition de l'analyse LL(1) sur des exemples (2/2)

Exo Adapter la démarche précédente pour cette BNF équivalente à la précédente

$$S ::= b \mid X\ X \qquad X ::= a\ S$$

avec deux procédures mutuellement récursives :

```
void parse_S() { ... }
void parse_X() { ... }
```

Exo Essayer d'adapter la démarche précédente pour chacune des BNF du langage $\{\ a^n b^n a \mid n \in \mathbb{N}\ \}$:

1. $S ::= a\ X$
   $X ::= A\ b\ a \mid \epsilon$
   $A ::= a\ A\ b \mid \epsilon$
2. $S ::= A\ a$
   $A ::= a\ A\ b \mid \epsilon$

NB : on pourra simuler l'algorithme dans la reconnaissance de "a" puis "$a^3\ b^3\ a$".

Réduction de GHC

Avant de considèrer si G est LL(1), il faut d'abord la *réduire* :

- supprimer de G les non-terminaux A *improductifs*, c-à-d tq il n'existe pas $w \in \mathcal{V}_T^*$ tq $A \Rightarrow^* w$.
- supprimer de G les non-terminaux A *inaccessibles*, c-à-d tq il n'existe pas α_1 et α_2 de $\mathcal{V}^*$ tq $S \Rightarrow^+ \alpha_1\ A\ \alpha_2$.

NB : supprimer A de $G \rightsquigarrow$ supprimer ttes les règles où A figure.

Exo 5.9 Montrer que réduire G ne change pas L_G.

Thm les propriétés "*être improductif*" et "*être inaccessible*" sur une GHC qcq sont décidables.
(Par calculs de point-fixe similaires à ceux du chapitre 2).

Corollaire il y a un algo pour réduire une GHC qcq.

Définition des grammaires LL(1) (Lewis/Stearns 1968)

Soit G une GHC **réduite**. Soit $\$ \notin \mathcal{V}$ (sentinelle de fin d'entrée).
On pose $\mathcal{V}_{T \uplus \$} \stackrel{def}{=} \mathcal{V}_T \uplus \{\$\}$.

Définition du directeur $\mathrm{Dir}(\pi)$ d'une règle $\pi = A \to \alpha$.
$\mathrm{Dir}(\pi)$ est défini comme l'ensemble des $a \in \mathcal{V}_{T \uplus \$}$ pour lesquels il existe $w_1 \in \mathcal{V}_T^*$, w_2 et w_3 de $\mathcal{V}_{T \uplus \$}^*$ tq

$$S\ \$ \Rightarrow^* w_1\ A\ w_2 \quad \text{et} \quad \alpha\ w_2 \Rightarrow^* a\ w_3$$

NB 1 : dans def ci-dessus, on peut avoir $\alpha \Rightarrow^* \epsilon$.
NB 2 : G réduite implique $\mathrm{Dir}(\pi) \neq \emptyset$

Intuition dans parsing récursif, une règle π de forme $A \to \alpha$ ne *peut* dériver "*a w*" que si $a \in \mathrm{Dir}(\pi)$.

Déf G est dite LL(1) ssi pour ttes régles $A \to \alpha$ et $A \to \beta$,

$$\alpha \neq \beta \text{ implique } \mathrm{Dir}(A \to \alpha) \cap \mathrm{Dir}(A \to \beta) = \emptyset$$

Exo 5.10 Montrer que G LL(1) implique G non ambiguë.

Calcul du directeur des règles

On décompose le calcul en :

$$\mathrm{Dir}(A \to \alpha) \quad = \quad \mathrm{Prem}(\alpha) \ \cup \ \mathcal{E}(\alpha).\mathrm{Suiv}(A)$$

où

1. "$\mathcal{E}(\alpha).X$" est une notation pour "si $\alpha \Rightarrow^* \epsilon$ alors X sinon $\emptyset$"
2. $\mathrm{Prem}(\alpha) = \{a \in \mathcal{V}_T \mid$ il existe $w \in \mathcal{V}_T^*$ tq $\alpha \Rightarrow^* a\ w\}$.
3. $\mathrm{Suiv}(A)$ est l'ensemble des $a \in \mathcal{V}_{T \uplus \$}$ tels qu'il existe $w_1 \in \mathcal{V}_T^*$ et $w_2 \in \mathcal{V}_{T \uplus \$}^*$ avec $S\ \$ \Rightarrow^* w_1\ A\ a\ w_2$.

Plan de la suite : calcul de Prem et Suiv par **commutation** de +petit point-fixe.

Calcul de Prem

Même méthode que $\mathcal{E}$ en ramenant le calcul à
$\mathrm{Prem}(X) = \{a \in \mathcal{V}_T \mid \exists w \in \mathcal{V}_T^*,\ a\,w \in X\}$.

Système d'équations pour lemme de commutation
Transformation de chaque équation "$X_k ::= e_k$" de la BNF en équation "$\mathrm{Prem}(X_k) = \mathrm{Prem}(e_k)$" où "$\mathrm{Prem}(e_k)$" calculé par récursion structurelle sur syntaxe de e_k :

- $\mathrm{Prem}(\epsilon) = \emptyset$
- Pour tout $a \in \mathcal{V}_T$, $\mathrm{Prem}(a) = \{a\}$
- $\mathrm{Prem}(\alpha.\beta) = \mathrm{Prem}(\alpha) \cup \mathcal{E}(\alpha).\mathrm{Prem}(\beta)$
- $\mathrm{Prem}(\alpha \mid \beta) = \mathrm{Prem}(\alpha) \cup \mathrm{Prem}(\beta)$

Calcul de +ptit point-fixe sur $E = [1, |\mathcal{V}_N|] \times \mathcal{V}_T$.

Calcul de Suiv

Système d'équations variables $(\mathrm{Suiv}(X))_{X \in \mathcal{V}_N}$ d'équation

$$\mathrm{Suiv}(X) = \quad \text{si } X \text{ axiome alors } \{\$\} \text{ sinon } \emptyset \quad \cup \quad \bigcup_{Y \to \alpha.X.\beta \in \mathcal{R}} \mathrm{Prem}(\beta) \cup \mathcal{E}(\beta).\mathrm{Suiv}(Y)$$

NB dans énumération ci-dessus des règles de forme "$Y \to \alpha.X.\beta$" une même règle de $\mathcal{R}$ est utilisée autant de fois que X apparaît dans le membre droit. Par ex, la règle $B \to a.A.b.A$ compte avec "$\alpha = a, \beta = b.A$" puis avec "$\alpha = a.A.b, \beta = \epsilon$".

Calcul du +petit pt fixe sur $E = [1, |\mathcal{V}_N|] \times \mathcal{V}_{T \uplus \$}$.

Mini-Exemple

Exo 5.11† Pour chacune des 4 grammaires ci-dessous du langage $\{a^n b^m \mid 0 \leq n \leq m\}$:

1. $S \to a\ S\ B \mid \epsilon \mid B$
   $B \to b\ B \mid b$
2. $S \to a\ S\ b\ B \mid B$
   $B \to b\ B \mid \epsilon$
3. $S \to a\ S\ b \mid B$
   $B \to b\ B \mid \epsilon$
4. $S \to A\ B$
   $A \to a\ A\ b \mid \epsilon$
   $B \to b\ B \mid \epsilon$

- Calculer le directeur de chacune des règles.
- Utiliser le calcul de directeur pour calculer l'ensemble des dérivations gauche de $a^2 b^5$

Plan de la section 5.2

Cadre de la présentation

Cadre simplifié :

- arrêt du programme à la première erreur ;
- token sans attribut (cf. généralisation en TP).

Analyseur lexical

```
typedef enum { ... } token;
token next();
```

Interface avec l'analyseur lexical

```
/* variable globale de look-ahead */
token current;

/* vérifie "current==expected" puis fait "current=next();" */
void parse_token(token expected);
```

Principe de l'analyseur récursif

Analyseur donné par procédures *mutuellement récursive*.
Intuition : arbre d'analyse reconnu = arbre des appels récursifs.

Ainsi, pour tt non-terminal de profil $X{\downarrow}H_1 \ldots {\downarrow}H_n{\uparrow}S_1 \ldots {\uparrow}S_m$,

```
void parse_X(H1 h1, ..., Hn hn, S1 *s1, ..., Sm *sm);
```

reconnaît le *+long préfixe* de l'entrée qui correspond à X et affecte attributs comme spécifié dans équation de $X{\downarrow}h_1 \ldots {\downarrow}h_n{\uparrow}s_1 \ldots {\uparrow}s_m$.

Attention :

- `parse_X` lit le 1er token dans `current`.
  En sortie, `current` sur le 1er token qui suit le préfixe reconnu.
- Échec de `parse_X` $\Rightarrow$ soit pas de préfixe de X dans l'entrée, soit +long préfixe pas suivi d'un token de $\mathrm{Suiv}(X)$.

Codage "automatique" des équations sur un exemple

Équation annotée avec directeurs

$$\{a_1, \ldots, a_n\} \quad X{\downarrow}h{\uparrow}s \rightarrow \quad A{\uparrow}s\ c$$
$$\{b_1, \ldots, b_m\} \qquad\qquad | \quad C{\downarrow}h{\uparrow}s_0\ e\ D{\downarrow}f(h, s_0){\uparrow}s_1 \quad s := g(s_0, s_1)$$

traduite en

```
void parse_X(H h, S *s){
        S0 s0; S1 s1;
        switch (current) {
        case a1: ... : case an:
                parse_A(s);
                parse_token(c); break;
        case b1: ... : case bm:
                parse_C(h, &s0);
                parse_token(e);
                parse_D(f(h, s0), &s1);
                *s = g(s0, s1); break;
        default: unexpected_token("X");
        }
}
```

Incompatible avec dép droite/gauche ds éval des attributs !

Restriction correspondant aux "*grammaires L-attribuées*"

Exemple d'analyseur LL(1) dans `example_LL1.c`

Spécification de l'exemple

Accepte mots d'entrée $a^n b^n$ (type `A`) ou $b^n c$ (type `B`) et retourne le type de mot reconnu (`A` ou `B`) ainsi que son nombre de "b".

Non-terminaux $S\uparrow\{\texttt{A},\texttt{B}\}\uparrow\mathbb{N}$ $A\uparrow\mathbb{N}$ $B\uparrow\mathbb{N}$

Grammaire LL(1) attribuée & annotée par directeurs

$$
\begin{array}{lrll}
\{a,\$\} & S\uparrow t\uparrow n \rightarrow & A\uparrow n & t := \texttt{A} \\
\{b,c\} & \mid & B\uparrow n\ c & t := \texttt{B} \\
 & & & \\
\{a\} & A\uparrow n \rightarrow & a\ A\uparrow n_0\ b & n := n_0 + 1 \\
\{b,\$\} & \mid & \epsilon & n := 0 \\
 & & & \\
\{b\} & B\uparrow n \rightarrow & b\ B\uparrow n_0 & n := n_0 + 1 \\
\{c\} & \mid & \epsilon & n := 0
\end{array}
$$

Exo 5.12 arbre des appels récursifs de $a^2 b^2$?

Message d'erreur en analyse LL(1)

Sur cette portion de la BNF

$$\begin{array}{lll} \{a\} & A \rightarrow & a\ A\ b \\ \{b, \$\} & \mid & \epsilon \end{array}$$

3 choix d'implémen. qui changent juste messages d'erreur

```
void parse_A(){
  switch(current){
  case b: case eof:
    break;
  default:
    parse_token(a);
    parse_A();
    parse_token(b);
    break;
  }
}
```

```
void parse_A(){
  switch(current){
  case a:
    parse_token(a);
    parse_A();
    parse_token(b);
    break;
  default:
    break;
  }
}
```

```
void parse_A(){
  switch(current){
  case a:
    parse_token(a);
    parse_A();
    parse_token(b);
    break;
  case b: case eof:
    break;
  default:
    unxpct("a b $");
  }
}
```

Exemple d'erreur

aac
↑ attend a

aac
↑ attend b

aac
↑ attend a b $

Plan de la section 5.2

Tout langage régulier L est LL(1)

BNF LL(1) de L = système d'équations (linéaires à droite) de l'automate déterministe minimal de L moins l'éventuel état puit.

Exo 5.13 Faire la preuve que la BNF est bien LL(1).

...

Les calculs de directeurs LL(1) s'étendent aux EBNF

Définition EBNF = BNF avec expressions régulières en membre droits d'équations
$\Rightarrow$ **expressif & efficace** pour engendrer analyseurs LL
(cf. méta-interpréteur ANTLR)

Exemple 1 $X ::= \alpha_1.(\alpha_2)^*.\alpha_3$
se réécrit (pour le calcul de directeur) en

$$X ::= \alpha_1.X'.\alpha_3 \qquad X' ::= \alpha_2.X' \mid \epsilon$$

Exemple 2 $X ::= \alpha_1.(\alpha_2 \mid \alpha_3).\alpha_4$
se réécrit (pour le calcul de directeur) en

$$X ::= \alpha_1.X'.\alpha_4 \qquad X' ::= \alpha_2 \mid \alpha_3$$

EBNF LL(1) L-attribuée pour associativité à droite

Équation non LL(1) suivante (codant '**' associatif à droite)

$$\begin{array}{rcll} exp_1{\uparrow}r & ::= & exp_0{\uparrow}r_1 \text{ '**' } exp_1{\uparrow}r_2 & r := r_1^{r_2} \\ & | & exp_0{\uparrow}r & \end{array}$$

après factorisation de exp_0, équivaut à :

$$exp_1{\uparrow}r ::= exp_0{\uparrow}r_1 \ (\epsilon\{r := r_1\} \mid \text{'**'} \ exp_1{\uparrow}r_2\{r := r_1^{r_2}\})$$

Exo 5.14† Montrer que l'équation est LL(1) ssi '**' $\notin \mathrm{Suiv}(exp_1)$

Si LL(1), équation EBNF qui s'implémente en :

```
void parse_exp1(int *r){
        int r1, r2;
        parse_exp0(&r1);
        if (current=='**') {
                parse_token('**');
                parse_exp1(&r2);
                *r=pow(r1,r2);
        } else { *r=r1; }
}
```

EBNF LL(1) L-attribuée pour associativité à gauche

Équation non LL(1) suivante (codant '−' associatif à gauche)

$$\begin{array}{rcll} exp_1{\uparrow}r & ::= & exp_1{\uparrow}r_1\ \text{`}-\text{'}\ exp_0{\uparrow}r_2 & r := r_1 - r_2 \\ & | & exp_0{\uparrow}r & \end{array}$$

par "symétrique" du lemme d'Arden, équivaut à :

$$exp_1{\uparrow}r ::= exp_0{\uparrow}r_1\{r := r_1\}(\text{`}-\text{'}\ exp_0{\uparrow}r_2\{r := r - r_2\})^*$$

Exo 5.15† Montrer que l'équation est LL(1) ssi '−' $\notin \mathrm{Suiv}(exp_1)$

Si LL(1), équation EBNF qui s'implémente en :

```
void parse_exp1(int *r){
        int r1, r2;
        parse_exp0(&r1);
        *r=r1;
  Loop: if (current=='-') {
                parse_token('-');
                parse_exp0(&r2);
                *r -= r2;
                goto Loop;
        }
}
```

Exemples fondamentaux

Exo 5.16† On considère le langage **E** du chapitre 4 (diapo 4) pour la sémantique non-ambiguë donnée par les règles précédences usuelles :

1. Écrire une EBNF attribuée LL(1) équivalente.
2. En déduire le pseudo-code C de l'analyseur LL(1).
3. Donner aussi la BNF attribuée LL(1) équivalente à cette EBNF.

Exo 5.17† Écrire une EBNF attribuée LL(1) qui reconnaît le langage des expressions arithmétiques de l'exemple Bison-Yacc du chapitre 4, avec la même sémantique.

Plan de la section 5.2

Au-delà de l'analyse LL(1)

Intérêt des grammaires LL(1) algo d'analyse syntaxique simple et efficace.

Généralisable en LL(n) avec n symboles de pré-vision, voire en LL(*) avec buffer de pré-vision = langage régulier
cf. méta-compilateur AntLR3 pour grammaires LL(*).

Grammaires LR souvent utilisées pour langages de programmation classiques. Algo d'analyse syntaxique plus complexe. Nécessite méta-compilateur (e.g. Yacc).

Mini état de l'art sur les méta-compilateurs

• Analyse LR(1) (ou sa variante LALR(1)) est prédominante depuis longtemps (avec Yacc). LR(k) pour $k > 1$ est actuellement considéré comme impraticable.
NB : toute grammaire LL(k) est LR(k).

• Analyse LL(*) apparue en 2005 avec AntLR (AntLR v3).

• Il existe des *langages* qui ne peuvent pas être reconnus en LR(1) mais qui peuvent l'être en LL(*) et réciproquement ! Par ailleurs, le langage des palindromes ne peut être reconnu ni en LL(*), ni en LR(k).

Exemple de langage LR(1) qui n'est pas LL(*)

Le langage $\{a^m b^n \mid m \geq n\}$ est reconnu par une grammaire LR(1) :

$$\begin{array}{rcl} S & ::= & a\ S \mid X \\ X & ::= & a\ X\ b \mid \epsilon \end{array}$$

Mais il n'existe aucune grammaire LL(*) qui reconnaît ce langage.
NB : résultat non trivial, car par contre, pour tout p, il existe une grammaire LL(1) qui reconnaît $\{a^m b^n \mid n + p \geq m \geq n\}$.

Exemple de langage LL(2) qui n'est pas LR(1)

Le langage $\{a^n(ab)^n \mid n \in \mathbb{N}\}$ est reconnu par une grammaire LL(2) :

$$S \ ::= \ a \ S \ a \ b \mid \epsilon$$

mais il n'existe aucune grammaire LR(1) (et donc aucune LL(1)) qui reconnaît ce langage.

NB : grammaire ci-dessus LL(2), donc aussi LR(2) et LL(*).