PF chapitre 6 : analyse syntaxique

Jean-François Monin

POLYTECH GRENOBLE

Plan

Grammaires d'expressions (cf Cours de LT)

Principe de programmation d'une analyse syntaxique

Analyse en deux temps

Définitions

Soit un ensemble $\mathcal{T}$

- mot = séquence (éventuellement vide) d'éléments de T
- $\mathcal{T}^*$ = ensemble des *mots* sur $\mathcal{T}$

Une représentation possible

$\mathcal{T}^* = \mathcal{T}$ `list`

La concaténation sur $\mathcal{T}^*$ est :

- notée par juxtaposition
- associative $(uv)w = u(vw)$

Définitions

- *Langage* sur $\mathcal{T}$ = sous-ensemble de $\mathcal{T}^*$.
- *Lexème* = unité lexicale ou mot.
- Analyse lexicale : découpage d'une suite de $\mathcal{T}$ en mots.
- Analyse syntaxique : organisation des lexèmes en phrase.

Définitions

- *Langage* sur $\mathcal{T}$ = sous-ensemble de $\mathcal{T}^*$.
- *Lexème* = unité lexicale ou mot.
- Analyse lexicale : découpage d'une suite de $\mathcal{T}$ en mots.
- Analyse syntaxique : organisation des lexèmes en phrase.

Exemples :

- Langue naturelle $\mathcal{T}_l = \{'a', 'b', ...'z', 'A' ...'Z', ' '\}$
  - Les lexèmes légaux sont : les mots du dictionnaire.
  - L'analyse syntaxique vérifie qu'il y a un sujet, un verbe...

Définitions

- *Langage* sur $\mathcal{T}$ = sous-ensemble de $\mathcal{T}^*$.
- *Lexème* = unité lexicale ou mot.
- Analyse lexicale : découpage d'une suite de $\mathcal{T}$ en mots.
- Analyse syntaxique : organisation des lexèmes en phrase.

Exemples :

- Langue naturelle $\mathcal{T}_l = \{'a', 'b', ...'z', 'A' ...'Z', ' '\}$
  - Les lexèmes légaux sont : les mots du dictionnaire.
  - L'analyse syntaxique vérifie qu'il y a un sujet, un verbe...
- Expressions algébriques $\mathcal{T}_s = \{+, *, (,), \text{Ent } (n), \text{Id } (i)\}$
  - Les lexèmes sont directement les symboles de $\mathcal{T}_s$: pas d'analyse lexicale
  - Analyse syntaxique : bon parenthésage, opérateurs infixes...

Vocabulaires

- Vocabulaire *terminal* : $\mathcal{T} = \{$a, b, c, ...$\}$
- Mots sur $\mathcal{T}$ notés u, v, w, ...
- Vocabulaire *non-terminal* : $\mathcal{N} = \{$A, B, C ...$\}$,
  chaque élément de $\mathcal{N}$ désigne un langage sur $\mathcal{T}$
- *Extension* à $\mathcal{N}^*$ et à $(\mathcal{T} \cup \mathcal{N})^*$ de la concaténation sur $\mathcal{T}^*$:
  $UV = \{uv \mid u \in U \wedge v \in V\}$

Exemples :

- $U =$ ensemble des mots formés d'exactement un chiffre
- $A =$ ensemble des mots formés d'au moins un chiffre
- $C =$ ensemble des mots formés de 0 ou plusieurs chiffres

Langages

Soit $\mathcal{T} = \{1, 2, \ldots, 9, +\}$ et $\mathcal{N} = \{E\}$

Règles

- E ::= E + E
- E ::= *n*

Grammaire = ensemble exhaustif de règles décrivant les expressions acceptées

Problème de la reconnaissance

Étant donné un mot u, et un non terminal S définissant un langage $\mathcal{S}$, déterminer si $u \in \mathcal{S}$.

Problème de la reconnaissance

Étant donné un mot u, et un non terminal S définissant un langage $\mathcal{S}$, déterminer si $u \in \mathcal{S}$.

Au passage : calculer une forme structurée du mot reconnu.

Exemple (criticable)

Soit la grammaire suivante

- E ::= T + T
- E ::= T
- T ::= n
- T ::= (E)

Remarque : cette grammaire est limitée

Exemple (criticable)

Soit la grammaire suivante

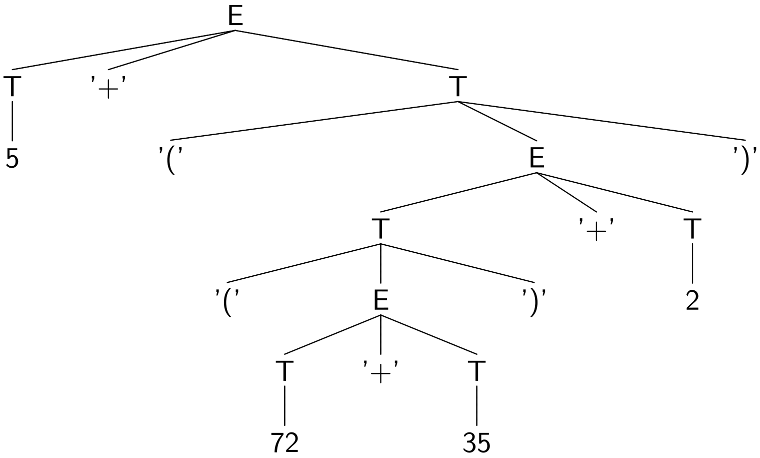
- E ::= T + T
- E ::= T
- T ::= *n*
- T ::= (E)

Remarque : cette grammaire est limitée

$\boxed{3 + 5 + 1 \text{ ne fait pas partie du langage défini par E}}$

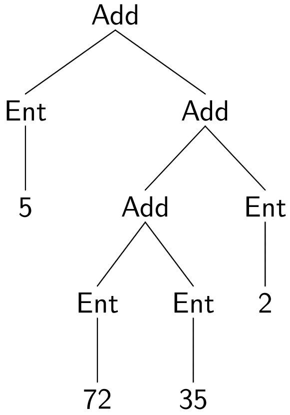
Arbre syntaxique, arbre concret

$5 + ((72 + 35) + 2)$



Arbre abstrait

$5 + ((72 + 35) + 2)$



Plan

Grammaires d'expressions (cf Cours de LT)

Principe de programmation d'une analyse syntaxique

Analyse en deux temps

Décomposition naïve systématique par essais-erreurs

Pour une règle $S ::= S_1 S_2 \dots S_n$

- Décomposer x de toutes les manières possibles sous la forme de n concaténations

$$x = x_1 x_2 \dots x_n$$

  - pour chaque i, déterminer si $x_i \in S_i$
  - si échec, essayer une autre décomposition
- Si pas de solution, essayer une autre règle pour S

Décomposition naïve systématique par essais-erreurs

Pour une règle $S ::= S_1 S_2 \ldots S_n$

- Décomposer x de toutes les manières possibles sous la forme de n concaténations

$$x = x_1 x_2 \ldots x_n$$

  - pour chaque i, déterminer si $x_i \in S_i$
  - si échec, essayer une autre décomposition
- Si pas de solution, essayer une autre règle pour S

C'est (très) inefficace

Analyse descendante récursive : aspirateurs

À chaque mot u correspond un aspirateur p_u

= fonction (partielle) qui,
étant donné un mot x, rend x privé de son préfixe u
(« consomme ou aspire u en préfixe de x ») :

$$p_u(x) = y \textit{ ssi } x = uy$$

À chaque non-terminal N correspond un aspirateur p_N

= fonction (partielle) qui
aspire un élément de N en préfixe d'un mot x :

$$p_N(x) = y \text{ ssi } \exists u \in N,\ x = uy$$

Exemple : pour une règle $U ::= \text{a}\ V\ \text{b}\ T\ W$ on a :

$$p_U(x) = p_W\left(p_T\left(p_{\text{b}}\left(p_V\left(p_{\text{a}}\left(x\right)\right)\right)\right)\right)$$

Analyse descendante récursive sur des listes

On cherche à reconnaître un langage très simple :

les mots de la forme $((\ldots(a)\ldots))$ bien parenthésés

Analyse descendante récursive sur des listes

On cherche à reconnaître un langage très simple :

les mots de la forme $((\ldots(a)\ldots))$ bien parenthésés

La grammaire suivante convient :

- S ::= (S)
- S ::= a

Analyse descendante récursive sur des listes

On cherche à reconnaître un langage très simple :

les mots de la forme $((\dots(a)\dots))$ bien parenthésés

La grammaire suivante convient :

- S ::= (S)
- S ::= a

On représente les mots comme des listes de caractères et on programme les analyseurs p_u pour tous les terminaux et non-terminaux.

Précautions

JAMAIS de récursion gauche

- Interdire les règles de la forme $U ::= Ux$
  Sinon l'algorithme sous-jacent boucle :
  « Pour (tenter d') aspirer un U en préfixe,
  on va commencer par (essayer d') aspirer un U en préfixe, puis etc. »
- Même sous forme cachée, indirecte
  $U ::= VabA$
  $V ::= Ucde$

Ne pas commencer par ϵ en cas de choix

- Exemple, un U optionnel : $O ::= U \mid \epsilon$
- $O ::= U$
  $O ::= \epsilon$

Choix de la règle

Plusieurs règles pour un non terminal

Cas simple : chaque règle commence par un terminal

Sélection de la règle selon le terminal débutant le membre droit

Cas plus général

- solution 1 : transformer la grammaire
  pour se ramener au cas simple
- solution 2 : mécanisme d'exception
  `try` *membre droit règle 1*
  `with` Echec → *membre droit règle suivante*
- solution 3 (OCaml) : parser (*en voie d'abandon ?*)

Autres structures pour les séquences

- listes paresseuses
- stream

Plan

Grammaires d'expressions (cf Cours de LT)

Principe de programmation d'une analyse syntaxique

Analyse en deux temps

Analyses successives : lexicale puis syntaxique

Analyse lexicale

Regroupe les caractères consécutifs en lexèmes (en anglais : token)

```
type token =
    | Tident of char list
    | Tent of int
    | Tspeciaux of char list
    | Tparouv
    | ...
```

Analyses successives : lexicale puis syntaxique

Analyse lexicale

Regroupe les caractères consécutifs en lexèmes (en anglais : token)

```
type token =
     | Tident of char list
     | Tent of int
     | Tspeciaux of char list
     | Tparouv
     | ...
```

Analyse syntaxique

Reconnaît la structure (en arbre) de la séquence des lexèmes

Articulation

séquence de caractères $\rightarrow$ séquence de lexèmes
séquence de lexèmes $\rightarrow$ AST

Fichiers ml

Démos

- semaine 1 : `analist_exemples.ml`
- semaine 2 : `anacomb_decouverte.ml`

Librairie TD-TP, projet

- semaine 1 : `analist.ml`
- semaine 2 : `anacomb.ml`

Facultatif

- `anacomb_decouverte_star.ml`