Programmation Fonctionnelle (PF)

INFO4

Cours 5 : analyse syntaxique

Jean-François Monin, Benjamin Wack

POLYTECH GRENOBLE

2019 - 2020

Plan

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 (ab)c= a(bc)

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

Vocabulaires

- Vocabulaire *terminal* : $\mathcal{T} = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \ldots\}$
- Mots sur $\mathcal{T}$ notés u, v, w, ...
- Vocabulaire *non-terminal* : $\mathcal{N} = \{A, B, C \ldots\}$,
  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 :

- E = ensemble des expressions
- T = ensemble des termes

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

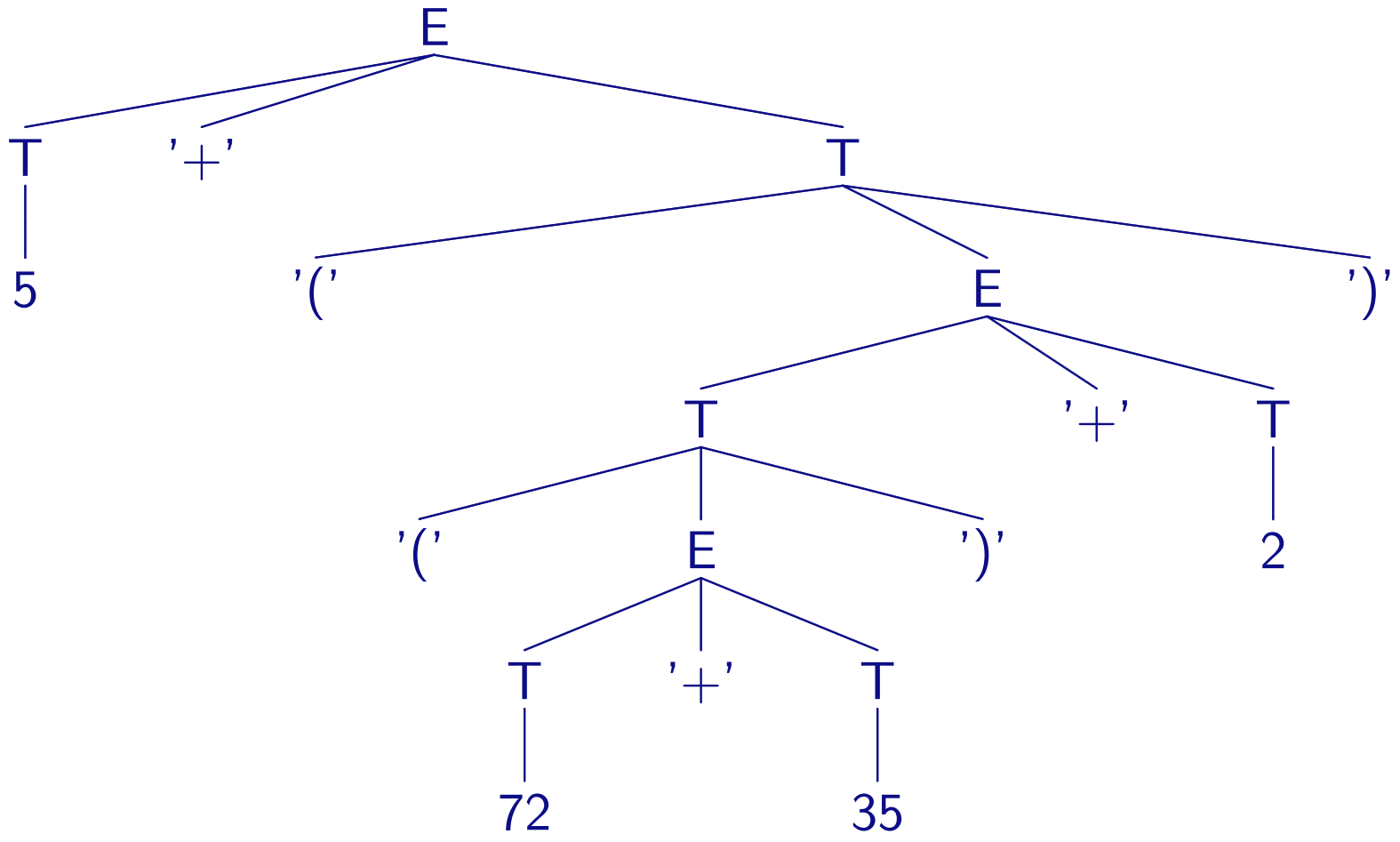
- E : := T + T
- E : := T
- T : := *n*
- T : := (E)

Remarque : cette grammaire est limitée

3 + 5 + 1 ne fait pas partie du langage défini par E

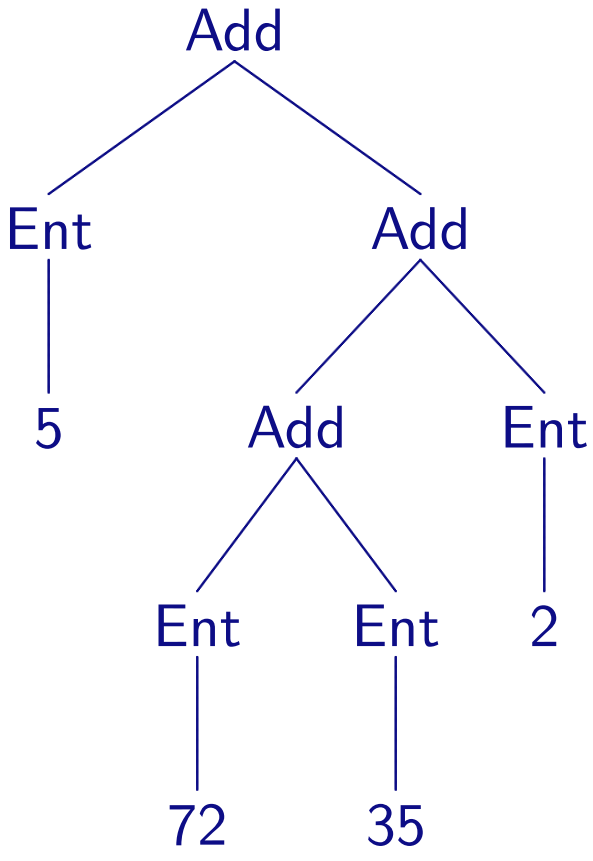
Arbre syntaxique

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



Arbre abstrait

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



Plan

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

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

À chaque mot u

on fait correspondre la fonction (partielle) p_u qui consomme u en préfixe d'un mot x :

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

À chaque non-terminal N

on fait correspondre une fonction (partielle) p_N qui consomme 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 $U ::= \mathrm{a}\ V\ \mathrm{b}\ T\ W$ on a :

$$p_U(x) = p_W(p_T(p_\mathrm{b}(p_V(p_\mathrm{a}(x)))))$$

Analyse descendante récursive sur des listes

On cherche à reconnaître un langage très simple :

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

Analyse descendante récursive sur des listes

On cherche à reconnaître un langage très simple :

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

La grammaire suivante convient :

- S : := (S)
- S : := x

Analyse descendante récursive sur des listes

On cherche à reconnaître un langage très simple :

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

La grammaire suivante convient :

- S : := (S)
- S : := x

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

Choix de la règle

Plusieurs règles pour un non terminal

Précautions

- pas de récursion gauche, même cachée (interdire $U := Ua$)
- ne pas commencer par ϵ

Cas simple : chaque règle commence par un terminal
on filtre sur le premier élément débutant le mot analysé
Cas plus général

- solution 1 : transformer la grammaire
  (cf cours Langages et Traducteurs)
- solution 2 :
  `try` *membre droit règle 1*
  `with` Echec $\rightarrow$ *membres droits règles suivantes*
- solution 3 (Ocaml) : flots (streams)

Plan

Flots (Stream)

Structure séquentielle polymorphe paresseuse

- séquentielle polymorphe : comme les listes
- *paresseuse* : construite à mesure des demandes de consommation (activation d'un filtrage)
- consommée (détruite) à mesure qu'elle est analysée

Intérêts

- éviter la construction d'une grosse structure de données intermédiaire
  - typiquement : fichier vu comme flot de caractères...
  - transformé en flot de lexèmes
- facilités d'expression : `parser`
- $\Rightarrow$ utilisation idéale dans les analyseurs récursifs descendants

Exemple : Pascal -> Caml, compilation, etc ...

Consommation

Idée : consommer peu de mémoire en ne travaillant que sur une fenêtre glissante de la donnée à analyser.

Avantages et inconvénients

- Bonne discipline de programmation forcée par l'utilisation des flots
- Impossibilité de revenir en arrière

Comment faire ?

- Utiliser des arguments supplémentaires pour conserver les données issues du passé
- Conception de la grammaire : partager les préfixes communs

Flots : construction

Extension syntaxique requise :

```
#use "topfind";;
#camlp4o;;
```

OU

```
#load "dynlink.cma";;
#load "camlp4o.cma";;
```

Constructeurs

- Stream.of_list, Stream.of_string, Stream.of_channel
- $[< \ldots >]$ avec éléments précédés de '

Ne pas mélanger ces 2 constructeurs

```
# let flux_int_19 =[<'1; '9>];;
val flux_int_19 : int Stream.t = <abstr>
# let flux_char_ab =[<''a'; ''b'>];;
val flux_char_ab : char Stream.t = <abstr>
# let flux_char_ab_cd = [< flux_char_ab; flux_char_cd>];;
# let flux_char_1934 = Stream.of_string "1934";;
```

Flots : production paresseuse

```
# let rec nat_stream = fun n -> [< 'n ; nat_stream (n+1) >]
val nat_stream : int -> int Stream.t = <fun>

# let nat = nat_stream 0 ;;
val nat : int Stream.t = <abstr>
```

Interlude : `function`

Syntaxe abrégée de fonctions

`function` est un raccourci syntaxique pour

`fun` $x \rightarrow$ `match` x `with`

Interlude : `function`

Syntaxe abrégée de fonctions

`function` est un raccourci syntaxique pour
`fun x` $\rightarrow$ `match x with`

Exemples

```
let rec longueur = function
  | [] -> 0
  | t :: q -> 1 + longueur q
let rec longacc a = function
  | [] -> a
  | t :: q -> longacc (1+a)
```

Interlude : `function`

Syntaxe abrégée de fonctions

`function` est un raccourci syntaxique pour
`fun` x $\longrightarrow$ `match` *x* `with`

Exemples

```
let rec longueur = function
  | [] -> 0
  | t :: q -> 1 + longueur q
let rec longacc a = function
  | [] -> a
  | t :: q -> longacc (1+a)
```

ATTENTION AU PARAMÈTRE IMPLICITE

Utilisation (analyse) d'un flot

Syntaxe semblable à `function` : paramètre de flot implicite

```
let analyse = parser
  | [< 'elt1 ; 'elt2 >] -> expr1
  | [< 'elt3 >] -> expr2
  | [< 'elt4; 'elt5; 'elt6 >] -> expr3
```

`analyse` est une fonction qui

- prend un flot en argument
- reconnaît un flot qui
  - soit commence par `elt1` puis `elt2` (et rend *expr*1)
  - soit commence par `elt3` (et rend *expr*2)
  - etc.

Les éléments peuvent être des constantes, des variables,
plus generalement des motifs
Exemple : '0'..'9' as c

Sémantique de `parser`

```
let analyse = parser
  | [< 'elt1 ; 'elt2 >] -> expr1
  | [< 'elt3 >] -> expr2
  | [< 'elt4; 'elt5; 'elt6 >] -> expr3
```

Sémantique de `parser`

```
let analyse = parser
  | [< 'elt1 ; 'elt2 >] -> expr1
  | [< 'elt3 >] -> expr2
  | [< 'elt4; 'elt5; 'elt6 >] -> expr3
```

Si le flot en argument

- commence par `elt1`,
  alors il doit se poursuivre par `elt2`
  la valeur rendue sera *expr*1

Sémantique de `parser`

```
let analyse = parser
  | [< 'elt1 ; 'elt2 >] -> expr1
  | [< 'elt3 >] -> expr2
  | [< 'elt4; 'elt5; 'elt6 >] -> expr3
```

Si le flot en argument

- commence par `elt1`,
  alors il doit se poursuivre par `elt2`
  la valeur rendue sera *expr*1
- commence par `elt3`
  la valeur rendue sera *expr*2

Sémantique de `parser`

```
let analyse = parser
  | [< 'elt1 ; 'elt2 >] -> expr1
  | [< 'elt3 >] -> expr2
  | [< 'elt4; 'elt5; 'elt6 >] -> expr3
```

Si le flot en argument

- commence par `elt1`,
  alors il doit se poursuivre par `elt2`
  la valeur rendue sera *expr*1
- commence par `elt3`
  la valeur rendue sera *expr*2
- commence par `elt4`,
  alors il doit se poursuivre par `elt5` puis `elt6`
  la valeur rendue sera *expr*3

Sémantique de `parser`

```
let analyse = parser
  | [< 'elt1 ; 'elt2 >] -> expr1
  | [< 'elt3 >] -> expr2
  | [< 'elt4; 'elt5; 'elt6 >] -> expr3
```

Si le flot en argument

- commence par `elt1`,
  alors il doit se poursuivre par `elt2`
  la valeur rendue sera *expr*1
- commence par `elt3`
  la valeur rendue sera *expr*2
- commence par `elt4`,
  alors il doit se poursuivre par `elt5` puis `elt6`
  la valeur rendue sera *expr*3
- sinon c'est une erreur (levée d'exception)

Analyse d'un flot : avec "consommation"

Dès qu'un élément est reconnu, il est consommé

Analyse d'un flot : avec "consommation"

Dès qu'un élément est reconnu, il est consommé

Récupération du premier élément d'un flot

```
# let next = parser
    [< 'x >] -> x ;;
```

Exemple :

```
# next nat;;
- : int = 0
# next nat;;
- : int = 1
```

Motif de flot vide [< >]

Un flot vide est toujours reconnu en préfixe d'un flot

Motif de flot vide [< >]

Un flot vide est toujours reconnu en préfixe d'un flot

Exemple

```
let analyse = parser
  | [< 'elt1 ; 'elt2 >] -> expr1
  | [< 'elt3 >] -> expr2
  | [<  >] -> exprcasvide
  | [< 'elt4; 'elt5; 'elt6 >] -> expr3
```

Motif de flot vide [< >]

Un flot vide est toujours reconnu en préfixe d'un flot

Exemple

```
let analyse = parser
  | [< 'elt1 ; 'elt2 >] -> expr1
  | [< 'elt3 >] -> expr2
  | [<  >] -> exprcasvide
  | [< 'elt4; 'elt5; 'elt6 >] -> expr3
```

- Le flot en entrée est toujours accepté
- les motifs qui suivent le motif vide ne sont jamais vus

Motifs de flots plus généraux

Les motifs sont des séquences (éventuellement vides)

- d'éléments (constants, variables, motifs)
  marqués par une apostrophe
- de suites d'éléments reconnus par une fonction d'analyse

`[< ...; r = analyse1 ; ... >] -> ...`

- appel de fonction sans apostrophe
- l'appel est placé dans le motif de flot
- la fonction est appliquée implicitement au flot courant
- liaison du résultat de cette fonction à un nom
  syntaxe semblable à **let** *nom* = fonc `in` ... mais sans **let** ni `in`
  r est disponible à droite de `analyse1`
  (motifs suivants et expression retournée à droite du `->`)

Signification du point-virgule dans un motif de flot

Indique « puis » indépendamment du nombre d'éléments reconnus :

- exactement 1 dans le cas d'un élément comme `'elt2`
- un nombre quelconque dans le cas d'un appel à une fonction

Plan

Analyseur de flot correspondant à une grammaire

Correspondance 1-1 entre un non-terminal et la fonction d'analyse qui consomme ce non-terminal en début de flot

Analyseur de flot correspondant à une grammaire

Correspondance 1-1 entre un non-terminal et la fonction d'analyse qui consomme ce non-terminal en début de flot

Dans la fonction correspondant au non-terminal N,
chaque motif de filtrage `[< ... >]` correspond au membre droit d'une règle de production de N.

Analyseur de flot correspondant à une grammaire

Correspondance 1-1 entre un non-terminal et la fonction d'analyse qui consomme ce non-terminal en début de flot

Dans la fonction correspondant au non-terminal N,
chaque motif de filtrage `[< ... >]` correspond au membre droit d'une règle de production de N.

A ::= B C

B ::= (B)
B ::= [B]
B ::= ϵ

C ::= #

Analyseur de flot correspondant à une grammaire

Correspondance 1-1 entre un non-terminal et la fonction d'analyse qui consomme ce non-terminal en début de flot

Dans la fonction correspondant au non-terminal N,
chaque motif de filtrage `[< ... >]` correspond au membre droit d'une règle de production de N.

A ::= B C	**let** epluche**A** = `parser`
	$\quad \mid [< b = \text{epluche}B\,;\ c = \text{epluche}C\ >] \rightarrow \ldots$
B ::= (B)	**let** `rec` epluche**B** = `parser`
B ::= [B]	$\quad \mid [< \text{''('}\,;\ b = \text{epluche}B\,;\ \text{''}) \text{'}\ >] \rightarrow \ldots$
B ::= ϵ	$\quad \mid [< \text{''['}\,;\ b = \text{epluche}B\,;\ \text{'']'}\ >] \rightarrow \ldots$
	$\quad \mid [< >] \rightarrow \ldots$
C ::= #	**let** epluche**C** = `parser`
	$\quad \mid [< \text{''\#'}\ >] \rightarrow \ldots$

Reconnaissance pure

Pas d'information supplémentaire construite durant le parcours

A ::= B C

B ::= (B)
B ::= [B]
B ::= ϵ

C ::= #

Reconnaissance pure

Pas d'information supplémentaire construite durant le parcours

A ::= B C	**let** epluche**A** = `parser`
	$\quad$ | [< () = epluche**B** ; _ = epluche**C** >] → ()
B ::= (B)	**let** `rec` epluche**B** = `parser`
B ::= [B]	$\quad$ | [< ''('; () = epluche**B** ; '')' >] → ()
B ::= ϵ	$\quad$ | [< ''['; () = epluche**B** ; '']' >] → ()
	$\quad$ | [< >] → ()
C ::= #	**let** epluche**C** = `parser`
	$\quad$ | [< ''#' >] → ()

Nombre de parenthèses

Calcul d'un entier durant le parcours

A ::= B C

B ::= (B)
B ::= [B]
B ::= ϵ

C ::= #

Nombre de parenthèses

Calcul d'un entier durant le parcours

A ::= B C
B ::= (B)
B ::= [B]
B ::= ϵ
C ::= #

```
let eplucheA = parser
    | [< n = eplucheB ; () = eplucheC >] -> n
let rec eplucheB = parser
    | [< ''('; n = eplucheB ; '')' >] -> n + 2
    | [< ''['; n = eplucheB ; '']' >] -> n
    | [< >] -> 0
let eplucheC = parser
    | [< ''#' >] -> ()
```

Arguments supplémentaires

Calcul d'un entier par accumulateur

A ::= B C

B ::= (B)
B ::= [B]
B ::= ϵ

C ::= #

Arguments supplémentaires

Calcul d'un entier par accumulateur

A ::= B C	**let** epluche**A** = `parser`
	$\quad \mid [< n = \text{epluche}B\ 0\,;\ () = \text{epluche}C\ >] \rightarrow n$
B ::= (B)	**let** `rec` epluche**B** a = `parser`
B ::= [B]	$\quad \mid [< \text{''('}\,;\ n = \text{epluche}B\ (a+1)\,;\ \text{'')'}\ >] \rightarrow n+1$
B ::= ϵ	$\quad \mid [< \text{''['}\,;\ n = \text{epluche}B\ a\,;\ \text{'']'}\ >] \rightarrow n$
	$\quad \mid [< >] \rightarrow a$
C ::= #	**let** epluche**C** = `parser`
	$\quad \mid [< \text{''\#'}\ >] \rightarrow ()$

Comparaison avec l'analyseur de liste

Les idées sont les mêmes que pour l'analyse de listes, avec les différences suivantes

- on ne s'encombre pas avec la liste (le flot) en cours d'analyse, il est implicitement présent
- plus grande facilité pour exprimer les choix
  (inutile de calculer le premier terminal qui pilote le choix)
- effet de bord : on ne revient pas sur un terminal consommé

Attention : nécessité de factoriser à gauche

- E ::= T + T
- E ::= T

En

- E ::= T SE
- SE ::= + T
- SE ::= ϵ

Sinon second motif jamais essayé.

Plan

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

Flot de caractères → reconnaissance d'un lexème (1)

Reconnaissance d'un caractère

```
let chiffre = parser [<'''0'..'9' as x >] → x
let lettre = parser [<'''a'..'z' | 'A'..'Z' as x >] → x
let special = parser [<''' :' | ' =' | ' <' | ' >' as x >] → x
let alphanum = parser
    | [<x = lettre >] → x
    | [<x = chiffre >] → x
```

Caractères consécutifs de même catégorie

```
let rec lettres = parser
    | [< x = lettre ; l = lettres >] → x :: l
    | [< >] → []
```

Flot de caractères → reconnaissance d'un lexème (2)

Caractères consécutifs de même catégorie

```
let rec alphanums = parser  etc.
let rec speciaux = parser  etc.
let rec horner n = parser  etc.
```

Un lexème, avec élimination des blancs précédents

```
let rec lexeme = parser
    | [< ' ' ' ; lx = lexeme >] → lx
    | [< x = lettre ; l = alphanums >] → Tident (x :: l)
    | [< x = chiffre ; n = horner (digit x)>] → Tent (n)
    | [< x = special ; l = speciaux >] → Tspeciaux (x :: l)
```

Flot de caractères → flot de lexèmes : version simple

Intermède : flot de caractères → liste de lexèmes

```
let rec liste_lexemes = parser
    | [< tk = lexeme ; l = liste_lexemes >] → tk :: l
    | [< >] → []
```

flot de caractères → flot lexèmes : semblable

```
let rec flot_lexemes = parser
    | [< tk = lexeme ; l = flot_lexemes >] → [<'tk ; l >]
    | [< >] → [< >]
```