LPAC

Langage pour l'Algèbre et le Calcul

Ensimag - 2ème Année

1992 - 1993

Contents

Chapter 1

Introduction

1.1 Classe d'équivalence - Forme normale

Soit E un ensemble et $\equiv$ une relation d'équivalence sur E. On appelle *forme normale* (canonique) l'unique représentation associée à une classe d'équivalence.

Tout algorithme consiste à associer à une forme donnée en entrée une forme normale en sortie, pour une certaine relation d'équivalence.

Exemple : " Trier une partie (finie) de $\mathbb{Z}$ par ordre strictement croissant"
On définit ici E et $\equiv$ par :

$$E = \{a \in P(\mathbb{Z}) / card(a) < +\infty\}$$

et

$$(a \equiv b) \Longleftrightarrow \left\{ \begin{array}{lcl} \forall x_i \in a, \exists x_j \in b & : & x_i = x_j \\ \forall x_i \in b, \exists x_j \in a & : & x_i = x_j \end{array} \right.$$

(Autrement dit $a = b$ dans $P(\mathbb{Z})$)

Soit:

$$\begin{array}{rcccl} \psi & : & E & \rightarrow & C = \{(x_1, \ldots, x_n) \in \mathbb{Z}^n / (n \in N) et((i < j) \Longrightarrow (x_i < x_j))\} \\ & : & a = x_1, \ldots, x_n & \mapsto & y(a) = (x_1, \ldots, x_n) / (i < j) \Longrightarrow (x_i < x_j) \end{array}$$

Ainsi : "Trier $a \in E$" devient équivalent à "Calculer $\psi(a)$".

Pour se ramener à une forme normale (i.e. calculer $\psi(a)$), une manière souvent efficace et souple est d'appliquer des *règles de réécriture*.

Ainsi, en reprenant l'exemple du tri ci-dessus et en appelant # l'opérateur de concaténation dans C, on peut se ramener à la forme normale en appliquant les règles :

Trier () `-->` ()
Trier ($a = \{x_1, ...\}$) `-->` Trier ($\{x \in a / x < x_1\}$)#(x_1)#Trier($\{x \in a / x > x_1\}$)

Motivation: La caractéristique principale de LPAC est de permettre d'associer à tout type de données une forme normale. Il est alors possible de ramener à tout moment un objet sous forme normale (phase de *réduction*). En outre, certaines opérations nécessitent une réduction préalable des objets pour être évaluées. C'est notamment le cas de l'égalité entre deux objets : il est possible de comparer deux objets d'un même type, mais avant de les comparer, les objets doivent être ramenés sous forme normale.

Remarque: un opérateur pouvant prendre comme opérande des données d'un type quelconque (ex. égalité, impression...) est dit *surchargé*.

1.2 Application au Calcul Algébrique dans $\mathbb{Q}[X]$ et remarques

Un polynôme de $\mathbb{Q}[X]$ peut être considéré comme étant soit un monôme, soit la somme de deux polynômes. L'opération d'addition + peut donc être considérée comme un constructeur de $\mathbb{Q}[X]$.

En LPAC, il est possible de définir des types construits : la déclaration conforme à ce qui est écrit ci-dessus s'énonce en Lpac :

```
TYPE  QX  ==
   CONST
      zero
    | mon   DE  entier # entier
    | plus  DE  QX # QX
   FINCONST
```

où `mon[n,d]` représente le monôme: nx^d.
Le polynôme nul peut alors être représenté par `mon[0,0]`, mais aussi par `mon[0,5]` ou encore : `plus[mon[-4,2],plus[mon[1,2],mon[3,2]]]`, ou encore `zero`.

Si l'on veut tester l'égalité de deux polynômes, il est nécessaire de pouvoir les ramener à une forme normale. Il est donc important de choisir une représentation de référence d'un polynôme. Soit $P \in \mathbb{Q}[X]$. On choisit $\psi(P)$, définie comme suit, comme représentation normale pour P:

$$\psi(P) = \sum_{i=0}^{n} a_i x^{\delta_i} \quad \text{avec} \begin{cases} a_i \neq 0 \\ i < j \Longrightarrow \delta_i > \delta_j \end{cases}$$

(autrement dit, somme des monômes non nuls de P, classés par degrés strictement décroissants)

Ramener un polynôme sous sa forme normale ψ peut alors être réalisé à l'aide des règles de réécriture suivantes :

$$
\begin{array}{rcll}
0.X^{d_2} & \longrightarrow & 0 & \\
0+P & \longrightarrow & P & \\
P+0 & \longrightarrow & P & \\
a_1X^{d_1}+a_2X^{d_2} & \longrightarrow & (a_1+a_2)X^{d_1} & \text{si} \quad d_1=d_2 \\
 & & a_2X^{d_2}+a_1X^{d_1} & \text{si} \quad d_1<d_2 \\
 & & a_1X^{d_1}+a_2X^{d_2} & \text{si} \quad d_1>d_2 \\
a_1X^{d_1}+(a_2X^{d_2}+P) & \longrightarrow & (a_1+a_2)X^{d_1}+P & \text{si} \quad d_1=d_2 \\
 & & a_2X^{d_2}+(a_1X^{d_1}+P) & \text{si} \quad d_1<d_2 \\
 & & a_1X^{d_1}+(a_2X^{d_2}+P) & \text{si} \quad d_1>d_2 \\
(P+Q)+R & \longrightarrow & P+(Q+R) & (+ \text{ associative})
\end{array}
$$

Ces règles s'écrivent en LPAC de la manière suivante :

```
REGLES
   DECL  a1 , d1 , a2 , d2 :  entier
   DECL  p , q , r         :  QX

     mon[0,_]         --> zero
   | plus[p, zero]    --> p    % p+0  -->  p %
   | plus[zero, p]    --> p    % 0+p  -->  p %
   | plus[mon[a1, d1], mon[a2, d2]] -->
              si       d1=d2  alors mon[a1+a2, d1]
              sinon si d1<d2  alors plus[mon[a2, d2], mon[a1, d1]]
                    sinon           plus[mon[a1, d1], mon[a2, d2]]
                    fsi
              fsi
   | plus[mon[a1, d1], plus[mon[a2, d2],p]]  -->
              si       d1=d2 alors plus[mon[a1+a2, d1], p]
              sinon si d1<d2 alors plus[mon[a2, d2], plus[mon[a1, d1],p]]
                    sinon          plus[mon[a1, d1], plus[mon[a2, d2],p]]
                    fsi
              fsi
   | plus[plus[p, q], r]  --> plus[ p, plus[q, r]]  % Associativite %
FREGLES
```

Ainsi, lors de la définition d'un type en LPAC, on peut déclarer les différentes règles qui permettent de ramener un objet du type sous forme normale. Réduire un objet grâce aux règles associées à son type peut alors être réalisé grâce à la primitive Lpac : *reduction*

```
reduction [ <objet> ]
```

Il est aussi possible d'associer au type défini une représentation externe. Imprimer un objet (c'est à dire lui associer sa forme externe sous forme de

chaîne de caractères) peut être réalisé grâce à la fonction d'impression associée à son type par la commande : *impression*

```
impression [ <objet> ]
```

On peut toujours afficher une chaîne de caractères sur la sortie standard, grâce à la commande : *affichage*

```
affichage [ <objet> ]
```

La fonction d'impression d'un polynôme sous le format somme de monômes s'écrit en LPAC :

```
IMPRESSION   p  ==
   DECL  n , d :  entier
   DECL  q , r   :  QX
   SELON  p  :
      zero      --> impression['0']
    | mon[n,d]  --> impression[n] ^ impression['X**'] ^ impression[d]
    | plus[q,r] --> impression[q] ^ impression['+'] ^ impression[r]
   FSELON  ;;
```

Remarque : les primitives `reduction` et `impression` sont bien sûr surchargées.

N.B. : la déclaration LPAC complète du type polynôme, avec règles de normalisation et fonction d'impression associées, est donnée page 49.

1.3 Sur l'utilisation des règles de réécriture

Il y a deux problèmes liés à l'utilisation de règles de réécriture:

1. Certaines règles peuvent entraîner une non-terminaison : (ex. : `a + b --> b + a`, ou `i --> i+1`)
2. l'ordre d'application des règles peut avoir une importance quant à la forme finale obtenue.

On dit qu'un ensemble de règles est

- *confluent* si quelque soit l'ordre d'application des règles on aboutit à une unique forme finale.

- *nœtherien* si quelque soit l'ordre d'application des règles on aboutit à une forme finale en un temps fini.

Les ensembles de règles présentés précédemment sont confluents, mais pas nœtheriens: l'application de la règle d'addition de deux monômes peut retourner la même somme de deux monômes. La détection qu'un système de règles est confluent est difficile: nous supposerons donc toujours que les règles LPAC données par l'utilisateur sont effectivement confluentes.
De plus, lorsqu'une expression ne peut pas être modifiée par une règle qui la filtre, nous dirons qu'elle correspond à une forme finale: si une telle forme peut être obtenue par l'application d'un nombre fini de règles, alors la réduction Lpac s'arrête en un temps fini. C'est pourquoi le système de règles décrit pour les polynômes est valide.

1.4 Autres Applications : Calcul Fonctionnel

L'application de règles de réécriture peut aussi être un moyen commode pour décrire certains algorithmes.

Par exemple, dériver ou intégrer un polynôme de$\mathbb{Q}[X]$ peut être réalisé grâce aux règles suivantes :

f' ($f = cte$)	-->	0
$(X)'$	-->	1
$(f+g)'$	-->	$f'+g'$
$(f.g)'$	-->	$f'.g + f.g'$
$\int u'.v$	-->	$[u.v] - \int u.v'$ (formule d'intégration par parties)

Il existe bien d'autres applications : la génération de code que vous décrirez en Lisp pourrait ainsi être -parfois- souplement décrite par des règles de réécriture.

Chapter 2

Patterns et Filtrage

Le but de ce chapitre est de préciser la notion de filtrage, et sa sémantique dans le langage Lpac.

2.1 Objets de base

On considère comme objets de base les objets définis ci-dessous :

- *élément atomique* == entier
  | booléen
  | chaîne de caractères (entre ')
  ex. `+123  3  -2  '123'  vrai  'turlu'`
- *constructeur* == symbole particulier pris dans un ensemble fini de symboles. Dans ce chapitre, les symboles suivants seront considérés comme constructueurs:

  `liste  mot  nil  cons  ent  plus  fact`
- *objet* == élément atomique
  | constructeur
  | constructeur `[` objet `]`
  | `(` objet `,` objet `)`
  ex. `liste[ (3, ent[5]), (liste[ mot['a','b'], nil])]`
  N.B. : les parenthèses sont omises lorsqu'elles sont implicites (elles sont toujours prioritaires par rapport aux crochets `[ ]`).

2.2 Gabarits et patterns

Pour représenter un ensemble d'objets, on introduit la ntion de gabarits et de patterns, qui font intervenir deux nouveaux éléments: les noms et

le joker.

- *nom* == variable (qui sera toujours typée par la suite)
  ex. `u toto5 farhenheit`

  Dans ce chapitre, tous les identificateurs autres que les constructeurs précités et les deux constantes `vrai` et `faux` seront considérés comme noms.

  On dira qu'une variable est *liée* si elle a été associée à une valeur (par exemple, initialisée). Une variable non liée est aussi dite *libre*.

- *joker* == variable particulière (notée '_'), qui ne peut être liée ni à une valeur ni à un type(qu'on ne peut donc référencer)

- *gabarit* (template) == objet arbitraire (objet comprenant éventuellement des noms ou des jokers) représentant une classe d'objets
  gabarit == élément atomique
  | constructeur
  | constructeur `[` gabarit `]`
  | nom
  | joker
  | `(` gabarit `,` gabarit `)`
  ex. `liste [ (_ ,2), (toto, 'ab')]`

- *pattern* (patron) == couple $\{t, C\}$ où t désigne un gabarit et C un ensemble de conditions: en Lpac, ces conditions correspondent à l'appartenance de chaque variable et constante qui apparaît dans le pattern à son propre type.
  ex. `plus [ (_ , 2), toto, 'ab']` , avec `toto` de type `entier`

  N.B. : lorsqu'une variable est liée à une valeur, un pattern la contenant est le même que celui obtenu en remplaçant toutes les occurences de cette variable dans ce pattern par sa valeur.

2.3 Filtrage et réécriture

- Un gabarit t *génère* un objet s (ou : s est généré par t) ssi s peut être obtenu en substituant les jokers et les noms libres dans t par des objets particuliers.

  Un gabarit t génère un ensemble d'objets S (ou : S est généré par t) ssi t génère chaque élément de S.

  ex. `('+',(_,2))` génère `('+',('x',2))` ou `('+',(plus[3,'b'],2))` mais ne génère pas l'expression `('+',('x',(3,2)))`
  Ainsi:

  – le gabarit `_` génère n'importe quel objet de taille 1 le gabarit
  – le gabarit `u,u` (`u` libre) génère n'importe quel objet de taille 2 (ex. `2,'x'`)

- Soit t un gabarit et s un objet.
  t *filtre* s ssi t génère s et les objets correspondant à des noms (libres ou liés) identiques dans t sont identiques dans s.

  ex. `u,u` filtre `2,2` et `'x','x'` mais ne filtre pas `2,'x'`
  `cons[ 2, x ]` filtre `cons[ 2, nil ]`

- Soit $p = \{t, C\}$ un pattern (t : gabarit, C : ensemble de conditions)
  (p *filtre* s) ssi t filtre s et les conditions C de p appliquées à s sont vraies (i.e. dans le cadre de LPAC, le type d'un nom libre (ou lié) de t est le même que celui de la valeur qui lui correspond dans s.

  ex. `u,u` avec `u entier` filtre `2,2` mais pas `'x','x'`
  `_, _` filtre tout couple (objet de taille 2)

- On appelle *règle de réécriture* un couple ($p - - > R$) où p est un pattern et R un objet (expression de remplacement)

  ex.

```
fact [  0 ]            -> 1
fact [ u ]  {u entier} -> si (u>0) alors u * fact [ u-1 ]
                                   sinon fact[0]
                          fsi
```

- On appelle *filtrage* (ou matching) l'action consistant à lier dans un pattern les variables libres (noms libres) à des valeurs qui permettent de le rendre égal à un objet (lorsque le pattern filtre l'objet)

  ex. filtrer le pattern `{ (u,2),(v,v)  {u entier, v entier#entier}}` avec l'expression `(3,2),((5,2),(5,2))` a pour résultat de lier les variables `u` et `v` aux valeurs: `3` et `(5,2)`.

- On appelle unification l'action consistant à trouver les valeurs auxquelles il faut lier les variables non liées (noms) pour rendre les deux patterns égaux - autrement dit, trouver l'ensemble des séquences filtrées par deux patterns -.

  ex. unifier `{(u,(u,v))  {}}` avec `{( (w,_), (( _,4),3 )), {}}`
  a pour résultat le pattern `{ ((x,4), ((x,4),3))    {} }`

Chapter 3

Le Langage LPAC

3.1 Introduction - Motivation

Lpac est un langage fonctionnel et symbolique et qui intègre la manipulation de patterns. Il permet l'application de règles de réécriture et le filtrage.

Figure 3.1: Structure générale du langage LPAC

Remarque : Dans les paragraphes suivants de cette partie, les principaux schémas syntaxiques de Lpac sont présentés. La grammaire complète du langage est décrite dans le chapitre 4.

3.2 Types

Lpac est un langage fortement typé : tout objet manipulé dans un algorithme Lpac doit donc être déclaré comme appartenant à un type connu. Il existe trois différentes catégories de types :

3.2.1 Types pré-définis

Ce sont les types: entier (`entier`), chaîne de caractères (`string`), booléen(`booleen`), et vide(`void`).

Le type vide correspond à des expressions sans résultat (ex. `tantque`). Les valeurs du type booléen sont : `vrai` et `faux`.

3.2.2 Types énumérés

Ce sont les types produits, analogue au Record de Pascal.

Syntaxe :

```
<type1> # <type2> # ... # <typen>
```

ex.1
`3 , 'toto'` est de type `entier # string`

Remarque Importante. `#` et `,` ne sont pas associatifs : `u,(v,w)` n'a pas même type que `(u,v),w`
Pour des raisons de commodité d'écriture, on adopte la convention : `u, v, w == u,(v,w)` (et l'on dit alors, par abus de langage, que `#` et `,` sont associatifs à droite).

ex.2
`1,(2,3)` est de type `entier#(entier#entier)`, et est complètement différent (car pas de même type) de `(1,2),3` qui est de type `(entier#entier)#entier`.

3.2.3 Types foncteurs

Le type foncteur `E -> F` est celui des fonctions dont le plan de départ (type des paramètres) est `E` et qui retourne une valeur de type `F`.

Syntaxe :

```
<type1> -> <type2> -> ... -> <typen>
```

ex. : `(entier # string) -> entier`

Remarque Importante. L'opérateur `->` n'est pas associatif : une fonction qui prend son paramètre dans `E` et qui retourne une expression de type `F->G` n'est pas comparable à une fonction qui prend son paramètre dans `E->F` et qui retourne une expression de type `G`.

3.2.4 Types construits

Ce sont les types union, analogue au *Record*...*Case* de Pascal

Syntaxe :

```
type   <nom_type> == <constructeurs>   <imprimer>   <regles>

<constructeurs>   == const <liste_type_decl>  fconst

<liste_type_decl> == <type_decl> | <liste_type_decl> | <type_decl>

<type_decl>       == <ident>     | <ident> de <type>
```

ex. :

```
type complexe ==
       const
            reel_pur  de entier
          | imag_pur  de entier
          | reel_imag de entier # entier
       fconst
```

Format d'impression pour un type

On peut associer à un type construit un format d'impression du type : il permet d'associer à un objet du type la chaîne de caractères le représentant sous sa forme externe. Ce format se décrit à l'aide d'une fonction dont la définition est intégrée dans la déclaration du type. L'appel à la primitive surchargée `impression` pemet d'imprimer un objet LPAC quelconque, grâce à la fonction d'impression associée à son type.

Syntaxe d'imprimer :

```
impression <ident>  == <corps de la fonction d'impression>
```

ex. : La définition du type complexe est complétée ici :

```
type complexe ==
       const
            reel_pur  de entier
          | imag_pur de entier
          | reel_imag de entier # entier
       fconst
       impression   z   ==
          decl  n , p  :  entier
          selon  z  :
               reel_pur [ n ]    --> impression[n]
             | imag_pur [ n ]    --> 'i.' ^ impression[n]
             | reel_imag [ n, p] --> impression[n] ^ '+i.' ^ impression[p]
          fselon
```

A chaque type pré-défini est associé une fonction d'impression. Le format d'impression d'un type énuméré est défini comme suit.
Soit `x1` de type `T1` et `x2` de type `T2`; le format d'impression de `x1,x2` est :

```
( impression[x1] , impression[x2] )
```

Avec cette convention, tous les objets de types énumérés ne contenant que des types pré-définis ou construits avec fonction d'impression associée peuvent être imprimés.

Pour écrire un objet sur la sortie standard, la primitive s'appelle `affichage` (cf introduction et dans ce chapitre la sous-section 3.9.1).

Règles de réécriture

Les règles de réécriture sont une des caractéristiques fondamentales associées aux types construits. Elles permettent de ramener un objet d'un type donné à une forme normale.

Syntaxe :

```
regles  <decl> <liste_de_reecriture>   fregles

<reecriture> : <pattern>  -->  <expression>
<pattern>    : (   <pattern>   )
             | <pattern> , <pattern>     % types enumeres     %
             | <ident>   [ <pattern> ]   % types construits   %
             | <ident>                   % noms               %
             | _                         % joker             %
             | vide                      % sequence vide      %
```

```
                | int                          % atomes  entiers    %
                | string                       % atomes  chaines    %
```

ex. : au type “complexe” défini précédemment, on peut associer - à titre d'exemple - les règles de réduction suivantes :

```
type complexe ==
       const
            reel_pur  de entier
          | imag_pur de entier
          | reel_imag de entier # entier
       fconst
       impression   z   ==
            decl  n , p  :  entier
            selon  z  :
                 reel_pur[n]    --> impression[n]
               | imag_pur[n]    --> 'i.' ^ impression[n]
               | reel_imag[n,p] --> impression[n] ^ '+ i.' ^ impression[p]
            fselon
       regles
            decl  n  :  entier
            reel_imag [0, n] --> imag_pur [n]
          | reel_imag [n, 0] --> reel_pur [n]
          | imag_pur [0]     --> reel_pur [0]
       fregles
```

Remarque : on ne s'occupe pas de vérifier si les systèmes de règles donnés sont nœthériens et confluents. Il pourra donc y avoir des problèmes dans l'application des règles : dans l'exemple ci-dessus, si l'on ajoute la règle :

```
          | imag_pur [ n] --> reel_imag[0,n]
```

il y a certains problèmes...

Les règles sont activées par l'appel à la fonction surchargée `reduction`. La commande:

```
         reduction  [ u ]
```

permet de réduire grâce aux règles associées au type de u la valeur de la variable LPAC u.

3.3 Variables globales

Même si Lpac est un langage essentiellement fonctionnel, il est possible de déclarer des variables. Comme en Pascal, elles existent dans l'environnement global dès leur déclaration - et jusqu'à une éventuelle redéclaration...-. Deux contraintes importantes sont imposées aux variables globales :

- Toute variable doit être typée lors de sa déclaration
- Toute variable doit être initialisée lors de sa déclaration

Syntaxe :

```
var  <ident>    :   <type_expr>    ==   <expression>
```

ex. :

```
var toto : entier                          ==   3 ;;
var u    : entier # (entier # entier)      ==   3, (4, 5) ;;
```

3.4 Déclaration de fonctions

Syntaxe :

```
fonction <ident> <plan> : <type_expr> == <corps_de_fonction>
```

`<plan>` décrit l'ensemble de départ de la fonction : il donne la description précise des paramètres de la fonction. Le seul mode de passage de paramètres est le passage par valeur. `<type_expr>` décrit l'ensemble d'arrivée de la fonction.

ex.1 :

```
fonction max[u:entier, v:entier]:entier == <corps de la fonction max>;;
```

La fonction `max` ainsi définie est de type : `(entier # entier) -> entier`.

Une fonction est donc considérée comme un objet du langage : elle a un type. On pourra donc l'affecter à une variable du même type, ou la passer en paramètre dans l'appel d'une autre fonction.

ex.2 :

```
var f   :   (entier # entier) -> entier   ==   max ;;
```

ex.3 : supposons qu'outre la fonction `max`, on définisse aussi les fonctions `min` et `g` :

```
fonction min[u:entier, v:entier]:entier == <corps de la fonction min>;;
fonction g[f:(entier#entier)->entier,u:entier,v:entier]:entier==f[u,v];;
```

On pourra alors écrire :

```
g [ min, 3, 4 ];;
g [ max, 5, 2 ];;
```

c'est à dire passer la fonction `min` (ou la fonction `max`) comme paramètre de la fonction `g`.

3.5 Expressions

Les expressions constituent l'ensemble des opérations élémentaires que permet Lpac. Elles sont assez variées et souvent très différentes des expressions Pascal dans leur fonctionnement.

Syntaxe :

```
  ( <expression> )                                % Parenthesage          %
| <expression> ; <expression>                     % Suite d'expressions   %
| tantque <expression> faire <expre ssion> fait   % Boucle                %
| si <expression> alors <expression > fsi         % Test                  %
| si <expression> alors <expression >
                  sinon <expression> fsi         % Test                  %
| selon <expression>:<appel_par_pattern> fselon   % Appel par pattern     %
| soit <liaison> dans <expression> fsoit          % Liaison               %
| <expression> := <expression>                    % Affectation          %
| <expression> , <expression>                     % Concatenation        %
| <expression> [ <expression> ]                   % Appel de fonction
                                                    ou de constructeur    %
| <expression><operateur binaire><expression>     % Operations binaires   %
| <operateur unaire>  <expression>                % Operations unaires    %
| ident                                           % Identificateur       %
| int                                             % Constante entier     %
| string                                          % Constante chaine     %
| empty                                           % Constante vide       %
```

Les différentes opérations arithmétiques - unaires et binaires - sont les suivantes :

1. opérateurs unaires : `+` (plus unaire), `-` (moins unaire)
2. opérateurs binaires :
   - `ou` (ou booléen), `&` (et booléen)

Opérateurs	Priorité	Associativité	Nombre d'opérandes	Type des opérandes
;	1	à gauche	2	tous types
:=	2	non associatif	2	de même type
,	3	à droite	2	tous types
ou	4	à gauche	2	booléen
&	5	à gauche	2	booléen
non	6	à gauche	1	booléen
=	7	à gauche	2	de même type
<>	8	à gauche	2	de même type
< <= > >=	9	à gauche	2	entier
^	10	à gauche	2	string
+ -	11	à gauche	2	entier
* / \	12	à gauche	2	entier
+ -	13	à gauche	1	entier
[ ]	14	à droite	2	tous types

Figure 3.2: Précédence et associativité des principaux opérateurs

- `^` (concaténation de chaînes de caractères),
- `<=` (inférieur ou égal), `<` (inférieur strict), `>=` (supérieur ou égal), `>` (supérieur strict),
- `+` (addition), `-` (soustraction), `*` (produit), `/` (division euclidienne), `\` (reste euclidien)
- `=` (égal), `<>` (différent),

Le figure 3.2 indique les priorités pour les principaux opérateurs.

Remarques :

- Toutes les expressions "classiques" sont autorisées en Lpac.
- A noter deux nouvelles instructions : l'appel par pattern et la liaison qui sont présentées plus précisément dans les paragraphes suivants.

ex.1 : expression arithmétique:

```
2 + 3 * ( u - 2 ) /  si (a = b) alors 3 sinon 5 fsi
```

ex.2 : appel de fonction avec paramètres:

```
f [ 3, g[ 4 , 5 ] + 7 ]
```

ex.3 : boucle:

```
i := 1 ; total := 0 ;
tantque  i <= 10  faire
   total := total + i * 3
   i := i + 1;
fait
```

ex.4 : calcul de pgcd (pour entiers positifs):

```
tantque (v<>0) faire
   u,v := si (v<u) alors u-v,v sinon v-u,u fsi
fait; u
```

3.6 Liaisons

C'est la première instruction qui différencie fondamentalement Lpac de Pascal, et qui le rapproche de Lisp. La liaison permet en fait de filtrer un pattern avec une expression. Deux cas se présentent lorsqu'un indentificateur de variable figure dans un pattern :

- la variable est déjà liée à une valeur: dans le pattern, c'est alors cette valeur de la variable qui est considérée.
- la variable n'est pas initialisée (elle n'est donc liée à aucune valeur): dans le pattern, la variable est alors liée avec la valeur *unique* qui permet de rendre le filtrage possible.

Syntaxe :

```
soit    <liaison>   dans   <expression>   fsoit

liaison :   <liaison>   et   <liaison>
        |   <pattern>   ==   <expression>
```

ex.1 :

```
decl u,v : entier
soit u, v  ==  3, 4  dans  u + v fsoit;
```

La valeur de cette expression est 7 si u et v ne sont pas liés.

ex.2 :

```
decl u,v : entier
soit  u  ==  3  et  v  ==  4  dans u + v fsoit;
```

(idem à ex.1)

ex.3 : en supposant `f : entier -> entier # (entier # entier)`

```
decl u : entier
soit u,(u,_) ==  f [7] dans u fsoit
```

Pour que le filtrage soit ici possible, il faut donc que les deux premières composantes de `f[7]` soient égales, auquel cas la valeur de la première composante de `f` sera associée à `u`. Si ce n'est pas le cas, une erreur devra être produite lors de l'exécution.

Remarques :

- Une liaison n'a d'effet que dans l'expression qui lui est associée: soit le programme suivant (`u` est supposée non liée, `u` et `v` sont de type `entier`):

  ```
  soit u==1 dans v:=u fsoit
  ```

  Après cette liaison, `v` est liée à la valeur `1`, alors que `u` n'a pas de valeur (`u` reste libre).

- `p1==e1 et p2==e2` est équivalent à `:(p1,p2) == (e1,e2)`.

3.7 Appel par pattern

Comme la liaison, l'appel par pattern permet de filtrer un pattern avec une expression.

Syntaxe :

```
selon <expression>   :
    <pattern1>   -->   <expression1>
  | <pattern2>   -->   <expression2>
  |  ...
  | <patternk>   -->   <expressionk>
fselon
```

On cherche le premier `i` tel que `<patterni>` filtre `<expressioni>`. On filtre alors `<patterni>` avec `<expressioni>`, et on renvoie la valeur de `<expressioni>` comme résultat de l'appel par pattern.

Remarque : Différentes précisions seront à donner, notamment concernant les règles de typage, et les erreurs éventuelles à l'exécution lorsqu'aucun filtrage n'est possible.
ex. : En reprenant le type `complexe`, on peut définir la forme externe d'un complexe z sous la forme $Re(z) + i.Im(z)$ de la façon suivante:

```
selon  z  :
    reel_pur [ n ] --> impression[n]
  | imag_pur [ n ] --> impression['i.'] ; impression[n]
  | reel_imag[n,p] --> impression[n] ^ impression['+ i.'] ^ impression[p]
fselon
```

3.8 Déclaration de variables locales

On peut déclarer des variables locales à une fonction :

Syntaxe :

```
decl  <ident1>, ..., <identk>   :   <type>
```

ex. :

```
fonction pairp [ n : entier ] : entier  ==
   decl p : entier
   p := n \ 2;
   si  p = 0  alors  1  sinon  0 fsi
   ;;
```

3.9 Extensions - Compléments

Ce paragraphe contient les différentes particularités "annexes" de Lpac qui n'ont pas encore été présentées.

3.9.1 La primitive `affichage`

La primitive `impression` permet d'associer à un type construit une forme externe, sous forme de chaîne de caractères; mais elle ne réalise pas l'impression

sur la sortie standard de cette chaîne.
La primitive `affichage` permet d'envoyer la forme externe d'un objet (de type quelconque, mais dont on peut obtenir une forme externe...) sur la sortie standard.
Le format de sortie est alors la chaîne de carctères correspondant à l'expression, le caractère : suivi du type Lpac de l'expression.

ex.

```
affichage[ 2,(vrai, ('toto' ^ impression[4])) ];;
```

permet d'envoyer sur la sortie standard :

```
(2, (vrai, 'toto4') : entier # (booleen # string)
```

et

```
affichage[ reel_imag[2, 3] ];;
```

permet d'envoyer sur la sortie standard :

```
2 + i.3 : complexe
```

3.9.2 Commentaires

Est considéré comme commentaires tout ce qui figure entre deux caractères `%`.
ex. : `% Bon appetit...%`

3.9.3 Déclaration `et` : Récursivité croisée

Lorsque deux fonctions s'appellent mutuellement, la déclaration `et` permet de les déclarer simultanément (de façon à pouvoir les compiler et vérifier la cohérence de leurs types simultanément). La déclaration et est tout à fait analogue à la directive Pascal forward.

ex. :

```
fonction  f  [...] :...== ... g[...]...;;
et
fonction  g  [...] :...== ... f[...]...;;
```

La compilation des déclarations `et` n'est pas demandée explicitement. Elle pourra être envisagée en fin de projet.

Chapter 4

Grammaire de Lpac

Dans les paragraphes suivants, l'ensemble des unités lexicales et les différents schémas syntaxiques de Lpac sont présentés, sous le format entrée de données de Leyacc. Le mode de construction de l'arbre abstrait est indiqué ensuite.

4.1 Unités lexicales

Le symbole associé à l'unité lexicale est donné après le mot clé `token`; les chaînes de caractères correspondant à cette unité lexicale sont encadrées par les délimiteurs `/*L` et `*/`.

```
%token FIN_EXPR         /*L ;;                                  */
%token ET               /*L et ET and AND                       */
%token FONCTION         /*L fonction FONCTION function FUNCTION  */
%token VAR              /*L var VAR                             */
%token TYPE             /*L type TYPE                           */

%token DEF              /*L ==                                  */

%token DECL             /*L decl DECL                           */
%token EMPTY            /*L ()                                  */
%token TYPE_FCT         /*L ->                                  */

%token CONS             /*L const CONST                         */
%token FCONS            /*L fconst  FCONST  endconst ENDCONST    */
%token REGLES           /*L regles REGLES rules RULES           */
%token FREGLES          /*L fregles FREGLES endrules ENDRULES    */
%token DE               /*L de DE of OF                         */

%token SOIT             /*L soit SOIT let LET                   */
```

```
%token DANS             /*L dans DANS in IN                        */
%token FSOIT            /*L fsoit FSOIT endlet ENDLET              */

%token TANTQUE          /*L tantque TANTQUE while WHILE            */
%token FAIRE            /*L faire FAIRE do DO                      */
%token FAIT             /*L fait FAIT done DONE                    */

%token SELON            /*L selon SELON case CASE                  */
%token FSELON           /*L fselon FSELON endcase ENDCASE          */

%token REC              /*L -->                                    */

%token AVEC             /*L avec AVEC with WITH                    */
%token JOKER            /*L _                                      */
%token SI               /*L si SI if IF                            */
%token ALORS            /*L alors ALORS then THEN                  */
%token SINON            /*L sinon SINON else ELSE                  */
%token FSI              /*L fsi FSI endif ENDIF                    */
%token AFFECTATION      /*L :=                                     */

%token ET_BOOL          /*L &                                      */
%token OU_BOOL          /*L ou OU or OR                            */
%token NON              /*L non NOT not NOT                        */
%token DIF              /*L /= <>                                  */
%token INF_EGAL         /*L <=                                     */
%token INF              /*L <                                      */
%token SUP_EGAL         /*L >=                                     */
%token SUP              /*L >                                      */

%token PLUS             /*L +                                      */
%token MOINS            /*L -                                      */
%token MULT             /*L *                                      */
%token DIV              /*L /                                      */
%token REM              /*L \                                      */

%token INT
%token IDENT
%token STRING
```

4.2 Précédences et associativité des opérateurs

```
%right       TYPE_FCT   /* a -> b -> c == a -> (b -> c) */
%right       '#'

%left        ET
```

```
%nonassoc    DEF
%left        '|'

%left        ';'
%nonassoc    AFFECTATION
%right       AVEC

%right       ','

%left        OU_BOOL
%left        ET_BOOL
%left        NON
%left        '='
%left        DIF
%left        INF_EGAL INF SUP_EGAL SUP

%left        '^'
%left        PLUS MOINS
%left        MULT DIV REM
%left        UNARY
%right       '['
```

4.3 Schémas syntaxiques

```
expression_evaluable :
   error FIN_EXPR
 | error '\000'
 | declaration FIN_EXPR
 | expression FIN_EXPR
 | FIN_EXPR
 | '\000'
 ;

declaration :
   declaration ET declaration    (A ne pas considerer)
 | FONCTION IDENT plan ':' type_expr DEF corps_de_fonction
 | VAR IDENT ':' type_expr DEF expression
 | TYPE IDENT DEF constructeurs imprimer regles
 ;

corps_de_fonction :
   decl_locales expression
 ;

decl_locales :
```

```
   /* vide */
 | decl_locales DECL liste_ident ':' type_expr
 ;

plan :
   '[' plan ']'
 |  liste_ident ':' type_expr ',' plan
 | liste_ident ':' type_expr
 | EMPTY
 ;

liste_ident :
   IDENT
 | liste_ident ',' IDENT
 ;

type_expr :
   '(' type_expr ')'
 | type_expr TYPE_FCT type_expr
 | type_expr '#' type_expr
 | IDENT
 ;

constructeurs :
   CONS liste_type_decl FCONS
 ;


liste_type_decl :
   liste_type_decl '|' type_decl
 | type_decl
 ;

type_decl :
   IDENT
 | IDENT DE type_expr
 ;

imprimer :
   /* vide */
 | IDENT IDENT DEF corps_de_fonction
 ;
regles :
   /* vide */
 | REGLES decl_locales liste_de_regles FREGLES
 ;
```

```
liste_de_regles :
   liste_de_regles '|' pattern REC expression
 | pattern REC expression
 ;

expression :
   '(' expression ')'
 | expression ';' expression
 | TANTQUE expression FAIRE expression FAIT
 | SELON expression ':' appel_par_pattern FSELON
 | SOIT liaison DANS expression FSOIT
 | expression AVEC liaison
 | SI expression ALORS expression FSI
 | SI expression ALORS expression SINON expression FSI
 | '"'
 | expression AFFECTATION expression
 | expression ',' expression
 | expression OU_BOOL expression
 | expression ET_BOOL expression
 | expression '=' expression
 | expression DIF expression
 | expression INF_EGAL expression
 | expression INF expression
 | expression SUP_EGAL expression
 | expression SUP expression
 | expression '^' expression
 | expression PLUS expression
 | expression MOINS expression
 | expression MULT expression
 | expression DIV expression
 | expression REM expression
 | PLUS expression  %prec UNARY
 | MOINS expression  %prec UNARY
 | expression '[' expression ']'
 | INT
 | IDENT
 | EMPTY
 | STRING
 ;

liaison :
   '(' liaison ')'
 | pattern DEF expression
 | liaison ET liaison
 ;

appel_par_pattern :
```

```
   appel_par_pattern '|' pattern REC expression
 | pattern REC expression
 ;

pattern :
   '(' pattern ')'
 | pattern ',' pattern
 | IDENT '[' pattern ']'
 | IDENT
 | JOKER
 | EMPTY
 | INT
 ;
```

4.4 Grammaire de l'arbre abstrait

La meilleure façon de comprendre la construction de l'arbre abstrait est de visualiser les quelques exemples donnés en annexe et d'utiliser directement l'analyseur (dont le fonctionnement est aussi décrit en annexe).

```
;-- 1. Expressions evaluables
; expression evaluable ==
;   expr-eval-eof
; | expr-eval-vide
; | expr-eval-decl de declaration
; | expr-eval-expr de expression

;-- 2. Declarations
; declaration ==
;   decl-et de declaration et declaration  ( A ignorer !!!)
; | decl-fct de chaine # plan # type-expr # corps-de-fonction
; | decl-var de chaine # type-expr # expression
; | decl-type de chaine # constructeurs # imprimer # regles

;-- 3. Corps de fonction
; corps-de-fonction ==
;   corps-de-fct de decl-locales # expression

;-- 4. Declarations locales
; decl-locales ==
;   decl-loc-vide
; | decl-loc-liste-var de liste-ident # type-expr
; | decl-loc-et de decl-locales # decl-locales
```

```
;-- 5. Plan
; plan ==
;   plan-et de plan # plan
; | plan-liste-ident de liste-ident # type-expr
; | plan-vide

;-- 6. liste-ident
; liste-ident ==
;   liste-var de chaine # liste-ident
; | var-ident de chaine

;-- 7. Expressions de type
; type-expr ==
;   type-expr-fct de type-expr # type-expr
; | type-expr-prod de type-expr # type-expr
; | type-expr-ident de chaine

;-- 8. Constructeurs
; constructeurs ==
;   const-et de constructeurs # constructeurs
; | const-decl de type-decl

;-- 9. Declarations de type
; type-decl ==
;   type-decl-ident de chaine
; | type-decl-comp de chaine # type-expr

;-- 10. Fonction d'impression pour un type
; imprimer ==
;   imp-vide
; | imp-fct de chaine # chaine # corps-de-fonction

;-- 11. Regles de simplification pour un type
; regles ==
;   regles-vide
; | regles-decl de decl-locales # regles-liste

;-- 12. Liste de regles
; regles-liste ==
;   regles-pat de pattern # expression
; | regles-et de regles # regles

;-- 13. Expressions
; expression ==
;   expr-seq de expression # expression
; | expr-boucle de expression # expression
; | expr-app-pat de expression # appel-par-pattern
```

```
; | expr-liaison-locale de liaison # expression
; | expr-si-al-sinon de expression # expression # expression
; | expr-si-alors de expression # expression
; | expr-affect de expression # expression
; | expr-couple de expression # expression
; | expr-algebrique de ...
; | expr-ou-bool de expression # expression
; | expr-et-bool de expression # expression
; | expr-egal de expression # expression
; | expr-diff de expression # expression
; | expr-inf-egal de expression # expression
; | expr-inf de expression # expression
; | expr-sup-egal de expression # expression
; | expr-sup de expression # expression
; | expr-strg-concat de expression # expression
; | expr-plus de expression # expression
; | expr-moins de expression # expression
; | expr-mult de expression # expression
; | expr-div de expression # expression
; | expr-rem de expression # expression
; | expr-plus-u de expression # expression
; | expr-moins-u de expression # expression
; | expr-apply de expression # expression
; | expr-int de entier
; | expr-ident de chaine
; | expr-empty
; | expr-string de chaine

;-- 14. Liaisons
; liaison ==
;   liaison-et de liaison # liaison
; | liaison-un de pattern # expression

;-- 15. Appel par pattern
; appel-par-pattern ==
;   appel-pat-et de appel-par-pattern # appel-par-pattern
; | appel-pat-un de pattern # expression

;-- 16. Patterns
; pattern ==
;   pattern-couple de pattern # pattern
; | pattern-apply de chaine # pattern
; | pattern-ident de chaine
; | pattern-joker
; | pattern-empty
; | pattern-int de entier
```

Chapter 5

But du Projet

Le but du projet est l'écriture d'un compilateur pour le langage Lpac, c'est à dire un programme qui reçoit en entrée un algorithme Lpac, et qui est capable de :

- l'analyser syntaxiquement : i.e. vérifier si il correspond bien à la grammaire du langage décrite au III.
- l'analyser sémantiquement : i.e. vérifier la cohérence des types des objets manipulés.
- générer du code compréhensible pour la machine où l'on implémente le langage Lpac. Le langage cible choisi est LeLisp, implanté sur Bull-SPS 7.

5.1 L'analyseur syntaxique

Il a été généré automatiquement à l'aide de LeYacc (version de Yacc adaptée à la génération en LeLisp). Il fournit un arbre abstrait du programme - si la syntaxe est évidemment conforme à la grammaire -. Cet arbre abstrait est la représentation, sous forme de donnée Lisp manipulable, de l'algorithme Lpac initial. Chaque unité lexicale et syntaxique y a été reconnue, et analysée.

5.2 L'analyseur sémantique

Il permet de vérifier si les objets manipulés par l'algorithme ont bien un typage correct.
ex. : `u + 3` n'a de sens que si `u` est entier.

Figure 5.1: Situation du compilateur Lpac

5.3 Le générateur de code

A partir de l'arbre abstrait, il génère le code LeLisp exécutable qui correspond aux commandes Lpac décrites dans l'algorithme.

5.4 Remarques

De nombreuses restrictions sont à faire aussi bien au niveau du typage des expressions Lpac qu'au sens à leur donner pour leur exécution. En effet, dans ce fascicule ne sont décrits que des exemples typiques pour expliquer approximativement les instructions, sans s'occuper des problèmes de typage ou de non-filtrage (etc...). La compilation systèmatique oblige à préciser quel sens exact est donné à une instruction, le but ultime étant d'autoriser le plus de commandes possibles, tout en assurant que leur exécution n'entraînera pas d'erreurs.

Chapter 6

Quelques Exemples Lpac

Pour que vous puissiez vous familiariser avec le langage LPAC, un compilateur est mis a votre disposition. Ce compilateur devrait ressembler au votre... mais pour éviter des copies abusives, il ne travaille pas sur les mêmes arbres abstraits et n'implante pas tout à fait la même sémantique. Pour appeler ce compilateur sur un fichier déjà écrit avec le suffixe `.lpac` : exemple `toto.lpac`

```
/users/jlroch/lpac/bin/clpac toto
```

permet de compiler le fichier `toto.lpac` : si la compilation est réussie, un fichier lelisp `toto.ll` est créé. Pour exécuter ce fichier :

```
/users/jlroch/lpac/bin/execlpac toto
```

lance lelisp avec en entrée le chargement de `toto.ll` puis l'appel à la fonction `(main)` qui est définie dans `toto.ll`.

Vous pouvez tester ce compilateur et surtout le vôtre sur les exemples lpac figurant dans le directory `/users/jlroch/lpac/EXEMPLES`.
Vous trouverez aussi des fichiers de test pour votre compilateur dans le directory `/users/jlroch/lpac/TESTS`.

Nous reprenons ici les exemples décrits au chapitre 1.

6.1 Axiomatique de Peano

Définition de $\mathbb{N}$:

- 0 désigne un élément particulier de $\mathbb{N}$
- $succ : \mathbb{N} \longrightarrow \mathbb{N}$ (application)
- Axiomes :
  - $0 \neq succ(a) \forall a \in \mathbb{N}$
  - *succ* est injective

  - Tout sous-ensemble de $\mathbb{N}$ qui contient 0 et le successeur de tout élément du sous-ensemble, coïncide avec $\mathbb{N}$

- Addition :
  - $0 \oplus m = m$
  - $succ(n) \oplus m = succ(n \oplus m)$
- Multiplication :
  - $0 \otimes m = 0$
  - $succ(n) \otimes m = n \otimes m \oplus m$

La définition Lpac de $\mathbb{N}$ suit cette axiomatique de très près:

```
type  N ==
   const
        zero
      | succ  de N
   fconst
   impression  n  ==
        decl  p  :  N
        selon  n  :
             zero --> '0'
           | succ[p] --> 'succ(' ^ impression[p] ^ ')'
        fselon
;;

fonction  plus [  l : N, m : N ] : N ==
        decl  n : N
        selon  l  :
             zero  --> m
           | succ[n] --> succ[plus[n,m]]
        fselon ;;

 fonction  mul [  l : N, m : N ] : N ==
        decl  n : N
        selon  l  :
             zero  --> zero
           | succ[n] --> plus[mul[n,m],m]
        fselon ;;
```

Remarque On aurait aussi pu définir `plus` et `mul` comme des constructeurs du type $\mathbb{N}$. Les règles qui les définissent auraient alors été définies à l'aide de la primitive `regles`.

6.2 Tri

On reprend l'algorithme décrit au chapitre 1. Le type liste d'entiers peut être défini de la façon suivante :

```
type  liste  ==
 const
   nil
  | cons  de  entier # liste
 fconst
 impression   l  ==
  decl  n   :  entier
  decl  cdrl   :  liste
  selon  l  :
    nil            --> '()'
   | cons[n, cdrl] --> '(' ^ impression[n] ^ impression [ cdrl ] ^ ')'
  fselon  ;;
```

Les primitives de manipulation des objets du type liste s'écrivent :

```
fonction   car  [ l : liste ] : entier  ==
    % car n'est defini que si l n'est pas vide $
     decl  n  :  entier
     soit cons[ n, _]  ==  l dans n fsoit   ;;

fonction   cdr  [ l : liste ] : liste  ==
  decl  cdrl  :  liste
  selon  l  :
      nil          --> nil
   | cons[_, cdrl] --> cdrl
  fselon  ;;

fonction   append  [ l1, l2 : liste ] : liste  ==
  %  Programmation recursive classique  %
  si  l1 = nil  alors  l2
                sinon  cons[ car[l1], append [ cdr[l1], l2] ]
  fsi ;;

fonction   append  [ l1, l2 : liste ] : liste  ==
  %  Programmation equivalente, mais en utilisant le filtrage  %
  decl  n  :  entier
  decl  cdr_l1  :  liste
  selon  l1  :
     nil             --> l2
   | cons[n, cdr_l1] --> cons[n, append [ cdr_l1, l2] ]
  fselon  ;;
```

Et enfin, l'algorithme de tri proprement dit :

N.B. : la fonction partition permet de construire à partir d'une liste l et d'un élément entier *pivot* les listes $linf$ et $lsup$ définies par :

$$linf = \{x \in l / x < pivot\} \quad \text{et} \quad lsup = \{x \in l / x > pivot\}$$

et retourne comme résultat le couple formé par ces deux listes.

```
fonction   partition  [ l : liste, pivot : entier] : liste # liste ==
  decl  n  :  entier
  decl  linf , lsup , cdr_l  :  liste
  selon  l  :
     nil            --> nil, nil
   | cons[n, cdr_l] --> soit linf,lsup==partition[cdr_l,pivot] dans
        ( si   n < pivot      alors cons[n,linf], lsup
          sinon si n > pivot  alors linf, cons[n,lsup]
          sinon                     linf, lsup
          fsi
          fsi)
        fsoit
  fselon  ;;
```

```
fonction   tri  [ l : liste ] : liste  ==
  %  Tri effectif\ldots  %
  decl  n  :  entier
  decl  linf , lsup  cdr_l  :  liste
  selon  l  :
     nil            --> nil
   | cons[n, cdr_l] --> soit linf, lsup == partition[cdr_l, n]
                        dans append[ tri[linf], cons [n, tri[lsup]] ]
                        fsoit
  fselon  ;;
```

Remarque : on retrouve bien les deux règles du tri (cf I.1) décrites au chapitre 1.

6.3 Arithmétique polynomiale

On reprend ici l'exemple décrit au I.3. On intègre la somme de deux polynômes comme constructeur du type polynôme. Cela revient en fait à dire que la somme de deux polynômes est un polynôme (loi interne).

N.B : Dans cette définition, le constructuer `zero` (cf page 2) a été supprimé : cette suppression est-elle valide ? Et si non, comment modifier le type ou les règles pour rendre le système confluent et nœthérien ?

```
TYPE  QX ==
 CONST
    mon  OF  entier # entier  % mon[n,d] represente le monome n.x^d %
  | plus OF QX # QX
 ENDCONST
 impression   l  ==
  DECL  n , d :  entier
  DECL  q , r   : QX
  CASE  p  :
    mon[n,d] --> impression[n] ^ 'X**' ^ impression[d]
   | plus [q, r] --> impression [ q ] ^ '+' ^ impression [ r ]
  ENDCASE  ;;
 RULES
   DECL  a1 , d1 , a2 , d2 :  entier
   DECL  p , q , r    : QX
   plus[p, mon[0,_]]  --> p    % p+0  -->  p %
  | plus[mon[0,_], p]  --> p    % 0+p  -->  p %
  | plus[mon[a1, d1], mon[a2, d2]] --> IF   d1=d2 THEN mon[a1+a2, d1]
        ELSE IF d1<d2  THEN plus[mon[a2, d2], mon[a1, d1]]
           ELSE plus[mon[a1, d1], mon[a2, d2]]
         ENDIF
        ENDIF
  | plus[mon[a1, d1], plus[mon[a2, d2],p]]  -->
        IF   d1=d2 THEN plus[mon[a1+a2, d1], p]
        ELSE IF d1<d2 THEN plus[mon[a2, d2],plus[mon[a1, d1],p]]
           ELSE plus[mon[a1, d1],plus[mon[a2, d2],p]]
         ENDIF
        ENDIF
  | plus[ plus[p, q], r]  --> plus[ p, plus[q, r]]  % Associativite %
 ENDRULES
```

6.4 Dérivation - Intégration

Nous avons vu que les expressions de filtrage s'écrivaient "comme" en mathématiques. On le retrouve bien en Lpac sur les deux règles de réécriture concernant la dérivation d'un produit et l'intégration par parties :

$(f.g)'$`-->`$f'.g + f.g'$ donne en Lpac :

```
derive [ mul[f,g]] -->  plus[ mul[derive[f], g],  mul[f, derive[g]]]
```

et $\int u'.v$ `-->` $[u.v] - \int u.v'$ (formule d'intégration par parties) s'écrit en Lpac - avec un petit changement de variables :

```
soit w == integre[u] dans
    integre[mul[u,v]] -->  moins[mul[w,v], integre[mul[w,derive[v]]]]
fsoit
```

Chapter 7

Utilisation de l'Analyseur et Implémentation du Compilateur

Vous devrez écrire deux commandes :

- `clpac toto` : permet d'appeler votre compilateur sur le fichier `toto.lpac`
- `execlpac toto` : permet d'executer le fichier objet `toto.ll` généré par la compilation de `toto.lpac`

L'analyseur synatxique vous est fourni : le but de ce chapitre est de préciser comment vous pouvez l'utiliser, ainsi que de définir l'architecture logicielle de votre projet.

7.1 Les fichiers fournis et l'interface avec vos fichiers

Le directory `/users/jlroch/lpac` contient les fichiers nécessaires à la réalisation du compilateur lpac, à savoir :

- 2 fichiers lelisp :
  - files.ll : commandes pour le chargement des fichiers du compilateur
  - interpreteur.ll : énoncé des fonctions à réaliser
- 3 fichiers compilés :
  - load-syntax.o : analyseur syntaxique pour le langage lpac (partie 1)
  - lpac.o : analyseur syntaxique pour le langage lpac (partie 2)

  - lpgo.o : chargement de l'ensemble des fichiers du compilateur

- 1 fichier lpac :
  - essai.lpac : exemple de fonction lpac qui calcule le pgcd de deux entiers

Organisation du travail

1. Dans votre directory principal, créez deux directories `Objet` et `Source`
2. Recopiez les 2 fichiers `files.ll` et `interpreteur.ll` dans votre directory `Source`

Tous vos fichiers lelisp pour le compilateur doivent avoir le suffixe `.ll`. Lors de la mise au point d'un fichier, travaillez dans votre directory principal. Lorsque vous estimez que le fichier est terminé, placez-le dans le directory `Source`, et compilez-le, en placant le fichier compilé correspondant dans le directory `Objet`.

Exemple : vous avez terminé le fichier: `toto.ll` qui permet la compilation de ...

1. vous le placez dans `Source` :
   `mv toto.ll Source`
2. vous le compilez (sous lelisp bien sûr) depuis le directory principal
   `(compilefiles "Source/toto.ll" "Objet/toto.o" )`
3. vous récupérez alors le fichier compilé: `toto.o` dans le directory `Objet`
4. vous le rajoutez dans la liste des fichiers automatiquement chargés par `lpgo` : pour cela, il suffit de rajouter dans votre fichier `files.ll` :
   `(loadfile (getfile "toto:) t)` (le rôle de getfile est d'aller chercher les fichiers, suivant le chemin : `.Objet Source /users/jlroch/lp` rajoutant les extensions `.o` et `.ll`. Autrement dit, si vous avez compilé le fichier `Source/toto.ll`, c'est le fichier `Objet/toto.o` qui sera chargé, à moins que `toto.ll` ne se trouve encore dans votre directory principal...)

7.2 Utilisation du compilateur

Pour charger le compilateur depuis votre directory principal, il suffit de faire :

```
$ lelisp
***  Le_Lisp  (by BULL-SEMS / INRIA)
```

```
= Systeme standard ...

? ; On charge les fichiers du compilateur deja \'ecrits
? (load "/users/jlroch/lpac/lpgo.o")
= /users/jlroch/lpac/lpgo.o
?
```

On peut maintenant faire deux types de manipulations :

- Appeler le compilateur sur un fichier par la fonction parse-file :

```
?  (parse-file "/users/jlroch/lpac/essai.lpac")
  ; permet de compiler le fichier/users/jlroch/lpac/essai.lpac
(expr-eval-decl
    (decl-fct
        "pgcd"
        (plan-liste-ident
            (liste-var (var-ident "u") "v")
            (type-expr-ident "entier"))
        (type-expr-ident "entier")
        (corps-de-fct
            (decl-loc-vide)
            (expr-si-al-sinon
                (expr-egal (expr-ident "v") (expr-int 0))
                (expr-ident "u")
                (expr-apply
                    (expr-ident "pgcd")
                    (expr-couple
                        (expr-ident "v")
                        (expr-rem (expr-ident "u") (expr-ident "v")))))))))
(expr-eval-eof)
= ()
?
? ;      On se retrouve alors sous lelisp
```

- Appeler le compilateur en permanence sur l'entree Unix standard :

```
? (lpac) ; On appelle l'interpreteur sur l'entree standard.
         ; On a la banniere :

******* Lpac version 1.2

lpac->  % On a alors le prompt "lpac-> " qui indique que l'on est         %
        % sous lpac.                                                      %
        % On peut alors taper des expressions lpac qui sont directement   %
        % analysees, et grace a vous prochainement compilees et executees %
```

```
lpac->  si x=y alors 31415926536
lpac->         sinon f[foo,bar]
lpac->  fsi;;
(expr-eval-expr
    (expr-si-al-sinon
        (expr-egal (expr-ident "x") (expr-ident "y"))
        (expr-int 31415930000.)
        (expr-apply
            (expr-ident "f")
            (expr-couple (expr-ident "foo") (expr-ident "bar")))))
lpac->
lpac->  % tant que vous n'aurez pas ecit une fonction quit en Lpac et que %
        % vous ne l'aurez pas compilee,la seule facon de sortir est ^\    %
lpac-> signal 3
OUPPS! J'ai failli faire un core
$   ; On se retrouve alors sous shell
```

7.3 Réalisation du compilateur : le fichier `interpreteur.ll`

Il s'agit de réaliser la fonction `traiter-arbre` qui est dans votre fichier `interpreteur.ll`. Cette fonction reçoit en entrée l'arbre abstrait d'une expression Lpac, et rend un résultat sans importance. L'effet de bord est la compilation de l'arbre (typage, gestion de l'environnement, évaluation, impression du résultat). Pour l'instant, cette fonction ne fait qu'imprimer l'arbre abstrait...

Remarque : On ne demande pas la réalisation de l'analyseur d'expressions algébriques (la fonction `analyse-expr-algebrique`). La fonction donnée ici ne fait que lire les caractères situés entre deux guillemets (" y compris) sur l'entrée standard.

7.4 Quelques exemples d'arbres abstraits

```
; Le-Lisp (by INRIA) version 15.21  (25/Dec/87)  [sun]
; Systeme standard modulaire : mar  4 oct  88 11:41:50
= (31bitfloats date microceyx debug pepe virbitmap defstruct virtty compiler
pretty abbrev loader)
? (load "../Objet/lpgo.o")
= ../Objet/lpgo.o
```

7.4.1 Les entiers naturels

```
(comline '#"cat type-N.lpac")
%=============================================================================%
%=============================================================================%

type N ==
  const
      zero
    | succ  de  N
  fconst
  impression n ==
     decl p : N
     selon n :
         zero    --> '0'
       | succ[p] --> 'succ(' ^ impression[p] ^ ')'
     fselon ;;


fonction  plus [ l, m : N ] : N ==
  decl n : N
  selon l :
      zero      --> m
    | succ[n]   --> succ[ plus[n,m] ]
  fselon ;;


fonction  mult [ l, m : N ] : N ==
  decl n : N
  selon l :
      zero      --> zero
    | succ[n]   --> plus[ mult[n,m], m  ]
  fselon ;;

%=============================================================================%

? (parse-file "type-N.lpac")
(expr-eval-decl
    (decl-type
        "N"
        (const-et
            (const-decl (type-decl-ident "zero"))
            (const-decl (type-decl-comp "succ" (type-expr-ident "N"))))
        (imp-fct "impression"
                 "n"
                 (corps-de-fct
```

```
                    (decl-loc-liste-var
                        (var-ident "p")
                        (type-expr-ident "N"))
                    (expr-app-pat
                        (expr-ident "n")
                        (appel-pat-et
                            (appel-pat-un
                                (pattern-ident "zero")
                                (expr-string "0"))
                            (appel-pat-un
                                (pattern-apply "succ" (pattern-ident "p"))
                                (expr-strg-concat
                                    (expr-strg-concat
                                        (expr-string "succ(")
                                        (expr-apply
                                          (expr-ident "impression")
                                          (expr-ident "p")))
                                    (expr-string ")")))))))
        (regles-vide)))
(expr-eval-decl
    (decl-fct
        "plus"
        (plan-liste-ident
            (liste-var (var-ident "l") "m")
            (type-expr-ident "N"))
        (type-expr-ident "N")
        (corps-de-fct
            (decl-loc-liste-var (var-ident "n") (type-expr-ident "N"))
            (expr-app-pat
                (expr-ident "l")
                (appel-pat-et
                    (appel-pat-un (pattern-ident "zero") (expr-ident "m"))
                    (appel-pat-un
                        (pattern-apply "succ" (pattern-ident "n"))
                        (expr-apply
                            (expr-ident "succ")
                            (expr-apply
                                (expr-ident "plus")
                                (expr-couple
                                    (expr-ident "n")
                                    (expr-ident "m")))))))))))
(expr-eval-decl
    (decl-fct
        "mult"
        (plan-liste-ident
            (liste-var (var-ident "l") "m")
            (type-expr-ident "N"))
```

```
        (type-expr-ident "N")
        (corps-de-fct
            (decl-loc-liste-var (var-ident "n") (type-expr-ident "N"))
            (expr-app-pat
                (expr-ident "l")
                (appel-pat-et
                    (appel-pat-un (pattern-ident "zero") (expr-ident "zero"))
                    (appel-pat-un
                        (pattern-apply "succ" (pattern-ident "n"))
                        (expr-apply
                            (expr-ident "plus")
                            (expr-couple
                                (expr-apply
                                    (expr-ident "mult")
                                    (expr-couple
                                        (expr-ident "n")
                                        (expr-ident "m")))
                                (expr-ident "m")))))))))
(expr-eval-eof)
%=============================================================================%
```

7.4.2 Les polynômes à une indéterminée

```
%=============================================================================%
? (comline '#"cat type-QX.lpac")
%=============================================================================%
%=============================================================================%
TYPE QX ==
  CONST
      mon  OF  entier # entier
    | plus OF  QX # QX
  ENDCONST
  impression l ==
    DECL n,d : entier
    DECL q,r : QX
    CASE p :
        mon [n,d]  -->   impression[n] ^ 'X**' ^ impression[d]
      | plus [q,r] -->   impression[q] ^ '+' ^ impression[r]
    ENDCASE
  RULES
      DECL  a1, d1, a2, d2  : entier
      DECL  p, q, r  : QX
      plus[p, mon[0,_]]                    --> p
    | plus[mon[0,_], p]                    --> p
    | plus[mon[a1,d1], mon[a2,d2]]         -->
```

```
                    IF      d1=d2 THEN mon[a1+a2,d1]
                    ELSE IF d1<d2 THEN plus[mon[a2,d2], mon[a1,d1]]
                                  ELSE plus[mon[a1,d1], mon[a2,d2]]
                         ENDIF
                    ENDIF
  | plus[mon[a1,d1], plus[mon[a2,d2],p]] -->
                    IF      d1=d2 THEN plus[mon[a1+a2,d1], p]
                    ELSE IF d1<d2 THEN plus[mon[a2,d2], plus[mon[a1,d1],p]]
                         ELSE plus[mon[a1,d1], plus[mon[a2,d2],p]]
                         ENDIF
                    ENDIF
  | plus[ plus[p,q], r]                  -->  plus [ p, plus[q,r]]
ENDRULES;;
%=============================================================================%

? (parse-file "type-QX.lpac")
(expr-eval-decl
    (decl-type
        "QX"
        (const-et
            (const-decl
                (type-decl-comp
                    "mon"
                    (type-expr-prod
                        (type-expr-ident "entier")
                        (type-expr-ident "entier"))))
            (const-decl
                (type-decl-comp
                    "plus"
                    (type-expr-prod
                        (type-expr-ident "QX")
                        (type-expr-ident "QX")))))
        (imp-fct "impression"
                 "l"
                 (corps-de-fct
                     (decl-loc-et
                         (decl-loc-liste-var
                             (liste-var (var-ident "n") "d")
                             (type-expr-ident "entier"))
                         (decl-loc-liste-var
                             (liste-var (var-ident "q") "r")
                             (type-expr-ident "QX")))
                     (expr-app-pat
                         (expr-ident "p")
                         (appel-pat-et
                             (appel-pat-un
                                 (pattern-apply
```

```
                                    "mon"
                                    (pattern-couple
                                        (pattern-ident "n")
                                        (pattern-ident "d")))
                                (expr-strg-concat
                                    (expr-strg-concat
                                        (expr-apply
                                          (expr-ident "impression")
                                          (expr-ident "n"))
                                        (expr-string "X**"))
                                    (expr-apply
                                        (expr-ident "impression")
                                        (expr-ident "d"))))
                            (appel-pat-un
                                (pattern-apply
                                    "plus"
                                    (pattern-couple
                                        (pattern-ident "q")
                                        (pattern-ident "r")))
                                (expr-strg-concat
                                    (expr-strg-concat
                                        (expr-apply
                                          (expr-ident "impression")
                                          (expr-ident "q"))
                                        (expr-string "+"))
                                    (expr-apply
                                        (expr-ident "impression")
                                        (expr-ident "r"))))))))
    (regles-decl
        (decl-loc-et
            (decl-loc-liste-var
                (liste-var
                    (liste-var (liste-var (var-ident "a1") "d1") "a2")
                    "d2")
                (type-expr-ident "entier"))
            (decl-loc-liste-var
                (liste-var (liste-var (var-ident "p") "q") "r")
                (type-expr-ident "QX")))
        (regles-et
            (regles-et
                (regles-et
                    (regles-et
                        (regles-pat
                            (pattern-apply
                                "plus"
                                (pattern-couple
                                    (pattern-ident "p")
```

```
                        (pattern-apply
                          "mon"
                          (pattern-couple
                            (pattern-int 0)
                            (pattern-joker)))))
                    (expr-ident "p"))
                (regles-pat
                    (pattern-apply
                        "plus"
                        (pattern-couple
                            (pattern-apply
                              "mon"
                              (pattern-couple
                                (pattern-int 0)
                                (pattern-joker)))
                            (pattern-ident "p")))
                    (expr-ident "p")))
            (regles-pat
                (pattern-apply
                    "plus"
                    (pattern-couple
                        (pattern-apply
                            "mon"
                            (pattern-couple
                              (pattern-ident "a1")
                              (pattern-ident "d1")))
                        (pattern-apply
                            "mon"
                            (pattern-couple
                              (pattern-ident "a2")
                              (pattern-ident "d2")))))
                (expr-si-al-sinon
                    (expr-egal
                        (expr-ident "d1")
                        (expr-ident "d2"))
                    (expr-apply
                        (expr-ident "mon")
                        (expr-couple
                            (expr-plus
                              (expr-ident "a1")
                              (expr-ident "a2"))
                            (expr-ident "d1")))
                    (expr-si-al-sinon
                        (expr-inf
                            (expr-ident "d1")
                            (expr-ident "d2"))
                        (expr-apply
```

```
                                (expr-ident "plus")
                                (expr-couple
                                  (expr-apply
                                    (expr-ident "mon")
                                    (expr-couple
                                      (expr-ident "a2")
                                      (expr-ident "d2")))
                                  (expr-apply
                                    (expr-ident "mon")
                                    (expr-couple
                                      (expr-ident "a1")
                                      (expr-ident "d1")))))
                            (expr-apply
                                (expr-ident "plus")
                                (expr-couple
                                  (expr-apply
                                    (expr-ident "mon")
                                    (expr-couple
                                      (expr-ident "a1")
                                      (expr-ident "d1")))
                                  (expr-apply
                                    (expr-ident "mon")
                                    (expr-couple
                                      (expr-ident "a2")
                                      (expr-ident "d2")))))))))
(regles-pat
    (pattern-apply
        "plus"
        (pattern-couple
            (pattern-apply
                "mon"
                (pattern-couple
                    (pattern-ident "a1")
                    (pattern-ident "d1")))
            (pattern-apply
                "plus"
                (pattern-couple
                    (pattern-apply
                      "mon"
                      (pattern-couple
                        (pattern-ident "a2")
                        (pattern-ident "d2")))
                    (pattern-ident "p")))))
    (expr-si-al-sinon
        (expr-egal (expr-ident "d1") (expr-ident "d2"))
        (expr-apply
            (expr-ident "plus")
```

```
                    (expr-couple
                        (expr-apply
                            (expr-ident "mon")
                            (expr-couple
                              (expr-plus
                                (expr-ident "a1")
                                (expr-ident "a2"))
                              (expr-ident "d1")))
                        (expr-ident "p")))
                (expr-si-al-sinon
                    (expr-inf (expr-ident "d1") (expr-ident "d2"))
                    (expr-apply
                        (expr-ident "plus")
                        (expr-couple
                            (expr-apply
                              (expr-ident "mon")
                              (expr-couple
                                (expr-ident "a2")
                                (expr-ident "d2")))
                            (expr-apply
                              (expr-ident "plus")
                              (expr-couple
                                (expr-apply
                                  (expr-ident "mon")
                                  (expr-couple
                                    (expr-ident "a1")
                                    (expr-ident "d1")))
                                (expr-ident "p")))))
                    (expr-apply
                        (expr-ident "plus")
                        (expr-couple
                            (expr-apply
                              (expr-ident "mon")
                              (expr-couple
                                (expr-ident "a1")
                                (expr-ident "d1")))
                            (expr-apply
                              (expr-ident "plus")
                              (expr-couple
                                (expr-apply
                                  (expr-ident "mon")
                                  (expr-couple
                                    (expr-ident "a2")
                                    (expr-ident "d2")))
                                (expr-ident "p")))))))))
        (regles-pat
            (pattern-apply
```

```
                        "plus"
                        (pattern-couple
                            (pattern-apply
                                "plus"
                                (pattern-couple
                                    (pattern-ident "p")
                                    (pattern-ident "q")))
                            (pattern-ident "r")))
                    (expr-apply
                        (expr-ident "plus")
                        (expr-couple
                            (expr-ident "p")
                            (expr-apply
                                (expr-ident "plus")
                                (expr-couple
                                    (expr-ident "q")
                                    (expr-ident "r")))))))))))
(expr-eval-eof)
```