Programmation Logique et Par Contraintes Avancée
Cours 1 – Introduction

Ralf Treinen

Université Paris Cité
UFR Informatique
Institut de Recherche en Informatique Fondamentale

treinen@irif.fr

10 janvier 2024

Organisation 2023/2024

- 11 cours
- On commence par 1.5h de cours (Sophie-Germain 1001), puis après une petite pause on passe à la partie TP (Sophie-Germain 2032 ?)
- Copies des transparents, et exemples de code, aussi disponible sur `https://www.irif.fr/~treinen/teaching/plpc/`
- Modalités de contrôle de connaissances :

$$50\%\text{examen} + 50\%\text{CC}$$

Plan du module

- Oz : un langage multi-paradigme [2 semaines]
- Programmation concurrente par flot de données [1 semaine]
- (optionnel : Programmation logique en Oz [1 semaine])
- Programmation par contraintes : techniques avancées pour la résolution de problèmes combinatoires [7 semaines]

Pre-requis du module

- Programmation fonctionnelle (par exemple OCaml, Haskell, Scala, Lisp, Python, ...)
  Style de programmation prévalent en Oz.
  On utilisera par exemple le pattern matching, des fonctions d'ordre supérieur comme `map`.
- Notions de base de la logique du premier ordre.
  Le modèle de la mémoire en Oz est basé sur la notion de *contraintes*.
- Il n'est pas nécessaire de connaitre le langage Prolog.
  Nous en parlons un peu au premier cours, mais c'est seulement pour motiver le besoin d'une approche différente.

Contenu chapitre 1

Prolog et la Programmation Logique

Premier pas en Oz

Un petit rappel : Prolog

- Abréviation de *programmation en logique*
- Développé au début des années 70 indépendamment par les équipes d'Alain Colmerauer (Marseille) et Robert Kowalski (Edinbourgh).
- Paradigme de la programmation *déclarative* : On utilise une logique pour *déclarer* la sémantique souhaitée du programme, puis le compilateur va se débrouiller pour trouver un moyen de l'exécuter efficacement (c'était au moins l'idée).
- Un programme Prolog définit des *prédicats*.
- Données : termes (éventuellement avec des variables), entiers.

Exemple d'un programme Prolog

Le prédicat $append(X, Y, Z)$ doit être vraie ssi Z est la concaténation des deux listes X et Y.

```
append([],Y,Y).
append([H|X],Y,[H|Z]) :- append(X,Y,Z)
```

À lire comme :

$$\begin{aligned} \forall Y : \quad & append([], Y, Y) \\ \forall H, X, Y, Z : \quad & append(X, Y, Z) \Rightarrow append([H|X], Y, [H|Z]) \end{aligned}$$

Ingrédients de la sémantique opérationnelle

- *Unification* (vient de la logique du premier ordre) : elle explique comment résoudre des équations entre termes.
- *Résolution* (également de la logique du premier ordre, mais ici on a seulement besoin d'un cas particulier) : elle explique comment appliquer la définition d'un prédicat, en utilisant l'unification.
- *Arbre de recherche et retour en arrière* (angl. : *backtracking*) : technique de programmation classique pour les problèmes combinatoires. Les interpréteurs Prolog utilisent une implémentation très astucieuse (Warren Abstract Machine).

Unification

- Inventée par Jacques Herbrand, réinventée par John Alan Robinson (celui avec la résolution).
- Permet de résoudre un système d'équations entre termes symboliques.
- Exemple : $f(x, a) = f(b, y)$:
  Solution $x = b, y = a$.
- Exemple : $f(x, a) = f(y, b)$:
  pas de solution !

Résolution

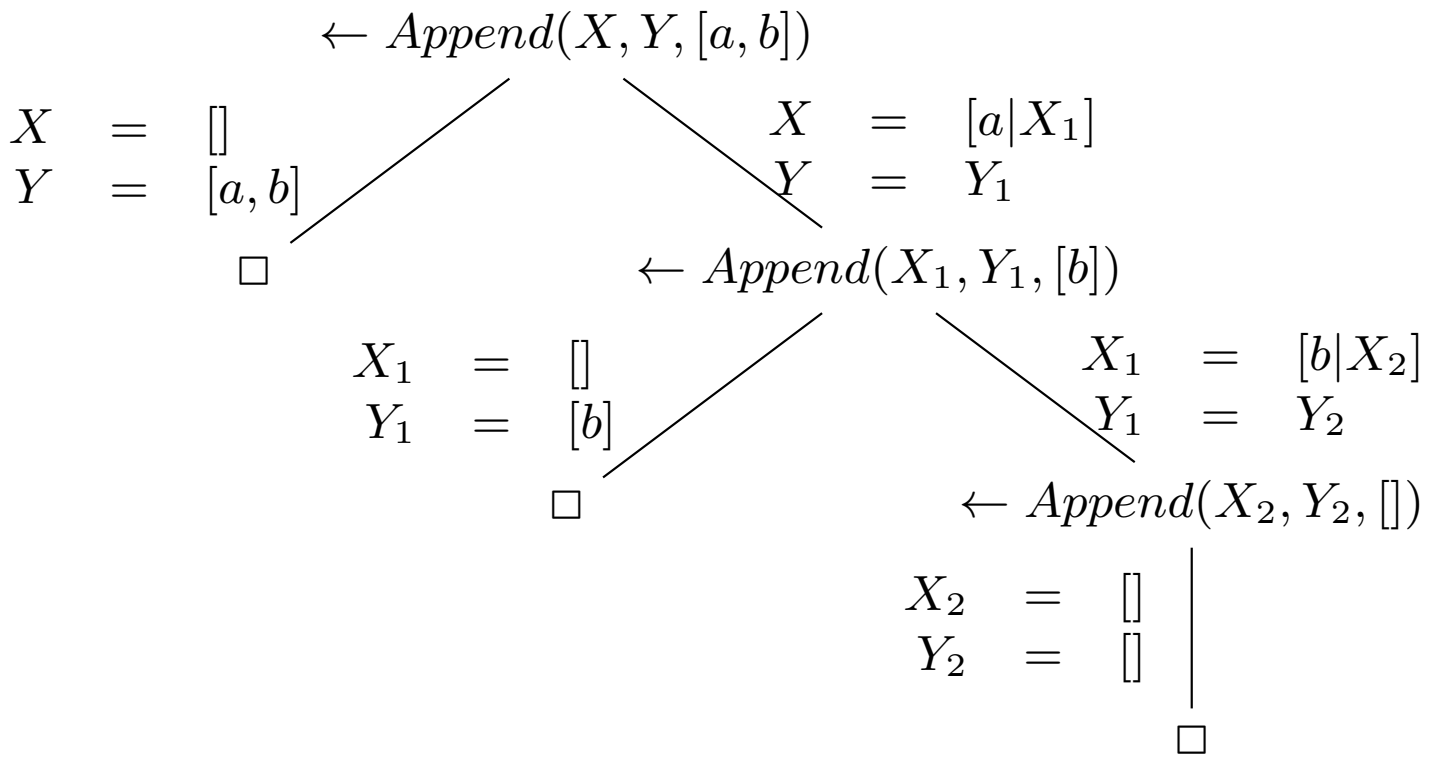
- Inventée par J. Alan Robinson 1965 comme procédure de recherche de preuve dans la logique du premier ordre.
- Prolog : cas particulier car toutes les clauses sont Horn (les clauses du programmes, ainsi que la requête).
- *Clause Horn* : clause qui contient au plus un atome positif.
- *Requête* : les termes qu'on cherche à « évaluer » par rapport au programme (plus exactement : on cherche des valeurs de leurs variables qui constituent une solution).

L'arbre de recherche

- Étant donné le premier atome de la requête il y a en général plusieurs clauses du programme avec lesquels on peut résoudre.
- Il faut essayer toutes les possibilités pour ne pas perdre une solution potentielle.
- Pour chaque étape de résolution on peut avoir plusieurs alternatives, qui forment alors les arêtes dans un *arbre de recherche*.
- L'évaluation d'un programme Prolog consiste en un parcours de cet arbre de recherche qui est construit à la volée par l'interpréteur.

Exemple : le prédicat append

```
Append([],Y,Y).
Append([H|X],Y,[H|Z]) :- Append(X,Y,Z)
```



Une critique de Prolog (1)

- On a seulement droit à des clauses, tout calcul se fait par résolution et construction d'un arbre de recherche.
- Or, dans un programme il y a normalement beaucoup de calcul déterministe qui ne nécessite pas de recherche.
- ☺ Il y a des techniques de compilation pour exécuter très efficacement un programme Prolog quand le calcul est déterministe (WAM).
- ☹ Il est quand même très pénible de programmer tout avec des prédicats et avec des clauses !

Une critique de Prolog (2)

- Pas de types ! !
- ☺ Cela permet en Prolog quelques astuces, comme par exemple confondre des termes avec des atomes, écrire un interpréteur Prolog en quelques lignes de Prolog.
- ☹ Dans l'absence d'un système de typage (statique, c.-à-d. avant l'exécution du programme) il est beaucoup plus difficile de détecter les erreurs de programmation (voir Python vs. OCaml !).

Une critique de Prolog (3)

- Prolog ne fournit aucun moyen de *structuration* du programme en unités encapsulées :
  - des modules (sauf extensions dans certains compilateurs)
  - des objets
- ☹ En Prolog on est constamment obligé de violer tous les bonnes principes du Génie Logiciel qui préconisent une structuration en modules, en classes, en types abstraits, etc.

Une critique de Prolog (4)

- La programmation déclarative est un idéal très noble ...
- ... dont (la pratique de) Prolog est très loin :
  - On ne peut pas éviter l'algorithmique, même pas en Prolog
  - cut et la « négation » de Prolog.
  - Prédicats « méta-logiques » qui permettent une introspection de termes (comme le prédicat *var* qui teste si un terme est une variable).

Extensions de Prolog

- ☹ Prolog « classique » ne connaît que les termes et l'unification comme mécanisme de résolution de contraintes.
- ☺ Les Prolog modernes (Prolog III, GNU Prolog, Yap, ...) intègrent aussi des autres systèmes de contraintes (équations linéaires, domaines finis, ...)
- ☹Toute fois cela n'est pas suffisant, il manque :
  - La possibilité de définir de nouveaux systèmes de contraintes
  - La possibilité de définir sa propre stratégie de recherche

Alors, quoi faire avec la programmation logique ?

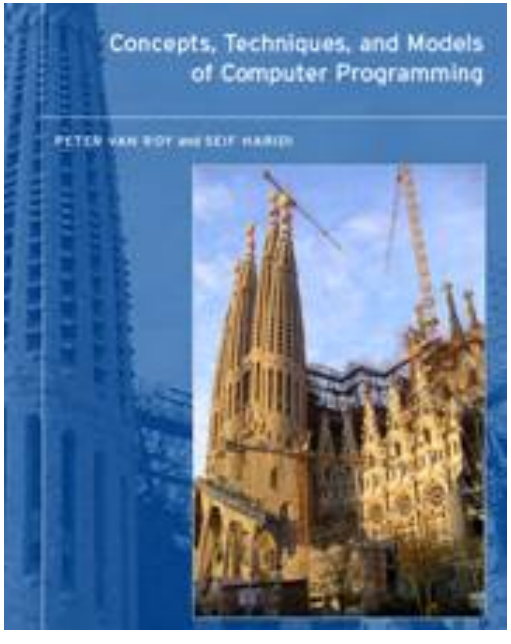
- La programmation par recherche est très utile pour certaines applications : optimisation, planification, ordonnancement, ...
- Prolog est un langage *minimaliste* pour la programmation par recherche qui manque des constructions qui existent dans d'autres langages de programmation.
- Il faut intégrer la programmation par recherche avec des autres concepts utils de langages de programmation :
  - soit par un langage multi-paradigme (Oz, Alice)
  - soit par une bibliothèque pour la fonctionnalité de recherche (GeCode, ILOG solver library)

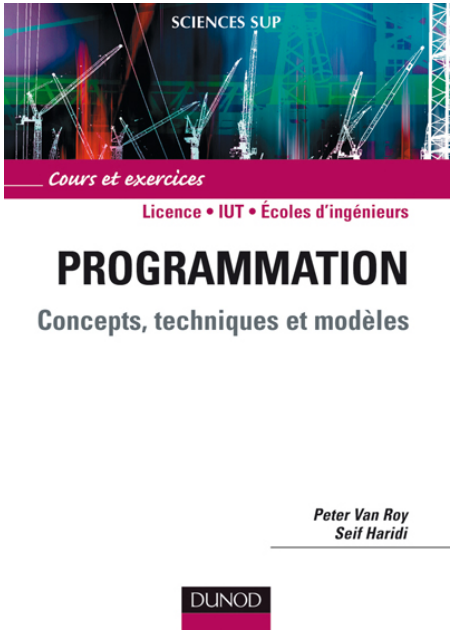
Oz et Mozart

- *Oz* est le nom du langage de programmation
- *Mozart* est le nom de l'implémentation du langage
- ☺ Mozart est un logiciel libre !
- Né en 1991 (Gert Smolka et. al.)
- Contributions essentielles de l'université de Saarbrücken (Allemagne), SICS (Suède), UC de Louvain (Belgique).
- Développement aujourd'hui piloté par un groupe international.

Le livre

- *Concepts, Techniques, and Models of Computer Programming*, par Peter Van Roy et Seif Haridi.
- La version française est abrégée et pas très utile pour ce cours, elle ne parle pas des contraintes !





Documention sur Mozart/Oz disponibles en ligne

- `http://mozart.github.io/mozart-v1/doc-1.4.0/` en particulier le *Tutorial* et *The Oz Base Environment*.
- Il y a les mêmes documents sur les PC de l'UFR à l'adresse `/usr/local/doc/mozart`
- Nous utilisons la version 1.4. Il y a une pre-release de la version 2 qui pour l'instant ne contient pas les contraintes.

Oz

Oz est un langage multi-paradigme : il permet la

- programmation fonctionnelle
- programmation impérative
- programmation par objets
- programmation logique et par contraintes
- programmation concurrente *dataflow*
- programmation distribuée
- programmation des interfaces graphiques
- ...

Oz

- La syntaxe de Oz est inspirée par LISP :
  L'application d'une fonction (procédure, méthode) s'écrit

$$\{func \quad arg_1 \quad \ldots \quad arg_n\}$$

- La syntaxe permet des expressions composées, comme

$$\{f \quad a_1 \quad \{g \quad a_2 \quad a_3\}\}$$

- Typage *dynamique* (malheureusement) : des erreurs de typage sont détectées seulement pendant l'exécution, comme pour Python par exemple.
- Il n'y a pas d'application partielle comme en OCaml.

L'environnement de programmation

- Basé sur GNU Emacs
- Lancez l'environnement par la commande

  `oz <fichier>.oz`

  cela ouvre une session d'emacs avec une fenêtre d'édition (Oz) et une fenêtre (`*Oz Compiler*`) où le compilateur affiche ses messages.
- Quand on exécute un programme Oz, les affichages paraissent dans une autre fenêtre `*Oz Emulator*`.
- Utiliser la fonction *Show/Hide -> Emulator* du menu Oz pour basculer.

L'environnement de programmation

- Utiliser le menu Oz d'Emacs quand il est dans le mode oz
- Utiliser les raccourcies clavier
  (C-x : appuyez les touches `Ctrl` et `x` au même moment) :
  - `C-. C-l` exécuter la ligne courante ;
  - `C-. C-r` exécuter la région courante (délimité avec la souris, ou `C-<espace>`) ;
  - `C-. C-p` exécuter le paragraphe courant (délimité par des lignes vides) ;
  - `C-. C-b` exécuter le tampon (buffer) courant.
  - `C-. c` montrer la sortie du compilateur.
  - `C-. e` montrer la sortie de l'émulateur.

Exemples (windows.oz)

```
% prints in the emulator window
{Show 'Hello, World!'}

% prints in the browser window
{Browse 'Hello, World!'}
```

Déclaration de variables

- Les noms des variables (et des fonctions, procédures) commencent toujours sur des lettres en MAJUSCULES.
- `local X Y Z in statement end`
  déclare les variables X, Y, Z, leur portée est *statement*.
- `declare X Y Z in statement`
  déclaration « ouverte » : la portée de X, Y, Z est globale, leur portée n'est pas limitée (comme un `let x = ...;;` en OCaml).

Exemples (local.oz)

```
% local definition
local X in
   X = 3
   {Browse X}
end

% X is not known here !
{Browse X}
```

Exemples (global.oz)

```
% global (open-ended) definition
declare X
X=4
{Browse X}

% Oz is a SINGLE ASSIGNMENT LANGUAGE
X = 5 % unification error
```

Variables et types

- Affectation d'une valeur à une variable : `VARIABLE = valeur`
- Variables sont à affectation *unique* : on peut leur donner une fois une valeur (comme `let x = ... in ...` en OCaml), mais cette valeur peut être un terme composé qui contient des variables (comme dans la programmation logique).
- Typage *dynamique* (similaire à Python) : Les *valeurs* (pas les variables !) portent des types, l'application d'une opération à des valeurs du mauvais type déclenche une erreur.

Exemples (typing.oz)

```
declare A B
A = 5
B = 'Tinman'
{Browse A + B}
```

Les types

- Il y a des types primaires et de types secondaires.
- Types primaires : Leur union couvre tout l'univers de valeurs.
- Deux types primaires sont soit sous-type un de l'autre, soit ils sont disjoints.
- Les types primaires les plus importants (pour l'instant) :

  `Number` avec des sous-types `Int` et `Float`
  `Record` avec le sous-type `Tuple`
  `Procedure` (les procédures sont des valeurs de première classe)

- Ou sont les fonctions ? Voir la semaine prochaine !

Valeurs numériques

- Une valeur numérique peut être entière (`Int`) ou flottante (`Float`). Les caractères (`Char`) sont un sous-type du type `Int`.
- Le moins unaire s'écrit $\sim$.
- Les flottants doivent être notés avec un point décimal :

$$\sim 3.141, \quad 4.5E3, \quad \sim 12.0e \sim 2$$

- Pas de conversion automatique entre Int et Float.

Littéraux

- sont des « mots » atomiques. Il y en a deux sortes, seulement la première est importe pour nous :
  - *atomes* : séquence de caractères alphanumériques, commencant sur une minuscule, ou une chaine arbitraire entre apostrophes `'`
- Exemple d'atomes :

```
a    foo    '='    ':='    'OZ 3.0'    'Hello World'
```

- pas à confondre avec les String, qui sont en fait des listes de caractères.

Exemples (literals.oz)

```
% literals
local X Y B in
   X = foo    % apostrophes pas nécessaires
   Y = '=='   % apostrophes nécessaires
   B = true   % les booléens sont des atomes
   {Browse [X Y B]}
end
```

Listes, Tuples et Enregistrement

- Le type de *listes* est un sous-type de *tuples*
- Le type de *tuples* est un sous-type de *enregistrement*

Tuples

- Un *tuple* consiste en une *étiquette* (angl. : *label*), et un certain nombre d'arguments.
- Similaire à un élément d'un type algébrique on OCaml, sauf qu'il n'y a pas de déclaration de type on Oz.
- L'étiquette est un atome (utiliser des apostrophes quand l'atome ne consiste pas en des alphanumériques seulement)
- Exemple :

```
f(a b c)
f(a bar(b c) foo(d e f) f(42))
```

- fonction `Label` pour extraire l'étiquette d'un tuple
- fonction `Width` pour obténir le nombre d'arguments
- accéder à des argument avec `.1`, `.2`, etc

Exemples (tuples.oz)

```
declare X in

X = foo(a bar(b c))

{Show X}

{Show {Label X}}

{Show {Width X}}

{Show X.1}

{Show X.7}
```

Filtrage par motif

- Une instruction peut aussi être un filtrage par motif :

```
case E
  of Pattern_1 then S1
  [] Pattern_2 then S2
  [] ...
  else S end
```

- La partie **else** est optionnelle.
- Les variables dans le motif sont des variables locales, sauf si précédée par `!`.

Exemples (case.oz)

```
declare X in

X = blabla(17 abc foobar)

{Show X}

case X
of coocoo(Y Z) then
   {Show 'we have coocoo'}
[] blabla(Y1 Y2 Y3) then
   {Show 'we have blabla'}
   {Show Y1} {Show Y2} {Show Y3}
end
```

Listes

- Les listes sont un type secondaire (sous-type des tuples).
- Deux syntaxes équivalentes :

```
1|2|3|nil
[1 2 3]
```

- Ceci n'est qu'une abréviation pour des tuples imbriqués, donc une autre façon d'écrire la même liste est

```
'|'(1 '|'(2 '|' (3 nil)))
```

- Puisque une liste est un tuple, on accède à la tête par `.1` et au reste de la liste par `.2`

Exemples (liste.oz)

```
declare L in

L = [a b c d e]

{Show L.1}

{Show L.2}
```

Chaînes de caractères

- Abréviation pour des *listes* d'entiers (les codes entières des charactères).
- Les trois valeurs suivantes sont égales :

```
"OZ␣3.0"
[&O &Z &  &3 &. &0]
[79 90 32 51 46 48]
```

Enregistrements (angl. : *Records*)

- Contrairement à OCaml il n'y a pas de déclaration de type.
- Un record consiste en une étiquette (*label*), et une séquence de paires d'une clefs (appelés *feature*) et d'une valeur.
- On peut utiliser les mêmes labels avec des features différents, et inversement (un peu dans l'esprit des variants polymorphes de OCaml)
- Examples :

```
f(a: av b: bv c: 17 d: 42)
f(first: abcd second: g(primo:17 secundo: 42))
```

Exemples (records1.oz)

```
declare T
T = f(a: av b: bv c: 17 d: 42)
{Browse T}

% traditional terms are a special case of records
declare W
W = f(2:a 1:b)
{Browse W}

% the order of children does not matter
{Browse f(a:1 b:2) == f(b:2 a:1)}
```

Tuples et Enregistrements

- Un tuple est simplement un enregistrement avec des clefs numériques $1, 2, \ldots$
- Ainsi

```
f(a b c d)
```

est la même chose que

```
f(1:a 2:b 3:c 4:d)
```

Opérations sur des enregistrements

- Sélectionner la valeur de x avec la clef c : `x.c`
- Arité de X (liste des clefs de X) : `{Arity X}`
- Label de X : `{Label X}`
- Ajouter une paire à un enregistrement : `{AdjoinAt R F X}` donne R avec en plus la paire (F,X). Quand $R1$ a déjà la clef F alors le résultat est R, sauf que la valeur à la clef F est X.
- Il y a des variantes (joindre deux enregistrements, joindre une liste de paires à un enregistrement, ...). Voir la doc.
- Toutes ses opérations s'appliquent également aux tuples (qui ne sont qu'un cas particulier des enregistrements !)

Exemples (records2.oz)

```
% operations on records
{Browse T.b}

{Browse W.1}

{Browse {Arity T}}
{Browse {Arity W}}

{Browse {Label T}}

{Browse {AdjoinAt T new 42}}
```

Exemples (records3.oz)

```
% unification
declare X Y Z in
f(X 1)=f(2 Y)
{Browse [X Y]}

% trees may be infinite
local X in
   X=f(X)
   {Browse X}
end
```

Déclaration de procédures

```
local Max X Y Z in
  proc {Max A B R}
    if A >= B then R = A else R = B  end
  end
  X = 5
  Y = 10
  {Max X Y Z} {Browse Z}
end
```

Procédures

- **proc** `{P X1 ... Xn}` ... **end**

  est une instruction qui lie l'identificateur P à une procédure.
- Il faut donc *déclarer* l'identificateur P.
- Définition de procédures : liaison statique (angl. *lexical scoping*).
- Les procédures sont des valeurs de première classe, on peut donc les passer comme argument à une autre procédure.

Exemples (proc1.oz)

```
declare X P
in
   proc {P Y} {Browse Y+Y} end
   X = 5
   {P X}
```

Exemples (proc1-static.oz)

```
declare X in
X = 1
declare P in
proc {P Y} {Browse X+Y} end
declare X in
X = 2
{P 3}
```

Exemples (proc2.oz)

```
% procedure and pattern matching
declare Length in
proc {Length L Result}
   case L
   of nil then Result=0
   [] H|R then local Restlength in
                  {Length R Restlength}
                  Result=Restlength+1
               end
   end
end

local R in
   {Length [1 2 3 4 5] R}
   {Browse R}
end
```

Exemples (proc3.oz) I

```
% binary search trees
declare Insert in
proc {Insert Key Value TreeIn TreeOut}
   case TreeIn
   of nil then TreeOut = tree(Key Value nil nil)
   [] tree(K1 V1 T1 T2) then
      if Key == K1 then TreeOut = tree(Key Value T1 T2)
      elseif Key < K1 then
         local T in
            TreeOut = tree(K1 V1 T T2)
            {Insert Key Value T1 T}
         end
      else
         local T in
            TreeOut = tree(K1 V1 T1 T)
            {Insert Key Value T2 T}
         end
```

Exemples (proc3.oz) II

```
      end
   end
end

local X1 X2 X3
in
   {Insert toto 5 nil X1}
   {Insert titi 17 X1 X2}
   {Insert coocoo 42 X2 X3}
   {Browse X3}
end
```

Quelques pièges de la syntaxe

- Identificateurs (aussi noms de procédures, fonctions) commencent par une majuscules
- Atomes (valeurs symboliques) commencent par une minuscule
- Applications des procédures et fonctions entre accolades { et }
- Élements d'une séquence séparés par des espaces
- Moins unaire : $\sim$
- Séparer les cas d'une distinction de cas par `[]`
- La liste vide s'écrit `nil`