

# Résumé de LINFO1104

compilation du 25 février 2023

Thomas Debelle

Juin 2023

# Table des matières

|          |  |          |
|----------|--|----------|
| <b>1</b> | <b>Introduction</b>                      | <b>3</b> |
| 1.1      | Les Paradigmes . . . . .                 | 3        |
| <b>2</b> | <b>Les différents Paradigmes</b>         | <b>4</b> |
| 2.1      | Functional Programming . . . . .         | 4        |
| 2.2      | Conseils pour la syntaxe d'Oz . . . . .  | 5        |
| <b>3</b> | <b>Programmation symbolique</b>          | <b>6</b> |
| 3.1      | Listes . . . . .                         | 6        |
| 3.1.1    | Définition formelle . . . . .            | 6        |
| 3.2      | Pattern matching . . . . .               | 6        |
| 3.3      | Introduction au langage Kernel . . . . . | 7        |
| 3.4      | Les arbres . . . . .                     | 7        |
| 3.4.1    | Ordered Binary tree . . . . .            | 7        |
| 3.5      | Tuples et Records . . . . .              | 8        |
| 3.5.1    | Tuples . . . . .                         | 8        |
| 3.5.2    | Similitude Tuples et liste . . . . .     | 8        |
| 3.5.3    | Les Records . . . . .                    | 9        |
| 3.5.4    | Résumé . . . . .                         | 9        |
| 3.6      | Sémantique Formelle . . . . .            | 9        |
| 3.6.1    | Les environnements . . . . .             | 9        |

# Préface

Bonjour à toi !

Cette synthèse recueille toutes les informations importantes données au cours, pendant les séances de tp et est amélioré grâce au note du Syllabus. Elle ne remplace pas le cours donc écoutez bien les conseils et potentielles astuces que les professeurs peuvent vous donner. Notre synthèse est plus une aide qui on l'espère vous sera à toutes et tous utiles.

Elle a été réalisée par toutes les personnes que tu vois mentionné. Si jamais cette synthèse a une faute, manque de précision, typo ou n'est pas à jour par rapport à la matière actuelle ou bien que tu veux simplement contribuer en y apportant ta connaissance ? Rien de plus simple ! Améliore la en te rendant [ici](#) où tu trouveras toutes les infos pour mettre ce document à jour. *(en plus tu auras ton nom en gros ici et sur la page du github)*

Nous espérons que cette synthèse te sera utile d'une quelconque manière ! Bonne lecture et bonne étude.

# Chapitre 1

## Introduction

### 1.1 Les Paradigmes

Une paradigme, est une façon d'approcher et apporter une solution à un problème. De ce fait, chaque langage de programmation utilise 1 voir 2 paradigmes. Ce cours couvrat 5 paradigmes cruciaux qui sont :

1. "Functionnal Programming"
2. "Object Oriented Programming"
3. "Functional DataFlow Programming"
4. "Actor DataFlow Programming or Multi-Agent"
5. "Active Objects"

Et pour découvrir ces paradigmes, nous utiliserons les langages de programmations "Oz" qui est un langage de recherche multi paradigme ainsi que "Erlang".

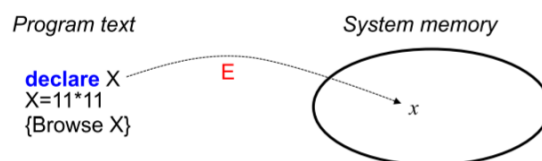
## Chapitre 2

# Les différents Paradigmes

Comme mentionner plus haut, on rencontrera 5 paradigmes dont voici le premier.

### 2.1 Functional Programming

Avec ce paradigme, on impose qu'une variable peut être nommée qu'une seule fois ! Donc :  $X = 10$  mais on ne peut pas plus loin dire  $X = 9$ .  $X$  est déjà attribué. On peut penser que cela risque d'être handicapant alors qu'en réalité, cela rend notre code plus simple à déboguer. De plus, nombreux sont les langages et microservices utilisés qui implémentent la programmation fonctionnelle. Formellement, quand on déclare une variable et qu'on l'assigne à une valeur ceci se passe. Une chose importante à noter est que cette façon de programmer peut être réalisée dans n'importe quel langage de programmation. On peut également redéclarer un identificateur. C'est-à-dire écrire " $X = 42$ " et plus loin en ayant redéclaré une variable " $X = 11$ " car ces deux déclarations pointent à deux éléments totalement différents dans la mémoire.



Un "Scope" ou portée est une propriété centrale en programmation. En effet, c'est le scope qui nous permet d'avoir différentes valeurs pour des variables qui ont le même nom. Naturellement, elle ne représente pas la même chose car elle diffère de leur scope. On peut déterminer le scope d'une variable sans même exécuter le code. Il nous suffit d'analyser le code qui comprend un "**lexical scoping**" ou un "**static scoping**".

FIGURE 2.1 – Déclaration d'une variable

```
local
  X
in
  X = 42 {Browse X}
local
  X
in
  X = 11 {Browse X}
end
{Browse X}
end
```

FIGURE 2.2 – Exemple de code avec des scopes différents

## 2.2 Conseils pour la syntaxe d'Oz

# Programmation symbolique

On dit d'une liste est **récursive** si elle se définit par elle-même. C'est-à-dire elle fait appel à elle-même. On utilise la récursion pour les calculs et pour stocker des données. Une liste est soit vide ou soit une pair *d'une valeur suivi par une autre liste*. En **OZ** les variables, procédures et fonctions **doivent** commencer par une majuscule!

En utilisant la notation **Extended Backus-Naur Form** ou *EBNF* pour les intimes, on écrit une liste comme :  $\langle \text{List } T \rangle ::= \text{nil} \mid T \mid \langle \text{List } T \rangle$ . Une chose importante à noter est le deuxième "ou" qui s'écrit comme  $\mid$  signifiant qu'il n'appartient pas à la définition de List T mais plutôt à l'ensemble  $T \mid \langle \text{List } T \rangle$ . Si on lit ceci, on dirait "*Une list d'élément représentant T correspond à un élément vide ou un élément représentant T suivi d'une autre Liste d'élément T.*"

En Oz, la *head* est accessible via [list.1](#) et la *tail* est obtenu via [list.2](#).

```
fun {Sum L}
  case L
  of nil then 0
  [] H|T then H+{Sum T}
  end
end
```

*A clause* →

Ci-contre, on voit une fonction classique en Oz qui analyse une liste et détermine si elle est d'une structure correcte. Le `[]` correspond au case où l'élément `L` est une liste avec une Head et une Tail. On appelle cela une *Clause* et H/T

of. est le pattern de la clause. Le premier cas est défini par

### 3.3 Introduction au langage Kernel

Le langage Kernel est la première partie de la sémantique formelle d'un langage de programmation. Une règle importante est que tout programme écrit en programmation fonctionnelle *peut être traduit en langage kernel*. Les grands principes du langage Kernel sont :

- Tous les résultats intermédiaires de calculs sont visibles.
- Toutes les fonctions deviennent des *procédures* avec un argument en plus. Cet argument donne le résultat de la fonction.
- Les fonctions dans une fonction sont sorties de leur fonction et on leur donne un nouvel identificateur.

Les résultats de la traduction : Les programmes Kernel sont plus longs mais on voit facilement comment un programme s'exécute et on voit si il est *tail-recursive*

### 3.4 Les arbres

Les arbres sont des structures de données extrêmement utiles et utilisées. On peut y stocker des données spécifiques, faire des calculs, ... Les arbres illustrent bien *la programmation orientée but*. Par le standard *EBNF*, on définit un arbre comme suit :  $\langle \text{tree } T \rangle ::= \text{leaf} \mid t(T \langle \text{tree } T \rangle \dots \langle \text{tree } T \rangle)$ . Donc un arbre est une feuille ou *leaf* qui est suivie par un ensemble de *sous-arbres*. Les arbres sont forts similaires au liste si ce n'est que les listes n'ont qu'une sous-listes alors qu'un arbre peut avoir plusieurs sous-arbres.

#### 3.4.1 Ordered Binary tree

Un arbre de ce type à 2 particularités :

- **Binary** : toutes les éléments hors les feuilles possèdent 2 sous-arbres.
- **Ordered** : pour chaque arbre, la clé à gauche est plus petite que la clé de l'arbre et la clé à droite est plus grande.

Ce type d'arbre est très utile pour par exemple effectuer des recherches binaires et permet de facilement et rapidement trouver des données.

#### Lookup K T

Nous permet de trouver une valeur. Ce programme est plutôt simple et il nous suffit de regarder la clé de l'arbre où on est. Puis on compare avec notre recherche, si on est plus grand, on va à droite sinon à gauche. On répète le processus jusqu'à trouver la clé.

Lookup est très efficace car il s'exécute en *log<sub>2</sub>n*, le pire cas est si l'arbre n'est pas équilibré et il ressemble à une liste. Mais en général, en ayant un nombre suffisant de données, il est très rare d'avoir un arbre non équilibré.

#### Insert K W T

Il existe 4 possibilités.

1. remplace une feuille.
2. on remplace un noeud.
3. on remplace un sous-arbre à gauche.
4. on remplace un sous-arbre à droite.



Le premier cas est le plus simple car on crée simplement un nouvel sous-arbre avec 2 feuilles. Si on remplace un noeud, on change la clé et la valeur du noeud. Pour remplacer un sous-arbre, on garde les mêmes clés et valeur de Y pour le noeud mais on change le sous-arbre à gauche ou à droite en fonction.

## Delete K T

Celle-ci est plus compliqué, on a 4 possibilités

1. La valeur qu'on veut supprimer n'existe pas
2. On supprime une feuille.
3. on supprime un sous-arbre à gauche.
4. on supprime un sous-arbre à droite.

Voici.

## 3.5 Tuples et Records

### 3.5.1 Tuples

Un tuple est une manière de stocker des données de différents tuples, l'*ordre* est *important* dans un tuple. On doit également donné un nom, un **label**.

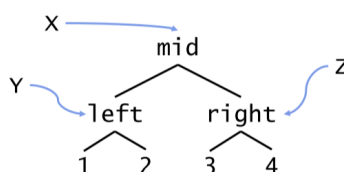
```
X = state(1 b 2)
{Browse {Label X}}
{Browse {Width X}}
```

La première ligne définit un tuple ayant pour *label* "state". La seconde ligne imprime le label du tuple. La dernière affiche sa taille. (c'est donc un entier toujours positif ou 0)

Les champs dans les tuples sont numérotés de 1 à **width X**. On appelle aussi le champ (field) une "*feature*". Un tuple possède toutes ces features de manière consécutives.

On peut donc ainsi construire des structures de données plus compliquées comme des arbres :

```
declare
Y = left(1 2) Z = right(3 4)
X = mid(Y Z)
```



## Comparaison

Il est très simple de comparer des tuples via "=", il ne faut simplement comparer leur valeur à chaque champ. Attention au *loop* causé par les approches naïves.

### 3.5.2 Similitude Tuples et liste

En effet, une liste qui n'est autre que "H|T" peut facilement être traduit en tuple ']' (H T). Quand on peut déterminer un même élément via différentes manières, on appelle ça du *sucré syntaxique*. Dans le *kernel*, on fait au plus simple donc que des tuples.

### 3.5.3 Les Records

Les "*records*" sont une **généralisation** des tuples. La différence avec les tuples est que le *field* peut être n'importe quel valeur et ne doit pas être consécutif. Donc ceci est un *record* correct :

```
X = state(a:1 2:a b:2)
Y = inv(3:a 2:b 1:c)
```

Donc la position d'une valeur et son *field* n'importe plus et on peut déclarer dans le sens qu'on veut. Si on ne nomme pas un *field* dans un *record*, Oz va attribuer un nombre commençant à 1 et qui n'est pas utilisé par un autre champ.

### 3.5.4 Résumé

- Un *atom* est un record de width 0.
- Un tuple est un record avec des champs étant numéroté de manière consécutive de 1 à width  $X$ . (consécutive, donc on skip pas. pas forcément dans l'ordre dans la déclaration)
- Une liste est réalisé avec des *tuples* et des  $(X \ Y)$
- **1** seule *structure de donné* dans le kernel pour rester simple.

## 3.6 Sémantique Formelle

### 3.6.1 Les environnements

Un environnement est une fonction qui passe des *identifieurs* aux *variables en mémoire* autrement dit :  $E_1 = X \rightarrow x, Y \rightarrow y$

#### Environnement contextuel

Un *environnement contextuel* d'une fonction contiens tous les *identificateurs* qui sont usés dans la fonction mais déclaré *en dehors*. Donc ce sont des fonctions qui lorsqu'on appelle une variable va pointer en dehors du scope de la fonction.

#### Stocker une Procédure

Les procédures sont stocker dans la mémoire sous le forme de procédure anonyme symboliser par le "\$".

```
local P Q in
  {Browse 'do something'}
  proc {Q}
    {P}
  end
  {Browse 'another something'}
end
```

Notre "proc Q" sera stocker comme : " $q = (\text{proc}\$P \text{ end}, P \rightarrow p)$ ". On lit donc, la procédure *anonyme* (\$), fais un appel à P (P) et fini (end), son *environnement contextuel* fait que lorsqu'on appelle "P" on va récupérer la valeur "p" en mémoire ( $P \rightarrow p$ ). Donc on voit que l'*environnement contextuel* est stocker avec le code de procédure.

On appelle également la valeur d'une procédure une "*closure*" ou une "*lexically scoped closure*" car elle ferme les identificateurs libres quand défini.