UCLouvain

Résumé de LINFO1104

compilation du 12 juin 2023

Thomas Debelle

Juin 2023

Table des matières

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 couvrira 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" à la fin du cours.

Chapitre 2

Les différents Paradigmes

2.1 Functional Programming

Avec ce paradigme, on impose qu'une variable peut être nommé 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 débuguer. 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 réalisé 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 redéclarer une variable "$X = 11$" car ces deux déclarations pointent à deux éléments totalement différents dans la mémoire et ont des *scopes* différents.

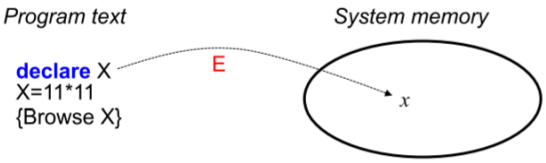


FIGURE 2.1 – Déclaration d'une variable

Un "Scope" ou portée est une propriété centrale en programmation. En effet, c'est le scope qui nous permet d'avoir différente valeur 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".

```
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 scope différents

Chapitre 3

Programmation symbolique

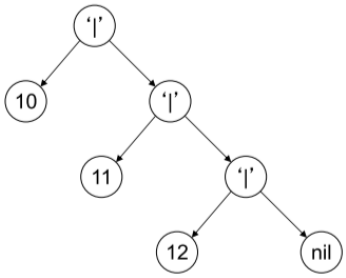
3.1 Listes

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.*

3.1.1 Définition formelle

En utilisant la notation **Extended Backus-Naur Form** ou *EBNF* pour les intimes, on écrit une liste comme : <List T> : := nil | T '|' <List T>. Une chose importante à noter est le deuxième "ou" qui s'écrit comme '|' signifiant qu'il n'appartient pas à la définition de List T mais plutôt à l'ensemble T '|' <List T>. 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.*

Donc une List d'entier se définit comme : <List <Int> >. Une chose importante à remarquer est que j'ai utilisé le mot "représentation" en effet <Int> n'est pas un entier mais une représentation d'entier.



Pour définir une liste en Oz, on utilise soit la notation [1 2 3] ou 1 | 2 | 3 | nil.(il existe d'autre manière semblable qu'on verra plus loin) C'est 2 déclarations reviennent à la même chose en mémoire. Une utilité des listes est leur facilité à être représenté sous forme d'arbre comme montré ci-contre. La *head* est accessible via list.1 et la *tail* est obtenu via list.2.

3.2 Pattern matching

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

Grâce à cette représentation en arbre, il est facile de voir si une liste est bien une liste.
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 cas où l'élément L est une liste avec une Head et une Tail. On appelle cela une *Clause* et H|T est le pattern de la clause. Le premier cas est définit par of.

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 fonctionelle *peut être traduit en langage kernel*. Les grands principes du langage Kernel sont :

- Tous les résultats intermédiaires de calculs sont visibles. Donc on a 1 opération par ligne et la déclaration en locale de toutes les variables.
- Toutes les fonctions deviennent des *procédures anonymes* avec un argument en plus. Cet argument donne le résultat de la fonction.
- Les fonctions dans une fonction sont sortis 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 orienté but*. Par le standard *EBNF*, on définit un arbre comme suit : <tree T> : := leaf | t(T <tree T> ... <tree T>). Donc un arbre est une feuille ou *leaf* qui est suivie par un ensemble de *sous-arbres* (Il faut noter que le $t(...)$ est une façon d'écrire un *record* de label "t" qui est vu au 3.5.3). 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 a 2 particularités :

- Binary : toutes les éléments hors les feuilles possèdes 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 $log_2 n$, 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 créé 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.

TODO

3.5 Tuples et Records

3.5.1 Tuples

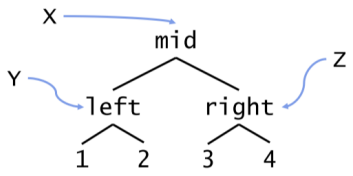
Un tuple est une manière de stocker des données de différents tuples, l'*ordre* est *important* dans un tuple. On doit également donné un nom, un **label**.

```
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 compliqués 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 faut simplement comparer leur value 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 *sucre syntaxique*. Dans le *kernel*, on fait au plus simple donc que des tuples.

```
List1 = [1 2 3]
List2 = (1:1 2:(1:2 2:(1:3 2:nil)))
List1 == List2 //Vrai
```

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 ceux-ci sont des *records* corrects :

```
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 réalisée avec des *tuples* et des $(X \quad Y)$. X étant une donnée et Y étant une sous-liste de données.
- **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 : $E_1 = (X \to x, Y \to y)$

Environnement contextuel

Un *environnement contextuel* d'une fonction contient tous les *identificateurs* qui sont utilisés dans la fonction mais déclarés *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 = (proc{$}{P} end, $\{P \to p\}$)". On lit donc, la procédure *anonyme* ($), fais un appel à P ({P}) et finit (end), son *environnement contextuel* fait que lorsqu'on appelle "P" on va récupérer la valeur "p" en mémoire ($\{P \to p\}$). Donc on voit que l'*environnement contextuel* est stocké 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 définis.
Donc l'avantage d'un environnement contextuel est d'être sûr qu'on appellera la bonne valeur même si elle est déclaré en dehors de la fonction.

Un *identificateur libre* est un identificateur utilisé dans une *fonction* qui est déclaré *en dehors* de la fonction.
Les arguments d'une procédure **ne sont pas** des identificateurs libres car l'argument défini l'identificateur.

3.6.2 Sémantique

Il est important de comprendre le fonctionnement même d'un programme car si on ne comprend pas comment celui-ci fonctionne, il nous domine. *If you do not understand something, then you do not master it – it masters you !*

Définition

La *sémantique* d'un langage de programmation est une explication *précise* de comment un programme s'exécute. Nous verrons la sémantique pour tous les paradigmes. Il en existe 4 types :

1. Sémantique opérationnelle : explique un programme sur base d'*exécution* sur un PC simplifié appelé *la machine abstraite*. → Fonctionne pour tous les paradigmes.
2. Sémantique axiomatique : explique un programme sur base d'*implication*. C'est-à-dire que certaines *propriétés* présentes avant l'exécution, et d'autres seront présentes après. → très utilisé pour la programmation orientée objet comme *Java*.
3. Sémantique de notation : explique un programme comme une *fonction* sur un domaine abstrait. Donc simplifie l'analyse mathématique d'un programme. (utilisé dans *Haskell* et *Scheme*)
4. Sémantique logique : explique un programme comme étant un *modèle logique* basé sur des *axiomes logiques*. Le résultat est une propriété correcte dérivée des axiomes. (cela est implémenté par exemple dans *Prologue* ou dans la *programmation sous contrainte*)

3.6.3 Sémantique opérationnelle

Ce type de sémantique à 2 parties majeures :
— Langage Kernel : traduit le programme en langage Kernel.
— Machine abstraite : puis exécute le programme sur la machine abstraite.

1. Langage Kernel complet

Pour définir correctement une sémantique, il faut tout d'abord s'intéresser à son langage Kernel complet. On peut également prouver qu'un programme est correct en analysant son kernel. Par exemple, prenons ce code kernel :

```
<s> ::= skip
   | < s >1< s >2
   | local <x> in <s> end
   | < x >1=< x >2
   | <x>=<v>
   | if <x> then < s >1 else < s >2 end
   | {<x> < y >1, ..., < y >n}
   | case <x> of <p> then < s >1 else < s >2 end

<v> ::= <number> | <procedure> | <record>
<number> ::= <int> | <float>
<procedure> ::= proc {$ < x >1, ..., < x >n} <s> end
<record>, <p> ::= <lit> | <lit >(< f >1:< x >1, ..., < f >n:< x >n)
```

donc "<s>" contient le programme exécuté, "<v>" est une structure de donné contenant différent type de structure de donné qui sont définis juste en dessous.

2. La machine abstraite

Voici ci-dessous comment s'exécute un programme initialement.

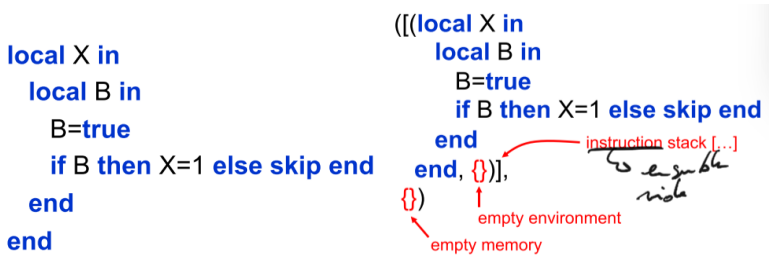


FIGURE 3.1 – à gauche : programme écrit en Oz à droite : état initiale d'exécution

Au début, l'environnement et la mémoire sont vides. L'état d'exécution est écrit typiquement comme :

$([(<s>,E)],\ \sigma)$

Sur la machine abstraite, on va d'instructions en instructions. C'est-à-dire on descend petit à petit. donc on a pour la suite :

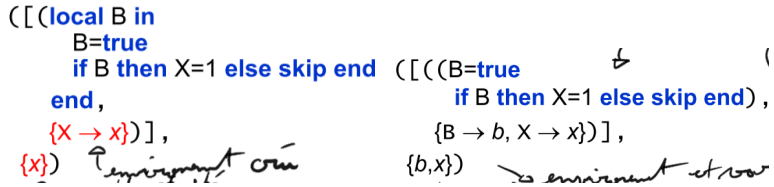


FIGURE 3.2 – à gauche : on descend de 1 cran à droite : on descend encore de 1 cran

On voit que au fur et à mesure qu'on descend, la pile de mémoire et d'environnement s'agrandit. Ensuite on va *séparer* la *composition séquentielle* comme suit :

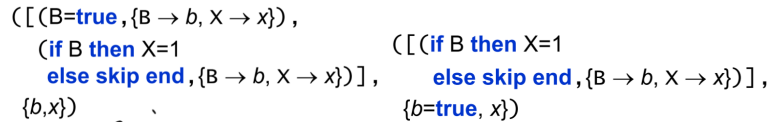


FIGURE 3.3 – à gauche : on sépare en deux à droite : on attribue à b la valeur définit à gauche

Une nouvelle instruction va s'ajouter à cause du "then" de notre condition :

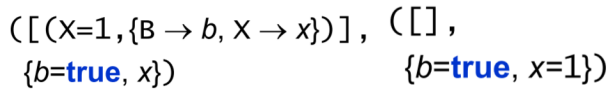


FIGURE 3.4 – à gauche : la nouvelle instruction à droite : les instructions sont vides, c'est fini

3. Définir la machine abstraite

— Pour chaque instructions dans le langage Kernel, on associe sa règle dans la machine abstraite
— Chaque instructions prends un état d'exécution en entrée et sort un état d'exécution en sortie $\rightarrow (ST, \sigma)$.

L'instruction la plus simple est "skip" car il fonctionne comme $([(skip, E), S_2, ..., S_n], \sigma)$ et renvoie $([S_2, ..., S_n], \sigma)$

Instructions	entrée	sortie
skip	1	2
$(< s >_1 < s >_2)$	$([(S_a S_b), S_2, ..., S_n], \sigma)$	$([S_a, S_b, S_2, ..., S_n], \sigma)$
local in <x> in <s> end	$([(\text{local } < x > \text{ in } < s > \text{ end }, E), S_2, ..., S_n], \sigma)$	$([(< s >, E + \{< x > \rightarrow x\}), S_2, ..., S_n], \sigma)$

Il y a également d'autres types d'instructions dont on détaillera pas le langage en machine abstraite :
- <x>=<v> (crée et assigne une valeur) : quand <v> est une procédure, on **doit** créer un environnement contextuel.
- if <x> then $< s >_1$ else $< s >_2$ end (condition) : si <x> n'est pas attribué, l'instruction va attendre ("*block*") jusqu'à ce que <x> soit attribuer à une valeur.
- case <x> of <p> then $< s >_1$ else $< s >_2$ end : Le system de "*case*" se construit en combinant des structures de données Kernel.
- $\{< x >< y >_1, ..., < y >_n\}$: ceci est la base de l'abstraction de donnée

Par ailleurs, voici d'autres concepts de machine abstraite :
- Single-assignment memory $s = \{x_1 = 10, x_2, x_3 = 20\}$: Définition d'une variable et la valeur associée.
- Environnement $E = \{X \rightarrow x, Y \rightarrow y\}$: Lien entre un identificateur et son lien dans la mémoire
- Instruction sémantique $(< s >, E)$: Une instruction avec son environnement.
- Stack Sémantique $ST = [(< s >_1, E_1), ..., (< s >_n, E_n)]$: Un stack d'instructions sémantiques.
- État d'exécution (ST, σ) : Une paire d'un stack sémantique et sa mémoire.
- Execution $(ST_1, s_1) \rightarrow (ST_2, s_2) \rightarrow (ST_3, s_3) \rightarrow ...$: Une séquence d'état d'exécution.

4. Programme correct

grâce à la sémantique, on sait prouver qu'un programme est correct. On dit qu'un programme produit une solution correcte, on l'appelle une *spécification*.
Donc on prouve qu'un programme satisfait la *spécification* quand on utilise une certaine *sémantique*.
La sémantique lie le *programme* à un résultat mathématique appelé *spécification*.

Donc on lie une vérité mathématique à un programme. Et on prouve cela via ces différentes étaps : (exemple avec une factorielle)

1. On commence avec la spécification du programme.
2. Notre programme est *récursif* donc on va utiliser une preuve mathématique par *induction*.
3. On doit prouver le cas de base et le cas général.
4. On utilise la sémantique pour prouver la véracité de notre programme.

5. Procédures

Les procédures sont la base de toutes abstractions de données.
Il y a deux choses importantes dans une *procédure* : sa définition et son appel.

Définition : on crée l'environnement *contextuel*. Puis, on stocke le code de la procédure et son environnement.

Appel : on crée un nouvel environnement combinant l'environnement *contextuel* de la procédure et les variables *formelles*. Ensuite, le tout est exécuté.

```
local Z in
  Z=1
  proc{P X Y}Y=X+Z end
end
```

Ici, le seul identificateur *libre* est **Z** qui est donc déclaré en dehors de la *procédure*. Donc à l'exécution de **P**, **Z** est connu donc **Z** fait partie de l'environnement contextuel de la *procédure*.

```
local P in
  local Z in
    Z=1
    proc{P X Y}Y=X+Z end
  end
  local A B in
     A=10
    {P A B}
    {Browse B}
  end
end
```

Ici, à la ligne de la création de la procédure P, son environnement contextuel est $E_c = \{Z \to z\}$.On va stocker dans la mémoire notre instruction et son environnement *contextuel* c'est la *semantic rule*. Au moment de l'exécution de P avec les valeurs A et B, on va donc ajouter un environnement qui est de la sorte : $E_P = \{Y \to b, X \to a, Z \to z\}$. Donc ce deuxième environnement est stocker dans le *stack sémantique* **pas** dans la mémoire avec le fonction ! Donc en langage *sémantique*, la définition d'une procédure ressemble à cela :

- **Instruction sémantique** : (<x>=proc{$ $< x >_1, ..., < x >_n$} <s> end, E)
  - **Arguments formels** : $< x >_1, ..., < x >_n$
  - **Identificateurs libres de <s>** : $< z >_1, ..., < z >_k$
  - **Environnement contextuel** : $E_C = E_{|<z>_1,...n<z>_k}$ (que les identificateurs libres)
- Cela crée une liaison en mémoire de la forme : x(proc{$ $< x >_1, ..., < x >_n$}<s>, end, E_C)

Maintenant, voyons pour un *appel sémantique* :

- **Instruction sémantique** : ({$\langle x \rangle \langle y \rangle_1 ... \langle y \rangle_n$}, E)
  - Si la condition est *false* donc $E(\langle x \rangle)$ n'est pas lié.
  - Si $E(\langle x \rangle)$ n'est **pas** une procédure, on a une erreur de *condition*.
  - Si $E(\langle x \rangle)$ est une procédure *mais* avec le mauvais nombre d'argument, on a aussi une erreur de *condition*.

Une chose primordiale à comprendre est comment sont stocké les instructions. Elles sont stocké sur une **pile** (*stack*). Donc on a l'instruction sémantique sur le stack : ({$\langle x \rangle \langle y \rangle_1 ... \langle y \rangle_n$}, E) avec la définition de procédure dans la *mémoire* comme cela : $E(\langle x \rangle) = proc\{\$\langle z \rangle_1, ..., \langle z \rangle_n\}\langle s \rangle \text{end}, E_c)$
Ensuite, on met ces instructions sur la *pile* ($\langle s \rangle, E_C + \{\langle z \rangle_1 \to E(\langle y \rangle_1), ..., \langle z \rangle_n \to E(\langle y \rangle_n)\}$)

La machine abstraite fait 2 choses :

1. Adjonction : $E_2 = E_1 + \{X \to y\}$ Donc ajoute une paire (identificateur → variable) à l'environnement. Ré-écrit par dessus E_1 si existe déjà. Utile pour local $< x >$ in $< s >$ end.
2. Restriction : $E_C = E_{|\{X,Y,Z\}}$ Donc limite les *identificateurs* dans un environnement. On a besoin de cela pour calculer l'environnement *contextuel*.

Une adjonction :

```
local X in
  (E1) X=1
  local X in
    (E2) X=2
    {Browse X}
  end
```

```
end
E1 = {Browse → b, X → x}
E2 = E1 + {X → y} = {Browse → b, X → y}
```

Une restriction :

```
local A B C AddB in
  A=1 B=2 C=3 (E)
  fun {AddB X} (EC : contextual environment)
    X+B
  end
end
E = {A → a, B → b, C → c, AddB → a' }
```
$E_C = E_{|\{B\}} = \{B \to b\}$

3.6.4 Résumé

Définir la sémantique permet de relier les programmes au mathématique. On donne des instructions *sémantique* au *kernel* pour qu'il sache comment exécuter dans la *machine abstraite*. La sémantique nous permet de prouver qu'un programme est *correct*.
La sémantique est au cœur de la programmation. Une nouvelle librairie est comme si on ajoutait des instructions au programme donc on augmente sa sémantique.
Quand on écrit un programme, il faut comprendre la sémantique (l'utilisateur n'a pas besoin de savoir). La sémantique doit être simple et complète.
On peut voir la sémantique comme le langage de programmation *ultime*.
Il ne faut pas oublier que les pc sont baser sur les mathématiques *discrètes*.

3.7 Rappel procédure sémantique

Tout d'abord, en programmation nous avons différentes étapes qui reposent chacun sur les précédentes. Fermeture → Programmation d'ordre supérieur → Abstraction des données → Technologie de l'information.

Rappel sur l'exécution d'un programme :

```
{Browse {Inc 10}}                    #Langage pratique (classique)
```

```
local M in                           #Langage Kernel
  local N in
    M=10
    {Inc M N}
    {Browse N}
  end
end
```

A l'exécution,$[(\{IncMN\}, \{M \to m, N \to n, Inc \to i, Browse \to b\}), (\{BrowseN\}, \{M \to m, N \to n, Inc \to i, Browse \to b\})], \{m = 10, n, i = (proc\{\$XY\}Y = X + Aend, \{A \to a\}), a = 1, b = (...browsercode...)\}$ et Inc va référencer cela :
$[(Y = X+A, \{A \to a, X \to m, Y \to n\}), (\{BrowseN\}, \{M \to m, N \to n, Inc \to i, Browse \to b\})], \sigma$
Il est important de remarquer que dans la mémoire, quand on stocke une procédure, on stocke le tout donc avec son environnement contextuel.

Chapitre 4

Programmation d'ordre supérieur

Ce concept découle directement du concept d'*environnement contextuel*. Dans un *procédure* ou *fonction* (les mêmes pour un langage kernel) peuvent prendre des valeurs ou des fonctions en arguments.

Définition

— Une fonction est dit de premier ordre si elle ne prend et ne ressort aucune fonction.
— Une fonction est N+1 si son entrée et sortie prennent en tout N fonctions en argument.

Nomenclature des différentes fonctions :

— **Une génératrice** est le fait de prendre une fonction en entrée d'une fonction.
— **Une instantiation** est le fait de retourner une fonction en sortie d'une fonction.
— **Une composition de fonctions** est le fait de prendre 2 fonctions en entrée et on retourne leur composition.

Utilisation

Via la programmation d'ordre supérieure, on peut cacher un accumulateur. On dit qu'on fait une *abstraction d'accumulateur*.
Une fonction type est la fonction **FoldL** (*reduce*). En effet, la fonction FoldL fait :

```
declare
fun {FoldL L F U}
   case L
   of nil then U
   [] H|T then {FoldL T F {F U H}}
   end
end

{FoldL LIST Function Acc}
```

On peut, un peu dans le même style, faire de l'encapsulation afin de cacher sa valeur à l'intérieur.
→ C'est la base de *l'abstraction de donnés*.

Il faut faire attention à l'**exécution retardé**. En effet, si on ne stocke pas le résultat d'une fonction, elle ne sera exécuté que quand on appellera la valeur. Donc cela peut prendre beaucoup de place en mémoire de stocker une fonction plutôt que son résultat.

Chapitre 5

Lambda Calcul

1. C'est un modèle de calcul qui est Turing complete.
2. **Tous** les types de données peuvent être encodé en lambda calcul.
3. Par le théorème de **Church-Rosser**, le lambda calcul est confluent. Même résultat peu importe l'ordre de réduction
4. C'est la base de la programmation ordre supérieur et de la programmation formelle.

5.1 Introduction

Le *lambda calcul* est une manière mathématique formelle pour représenter des calculs informatiques. Cela a été créé avant l'arrivée des ordinateurs. Cela ne contient que des **définitions**, **appels** et utilise des **liens de variables** et de la **substitution**.

Le *lambda calcul* est une manière universelle de calcul. On peut l'utiliser pour simuler des **machines de Turing**.

5.1.1 Fonctionnement

Le lambda calcul n'a que des *fonctions anonymes d'un seul argument*. Donc si on veut en avoir plusieurs, il faut combiner les fonctions :

$\mathrm{sum\_square(x,y)} = x^2 + y^2$ //fonction classique
$\mathrm{(x,y)} \rightarrow x^2 + y^2$ //fonction anonyme
$\mathrm{x} \rightarrow (\mathrm{y} \rightarrow x^2 + y^2)$ //d'une seul argument
$\lambda x.\lambda y.x^2 + y^2$ //en lambda calcul

Le fait de combiner et de "*nest*" des fonctions s'appellent le currying.

5.1.2 Syntaxe

Les expressions lambdas sont composées de :
— Variables (x,y, ...)
— Du symboles d'abstractions (λ) et de point (.)
— Et des parenthèses
En syntaxe EBNF c'est :

$\mathrm{t} ::= \mathrm{x} \mid (\lambda \mathrm{x}.\mathrm{t}) \mid t_1 \quad t_2$

(λx.t) : est appelé une abstraction (*définition de fonction*)
$t_1 \quad t_2$: c'est *l'appel de fonction.*

5.1.3 En Oz

1. La définition de fonction (λx.t)

   `fun {$ X} T end`

2. L'appel de fonction ($t_1 \quad t_2$)

   $\{T_1 \quad T_2\}$

Le currying en Oz :

1. Définition

   `F = fun {$ X} fun {$ Y} T end end`

2. Appel

   `{{F X} Y}`

5.1.4 Sémantique des expressions lambdas

Le sens d'une expression lambda dépend de comment on peut la réduire. Il en existe 3 types.

1. α-renaming : change le nom des variables liés
2. β-reduction : applique une fonction à un argument
3. η-reduction : enlève les variables inutilisés

Variables libres et liées

Si nous avons : $\lambda x.t$ on dit que *l'opérateur* λx lie la variable x à t. Mais la variable t est libre car n'est lié à *aucune* fonction. Si x est libre dans t alors on dit qu'on capture x.
On dénote $FV(t)$ l'ensemble des variables libres :
— $FV(x) = \{x\}$ où x est une variable
— $FV(\lambda x.t) = FV(t) \backslash \{x\}$
— $FV((t_1 \quad t_2)) = FV(t_1) \cup FV(t_2)$

α-renaming

Ainsi, on peut changer le nom d'une variable lié :

λ x . x $\rightarrow_\alpha \lambda$y . y

Il faut faire attention à ce qu'on ait pas de *conflit de nom* et de *capture de variable.*
Des termes qui diffèrent d'un α-renaming sont dits α-équivalents.

Substitution

La substitution de $t_1[x := t_2]$ remplace toutes les occurrences libres de x dans t_1 par t_2.
On a parfois recourt à l'α-*renaming*. En effet, la substitution ne peut pas *capturer* des variables libres. Donc $(\lambda x.y)[y := x]$ peut être transformé en $(\lambda z.y)[y := x]$. La définition :

$$x[x := t] = t \qquad y[x := t] = y, \text{ if } x \neq y \qquad (t_1 \quad t_2)[x := t] = (t_1[x := t])(t_2[x := t])$$
$$(\lambda x.t_1)[x := t_2] = \lambda x.t_1 \quad (\lambda y.t_1)[x := t_2] = \lambda y.(t_1[x := t_2]), \quad \text{if } x \neq y \wedge y \notin FV(t_2)$$

β-reduction

C'est une application de fonction et se décrit via la substitution. (cfr 5.1.4)
Définition : $(\lambda x.t_1)t_2 \rightarrow t_1[::= t_2]$
Exemple : $(\lambda x.(x \quad x))y \rightarrow (y \quad y)$

η-reduction

C'est l'idée que 2 fonctions sont les *mêmes* si elles produisent le *même résultat* pour tous les arguments possibles. On appelle cela extensibilité. Donc 2 fonctions sont les mêmes si elles ont les mêmes propriétés extérieures.
Définition : $\lambda x.(t \quad x) \to t$ if $x \notin FV(t)$

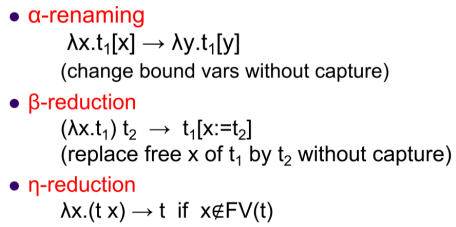


FIGURE 5.1 – Résumé

Convention de notation

Quand on manipule des expression lambda, il est important de suivre quelques règles :
— Enlever les parenthèses les plus extérieur. ex : $(t_1 \quad t_2) \to t_1 \quad t_2$
— Les applications sont associatives depuis la gauche. ex : $t_1 \quad t_2 \quad t_3 \to ((t_1 \quad t_2) \quad t_3)$
— On étend vers la droite ex : $\lambda x.t_1 t_2 \to \lambda x.(t_1 t_2)$
— On peut simplifier les séquences d'abstractions. ex : $\lambda x.\lambda y.\lambda z.t \to \lambda xyz.t$

5.2 Types de données

Le lambda calcul peut faire tout type de calcul, il est Turing complete.
Pour le démontrer, on va encoder des chiffres, opérations, ... en lambda calcul. Donc tous les chiffres, booléens, ... seront encodés en lambda calcul

5.2.1 Nombres

On utilise une notation appelée Church Numerals :
— $0 \triangleq \lambda f.(\lambda x.x)$
— $1 \triangleq \lambda f.(\lambda x.(f \quad x))$
— $2 \triangleq \lambda f.(\lambda x.(f(f \quad x))$
— $3 \triangleq \lambda f.(\lambda x.(f(f(f \quad x))$
On peut remarquer que ces fonctions sont d'ordre supérieur. En effet, elles prennent en argument une fonction et ressortent une fonction. (d'un seul argument)
Le nombre n retourne donc la fonction qui est une composition de n fois de f. Donc on dit que **f est appliqué n fois**.

5.2.2 Opérations

Fonction successeur

Prend le nombre de Church n et retourne $n + 1$.

$$SUCC \triangleq \lambda n.\lambda f.\lambda x.f((n \quad f)x) \qquad SUCC \triangleq \lambda n.\lambda f.\lambda x.f(n \quad f \quad x)$$

Addition

Retourne donc la composition de $m + n$.

$$PLUS \triangleq \lambda m.\lambda n.\lambda f.\lambda x.(m \quad f)((n \quad f)x) \qquad PLUS \triangleq \lambda m.\lambda n.\lambda f.\lambda x.m \quad f(n \quad f \quad x)$$

Multiplication

$$MULT \triangleq \lambda m.\lambda n.\lambda f.m(n \quad f) \qquad MULT \triangleq \lambda m.\lambda n.m(PLUS \quad n) \quad 0$$

La deuxième façon de faire $MULT$ est comme si on disait, "fait une somme m fois en commençant à 0.

Exponentielle

$$POW \triangleq \lambda b.\lambda e.e \quad b$$

Prédécesseur

$$PRED \triangleq \lambda n.\lambda f.\lambda x.n(\lambda g.\lambda h.h(gf))(\lambda u.u)$$

Soustraction

Grâce à la méthode du prédécesseur, on peut définir la soustraction :

$$SUB \triangleq \lambda m.\lambda n.n \quad PRED \quad m$$

Donc pour $m - n$ on effectue la fonction prédécesseur n fois en commençant à m.

5.2.3 Opération logique

Booléens

$$TRUE \triangleq \lambda x.\lambda y.x \qquad FALSE \triangleq \lambda x.\lambda y.y$$

Opérateur logique

$$AND \triangleq \lambda p.\lambda q.p \quad q \quad p \qquad OR \triangleq \lambda p.\lambda q.p \quad p \quad q$$
$$NOT \triangleq \lambda p.p \quad FALSE \quad TRUE \qquad IFTHENELSE \triangleq \lambda p.\lambda a.\lambda b.p \quad a \quad b$$

Prédicat

Ce sont des fonctions qui retournent une valeur booléenne :

$$ISZERO \triangleq \lambda n.n(\lambda x.FALSE)TRUE \qquad LEQ \triangleq \lambda m.\lambda n.ISZERO(SUB \quad m \quad n)$$

Paire

Ce sont des tuples de *width = 2* et peuvent être définis en terme de booléens :

$$PAIR = (x, y) \text{ le tuple} \qquad PAIR \triangleq \lambda x.\lambda y.\lambda f.f \quad x \quad y$$
$$FIRST \triangleq \lambda p.p \quad TRUE \qquad SECOND \triangleq \lambda p.p \quad FALSE$$
$$NIL \triangleq \lambda x.TRUE \qquad NULL \triangleq \lambda p.p(\lambda x.\lambda y.FALSE)$$

Ceci nous introduit à l'abstraction de donnée. Une chose cruciale à comprendre est que depuis les 3 principes simples du lambda calcul, (définition de fonction, utilisation de fonction, des variables en arguments) on peut tout réaliser !

Liste

En sachant qu'une liste est un ensemble de valeur de nil, on sait utiliser des listes en lambda calcul. On peut même les utiliser pour "*simplement*" définir la fonction PRED via SHIFTINC :

$$SHIFTINC \triangleq \lambda x.PAIR(SECOND \quad x)(SUCC(SECOND \quad x)) \quad (m,n) \to (n,n+1)$$
$$PRED \triangleq \lambda n.FIRST(n \quad SHIFTINC(PAIR \quad 0 \quad 0))$$

En effet cela fonctionne car le premier élément sera $n-1$ par rapport au second.

5.2.4 Fonctions récursives

Comme les fonctions sont anonymes, on ne peut pas faire d'appel direct récursif. On va faire en sorte qu'une fonction devient l'argument d'une expression lambda. Par exemple : $G \triangleq \lambda f.\lambda n.$(if n = 0 then 1 else n(f n-1))

Y combinator

Cela nous permet de passer de $Y(g)$ à $g(Y \quad g)$. Ainsi, on sait faire des factorielles de manière récursive :

$$Y \triangleq \lambda g.(\lambda x.g(x \quad x))(\lambda x.g(x \quad x))$$

5.2.5 Théorème de Church-Rosser

Ce théorème nous dit que l'**ordre de réduction** n'a pas d'importance.
Concrètement, si a se réduit à b avec aucune ou plusieurs étapes et que si a se réduit à c avec aucune ou plusieurs étapes. Alors, il existe un d tel que b et c peuvent se réduire à ce dernier.
Le programme est confluent ou possède la *propriété de Church-Rosser*.

5.2.6 Le lambda calcul et les langages de programmation

La *programmation fonctionnelle* peut être comprise en terme de lambda calcul.
— Les valeurs de procédures (donc *lexically scope closures*) **sont** des fonctions lambda.
— l'*Eager et lazy evaluation* est sont des stratégies de réduction différentes.

2 approches

Eager évaluation méthode de réduction qui commence par l'intérieur donc on exécute un programme comme suit :	La lazy évaluation elle, va calculer que ce qui est nécessaire et réduit si nécessaire pour l'exécution. On va de l'extérieur à l'intérieur
{Double {Average 5 7}} → {Double ((5+7)/2)} → {Double (12/2)} → {Double 6} → 6+6 → 12	{Double {Average 5 7}} → {Average 5 7}+{Average 5 7} → ((5+7)/2)+{Average 5 7} → (12/2)+{Average 5 7} → 6+{Average 5 7} → 6+((5+7)/2) → 6+(12/2) → 6+6 → 12

If statement

Dans la plupart des langages de programmation, la condition if est réalisé de manière applicative. La partie de l'action then est réalisé de manière lazy. Donc si une action produit une erreur mais quelle est dans une clause qui ne serait pas atteint, alors le programme ne crashe pas.

5.2.7 Astuces pour le Lambda Calcul

Voici quelques astuces pour vous aider à maitriser le lambda calcul qui peut sembler de prime abord barbare :

1. Ajouter des parenthèse!! Ex : $\lambda x.\lambda y.x + y \quad 1 \quad 2 \rightarrow (\lambda x.(\lambda y.x + y)1)2$ Cela peut sembler anodin mais cela permet de mieux comprendre et lire les expressions qui peuvent vite devenir lourdes. Aussi, garder les parenthèses ça peut aider à voir plus clair.
2. Utiliser des abréviations. Car on n'a pas toujours besoin de savoir comment cela fonctionne et permet de faire partie par partie une fonction.
3. Pour bien réaliser un *η-reduction*, commencer par faire une *β-reduction*. Vous voyez ce qu'il vous reste. Vous faites une *α-renaming* pour retrouver une expression qui correspond à une partie de l'expression de base.
4. Cet article est très utile pour comprendre le *y-combinator* et mieux disséquer les opérations en Lambda Calcul.

5.2.8 Variation et extension

Le lambda calcul est une base fondamentale dans la théorie de l'informatique. Dans les extensions du lambda calcul, on retrouve :

— Lambda calcul typé : donc avec des variables et fonctions
— System F : avec des variables de **types**
— Constructions de calcul : les types sont les valeurs de classes premières
— Combinateur logique : logique sans variables
— Calcul de combinateur SKI : comme le lambda calcul mais sans substitution de variables. On utilise les combinateurs S,K et I.
— Langage Kernel d'OZ : le lambda calcul avec des variables *dataflow*(une seule attribution), exécution *dataflow*, *threads* et evaluation *lazy* explicite.

Chapitre 6

État mutable et abstraction des données

Une chose importante à réaliser, il n'y a pas de notion de temps en programmation. Toutes les fonctions qu'on a sont mathématiques et ne change pas en fonction du temps. Un programme *ne peut observer* son évolution dans le temps sur l'ordinateur.

6.1 Motivation

Une des solutions possibles, on définit le temps abstrait comme une suite de valeurs. On appelle cela un **état**.
Formellement, un state est une séquence calculé de manière *progressive* et contient les résultats *intermédiaire*. Le paradigme fonctionnel peut utiliser les états. ci-dessous la définition de **Sum** comme un état :

```
fun {Sum Xs A}
   case Xs
   of nil then A
   [] X|Xr then
      {Sum Xr A+X}
   end
end
{Browse {Sum [1 2 3 4] 0}}
```

Xs	A
[1 2 3 4]	0
[2 3 4]	1
[3 4]	3
[4]	6
nil	10

Mais cela n'est pas suffisant. En effet, on voudrait que le programme remarque *lui-même* ses propres changements. Il faut faire une **extensions**, on rentre dans un nouveau **paradigme**.

6.2 État explicite

On va essayer de montrer les changements en rendant les états *explicites*. On appelle l'extension une **cellule** ou *cell* (c'est l'équivalent d'un *pointeur* en C).
Donc tout comme en C, on peut changer la référence en changeant le pointeur ou en changeant la valeur pointée.

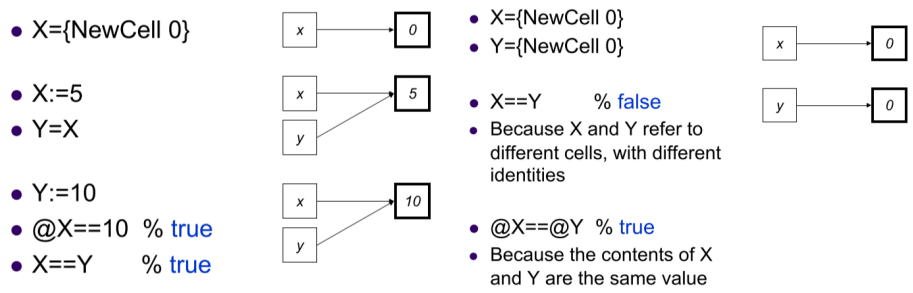
Cell

Une cellule à une identité et un contenu :
— identité : c'est constant et correspond au nom/pointeur de la cellule
— contenu : c'est le contenu stocké qui lui peut varier.

```
A=5; B=6
C={NewCell A}      // on crée une nouvelle cellule
{Browse @C}        // on montre le contenu avec @ ici 5
C:=B               // on ré-attribue le pointeur qui pointe vers B maintenant
{Browse @C}        // vaut 6
```

6.2.1 Exemple

Une chose à bien comprendre est que lorsqu'on réalise l'opération :=, on remplace le contenu de la *box*. Si on lie 2 cell via = on relie notre variable vers la même cell. Donc on peut modifier le contenu de la box via une des deux variables. (6.2.1)



6.3 Sémantique de cellules

On va maintenant étendre la machine abstraite afin d'expliquer comment les *cells* fonctionnent. Tout d'abord, nous avons maintenant **2 types de données**. Les données immutables ou assignement unique et les mutables ou assignements multiples (les cellules).
Une cellule est une paire de 2 *variables*.

— Une variable constante qui est lié au nom de la *cell*
— Une variable qui est lié au contenu de la cellule

Quand on assigne du nouveau contenu à une *cell*, on change la paire (seulement la deuxième variable).

Pour stocker le tout, on a 2 parties comme $\sigma = \sigma_1 \cup \sigma_2$.

— Single assignement : $\sigma_1 = \{t, v, x = \xi, y = \zeta, z = 10, w = 5\}$
— Multiple assignement : $\sigma_2 = \{x : t, y : w\}$

Quand on réalise l'opération $X := Z$ on change σ_2. $x : t \rightarrow x : z$

6.3.1 Programmation impérative

La **programmation impérative** est le nom du nouveau *paradigme*. C'est donc le cumul du *functional paradigm* et du concept de cellules.
La programmation impérative est fondamentale en **programmation orienté objet**.

Langage Kernel de la programmation impérative

```
<s> ::= skip
    | < s >1 < s >2
    | local <x> in <s> end
    | < x >1=< x >2
    | <x>=<v>
    | if <x> then < s >1 else < s >2 end
    | {< x >, < y >1, ..., < y >n}
```

```
   | case <x> of <p> then <s>_1 else <s>_2 end
   | {NewCell <y> <x>}
   | <x>:=<y> ____ {Exchange <x> <y> <z>}
   | <y>=@<x> __|
<v> ::= <number> | <procedure> | <record>
<number> ::= <int> | <float>
<procedure> ::= proc {$ <x>_1, ..., <x>_n} <s> end
<record>, <p> ::= <lit> | <lit>(<f>_1:<x>_1 ... <f>_n:<x>_n)
```

On remplace souvent l'attribution et l'appel du contenu par la fonction *Exchange* qui est une opération atomique (c'est-à-dire 2 opérations indissociables en 1).

6.4 Nécessité de l'état mutable

On dit qu'un programme est modulaire pour une partie si on peut changer cette dernière sans changer le reste du programme. TODO ajouter plus d'info

6.4.1 Comparaison

Dans la programmation fonctionnelle
- Un composant **ne change jamais** de comportement (c'est permanent).
- Mettre à jour un composant signifie que sont interface change et donc on doit mettre à jour d'autres composants.

Dans la programmation impérative
- Un composant peut être mis à jour sans changer son interface et donc on ne doit pas tout mettre à jour.
- Un composant peut changer de comportement à cause des actions passées sur une cellule.

On peut avoir le meilleur des 2 mondes, en simplifiant les mises à jour de composants tout en s'assurant que les cellules se comportent comme on le souhaite.

6.5 Abstraction des données

C'est une des bases pour construire des logiciels complexes. L'abstraction des données est supportée par *les fonctions d'ordre supérieur, static scoping et l'état explicite*. La première chose à réaliser est l'encapsulation.

6.5.1 Encapsulation

C'est le fait de donner à l'utilisateur qu'un nombre *limité* de fonctions et d'instanciations. Par exemple, une TV peut être commandé via une télécommande ayant un nombre limitée de fonction. Son fonctionnement est *encapsulé* dans sa boite et l'utilisateur ne peut pas faire des modifications (*techniquement*) à l'intérieur.

Définition

C'est donc une partie d'un programme qui a une partie *extérieure* (au contact avec l'utilisateur) et une partie *intérieur* (fonctionnement interne). Toutes les opérations à l'intérieur doivent passer par l'interface pour être donné à l'utilisateur.
On limite les *fonctionnalités* et l'utilisateur doit suivre des règles établis de notre interface pour s'assurer que son résultat est bien correct.
L'encapsulation doit être supporté par le langage de programmation.

Avantage

1. On garantit que si on utilise la partie *extérieur*, les résultats sont corrects
2. On simplifie l'utilisation du programme :
   — L'utilisateur **ne doit pas** savoir comment le programme fonctionne.
   — On peut partitionner notre programme pour simplifier le développement et l'utilisation
3. On simplifie le développement de grands programmes. Un développeur est responsable de son implémentation et peut la maintenir.
4. Un développeur ne doit pas connaitre tout le code source pour utiliser les interfaces des autres développeurs.

6.5.2 Les 2 types

Objets

On regroupe dans une même entité les valeurs et opérations. Ex : la *télévision*.

Type de données abstraites

On sépare les valeurs et opérations. Ex : un *distributeur de boisson*→(*on utilise un pièce (fonction) et on reçoit un objet*)

6.6 Le type de données abstraites

Donc cela consiste a un ensemble de *valeurs* et d'*opérations*. Donc par exemple les nombres et les opérations arithmétiques usuelles.
Dans la plupart des *ADT* les valeurs et les opérations n'ont pas **d'états**. Donc valeurs constantes et les opérations n'ont pas de mémoire interne.
Donc par exemple pour un stack, on l'instantie via une fonction qui nous retourne le type de données.

6.6.1 Encapsulation

Le problème avec cette implémentation vu au-dessus d'un stack est que les données ne sont pas protégées donc peuvent être modifier outre les interfaces.

On va protéger les données. On utilisera un Security Wrapper. On utilise ça comme suit :

```
{NewWrapper Wrap Unwrap} //crée une nouvelle clée d'encryption
W={Wrap X} //encrypte
X={Unwrap W} //décrypte
```

C'est donc une sorte d'encryption/décryption. Ainsi, on cache et empêche l'accès aux valeurs car on va encrypter les données primaires. L'utilisateur ne peut invoquer les fonctions Wrap et Unwrap.
Voici l'implémentation d'un stack :

```
local Wrap Unwrap in
  {NewWrapper Wrap Unwrap}

  fun {NewStack} {Wrap nil} end
  fun {Push W X} {Wrap X|{Unwrap W}} end
  fun {Pop W X} S={Unwrap W} in X=S.1 {Wrap S.2} end
  fun {IsEmpty W} {Unwrap W}==nil end
end
```

6.6.2 Remarque

Le premier langage de programmation à implémenter l'*ADT* est CLU et précédé de Simula 67 en 1967.
Ces langages supportent également la protection, ... De nombreux langages actuels supportent implicitement l'*ADT* (par exemple : JAVA, les nombres sont *ADT* et les objets en java ont des propriétés *ADT*)

6.7 Les objets

Cela représente donc à la fois un ensemble de valeur et d'opérations. Donc au début, on instantie notre stack. Par la suite on réalise des opérations de la sorte :

```
S={NewStack}
{S push(X)}
{S pop(X)}
{S isEmpty(B)
```

Donc on voit que notre S est une sorte de fonction et instance, les deux à la fois. Concrètement, on implémente un objet pour un Stack comme ci-dessous :

```
fun {NewStack}
   C={NewCell nil}
   proc {Push X} C:=X|@C end
   proc {Pop X} S=@C in C:=S.2 X=S.1 end
   proc {IsEmpty B} B=(@C==nil) end
in
   proc {$ M}
      case M of push(X) then {Push X}
      [] pop(X) then {Pop X}
      [] isEmpty(B) then {IsEmpty B} end
   end
end
```

Chaque appel à NewStack va donc créer un nouvel objet. Pour réaliser ses procédures, on passe 1 seul argument à notre objet. Pas besoin de Wrapper car on ne peut accéder à ces fonctions que via la création de l'objet stack. On le dissimule donc via le dynamic scoping.

6.7.1 Remarque

De nos jours, les objets sont *omniprésents* en programmation.
Ils sont apparus avec simula67 et a inspiré la plupart des langages actuels.
Cependant, la plupart des langages orientés objets sont en réalité des *ADT* même s'ils incorporent les deux (*ils ont en plus les modules et composants*).

6.8 Les 4 types d'abstraction de données

Les 2 à ajouter sont :

1. Les types de données abstraites **avec état** (*stateful ADT*)
   (a) Très utilisé en C mais sans l'*encapsulation* puisqu'impossible en C.
   (b) On les retrouve dans les classes Java avec des attributs *statiques*.

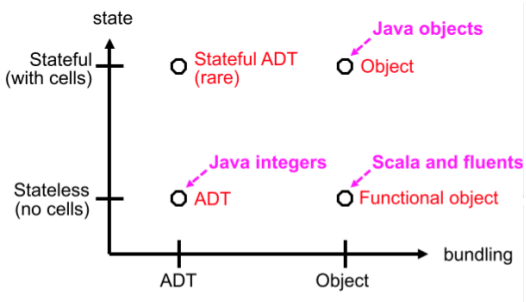


FIGURE 6.1 – Résumé de l'abstraction des données

2. Les objets **sans état** (*functional objects*) :
   (a) Les objets sont immutables Donc appeler un objet, retourne un **nouvel** objet avec une **nouvelle** valeur
   (b) Plus en plus à la mode (*oue oue Scala #EPFL*)

6.8.1 Functional Objects

Constructions

Un *objet fonctionnel* n'a pas de sécurité (pas de cellules ni de wrappers) cela utilise que de la programmation d'ordre supérieur.

```
local
   fun {StackObject S}
      fun {Push E} {StackObject E|S} end
      fun {Pop S1}
         case S of X|T then S1={StackObject T} X end end
      fun {IsEmpty} S==nil end
   in
      stack(push:Push pop:Pop isEmpty:IsEmpty)
   end
   in
      fun {NewStack} {StackObject nil} end
end
```

En Scala

C'est une sorte de forme hybride entre programmation fonctionnel et orienté objet. Scala supporte les 2 *paradigmes*.
En Scala, on peut définir des objets immutables qui retournent, *eux-mêmes*, des objets immutables. Les objets immutables sont des objets *fonctionnels*(*donc pas de changement*).

6.8.2 Stateful ADT

Voici l'implémentation d'un stack en *stateful ADT* :

```
local Wrap Unwrap
   {NewWrapper Wrap Unwrap}
   fun {NewStack} {Wrap {NewCell nil}} end
   proc {Push S E} C={Unwrap S} in C:=E|@C end
   fun {Pop S} C={Unwrap S} in
      case @C of X|S1 then C:=S1 X end end
   fun {IsEmpty S} @{Unwrap S}==nil end
in
   Stack=stack(new:NewStack push:Push pop:Pop isEmpty:IsEmpty)
end
```

On utilise donc à la fois une cellule et un wrapper. Pour mettre à jour un stack, on n'utilise pas de *wrapper* car on va utiliser des *cellules* (ex : pour *Push, Pop et IsEmpty*)

6.9 Remarques supplémentaires

— Jave est designé pour supporter les abstractions de données :
  1. True data abstraction (encapsulation, garbage collector)
  2. Tout entité en Java est un **objet** ou un **ADT**
  3. Supporte les principes de design orienté objet.

— Scala a 2 grand principes :
  1. Séparation entre les états mutables et immutables (programmation fonctionnelle)
  2. Tout est un objet (même les fonctions)

Scala, en plus de fonctionner sur la JVM, est un successeur important de Java et est très versatile via sa puissance expressive.

6.10 Conclusion

L'abstraction des données est un concept nécessaire pour construire des programmes plus *complexes.*

— L'abstraction des données est construite sur : *la programmation d'ordre supérieure, le static scoping, les états explicites, les records et les clés secrètes*
— L'abstraction des données est définie via ces concepts. Ainsi, on connait la sémantique de l'abstraction de donnée.

Il existe 4 types d'abstraction des données orientées selon 2 axes :

1. Objet et ADTs sur un axe
2. Stateful et Stateless

Il existe 2 types d'abstraction des données qui sont dominantes mais les 2 dernières sont toujours utiles pour certains types d'abstraction.
Quasi tous les langages de programmation modernes supportent l'abstraction. Ils supportent généralement plusieurs types. Ils sont souvent plus des langages "*orientés abstractions des données*" que simplement "*orienté objet*".

Chapitre 7

Les exceptions

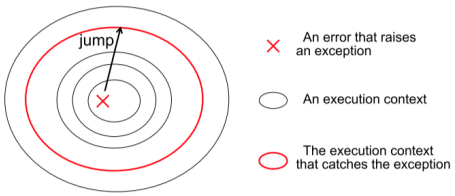
Une exception est lorsqu'un programme ne se comporte pas comme prévu ou que les inputs ne peuvent pas être utilisé (*par exemple* : division par zéro, fichier non-existant, ...)
Ainsi, on peut faire fonctionner le programme même si il rencontre une erreur car celle-ci peut être "*intercepté*" via une exception.

7.1 Fonctionnement

On voudrait donc que le programme ne cesse pas quand il rencontre un soucis et on veut qu'une erreur impacte *au minimum* le programme.
On utilise le principe du confinement.

Le confinement

Donc notre programme est un ensemble imbriqué de contexte d'exécution.
Donc si une erreur apparait, elle influence le contexte et pas tout le programme.
On protège chaque exécution par des *exceptions handlers* qui sont comme des "douaniers" entre chaque zone de contexte. Ainsi, on s'assure que l'erreur est contenu et minimisée.



7.1.1 En Oz

On a les *try* et *raise* qui sont deux nouvelles instructions :

```
try <s1> catch <y> then <s2> end        #On crée l'intercepteur
raise <x> end                           #On établit une erreur
```

Try et Raise

Le *try* met un *marker* sur le stack et puis exécute l'instruction $< S1 >$.
Si il n'y a aucune erreur, l'instruction s'exécute normalement et le *marker* est enlevé à la fin.
Si *raise* est appelé pour signifier une erreur, le stack d'instructions est vidé jusqu'au *marker*. Ensuite, on exécute $< s2 >$. Dans notre exemple, $< y >$ est la même erreur que $< x >$. De plus, la porté de <y> couvre exactement jusqu'à $< s2 >$.

Finally

En plus de ces deux instructions, on peut rajouter une instruction s'appelant *finally* qui comporte des étapes à exécuter peu importe si une erreur arrive ou pas.

7.1.2 En Java

C'est un objet hérité de la class des *Exceptions* (elle-même sous-classe des *Throwable)* et il existe 2 types d'exceptions :

- Checked exceptions : le compilateur vérifie que les méthodes ne *throw* que des exceptions déclarés pour la classe
- Unchecked exceptions : pour gérer les exceptions que le compilateur ne peut pas vérifier (*les rayons cosmiques*). Elles héritent des *RuntimeException* et *Error*.

Donc en tant que développeur on check les erreurs *checked*.

7.1.3 Utilisation correcte

On n'utilise pas une exception pour un comportement **attendu ou prévisible**. C'est-à-dire que, par exemple, on ne lit pas un fichier jusqu'au moment d'une erreur. Non, le fait que le fichier se finisse est prévisible et ne demande pas d'exception.

Chapitre 8

Programmation simultanée

On a besoin de la *programmation simultanée* car notre monde progresse avec le temps et chaque chose progresse indépendamment des autres.

Simultanéité en programmation

1. Système distribuée : relié les pc entre eux par un réseau (data center, ...)
   (a) Une activité en simultanée est appelée un *noeud de calcul*.
   (b) Chaque *noeud de calcul* a ses propres ressources (CPU, mémoire, ...).
2. Système d'exploitation : le controle d'un pc
   (a) Une activité en simultanée est appelée un *processus*.
   (b) Tous les *processus* partagent les mêmes ressources mais ont des espaces allouées de mémoires distincts.
3. Processus : l'exécution d'un seul programme
   (a) Une activité en simultanée est appelée un *thread*.
   (b) Les *threads* partagent le même espace mémoire.

Ce principe est très naturel car est présent tout autour de nous. Une activités indépendante est donc simultanée. Cependant, cela doit être supporté par le langage lui-même.

8.1 Les bases

On peut avoir des événements qui progressent en même temps mais certaines activités doivent pouvoir communiquer entre elle et se synchroniser.

Complexité

On peut faire face à de nombreux soucis en utilisant des *threads* : nondeterminism, race conditions, reentrancy, deadlocks, livelocks, fairness, consistency, shared data.

Mais, on peut faire en sorte que la *programmation simultanée* soit aussi simple que la programmation séquentielle.
Il faut que le paradigme soit correctement choisi. On verra donc le deterministic dataflow qui est une forme de programmation fonctionnelle.

8.2 Deterministic dataflow

Les 3 grands paradigmes de la programmation simultanée :

1. Deterministic dataflow : (*le plus simple et meilleur*)
   — Aussi appelé le *functional dataflow*.
   — Supporte toutes les techniques de la programmation fonctionnelle.
2. Message-passing concurrency : (Erlang et Scala)
   — Les activités envoient des messages entre eux.
3. Shared-state concurrency : (moniteurs Java)
   — Les activités partagent les mêmes données et essayent de travailler ensemble.
   — Plutôt compliqué et encore fort utilisé aujourd'hui.

Variable non-liée

C'est quand on crée une variable dans la mémoire sans qu'elle soit lié à une valeur.
En Oz, quand on veut afficher une variable qui n'est pas encore déclaré, Oz va attendre jusqu'à ce qu'on déclare la variable. Donc ci-dessous, Oz va attendre indéfiniment juste avant l'addition de X :

```
local X Y in
  Y=X+1
  {Browse Y}
end
```

Donc, un Thread peut interagir sur ce code si il attribue une valeur à X. C'est en cela que correspond le dataflow execution :

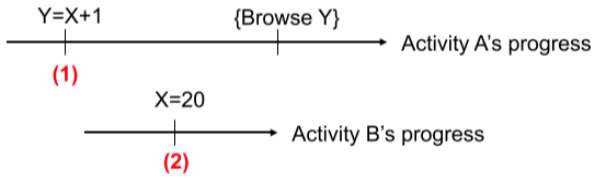


FIGURE 8.1 – Exemple d'exécution

L'activité A attend à (1) jusqu'à ce que l'activité B arrive à (2).

8.3 Threads

C'est donc cela qui nous permet d'implémenter la programmation simultanée en Oz. C'est donc un *fil* d'exécution qui s'exécute en même temps.
Chaque *Threads* fonctionnent de manière séquentielle et sont indépendants l'un de l'autre.
Cependant il faut bien comprendre quelques détails :

— Il n'y a aucun ordre spécifique pour l'exécution des Threads.
— Le système exécute tous les threads en même temps en utilisant l'interleaving semantics. Donc on a un thread qui exécute un à un les threads et change continuellement.
— Le système garantit que chaque Thread reçoit les ressources égales et nécessaires.

Les Threads peuvent *communiquer* entre eux si ils partagent une variable.

Oz

En Oz, créer des Threads est très peu "énergivores" donc on peut se permettre d'en créer un grand nombre.
De plus, on écrit un Thread comme :

```
thread <s> end
```

Donc on peut créer ce programme :

```
declare X
thread {Browse X+1} end
thread {X=1} end
```

Il y a donc 2 façons que les Threads se lancent : ligne 2 → ligne 3 ou ligne 3 → ligne 2. Cependant, le même résultat est le même, 2 est affiché.

Le Browser

Le *Browser* en Oz s'exécute avec son propre Thread.
Pour chaque variable non-liée qui est affiché, on crée un thread dans le *Browser* qui attend jusqu'à ce qu'on lui attribue une valeur.
Cependant, cela ne fonctionne pas avec les *cellules*. Le Browser utilise le paradigme "*functional dataflow*" et <il ne regarde donc pas l'intérieur de la cellule.

8.3.1 Streams et Agents

Un stream est une liste qui se finit par une variable non défini. On peut étendre la liste au tant que l'on veut.
On peut les utiliser comme une façon de communiquer entre des Threads. (ex : *producteur consommateur*)

Producteur Consommateur

- Producteur : génère un stream de données.
- Consommateur : lis un stream et réalise une action dessus.
- Filtre : lis et génère un stream.

Agent

C'est une activité *concurrente* qui lis et écris dans un stream. C'est une raison principale de la **single assignement** car ainsi on peut faire de la récursion terminale qui fait que le *dataflow deterministic* en un paradigme pratique.
Toutes les listes de fonctions peuvent utiliser comme un agent. (on peut aussi faire la *higher order programming*)

8.3.2 Sémantique des Threads

On étend la machine abstraite. Chaque Threads possèdent son propre stack sémantique mais partagent la même mémoire.
On exécute une étape du stack sémantique à la fois et à tour de rôle.
Le Scheduler décide sur quelle thread s'exécute via une technique d'*entrelacement*.

Entrelacements

C'est une manière d'aborder des threads plus simples que de la *vraie concurrence*. En entrelacement un processeur ne peut pas écrire en même sur une zone de mémoire. (on a quand même un cache coherence protocol qui s'assure qu'on n'écrit pas en même temps)

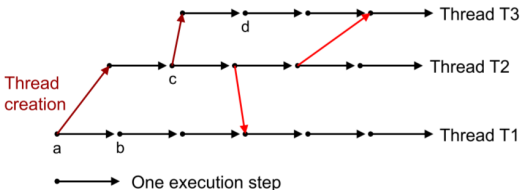
8.4 Exécution de programmes concurrents

Séquentiel

Une exécution d'un programme séquentiel est son exécution sur un thread. Les exécutions séquentielles sont d'ordres totals donc un *ordre définit entre toutes les pairs d'états.*

Ordre partiel

Donc un programme concurrent est un programme avec plus qu'un Thread. Donc on dit qu'on a un ordre partiel. On n'a pas de pair d'exécution définit.
Mais on a toujours qu'une seule exécution par étape



8.5 Le non-déterminisme

Le non-déterminisme d'un système est la possibilité d'un système à réaliser des décisions indépendamment de la volonté du développeur.
Un exemple typique est le planificateur (*scheduler*). Il décide quel thread s'exécute à quel moment.
Ce principe de non-déterminisme est toujours présent dans un système concurrent. En effet, les activités sont toutes indépendantes et le système doit générer cela de manière non-déterministe.

Exemple

On peut penser au fait qu'on ne sait pas qui du thread A ou B s'exécutera en premier. Ce choix varie à l'exécution et est fait par le système.
Pareil pour l'écriture dans une cellule qui à un résultat qui varie.

Piège

Attention, ce n'est pas parce que 2 threads résultent à un même résultat *tout le temps* qu'ils ne sont pas non-déterministes. Le non-déterminisme est le fait qu'on ne sait pas quel thread se lance en premier.

8.5.1 Gestion

On doit toujours gérer le non-déterminisme et il ne devrait jamais influencer le fait qu'un programme est correct. C'est pour cela qu'on évite d'utiliser des threads **et** des cellules

Deterministic dataflow

Avec ce paradigme, le résultat d'un programme est toujours le même. Il n'y a aucun non-déterminisme *visible*.

8.5.2 Fonctionnement d'un planificateur

Le planificateur laisse un thread s'exécuter environ toutes les 10 ms. On appelle ce temps un *time slice*. (Sur des PC multi-cœurs on peut avoir plusieurs exécution simultanée)
Un thread est soit en mode runnable si l'instruction en haut de son stack *n'attend pas* de variable de *dataflow*. Sinon, il est en état suspended donc il attend une assignation de variable.

Équitable

Il est équitable si :
— N'importe quel thread *runnable* va se lancer dans un temps *fini*.
— On accorde une priorité à chaque threads mais cela leur accorde juste plus ou moins de pourcentage de *temps* d'exécution.

Si un *planificateur* est équitable, on peut s'assurer que notre programme est correcte. Sinon, certains programmes ne se lanceraient pas et pourraient produire des réponses érronés.

8.6 Concurrency for dummies

sobrement intitulé par Peter van Roy

En programmation mulit-agent, la concurrence n'influence pas le résultat. La concurrence change juste l'ordre d'exécution. On peut ajouter des threads à la volée sans changer le résultat.

Cela fonctionne seulement en programmation fonctionnelle.

Quand on veut *browse* un thread, on aura une sortie sous forme de "_" quand un thread se finit et attend quelque chose.

8.7 Programmation multi-agent

Par exemple, on fait le crible d'Eratosthène, on va créer une méthode producer consomer un peu plus évolué qui relance un thread à chaque exécution.

```
fun {Sieve Xs}
   case Xs
   of nil then nil
   [] X|Xr then X|{Sieve thread {Filter Xr X} end}
   end
end
declare Xs Ys in
thread Xs={Prod 2} end
thread Ys={Sieve Xs} end
{Browse Ys}
```

8.8 Digital logic simulation

Grâce au *deterministic dataflow paradigm* cela est facile à implémenter. Ainsi, on peut implémenter des circuits sans mémoire et avec mémoire qu'on appelle les circuits à logique séquentielle.

On représente les *signaux* comme des **streams** et les *portes logiques* comme des **agents**.

8.8.1 Modélisation

On sait qu'un signal *digital* est une tension qui évolue en fonction du temps. Les signaux digitaux sont soit 0 et 1 mais peuvent avoir du bruits, glitches, résonance, ...

Une *porte logique* a une entrée et sortie digitale. Ici, on va simplifier les circuits en supposant qu'ils sont parfaits.

Signaux comme stream

On modélise un signal comme un stream avec des 0 et 1 tel que $S = a_0|a_1|a_2|...|a_i|....$

Porte digitale comme agent

C'est bien plus qu'une fonction booléenne. C'est une **entité active**. Par exemple :

```
fun {And A B} if A==1 andthen B==1 then 1 else 0 end end
fun {Loop S1 S2}
  case S1#S2 of (A|T1)#(B|T2) then {And A B}|{Loop T1 T2} end
end
thread Sc={Loop Sa Sb} end
```

Création multiple

On va implémenter une abstraction pour construire plusieurs logic gate. On va nommer cela "{GateMaker Fun}". Cela nous crée des gates se basant sur la fonction *Fun*.

On a ainsi 3 niveaux d'abstraction qu'on peut voir comme de la *programmation orienté objet*.

1. GateMaker est comme une classe générique, qui fabrique.
2. FunG est comme une classe.
3. La *gate logique* est une sorte d'objet.

```
fun {GateMaker F}
  fun {$ Xs Ys}
    fun {GateLoop Xs Ys}
      case Xs#Ys of (X|Xr)#(Y|Yr) then
        {F X Y}|{GateLoop Xr Yr}
      end
    end
  in
    thread {GateLoop Xs Ys} end
  end
end
```

8.8.2 Logique combinatoire

La logique combinatoire **n'a pas** de mémoire et tous les calculs sont faits en même temps. Donc une gate est une combinaison de fonction et chaque gate peut être inter-connecté.
Ceci nous permet d'additionner des nombres et C_{in} permet de prendre en compte une retenu d'un full adder précédent et un C_{out} permet de transmettre une retenu de chiffre.

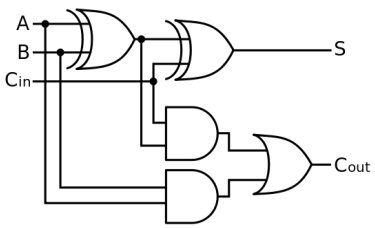
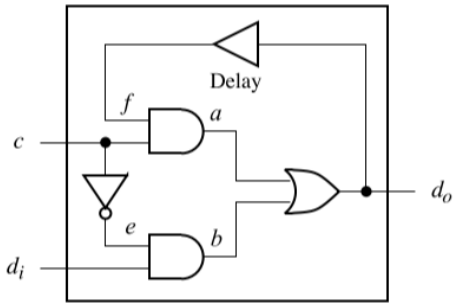


FIGURE 8.2 – Un Full Adder

8.8.3 Logique séquentielle

On a une mémoire des valeurs passées et cela influence. On réalise cela via une gate de délai ou "*Delay gate*". $S = a_0|a_1|a_2|...|a_i|...$ et $T = b_0|b_1|b_2|...|b_i|....$ On a que $b_i = a_{i-1} \Rightarrow T = 0|S$

Latch



Le but est de créer une sorte de bouton pressoir qui une fois pressé reste pressé jusqu'au moment où on represse dessus. Autrement dit, un verrou (*latch*). On peut "*freeze*" le système quand $c = 1$.

8.8.4 Résumé

On construit un programme multi-agent utilisant des streams (liste avec un *tail* non lié) et des agents (des listes fonctionnant sur un thread).

On se base sur 2 idées :

1. Single-assignement variables qui synchronise à la liaison
2. Threads qui définit une séquence d'instructions exécutés.

Cela n'a pas de déterminisme non observable (pas de "*race conditions*"). Le deterministic dataflow est une forme de programmation fonctionnelle.

8.9 Limitations du dataflow deterministique

Le plus gros soucis est qu'on ne peut pas réaliser des programmes où le non-déterminisme doit être *visible*. Ceci est important pour les interactions *clients/serveurs*.

8.9.1 Client serveur

On a un ensemble de client communiquant à un serveur. Les clients sont donc des agents *concurrents*.

On peut essayer via un simple code qui prends 2 streams (pour 2 clients) et performe des actions quand on a besoin. Problème pas *scalable* et quand le client 1 envoie un message cela bloque pour le client 2.

Notre souci est que nous attendons sur des paternes seuls. Avec un case on attend sur un seul pattern. Ici, on veut attendre sur de **multiple** pattern.

Comprendre le non-déterminisme

Le non-déterminisme est un choix fait en dehors du controle du programme. C'est ce qu'on recherche avec le client/serveur car on ne veut pas être retardé car le serveur attend le message d'un autre.

8.9.2 Dépasser les limitations

On va étendre le **langage kernel** avec nouveau concept. On pourrait faire cela avec une fonction de wait mais cela n'est pas du tout *scalable* car on va devoir attendre et gérer de plus en plus de cas.

8.10 Ports

On va créer des Ports qui sont des *streams nommés*. On peut faire :

```
P = {NewPort S}          // Cree un port P avec son stream S
{Send P X}               // Envoie X a la fin du stream du port P
```

Ainsi tous les clients envoient leur requête dans ce stream et le serveur se charge de les gérer. Et ainsi cela crée un ordre aléatoire grâce aux *Threads* donc *non-déterministique*.

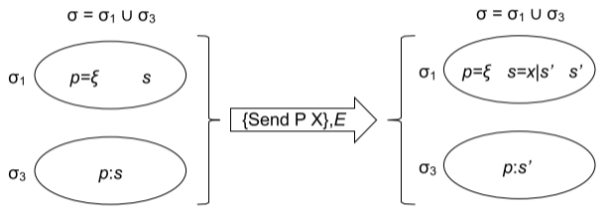
8.10.1 Sémantique

On a maintenant dans notre langage **kernel** que σ_1 stocke les valeurs à assignation unique. σ_2 stocke les cellules et σ_3 stocke des ports qui sont des **paires de variables**.

Langage kernel

P={NewPort S}, {P →p, S → s} //$p, s \in \sigma_1$ crée $p = \xi$ et paire $p : s$ à σ_2
{Send P S}, {P →p, X → x} //$p = \xi$ que $s \in \sigma_1, p : s \in \sigma_2$ crée $s = x|s'$ et update $p : s$

L'opération d'ajout à la fin du stream d'un port est atomique donc cela se fait en **une seule étape**.



8.11 Message-passing concurrency

C'est un nouveau *paradigme* pour la programmation concurrente. C'est du **deterministic dataflow** avec des **ports**. On l'appelle aussi le multi-agent actor programming.
On a des objets *ports* et des objets *actifs*.

8.11.1 Stateless port objects

Un tel objet est une *combinaison* d'un port, d'un thread et d'une fonction de liste récursive. On appelle aussi cela un *stateless agent*.
Un agent est définit selon sa manière qu'il répond à un message. Chaque agent à son **propre thread** donc pas de soucis de concurrence.

Exemple

On peut définir une fonction simple arithmétique de la sorte :

```
proc {Math M}
   case M of
   add(N M A) then A=N+M
   [] mul(N M A) then A=N*M
   ...
   end
end
```

```
MP={NewPort S}
proc {MathProcess Ms}
   case Ms of M|Mr then
      {Math M} {MathProcess Mr}
   end
end

thread {MathProcess S} end
```

On peut aussi utiliser un forAll

```
proc {ForAll Xs P}
   case Xs of nil then skip
   [] X|Xr then {P X} {ForAll Xr P}
   end
end
```

```
proc {MathProcess Ms}
   {ForAll Ms Math}
end
```

Pour construire des objets ports stateless de manière générique. Notre deuxième implémentation utilise la syntaxe des *for loops*.

```
fun {NewPortObject0 Process} Port Stream in
   Port={NewPort Stream}
   thread {ForAll Stream Process} end
   Port
end
```

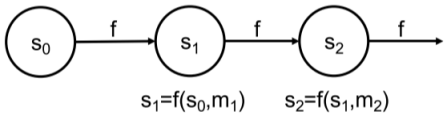
```
fun {NewPortObject0 Process} Port Stream
   Port={NewPort Stream}
   thread for M in Stream do {Process M}
   Port
end
```

8.11.2 Stateful port objects

C'est donc la même chose qu'au 8.11.1 mais ici, les agents ont une **mémoire**.

On appelle sa *mémoire interne* un "state" qui est défini comme ci-contre.

On a des fonctions de *transition* qui passe d'état en état. Cette fonction est de type $F : State \times Msg \longrightarrow State$.




```
fun {NewPortObject Init F}
   proc {Loop S State}
      case S of M|T then
         {Loop T {F State M}} end
   end
      P
in
   thread S in
      P={NewPort S} {Loop S Init} end
   P
end
```

```
proc {Loop S State}
   case S of M|T then {Loop T {F State M}} end
end
```

La fonction F dans notre *loop* est une fonction binaire qui part d'un état initial et effectue différentes opérations. Ce n'est rien d'autre qu'un *Fold*.

8.11.3 Exemples

Updated NewPortObject

```
fun {NewPortObject Init F}
   P Out
in
   thread S in P={NewPort S} Out={FoldL S F Init} end
   P
end
```

Out est l'état final qui termine les agents.

Cell Agent

```
fun {CellProcess S M}
   case M
   of assign(New) then New
   [] access(Old) then Old=S S
   end
end
```

Notre agent se comporte comme une *cellule*. Les cellules et les ports sont équivalents en terme d'expression.

Uniform Interface

```
// On crée et utilise un cell agent
declare Cell
Cell={NewPortObject CellProcess 0}
{Send Cell assign(1)}
local X in {Send Cell access(X)} {Browse X} end

// On veut avoir les mêmes interfaces en tant qu'objet
{Cell assign(1)}
local X in {Cell access(X)} {Browse X} end

// On change la sortie pour être une procédure
fun {NewPortObject Init F}
   P Out
in
   thread S in P={NewPort S} Out={FoldL S F Init} end
   proc {$ M} {Send P M} end
end
```

8.12 Active objects

C'est un object qui est une sorte de "*port object*" et qui a des comportements comme une classe. Donc on a le *polymorphisme* et l'*héritance* de classe tout en ayant du "*message passing*".

8.12.1 Définition Classe et Objet

```
class Counter
   attr i
   meth init(X)
      i := X
   end
   meth inc(X)
      i := @i + X
   end
   meth get(X)
      X=@i
   end
end
```

```
{Ctr inc(10)}
{Ctr inc(5)}
local X in
   {Ctr get(X)}
   {Browse X}
end
```

FIGURE 8.3 – Création et appelle d'un objet

Pour initialiser un objet, on réalise l'opération $CTR = \{New\ Counter\ init(0)\}$.

Active Object

```
fun {NewActive Class Init}
    Obj={New Class Init}
    P
in
    thread S in
        {NewPort S P}
        for M in S do {Obj M} end
    end
    proc {$ M} {Send P M} end
end
```

On combine donc les classes et les ports mais on utilise l'interface *uniforme* de Oz pour leur donner l'apparence d'un objet standard en Oz.

8.12.2 Différence entre passif et actif

On appelle passif, les objets en Oz qui s'exécute **pas dans leur propre thread**. Les actifs eux ont leur propre thread d'exécution.

Avantage

On peut sans aucun risque appeler un objet actif car ses données sont dans son propre thread à l'inverse des passifs.

Dans un objet passif, ses méthodes peuvent s'exécuter de manière **concurrente** à l'inverse des actifs qui seront toujours **séquentielle**.

Les objets passifs ne sont pas *concurrency-safe*.

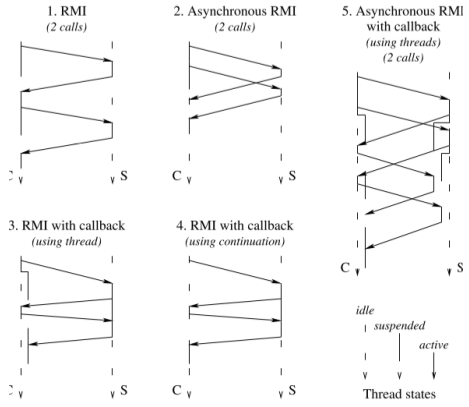
8.13 Message protocol

Ce sont des *séquences* de messages entre plusieurs parties qui peuvent être compris à des niveaux d'abstractions plus élevés que les messages *individuels*.

On commence avec un RMI simple et puis on fait des appelles asynchrones et on y en rajoute.

Le protocole le plus compliqué est le **RMI asynchrone avec callbacks**.

En gros, **RMI** c'est le fait d'envoyer des messages et recevoir des messages. **Asynchrone**, on n'attend pas le message de retour d'un serveur. **Callback**, on envoie un message à un serveur et on lui envoie un port où il peut envoyer son message.



8.14 Memory Management & Garbage Collection

Si jamais on réalise un programme qui boucle et possède une variable qu'on incrémente à chaque fois, on va se retrouver avec n variables alors que nous n'avons besoin que d'une seule.

Notre stack d'instruction est constant via la last call optimization mais notre mémoire ne fait que grandir.

8.14.1 La représentation des données dans la mémoire

On utilise des *memory words* qui prennent *64-bit*. Au démarrage d'un programme, le système attribue un certain nombre de *memory words* pour son exécution. Ce nombre peut varier et est géré par le programme.

Cycle de vie des mots

Il y a **3 états** distincts : *actif*, *inactif* et *libre*. Au début, tout est libre. Un mot devient actif quand il est *alloué* et devient inactif si on ne l'utilise plus. Un mot redevient libre si on le *libère* quand il est actif ou s'il est manuellement ou automatiquement (*garbage collector*) retourné en mode libre depuis l'état inactif.

Bien évidemment on voudrait qu'un mot plus utilisé et inactif redevienne libre pour qu'il puisse libérer de la place.

Soucis

Quand on veut rappeler des mots supposés *inactifs*, on peut avoir du dangling reference. C'est quand on veut reprendre un mot mais qu'il va être utilisée par le programme (*segfault*). On a aussi des memory leak où des mots ne sont jamais rappelés et causent des ralentissements et des utilisations de la mémoire inutile.

Inactif

Un mot est considéré comme **inactif** quand on peut plus y accéder depuis le stack.
L'exécution d'un programme est déterminé par le *stack*. Il faut bien s'assurer qu'il n'existe plus de chemin indirect menant à une variable avant de la considérer comme inactive.

Rappel

On peut soit le faire manuellement (avec des free et malloc en C) mais pas facile. Ou bien, on utilise un *garbage collector* (ramasse-miette) qui est plus pratique et robuste si bien implémenté.

8.14.2 Active memory versus memory consumption

- La mémoire active : combien de mots le programme a besoin à un certain temps. Une **in-memory database** a une grande quantité de mémoire active mais une petite consommation (car on n'utilise qu'une partie de la database).
- La consommation de la mémoire : est la quantité de mots alloués par unité de temps. Donc une **simulation du mouvement des molécules dans une boite** a une grand consommation de mémoire mais peu de mémoire active.

8.15 Deterministic dataflow with ports

Simplifie l'utilisation de programme concurrent. On y ajoute des ports si nécessaire.

8.15.1 Composition concurrente (nombre fixe de Thread)

Cela permet de la composition concurrente (dynamique ou statique) et élimine les dépendances séquentiels.
Si on a plusieurs thread, parfois on doit revenir au principal. Donc le thread original attend que les autres s'arrêtent. C'est ce qu'on appelle la composition concurrente.

$(< s >_1 \| < s >_2)$ // Crée deux threads et attend que les deux se terminent
$< s >_3$ // S'exécute que quand les deux se terminent

Implémentation

On implémente cela en utilisant des variables *dataflow*. On va utiliser la constante unit quand la valeur n'importe pas.

Cela n'a pas d'importance l'ordre dans lequel on attend.

```
local X1 X2 in
   thread <s>1 X1=unit end
   thread <s>2 X2=unit end
   {Wait X1}
   {Wait X2}
end
```

Higher-order abstraction

On va définir une fonction *{Barrier Ps}* avec une liste d'instruction *Ps*.

```
proc {Barrier Ps}
  Xs={Map Ps fun {$ P} X in thread {P} X=unit end X end}
in
  for X in Xs do
    {Wait X}
  end
end
```

Cette fonction *Barrier* peut être réalisé grâce au *deterministic dataflow* uniquement.

Donc on peut créer une abstraction pour cela de types *conc* $< s >_1 \| < s >_2 \| \ldots \| < s >_n$ *end* qui la même chose que $\{Barrier[proc\{\$\} < s >_1 end \quad proc\{\$\} < s >_2 end \quad \ldots \quad proc\{\$\} < s >_n end]\}$

8.15.2 Composition concurrente (nombre variable de Thread)

On doit arriver à créer de nouveau thread qui se synchronise avec les threads actuels. Cette abstraction ne peut pas être écrite en deterministic dataflow. Effectivement, l'ordre de création est inconnu donc **non-déterministique**. On va définir cela avec un port.

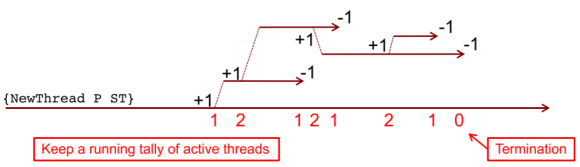
Spécification de l'abstraction

Le Thread principal attend que les autres se terminent.

```
{NewThread proc {$} <s> end SubThread}
{SubThread proc {$} <s> end}
```

La fonction *NewThread* crée un nouveau calcul $< s >$ dans le thread *principal* et ressort dans la procédure *SubThread*.

Donc le *NewThread* se termine qu'après que tous les *SubThread* se terminent. *SubThread* crée un second thread avec $< s >$. Les deux $< s >$ peuvent appeler le *SubThread* et ainsi de suite récursivement. On a ainsi une profondeur d'arbre arbitraire.



On utilise un **port** pour compter le nombre de *Threads* actifs.

```
proc {NewThread P SubThread}  % SubThread is an output
   S Pt={NewPort S}
in
   proc {SubThread P}
      {Send Pt 1}
      thread
         {P} {Send Pt ~1}  % Minus sign in Oz is tilde
      end
   end
   {SubThread P} % Main computation
   {ZeroExit 0 S}   % Keep running sum on S and stop when 0
end
```

Pour que cela soit correct, on doit bien incrémenter avant de créer un thread et dé-incrémenter juste avant de le finir sinon on pourrait parfois tomber à 0 threads actifs en cours d'exécution alors que ce n'est pas vraiment le cas.

Usage de l'invariant

On pose que la somme des éléments S doit être supérieur ou égal au nombre de threads actifs. Si on tombe à 0 on a donc tout fini.

On a 4 actions à une exécution :

1. On envoie 1 : donc notre invariant est vrai
2. On commence un thread : cela se base sur la véracité du premier point. Cela est donc vrai aussi.
3. On retire 1 : le thread ne fait plus rien d'utile et on le fait juste avant de le supprimer. Donc toujours vrai.
4. On arrête le Thread : rien de spécial, toujours vrai.

On va créer une procédure *{ZeroExit N S}* qui voit si on tombe à 0 :

```
proc {ZeroExit N S}
   case S of X|S2 then
      if N+X==0 then skip
      else {ZeroExit N+X S2} end
   end
end
```

8.15.3 Conclusion

3 grands paradigmes pour la programmation concurrente :

1. Deterministic Dataflow : le meilleur mais ne peut pas exprimer le *non-déterminisme* (qui est utile dans le cloud).
2. Message Passing : très général mais plus compliqué. On a des "*stateful agents*" qui communiquent entre eux avec des messages **asynchrones**. (Erlang)
3. Deterministic Dataflow with Ports : meilleur approche. On écrit un programme en grande partie comme en *deterministic dataflow*. On y ajoute des ports quand on en a besoin.

Chapitre 9

Erlang

9.1 Introduction

9.1.1 Force d'Erlang

On a des agents (processus) qui sont légers et s'envoient des **messages** entre eux. Via cela, on peut avoir de la concurrence et avoir de nombreux processus qui fonctionnent ensemble et parallèlement. On peut ainsi faire des programmes sur plusieurs pc. Il y a une vision de "*Let it fail*" car le langage est robuste et sait intercepter et traiter des erreurs. Si une composante ne fonctionne pas, tout le programme ne s'arrête pas nécessairement. On a des arbres **superviseurs** qui regardent s'il y a des erreurs.

9.1.2 Les bases

Erlang est composé de processus qui sont des "*active agents*" qui communiquent de manière asynchrone via du "asynchronous FIFO message passing". Erlang ne partage rien, toutes les données sont copiées. En erlang, chaque processus à une sorte de boite au lettre qui est accessible via **pattern matching** et ne dépend pas de l'ordre.

9.2 Les performances

Pour créer de nouveaux processus et pour envoyer des messages entre, Erlang est extrêmement rapide. Cela est très utile pour des sites web. Là où Apache crash à 4000 requêtes et une bande passante qui se dégrade, Erlang peut en avoir 40000.

9.2.1 Switch AXD301

C'est un switch pour la télécommunication qui est basé sur Erlang et agrémenté via du Java et C. On a une perte linéaire quand on a une surcharge. Quand on demande 1000% des capacités du switch, on a 40% d'efficacité. AXD 301 version 3.2 a 1MLOC Erlang, 900KLOC C/C++, 13KLOC Java.

9.3 Concept de Base

9.3.1 Pure Fonctional Core

Dans chaque processus, Erlang fonctionne comme un langage purement fonctionnel. Chaque variable à une assignation unique. Les fonctions sont des valeurs avec du "*lexically scoped higher-order programming*. Le pattern matching est utilisable dans les *case*, *if* et *receive*.

Toutes les données sont symboliques. Les types de données sont similaires à Oz mais on a également les binary vectors (utilisé pour les calculs de protocoles).

9.3.2 Organisation

Un programme en Erlang est composé de *module* donc chaque fichier s'écrit comme suit : (ex : dans un fichier *math.erl*)

```
-module(math).
-export([areas/1]).
-import(lists, [map/2]).

areas(L) -> lists:sum(map(fun(I) -> area(I) end, L)).

area({square,X}) -> X*X;
area({rectangle,X,Y}) -> X*Y.
```

L'export montre les fonctions à exporter et on y indique le nombre d'arguments. On importe des modules et fonctions. On a ainsi une dépendance de modules.

9.4 Message Passing

9.4.1 Création et envoi

On va créer un nouveau processus qui exécute la fonction "*Fun*" via la fonction "*spawn*". Pid = spawn(Fun). Cela nous renvoie un identifiant de processus. La fonction "*Fun*" peut être anonyme ou non. Le Pid est unique et constant.

On envoie un message via Pid! Message. Cela s'envoie de manière asynchrone et toutes les données sont copiées.

9.4.2 Réception

On intercepte un message via un receive. La boite au lettre est une liste trié de message. On les récupère via le receive puis on fait du pattern matching.
Si la boite au lettre est vide, le processus attend. Sinon, il essaye de match avec un des cas possibles et l'exécute. Si rien ne match, alors le receive se bloque et attend le prochain message. Il reste dans la boite au lettre. Il faut éviter que cela reste indéfiniment dans la boite sinon *memory leak*.

Patterns est une structure de donnée symbolique contenant des identificateurs de variable et des gardes qui sont de simples tests.

9.5 Process Linking

On relie Pid1 à Pid2. PID1 appelle **link(Pid2)** pour faire une liaison. Et vice versa pour faire bidirectionnelle.

9.5.1 Exit

Un processus s'arrête et envoie un **signal d'exit**. Il envoie la raison de son arrêt aux processus qui sont reliés. Si le processus s'arrête normalement, il renvoie l'atome "*normal*". Si il rencontre une erreur à l'exécution il renvoie quelque chose comme {Reason, Stack}. Le comportement classique est que tous les processus s'arrêtent à cause de l'erreur. On peut aussi le faire capturer des erreurs via "*process_flag(trap_exit, true)*". Ainsi, cela est envoyé comme un message sous forme {'EXIT', FromPid, Reason}. On peut également tuer un processus via la fonction exit.

9.6 Changement dynamique

On peut, en Erlang, modifier le code qui est en cours d'exécution. On a ainsi 2 versions des modules qui sont présentes. Les processus peuvent choisir de continuer avec l'ancien processus ou le nouveau. Si on utilise m :Fun cela force à utiliser la nouvelle version.

9.7 Abstraction pour des programmes robustes

Les erreurs ne peuvent être évités et donc on les manipule. Les fonctions sont fail-fast pour éviter les problèmes. Les erreurs sont détectables et les programmes ne partagent pas d'états entre eux.

9.7.1 Slogan

— **Let it fail** : On ne peut pas toujours éviter les erreurs et on doit faire avec. On simplifie les erreurs et on s'y attend.
— **Let some other process do error recovery** : On va utiliser des processus superviseur pour s'occuper des erreurs et relancer le programme.
— **Do not program defensively** : On ne fait pas de "check" d'erreur. Cela n'enlève pas toutes les erreurs et alourdit le programme sinon (fail-safe).

9.7.2 Erlang et OTP système

Systèmes

On a une certaine hiérarchie dans les **systèmes** :

1. Release : Contient toutes les informations pour construire et exécuter un système.
2. Application : Contient tout le logiciel nécessaire pour faire fonctionner une application. Souvent indépendantes l'un de l'autre et avec une certaine hiérarchie.
3. Behavior : Un ensemble de processus qui ensemble implémente un pattern concurrent. Donc contient un superviseur.
4. Worker : Un processus qui est une instance d'un comportement. (typiquement : *gen_server, gen_event ou gen_fsm*)

Comportement

On a 5 comportements :

1. Generic server (gen_server) : pour construire une architecture client/serveur. On a des registres, timeout, start/stop, state management, appel (a)synchrone, erreur.
2. Generic event handler/manager (get_event) : event handlers comme des logs qui répondent à des streams d'évènements et envoie et reçoit des notifications.
3. Generic finite state machine (gen_fsm) : Applications (ex : protocol stack) peuvent être représenter comme des machines d'états finis. On a un ensemble d'état et d'évènement qui donnent des actions et états.
4. Application : composant qui peut être initié et arrêté. Il est réutilisable.
5. Supervisor : arbre superviseur.

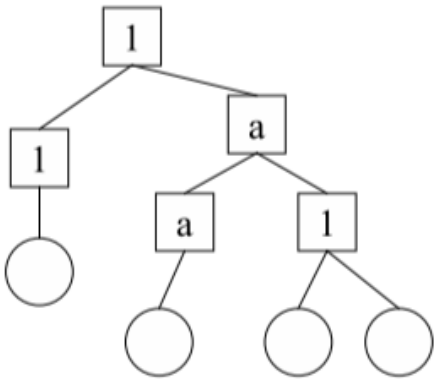
Ces comportements caches la plupart de la complexité de chaque concept. Surtout pour la concurrence et la tolérance à l'erreur.

9.8 arbre superviseur

Les arbres superviseurs prennent l'avantage que des erreurs sont souvent temporaires, donc si on échoue, on relance le processus.

9.8.1 Structure et principe

On a donc un arbre où la root est le noeud qui supervise ses branches. Elle arrête et relance des nodes enfants. Les superviseurs sont eux-mêmes inspectés pour éviter qu'ils crashent. On les démarre via un processus synchrone afin que l'état initial soit correct.
On a différente méthode de redémarrage comme : one_for_one **1** (redémarre un seul), one_for_all **a** (redémarre tous les enfants), rest_for_one. On est limité par le nombre de restart possible par interval. Si on excède cela, un superviseur plus élevé prend le relais. Cela est construit ainsi pour éviter les crash en continu.
Les □ représente des superviseurs et les travailleurs sont représentés via ◯.



9.9 Conclusions

La plateforme Erlang/OTP est une combinaison de Erlang et de librairie standard OTP qui permet de construire des programmes robustes et des systèmes distribuées :

- Erlang : processus (agent actif) qui s'envoie des messages asynchrones et ne partagent pas d'état.
  - Tous les processus sont définis en programmation fonctionnelle et supporte des mises à jour dynamiques.
  - On détecte les erreurs (fait partie du *process linking*).
- OTP : permet d'avoir des programmes robustes en utilisant les comportements, test et arbres superviseurs.
  - Un comportement est un pattern générique de concurrence.
  - Un arbre superviseur contrôle comment gérer les erreurs.
  - Les tests sont faits pour la concurrence et les systèmes distribués. On peut y injecter des erreurs.

Chapitre 10

Conseils pour la syntaxe d'Oz

Voici une liste d'astuces et de choses importantes à savoir sur Oz :
— Déclarer vos fonctions et variables avec une **majuscule** au début !
— Une procédure est une fonction qui ne retourne **rien**.
— Pour retourner une valeur dans une fonction, on écrit une ligne où on ne fait *aucune* assignation ou opération, ... (Ex : on veut retourner X on écrit sur une ligne X tout seul)
— Pour éviter d'avoir des surprises, toujours bien terminer sa liste par un $|nil$
— On peut toujours utiliser une fonction récursive avec **accumulateur** pour une fonction récursive. C'est même conseillé !
— On peut écrire en langage Kernel directement en oz
— On peut faire des listes de procédures via la notation Kernel d'une procédure
— Une liste est un tuple de type (1 :Num 2 :(1 :Num 2 :nil))
— Pour être plus "opti" utiliser les local in.
— Si notre code est le même que celui du voisin mais que vous avez des résultats différents, *relancez emacs ou VScode*. Nettoyer le buffer et feed tout le code dedans.
— Pour être plus *productif*, télécharger "Powertoys" sur Windows et activer l'option "**toujours afficher**" (**always on top**).