

# Compilation de code LINFO1104

compilation du 20 juin 2023

Thomas Debelle

Juin 2023

# Table des matières

<b>1</b>	<b>Code</b>	<b>4</b>
1.1	Les différents paradigmes . . . . .	4
1.1.1	Scoping . . . . .	4
1.2	Programmation symbolique . . . . .	4
1.2.1	Pattern matching . . . . .	4
1.2.2	Tuples et Records . . . . .	4
1.2.3	Arbre avec Tuples . . . . .	4
1.2.4	Similitude Tuples et listes . . . . .	5
1.2.5	Sémantique formelle . . . . .	5
1.2.6	Sémantique opérationnelle . . . . .	5
1.2.7	Procédures . . . . .	6
1.2.8	Rappel procédure sémantique . . . . .	6
1.3	Programmation d'ordre supérieur . . . . .	6
1.3.1	FoldL . . . . .	6
1.4	Lambda calcul . . . . .	6
1.4.1	Fonctionnement . . . . .	6
1.4.2	En Oz . . . . .	7
1.4.3	Sémantique des expressions . . . . .	7
1.4.4	Eager and Lazy evaluation . . . . .	8
1.5	État mutable et abstraction des données . . . . .	8
1.5.1	State . . . . .	8
1.5.2	Cellule . . . . .	8
1.5.3	Langage Kernel extension Cellule . . . . .	8
1.5.4	Fairy tale . . . . .	8
1.6	Type de données abstraites . . . . .	9
1.6.1	ADT avec encapsulation . . . . .	9
1.6.2	Objets stateful . . . . .	9
1.6.3	Functional object (stateless) . . . . .	10
1.6.4	Stateful ADT . . . . .	10
1.7	Les exceptions . . . . .	10
1.7.1	En Oz . . . . .	10
1.8	Programmation Simultanée . . . . .	11
1.8.1	Thread . . . . .	11
1.8.2	Multi-agent . . . . .	11
1.8.3	Porte digitale . . . . .	12
1.8.4	Serveur non fonctionnel et fonctionnel . . . . .	12
1.8.5	Port . . . . .	12
1.8.6	Message-passing concurrency . . . . .	13
1.8.7	Objets ports stateless générique . . . . .	13
1.8.8	Stateful port objects . . . . .	13
1.8.9	Autres Exemples . . . . .	14
1.9	Classe et Objet . . . . .	14

1.9.1	Définition Classe et Objet . . . . .	14
1.10	Deterministic dataflow with ports . . . . .	15
1.10.1	Composition concurrente . . . . .	15
1.11	Erlang . . . . .	16
1.11.1	Organisation . . . . .	16
1.11.2	Send & Receive . . . . .	16
1.11.3	Process Linking . . . . .	17
1.11.4	Dynamic code change . . . . .	17
1.11.5	Client server hotswappable . . . . .	17

# Préface

Bonjour à toi !

Cette synthèse recueille toutes les informations importantes données au cours, pendant les séances de tp et est améliorée 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 utile.

Elle a été réalisée par toutes les personnes que tu vois mentionnées. 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 tes connaissances ? 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

## Code

### 1.1 Les différents paradigmes

#### 1.1.1 Scoping

```
1 local
2   X
3 in
4   X = 42 {Browse X}    % Imprime 42
5   local X in
6     X = 11 {Browse X} % Imprime 11
7   end
8   {Browse X}           % Imprime 42
9 end
```

### 1.2 Programmation symbolique

#### 1.2.1 Pattern matching

```
1 fun {Sum L}
2   case L
3   of nil then 0
4   [] H|T then H+{Sum T}
5   end
6 end
```

#### 1.2.2 Tuples et Records

##### Tuples

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

#### 1.2.3 Arbre avec Tuples

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

## 1.2.4 Similitude Tuples et listes

```

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

```

## 1.2.5 Sémantique formelle

```

1 local P Q in
2   {Browse 'do something'}
3   proc {Q}
4     {P}
5   end
6   {Browse 'another something'}
7 end

```

## 1.2.6 Sémantique opérationnelle

### Langage Kernel complet

```

1 <s> ::= skip
2   | <s>1<s>2
3   | local <x> in <s> end
4   | <x>1=<x>2
5   | <x>=<v>
6   | if <x> then <s>1 else <s>2 end
7   | {<x> <y>1,...,<y>n}
8   | case <x> of <p> then <s>1 else <s>2 end
9
10 <v> ::= <number> | <procedure> | <record>
11 <number> ::= <int> | <float>
12 <procedure> ::= proc {$ <x>1,...,<x>n} <s> end
13 <record>, <p> ::= <lit> | <lit>(<f>1:<x>1,...,<f>n:<x>n)

```

### La machine abstraite

```

1 local X in
2   local B in
3     B=true
4     if B then X=1
5     else skip end
6   end
7 end

```

Listing 1.1 – Programme en Oz

```

1 ([[local X in
2   local B in
3     B=true
4     if B then x=1 else skip end
5     end
6     end, {}]],
7 {}))

```

Listing 1.2 – État initial

```

1 ([[local B in
2   B=true
3   if B then X=1 else skip end
4   end, {X → x}]],
5 {x})

```

```

1 ([[B=true
2   if B then X=1 else skip end),
3 {B → b, X → x}]],
4 {b, x})

```

```

1 ([[X=1, {B → b, X → x}]],
2 {b=true, x})

```

```

1 ([,
2 {b=true, x=1})

```

## 1.2.7 Procédures

### Adjonction

```
1 local X in
2   (E1) X=1
3   local X in
4     (E2) X=2
5     {Browse X}
6   end
7 end
8 E1 = {Browse → b, X → x}
9 E2 = E1 + {X → y} = {Browse → b, X → y}
```

### Restriction

```
1 local A B C AddB in
2   A=1 B=2 C=3 (E)
3   fun {AddB X} (EC: contextual environment)
4     X+B
5   end
6 end
7 E = {A → a, B → b, C → c, AddB → a' }
8 EC = E|{B} = {B → b }
```

## 1.2.8 Rappel procédure sémantique

```
1 {Browse {Inc 10}} #Langage pratique (classique)
2
3 local M in #Langage Kernel
4   local N in
5     M=10
6     {Inc M N}
7     {Browse N}
8   end
9 end
```

## 1.3 Programmation d'ordre supérieur

### 1.3.1 FoldL

```
1 declare
2 fun {FoldL L F U}
3   case L
4   of nil then U
5   [] H|T then {FoldL T F {F U H}}
6   end
7 end
8
9 {FoldL LIST Function Acc}
```

## 1.4 Lambda calcul

### 1.4.1 Fonctionnement

```

1 declare
2 fun {FoldL L F U}
3   case L
4   of nil then U
5   [] H|T then {FoldL T F {F U H}}
6   end
7 end
8
9 {FoldL LIST Function Acc}

```

## Fonctionnement

```

1 t ::= x | (λx.t) | t1 t2

```

## 1.4.2 En Oz

### Définition fonction

```

1 fun {$ X} T end

```

```

1 {T1 T2}

```

### Currying

```

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

```

```

1 {{F X} Y}

```

## 1.4.3 Sémantique des expressions

### α-renaming

```

1 λ x. x →α λy. y

```

### β-renaming

```

1 (λx.t1)t2 → t1 ::= t2
2 (λx.(x x))y → (y y)

```

### η-renaming

```

1 λx.(t x) → t if x ∉ FV(t)

```



## 1.4.4 Eager and Lazy evaluation

1 {Double {Average 5 7}} →	1 {Double {Average 5 7}} →
2 {Double ((5 + 7)/2)} →	2 {Average 5 7} + {Average 5 7} →
3 {Double (12/2)} →	3 ((5 + 7)/2) + {Average 5 7} →
4 {Double 6} →	4 (12/2) + {Average 5 7} →
5 6 + 6 →	5 6 + {Average 5 7} →
6 12	6 6 + ((5 + 7)/2) →
7 12	7 6 + ((12)/2) →
8 12	8 6 + 6 →
9 12	9 12

## 1.5 État mutable et abstraction des données

### 1.5.1 State

1 fun {Sum Xs A}	1 Xs            A
2    case Xs	2            -----
3    of nil then A	3 [1 2 3 4] 0
4    [] X Xr then {Sum Xr A+X}	4 [2 3 4] 1
5    end	5 [3 4] 3
6 end	6 [4] 6
7 {Browse {Sum [1 2 3 4] 0}}	7 nil 10

### 1.5.2 Cellule

```

1 A=5; B=6
2 C={NewCell A} // on crée une nouvelle cellule
3 {Browse @C}   // on montre le contenu avec @. Imprime 5
4 C:=B          // on change la valeur pointée par C en celle de B
5 {Browse @C}   // Imprime 6

```

### 1.5.3 Langage Kernel extension Cellule

```

1 <s> ::= skip
2   | <s>1 <s>2
3   | local <x> in <s> end
4   | <x>1=<x>2
5   | <x>=<v>
6   | if <x> then <s>1 else <s>2 end
7   | {<x>, <y>1, ..., <y>n}
8   | case <x> of <p> then <s>1 else <s>2 end
9   | {NewCell <y> <x>}
10  | <x>:=<y> ---- {Exchange <x> <y> <z>}
11  | <y>=@<x> --|
12 <v> ::= <number> | <procedure> | <record>
13 <number> ::= <int> | <float>
14 <procedure> ::= proc {$ <x>1, ..., <x>n} <s> end
15 <record>, <p> ::= <lit> | <lit>(<f>1:<x>1 ... <f>n:<x>n)

```

### 1.5.4 Fairy tale

```

1 fun {MF}
2   X = {NewCell 0}
3   fun {F ...}
4     X:=@X+1 % Definition of F
5   end
6   fun {G ...}
7     % Definition of G
8   end
9   fun {Count} @X end
10 in 'export'(f:F g:G c:Count)
11 end
12 M = {MF}

```

## 1.6 Type de données abstraites

### 1.6.1 ADT avec encapsulation

#### Wrapper

```

1 {NewWrapper Wrap Unwrap} //crée une nouvelle clé d'encryption
2 W={Wrap X}               //encrypte
3 X={Unwrap W}             //décrypte

```

#### ADT

```

1 local Wrap Unwrap in
2   {NewWrapper Wrap Unwrap}
3
4   fun {NewStack} {Wrap nil} end
5   fun {Push W X} {Wrap X|{Unwrap W}} end
6   fun {Pop W X} S={Unwrap W} in X=S.1 {Wrap S.2} end
7   fun {IsEmpty W} {Unwrap W}==nil end
8 end

```

### 1.6.2 Objets stateful

#### Utilisation

```

1 S={NewStack}
2 {S push(X)}
3 {S pop(X)}
4 {S isEmpty(B)}

```

#### Objet

```

1 fun {NewStack}
2   C={NewCell nil}
3   proc {Push X} C:=X|@C end
4   proc {Pop X} S=@C in C:=S.2 X=S.1 end
5   proc {IsEmpty B} B=(@C==nil) end
6 in
7   proc {$ M}
8     case M of push(X) then {Push X}
9       [] pop(X) then {Pop X}
10      [] isEmpty(B) then {IsEmpty B} end
11   end

```

```
12 end
```

## Objet avec un Record

```
1 fun {NewStack}  
2   C={NewCell nil}  
3   proc {Push X} C:=X|@C end  
4   proc {Pop X} S=@C in X=S.1 C:=S.2 end  
5   fun {IsEmpty} @C==nil end  
6 in  
7   stack(push:Push pop:Pop isEmpty:IsEmpty)  
8 end
```

### 1.6.3 Functional object (stateless)

```
1 local  
2   fun {StackObject S}  
3     fun {Push E} {StackObject E|S} end  
4     fun {Pop S1}  
5       case S of X|T then S1={StackObject T} X end end  
6     fun {IsEmpty} S==nil end  
7   in  
8     stack(push:Push pop:Pop isEmpty:IsEmpty)  
9   end  
10  in  
11    fun {NewStack} {StackObject nil} end  
12 end
```

### 1.6.4 Stateful ADT

```
1 local Wrap Unwrap  
2   {NewWrapper Wrap Unwrap}  
3   fun {NewStack} {Wrap {NewCell nil}} end  
4   proc {Push S E} C={Unwrap S} in C:=E|@C end  
5   fun {Pop S} C={Unwrap S} in  
6     case @C of X|S1 then C:=S1 X end end  
7   fun {IsEmpty S} @{Unwrap S}==nil end  
8 in  
9   Stack=stack(new:NewStack push:Push pop:Pop isEmpty:IsEmpty)  
10 end
```

## 1.7 Les exceptions

### 1.7.1 En Oz

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

#### Exemple typique

```
1 fun {Eval E}  
2   if {IsNumber E} then E  
3   else  
4     case E of plus(X Y) then {Eval X}+{Eval Y}
```

```

5      [] times(X Y) then {Eval X}*{Eval Y}
6      else raise badExpression(E) end
7      end
8      end
9      end
10
11  try
12    {Browse {Eval plus(23 times(5 5))}}
13    {Browse {Eval plus(23 minus(4 3))}}
14  catch X then {Browse X} end

```

## Finally

```

1 FH={OpenFile 'foobar'}
2 try
3   {ProcessFile FH}
4 catch X then
5   {Show '*** Exception during execution ***'}
6 finally {CloseFile FH} end % Always close the file

```

# 1.8 Programmation Simultanée

## 1.8.1 Thread

```

1 thread <s> end

```

## 1.8.2 Multi-agent

### Sieve

```

1 fun {Sieve Xs}
2   case Xs
3   of nil then nil
4   [] X|Xr then X|{Sieve thread {Filter Xr X} end}
5   end
6 end
7 declare Xs Ys in
8 thread Xs={Prod 2} end
9 thread Ys={Sieve Xs} end
10 {Browse Ys}

```

```

1 fun {CMap Xs F}
2   case Xs of nil then nil
3   [] X|Xr then
4     thread {F X} end | {CMap Xr F}
5   end
6 end

```

### Compteur thread

```

1 C={NewCell 0}
2 proc {Inc C}
3   {Exchange C X Y} Y=X+1
4 end
5
6 fun {Fib X}

```

```

7  if X==0 then 0
8  elseif X==1 then 1
9  else
10   thread {Inc C} {Fib X-1} end + {Fib X-2}
11 end
12 end

```

### 1.8.3 Porte digitale

#### And

```

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

```

#### Gatemaker

<pre> 1 fun {GateMaker F} 2   fun {\$ Xs Ys} 3     fun {GateLoop Xs Ys} 4       case Xs#Ys of (X Xr)#(Y Yr) then 5         {F X Y} {GateLoop Xr Yr} 6       end 7     end 8   in 9     thread {GateLoop Xs Ys} end 10  end 11 end </pre>	<pre> 1 % Creation d'un interrupteur 2 3 proc {Latch C Di Do} 4   A B E F 5 in 6   F={DelayG Do} 7   A={AndG C F} 8   E={NotG C} 9   B={AndG E Di} 10  Do={OrG A B} 11 end </pre>
--	---

### 1.8.4 Serveur non fonctionnel et fonctionnel

<pre> 1 proc {Server S1 S2} 2   case S1 S2 of (M1 T1) S2 3   then (handle M1) {Server T1 S2} 4   [] S1 (M2 T2) then 5     (handle M2) {Server S1 T2} 6   end 7 end </pre>	<pre> 1 proc {Server S} 2   case S of M T then 3     (handle M) 4     {Server T} 5   end 6 end 7 end </pre>
---	---

Listing 1.3 – non fonctionnel

Listing 1.4 – fonctionnel

### 1.8.5 Port

#### En Oz

```

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

```

#### Langage Kernel

```

1 P={NewPort S}, {P → p, S → s} //p,s ∈ σ1 crée p = ξ et paire p : s à σ2
2 {Send P S}, {P → p, X → x} //p = ξ que s ∈ σ1, p : s ∈ σ2 crée s = x|s' et update p : s

```

## 1.8.6 Message-passing concurrency

### MathProcess

<pre>1 proc {Math M} 2   case M of 3     add(N M A) then A=N+M 4     [] mul(N M A) then A=N*M 5     ... 6   end 7 end</pre>	<pre>1 MP={NewPort S} 2 proc {MathProcess Ms} 3   case Ms of M Mr then 4     {Math M} {MathProcess Mr} 5   end 6 end 7 thread {MathProcess S} end</pre>
---	---

### ForAll

<pre>1 proc {ForAll Xs P} 2   case Xs of nil then skip 3   [] X Xr then {P X} {ForAll Xr P} 4   end 5 end</pre>	<pre>1 proc {MathProcess Ms} 2   {ForAll Ms Math} 3 4 5 end</pre>
---	---

## 1.8.7 Objets ports stateless générique

```
1 fun {NewPortObject0 Process} Port Stream in
2   Port={NewPort Stream}
3   thread {ForAll Stream Process} end
4   Port
5 end
```

```
1 fun {NewPortObject0 Process} Port Stream in
2   Port={NewPort Stream}
3   thread for M in Stream do {Process M} end end
4   Port
5 end
```

## 1.8.8 Stateful port objects

```
1 fun {NewPortObject Init F}
2   proc {Loop S State}
3     case S of M|T then
4       {Loop T {F State M}} end
5   end
6   P
7 in
8   thread S in P={NewPort S} {Loop S Init} end
9   P
10 end
```

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

## 1.8.9 Autres Exemples

### Updated NewPortObject

```
1 fun {NewPortObject Init F}  
2   P Out  
3 in  
4   thread S in P={NewPort S} Out={FoldL S F Init} end  
5   P  
6 end
```

### Cell Agent

```
1 fun {CellProcess S M}  
2   case M  
3   of assign(New) then New  
4   [] access(Old) then Old=S S  
5   end  
6 end
```

### Uniform Interface

```
1 // On crée et utilise un cell agent  
2 declare Cell  
3 Cell={NewPortObject CellProcess 0}  
4 {Send Cell assign(1)}  
5 local X in {Send Cell access(X)} {Browse X} end  
6  
7 // On veut avoir les mêmes interfaces en tant qu'objet  
8 {Cell assign(1)}  
9 local X in {Cell access(X)} {Browse X} end  
10  
11 // On change la sortie pour être une procédure  
12 fun {NewPortObject Init F}  
13   P Out  
14 in  
15   thread S in P={NewPort S} Out={FoldL S F Init} end  
16   proc {$ M} {Send P M} end  
17 end
```

## 1.9 Classe et Objet

### 1.9.1 Définition Classe et Objet

```
1 class Counter  
2   attr i  
3   meth init(X)  
4     i := X  
5   end  
6   meth inc(X)  
7     i := @i + X  
8   end  
9   meth get(X)  
10    X=@i  
11  end  
12 end
```

```

1 {Ctr inc(10)}
2 {Ctr inc(5)}
3 local X in
4   {Ctr get(X)}
5   {Browse X}
6 end

```

## Active Object

```

1 fun {NewActive Class Init}
2   Obj={New Class Init}
3   P
4 in
5   thread S in
6     {NewPort S P}
7     for M in S do {Obj M} end
8   end
9   proc {$ M} {Send P M} end
10 end

```

## 1.10 Deterministic dataflow with ports

### Définition

```

1 (< s >1 || < s >2)      // Crée deux threads et attend que les deux se terminent
2 < s >3                  // S'exécute que quand les deux se terminent

```

### Implémentation

```

1 local X1 X2 in
2   thread < s >1 X1 = unit end
3   thread < s >2 X2 = unit end
4   {Wait X1}
5   {Wait X2}
6 end

```

### Higher-order abstraction

```

1 proc {Barrier Ps}
2   Xs={Map Ps fun {$ P} X in thread {P} X=unit end X end}
3 in
4   for X in Xs do
5     {Wait X}
6   end
7 end

```

### 1.10.1 Composition concurrente

```

1 proc {NewThread P SubThread} //SubThread is an output
2   S Pt={NewPort S}
3 in
4   proc {SubThread P}
5     {Send Pt 1}

```



```

6     thread
7         {P} {Send Pt ~1} //Minus sign is tilde
8     end
9 end
10 {SubThread P} //Main computation
11 {ZeroExit 0 S} //Keep running sum on S and stop when 0
12 end

```

## Compteur de thread

```

1 proc {ZeroExit N S}
2     case S of X|S2 then
3         if N+X==0 then skip
4         else {ZeroExit N+X S2} end
5     end
6 end

```

## 1.11 Erlang

### 1.11.1 Organisation

```

1 -module(math).
2 -export([areas/1]).
3 -import(lists, [map/2]).
4
5 areas(L) -> lists:sum(map(fun(I) -> area(I) end, L)).
6
7 area({square,X}) -> X*X;
8 area({rectangle,X,Y}) -> X*Y.

```

## Fonction

```

1 Pid = spawn(fun) % Pour spawn une fonction et avoir son numero de processus
2
3 fun(args) -> expr end % anonyme ou nomme
4 fun name/arity

```

## Receive

```

1 receive
2     pattern1 when guard1 -> expr1;
3     pattern2 when guard2 -> expr2;
4     ...
5     patternN when guardN -> exprN
6 end

```

### 1.11.2 Send & Receive

```

1 Pid ! Message,
2 ...
3
4 receive
5     Message1 ->
6     Actions1;

```

```

7   Message2 ->
8       Actions2;
9   ...
10  after Time ->
11      TimeOutActions
12 end

```

### 1.11.3 Process Linking

```

1 start() -> spawn(fun go/0).
2
3 go() ->
4     process_flag(trap_exit, true),
5     loop().
6
7 loop() ->
8     receive
9         {'EXIT',Pid,Why} -> ...
10    ... -> ..., loop()
11 end.

```

### 1.11.4 Dynamic code change

```

1 -module(m).
2
3 loop(Data, F) ->
4     receive
5         {From,Q} ->
6             {Reply,Data1}=F(Q,Data),
7             m:loop(Data1, F)
8     end.

```

Listing 1.5 – use new version

```

1 -module(m).
2
3 loop(Data, F) ->
4     receive
5         {From,Q} ->
6             {Reply,Data1}=F(Q,Data),
7             loop(Data1, F)
8     end.

```

Listing 1.6 – use old version

### 1.11.5 Client server hotswappable

<pre> 1 server(Fun, Data) -&gt; 2     receive 3         {new_fun,Fun1} -&gt; 4             server(Fun1,Data); 5         {rpc,From,ReplyAs,Q} -&gt; 6             {Reply,Data1} = Fun{Q,Data}, 7             From!{ReplyAs,Reply}, 8             server(Fun, Data1) 9     end. </pre>	<pre> 1 % Fonction RPC 2 3 4 rpc(A,B) -&gt; 5     Tag=new_ref(), 6     A!{rpc,self(),Tag,B}, 7     receive 8         {Tag,Val} -&gt; Val 9     end. </pre>
--	--