MI-GLSD-M1 -UEM213 : Programming languages paradigms

Chapter IV: Logic paradigm

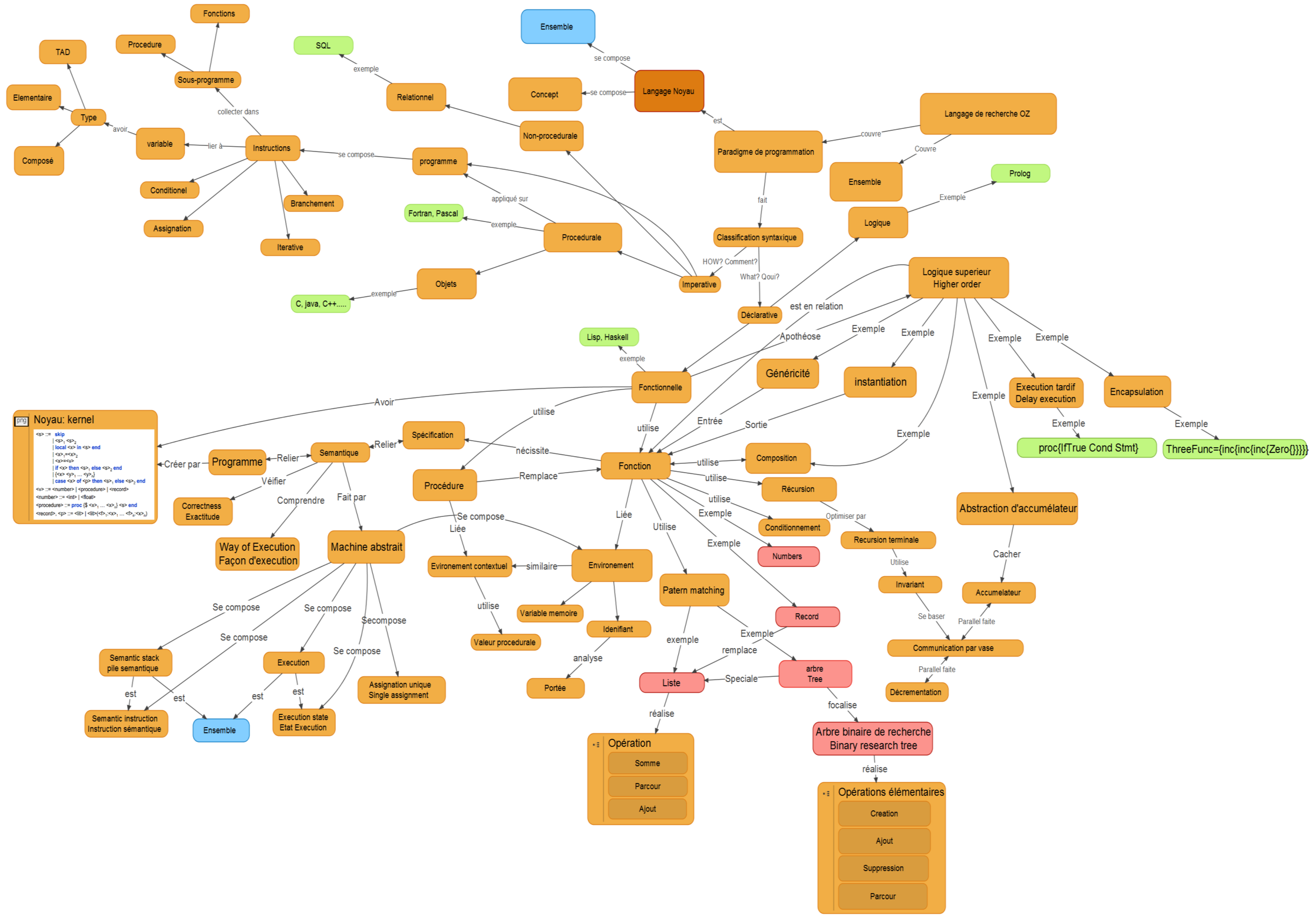
A. HARICHE

University of Djilali Bounaama, Khemis Meliana (UDBKM)

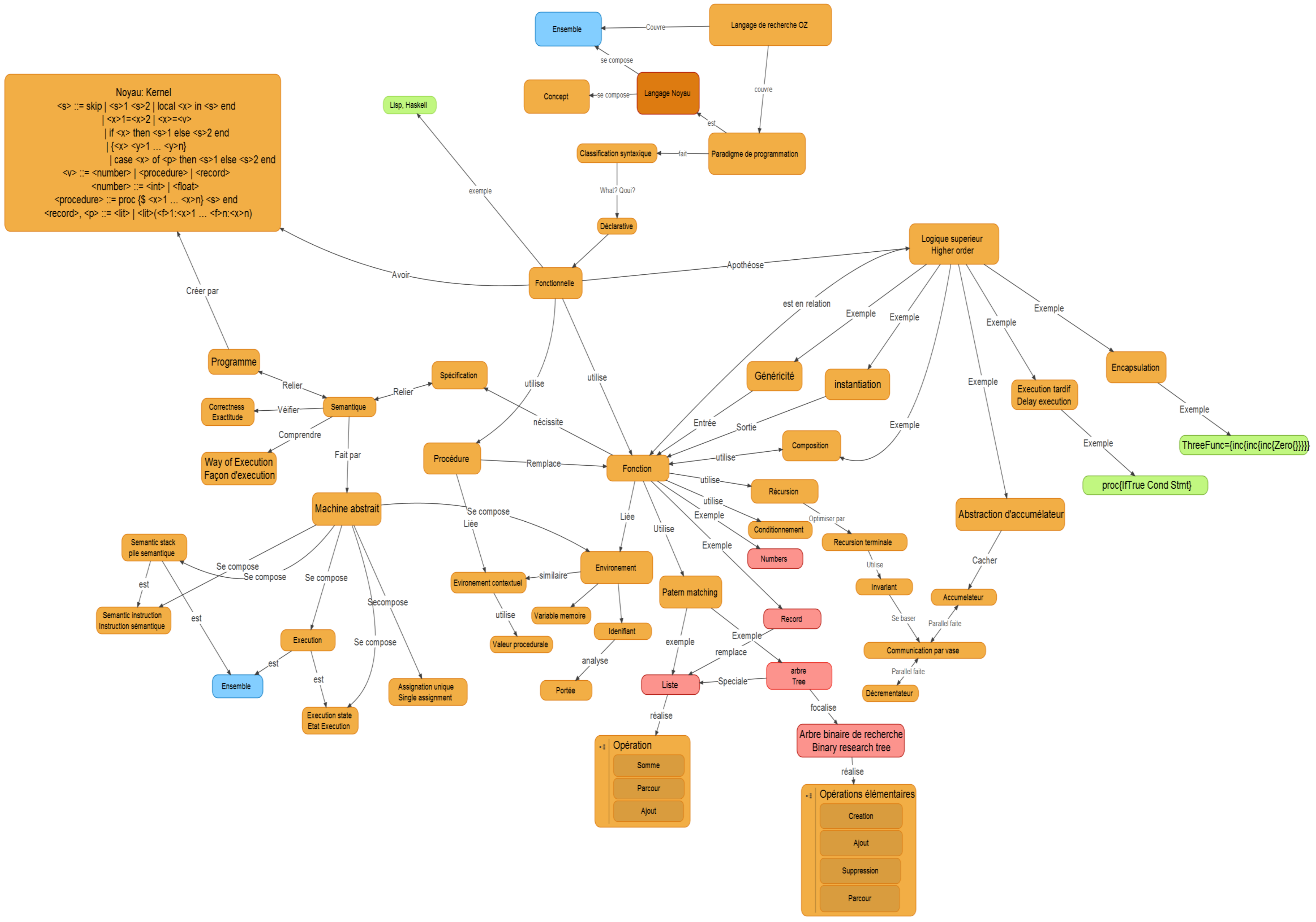
Faculty of Sciences & Technology

Mathematics & Computer Science Department

a.hariche@univ-dbkm.dz

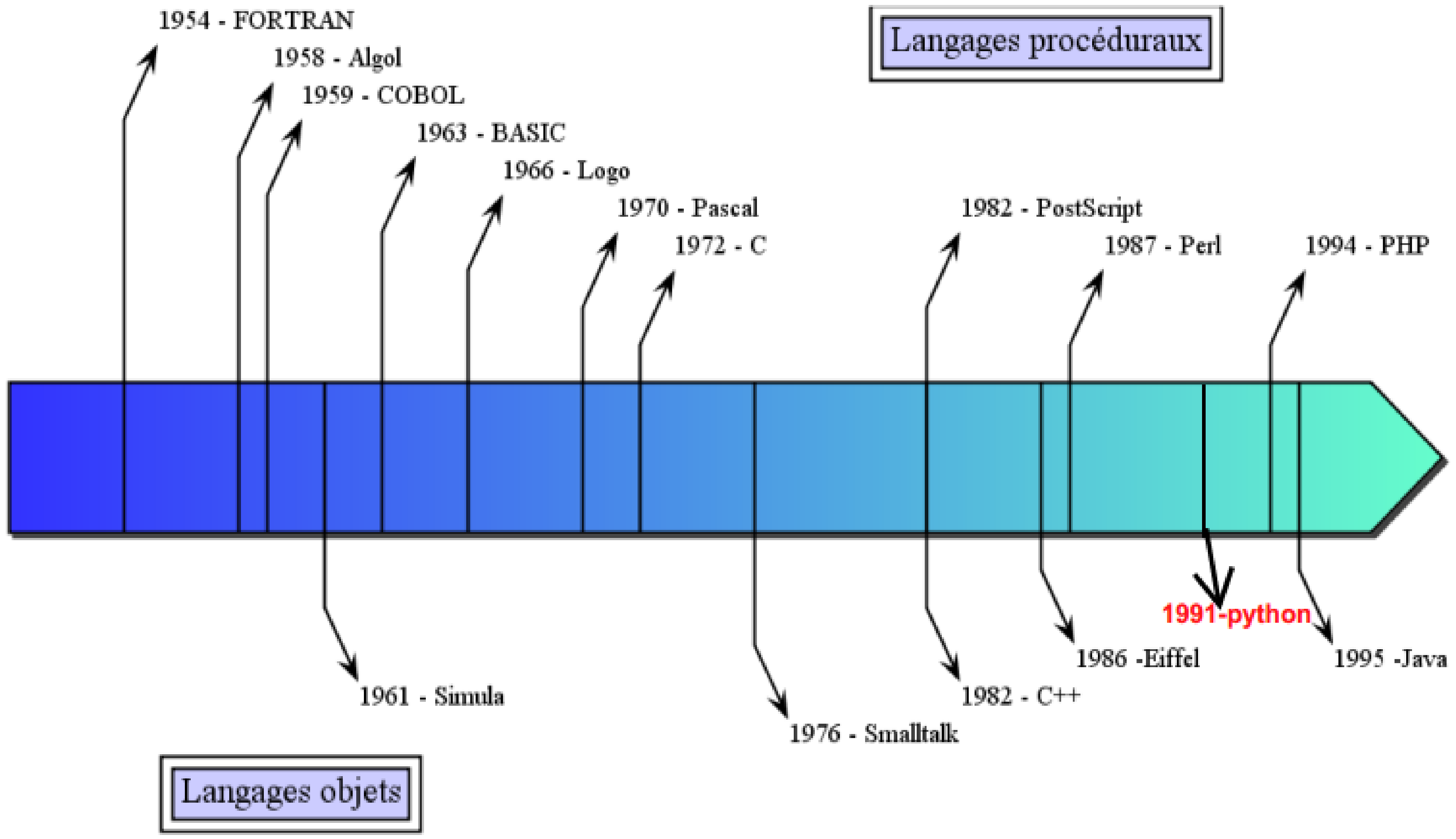

Noyau: kernel

```
<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)
```

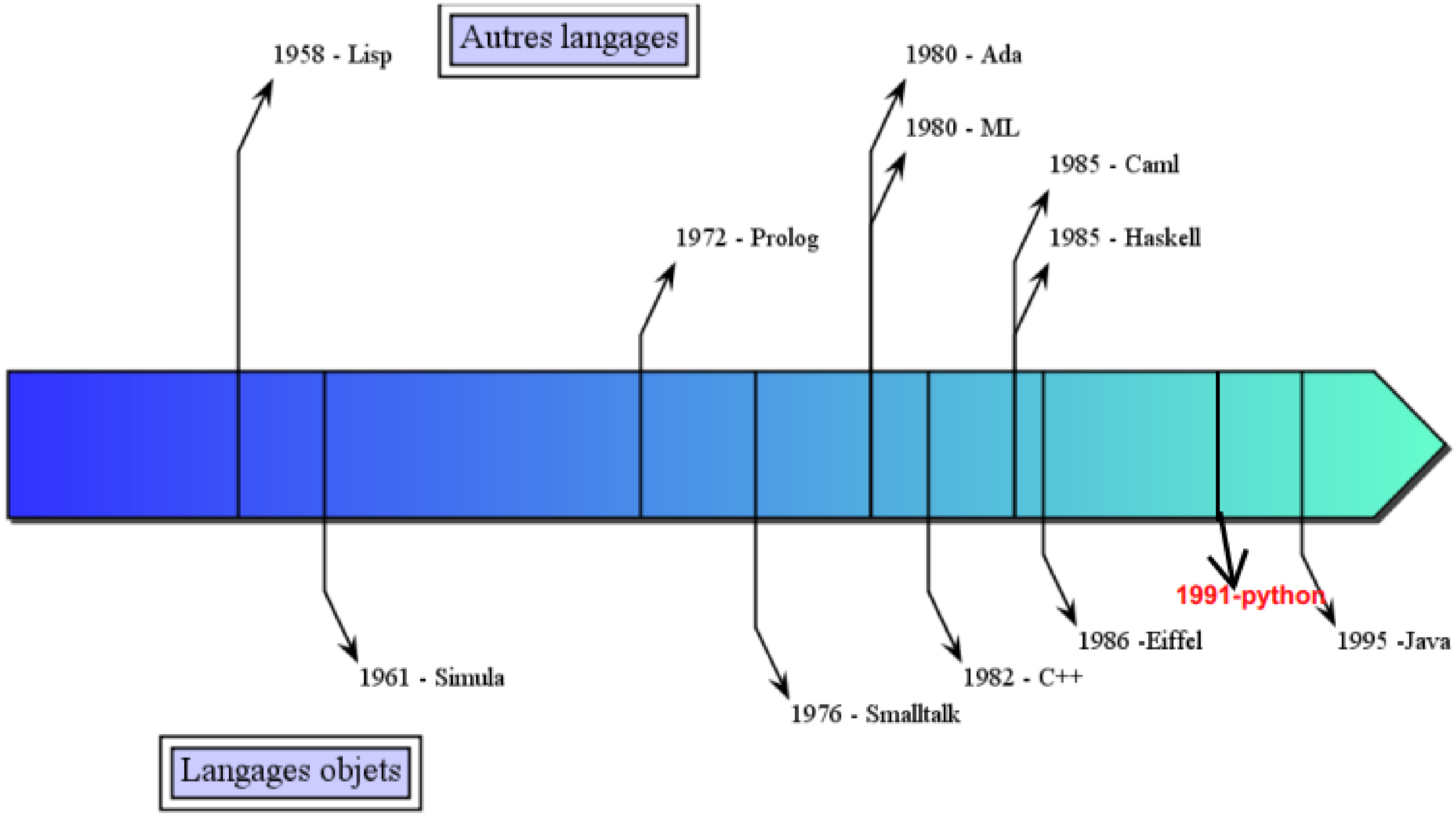


```
Noyau: Kernel
<s> ::= skip | <s>1 <s>2 | local <x> in <s> end
        | <x>1=<x>2 | <x>=<v>
        | if <x> then <s>1 else <s>2 end
        | {<x> <y>1 ... <y>n}
        | case <x> of <p> then <s>1 else <s>2 end
<v> ::= <number> | <procedure> | <record>
<number> ::= <int> | <float>
<procedure> ::= proc {$ <x>1 ... <x>n} <s> end
<record>, <p> ::= <lit> | <lit>(<f>1:<x>1 ... <f>n:<x>n)
```

Timeline of programming languages



Timeline of programming languages



Logic programming foundations?

- The logic programming is based on relations and not mapping:
- All logic programming languages restricted to Horn Clauses.
- Logic programming focuses on logical relation which could be true or false between tow sets $S \times T$:

$$\forall x \in S \wedge y \in T \cdot R(x, y) = \text{true} \vee R(x, y) = false \vee R(x, y) = none$$

- $R(x, y)$ could be seen as predicate.
- Relation can be unary binary or ternary.
- $R(x, y)$ could be seen as predicate.

Prolog :principales

- Prolog was created on 1980 by P.ROOSSEL inside Marseille university.
- Several distribution: PROLOG 1 , 2, 3, CPROLOG, visual PROLOG, EPILOG, Quintus turbo-PROLOG, XILOG, GOIDEL, **OZ** …etc..
- Prolog is used for :

✓ Interrogation of databases.
✓ Expert system realization.
✓ Native language comprehension.

- Prolog is limited to :

✓ No data-program distinction.
✓ No control structure.

- Prolog uses Backtracking technique.
- Executing on Prolog means finding a proof to an expression.
- Inter-object relation totally applied inside Prolog (invertibilty).

Prolog :concepts

- **Facts** : predicate(argument1, argument2,….)

Example: Male(Ali)
Father(Ali,Samia)

where **Predicate** is an affirmation that could be TRUE or FALSE.

- Arity: arguments counter.

Example : Male is unary fact with relation of 1 arity
Father is binary fact with a relation o arity equal to 2 objects.

- **Queries** and **Questions**: it can be launched as follow:

?-male(x)
X=Ali;

- ; means ask for next solution.
- « » means stop searching for solutions.
- Uppercase serve to variables when lowercase leads into canstants.

Prolog :concepts

- **Atomes** : alphanumerical strings that:
  - ✓ Start with a lowercase letter .
  - ✓ Strings between « » or ‘’.
  - ✓ Numbers
- **Variables** with an uppercase Xyz, A
- **Domaine** : separate between symbol (variables), integer(long), float …
- **Unification** : try to make two formulas identical by giving the value which contains.

Example: Father(X,X) and father(Omar,Ali) cannot be unified.

- **Substitution** : Giving argument on a formula its corresponding terms.

We note $\{X_1 = Value_1 \cdots X_n = Value_n\}$

A term **A** is said to be an **instance** of **B** if there is a substitution from **A** to **B**.

Example: Male(Omar) and Male(Ali) are instance of Male(x)

$\{X = Omar\}, \{X = Ali\}$ are corresponding substitutions.

Prolog :concepts

- **Shared variables** : Same variable with two arguments and still give the same value:

✓ Can be used for the conjuction of predicates of a same query .

Example: -?Father(X,X) will reach no goal.

-?Father(X,Y)Male(X) search all sons of the father Y.

- **Clauses** or **Rules:** Made to express **conjunctions of goals**.

✓ General form: $< Head >:- -C_1, C_2, C_3 \cdots C_n$

✓ :-- means « if »

✓ , means « and »

✓ ; means « or »

✓ $C_1, C_2, C_3 \cdots C_n$ Are sub-goals to demonstrate the head of the rule, it is necessary to demonstrate all its sub-goals.

✓ Otherwise the head is true if and only if the sub-goals are true

✓ This rule can also be called **Horn clause**.

✓ Rules can be recursives.

Example: GrandFather(x, y) :-- Father (x,z) ,Father (z, y).

Application Example : Family tree.

Prolog :Data-structure

●●**Types**: Prolog supports as types:

✓ *Standard* integer, float, char, symbol.

✓ *Compound* (see also data structure).

●● **Variables:** Follow this regular expression:

✓ $[A-Z]\{1\}|\_+[a-zA-Z]^{+}|[0-9]^{+}|\_$

✓ The scope of variables is global.

Example: Ax,Bz_nA3 are two variables.

●● **Constants:** Follow this regular expression:

✓ Numbers|Sybmol

✓ The scope of constants is local (limited by a clause).

Example:123456,ahmed, ‘Ali123@*/’ are constants

●● **Lists:** A list **L** whose elements are of type **type** is defined as follow:

L type* (Domains)

✓ [] is for representing an empty list.

✓ [a,b,c] is list of three char elements.

✓ [X!Y] is a list where X is car and Y is the cdr

✓ [a,b,c]=[a![b,c]]=[a,b![c]]=[a,b,c![]]

●●**Structures**: use the keyword Domain:

Example: datetype=date(integer,integer,integer)

Prolog :the program

Domains

```
New types constructors
```

Predicates

```
Types of related objects
```

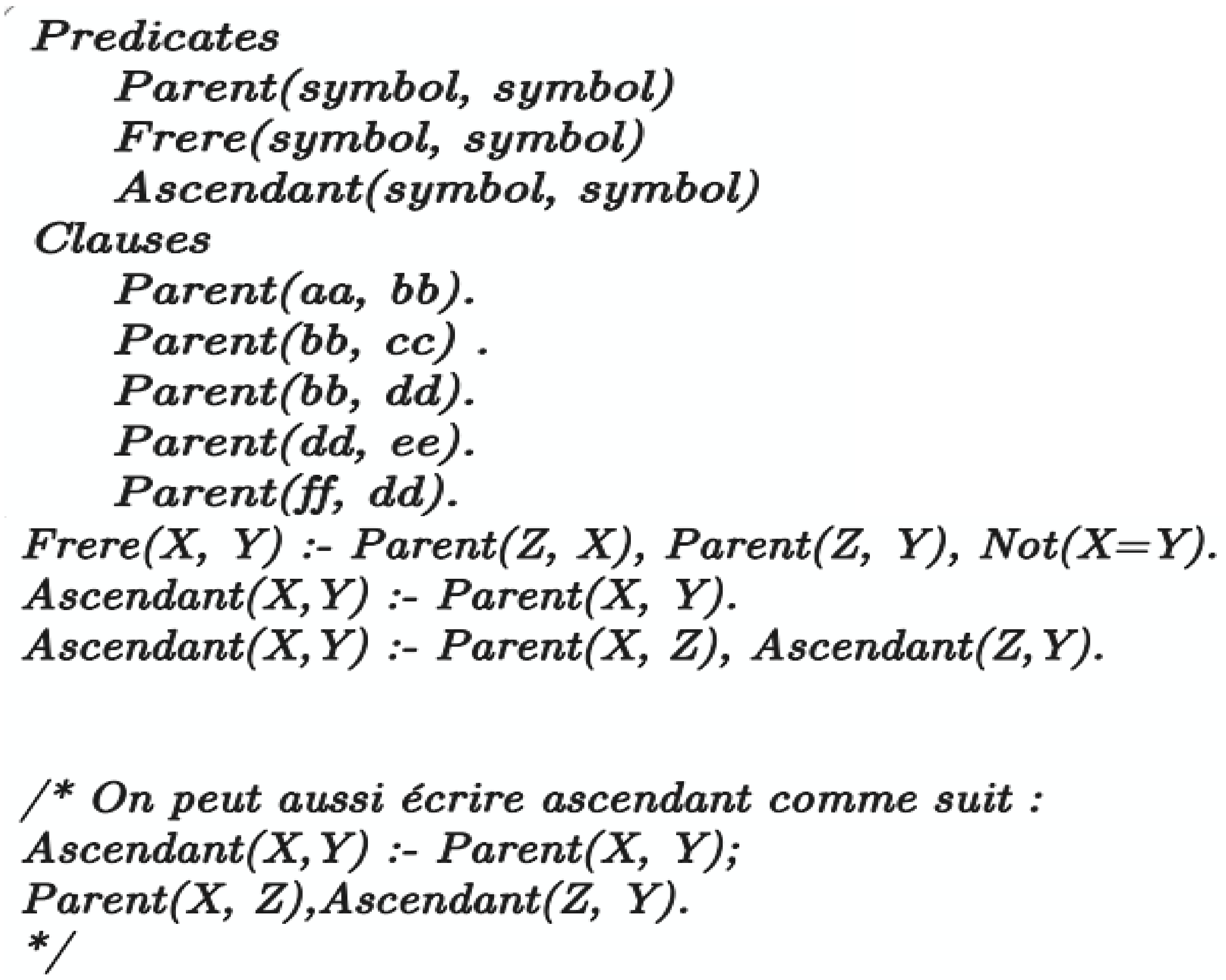
Clauses

```
Facts and clauses
```

Goal

```
The targetting goals
```

Prolog :Practical example



```
Predicates
      Parent(symbol, symbol)
      Frere(symbol, symbol)
      Ascendant(symbol, symbol)
Clauses
      Parent(aa, bb).
      Parent(bb, cc) .
      Parent(bb, dd).
      Parent(dd, ee).
      Parent(ff, dd).
Frere(X, Y) :- Parent(Z, X), Parent(Z, Y), Not(X=Y).
Ascendant(X,Y) :- Parent(X, Y).
Ascendant(X,Y) :- Parent(X, Z), Ascendant(Z,Y).


/* On peut aussi écrire ascendant comme suit :
Ascendant(X,Y) :- Parent(X, Y);
Parent(X, Z),Ascendant(Z, Y).
*/
```

Prolog :Practical example

Question : frere(dd, cc)

- ✓ *Réponse Yes*

Question : frere(X, dd)

- ✓ *Réponse X = cc*

Question : frere(dd, X)

- ✓ *Réponse X = cc*

Question : Ascendant(X, ee)

- ✓ *Réponse*

  X = dd
  X = aa
  X = bb
  X = ff

Question : Ascendant(X, cc)

- ✓ *Réponse*

  X = bb
  X = aa

Why to do Prolog-style logic programming in Oz?

- The Oz language is the result of a decade of research into programming based on logic:
- **Search-based logic programming (such as Prolog, CHIP, and cc“FD”)** and **committed-choice (concurrent)** and **logic programming(including at languages such as Parlog, FCP, and FGHC Concurrent Prolog and GHC)** with deep guards.
- Oz provides new abilities, such as first-class top levels and constant-time merge.
- Oz use interactive graphic tools *the Browser* and *the Explorer* useful for developing and running logic.
- The real strengths of Oz is *constraint programming* and *distributed programming*.
- Oz is equal to or better than any other system existing in the world today for Example compute-intensive constraint problems Oz provides parallel search engines that can be used transparently .i.e. without changing the problem specification.

How to do Prolog-style logic programming in Oz?

- Focuses on logic programming for general-purpose applications:
- Manipulating structured data according to logical rules (**rule-based expert systems or compilers)**.
- It scratches the surface of the real strengths of Oz with two truemendous areas: **constraint programming (**finite domains, finite sets of integers and natural language processing**)** and **distributed programming**.
- Deterministic logic programming .i.e. sequential logic programming without search for examining data structures with Browser.
- Extends this to nondeterministic logic programming, i.e., including search for the exploration of the search tree using first-class top levels with the Explorer.
- Committed-choice logic programming and search-based are combined.
- Oz Logic programming kernel language.
- Some glimpses in to constraints and distribution in Oz.

Deterministic Logic Programming

- Supported by three statements: **if** , **case** , and **cond**.
- Example : defines an *Append* **predicate** that can b e used to app end t wo lists.

```
declare
proc {Append L1 L2 L3}
case L1
of nil then L2=L3
[] X|M1 then L3=X|{Append M1 L2}
end
end
```

$$\begin{aligned}&\forall l_1, l_2, l_3 : \text{append}(l_1, l_2, l_3) \leftrightarrow \\ &\quad l_1 = \text{nil} \wedge l_2 = l_3 \vee \\ &\quad \exists x, m_1, m_3 : l_1 = x|m_1 \wedge l_3 = x|m_3 \wedge \text{append}(m_1, l_2, m_3)\end{aligned}$$

- In a **case** statement, variables in the branches of the **case** (like X and M1) are declared implicitly and their scope covers one branch of the case.
- The definition has a precise logical semantics in addition to its operational semantics(as it is presented in the red box)

Deterministic Logic Programming :Prolog Vs OZ

- Variable declaration and variable scope are defined quite differently in Oz and Prolog:
- **In Prolog:**
- Both **declaration** and **scope** are defined ***implicitly*** through the *clausal* syntax.
- Namely, variables in the head are universal over the whole clause and new variables in the body are existential over the whole clause body.

- **In Oz:**
- **Declaration** and **scope** are defined ***explicitly***. The scope is restricted to the statement in which the declaration occurs.
- This is important because Oz is fully compositional (all statements can be nested). Oz has syntactic support to make the explicit declarations less verbose.

- Elements of **Oz** lists are not separated by commas as in **Prolog**.

Deterministic Logic Programming: the power of OZ

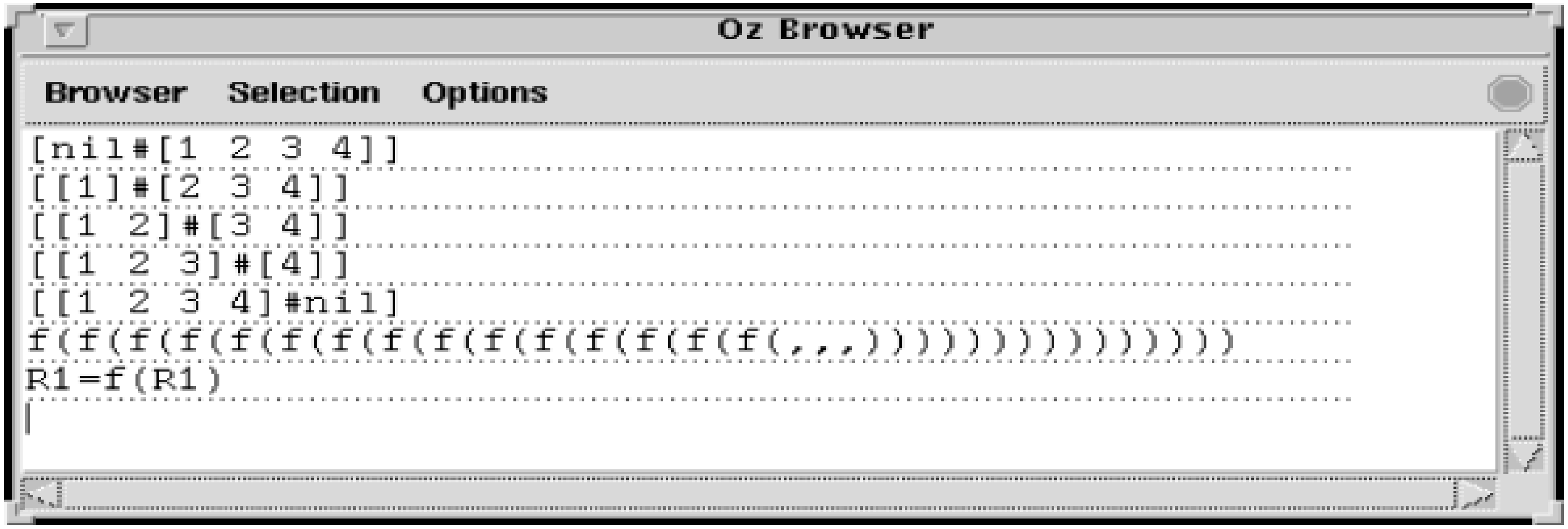
- This code displays the output of ***Append*** in the ***Browser*** which is fully concurrent and it can display any number of data structures simultaneously (see Figure).

```
declare A in
{Append [1 2 3] [4 5 6] A}
{Browse A}
```

```
{Browse {Append [1 2 3] [4 5 6]}}
```

- The display of a data structure containing unbound variables is up dated when one of the variables is bound.
- The Browser has options to let it either ignore sharing or display sharing. In the second case, shared sub-terms (including cycles) are displayed only once.
- A nondeterministic ***Append*** can be shown as the follow two definitions: **declare** X **in** {Browse X=f(X)}
- The first displays X as a tree, stopping at the default depth limit of 15.
- The second displays X as a minimal graph, which makes all cycles and sharing explicit.

Deterministic Logic Programming: the power of OZ



```
[nil#[1 2 3 4]]
[[1]#[2 3 4]]
[[1 2]#[3 4]]
[[1 2 3]#[4]]
[[1 2 3 4]#nil]
f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(,,,)))))))))))))))
R1=f(R1)
```

Figure 1: Screen shot of the Oz Browser.

- A nondeterministic ***Append*** can be shown as the follow two definitions: **declare** X **in** {Browse X=f(X)}
- The first displays X as a tree, stopping at the default depth limit of 15.
- The second displays X as a minimal graph, which makes all cycles and sharing explicit.

Non-deterministic Logic Programming (dis,choice)

- Supported by two concepts: disjunctions (***dis*** and ***choice***) and first-class top levels.
- Example : defines an nondeterministic version of *Append* **predicate**

```
declare
proc {FullAppend L1 L2 L3}
dis L1=nil L2=L3
[] X M1 M3 in
L1=X|M1 L3=X|M3 {FullAppend M1
L2 M3}
end
end
```

⇔

Both *Append* and *FullAppend* have exactly the same logical semantics. They differ only in their operational semantics. If used inside a top level, *FullAppend* gives results in cases where Append blocks.

- The **dis** statement is a determinacy-directed disjunction. The “X M1 M3 in “declares variables for the second branch of the disjunction.
- The choice statement is more primitive than dis ; it generates a choice point immediately for all its clauses without checking if any clauses fail.

Non-deterministic Logic Programming (First class Top Level)

```
{Browse {FullAppend [1 2] [3 4]}} % Shows [1 2 3 4] Deterministic
{Browse {FullAppend X Y [1 2 3 4]}} % Blocks Non-deterministic
```

- To get an answer, you need to execute the nondeterministic call in a top level. Here's how to create a top level with a given query.
- Example : defines an nondeterministic version of *Append* **predicate with top levels and query.**

```
declare
proc {Q A}
X Y
in
{FullAppend X Y [1 2 3 4]} A=X#Y
end
S={New Search.object script(Q)}
```

The `search.object` class makes it possible to create any number of **top levels**. The top levels are accessed like objects, can run concurrently, and can b e passed as arguments and stored in data structures. Top level are ***first class***.
In this example, procedure **Q** contains the query. The procedure's output is **A** , i.e., the **pair X#Y**

- Each **top level** is initialized with a **query**. The query is entered as a one-argument procedure called a ***script***.

Non-deterministic Logic Programming (First class Top Level)

```
{declare
S={New Search.object script(
proc {$ A} X#Y=A in {FullAppend X Y [1 2 3 4]} end)}
```

- This exploits two syntactic short-cuts. First, **the “$ " is a nesting marker** that implicitly **declares the variable Q**.
- Second, putting an equation left of in implicitly declares all variables of the equation's left-hand side. That is,”**X#Y=A in " declares X and Y , creates the pair X#Y , and unifies the pair with A**.
- You can get answers one by one I

```
{Browse {S next($)}}
```

```
[nil#[1 2 3 4]]
[[1]#[2 3 4]]
[[1 2]#[3 4]]
[[1 2 3]#[4]]
[[1 2 3 4]#nil]
nil
```

- . This is similar to the semicolon “ **;** " in an interactive Prolog session. It is not idenical, since the next must be called to get the first answer..

Non-deterministic Logic Programming (First class Top Level)

- The Oz discuss top levels with a few random remarks:

- Creating a new top level is very cheap; you should not hesitate to do so for each query.

- A program can consist of deterministic and nondeterministic predicates used together in any way. A top level script can call such a program; this is possible because both deterministic and nondeterministic predicates have logical semantics. Of course, only the nondeterministic predicates can create ***choice*** points.

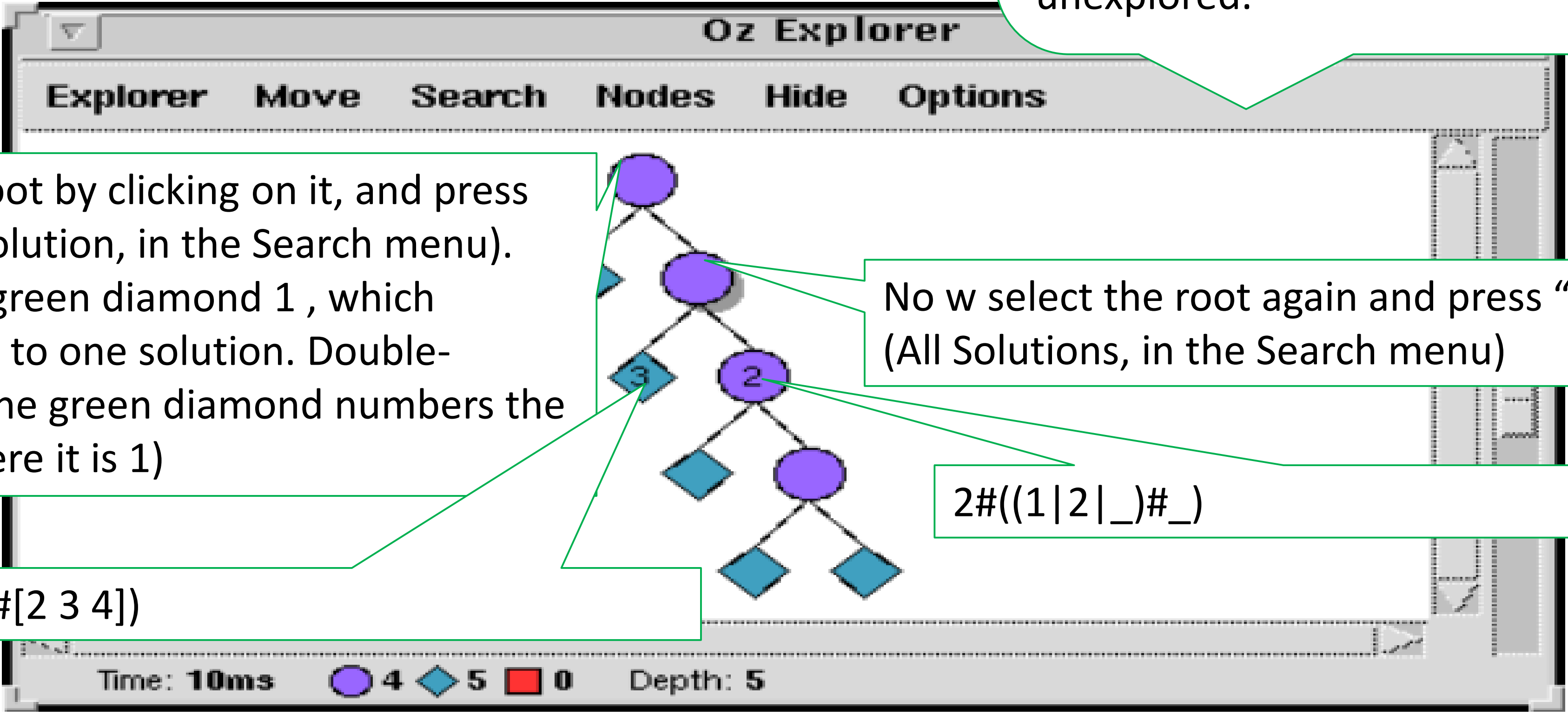
- It is easy to add information to an existing top level while it is active. It suffices for the script to have an external reference, i.e., to have a reference to something outside of the top level..

Non-deterministic Logic Programming (OZ explorer)

```
{Explorer.object script(
proc {$ A} X#Y=A in {FullAppend X Y [1 2 3 4]} end)}
```

- the Explorer, a graphic tool for interactive tree.

This opens a window that displays the search tree. Initially, just the root is displayed, as a gray circle. The circle means that the root has a choice point. The gray color means that the choice point is not fully explored. It is in fact completely unexplored.



Select the root by clicking on it, and press "n " (Next Solution, in the Search menu). This adds a green diamond 1 , which corresponds to one solution. Double-clicking on the green diamond numbers the diamond (here it is 1)

No w select the root again and press "a " (All Solutions, in the Search menu)

2#((1|2|_)#_)

3#([1]#[2 3 4])

Committed-choice Logic Programming

- The **case** and **if** statements are special cases of **cond**. which does a general don't-care choice, i.e., if the guard of any branch succeeds then execution can commit to that branch and discard all the others.
- Example : defines producer-consumer program with flow control **predicate**

```
declare
proc {Producer N L}
case L of X|Ls then X=N {Producer
N+1 Ls}
else skip end
end
fun {Consumer N L A}
if N>0 then X L1 in L=X|L1
{Consumer N-1 L1 A+X}
else A end
end
```

```
local L S in % Variable declaration
thread {Producer 0 L} end
thread S={Consumer 100000 L 0} end
{Browse S}
end
```

Committed-choice Logic Programming (Example)

- Example : defines producer-consumer program with flow control predicate

```
declare
proc {Producer N L}
case L of X|Ls then X=N {Producer
N+1 Ls}
else skip end
end
fun {Consumer N L A}
if N>0 then X L1 in L=X|L1
{Consumer N-1 L1 A+X}
else A end
end
```

```
local L S in % Variable declaration
thread {Producer 0 L} end
thread S={Consumer 100000 L 0} end
{Browse S}
end
```

The **producer** and **consumer** each runs in its own thread. The **producer** generates the list [0 1 2 3 ... and the **consumer** sums the list's first 100000 elements. The main thread immediately displays an unbound variable and later up dates the display to 4999950000 when the **consumer** terminates.

Committed-choice Logic Programming (If,case Vs Cond)

- Because only **case** and **if** are used, both the producer and the consumer have a precise logical semantics as well as an operational semantics. (This is not true for **cond** unless its conditions are mutually exclusive.)

- The if statement has the logical semantics $(c \wedge t) \vee (\neg c \wedge e)$ where ***c*** is the **boolean condition**, ***t*** is derived from the **then** part, and ***e*** is derived from the **else** part.

- The if statement has the following operational semantics. It waits until enough information exists to decide the **truth** or **falsity** of its **boolean condition**. A t that poin ***t***, it executes its **then** or **else** part.

Non-deterministic concurrent Logic Programming

- If a logic program has only a single thread and uses the **dis** statement to express nondeterminism, then its behavior is exactly like that of a **Prolog** program where the Prolog system is modified to do clause selection according to the Andorra principle. However, because of **concurrency** and **first-class top levels**, Oz lets you do much more.
- Example: two sequential logic programs. There is a design **choice** when running them, i.e., whether to put them in the **same top level** or in **different top levels**:
- If the programs are **independent**, e.g., two **independent queries** to a database, then they should b e run in **different top levels**. This ensures that each program gets a fair share of the processing power and that no wasted work is done

- If the programs are **dependent**. i.e., they are **cooperating** to solve one problem, then it is often best to run them in the **same top level**. This is not often used in logic programming, but it is very important for constraint Programming using a **thread** a **propagator** with **spaces** and never creates a choice point. This is so used in **finite domains**, **finite sets**, and **rational trees**.

OZ kernel for logic programming

```
⟨S⟩  ::=  ⟨C⟩

        |  if ⟨C⟩ then ⟨S1⟩ else ⟨S2⟩ end

        |  case X
             of f1(l11:Y11  ...  l1m:Y1m) then ⟨B1⟩
             ...
             [] fn(ln1:Yn1  ...  lnm:Ynm) then ⟨Bn⟩
           else ⟨S⟩ end

        |  cond
              ⟨G1⟩ then ⟨B1⟩ [] ...  []  ⟨Gn⟩ then ⟨Bn⟩
           else ⟨S⟩ end

        |  dis
              ⟨G1⟩ then ⟨B1⟩ [] ...  []  ⟨Gn⟩ then ⟨Bn⟩
           end

        |  choice ⟨S1⟩ [] ...  []  ⟨Sn⟩ end

        |  ⟨Spaces⟩
```

Logic programming with distributions & constraints?

- Oz was never in tended to be just a **Prolog** substitute. The main power of Oz is in **constraint programming** and **distributed programming**:
- Oz implements for **distributed programming** because
- it separates the aspects of language semantics and distribution structure .
- It is open and implemented by means of a network layer that contains a distributed algorithm for each type of language entity, as well as distributed garbage collection,
- It is fruitfully interactive with non-trivial fault tolerance abstractions within the language.

A.HARICHE a.hariche@univ-dbkm.dz

Logic programming with distributions & constraints?

- The **constraint** and **distribution** abilities of Oz can be combined, the Search module implements a parallel search engine that is very useful for compute-intensive constraint problems :
- giving it a list of machine names and a **script**.
- The parallelism is completely transparent, i.e., the problem is specified without any knowledge of whether it is executed in parallel or not,
- The same **script** can b e used with a **top level**, with the **Explorer**, and with a parallel search engine.

Logic paradigm briefly

- Prolog-style of doing logic programming in Oz not only but more.
- Introduce first-class top levels, concurrency, and the Browser and Explorer tools.
- Explore logic programming with Oz capabilities:
  - Explains how search-based and committed-choice logic programming with deep guards are integrated
  - Outlines how the logic programming support smoothly ties into constraint programming
  - The constraint and distribution abilities of Oz for current active research such as the one applied for constrain t debugging, fault tolerant and secure distributed execution

Many important ideas

Louv1.1x

- Identifiers and environments
- Functional programming
- Recursion
- Invariant programming
- Lists, trees, and records
- Symbolic programming
- Instantiation
- Genericity
- Higher-order programming
- Complexity and Big-O notation
- Moore’s Law
- NP and NP-complete problems
- Kernel languages
- Abstract machines
- Mathematical semantics

Louv1.2x

- Explicit state
- Data abstraction
- Abstract data types and objects
- Polymorphism
- Inheritance
- Multiple inheritance
- Object-oriented programming
- Exception handling
- Concurrency
- Nondeterminism
- Scheduling and fairness
- Dataflow synchronization
- Deterministic dataflow
- Agents and streams
- Multi-agent programming

PLP_Drive space

https://drive.google.com/drive/folders/1YBCIZzAldeiT19DIfDiREQwP-NAQ1qMN