MI-GLSD-M1 -UEM213 : Programming paradigms

Chapter III:Functional paradigm
Basic concepts

A. HARICHE

University of Djilali Bounaama, Khemis Meliana (UDBKM)

Faculty of sciences & technology

Mathematics & computer sciences department

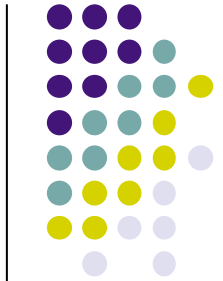
a.hariche@univ-dbkm.dz

Our first paradigm

- Functional programming
  - It is one of the simplest paradigms
  - It is the foundation of all the other paradigms
  - It is a form of *declarative programming*

- Our approach to functional programming
  - It is our first introduction to programming concepts
  - It is our first introduction to a kernel language
  - We use it to explain invariants and recursion
  - We give examples using integers, lists, and trees
  - We present higher-order programming: the apotheosis
  - We give a formal semantics based on the kernel language

Declarative programming: the long-term view



- Declarative programming is a vision for the future
  - Just say what result you want (give properties of the result)
  - Let the computer figure out how to get there
  - Declarative versus imperative: *properties* versus *commands*
- How do we make this vision real
  - Programming gets more support from the computer
  - With same programming effort, we can do more
- The whole history of computing is a progression toward more declarative
  - And faster and cheaper (all three are connected)

Declarative programming: the short-term view

- Declarative programming is the use of mathematics in programming (such as functions and relations)
  - A computation calculates a function or a relation
  - Use the power of mathematics to simplify programming (such as confluency and referential transparency)

- Very common in practice
  - Functional languages: LISP, Scheme, ML, Haskell, OCaml, ...
  - Logic languages (relational): SQL, constraint programming, Prolog, ...
  - Combinations: XSL (formatting), XSLT (transforming), …

- Also called "programming without state"
  - Variables and data structures can't be updated
  - Testing and verification is much simplified
  - Declarative versus imperative: *stateless* versus *stateful*

Key advantage of functional programming

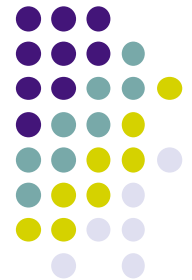
- "A program that works today will work tomorrow"
  - Functions don't change
  - All changes are in the arguments, not in the functions

- It is a programming style that should be encouraged in all languages
  - "Stateless server" for a client/server application
  - "Stateless component" for a service application

"Cookies" on the Web

- Learning functional programming helps us think in this style
  - All programs written in the functional paradigm are ipso facto declarative: an excellent way to learn to think declaratively

Now let's start programming...



- This completes the « philosophical » introduction of the course
- Now we will start programming in our first paradigm
  - Functional programming
- At the same time, we will introduce the Oz language and the Mozart system
  - Mozart's emacs interface, which we will use throughout the course

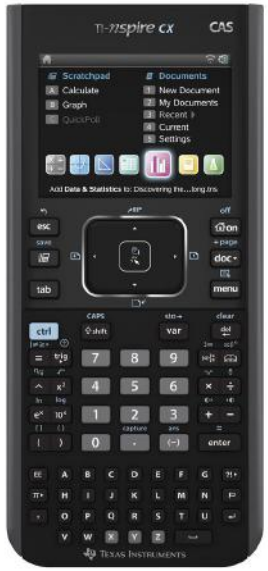
Interactive system

declare

X = 1234 * 5678

{Browse X}

- Select a region in the Emacs buffer
- Feed the region to the system
  - The text is compiled and executed
- Interactive system can be used as a powerful calculator →



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

Creating variables

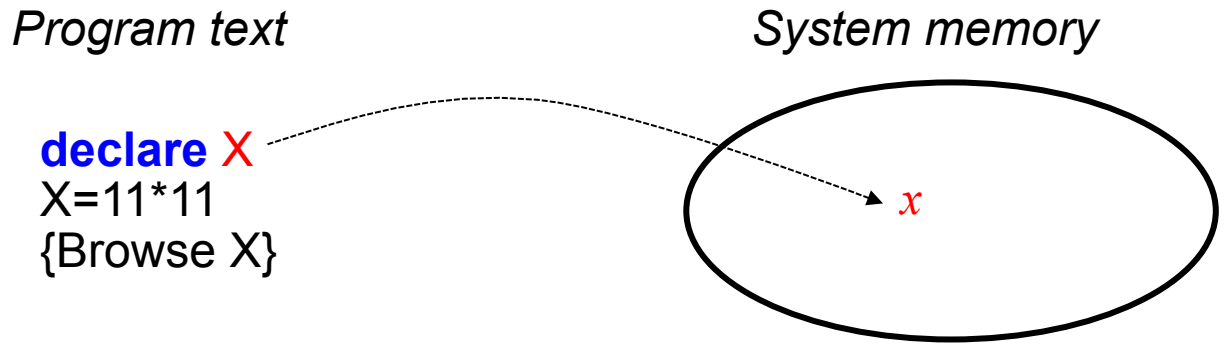
declare

X = 1234 * 5678

{Browse X}

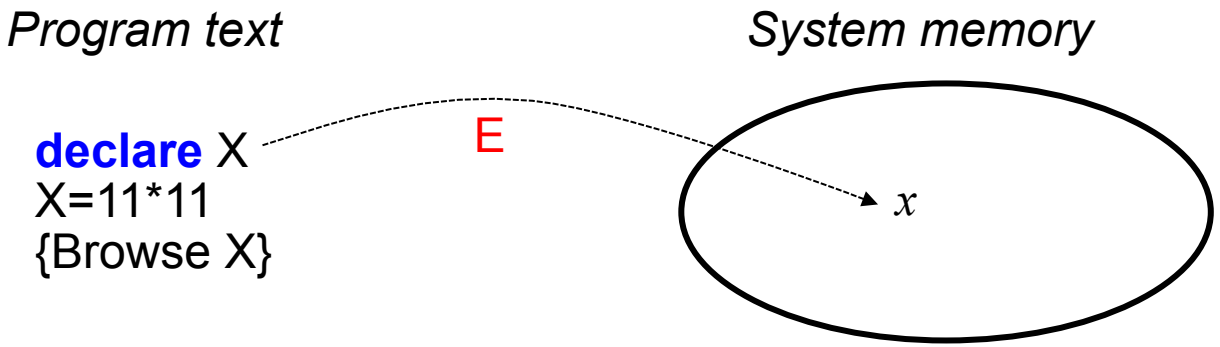
- **Declare** (create) a variable **designated** by X
- **Assign** to the variable the value 7006652
  - Result of the calculation 1234*5678
- **Call** the procedure Browse with the argument designated by X
  - Opens a window that displays 7006652

Variable and identifier



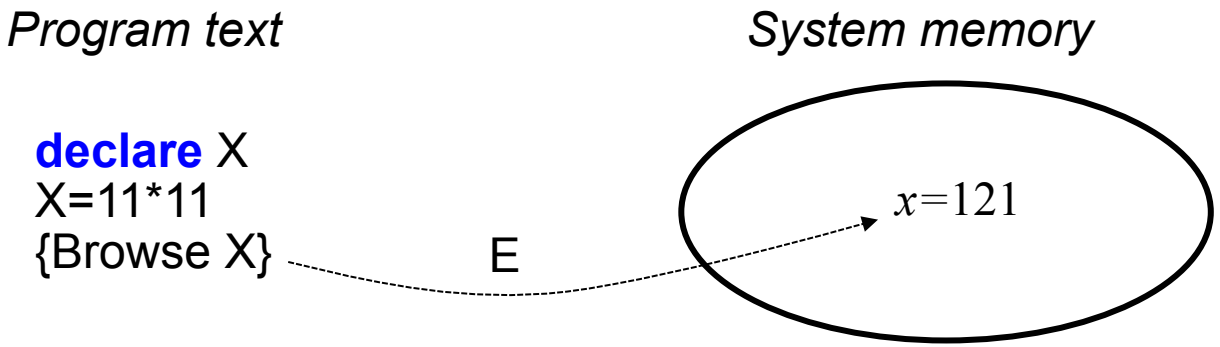
- There are two concepts hiding in plain view here
  - Identifier X : what you type (character sequence starting with capital)
    Var, A, X123, FirstCapitalBank
  - Variable x : what is in memory (used to store the value)
- Variables are short-cuts for values (= constants)
  - Can only be assigned to one value (like mathematical variables)
  - Multiple assignment is another concept! We will see it later in the course.
  - The type of the variable is only known when it is assigned (dynamic typing)

Environment



- **declare** is an interactive instruction
  - Creates a new variable in memory
  - Links the identifier and its corresponding variable
- Third concept: environment E={X→ x}
  - A function that takes an identifier and returns a variable: E(X) = x
  - Links identifiers and their corresponding variables (and the values they are bound to)

Assignment



- The assignment instruction X=121 binds the variable x to the value 121

Single assignment

- A variable can only be bound to one value
  - It is called a single-assignment variable
  - Why? Because we are in the functional paradigm!

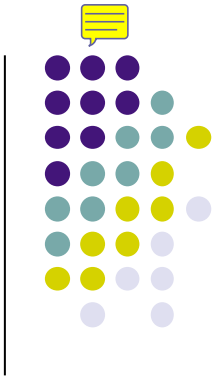
- Incompatible assignment: signals an error

  X = 122

- Compatible assignment: accepted

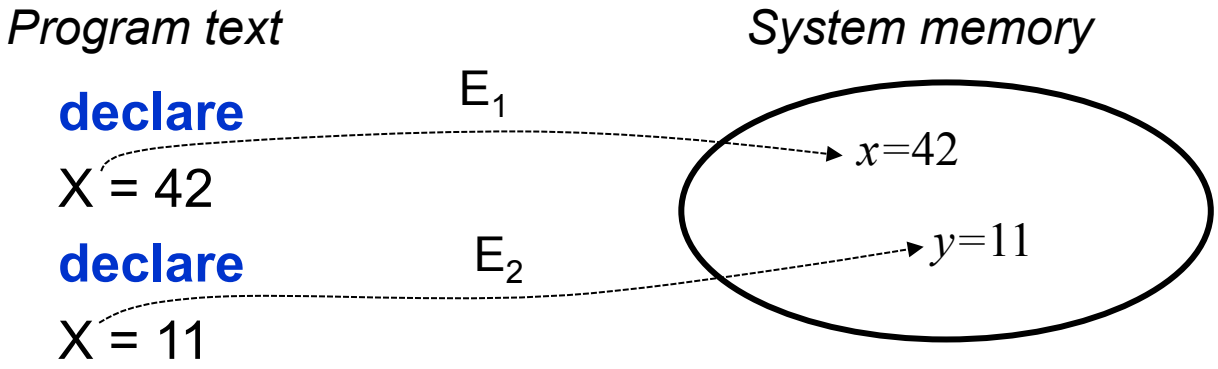
  X = 121

Why single assignment?



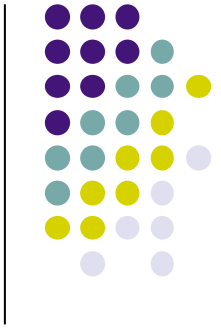
- Why do we restrict variables to be bound to one value?
  - It seems like a big handicap, not being able to assign again
- We do it because it gives advantages!
  - It's like following a law. Why is it a good idea to respect traffic rules? Because (among other things) it reduces the chance of having an accident.
- If we could assign more than once, we could break a correct program
  - But how can we program without multiple assignment? Actually, it's easy, as we will see.

Redeclaring an identifier



- An identifier can be redeclared
  - The same identifier refers to a different value
  - There is no conflict with single assignment.
    Each occurrence of X corresponds to a different variable.
- The interactive environment always has the last declaration
  - **declare** keeps the same correspondance until redeclared (if ever)
  - In this example X will refer to 11

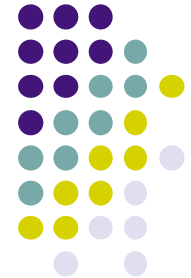
Scope of an identifier occurrence



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

- The instruction
  local X **in** <stmt> **end**
  declares X between **in** and **end**
- The scope of an identifier occurrence is that part of the program text for which the occurrence corresponds to the same variable declaration
- The scope can be determined by inspecting the program text; no execution is needed. This is called lexical scoping or static scoping.
- Why is there no conflict between X=42 and X=11, even though variables are single assignment?
- What will the third Browse display?

Tips on Oz syntax



- At this point you can see that Oz syntax is not like most syntaxes you may have seen before
- The most popular syntax in mainstream languages (C++, Java) is « C-like », where identifiers are statically typed (« int i; ») and can start with lowercase, and code blocks are delimited by braces
  - Oz syntax is definitely not C-like!
- Oz syntax is inspired by many languages: Prolog (logic programming), Scheme and ML (functional programming), C++ and Smalltalk (object-oriented programming), and so forth

Why is Oz syntax different?

- It is different because Oz supports many programming paradigms
  - The syntax is carefully designed so that the paradigms don't interfere with each other
  - It's possible to program in just one paradigm. It's also possible to program in several paradigms that are cleanly separated in the program text.
- So it is important for you not to get confused by the differences between Oz syntax and other syntaxes you may know
- Let me explain the main differences so that you will not be hindered by them

Main differences in Oz syntax

- Identifiers in Oz always start with an uppercase letter
  - Examples: X, Y, Z, Min, Max, Sum, IntToFloat.
  - Why? Because lowercase is used for symbolic constants (atoms).
- Procedure and function calls in Oz are surrounded by braces { ... }
  - Examples: {Max 1 2}, {SumDigits 999}, {Fold L F U}.
  - Why? Because parentheses are used for record data structures.
- Local identifiers are introduced by **local** ... **end**
  - Examples: **local** X **in** X=10+20 {Browse X} **end**.
  - Why? Because all compound instructions in Oz start with a keyword (here « **local** ») and terminate with **end**.
- Variables in Oz are single assignment
  - Examples: **local** X Y **in** X=10 Y=X+20 {Browse Y} **end**.
  - Why? Because the first paradigm is functional programming. Multiple assignment is a concept that we will introduce later.

Oz syntax in the programming exercises

- Most programming bugs, at least early on, are due to syntax errors (such as using a lowercase letter for an identifier)
- Please take into account the four main differences. Once you have assimilated them, reading and writing Oz will become straightforward.
- And now let’s do some more programming!

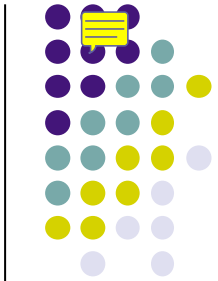
Functions

- We would like to execute the same code many times, each time with different values for some of the identifiers
  - To avoid repeating the same code, we can define a function
- Functions are shortcuts for program code to execute, just as variables are shortcuts for values
  - To be precise, functions are just another kind of value in memory, like numbers (as we will see later)
- Function Sqr returns the square of its input:

```
declare
fun {Sqr X} X*X end
```

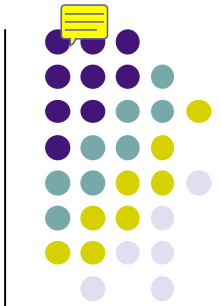
- The **fun** keyword identifies the function. The identifier Sqr refers to a variable that is bound to the function.

Numbers



- There are two kinds of numbers in Oz
  - Exact numbers: integers
  - Approximate numbers: floating point
- Integers are exact (arbitrary precision)
- Floats are approximations of real numbers (up to 15 digits precision – 64-bit internally)
- There is never any automatic conversion from exact to approximate and vice versa
  - To convert, we use functions IntToFloat or FloatToInt
  - Design principle: don’t mix incompatible concepts

Sum of digits function



- Function SumDigits calculates the sum of digits of a three-digit positive integer:

```
declare
fun {SumDigits N}
   (N mod 10) + ((N div 10) mod 10) +
   ((N div 100) mod 10)
end
```

- **mod** and **div** are integer functions
- / (division) is a float function
- * (multiplication) is a function on both floats and integers

SumDigits6

- Sum of digits of a six-digit positive integer

```
fun {SumDigits6 N}
   {SumDigits (N div 1000)} +
   {SumDigits (N mod 1000)}
end
```

- This is an example of function composition: defining a function in terms of other functions
  - This is a key ability for building large systems: we can build them in layers, where each layer is built by a different person
  - This is the first step toward data abstraction

SumDigitsR

- Sum of digits of any positive integer (first try)

```
fun {SumDigitsR N}
   (N mod 10) + {SumDigitsR (N div 10)}
end
```

- This function calls itself with a smaller value
  - But it never stops: we need to make it stop!

SumDigitsR

- Sum of digits of any positive integer (correct)

```
fun {SumDigitsR N}
   if (N==0) then 0
   else
      (N mod 10) + {SumDigitsR (N div 10)}
   end
end
```

- This introduces the conditional (**if**) statement
- This is an example of function recursion: defining a function that calls itself
  - This is a key ability for building complex algorithms: we divide a complex problem into simpler subproblems (divide and conquer)

The functional paradigm

- The functional paradigm as we have introduced it now, with the ability to calculate with numbers, to define functions, to do function composition and recursion, and to use the conditional statement (**if**), is a fully capable programming language

- We say it is Turing complete, since it can compute the same functions as a Turing machine
  - Since a Turing machine is the most powerful computer we know how to build (in terms of the kinds of functions that can be programmed), this means that we can do anything that any other computer can do

- We will see how to harness the power of recursion in the next two lessons (invariant programming and symbolic programming)
  - We will continue to harness the power of functions in the rest of the course (higher-order programming, data abstraction, concurrent programming)

Recursion and loops

- In the previous lesson we saw SumDigitsR:

```
fun {SumDigitsR N}
      if (N==0) then 0
      else (N mod 10) + {SumDigitsR (N div 10)} end
end
```

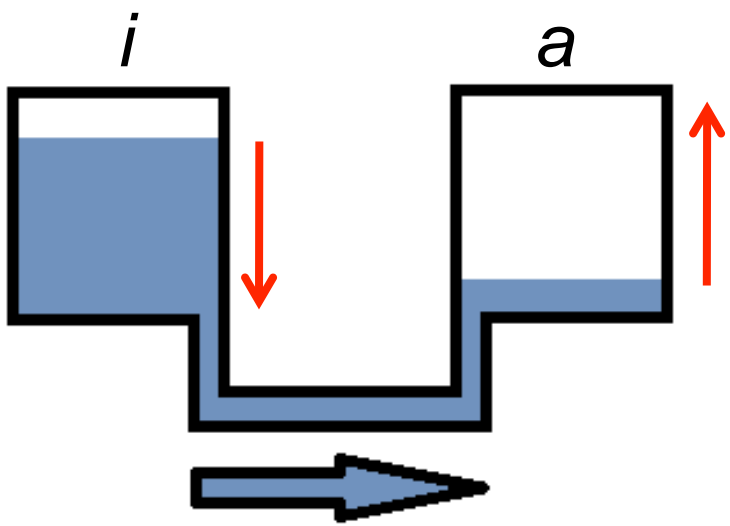
- The recursive call and the condition together act like a loop: a calculation that is repeated to achieve a result
  - Each execution of the function body is one iteration of the loop
- Recursion can be used to make a loop
  - In this lesson we will go to the root of this intuition

Invariant programming

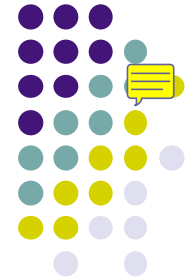
- A loop is a part of a program that is repeated until a condition is satisfied
  - Loops are an important technique in all paradigms
  - Loops are a special case of recursion, called *tail recursion*, where the recursive call is the last operation done in the function body
- We will give a general technique, invariant programming, to program correct and efficient loops
  - Loops are often very difficult to get exactly right, and invariant programming is an excellent way to achieve this
  - This applies to *both* declarative and imperative paradigms
- New concepts introduced in this lesson
  - Specification, accumulator, principle of communicating vases

Principle of communicating vases

- We invent a formula that splits the work into two parts:
  - *n! = i! * a*
- We start with *i=n* and *a=1*
- We decrease *i* and increase *a*, keeping the formula true
- When *i=0* then *a* is the result
- Here's an example when *n=4*:
  - 4! = 4! * 1
  - 4! = 3! * 4
  - 4! = 2! * 12
  - 4! = 1! * 24
  - 4! = 0! * 24



Sum of digits using invariant programming



- Each recursive call handles one digit
- So we divide the initial number *n* into its digits:
  - $n = (d_{k-1}d_{k-2}\cdots d_2 d_1 d_0)$ *(where d_i is a digit)*

- Let's call the sum of digits function *s(n)*
- Then we can split the work in two parts:
  - $s(n) = \underbrace{s(d_{k-1}d_{k-2}\cdots d_i)}_{s_i} + \underbrace{(d_{i-1} + d_{i-2} + \cdots + d_0)}_{a}$

- s_i is the work still to do and *a* is the work already done
- To keep the formula true, we set $i' = i+1$ and $a' = a+d_i$
- When $i=k$ then $s_k=s(0)=0$ and therefore *a* is the answer

Example execution

- Example with *n=314159*:

$$s(n) = s(d_{k-1}d_{k-2}\cdots d_i) + (d_{i-1} + d_{i-2} + \cdots + d_0)$$

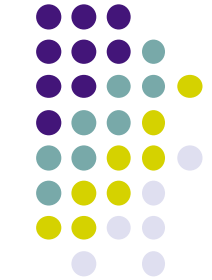
  - s(314159) = s(314159) + 0
  - s(314159) = s(31415) + 9
  - s(314159) = s(3141) + 14
  - s(314159) = s(314) + 15
  - s(314159) = s(31) + 19
  - s(314159) = s(3) + 20
  - s(314159) = s(0) + 23 = 0 + 23 = 23

Final program

- S = $(d_{k-1}d_{k-2}\cdots d_i)$
  A = $(d_{i-1} + d_{i-2} + \cdots + d_0)$

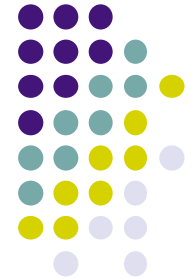
```
fun {SumDigits2 S A}
    if S==0 then A
    else
        {SumDigits2 (S div 10) A+(S mod 10)}
    end
end
```

What can we learn from these examples?



- We have now seen two examples of recursive functions
  - Factorial
  - Sum of digits
- For each example we have seen two versions
  - A version based on a simple mathematical definition
  - A version designed with invariant programming
- The second version has two interesting properties
  - It has *two* arguments; one of the two is an accumulator
  - The recursive call is the last operation in the function body (tail recursion)

The importance of tail recursion



- Let us now take a closer look at why tail recursion is important
- We will do a detailed comparison of the execution of Fact1 and Fact2
  - (This comparison is a first step toward the semantics given in lesson 6)
- We will see why Fact2 (with tail recursion) is more efficient than Fact1 (no tail recursion)
  - Fact1 is based on a simple mathematical definition
  - Fact2 is designed with invariant programming

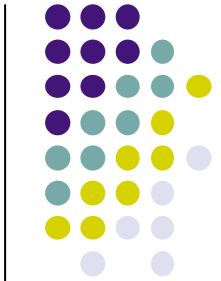
Comparing Fact1 and Fact2

- Tail recursion is when the recursive call is the last operation in the function body
- N * {Fact1 N-1} % No tail recursion

  After Fact1 is done, we must come back for the multiply.
  Where is the multiplication stored? On a stack!

- {Fact2 I-1 I*A} % Tail recursion
  The recursive call does not come back!
  All calculations are done *before* Fact2 is called.
  No stack is needed (memory usage is constant).

Stack explosion in Fact1

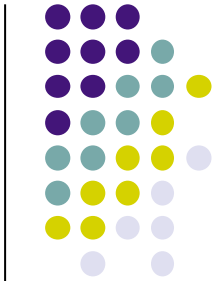


- 10 * {Fact1 10-1} ⇒
  10 * (9 * {Fact 9-1}) ⇒
  10 * (9 * (8 * {Fact 8-1})) ⇒
  ...
  10 * (9 * (8 * (7 * (6 * (5 * (...(1 * {Fact 0})...) ⇒
  10 * (9 * (8 * (7 * (6 * (5 * (...(1 * 1)...) ⇒
  ...
  3628800

Each line does one computation step

- {Fact2 10-1 10*1} ⇒
  {Fact2 9-1 9*10} ⇒
  {Fact2 8-1 8*90} ⇒
  ...
  {Fact2 1-1 1*3628800}

Comparing functional and imperative loops



- A while loop in the functional paradigm:

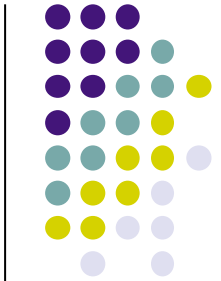
```
fun {While S}
    if {IsDone S} then S
    else {While {Transform S}} end /* tail recursion */
end
```

- A while loop in the imperative paradigm:
  (in languages with multiple assignment like Java and C++)

```
state whileLoop(state s) {
    while (!isDone(s))
        s=transform(s); /* assignment */
    return s;
}
```

- In *both* cases, invariant programming is an important design tool

Summary and a bigger example



- We summarize this lesson in a few sentences
  - A recursive function is equivalent to a loop if it is tail recursive
  - To write functions in this way, we need to find an accumulator
  - We find the accumulator starting from an invariant using the principle of communicating vases
  - This is called invariant programming and it is the only reasonable way to program loops
  - Invariant programming is useful in all programming paradigms
- Now let's tackle a bigger example!

A bigger example: calculating X^N

- Let's use invariant programming to define a function {Pow X N} that calculates X^N ($N \geq 0$)
- Let's start with a naive definition of x^n:

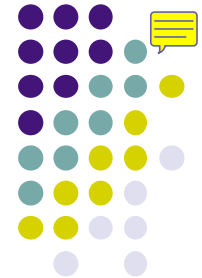
  $x^0 = 1$

  $x^n = x * x^{n-1}$ when $n>0$
- This gives a first program for {Pow X N} :

```
fun {Pow1 X N}
    if N==0 then 1
    else X*{Pow1 X N-1} end
end
```

- This function is highly inefficient in both time and space! Why? (there are two reasons)

Using a better definition of X^N



- Here is another definition of x^n:

$$x^0 = 1$$

$$x^n = x * x^{n-1} \text{ when } n>0 \text{ and } n \text{ is odd}$$

$$x^n = y^2 \text{ when } n>0 \text{ and } n \text{ is even and } y=x^{n/2}$$

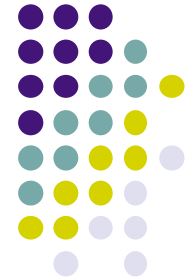
- This definition uses many fewer multiplications than the naive definition
  - And just like with the naive definition, we can use this definition to write a program
- Both definitions are also specifications
  - They are purely mathematical (no program code)

Second program for X^N

```
fun {Pow2 X N}
   if N==0 then 1
   elseif N mod 2 == 1 then
         X*{Pow2 X (N-1)}
   else Y in
         Y={Pow2 X (N div 2)}
         Y*Y
   end
end
```

This definition is better than the first, but it is still not tail recursive!

Calculating X^N with invariant programming



- We can do better than Pow2
  - We can write a tail-recursive program: a true loop
- We need an invariant
  - The invariant is the key to a good program
  - One part of the invariant will accumulate the result and another part of the invariant will disappear
  - What can we accumulate?

Reasoning on the invariant

- Here is an invariant: *(x and n constant; y, i, and a vary)*
  $x^n = y^i * a$
- We represent this invariant compactly as a triple:
  (y,i,a)
- Initially: *(y,i,a)* = *(x,n,1)*
- Let us decrease *i* while keeping the invariant true
- There are two ways to decrease *i* :
  - *(y,i,a)* ⇒*(y*y,i/2,a)* (when *i* is even)
  - *(y,i,a)* ⇒*(y,i-1,y*a)* (when *i* is odd)
- When *i=0* then the answer is *a*

Third program for X^N

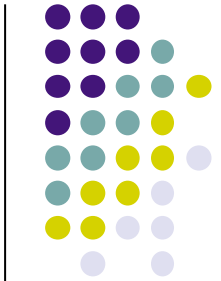
```
fun {Pow3 X N}
   fun {PowLoop Y I A}
      if I==0 then A
      elseif I mod 2 == 0 then
            {PowLoop Y*Y (I div 2) A}
      else  {PowLoop Y (I-1) Y*A} end
   end
in
   {PowLoop X N 1}
end
```

This program is a true loop (it is tail-recursive) and it uses very few multiplications

Invariants and goals

- Changing one part of the invariant forces the rest to change as well, because the invariant must remain true
  - The invariant's truth *drives* the program forward
- Programming a loop means finding a good invariant
  - Once a good invariant is found, coding is easy
  - Learn to think in terms of invariants!
- Using invariants is a form of goal-oriented programming
  - We will see another example of goal-oriented programming when we program with trees in lesson 5

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

MI-GLSD-M1 -UEM213 : Programming paradigms

Chapter III: Functional paradigm Advanced concepts

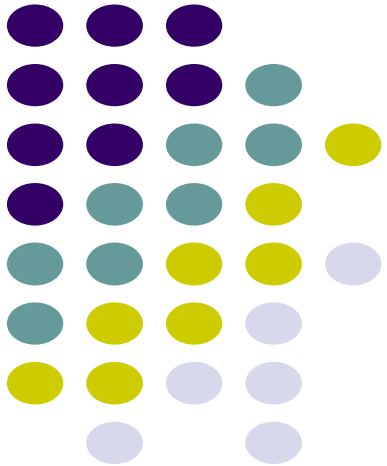
A. HARICHE

University of Djilali Bounaama, Khemis Meliana (UDBKM)

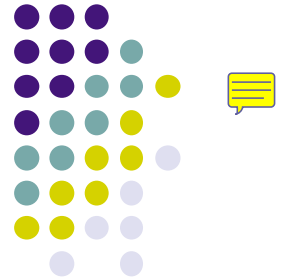
Faculty of sciences & technology

Mathematics & computer sciences department

a.hariche@univ-dbkm.dz



Definition of a list



- A list is a recursive data type: we define it in terms of itself
  - Recursion is used both for computations and data!
  - We need to know this when we write functions on lists

- A list is either an empty list or a pair of an element followed by another list
  - This definition is recursive because it defines lists in terms of lists. There is no infinite regress because the definition is used constructively to build larger lists from smaller lists.

- Let's introduce a formal notation

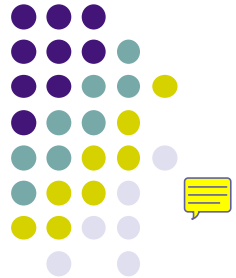
Syntax definition of a list

- Using an EBNF grammar rule we write:

  <List T> ::= nil | T '|' <List T>

- This defines the textual representation of a list
- EBNF = Extended Backus-Naur Form
  - Invented by John Backus and Peter Naur
  - <List T> represents a list of elements of type T
  - T represents one element of type T
- Be careful to distinguish between | and '|' : the first is part of the grammar notation (it means "or"), and the second is part of the syntax being defined

Some examples of lists



- According to the definition (if T is integers):

  nil
  10 | nil
  10 | 11 | nil
  10 | 11 | 12 | nil

- What about the bracket notation we saw before?
  - It is not part of the recursive definition of lists; it is an extra called *syntactic sugar*

Type notation

- <Int> represents an integer; more precisely, it is the set of all syntactic representations of integers
- <List <Int>> represents the set of all syntactic representations of lists of integers
- T represents the set of all syntactic representations of values of type T; we say that T is a type variable
  - Do not confuse a type variable with an identifier or a variable in memory! Type variables exist only in grammar rules.

Don’t confuse a thing and its representation

René Magritte, *La trahison des images*, 1928-29, oil,
Los Angeles County Museum of Art, Los Angeles.

- This is not a pipe.
  It is a digital display of a photograph of a painting of a pipe (thanks to Belgian surrealist René Magritte for pointing this out!).

1234

- This is not an integer.
  It is a digital display of a visual representation of an integer using numeric symbols in base 10.

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

Representations for lists

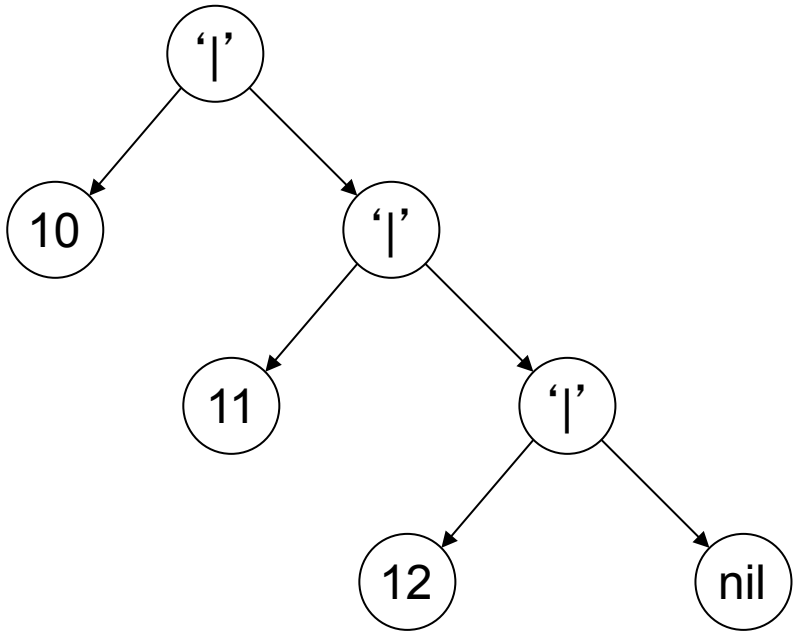
- The EBNF rule gives one textual representation
  - <List <Int>> ⇒
    10 | <List <Int>> ⇒
    10 | 11 | <List <Int>> ⇒
    10 | 11 | 12 | <List <Int>> ⇒
    10 | 11 | 12 | nil

We repeatedly replace the left-hand side of the rule by a possible value, until no more can be replaced

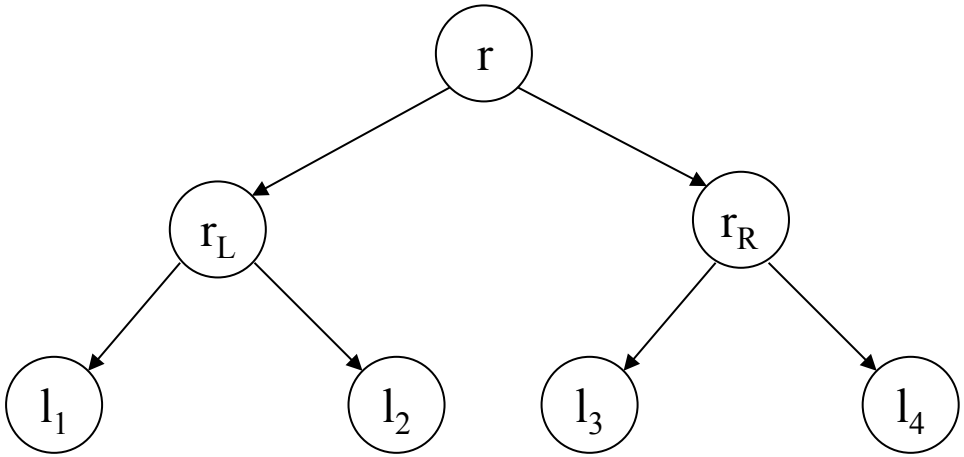
- Oz allows another textual representation
  - Bracket notation: [10 11 12]
  - In memory, [10 11 12] is identical to 10 | 11 | 12 | nil
  - Different textual representations of the same thing are called syntactic sugar

Graphical representation of a list

- Graphical representations are very useful for reasoning
  - Humans have very powerful visual reasoning abilities
- We start from the leftmost pair, namely 10 | <List <Int>>
  - We draw three nodes with arrows between them
  - We then replace the node <List <Int>> as before
- This is an example of a more general structure called a tree

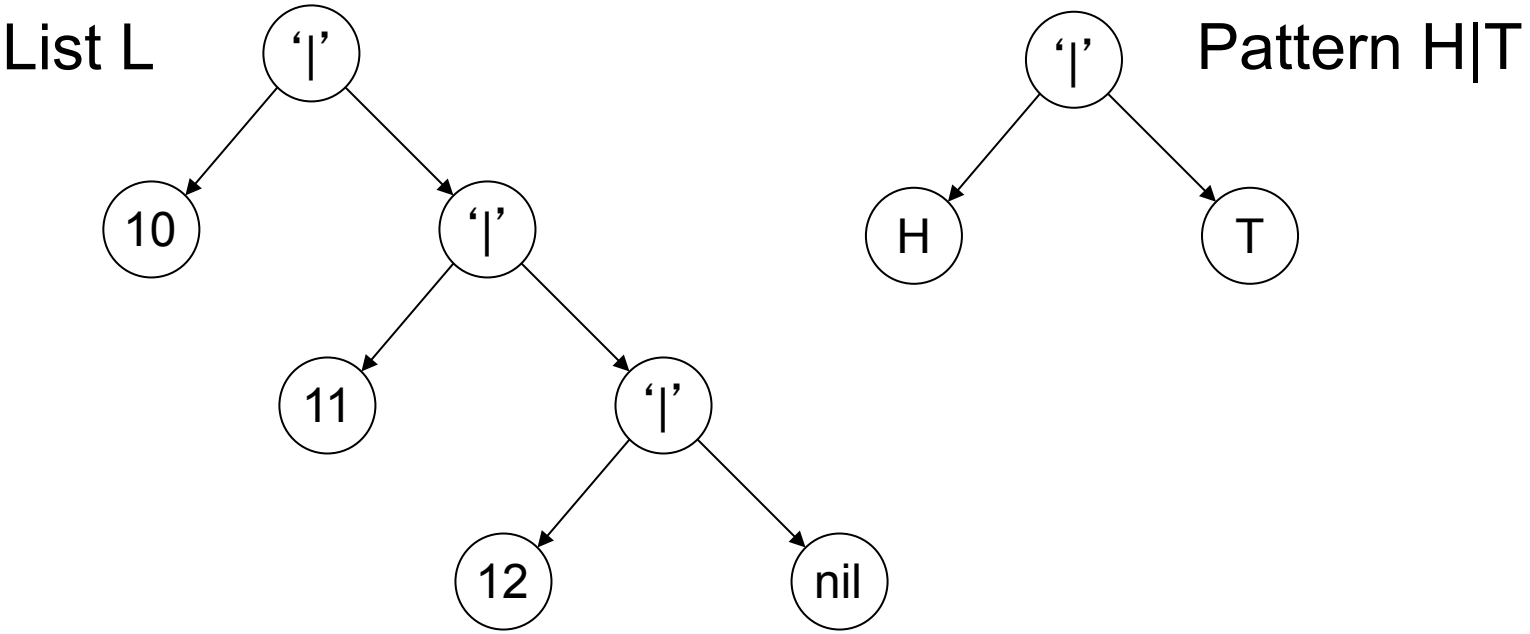


Trees and binary trees



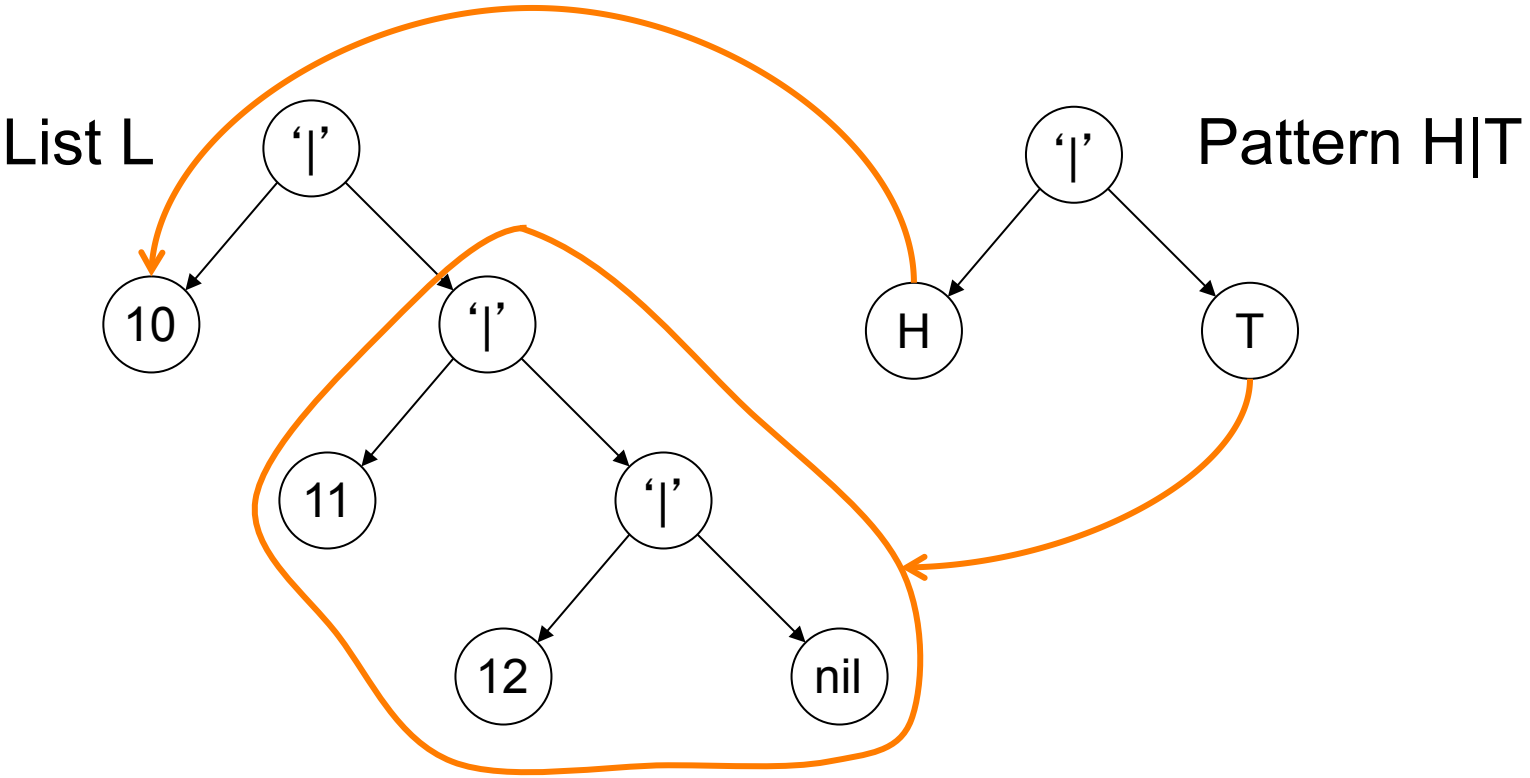
- A **tree** is either a leaf node (which is an empty tree) or a root node with arrows to a set of trees (called subtrees)
- A **binary tree** is a tree where all root nodes have exactly two subtrees (usually called left and right)

Pattern matching



- **case** L **of** H|T **then** ... **else** ... **end**

Pattern matching



- **case** L **of** H|T **then** ... **else** ... **end**
- H=10, T=11|12|nil

Functions that create lists

- Let us now define a function that outputs a list
  - We will use both pattern matching and recursion, as before, but this time the output will also be a list
  - We will define the Append function

The simple Append function is tail recursive

- We will see this by translating Append into the kernel language of the functional paradigm
- This translation shows that the recursive call is last
- This works because of single assignment: we create the output list before doing the recursive call

The kernel language

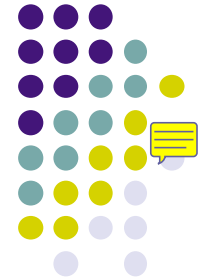
- As we mentioned in lesson 1, the kernel language is the simple core language of a programming paradigm
  - We have now seen enough concepts to introduce the kernel language of the functional paradigm

- All programs in the functional paradigm can be translated into the kernel language
  - All intermediate results of calculations are visible with identifiers
  - All functions become procedures with one extra argument
  - Nested function calls are unnested by introducing new identifiers

Kernel principle

- The kernel language is the first part of the formal semantics of a programming language
  - The second part is the *abstract machine* seen in lesson 6

Kernel language of the functional paradigm



- <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**
  | **proc** {<x> <x>$_1$ … <x>$_n$} <s> **end**
  | {<x> <y>$_1$ … <y>$_n$}
  | **case** <x> **of** <p> **then** <s>$_1$ **else** <s>$_2$ **end**
- <v> ::= <number> | <list> | ...
- <number> ::= <int> | <float>
- <list>, <p> ::= nil | <x> | <x> '|' <list>

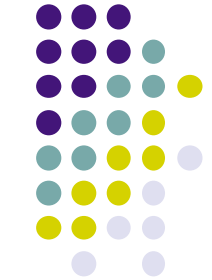
Almost complete!

We will see the full kernel language in lesson 4

Higher-order programming and records

- This lesson gives the final two concepts we need to complete the functional paradigm and its kernel language
  - Higher-order programming
  - Record data structures
- Higher-order programming is the ability to use functions (and procedures) as first-class entities in the language
  - As inputs and outputs of other functions
  - This is an enormously powerful ability that lies at the foundation of data abstraction (including object-oriented programming)
- Record data structures are a general compound data type that allows symbolic indexing
  - This is useful both for symbolic programming and for data abstraction

Higher-order programming



- Higher-order programming is based on two concepts
  - Contextual environment
  - Procedure value

- We introduce the contextual environment by means of a small exercise on static scope
  - This exercise shows naturally the need for the contextual environment

An exercise on static scope

What does this program display?

```
local P Q in
   proc {P} {Browse 100} end
   proc {Q} {P} end
   local P in
      proc {P} {Browse 200} end
      {Q}
   end
end
```

What is the scope of P?

```
local P Q in
   proc {P} {Browse 100} end
   proc {Q} {P} end
   local P in
      proc {P} {Browse 200} end
      {Q}
   end
end
```

What is the scope of P?

Scope of P

```
local P Q in
    proc {P} {Browse 100} end
    proc {Q} {P} end
    local P in
        proc {P} {Browse 200} end
        {Q}
    end
end
```

The P definition *inside* the scope

Contextual environment of Q

Scope of P

```
local P Q in
   proc {P} {Browse 100} end
   proc {Q} {P} end
   local P in
      proc {P} {Browse 200} end
      {Q}
   end
end
```

All the violet **P**'s refer to the same variable

Procedure Q must know the definition of **P** ⇒ it stores this in its *contextual environment*

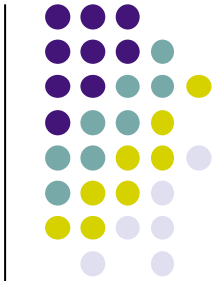
Contextual environment

- The contextual environment of a function (or procedure) contains all the identifiers that are used *inside* the function but declared *outside* of the function

```
declare
A=1
proc {Inc X Y} Y=X+A end
```

- The contextual environment of Inc is E_C = {A→*a*}
  - Where *a* is a variable in memory: *a=1*

Procedure value



- Procedure declarations look like statements:

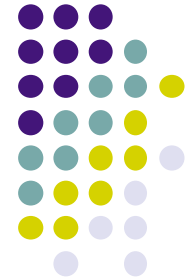
```
proc {Inc X Y} Y=X+A end
```

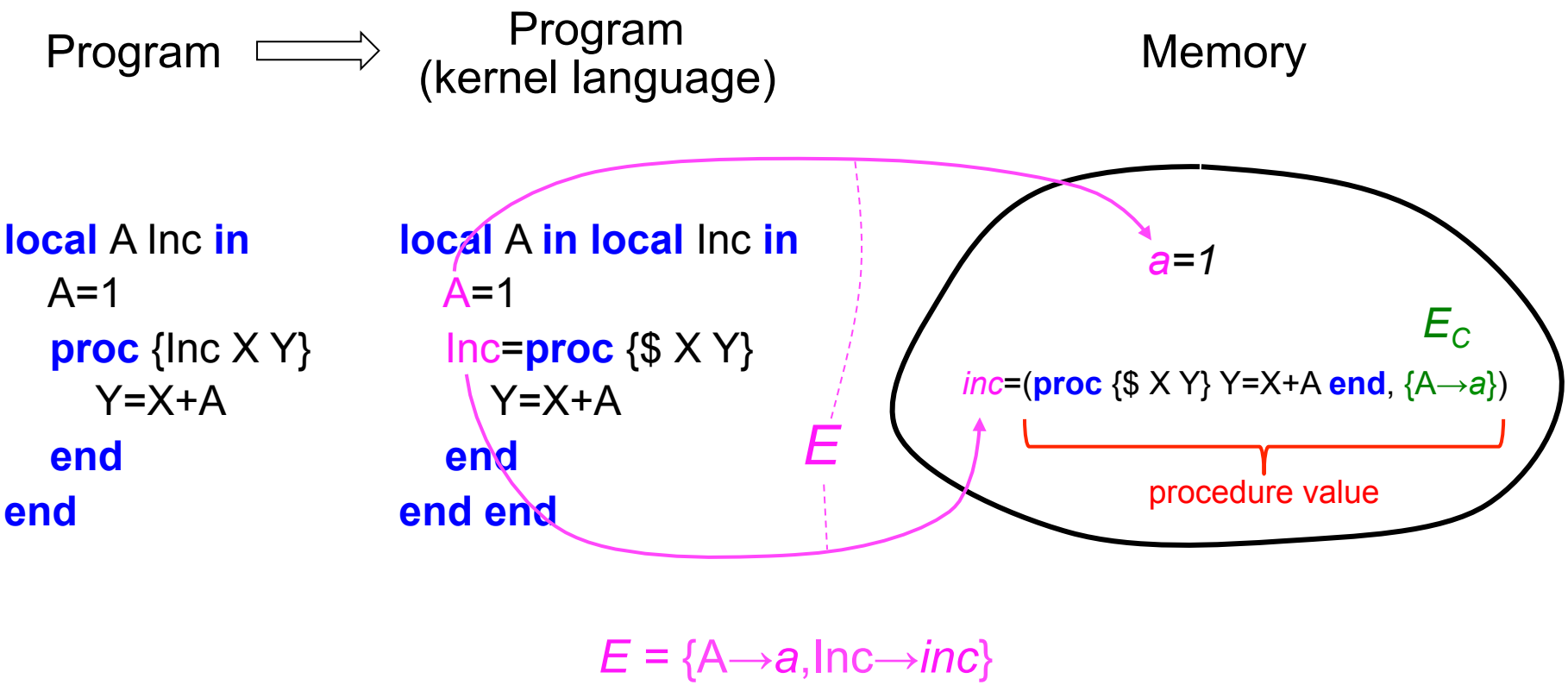
- But this is syntactic sugar! What really happens is that the identifier Inc refers to a variable that is bound to a procedure value:

```
Inc=proc {$ X Y} Y=X+A end
```

- The $ symbol is a placeholder to show that the procedure definition has no identifier. Instead of just removing the identifier, we replace it by a new symbol that cannot be confused with an identifier.

How procedures are stored in memory





Procedure values

- A procedure value is stored in memory as a pair:

inc = (**proc** {$ X Y} Y=X+A **end**, {A→*a*})

Procedure code — Contextual environment

- The variable *inc* is bound to the procedure value
  - Terminology: a procedure value is also called a *closure* or a *lexically scoped closure*

Higher-order programming

- Defining a procedure as a procedure value with a contextual environment is enormously expressive
  - It is arguably the most important invention in programming languages: it makes possible building large systems based on data abstraction
- Since procedures (and functions) are values, we can pass them as inputs to other functions and return them as outputs
  - Remember that in our kernel language, we consider functions and procedures to be the same concept: a function is a procedure with an extra output argument

Order of a function

- We define the order of a function (or procedure)
  - A function whose inputs and output are not functions is first order
  - A function is order N+1 if its inputs and output contain a function of maximum order N
- Let's give some examples to show what we can do with higher-order functions (where the order is greater than 1)
  - We will give more examples later in the course

Genericity

- Genericity is when a function is passed as an input

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

{Browse {Map fun {$ X} X*X end [7 8 9]}}
```

What is the order of Map in this call?

Instantiation

- Instantiation is when a function is returned as an output

```
declare
fun {MakeAdd A}
     fun {$ X} X+A end
end
Add5={MakeAdd 5}

{Browse {Add5 100}}
```

What is the order of MakeAdd?

What is the contextual environment of the function returned by MakeAdd?

Function composition

- We take two functions as input and return their composition

What is the contextual environment of the function returned by Compose?

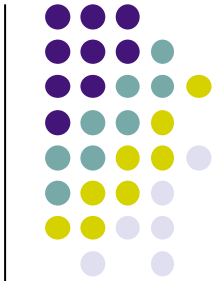
```
declare
fun {Compose F G}
      fun {$ X} {F {G X}} end
end
Fnew={Compose fun {$ X} X*X end
                    fun {$ X} X+1 end}
```

- What does {Fnew 2} return?
- What does {{Compose Fnew Fnew} 2} return?

Abstracting an accumulator

- We can use higher-order programming to do a computation that hides an accumulator
- Let's say we want to sum the elements of a list $L=[a_0\ a_1\ a_2\ \ldots\ a_{n-1}]$:
  - $S = a_0 + a_1 + a_2 + \ldots + a_{n-1}$
  - $S = (\ldots(((0 + a_0) + a_1) + a_2) + \ldots + a_{n-1})$
- We can write this generically with a function F:
  - $S = \{F\ \ldots\ \{F\ \{F\ \{F\ 0\ a_0\}\ a_1\}\ a_2\}\ \ldots\ a_{n-1}\}$
- Now we can define the higher-order function FoldL:
  - $S = \{FoldL\ [a_0\ a_1\ a_2\ \ldots\ a_{n-1}]\ F\ 0\}$
  - The accumulator is hidden inside FoldL!

Definition of FoldL



- Here is the definition of FoldL:

```
declare
fun {FoldL L F U}
      case L
      of nil then U
      [] H|T then {FoldL T F {F U H}}
      end
end
S={FoldL [5 6 7] fun {$ X Y} X+Y end 0}
```

The argument U is an accumulator

Encapsulation

- We can hide a value inside a function:

```
declare
fun {Zero} 0 end
fun {Inc H}
N={H}+1 in
      fun {$} N end
end
Three={Inc {Inc {Inc Zero}}}
{Browse {Three}}
```

- This is the foundation of encapsulation as used in data abstraction
- What is the difference if we write Inc as follows:

```
      fun {Inc H} fun {$} {H}+1 end end
```

Delayed execution

- We can define an statement and pass it to a function which decides whether or not to execute it

```
proc {IfTrue Cond Stmt}
      if {Cond} then {Stmt} end
end
Stmt = proc {$} {Browse 111*111} end
{IfTrue fun {$} 1<2 end Stmt}
```

- This can be used to build control structures from scratch (**if** statement, **while** loop, **for** loop, etc.)

Summary of higher-order

- We have given six examples to illustrate the expressiveness of higher-order programming:
  - Genericity
  - Instantiation
  - Function composition
  - Abstracting an accumulator
  - Encapsulation
  - Delayed execution

- We will use these techniques and others when we introduce the concepts of data abstraction
  - Data abstraction is built on top of higher-order programming!

Atoms and records

- A record is a general compound data type
  - Records are used to build many other compound data types
  - To explain records, we first introduce the atom data type
- An atom is a symbolic value
  - A sequence of lowercase letters and digits that starts with a letter
  - Also, a sequence of any characters delimited by single quotes
  - Example of a list containing five atoms:

```
declare
L=[john paul george ringo '1337 5|*34|<']
{Browse L.1}
{Browse L.2.1}
{Browse {Length L}}
```

Records

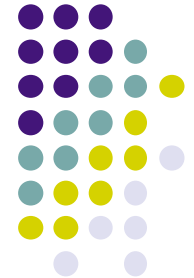
- A record groups a set of values into a single compound value
  - A record has a fixed number of values that can be accessed directly
- Each record has a label and a set of pairs of field names and fields
  - The label is an atom, the field names are atoms or integers, and the fields can be any value
  - The field names and fields are separated by a colon ':'
  - The position of a field in the record is not important; the records point(x:10 y:20) and point(y:20 x:10) are identical
  - All field names must be different; the syntax box(in:deadcat in:livecat) is illegal while box(in:cat alive:X) is legal
- Example record with five fields:

```
declare
R=rectangle(bottom:10 left:20 top:100 right:200 color:red)
```

Operations on records

- We only give the basic operations; many other operations exist in the Record module
  - The following examples use this record: R=rectangle(bottom:10 left:20 top:100 right:200)
- Record fields are accessed through the dot operation
  - {Browse (R.top-R.bottom)*(R.right-R.left)}
- The label and fields can be extracted directly
  - {Label R} returns rectangle (the value of the label)
  - {Width R} returns 4 (the number of fields)
  - {Arity R} returns [bottom left right top] (list of field names alphabetically)
- Records can be used in comparisons and pattern matching
  - {Browse R==rectangle(top:100 bottom:10 left:20 right:200)} displays **true**
  - **case** R **of** rectangle(bottom:A top:B left:C right:D) matches with A=10, B=100, C=20, D=200

Records are the only compound type



- Records are the only compound type in the kernel language
  - An atom is a record whose width is 0
  - A tuple is a record whose field names are successive integers starting with 1
    - If the numbering condition is not satisfied, the data item is not a tuple but it is still a record
    - Fields without numbers are automatically numbered starting with 1: pair(H T) is syntactic sugar for pair(1:H 2:T)
  - A list is a recursive data type built with records nil and H|T
    - Syntactic sugar: H|T same as '|'(H T) same as '|'(1:H 2:T)
- This keeps the kernel language simple
  - A single compound data type suffices to understand execution
  - All other types (lists, trees, and so on) are encoded with records

Some examples

- Given the following records: are they tuples or lists?
  - A=a(1:a 2:b 3:c)
  - B=a(1:a 2:b 4:c)
  - C=a(0:a 1:b 2:c)
  - D=a(1:a 2:b 3:c d)
  - E=a(a 2:b 3:c 4:d)
  - F=a(2:b 3:c 4:d a)
  - G=a(1:a 2:b 3:c foo:d)
  - H='|'(1:a 2:'|'(1:b 2:nil))
  - I='|'(1:a 2:'|'(1:b 3:nil))

The functional kernel language

- Now we have seen all the concepts in the functional paradigm that we will use
  - We can define its full kernel language
- We will use this kernel language to understand exactly what a functional program does
  - We have used it to see why list functions are tail-recursive
  - We will use it as part of the formal semantics (in lesson 6)
- Each time we introduce a new paradigm in the course we will define its kernel language
  - Each extends the functional kernel language with a new concept

The functional kernel language (in part)

- <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**
  | **proc** {<x> <x>$_1$ … <x>$_n$} <s> **end**
  | {<x> <y>$_1$ … <y>$_n$}
  | **case** <x> **of** <p> **then** <s>$_1$ **else** <s>$_2$ **end**

This is what we have seen so far

- <v> ::= <number> | <list> | ...
- <number> ::= <int> | <float>
- <list>, <p> ::= nil | <x> | <x> '|' <list>

The functional kernel language (in part)

- <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**
  | **proc** {<x> <x>$_1$ … <x>$_n$} <s> **end**
  | {<x> <y>$_1$ … <y>$_n$}
  | **case** <x> **of** <p> **then** <s>$_1$ **else** <s>$_2$ **end**
- <v> ::= <number> | <list> | ...
- <number> ::= <int> | <float>
- <list>, <p> ::= nil | <x> | <x> '|' <list>

This is what we have seen so far; it needs *two changes* to become the full kernel language of the functional paradigm

1. Procedure declarations (should be values)

2. Compound types (should be more than lists only)

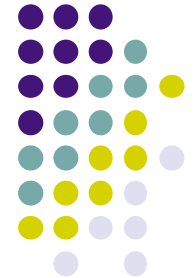
HARICHE A. a.hariche@univ-dbkm.dz

The functional kernel language (in part)

- <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**
  | ~~**proc** {<x> <x>$_1$ … <x>$_n$} <s> **end**~~
  | {<x> <y>$_1$ … <y>$_n$}
  | **case** <x> **of** <p> **then** <s>$_1$ **else** <s>$_2$ **end**
- <v> ::= <number> | <procedure> | <list> | ...
- <number> ::= <int> | <float>
- <procedure> ::= **proc** {$ <x>$_1$ … <x>$_n$} <s> **end**
- <list>, <p> ::= nil | <x> | <x> '|' <list>

1. Procedures are values in memory (like numbers and lists)

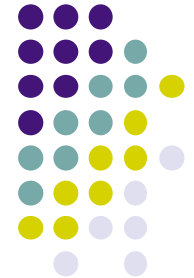
The functional kernel language (complete)



- <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> ~~| <list>~~ | <record>
- <number> ::= <int> | <float>
- <procedure> ::= **proc** {$ <x>$_1$ … <x>$_n$} <s> **end**
- <record>, <p> ::= <lit> | <lit>(<f>$_1$:<x>$_1$ … <f>$_n$:<x>$_n$)

2. Records subsume lists

The functional kernel language (complete)



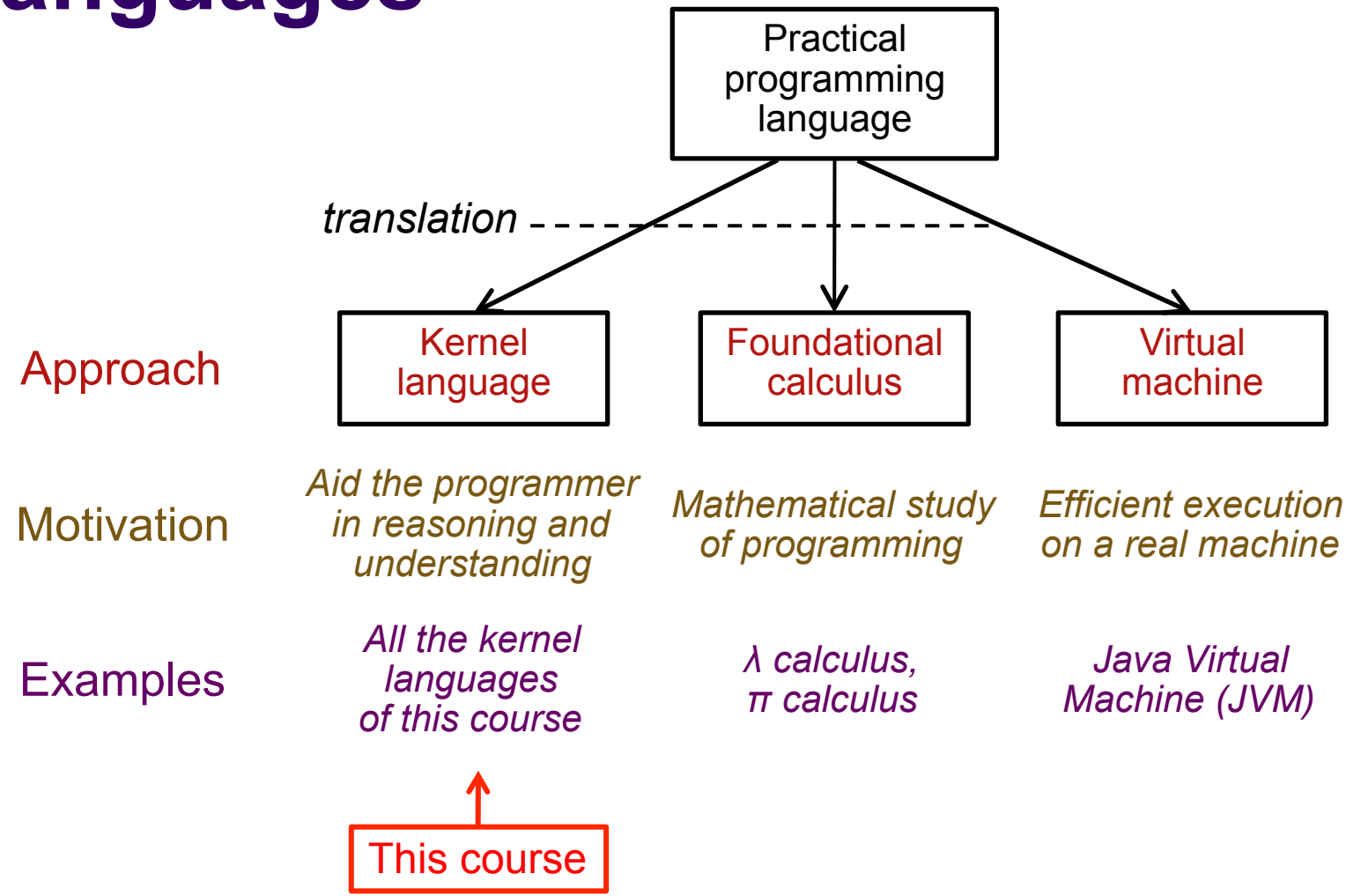
- <s> ::= **skip**
  | <s>$_1$ <s>$_2$
  | **local** <x> **in** <s> **end**
  | <x>$_1$=<x>$_2$
  | <x>=<v>
  | **if** <x> **then** <s>$_1$ **else** <s>$_2$ **end**
  | {<x> <y>$_1$ … <y>$_n$}
  | **case** <x> **of** <p> **then** <s>$_1$ **else** <s>$_2$ **end**
- <v> ::= <number> | <procedure> | <record>
- <number> ::= <int> | <float>
- <procedure> ::= **proc** {$ <x>$_1$ … <x>$_n$} <s> **end**
- <record>, <p> ::= <lit> | <lit>(<f>$_1$:<x>$_1$ … <f>$_n$:<x>$_n$)

Procedure values and records are important basic types. They allow, for example, to define all the concepts of object-oriented programming.

Kernel language of the functional paradigm

- <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>$_n$)

Three ways to understand languages

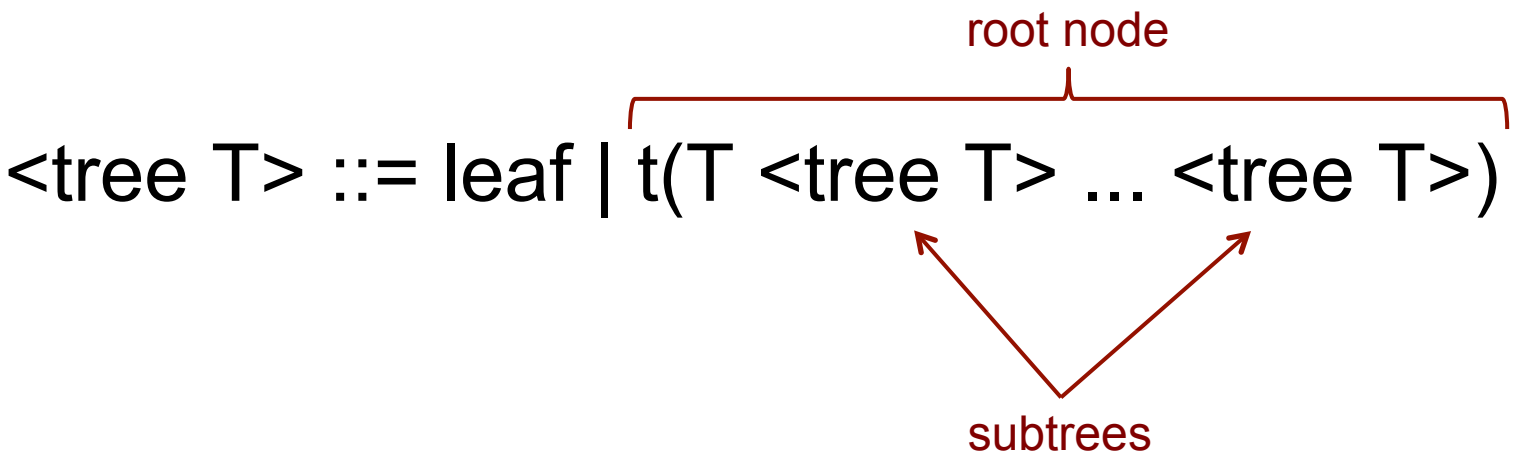


Trees

- Trees are the second most important data structure in computing, next to lists
  - Trees are extremely useful for efficiently organizing information and performing many kinds of calculations
- Trees illustrate well goal-oriented programming
  - Many tree data structures are based on a global property, that must be maintained during the calculation
- In this lesson we will define trees and use them to store and look up information
  - We will define ordered binary trees and algorithms to add information, look up information, and remove information

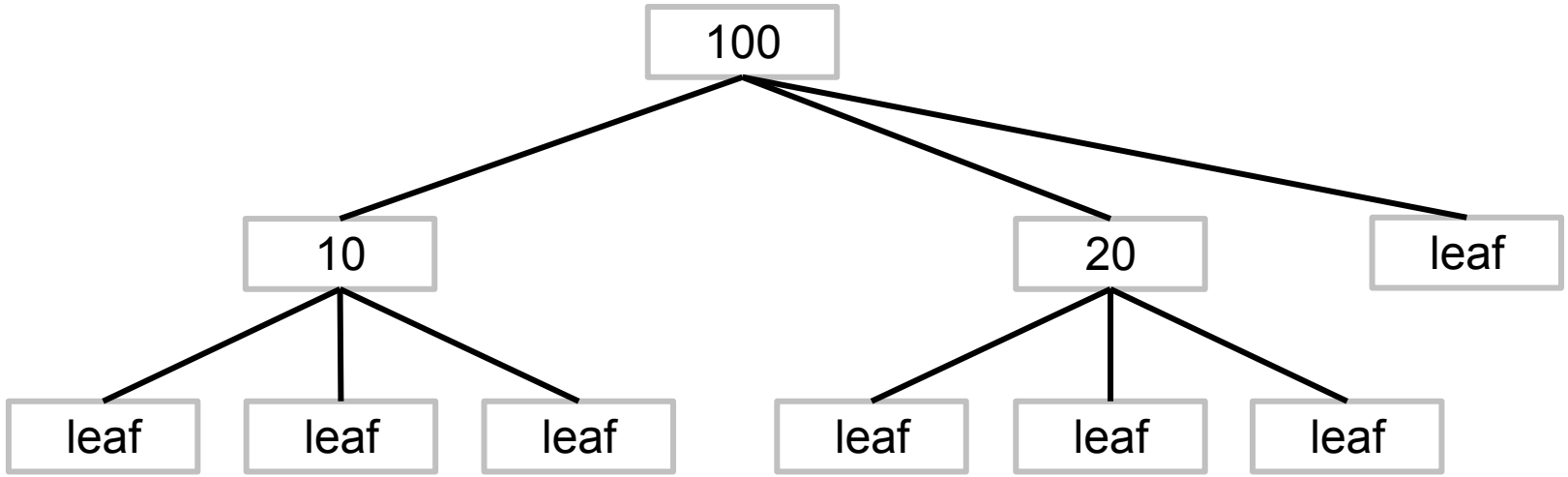
Trees

- A tree is a recursive structure: it is either an empty tree (called a leaf) or an element and a set of trees



Example tree

- **declare**
  T=t(100 t(10 leaf leaf leaf) t(20 leaf leaf leaf) leaf)



Trees compared to lists

- A tree is a recursive structure: it is either an empty tree (called a leaf) or an element and a set of trees

  <tree T> ::= leaf | t(T <tree T> ... <tree T>)

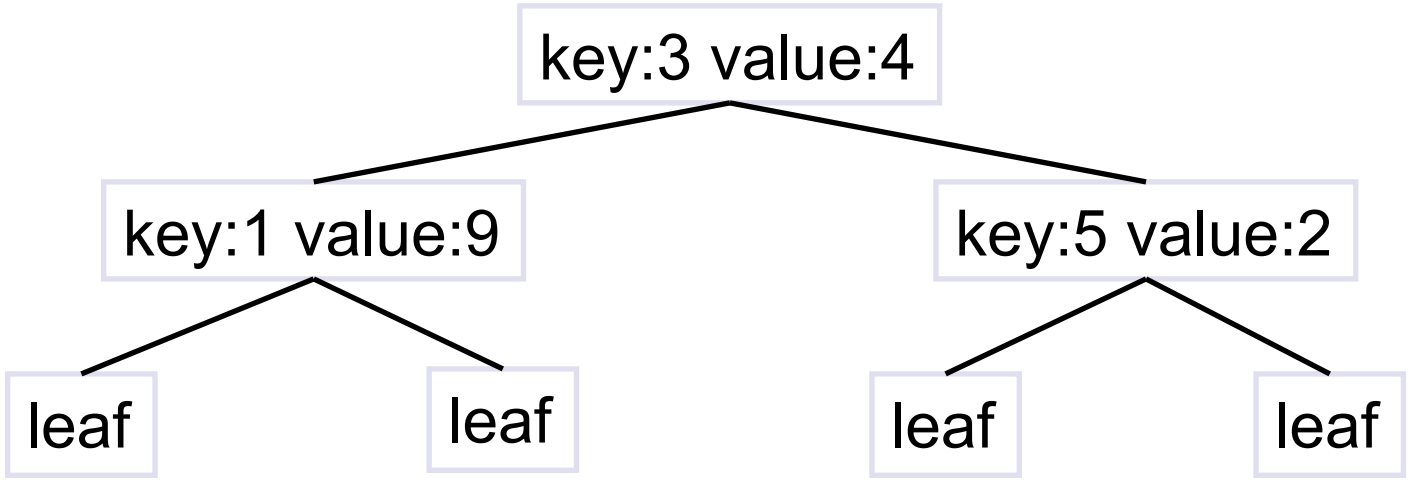
  <list T> ::= nil | ‘|’(T <list T>)

Notice the similarity with lists!

Ordered binary tree

- <obtree T> ::= leaf
  | tree(key:T value:T left:<obtree T> right:<obtree T>)

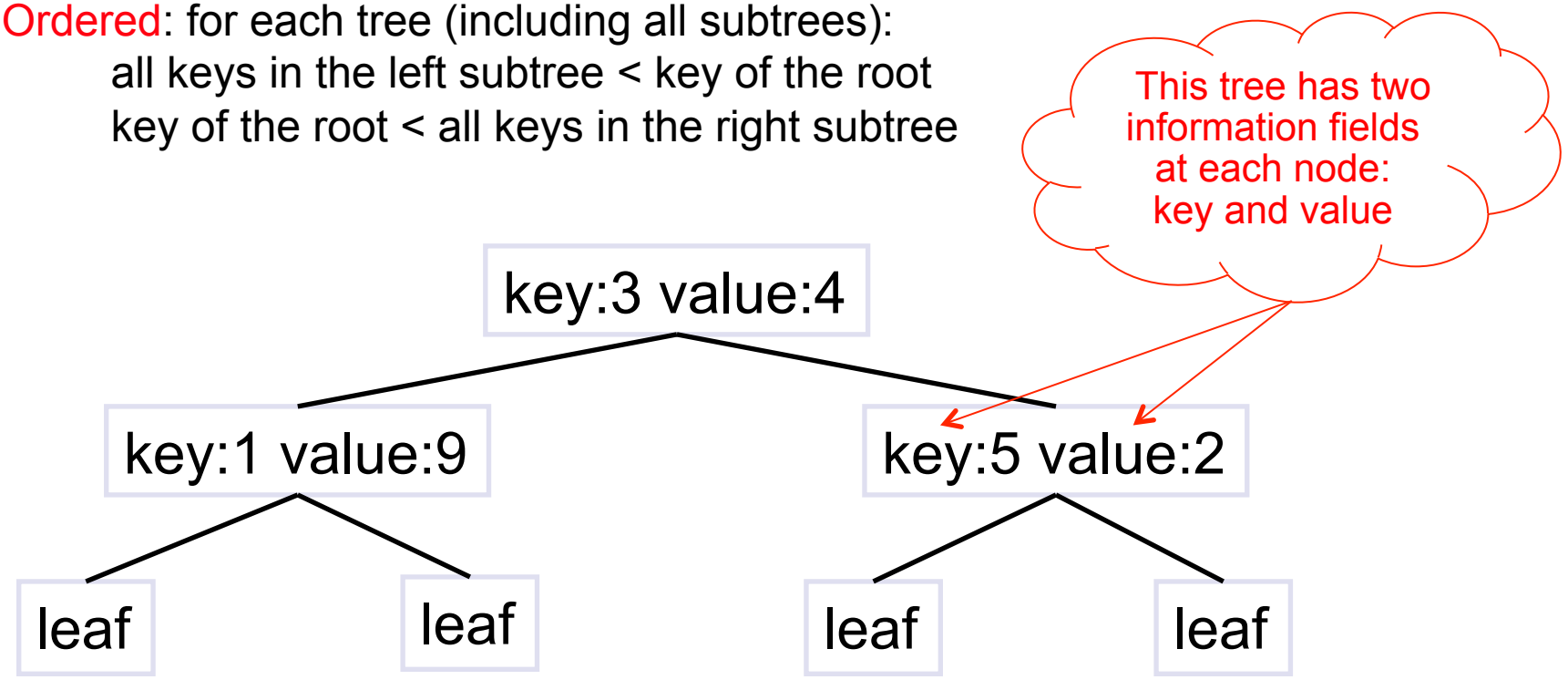
- Binary: each non-leaf tree has two subtrees (named left and right)
- Ordered: for each tree (including all subtrees):
  all keys in the left subtree < key of the root
  key of the root < all keys in the right subtree



Ordered binary tree

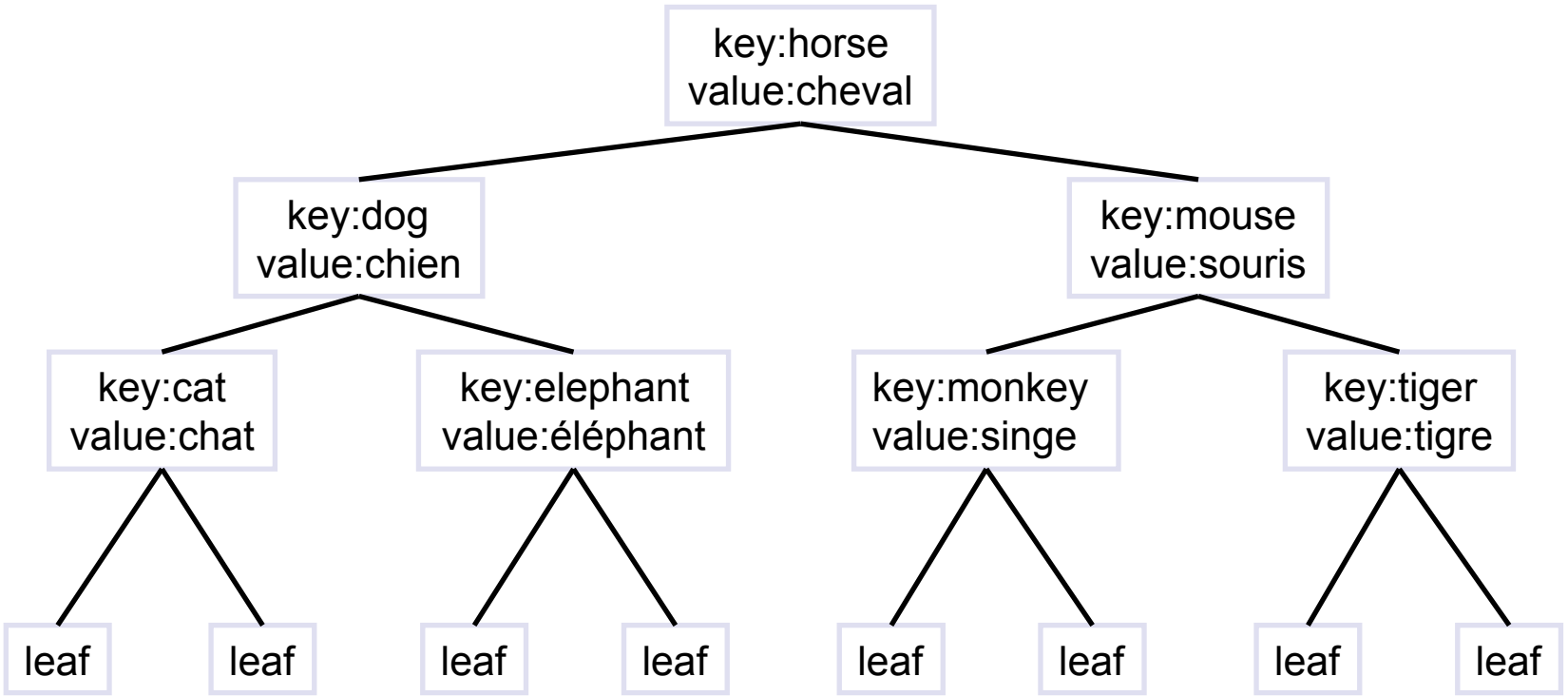
- <obtree T> ::= leaf
  | tree(key:T value:T left:<obtree T> right:<obtree T>)

- Binary: each non-leaf tree has two subtrees (named left and right)
- Ordered: for each tree (including all subtrees):
  all keys in the left subtree < key of the root
  key of the root < all keys in the right subtree



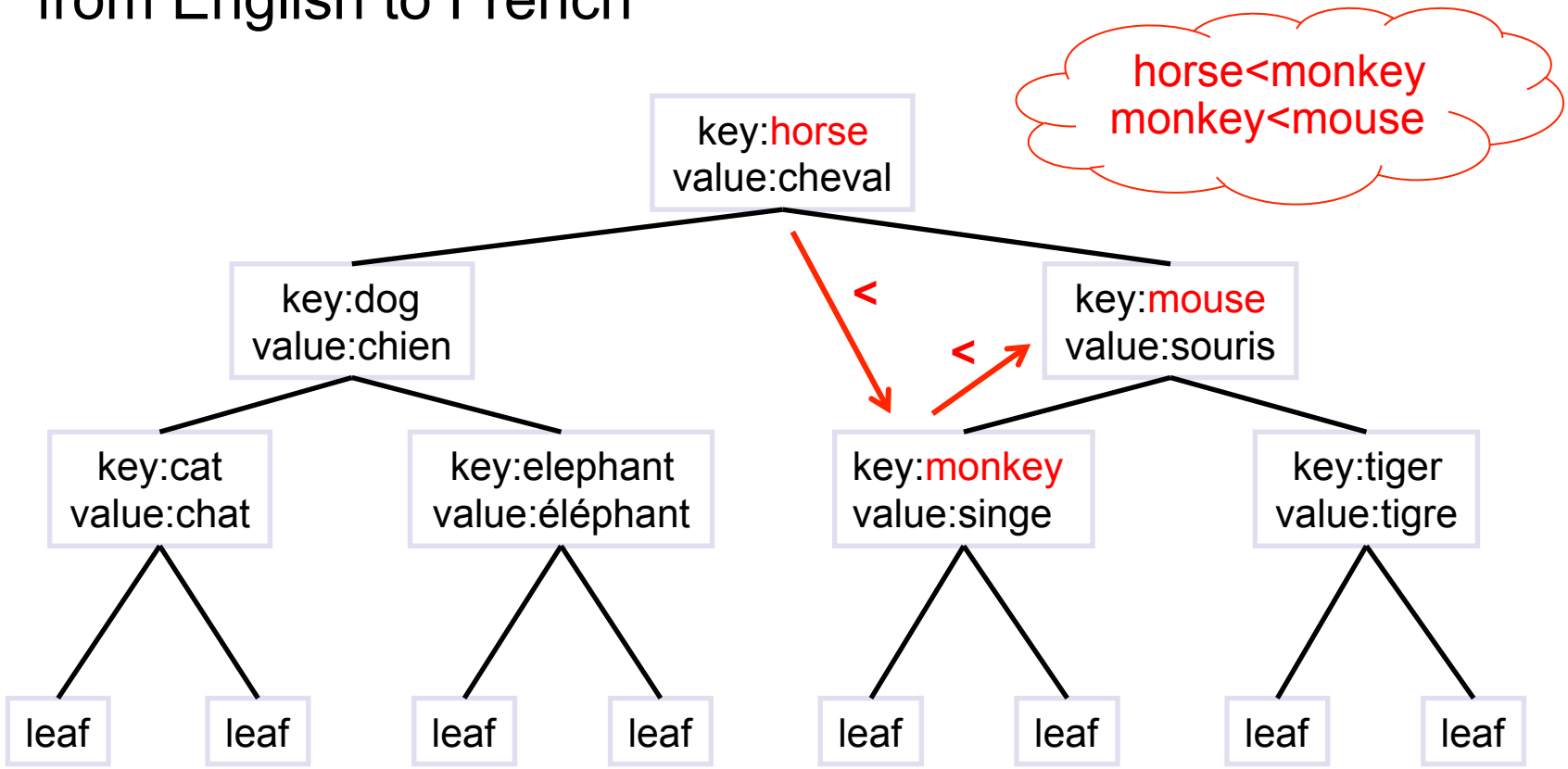
Ordered binary tree

- This ordered binary tree is a translation dictionary from English to French



Ordered binary tree

- This ordered binary tree is a translation dictionary from English to French



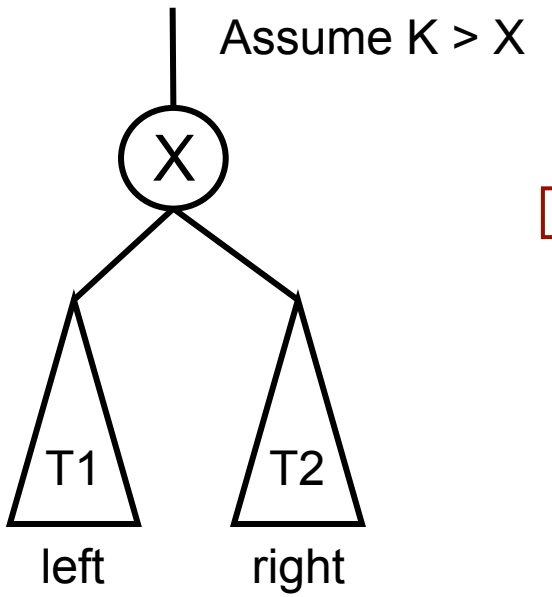
Search tree

- Search tree: A tree that is used to organize information, and with which we can perform various operations such as looking up, inserting, and deleting information

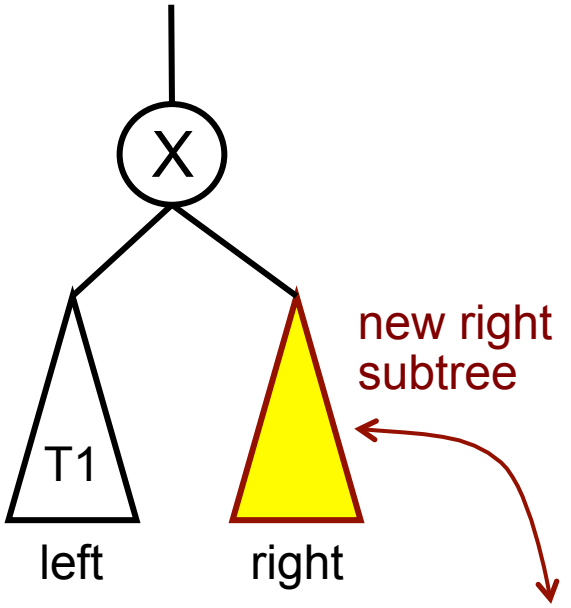
- Let's define these three operations:
  - {Lookup K T}: returns the value V corresponding to key K
  - {Insert K W T}: returns a new tree containing (K,W)
  - {Delete K T}: returns a new tree that does not contain K

Inserting a new key/value pair

Original tree



New tree



tree(key:X value:V left:T1 right:T2)

tree(key:X value:V left:T1 right:{Insert K W T2})

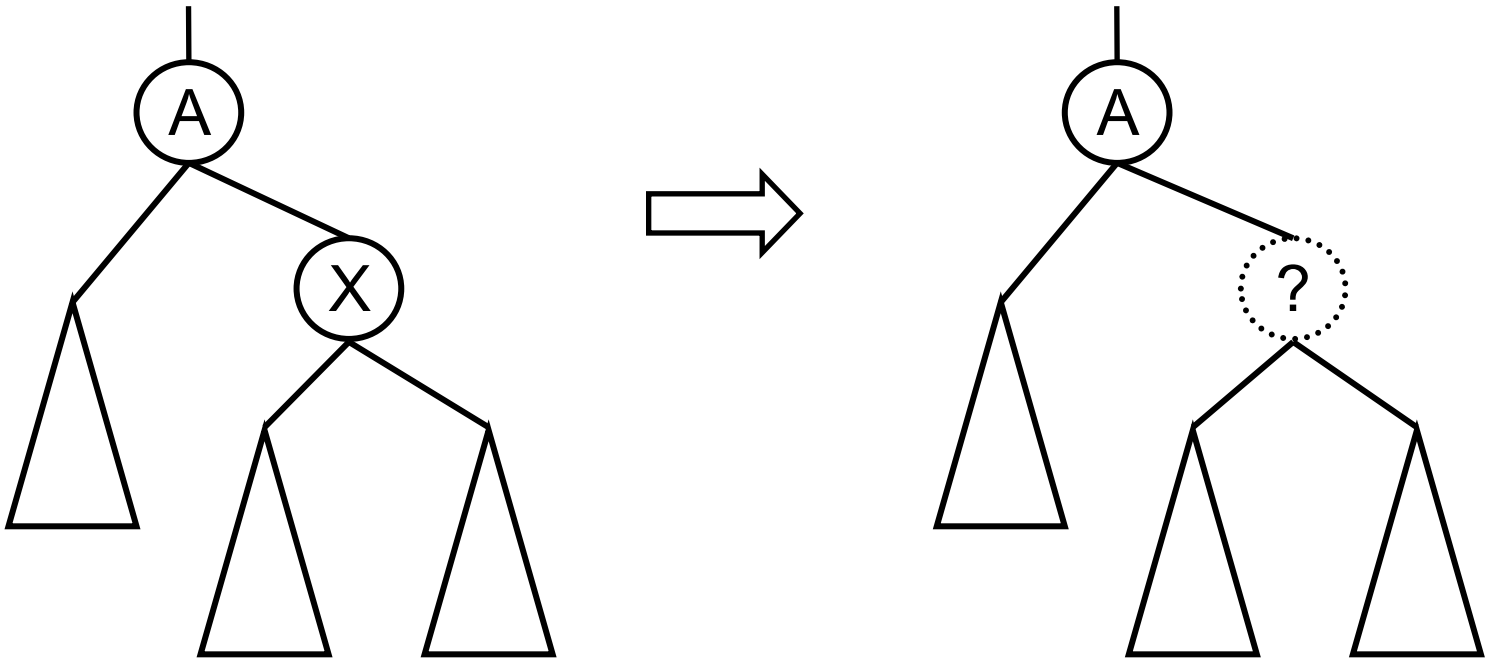
unchanged part

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

Efficiency

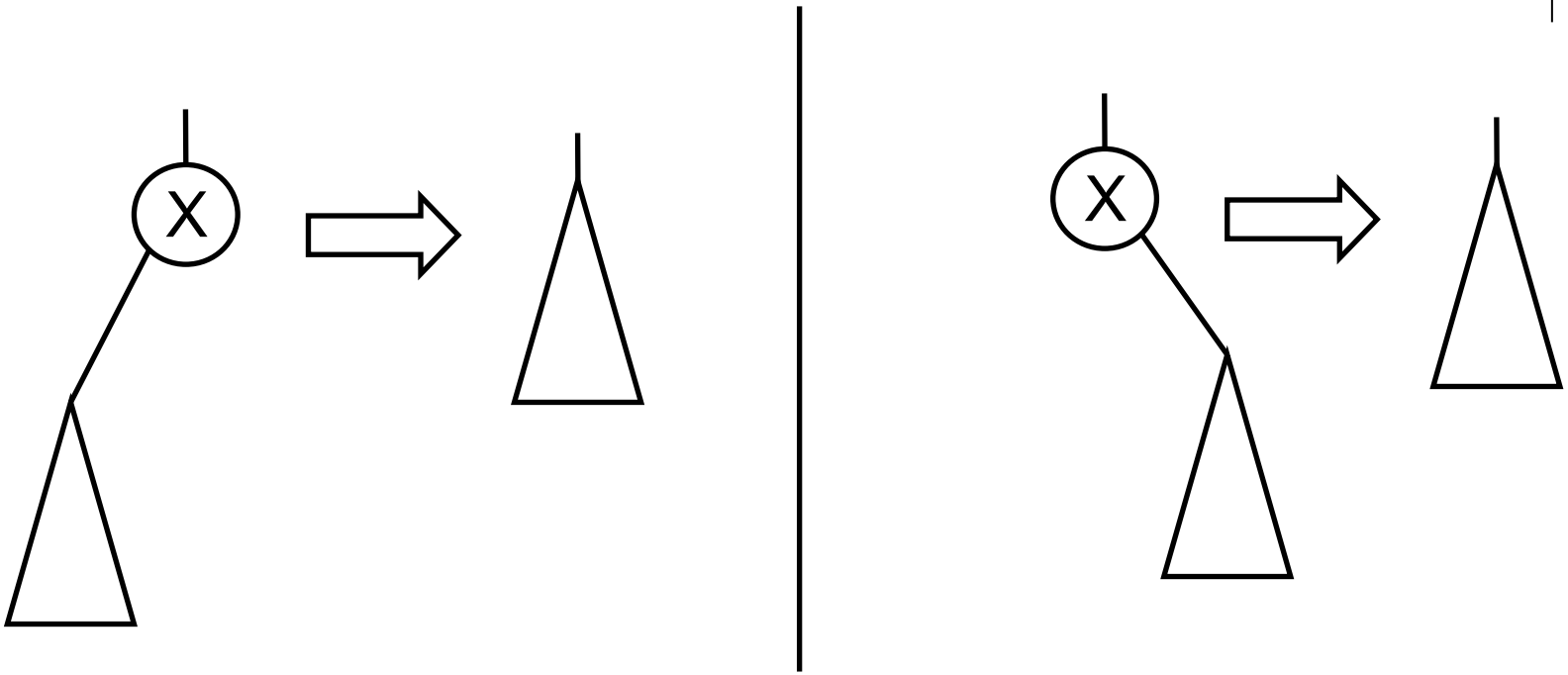
- How efficient is the Lookup function?
  - If there are *n* words in the tree, and each node's subtrees are approximately equal in size (we say the tree is balanced), then the average lookup time is proportional to log *n*
  - Tree lookup is much more efficient than list lookup: if for 1000 words the average time is 10, then for 1000000 words this will increase to 20 (instead of being multiplied by 1000!)
- If the tree is not balanced, say all the right subtrees are very small, then the time will be much larger
  - In the worst case, the tree will look like a list
- How can we arrange for the tree to be balanced?
  - There exist algorithms for balancing an unbalanced tree, but if we insert words randomly, then we can show that the tree will be approximately balanced, good enough to achieve logarithmic time

Deleting an element from an ordered binary tree



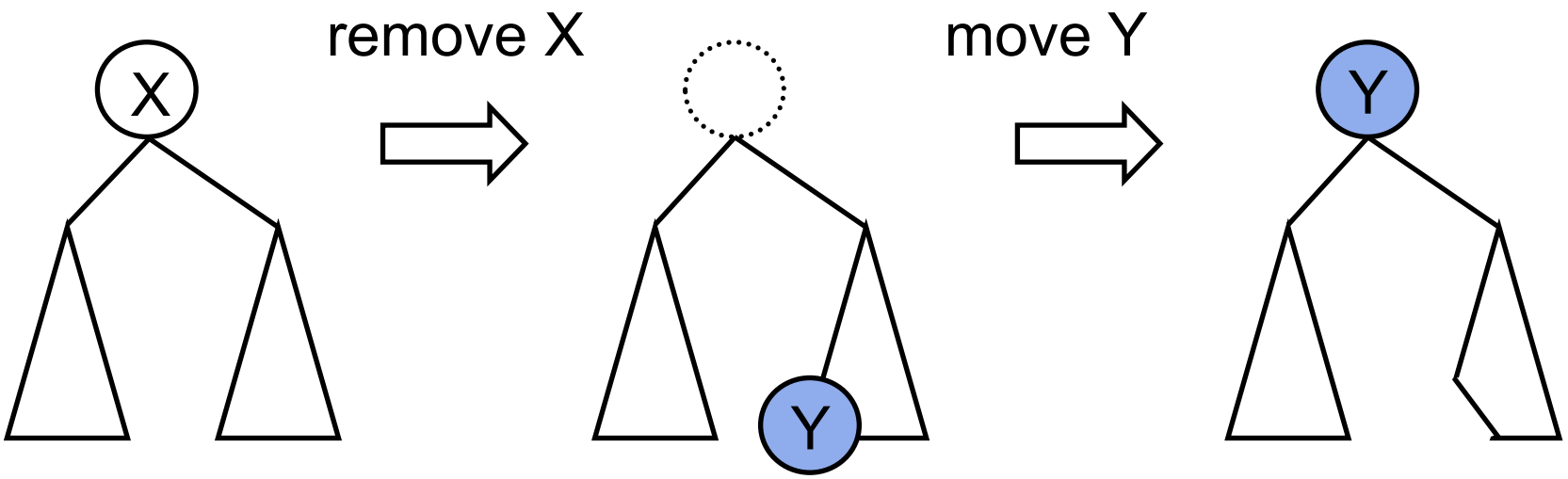
The problem is to repair the tree after X disappears

Deleting an element when one subtree is empty



It’s easy when one of the subtrees is empty:
just replace the tree by the other subtree

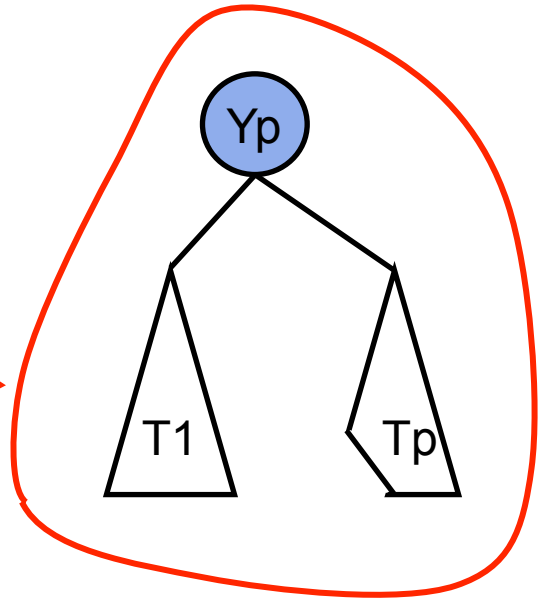
Deleting an element when both subtrees are not empty



The idea is to fill the ”hole” that appears after X is removed. We can put there the smallest element in the right subtree, namely Y.

We need a new function: RemoveSmallest

```
fun {Delete K T}
   case T
   of leaf then leaf
   [] tree(key:X value:V left:T1 right:T2) andthen K==X then
         case {RemoveSmallest T2}
         of none then T1
         [] triple(Tp Yp Vp) then
               tree(key:Yp value:Vp left:T1 right:Tp)
         end
   [] … end
end
```



- RemoveSmallest takes a tree and returns three values:
  - The new subtree Tp without the smallest element
  - The smallest element's key Yp
  - The smallest element's value Vp
- With these three values we can build the new tree where Yp is the root and Tp is the new right subtree

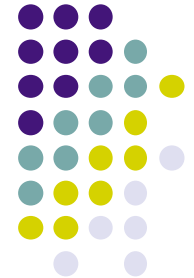
Recursive definition of RemoveSmallest

```
fun {RemoveSmallest T}
   case T
   of leaf then none
   [] tree(key:X value:V left:T1 right:T2) then
      case {RemoveSmallest T1}
      of none then triple(T2 X V)
      [] triple(Tp Xp Vp) then
         triple(tree(key:X value:V left:Tp right:T2) Xp Vp)
      end
   end
end
```

To understand this definition, draw diagrams with trees!

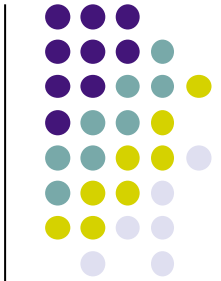
- RemoveSmallest takes a tree T and returns:
  - The atom none when T is empty
  - The record triple(Tp Xp Vp) when T is not empty

Why do we need semantics?



- If you do not understand something, then you do not master it – it masters you!
  - If you know nothing about how a car works, then a car mechanic can charge you whatever he wants
  - If you do not understand how government works, then you cannot vote wisely and the government becomes a tyranny
- The same holds true for programming
  - To write correct programs and to understand other people's programs, you have to understand the language deeply
  - All software developers should have this level of understanding
  - The goal of this lesson is to show you how

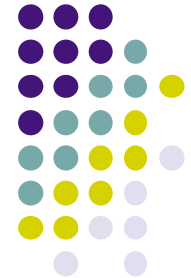
What is the semantics of a language?



- The semantics of a programming language, also called formal semantics or mathematical semantics, is a completely precise explanation of how programs execute that can be used to reason about program design and correctness
- We will give a semantics for all the paradigms of this course
  - We start by giving the semantics of the functional paradigm
- We have already seen the first part, namely the *kernel language*
  - In this lesson we will see the second part, namely the *abstract machine*

- Before taking the plunge into the abstract machine, let's take a step back and talk about semantics in general

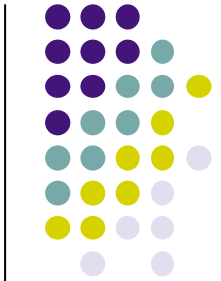
How can we define language semantics?



- Four general approaches have been invented:
  - Operational semantics: Explains a program in terms of its execution on a rigorously defined abstract machine
  - Axiomatic semantics: Explains a program as an implication: if certain properties hold before the execution, then some other properties will hold after the execution
  - Denotational semantics: Explains a program as a function over an abstract domain, which simplifies certain kinds of mathematical analysis of the program
  - Logical semantics: Explains a program as a logical model of a set of logical axioms, so program execution is deduction: the result of a program is a true property derived from the axioms

- The operational semantics works for all paradigms (since all programs run on computers!)
  - The other approaches are less general; they work best for some paradigms
  - To reason about correctness, we will complement the operational semantics with ideas taken from the other approaches

Operational semantics



- The operational semantics has two parts
  - Kernel language: first, translate the program into the kernel language
  - Abstract machine: then, execute the program on the abstract machine

- We will introduce the operational semantics with an example that uses it to prove correctness of a program
  - After this introduction, we will define the abstract machine and give an example of how it executes
  - Then we will define the semantic rules for each instruction of the kernel language
  - Finally, we will take a special look at procedure definition and call, since they are very important (higher-order programming and data abstraction)

Introduction to semantics

- Let's introduce our semantics by means of an example

- First, let's decide what the semantics will be used for in our example:
  - To ensure that the program is correct (this is called *verification*)
  - To make sure the program is well-designed
  - To explain the program to others
  - To calculate time and memory utilisation
  - To understand how the program manages memory (in particular, how it does *garbage collection*)

- Let's choose the first goal, namely correctness

When is a program correct?

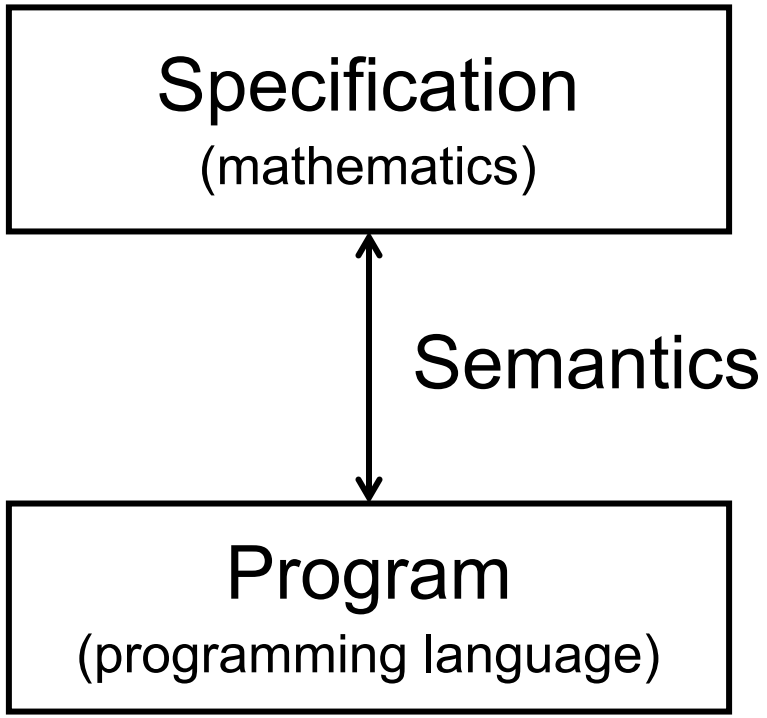
- “A program is correct when it does what we want it to”
- How can we be sure?

- There are two starting points:
  - **The program’s specification**: a mathematical definition of the result of the program as a function of the input
  - **The language semantics**: a precise mathematical model of how a program executes

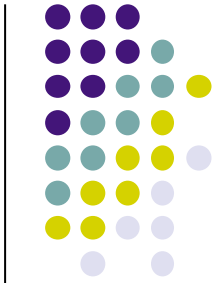
- We need to prove that the program satisfies the specification, when it executes according to the semantics

The three pillars

- The specification: what we want
- The program: what we have
- The semantics connects these two: proving that what we have executes according to what we want



Example: correctness of factorial



- The specification of {Fact N} (mathematics)

  $0! = 1$

  $n! = n \times ((n-1)!)$ when $n>0$

- The program (programming language)

```
fun {Fact N}
    if N==0 then 1 else N*{Fact N-1} end
end
```

- The semantics connects the two

Mathematical induction

- To make this proof for a recursive function we need to use mathematical induction
  - A recursive function calculates on a recursive data structure, which has a base case and a general case
  - We first show the correctness for the base case
  - We then show that if the program is correct for a general case, it is correct for the next case

- For integers, the base case is usually 0 or 1, and the general case *n-1* leads to the next case *n*

- For lists, the base case is usually nil or a small list, and the general case T leads to the next case H|T

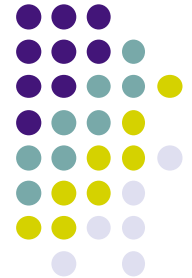
The inductive proof

- We must show that {Fact N} calculates *n!* for all *n*≥0

- Base case: *n=0*
  - The specification says: *0!=1*
  - The execution of {Fact 0}, *using the semantics*, gives {Fact 0}=1
    - *It's correct!*
- General case: *(n-1) → n*
  - The specification says: *n! = n×(n-1)!*
  - The execution of {Fact N}, *using the semantics*, gives {Fact N} = N*{Fact N-1}
    - We assume that {Fact N-1}=*(n-1)!*
    - We assume that the language correctly implements multiplication
    - Therefore: {Fact N} = N*{Fact N-1} = *n×(n-1)! = n!*
    - *It's correct!*

- Now we just need to understand the magic words "*using the semantics*"!

How to execute a program
using the semantics

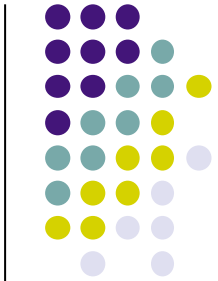


- We execute the program using the semantics by following two steps

- First, we translate the program into kernel language
  - The kernel language is a simple language that has all essential concepts
  - All programs in the practical language can be translated into kernel language
  - → We translate the definition of Fact into kernel language

- Second, we execute the translated program on the abstract machine
  - The abstract machine is a simplified computer with a precise mathematical definition
  - → We execute the call {Fact 0 R} on the abstract machine

Executing Fact using the semantics



- We need to execute both {Fact 0} and {Fact N} using the semantics
- First we translate the definition of Fact into kernel language:

```
proc {Fact N R}
   local B in
      B=(N==0)
      if B then R=1
      else local N1 R1 in
         N1=N-1
         {Fact N1 R1}
         R=N*R1
      end
   end
end
```

Execution of {Fact 0} (1)

- Let's first look at the function call {Fact 0}
- We execute the procedure call {Fact N R} where N=0
- We need a memory σ and an environment *E*:

  σ = {*fact*=(**proc** {$ N R} … **end**,{Fact→*fact* }), *n*=0, *r*}
  E = {Fact→*fact*, N→*n*, R→*r* }

- Here is what we will execute:

  {Fact N R}, E, σ

Execution of {Fact 0} (2)

- To execute {Fact N R} we replace it by the procedure body

- The instruction:

  {Fact N R}, {Fact→*fact*, N→*n*, R→*r* }, σ

  is replaced by the instruction:

```
local B in
    B=(N==0)
    if B then R=1 else … end
end, {Fact→fact, N→n, R→r }, σ
```

Execution of {Fact 0} (3)

- To execute the **local** instruction:

  local B **in**
  B=(N==0)
  if B **then** R=1 **else** … **end**
  end, {Fact→*fact*, N→*n*, R→*r* }, σ

  we do two operations:

  - We extend the memory with a new variable *b*
  - We extend the environment with {B → *b*}

- We then replace the instruction by its body:

  B=(N==0)
  if B **then** R=1 **else** … **end**,
  {Fact→*fact*, N→*n*, R→*r*, B → *b*}, σ∪{*b*}

Execution of {Fact 0} (4)

- We now do the same for:
  B=(N==0)
  and:
  if B **then** R=1 **else** … **end end**

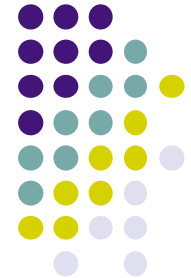
- This will first bind *b*=**true** and then bind *r=1*
- This completes the execution of {Fact 0}

- We have executed {Fact 0} with the semantics and shown that the result is 1
- To complete the proof, we still have to show that the result of {Fact N} is the same as N*{Fact N-1}

We have proved the correctness of Fact

- Let's recapitulate the approach
- Start with the specification and program of Fact
  - We want to prove that the program satisfies the specification
  - Since the function is recursive, our proof uses mathematical induction
- We need to prove the base case and the general case:
  - Prove that {Fact 0} execution gives 1
  - Prove that {Fact N} execution gives N*{Fact N-1}
- We prove both cases using the semantics and the Fact program
  - To use the semantics, we first translate Fact into kernel language, and then we execute on the abstract machine
- This completes the proof

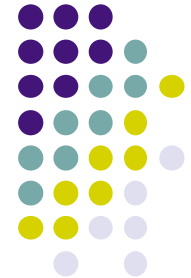
How to execute a program using the semantics



- We execute the program using the semantics by following two steps
- First, we translate the program into kernel language
  - The kernel language is a simple language that has all essential concepts
  - All programs in the practical language can be translated into kernel language
- Second, we execute the translated program on the abstract machine
  - The abstract machine is a simplified computer with a precise mathematical definition

→ Let's take a closer look at the abstract machine

Kernel language of the functional paradigm



- <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>$_n$)

Abstract machine concepts

- Single-assignment memory $\sigma = \{x_1=10, x_2, x_3=20\}$
  - Variables and the values they are bound to
- Environment $E = \{X \rightarrow x, Y \rightarrow y\}$
  - Link between identifiers and variables in memory
- Semantic instruction $(<s>, E)$
  - An instruction with its environment
- Semantic stack $ST = [(<s>_1, E_1), \ldots, (<s>_n, E_n)]$
  - A stack of semantic instructions
- Execution state (ST, σ)
  - A pair of a semantic stack and the memory
- Execution $(ST_1, \sigma_1) \rightarrow (ST_2, \sigma_2) \rightarrow (ST_3, \sigma_3) \rightarrow \ldots$
  - A sequence of execution states

Abstract machine execution algorithm

- **procedure** execute(<s>)
  begin
  ST:=[(<s>,{})]; /* Initial semantic stack: empty environment */
  σ:={}; /* Initial memory: empty (no variables) */
  while (ST≠{}) **do**
  pop(ST, SI); /* Pop semantic instruction into SI */
  (ST,σ):=rule(SI, (ST,σ)); /* Execute SI */
  end
  end

- While the semantic stack is nonempty, pop the instruction at the top of the semantic stack, and execute it according to its semantic rule
- Each instruction of the kernel language has a rule that defines its execution in the abstract machine
- (Note: When we introduce concurrency, we will extend this algorithm to run with more than one semantic stack)

The example instruction in kernel language

```
local X in
   local B in
      B=true
      if B then X=1 else skip end
   end
end
```

Start of the execution: the initial execution state

```
([(local X in
     local B in
       B=true
       if B then X=1 else skip end
     end
  end, {})],
 {})
```

- We start with an empty memory and an empty environment

The *local X in ... end* instruction

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

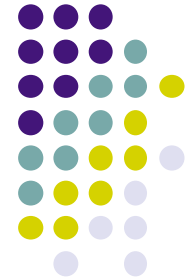
- We create a new variable *x* in memory
- We put the inner instruction on the stack and add X→*x* to its environment

The *local B in ... end* instruction

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

- We create a new variable *b* in memory
- We put the inner instruction on the stack and add B→*b* to its environment

The sequential composition instruction



$$([(B=\textbf{true}, \{B \rightarrow b, X \rightarrow x\}),$$
$$(\textbf{if } B \textbf{ then } X=1$$
$$\textbf{else skip end}, \{B \rightarrow b, X \rightarrow x\})],$$
$$\{b,x\})$$

- We split the sequential composition into its two parts
- We put the two instructions on the stack
- The environments stay the same

The *B=true* instruction

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

- We bind *b* to **true** in memory

The conditional instruction

$$([(X=1,\{B \rightarrow b, X \rightarrow x\})],$$
$$\{b=\textbf{true}, x\})$$

- We read the value of B
- Since B is **true**, it puts the instruction after **then** on the stack
- If B is **false**, it will put the instruction after **else** on the stack
- If B has any other value, then the conditional raises an error

- (Note: If B is unbound then the execution of the semantic stack stops until B becomes bound – this can only happen in another semantic stack, i.e., with concurrency)

The *X=1* instruction

```
([],
 {b=true, x=1})
```

- We bind *x* to 1 in memory
- Execution stops because the stack is empty

Semantic rules we have seen

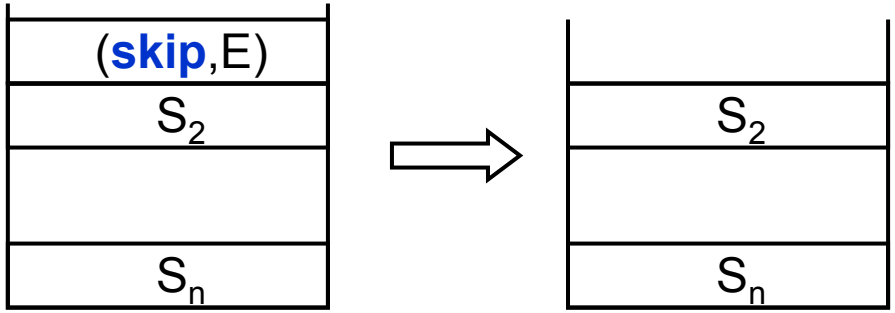
- This example has shown us the execution of four instructions:
  - **local** <x> **in** <s> **end** (variable creation)
  - $<s>_1$ $<s>_2$ (sequential composition)
  - **if** <x> **then** $<s>_1$ **else** $<s>_2$ **end** (conditional)
  - <x>=<v> (assignment)
- In the next unit we will see the semantic rules corresponding to these instructions

Semantic rules for kernel language instructions

- For each instruction in the kernel language, we will define its rule in the abstract machine
- Each instruction takes one execution state as input and returns one execution state
  - Execution state = semantic stack + memory
- Let's look at three instructions in detail:
  - **skip**
  - $\langle s \rangle_1$ $\langle s \rangle_2$ (sequential composition)
  - **local** <x> **in** <s> **end**
- We will see the others in less detail. You can learn about them in the exercises and in the book.

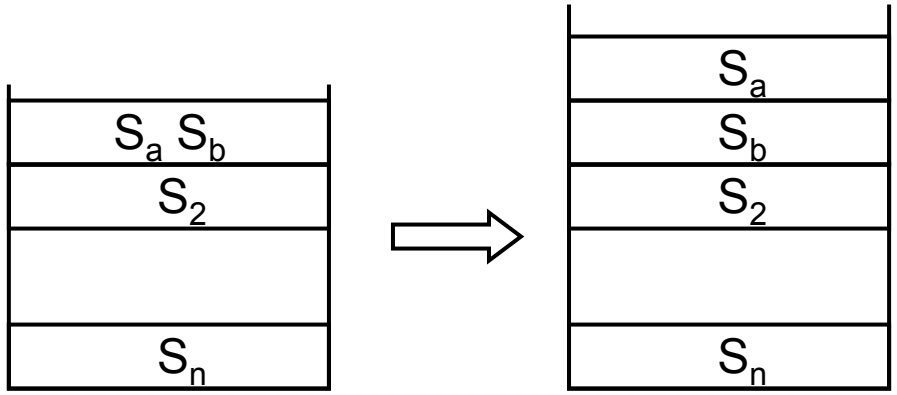
skip

- The simplest instruction
- It does nothing at all!
- Input state: ([(**skip**,E), S_2, …, S_n], σ)
- Output state: ([S_2, …, S_n], σ)
- That's all



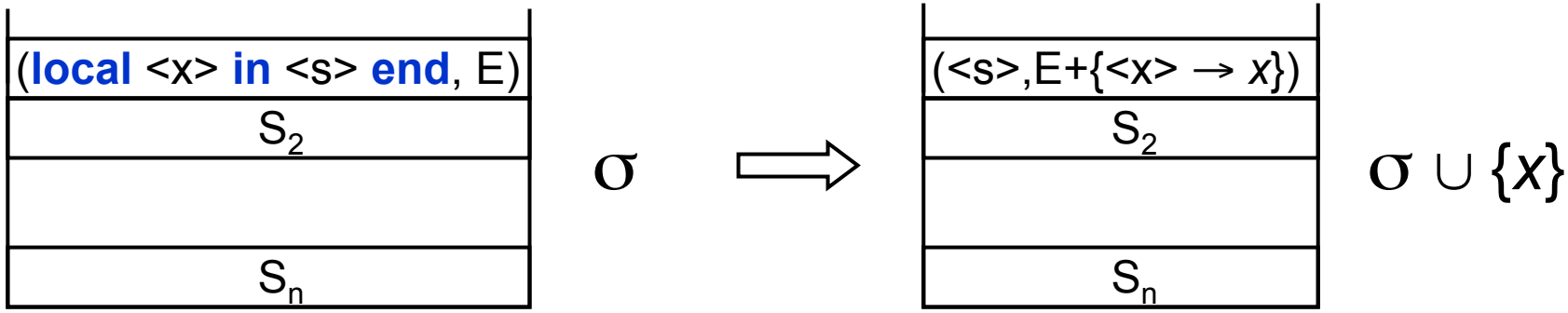
$\langle s \rangle_1 \langle s \rangle_2$ (sequential composition)

- Almost as simple as **skip**
- The instruction removes the top of the stack and adds two new elements
- Input state: $([(S_a\ S_b), S_2, \ldots, S_n], \sigma)$
- Output state: $([S_a, S_b, S_2, \ldots, S_n], \sigma)$

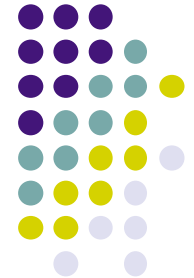


local <x> in <s> end

- Create a fresh new variable x in memory σ
- Add the link {X → *x*} to the environment E (using adjunction)



Some comments on the other instructions



- <x>=<v> (value creation + assignment)
  - Note: when <v> is a procedure, you have to create the contextual environment
- **if** <x> **then** $\langle s \rangle_1$ **else** $\langle s \rangle_2$ **end** (conditional)
  - Note: if <x> is unbound, the instruction will wait ("block") until <x> is bound to a value
  - The activation condition: "<x> is bound to a value"
- **case** <x> **of** <p> **then** $\langle s \rangle_1$ **else** $\langle s \rangle_2$ **end**
  - Note: **case** statements with more patterns are built by combining several kernel instructions
- {<x> $\langle y \rangle_1$ ... $\langle y \rangle_n$}
  - Note: since procedure definition and procedure call are the foundation of data abstraction, we will take a special look!

Procedure definition and procedure call

- Procedure definition and call are very important instructions, since they are the foundation of data abstraction
  - Higher-order programming
  - Layered program organization
  - Encapsulation
  - Object-oriented programming
  - Abstract data types
- This is why we will look at them separately

Procedure semantics

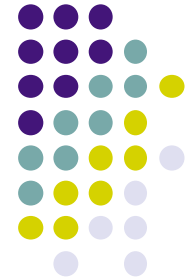
- Procedure definition
  - Create the contextual environment
  - Store the procedure value, which contains both procedure code and contextual environment
- Procedure call
  - Create a new environment by combining two parts:
    - The procedure's contextual environment
    - The formal arguments (identifiers in the procedure definition), which are made to reference the actual argument values
  - Execute the procedure body with this new environment
- We first give an example execution to show what the semantic rules have to do

Procedure example (1)

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

- The free identifiers of the procedure (here, just Z) are the ones declared outside the procedure
- When executing P, the identifier Z must be known
- Z is part of the procedure's contextual environment, which must be part of the procedure's definition

Procedure example (2)



```
local P in
   local Z in
      Z=1
      proc {P X Y} Y=X+Z end  % CE_P = {Z→z}
   end
   local B A in
      A=10
      {P A B}                 % P's body Y=X+Z must do b=a+z
      {Browse B}              % Therefore: E_P = {Y→b, X→a, Z→z}
   end
end
```

Semantic rule for procedure definition

- Semantic instruction:
  (<x>=**proc** {$ <x>$_1$ … <x>$_n$} <s> **end**, E)
  - Formal arguments:
    <x>$_1$, …, <x>$_n$
  - Free identifiers in <s>:
    <z>$_1$, …, <z>$_k$
  - Contextual environment:
    CE=$E_{|<z>1, ..., <z>k}$ (restriction of E)

- Create the following binding in memory:
  x=(**proc** {$ <x>$_1$ … <x>$_n$} <s> **end**, CE)

Semantic rule for procedure call (1)

- Semantic instruction:

  $(\{\langle x\rangle\ \langle y\rangle_1 \ldots \langle y\rangle_n\}, E)$

- If the activation condition is false ($E(\langle x\rangle)$ unbound)
  - Suspension (wait, do not execute)
- If $E(\langle x\rangle)$ is not a procedure
  - Raise an error condition
- If $E(\langle x\rangle)$ is a procedure with the wrong number of arguments ($\neq n$)
  - Raise an error condition

Semantic rule for procedure call (2)

- Semantic instruction on stack:

  $(\{\langle x\rangle\ \langle y\rangle_1\ \ldots\ \langle y\rangle_n\}, E)$

  with procedure definition in memory:

  $E(\langle x\rangle) = (\textbf{proc}\ \{\$\ \langle z\rangle_1 \ldots \langle z\rangle_n\}\ \langle s\rangle\ \textbf{end}, CE)$

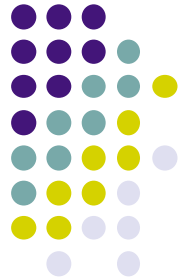
- Put the following instruction on the stack:

  $(\langle s\rangle, CE + \{\langle z\rangle_1 \rightarrow E(\langle y\rangle_1), \ldots, \langle z\rangle_n \rightarrow E(\langle y\rangle_n)\})$

Bringing it all together

- Defining the semantics brings many concepts together
  - Concepts we have seen before: identifier, variable, environment, instruction, procedure value, kernel language
  - New concepts: semantic instruction, semantic stack, memory, execution state, execution, abstract machine
- We gave semantic rules for the kernel language instructions, to show how they execute in the abstract machine
- We used the semantics to prove program correctness, by using it as bridge between specification and program

Discrete mathematics



- The abstract machine is built with discrete mathematics
- For our students at UCL, it is the first time they see a complicated system built with discrete mathematics!
  - Even engineering students, who are quite used to integrals, differential equations, and complex analysis, which are all continuous mathematics
- Discrete mathematics is important because that's how computing systems work (both software and hardware)
  - Surprising behavior and bugs become less surprising if you understand the discrete mathematics of computing systems
  - Too often, continuous models are used for computing systems
  - All this applies to the real world as well (beyond computing systems)

Why semantics is important

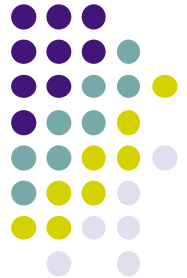
- Semantics is part of programming
  - As a programmer, you are extending the system's semantics: you are writing specifications, designing and implementing abstractions (which we will see soon), and reasoning about your work
- The design of any complicated system with parts that interact in interesting ways (like programming languages and programs) should be done hand in hand with designing a semantics
  - Designing a *simple* semantics is the only way to avoid unpleasant surprises and to guarantee a simple mental model
  - You don't need to understand the semantics to take advantage of it: its mere existence is enough
    - So users of your system will also reap the benefits of a simple semantics
- « Semantics is the ultimate programming language »
  - Invariants as the ultimate loop construct
  - Data abstractions as new kernel language instructions

Using semantics

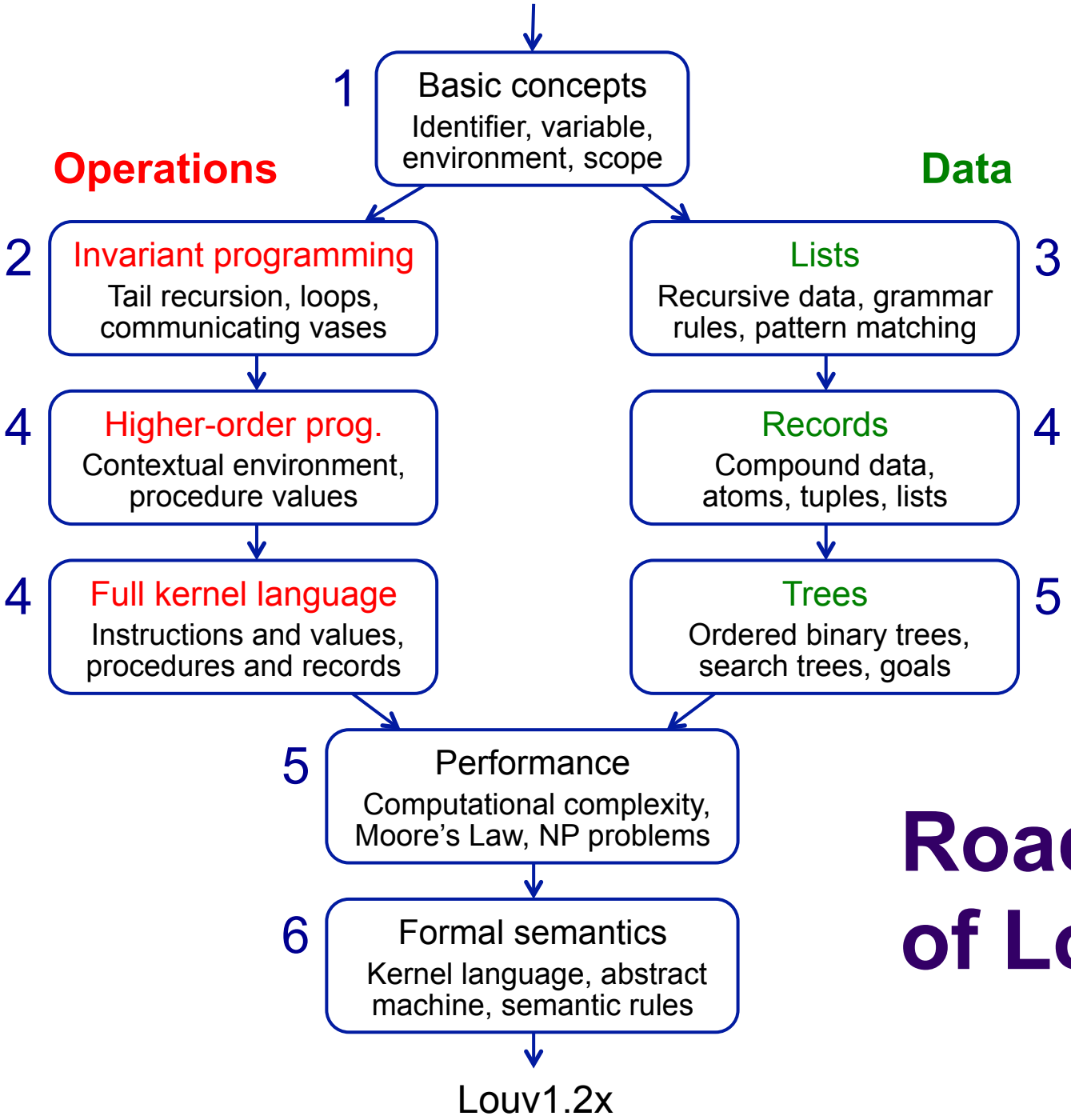
- Semantics has many uses:
  - For design (ensuring the design is simple and predictable)
  - For understanding (the nooks and crannies of programs)
  - For verification (correctness)
  - For debugging (a bug is only a bug with respect to a correct execution)
  - For visualization (a visual representation must be correct)
  - For education (pedagogical uses of semantics)
  - For program analysis and compiler design

- We don't need to bring in details of the processor architecture or compiler in order to understand many things about programs
  - For example, our semantics can be used to understand garbage collection (explained in the textbook)
  - We will use the semantics when needed in the rest of the course

Conclusions for Louv1.1x



- This is the last lesson of Louv1.1x
  - Only the final exam is left: be careful, you only have two tries for each question!
- We have covered a lot of ground!
  - It is worthwhile revisiting some videos in the previous lessons: you will understand more
  - We have seen these concepts in terms of functional programming, but they remain valid for all paradigms
- Let's briefly recapitulate what we have seen



Road map of Louv1.1x

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

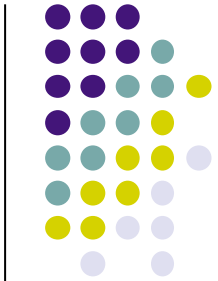
Next Paradigms
Toward Louv1.2x

- Higher-order programming
  - The foundation of data abstraction and object-oriented programming
- Single assignment
  - The foundation of deterministic dataflow concurrency
- Kernel language approach
  - The basis of all the paradigms we will see: they are extensions of the functional kernel language

Final words

- We hope you enjoyed this course
  - Despite, or perhaps because of, the unconventional approach and language
  - We don’t like to follow fashions in programming, we try to understand things as they are
- Louv1.2x sees many more concepts and is every bit as rich and challenging as Louv1.1x
  - We hope you will take the plunge and continue with Louv1.2x

Many important ideas



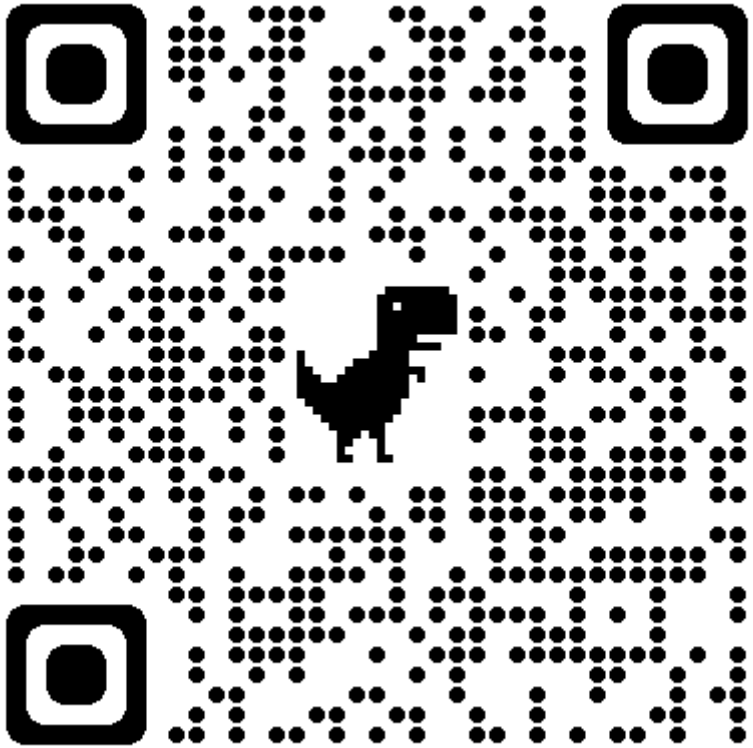
Louv1.1x

- 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

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