

# LINFO1104

## Concepts, paradigms, and semantics of programming languages

### Lecture 2

Peter Van Roy

ICTEAM Institute  
Université catholique de Louvain

[peter.vanroy@uclouvain.be](mailto:peter.vanroy@uclouvain.be)



1

## Overview of lecture 2

- Refresher of lecture 1
- Symbolic programming
  - Lists
  - Pattern matching
  - Trees
  - Tuples and records
- Formal semantics
  - Kernel language
  - Abstract machine
  - Proving correctness of programs
  - Semantic rules for kernel instructions
  - Semantics of procedures



2

# Refresher of lecture 1



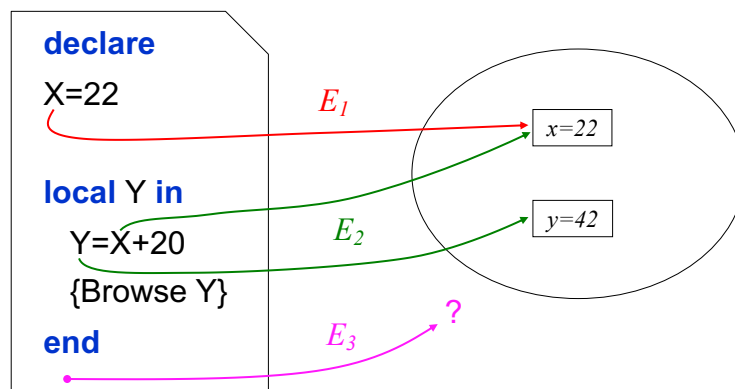
3

## Program and memory...



Program text

System memory



4

© 2013 P. Van Roy. All rights reserved.

4

## Environment



- Environments  $E_1, E_2, E_3$ 
  - Function from identifiers to memory variables
  - A set of pairs  $X \rightarrow x$ 
    - Identifier  $X$ , memory variable  $x$
- Example environment  $E_2$ 
  - $E_2 = \{X \rightarrow x, Y \rightarrow y\}$
  - $E_2(X) = x$
  - $E_2(Y) = y$

5

## 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
```

6

## 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
```

7

## What is the scope of **P**?



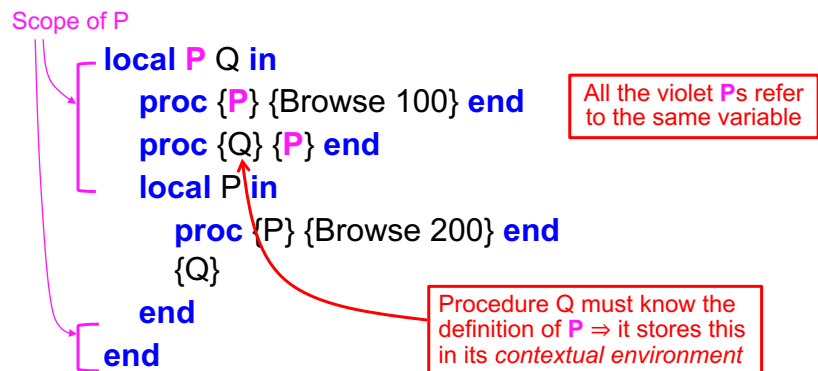
Scope of P

```
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

8

## Contextual environment of Q



9

## The 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**

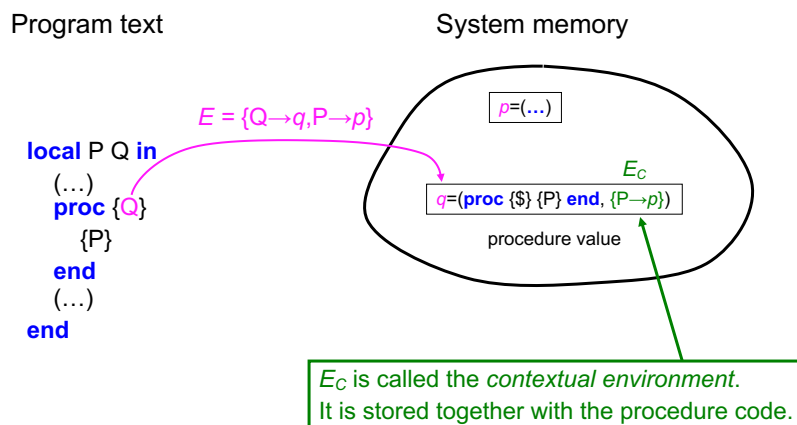
**A**=1

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

- The contextual environment of Inc is  $E_c = \{A \rightarrow a\}$ 
  - Where  $a$  is a variable in memory:  $a=1$

10

## How procedure Q is stored in memory



11

## Procedure values



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

$inc = (\text{proc } \{\$ X Y\} Y = X + A \text{ end}, \{A \rightarrow a\})$

Procedure code      Contextual environment

(without a name: '\$' replaces the identifier)

- The variable *inc* is bound to the procedure value
  - Terminology: a procedure value is also called a *closure* or a *lexically scoped closure*, because it “closes” over the free identifiers when it is defined

12

## How Q is defined and called



- Recall the definition of Q:  
`proc {Q} {P} end`  
When Q is defined, an environment  $E_c$  is created that contains P and  $E_c$  is stored together with Q's code
  - $E_c = \{P \rightarrow p\}$  is called the **contextual environment** of Q
- When Q is called,  $E_c$  is used to get the right value  $p$ 
  - This is guaranteed to always get the right value, even if there is another definition of P right next to the call of Q
- The identifiers in  $E_c$  are the identifiers inside Q that are defined outside of Q
  - They are called the **free identifiers** of Q

© 2013 P. Van Roy. All rights reserved.

13

13

## Free identifiers



- A **free identifier** of an instruction is an occurrence of an identifier inside the instruction that is declared outside the instruction
- The instruction:  
`local Q in  
 proc {Q A} {P A+1} end  
end`  
has one free identifier:  
 $\{P\}$
- The instruction:  
`local Z in Z=X+Y end`  
has two free identifiers:  
 $\{X, Y\}$

© 2013 P. Van Roy. All rights reserved.

14

14

# Lists



15

## Definition of a list



- A list is a **recursive** type: defined in terms of itself
  - Recursion is used both for computations and data!
  - We also use recursion for 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

16



## 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

17

## Some examples of lists



- According to the definition (if T is integer type):

nil  
10 | nil  
10 | 11 | nil  
10 | 11 | 12 | nil  
10 | 11 | 12 | 13 | nil

18

## 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.

19

## 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.

20

## Representations for lists



- The EBNF rule gives one textual representation

- $\langle \text{List } \langle \text{Int} \rangle \rangle \Rightarrow$   
10 |  $\langle \text{List } \langle \text{Int} \rangle \rangle \Rightarrow$   
10 | 11 |  $\langle \text{List } \langle \text{Int} \rangle \rangle \Rightarrow$   
10 | 11 | 12 |  $\langle \text{List } \langle \text{Int} \rangle \rangle \Rightarrow$   
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

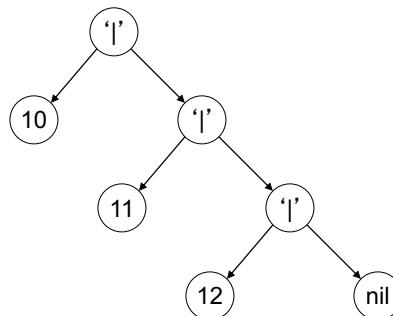
- Bracket notation: [10 11 12]
- In memory, [10 11 12] is identical to 10 | 11 | 12 | nil
- Different textual representations of the same thing are called **syntactic sugar**

21

## Graphical representation of a list

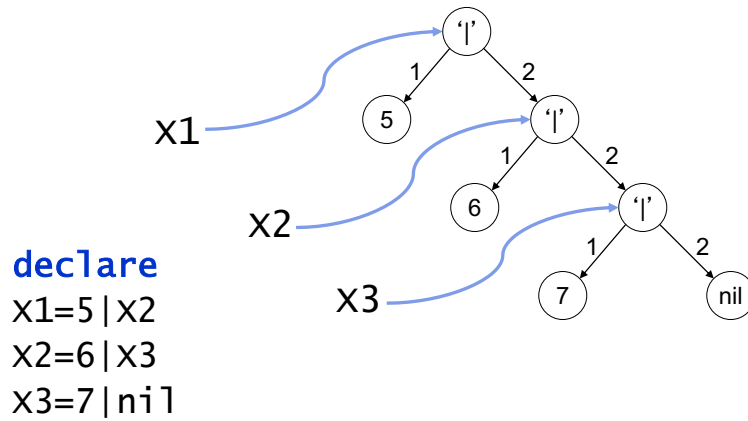


- Graphical representations are very useful for reasoning
  - Humans have very powerful visual reasoning abilities
- We start from the leftmost pair, namely 10 |  $\langle \text{List } \langle \text{Int} \rangle \rangle$ 
  - We draw three nodes with arrows between them
  - We then replace the node 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



22

## Building a list incrementally



© 2012 P. Van Roy. All rights reserved.

23

23

## Computing with lists



- A non-empty list is a pair of head and tail
- Accessing the head:  
X.1
- Accessing the tail:  
X.2
- Comparing the list with nil:  
`if X==nil then ... else ... end`

© 2012 P. Van Roy. All rights reserved.

24

24

## Head and tail functions



- We can define functions

```
fun {Head Xs}  
  Xs.1  
end
```

```
fun {Tail Xs}  
  Xs.2  
end
```

© 2012 P. Van Roy. All rights reserved.

25

25

## Example with Head and Tail



- {Head [a b c]}  
returns a
- {Tail [a b c]}  
returns [b c]
- {Head {Tail {Tail [a b c]}}}  
returns c
- Draw the graphical picture of [a b c]!

© 2012 P. Van Roy. All rights reserved.

26

26

## Functions on lists



27

## 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 Sum function to compute the sum of elements of a list
  - We give first the naïve version and then the smart version (based on invariants)

28

## Sum of list elements



- We are given a list of integers
- We would like to calculate their sum
  - We will define the function “Sum”
- Inductive definition following the list structure
  - Sum of an empty list: 0
  - Sum of a non-empty list L:  $\{\text{Head } L\} + \{\text{Sum } \{\text{Tail } L\}\}$

© 2012 P. Van Roy. All rights reserved.

29

29

## Sum of list elements (naïve method)



```
fun {Sum L}
  if L==nil then
    0
  else
    {Head L} + {Sum {Tail L}}
  end
end
```

© 2012 P. Van Roy. All rights reserved.

30

30

## Sum of list elements (with accumulator)



```
fun {Sum2 L A}
  if L==nil then
    A
  else
    {Sum2 {Tail L} A+{Head L}}
  end
end
```

What is the invariant?

© 2012 P. Van Roy. All rights reserved.

31

31

## Another example: Nth function



- Define the function {Nth L N} which returns the **nth element** of L
- The type of Nth is:  
`<fun {$ <List T> <Int>}:<T>>`
- Reasoning:
  - If  $N=1$  then the result is {Head L}
  - If  $N>1$  then the result is {Nth {Tail L}  $N-1$ }

© 2012 P. Van Roy. All rights reserved.

32

32



## The Nth function



- The complete definition:  
**fun** {Nth L N}  
  **if** N==1 **then** {Head L}  
  **elseif** N>1 **then**  
    {Nth {Tail L} N-1}  
  **end**  
**end**
- What happens if the nth element does not exist?

© 2012 P. Van Roy. All rights reserved.

33

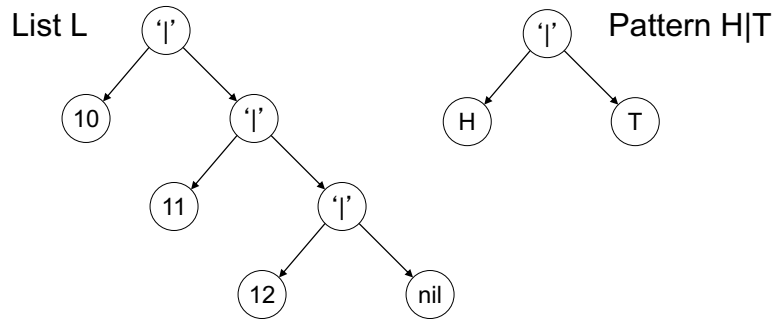
33

## Pattern matching



34

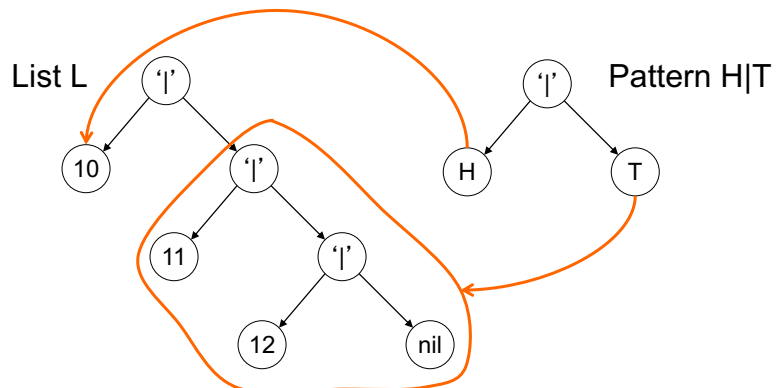
## Pattern matching



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

35

## Pattern matching



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

36

## Sum with pattern matching

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

© 2012 P. Van Roy. All rights reserved.

37

37

## Sum with pattern matching

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

*A clause* →

- “nil” is the *pattern* of the clause

© 2012 P. Van Roy. All rights reserved.

38

38

## Sum with pattern matching

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

A clause

- “H|T” is the *pattern* of the clause

© 2012 P. Van Roy. All rights reserved.

39

39

## Pattern matching

- The first clause uses **of**, the others use **[]**
- Clauses are tried in their textual order
- A clause matches if its pattern matches
- A pattern matches if its label and its arguments match
  - The identifiers in the pattern are assigned to their corresponding values in the input
- The first matching clause is executed, following clauses are ignored

© 2012 P. Van Roy. All rights reserved.

40

40

# Kernel language introduction



41

## The kernel language



- The kernel language is the **first part** of the formal semantics of a programming language
  - The **second part** is the *abstract machine* which we will see later on
- Remember in lecture 1, we explained that each programming paradigm has a simple core language called its kernel language
  - We now introduce the kernel language of functional programming
- All programs in functional programming can be translated into the kernel language
  - All intermediate results of calculations are visible ← Kernel principle
  - All functions become procedures with one extra argument
  - Nested function calls are unnested by introducing new identifiers

42

## Length of a list



```
fun {Length Xs N}
  case Xs
  of nil then N
  [] X|Xr then {Length Xr N+1}
  end
end
```

© 2012 P. Van Roy. All rights reserved.

43

43

## Length of a list translated into kernel language



- The instruction **case** (with one pattern) is part of the kernel language:

```
proc {Length Xs N R}
  case Xs
  of nil then R=N
  else
    case Xs
    of X|Xr then
      local N1 in
        N1=N+1
        {Length Xr N1 R}
      end
    else
      raise typeError end /* type error: see later in the course! */
    end
  end
end
```

© 2012 P. Van Roy. All rights reserved.

44

44

## A function is a procedure with one extra argument



- The kernel language does not need functions
  - It's enough to have procedures
  - Factored design: each concept occurs only once
- A function is translated as a procedure with one extra argument, which gives the function's result
- $N = \{\text{Length } L \ Z\}$   
is equivalent to:  
 $\{\text{Length } L \ Z \ N\}$

45

## Translating to kernel language



- All practical programs can be translated into kernel language
- How to translate:
  - Only kernel language instructions can be used
  - The consequence is that all « hidden » variables become visible
    - Functions become procedures with one extra argument
    - Nested expressions become sequences, with extra local identifiers
    - Each pattern has its own case statement
  - The kernel language is a subset of Oz!
    - It can be executed in Mozart
- Consequences:
  - Kernel programs are longer
  - It is easy to see when programs are tail-recursive
  - It is easy to see exactly how programs execute

46

## Kernel language of the functional paradigm (so far)



- $\langle s \rangle ::=$  skip  
|  $\langle s \rangle_1 \langle s \rangle_2$   
| **local**  $\langle x \rangle$  **in**  $\langle s \rangle$  **end** This is almost complete!  
|  $\langle x \rangle_1 = \langle x \rangle_2$   
|  $\langle x \rangle = \langle v \rangle$   
| **if**  $\langle x \rangle$  **then**  $\langle s \rangle_1$  **else**  $\langle s \rangle_2$  **end**  
| **proc**  $\{ \langle x \rangle \langle x \rangle_1 \dots \langle x \rangle_n \}$   $\langle s \rangle$  **end**  
|  $\{ \langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n \}$   
| **case**  $\langle x \rangle$  **of**  $\langle p \rangle$  **then**  $\langle s \rangle_1$  **else**  $\langle s \rangle_2$  **end**
- $\langle v \rangle ::= \langle \text{number} \rangle \mid \langle \text{list} \rangle \mid \dots$  ← still something missing
- $\langle \text{number} \rangle ::= \langle \text{int} \rangle \mid \langle \text{float} \rangle$
- $\langle \text{list} \rangle, \langle p \rangle ::= \text{nil} \mid \langle x \rangle \mid \langle x \rangle \text{ '}' \langle \text{list} \rangle$

47

## Trees



48



# Trees



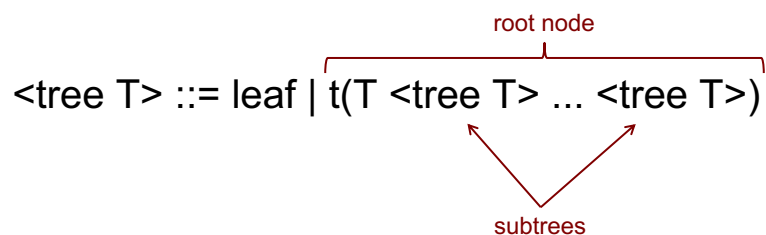
- Trees are the **second most important data structure** in computing, next to lists
  - Trees are extremely useful for efficiently organizing information and performing many kinds of calculations
- Trees illustrate well **goal-oriented programming**
  - Many tree data structures are based on a global 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

49

# Trees



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

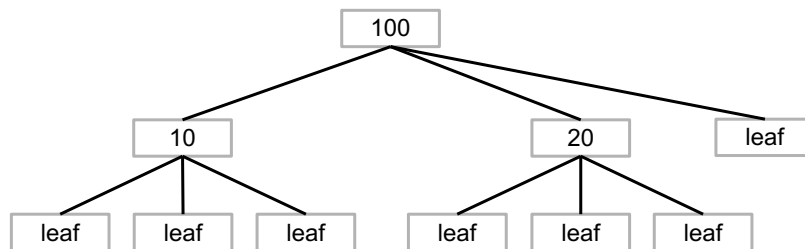


50

## Example tree

- **declare**

$T = t(100 \ t(10 \ \text{leaf} \ \text{leaf} \ \text{leaf}) \ t(20 \ \text{leaf} \ \text{leaf} \ \text{leaf}) \ \text{leaf})$



51

## 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

$\langle \text{tree } T \rangle ::= \text{leaf} \mid t(T \langle \text{tree } T \rangle \dots \langle \text{tree } T \rangle)$

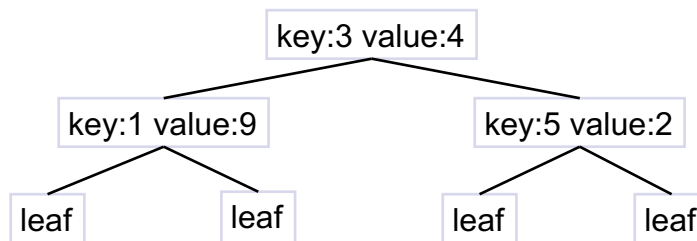
$\langle \text{list } T \rangle ::= \text{nil} \mid ' '(T \langle \text{list } T \rangle)$

Notice the  
similarity with lists!

52

## Ordered binary tree (1)

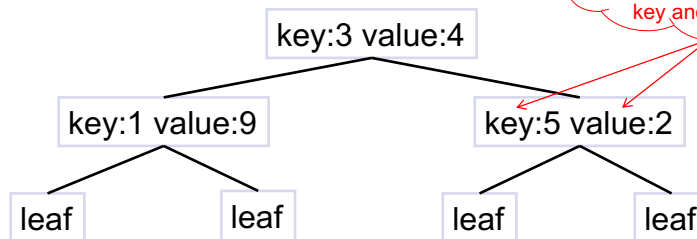
- `<obtree T> ::= leaf`  
                   | `tree(key:T value:T left:<obtree T> right:<obtree T>)`
- **Binary**: each non-leaf tree has two subtrees (named left and right)
- **Ordered**: for each tree (including all subtrees):  
                 all keys in the left subtree < key of the root  
                 key of the root < all keys in the right subtree



53

## Ordered binary tree (2)

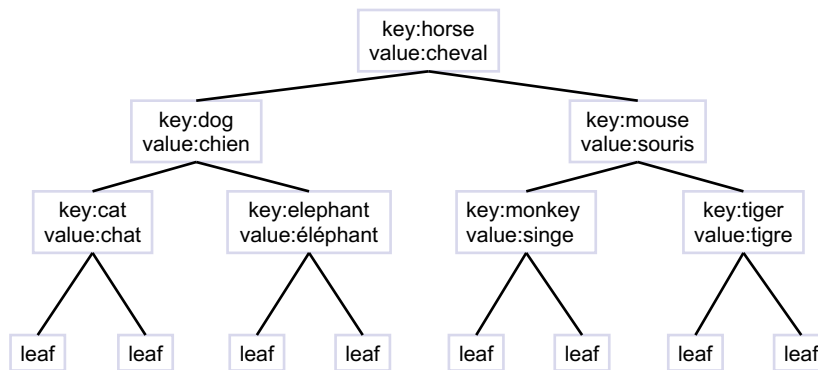
- `<obtree T> ::= leaf`  
                   | `tree(key:T value:T left:<obtree T> right:<obtree T>)`
- **Binary**: each non-leaf tree has two subtrees (named left and right)
- **Ordered**: for each tree (including all subtrees)  
                 all keys in the left subtree < key of the root  
                 key of the root < all keys in the right subtree



54

## Ordered binary tree (3)

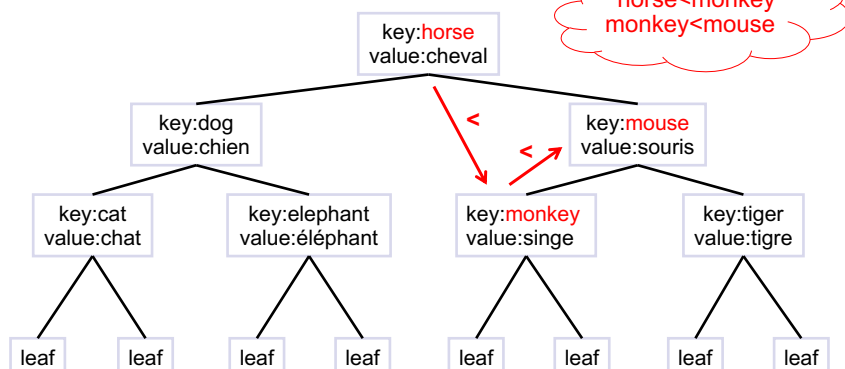
- This ordered binary tree is a translation dictionary from English to French



55

## Ordered binary tree (4)

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



56

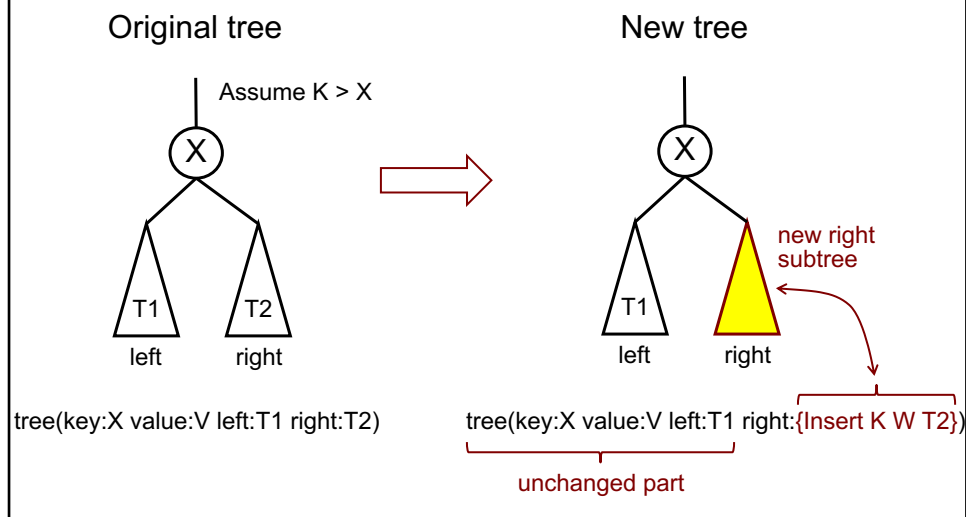
## Search tree



- **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 with added (K,W)
  - **{Delete K T}**: returns a new tree that does not contain K

57

## Inserting a new key/value pair



58

## Efficiency of Lookup



- 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  $\log_2 n$
  - 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

59

## Looking up information



- There are four possibilities:
  - K is not found
  - K is found
  - K might be in the left subtree
  - K might be in the right subtree
- ```
fun {Lookup K T}
  case T
  of leaf then notfound
  [] tree(key:Y value:V T1 T2) andthen K==Y then
    found(V)
  [] tree(key:Y value:V T1 T2) andthen K<Y then
    {Lookup K T1}
  [] tree(key:Y value:V T1 T2) andthen K>Y then
    {Lookup K T2}
  end
end
```

60

## Inserting information



- There are four possibilities:
- (K,W) replaces a leaf node
- (K,W) replaces an existing node
- (K,W) is inserted in the left subtree
- (K,W) is inserted in the right subtree

```

fun {Insert K W T}
  case T
  of leaf then tree(key:K value:W leaf leaf)
  [] tree(key:Y value:V T1 T2) andthen K==Y then
    tree(key:K value:W T1 T2)
  [] tree(key:Y value:V T1 T2) andthen K<Y then
    tree(key:Y value:V {Insert K W T1} T2)
  [] tree(key:Y value:V T1 T2) andthen K>Y then
    tree(key:Y value:V T1 {Insert K W T2})
  end
end
  
```

61

## Deleting information



- There are four possibilities:
- (K,\_) is not in the tree
- (K,\_) is removed immediately
- (K,\_) is removed from the left subtree
- (K,\_) is removed from the right subtree
- Right?

```

fun {Delete K T}
  case T
  of leaf then leaf
  [] tree(key:Y value:W T1 T2) andthen K==Y then
    leaf
  [] tree(key:Y value:W T1 T2) andthen K<Y then
    tree(key:Y value:W {Delete K T1} T2)
  [] tree(key:Y value:W T1 T2) andthen K>Y then
    tree(key:Y value:W T1 {Delete K T2})
  end
end
  
```

62

## Deleting information



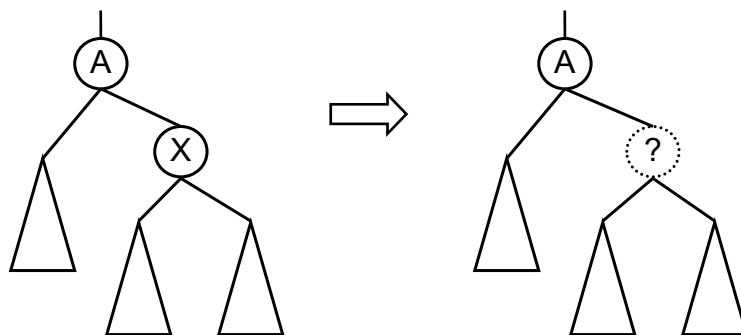
- There are four possibilities:
- (K,\_) is not in the tree
- (K,\_) is removed immediately
- (K,\_) is removed from the left subtree
- (K,\_) is removed from the right subtree
- Right? **WRONG!**

```

fun {Delete K T}
  case T
  of leaf then leaf
  [] tree(key:Y value:W T1 T2) andthen K==Y then
    leaf
  [] tree(key:Y value:W T1 T2) andthen K<Y then
    tree(key:Y value:W {Delete K T1} T2)
  [] tree(key:Y value:W T1 T2) andthen K>Y then
    tree(key:Y value:W T1 {Delete K T2})
  end
end
  
```

63

## Deleting an element from an ordered binary tree

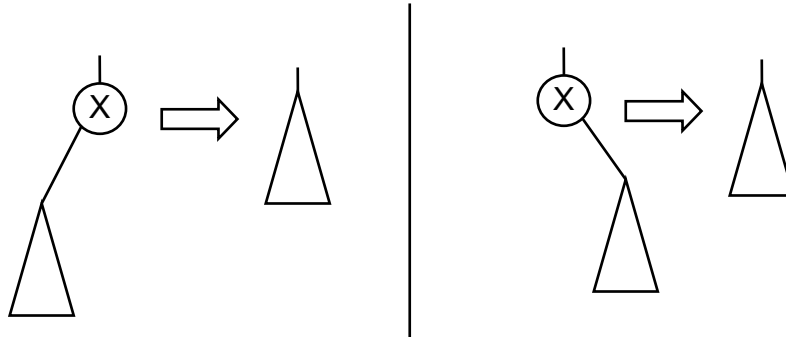


The problem is to **repair the tree** after X disappears

64



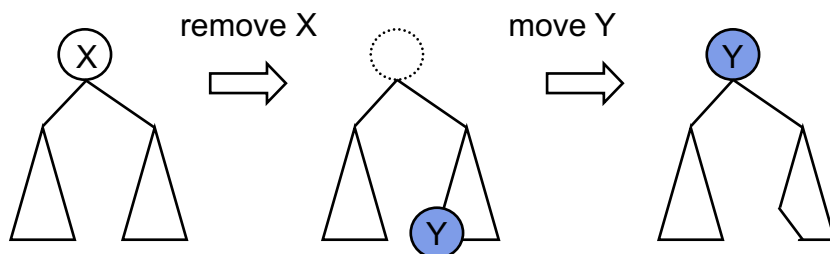
## Deleting the root when one subtree is empty



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

65

## Deleting the root when both subtrees are not empty

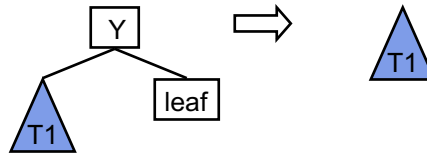


The idea is to fill the "hole" that appears after X is removed. We can put there the smallest element in the right subtree, namely Y.

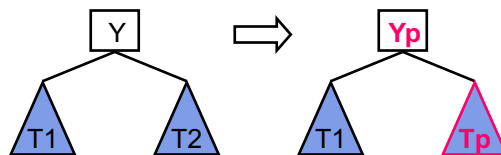
66

## Deleting the root

- To remove the root Y, there are two possibilities:



- One subtree is a leaf. Just return the other.
- Neither subtree is a leaf. Remove an element from one of its subtrees.

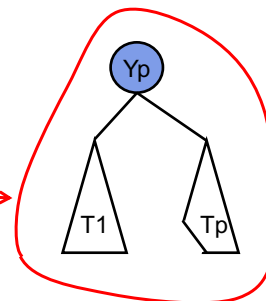


67

## 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

68

## 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

69

## Delete operation is complex



- Why is the delete operation so complex?
- It is because the tree satisfies a **global condition**, namely it is ordered
- The delete operation has to work to keep this condition true
- Many tree algorithms depend on global conditions and must work to keep the conditions true
- The interesting thing about a global condition is that it gives the tree a **spark of life**: the tree behaves a bit like it is alive (« **goal-oriented behavior** »)
  - Living organisms have goal-oriented behavior

70

## Goal-oriented programming



- Many tree algorithms depend on global properties and most of the work they do is in maintaining these properties
  - The ordered binary tree satisfies a global ordering condition. The insert and delete operations must maintain this condition. This is easy for insert, but harder for delete.
- Goal-oriented programming is widely used in artificial intelligence algorithms
  - It can give unexpected results as the algorithm does its thing to maintain the global property.
  - Goal-oriented behavior is characteristic of living organisms. So defining algorithms that are goal-oriented gives them a spark of life!

71

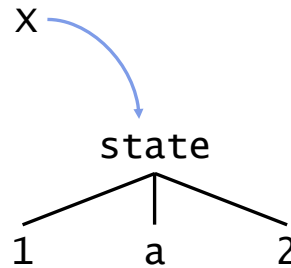
## Tuples and records



72

# Tuples

X=state(1 a 2)

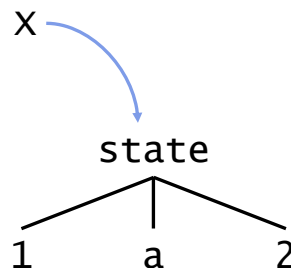


- A tuple allows grouping several values together
  - For example: 1, a, 2
  - The position is meaningful: first, second, third!
- A tuple has a label
  - For example: state

73

# Operations on tuples

X=state(1 a 2)



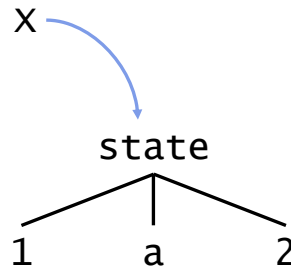
- {Label X} returns *the label* of tuple X
  - For example: state
  - The label is a constant, called an atom
- {width X} returns *the width* (number of fields)
  - For example: 3
  - Always a positive integer or zero

74

## Accessing fields ("." operation)



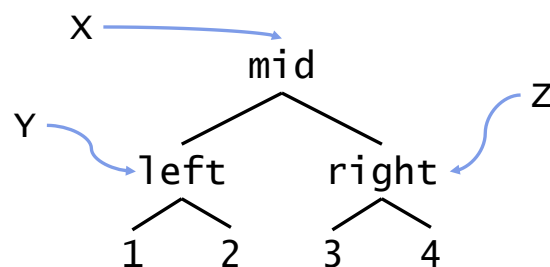
X=state(1 a 2)



- Fields are numbered from 1 up to {width X}
- X.N returns the nth field of tuple X:
  - X.1 returns 1
  - X.3 returns 2
- In the expression X.N, N is called the field name or "feature"

75

## Building a tree



- A tree can be built with tuples:  
**declare**  
Y=left(1 2) Z=right(3 4)  
X=mid(Y Z)

76

## Testing equality (==)



- Equality testing with a number or atom
  - Easy: the number or atom must be the same
- Equality testing of trees
  - Also easy: the two trees must have the same root tuples and the same subtrees in corresponding fields
  - Careful when the tree has a cycle!
    - Comparison with == works, but naïve programs may loop
    - Advice: avoid this kind of tree

77

## Tuples summary



- Tuple
  - Label
  - Width
  - Field
  - Field name, feature
- Accessing fields with “.” operation
- Build trees with tuples
- Pattern matching with tuples
- Comparing tuples with “==”

78

## A list is a tuple



- The list H|T is actually a tuple `'|'` (H T)
- Principle of simplicity in the kernel language: instead of two concepts (tuples and lists), only one concept is needed (tuple)
- Because of their usefulness, lists have a syntactic sugar
  - It is purely for programmer comfort, it makes no difference in the kernel language

79

## Syntax of lists as tuples



- A list is a special case of a tuple
- Prefix syntax (put the label `'|'` in front)
  - Nil  
`'|(5 nil)`  
`'|(5 '|(6 nil))`  
`'|(5 '|(6 '|(7 nil)))`
- Prefix syntax with field names
  - Nil  
`'|(1:5 2:nil)`  
`'|(1:5 2: '|(1:6 2:nil))`  
`'|(1:5 2: '|(1:6 2: '|(1:7 2:nil)))`

© 2012 P. Van Roy. All rights reserved.

80

80



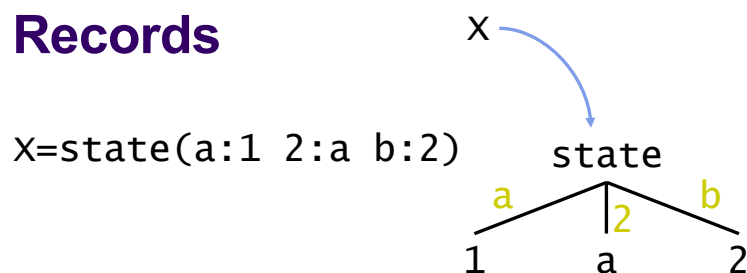
## Records



- A record is a generalization of a tuple
  - Field names can be atoms (i.e., constants)
  - Field names can be any integer
    - Does not have to start with 1
    - Does not have to be consecutive
- A record also has a label and a width

81

## Records



- The position of a field is no longer meaningful
  - Instead, it is the field name that is meaningful
- Accessing fields is done the same as for tuples
  - `x.a=1`

82



## Record operations

- Label and width operations:
  - {Label X}=state
  - {Width X}=3
- Equality test:
  - X==state(a:1 b:2 2:a)
- New operation: **arity**
  - Returns a list of field names
  - {Arity X}=[2 a b] (in lexicographic order)
  - Arity also works for tuples and lists!

83



## A tuple is a record

- The record:  
`x = state(1:a 2:b 3:c)`  
is the same as the tuple:  
`x = state(a b c)`
- In a **tuple**, all fields are numbered consecutively starting with 1
- What happens if we write:  
`x = state(a 2:b 3:c)`  
or  
`x = state(2:b 3:c a)`
- In a **enregistrement**, all fields without name are numbered consecutively starting with 1

84

## A list is a tuple and a tuple is a record $\Rightarrow$ many list syntaxes



- The list syntax  
`x1=5|6|7|nil`  
is a short-cut for  
`x1=5|(6|(7|nil))`  
which is a short-cut for  
`x1='|(5 '|(6 '|(7 nil)))`  
which is a short-cut for  
`x1='|(1:5 2:'|(1:6 2:'|(1:7 2:nil)))`
- The shortest syntax (the 'nil' is implied!)  
`x1=[5 6 7]`

© 2012 P. Van Roy. All rights reserved.

85

85

## The kernel language has only records



- In the kernel language there are only records
  - An atom is a record whose width is 0
  - A tuple is a record whose field names are numbered consecutively starting from 1
    - If this condition is not satisfied, the data structure is still a record but it is no longer a tuple
  - A list is built with tuples nil and '|'(X Y)
- This keeps the kernel language simple
  - It has just one data structure

86

## Kernel language with records



- $\langle s \rangle ::=$  skip  
 $\quad | \langle s \rangle_1 \langle s \rangle_2$   
 $\quad | \text{local } \langle x \rangle \text{ in } \langle s \rangle \text{ end}$   
 $\quad | \langle x \rangle_1 = \langle x \rangle_2$   
 $\quad | \langle x \rangle = \langle v \rangle$   
 $\quad | \text{if } \langle x \rangle \text{ then } \langle s \rangle_1 \text{ else } \langle s \rangle_2 \text{ end}$   
 $\quad | \text{proc } \{ \langle x \rangle \langle x \rangle_1 \dots \langle x \rangle_n \} \langle s \rangle \text{ end}$   
 $\quad | \{ \langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n \}$   
 $\quad | \text{case } \langle x \rangle \text{ of } \langle p \rangle \text{ then } \langle s \rangle_1 \text{ else } \langle s \rangle_2 \text{ end}$
- $\langle v \rangle ::= \langle \text{number} \rangle \mid \langle \text{record} \rangle \mid \dots$
- $\langle \text{number} \rangle ::= \langle \text{int} \rangle \mid \langle \text{float} \rangle$
- $\langle \text{record} \rangle, \langle p \rangle ::= \langle \text{lit} \rangle ( \langle f \rangle_1 : \langle x \rangle_1 \dots \langle f \rangle_n : \langle x \rangle_n )$  ← Records replace lists

87

## Exercises



- Which of these records are tuples?
  - $A = a(1:a \ 2:b \ 3:c)$
  - $B = a(1:a \ 2:b \ 4:c)$
  - $C = a(0:a \ 1:b \ 2:c)$
  - $D = a(1:a \ 2:b \ 3:c \ d)$
  - $E = a(a \ 2:b \ 3:c \ 4:d)$
  - $F = a(2:b \ 3:c \ 4:d \ a)$
  - $G = a(1:a \ 2:b \ 3:c \ \text{foo}:d)$
  - $H = 'l' (1:a \ 2:'l' (1:b \ 2:\text{nil}))$
  - $I = 'l' (1:a \ 2:'l' (1:b \ 3:\text{nil}))$

88

# Introduction to formal semantics



89

## 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
  - This understanding comes with the formal semantics

90

# What is the semantics of a language?



- The **semantics** of a programming language is a **fully precise explanation of how programs execute**
  - With it we can reason about program design and correctness
- We give the semantics for all paradigms of this course
  - We start by giving the semantics of functional programming
- Before taking the plunge, let's take a step back and talk about semantics in general

91

# Different approaches to 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**
    - This works for all paradigms!
  - **Axiomatic semantics**: Explains a program as an **implication**: if certain properties hold before the execution, then some other properties will hold after the execution
    - « If the precondition holds before, then the postcondition will hold after » as shown in LEPL1402
    - This works well for imperative paradigms (like object-oriented programming)
  - **Denotational semantics**: Explains a program as a **function** over an abstract domain, which simplifies certain kinds of mathematical analysis of the program
    - This works well for functional programming languages
  - **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
    - This works well for logic programming languages such as Prolog and constraint programming
- We will focus on operational semantics

92

# 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 in five parts
  1. **The full kernel language** for functional programming
  2. **Executing an example program** on the abstract machine
  3. **Defining the abstract machine** and its semantic rules
  4. **Proving the correctness** of an example program
  5. **Procedure definition and call** are special because they are the foundation of data abstraction. We define the semantic rules of procedure definition and call.

93

## Semantics 1: Full kernel language



94

## Kernel language of functional programming



- We have seen all concepts of functional programming
  - Now 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 **to prove correctness of programs**
- 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

95

## The functional kernel language (what we saw before)



- $\langle s \rangle ::=$ 
  - skip**
  - $\langle s \rangle_1 \langle s \rangle_2$
  - local**  $\langle x \rangle$  **in**  $\langle s \rangle$  **end**
  - $\langle x \rangle_1 = \langle x \rangle_2$
  - $\langle x \rangle = \langle v \rangle$
  - if**  $\langle x \rangle$  **then**  $\langle s \rangle_1$  **else**  $\langle s \rangle_2$  **end**
  - proc**  $\{ \langle x \rangle \langle x \rangle_1 \dots \langle x \rangle_n \}$   $\langle s \rangle$  **end**
  - $\{ \langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n \}$
  - case**  $\langle x \rangle$  **of**  $\langle p \rangle$  **then**  $\langle s \rangle_1$  **else**  $\langle s \rangle_2$  **end**
- $\langle v \rangle ::= \langle \text{number} \rangle \mid \langle \text{list} \rangle \mid \dots$
- $\langle \text{number} \rangle ::= \langle \text{int} \rangle \mid \langle \text{float} \rangle$
- $\langle \text{list} \rangle, \langle p \rangle ::= \text{nil} \mid \langle x \rangle \mid \langle x \rangle ' ' \langle \text{list} \rangle$

This is the kernel language with lists and procedure statements

Still incomplete

96



# The functional kernel language



- $\langle s \rangle ::=$  **skip**
  - |  $\langle s \rangle_1 \langle s \rangle_2$
  - | **local**  $\langle x \rangle$  **in**  $\langle s \rangle$  **end**
  - |  $\langle x \rangle_1 = \langle x \rangle_2$
  - |  $\langle x \rangle = \langle v \rangle$
  - | **if**  $\langle x \rangle$  **then**  $\langle s \rangle_1$  **else**  $\langle s \rangle_2$  **end**
  - | **proc**  $\{ \langle x \rangle \langle x \rangle_1 \dots \langle x \rangle_n \} \langle s \rangle$  **end**
  - |  $\{ \langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n \}$
  - | **case**  $\langle x \rangle$  **of**  $\langle p \rangle$  **then**  $\langle s \rangle_1$  **else**  $\langle s \rangle_2$  **end**
- $\langle v \rangle ::= \langle \text{number} \rangle \mid \langle \text{list} \rangle \mid \dots$
- $\langle \text{number} \rangle ::= \langle \text{int} \rangle \mid \langle \text{float} \rangle$
- $\langle \text{list} \rangle, \langle p \rangle ::= \text{nil} \mid \langle x \rangle \mid \langle x \rangle ' \langle \text{list} \rangle$

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. Records instead of lists (records subsume lists)

97

# The functional kernel language (procedure values)



- $\langle s \rangle ::=$  **skip**
  - |  $\langle s \rangle_1 \langle s \rangle_2$
  - | **local**  $\langle x \rangle$  **in**  $\langle s \rangle$  **end**
  - |  $\langle x \rangle_1 = \langle x \rangle_2$
  - |  $\langle x \rangle = \langle v \rangle$
  - | **if**  $\langle x \rangle$  **then**  $\langle s \rangle_1$  **else**  $\langle s \rangle_2$  **end**
  - | ~~**proc**  $\{ \langle x \rangle \langle x \rangle_1 \dots \langle x \rangle_n \} \langle s \rangle$  **end**~~
  - |  $\{ \langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n \}$
  - | **case**  $\langle x \rangle$  **of**  $\langle p \rangle$  **then**  $\langle s \rangle_1$  **else**  $\langle s \rangle_2$  **end**
- $\langle v \rangle ::= \langle \text{number} \rangle \mid \langle \text{procedure} \rangle \mid \langle \text{list} \rangle \mid \dots$
- $\langle \text{number} \rangle ::= \langle \text{int} \rangle \mid \langle \text{float} \rangle$
- $\langle \text{procedure} \rangle ::= \text{proc } \{ \$ \langle x \rangle_1 \dots \langle x \rangle_n \} \langle s \rangle \text{ end}$
- $\langle \text{list} \rangle, \langle p \rangle ::= \text{nil} \mid \langle x \rangle \mid \langle x \rangle ' \langle \text{list} \rangle$

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

This is called an "anonymous procedure". The procedure name is replaced by a placeholder "\$".

98

## The functional kernel language (records)



- $\langle s \rangle ::=$  **skip**  
 $\mid \langle s \rangle_1 \langle s \rangle_2$   
 $\mid \text{local } \langle x \rangle \text{ in } \langle s \rangle \text{ end}$   
 $\mid \langle x \rangle_1 = \langle x \rangle_2$   
 $\mid \langle x \rangle = \langle v \rangle$   
 $\mid \text{if } \langle x \rangle \text{ then } \langle s \rangle_1 \text{ else } \langle s \rangle_2 \text{ end}$   
 $\mid \{ \langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n \}$   
 $\mid \text{case } \langle x \rangle \text{ of } \langle p \rangle \text{ then } \langle s \rangle_1 \text{ else } \langle s \rangle_2 \text{ end}$
- $\langle v \rangle ::= \langle \text{number} \rangle \mid \langle \text{procedure} \rangle \mid \text{list} \mid \text{record}$
- $\langle \text{number} \rangle ::= \langle \text{int} \rangle \mid \langle \text{float} \rangle$
- $\langle \text{procedure} \rangle ::= \text{proc } \{ \$ \langle x \rangle_1 \dots \langle x \rangle_n \} \langle s \rangle \text{ end}$
- $\text{record}, \langle p \rangle ::= \langle \text{lit} \rangle \mid \langle \text{lit} \rangle (\langle f \rangle_1 : \langle x \rangle_1 \dots \langle f \rangle_n : \langle x \rangle_n)$

2. Records subsume lists

99

## The functional kernel language (complete)



- $\langle s \rangle ::=$  **skip**  
 $\mid \langle s \rangle_1 \langle s \rangle_2$   
 $\mid \text{local } \langle x \rangle \text{ in } \langle s \rangle \text{ end}$   
 $\mid \langle x \rangle_1 = \langle x \rangle_2$   
 $\mid \langle x \rangle = \langle v \rangle$   
 $\mid \text{if } \langle x \rangle \text{ then } \langle s \rangle_1 \text{ else } \langle s \rangle_2 \text{ end}$   
 $\mid \{ \langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n \}$   
 $\mid \text{case } \langle x \rangle \text{ of } \langle p \rangle \text{ then } \langle s \rangle_1 \text{ else } \langle s \rangle_2 \text{ end}$
- $\langle v \rangle ::= \langle \text{number} \rangle \mid \langle \text{procedure} \rangle \mid \langle \text{record} \rangle$
- $\langle \text{number} \rangle ::= \langle \text{int} \rangle \mid \langle \text{float} \rangle$
- $\langle \text{procedure} \rangle ::= \text{proc } \{ \$ \langle x \rangle_1 \dots \langle x \rangle_n \} \langle s \rangle \text{ end}$
- $\text{record}, \langle p \rangle ::= \langle \text{lit} \rangle \mid \langle \text{lit} \rangle (\langle f \rangle_1 : \langle x \rangle_1 \dots \langle f \rangle_n : \langle x \rangle_n)$

Procedure values and records are important basic types. They allow to define data abstractions including all of object-oriented programming.

100

## Semantics 2: Executing with the abstract machine



101

### Executing a program with the abstract machine



- We execute the program using the semantics by following two steps
- First, we translate the program into kernel language
  - We use the kernel language of functional programming
  - All programs 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 see an example execution

102

## The example instruction in kernel language



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

103

## 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, {}),
{})
```

↑ empty environment  
↑ empty memory

- The initial execution state has an empty memory {} and an empty environment {}
- We start execution with **local X in <s> end**

104

## 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  $x$  so the memory becomes  $\{x\}$
- We create a new environment  $\{X \rightarrow x\}$  so that  $X$  can refer to the new variable  $x$

105

## 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  $B \rightarrow b$  to its environment, giving  $\{B \rightarrow b, X \rightarrow x\}$

106

## The sequential composition instruction



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

- We split the sequential composition into its two parts
  - $B=\text{true}$  and  $\text{if } B \text{ then } X=1 \text{ else skip end}$
- We put the two instructions on the stack
- Each instruction gets the same environment

107

## The $B=\text{true}$ instruction



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

- We bind variable  $b$  to  $\text{true}$  in memory

108

## The conditional instruction



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

- 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, as we will see)

109

## The $X=1$ instruction



$([],$   
 $\{b=\text{true}, x=1\})$

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

110

## Semantic rules we have seen



- This example has shown us the execution of four instructions:
  - **local**  $\langle x \rangle$  **in**  $\langle s \rangle$  **end** (variable creation)
  - $\langle s \rangle_1 \langle s \rangle_2$  (sequential composition)
  - **if**  $\langle x \rangle$  **then**  $\langle s \rangle_1$  **else**  $\langle s \rangle_2$  **end** (conditional)
  - $\langle x \rangle = \langle v \rangle$  (assignment)
- We will define the semantic rules corresponding to these instructions

111

## Semantics 3: The abstract machine



112



## 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**  $\langle s \rangle, E$ 
  - An instruction with its environment
- **Semantic stack**  $ST = [\langle s \rangle_1, E_1, \dots, \langle s \rangle_n, E_n]$ 
  - A stack of semantic instructions
- **Execution state**  $(ST, \sigma)$ 
  - A pair of a semantic stack and the memory
- **Execution**  $(ST_1, \sigma_1) \rightarrow (ST_2, \sigma_2) \rightarrow (ST_3, \sigma_3) \rightarrow \dots$ 
  - A sequence of execution states

113

## Abstract machine execution algorithm



- **procedure** execute( $\langle s \rangle$ )  
**begin**  
 $ST := [\langle s \rangle, \{\}];$  /\* Initial semantic stack: one instruction, empty env. \*/  
 $\sigma := \{\};$  /\* Initial memory: empty (no variables) \*/  
**while** ( $ST \neq \{\}$ ) **do**  
 $\text{pop}(ST, SI);$  /\* Pop semantic instruction into SI \*/  
 $(ST, \sigma) := \text{rule}(SI, (ST, \sigma));$  /\* Execute SI according to its rule \*/  
**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)

114

## Semantic rules for kernel language instructions



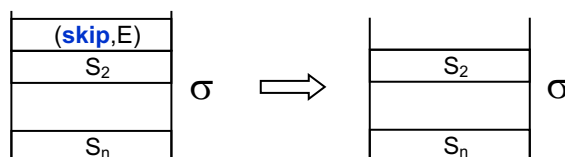
- For each instruction in the kernel language, we will define its rule in the abstract machine
- Each instruction takes one execution state as input and returns one execution state
  - Execution state = semantic stack ST + memory  $\sigma$
- Let's look at three instructions in detail:
  - **skip**
 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

115

## skip



- The simplest instruction
- It does nothing at all!
- Input state:  $([(\text{skip}, E), S_2, \dots, S_n], \sigma)$
- Output state:  $([S_2, \dots, S_n], \sigma)$
- That's all

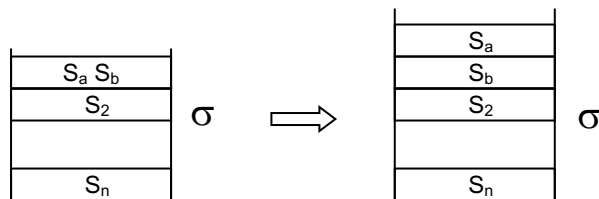


116

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



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

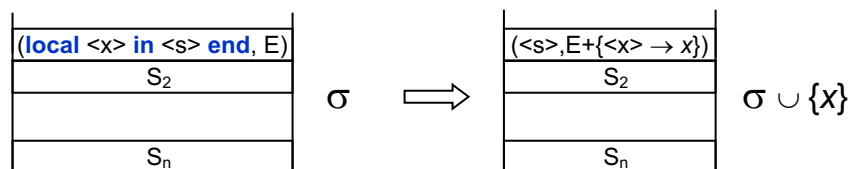


117

## local $\langle x \rangle$ in $\langle s \rangle$ end



- Create a fresh new variable  $x$  in memory  $\sigma$
- Add the link  $\{X \rightarrow x\}$  to the environment  $E$  (using adjunction)



118

## Some other instructions



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

119

## Semantics 4: Proving correctness with the semantics



120

## When is a program correct?



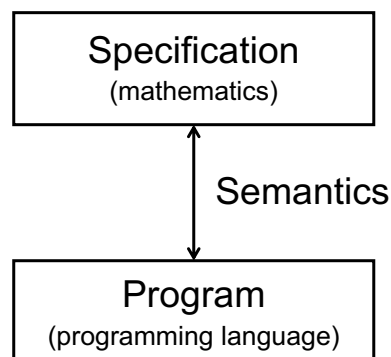
- “A program is correct when it does what we want”
  - How can we be sure?
- We need to make precise what we want it to do:
  - We introduce the concept of **specification**
- We need to prove that the **program** satisfies the **specification**, when it executes according to the **semantics**

121

## 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



122

## Example: correctness of factorial



- The **specification** of {Fact N} (mathematics)  
$$0! = 1$$
$$n! = n \times ((n-1)!) \text{ when } n > 0$$
- The **program** (programming language)  

```
fun {Fact N}  
  if N==0 then 1 else N*{Fact N-1} end  
end
```
- The **semantics** connects the two
  - Executing  $R=\{\text{Fact } N\}$  gives the result  $r=n!$

123

## 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

124

## The inductive proof



- We must show that {Fact N} calculates  $n!$  for all  $n \geq 0$
- **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:**  $(n-1) \rightarrow n$ 
  - The specification says:  $n! = n \times (n-1)!$
  - The execution of {Fact N}, *using the semantics*, gives {Fact N} =  $N \times \{\text{Fact } N-1\}$ 
    - We assume that  $\{\text{Fact } N-1\} = (n-1)!$
    - We assume that the language correctly implements multiplication
    - Therefore:  $\{\text{Fact } N\} = N \times \{\text{Fact } N-1\} = n \times (n-1)! = n!$
    - *It's correct!*
- Now we just need to understand the magic words “*using the semantics*”!

125

## 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 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 definition
  - → We execute {Fact 0 R} and {Fact N R} on the abstract machine

126

## 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
```

127

## 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  $\sigma$  and an environment  $E$ :

$\sigma = \{fact = (\text{proc } \{\$ N R\} \dots \text{end}, \{Fact \rightarrow fact\}), n=0, r\}$   
 $E = \{Fact \rightarrow fact, N \rightarrow n, R \rightarrow r\}$

- Here is what we will execute:

{Fact N R},  $E, \sigma$

128



## Execution of {Fact 0} (2)



- To execute {Fact N R} we replace it by the procedure body and we replace the calling environment by a new environment

- The instruction:

{Fact N R}, {Fact  $\rightarrow$  fact, N  $\rightarrow$  n, R  $\rightarrow$  r},  $\sigma$

is replaced by the instruction:

```
local B in
  B=(N==0)
  if B then R=1 else ... end
end, {Fact  $\rightarrow$  fact, N  $\rightarrow$  n, R  $\rightarrow$  r},  $\sigma$ 
```

Later on we will see how to replace the calling environment by a new environment inside the procedure body.

129

## 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  $\rightarrow$  fact, N  $\rightarrow$  n, R  $\rightarrow$  r},  $\sigma$ 
```

we do two operations:

- We extend the memory with a new variable  $b$
- We extend the environment with  $\{B \rightarrow b\}$

- We then replace the instruction by its body:

```
B=(N==0)
if B then R=1 else ... end,
{Fact  $\rightarrow$  fact, N  $\rightarrow$  n, R  $\rightarrow$  r, B  $\rightarrow$  b},  $\sigma \cup \{b\}$ 
```

130

## 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  $b = \text{true}$  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  $N * \{\text{Fact } N-1\}$

131

## 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  $n \times (\text{result of } \{\text{Fact } N-1\} \text{ execution})$
- 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

132

## Semantics 5: Semantic rules of procedures



133

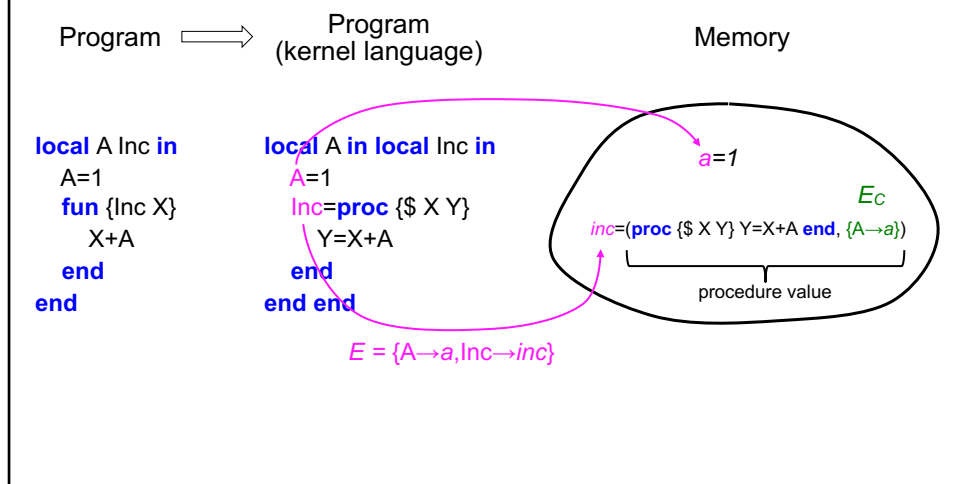
## Procedures are the building blocks of abstraction



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

134

## We recall how procedures are stored in memory



135

## Defining and calling procedures



- 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

136

## Procedure call 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 identifiers 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

137

## Procedure call example (2)



```
local P in
  local Z in
    Z=1
    proc {P X Y} Y=X+Z end % EC = {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: EP = {Y→b, X→a, Z→z}
  end
end
```

138

## Semantic rule for procedure definition



- Semantic instruction:  
 $(\langle x \rangle = \text{proc } \{ \$ \langle x \rangle_1 \dots \langle x \rangle_n \} \langle s \rangle \text{ end}, E)$ 
  - Formal arguments:  
 $\langle x \rangle_1, \dots, \langle x \rangle_n$
  - Free identifiers in  $\langle s \rangle$ :  
 $\langle z \rangle_1, \dots, \langle z \rangle_k$
  - Contextual environment:  
 $E_C = E|_{\langle z \rangle_1, \dots, \langle z \rangle_k}$  (restriction of  $E$  to free identifiers)
- Create the following binding in memory:  
 $x = (\text{proc } \{ \$ \langle x \rangle_1 \dots \langle x \rangle_n \} \langle s \rangle \text{ end}, E_C)$

139

## Semantic rule for procedure call (1)



- Semantic instruction:  
 $(\{ \langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n \}, E)$
- If the activation condition is false ( $E(\langle x \rangle)$  unbound)
  - Suspension (do not execute, wait until  $E(\langle x \rangle)$  is bound)
- 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

140

## Semantic rule for procedure call (2)



- Semantic instruction on stack:  
 $(\{\langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n\}, E)$   
 with procedure definition in memory:  
 $E(\langle x \rangle) = (\text{proc } \{\$ \langle z \rangle_1 \dots \langle z \rangle_n\} \langle s \rangle \text{ end}, E_C)$
- Put the following instruction on the stack:  
 $(\langle s \rangle, E_C + \{\langle z \rangle_1 \rightarrow E(\langle y \rangle_1), \dots, \langle z \rangle_n \rightarrow E(\langle y \rangle_n)\})$

141

## Computing with environments



- The abstract machine does two kinds of computations with environments
- **Adjunction:**  $E_2 = E_1 + \{X \rightarrow y\}$ 
  - Add a pair (identifier  $\rightarrow$  variable) to an environment
  - Overrides the same identifier in  $E_1$  (if it exists)
  - Needed for **local**  $\langle x \rangle$  **in**  $\langle s \rangle$  **end** (and others)
- **Restriction:**  $E_C = E|_{\{X, Y, Z\}}$ 
  - Limit identifiers in an environment to a given set
  - Needed to calculate the contextual environment

142



## Adjunction

- For a **local** instruction

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

- $E_1 = \{\text{Browse} \rightarrow b, X \rightarrow x\}$
- $E_2 = E_1 + \{X \rightarrow y\} = \{\text{Browse} \rightarrow b, X \rightarrow y\}$

143



## Restriction

- For a procedure declaration

```
local A B C AddB in  
  A=1 B=2 C=3 (E)  
  fun {AddB X} (EC: contextual environment)  
    X+B  
  end  
end
```

- $E = \{A \rightarrow a, B \rightarrow b, C \rightarrow c, \text{AddB} \rightarrow a'\}$
- $E_C = E_{\{B\}} = \{B \rightarrow b\}$

144



## Semantics summary

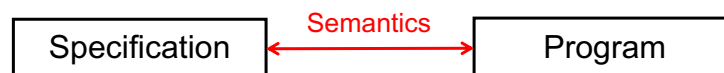


145

## Bringing it all together



- Defining the semantics brings many concepts together
  - Concepts we have seen before: identifier, variable, environment, memory, instruction, kernel language
  - New concepts: procedure value, semantic instruction, semantic stack, 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



146

## Discrete mathematics



- The abstract machine is built with discrete mathematics
- It is probably the most complex construction that you have seen built with discrete mathematics!
  - Engineering students are quite used to integrals, differential equations, and complex analysis, which are all continuous mathematics, and the abstract machine is a new construction!
- 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)

147

## Why semantics is important



- Semantics is an intrinsic 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 later on), 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
  - Users don't need to understand the semantics to take advantage of it: **its mere existence is enough**
    - Only the system's designers need to understand the semantics
- « Semantics is the ultimate programming language »
  - Invariants as the ultimate loop construct
  - Data abstractions as new kernel language instructions

148

## Using the 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 and termination)
  - 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
  - We will use the semantics when needed in the rest of the course

149

**“Semantics is the ultimate programming language”**



150