R: Control and data flow

Sławek Staworko

University of Lille

2020

Outline

Expressions

Function calls

Branching expression (if/else)

Loops

Scope and evaluation

Expressions

Expressions

Functional programming paradigm

- Program is an expression $\mathcal{E}$
- Running the program = evaluating the expression $\mathcal{E}$

Basic expression building blocks

- Assignment $\texttt{x} \leftarrow \mathcal{E}$ assigns a value to the variable and evaluates to the value of `x`
- Function application $\texttt{f(}\mathcal{E}_1\texttt{,...,}\mathcal{E}_n\texttt{)}$ calls the function and evaluates to the result returned by the function
- Composition $\mathcal{E}_0\texttt{;}\mathcal{E}_1$ evaluates to the value of the last expression $\mathcal{E}_1$
- Grouping $\{\ \mathcal{E}\ \}$ evaluates to the inner expression (for structuring purposes)

Expressions

Example

- `y←x←1` ↦

Expressions

Example

- `y←x←1` ↦ `1` `x = 1 and y = 1`
- `y←1+(x←1)` ↦

Expressions

Example

- `y←x←1` ↦ 1 x = 1 and y = 1
- `y←1+(x←1)` ↦ 2 x = 1 and y = 2
- `x←1;0` ↦

Expressions

Example

- `y←x←1` ↦ `1` `x = 1 and y = 1`
- `y←1+(x←1)` ↦ `2` `x = 1 and y = 2`
- `x←1;0` ↦ `0` `x = 1`
- `y←{x←1;0}` ↦

Expressions

Example

- `y←x←1` ↦ 1 `x = 1 and y = 1`
- `y←1+(x←1)` ↦ 2 `x = 1 and y = 2`
- `x←1;0` ↦ 0 `x = 1`
- `y←{x←1;0}` ↦ 0 `x = 1 and y = 0`
- `y←{x←{1;0}}` ↦

Expressions

Example

- y←x←1 ↦ 1 x = 1 and y = 1
- y←1+(x←1) ↦ 2 x = 1 and y = 2
- x←1;0 ↦ 0 x = 1
- y←{x←1;0} ↦ 0 x = 1 and y = 0
- y←{x←{1;0}} ↦ 0 x = 0 and y = 0
- 2 + 3 ↦ ‘+‘(2,3) ↦ 5
- substr(s←"abcde",i←nchar(s)-2,i+2) ↦

Expressions

Example

- y←x←1 ↦ 1 x = 1 and y = 1
- y←1+(x←1) ↦ 2 x = 1 and y = 2
- x←1;0 ↦ 0 x = 1
- y←{x←1;0} ↦ 0 x = 1 and y = 0
- y←{x←{1;0}} ↦ 0 x = 0 and y = 0
- 2 + 3 ↦ ‘+‘(2,3) ↦ 5
- substr(s←"abcde",i←nchar(s)-2,i+2) ↦ "cde"
  s = "abcde" and i = 3

Function calls

Positional and named arguments

- `f ← function (a,b) a + 3*b`
- `f(10,2)` ↦ `16`
- `f(2,10)` ↦ `32`
- `f(a=10,b=2)` ↦ `16`
- `f(b=2,a=10)` ↦ `16`

Default arguments

-
  ```
  f ← function (a,b=1,c=3) {
          if (b == 1) a + c else b + c
  }
  ```
- `f(10)` ↦ `13`
- `f(10,1)` ↦ `13`
- `f(10,2)` ↦ `5`
- `f(10,2,4)` ↦ `6`
- `f(10,c=4)` ↦ `14`
- `f(b=2,c=4)` ↦ `6`
- `f(b=1)` ↦ `error`

Variable length arguments . . .

Ellipsis . . .

Additional arguments of a function that can be passed on

Example

```
sapply ← function(v,f,...)  {
   res ← vector()
   for (i in 1:length(v)) {
      res[i] ← f(v[i],...)
   }
   res
}
```

```
sapply(1:4,function(x,y) x+y, 2) ↦ 3 4 5 6
sapply(1:4,‘+‘,2) ↦ 3 4 5 6
```

Branching expression (if/else)

Conditional expression (if/else)

$$\texttt{if (}\mathcal{E}\texttt{) }\mathcal{E}_1\texttt{ else }\mathcal{E}_2$$

- if $\mathcal{E}$ evaluates to `TRUE`, evaluate and return the value of $\mathcal{E}_1$; otherwise evaluate and return the value of $\mathcal{E}_2$
- $\mathcal{E}$ must be interpretable as logical
- the `else` part is optional; when missing and the condition $\mathcal{E}$ is not satisfied, the whole expression evaluates to `NULL`

Example

- `if (c(-1,1) > 0) "+" else "-"` ↦ `"-"`
- `if (any(c(-1,1) > 0)) "+" else "-"` ↦ `"+"`
- `if (0) "a" else "b"` ↦ `"b"`
- `if (-2) "a" else "b"` ↦ `"a"`
- `if ("FALSE") 1 else 0` ↦ `0`
- `if ("a") 1 else 0` ↦ `error`

Vectorised if/else

`ifelse(test, yes, no)`

- `test` an object which can be coerced to logical
- `yes` return values for true elements of `test`
- `no` return values for false elements of `test`

Example

- `ifelse(-2:2 < 0, "-", "+")` ↦ `"-" "-" "+" "+" "+"`
- `ifelse(1:4 %% 2 == 0, "E", "O")` ↦ `"O" "E" "O" "E"`

Vectorized if/else (contd.)

Common misconception

```
ifelse(test, yes, no)
```

is supposed to be equivalent to

```
{tmp←yes; tmp[!test]←no; tmp}
```

However, consider

- `x ← c(1,2,3,4,6,7,8,9)`
- `x %% 2 == 0 ↦ F T F T T F T F`
- `ifelse(x %% 2 == 0, c("E","e"), c("O","o","."))`
  `↦ "O" "e" "." "e" "E" "." "E" "o"`
- `{tmp←c("E","e"); tmp[x%%2!=0]←c("O","o","."); tmp}`
  `↦ "O" "e" "o" NA  NA  "." NA  "O"`

Switch statement

```
switch(x,"a","b","c")
if (x == 1) {
   "a"
} else if (x == 2) {
   "b"
} else if (x == 3) {
   "c"
} else {
   NULL
}
```

```
switch(s,a=1,b=2,c=3,4)
if (s == "a") {
   1
} else if (x == "b") {
   2
} else if (x == "c") {
   3
} else {
   4
}
```

Loops

Loops

Two main looping mechanisms

- Imperative: `for`, `while`, and `repeat`
- Declarative: `apply` function family

Imperative looping

Kinds of loops

- `for (`var `in` $\mathcal{C}$`)` $\mathcal{E}$ iterates over elements of a collection
- `while (`$cond$`)` $\mathcal{E}$ iterated as long as a given condition is satisfied
- `repeat` $\mathcal{E}$ iterates indefinitely (unless `break` is used)
- all loops evaluate to NULL

Flow control inside loops

- `next` interrupts the current iteration and control flow moves to the next to next one
- `break` interrupts the execution and exits the inner most loop

For loop

for (x in $\mathcal{C}$) $\mathcal{E}$

- $\mathcal{C}$ is a vector or list (hence also factor and data frame)
- $\mathcal{E}$ is evaluated for x being assigned consecutive values in $\mathcal{C}$
- side-effect: after having finished the execution the variable x is defined and carries the last assigned value

For loop

for (x in $\mathcal{C}$) $\mathcal{E}$

- $\mathcal{C}$ is a vector or list (hence also factor and data frame)
- $\mathcal{E}$ is evaluated for x being assigned consecutive values in $\mathcal{C}$
- side-effect: after having finished the execution the variable x is defined and carries the last assigned value

Example

```
sum ← function (v) {
   acc ← 0;
   for (x in v)
       acc ← acc + x;
   return(acc)
}

sum(c(1,4,2,6,1)) ↦ 14
```

For loop

`for` $(x$ `in` $\mathcal{C})$ $\mathcal{E}$

- $\mathcal{C}$ is a vector or list (hence also factor and data frame)
- $\mathcal{E}$ is evaluated for x being assigned consecutive values in $\mathcal{C}$
- side-effect: after having finished the execution the variable x is defined and carries the last assigned value

Example

```
sum ← function (v) {
   acc ← 0
   for (x in v)
       acc ← acc + x
   acc
}
```

sum(c(1,4,2,6,1)) ↦ 14

For loop (example)

Example

```
f ← function (v) {
    w ← numeric(length(v))
    for (i in 1:length(v)) {
        w[i] ← 2*v[i] + i
    }
    w
}
```

```
f(c(1,4,7)) ↦ 3 10 17 ≡ 2*c(1,4,7) + 1:3
```

For loop (examples)

Example

```
find_elem ← function (v,x) {
   for (i in 1:length(v)) {
      if (v[i] == x) {
         return(TRUE)
      }
   }
   FALSE
}
find_elem(c(1,4,7,10,3,2,1,4),2) ↦ TRUE
```

```
find_elem(v,x) ≡ any(v == x)
```

For loop (examples)

Example

```
find_pos ← function (v,x) {
   for (i in 1:length(v)) {
      if (v[i] == x) {
         return(i)
      }
   } }
```

```
find_pos(c(1,4,7,10,3,2,1,4),2) ↦ 6
```

```
find_pos(v,x) ≡ (1:length(v))[v == x][1]
```

For loop (example)

Example

```
sapply ← function(v,f) {
   res ← vector()
   for (i in 1:length(v)) {
      res[i] ← f(v[i])
   }
   res
}
```

```
sapply(1:4,function (x) x^2) ↦ 1 4 9 16
sapply(1:4,as.character) ↦ "1" "2" "3" "4"
sapply(1:4,function (x) x/2.0) ↦ 0.5 1.0 1.5 2.0
```

While loop

`while (cond)` $\mathcal{E}$

- execute $\mathcal{E}$ again and again as long as `cond` evaluates to `TRUE`

While loop (example)

Example

```
create_polynomial ← function (p) function (x) {
   y ← 0
   i ← length(p)
   while (i > 0) {
      y ← y + p[i] * (x^(length(p)-i))
      i ← i - 1
   }
   y
}
```

```
p ← create_polynomial(c(5,4,2,3))
```

$p(x) = 5x^3 + 4x^2 + 2x + 3$

```
p(1) ↦ 14
p(-1) ↦ 0
p(2) ↦ 63
```

Repeat loop

`repeat` $\mathcal{E}$

- execute $\mathcal{E}$ again and again until `break` is called

Repeat loop (example)

Example

```
find_root ← function(p, x1, x2) {
  repeat {
    y1 ← p(x1)
    y2 ← p(x2)
    xz ← (x1+x2)/2
    yz ← p(xz)
    if (sign(y1) == sign(yz))
      x1 ← xz
    if (sign(y2) == sign(yz))
      x2 ← xz
    if (abs(yz) < 10e-10)
      break
  }
  xz
}
find_root(create_polynomial(c(5,4,2,3),-10,10)) ↦ -1
```

Declarative iteration

Apply family of functions

- `sapply` operates on vectors
- `lapply` operates on list
- `apply` operates on matrices
- `mapply` operates on multiple vectors

Variable length arguments . . .

Ellipsis

Additional arguments that are passed through to other functions.

Example

```
sapply ← function(v,f,...)  {
   res ← vector()
   for (i in 1:length(v)) {
      res[i] ← f(v[i],...)
   }
   res
}

sapply(1:4,function(x,y) x+y, 2) ↦ 3 4 5 6
sapply(1:4,‘+‘,2) ↦ 3 4 5 6
```

Use case: function vectorisation

Most functions in R are vectorised

- when given a vector, the function is applied on every element
- sqrt : $\texttt{num}^* \rightarrow \texttt{num}^*$
- `sqrt(c(1,2,3))` ↦ `1.000000 1.414214 1.732051`

Non-vectorised function can be vectorised with `sapply`

-
```
loop ← function (n) {
   x ← 1
    for (i in 1:n)
       x ← sin(x)
    x
}
```
- `loop(1)` ↦ `0.8414710`
- `loop(2)` ↦ `0.7456241`
- `loop(c(1,2))` ↦ `error`
- `loop_vect ← function(v) sapply(v,loop)`
- `loop_vect(c(1,2))` ↦ `0.8414710 0.7456241`

Creating matrices with `sapply`

`sapply(v,f)` will return a matrix

- when `f` returns a vector of the same length each time

Example

- `f ← function (n) cos(seq(n, (n+1), 0.5))`
- `f(1)` ↦ `0.54030 0.07074 -0.41615`
- `f(2)` ↦ `-0.4161 -0.8011 -0.9900`
- `sapply(1:4,f)`

  ↧

$$\begin{bmatrix} 0.54030 & -0.4161 & -0.9900 & -0.6536 \\ 0.07074 & -0.8011 & -0.9365 & -0.2108 \\ -0.41615 & -0.9900 & -0.6536 & 0.2837 \end{bmatrix}$$

Matrix iteration with `apply`

`apply(M,dim,f,...)`

- if `dim = 1`, then `f` is called on every row
- if `dim = 2`, then `f` is called on every column
- if `dim = c(1,2)`, then `f` is called on every cell

Example

- `m` $\leftarrow \begin{bmatrix} 1 & 4 & 9 \\ 4 & 16 & 25 \end{bmatrix}$
- `apply(m,1,sum)` $\mapsto$ `14 45`
- `apply(m,2,sum)` $\mapsto$ `5 20 34`
- `apply(m,c(1,2),sqrt)` $\mapsto \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \end{bmatrix}$

Point-wise vectorizations with mapply

mapply(f,v_1,v_2,...,v_k,...)

- constructs a vector v such that
- v[i] ← f(v_1[i],v_2[i],...,v_k[i],...)
- v is a long as the longest of the input vectors (recycling is applied if lengths are not all equal)

Example

- mapply(function (x,y) x+y, c(1,3), c(4,6)) ↦ 5 9
  ≡ mapply('+', c(1,3), c(4,6)) ≡ c(1,3) + c(4,6)
-
```
f ← function (x,y) {
    for (i in 1:y)
        x ← sqrt(x+1)
    x
}
```
- mapply(f,1:3,c(1,5,10)) ↦ 1.4142 1.6191 1.6180

Scope and evaluation

Variable scope

Scoping

- what a ”*part of a program*” means?
- variable: where is its value stored

Two types of scoping

lexical depends on the location in the source code, where the variable is defined

dynamic depends on the execution context

Environment

- *environment* is a frame of reference for variable lookup
- organized into a *stack* (but actually a *tree*)
- if a variable doesn't exists in the current environment, then check its parent, then its grandparent, etc.
- *global* environment, storing all global variables, is at the end of the search path (*root* environment)
- when function is called a *new environment* is created, function parameters are new variables in the new environment
- the parent environment of a function call is always the environment where the function has been *defined* (and not where the function has been called)

What are the environments?

- `x ← 1`

```
x = 1
```

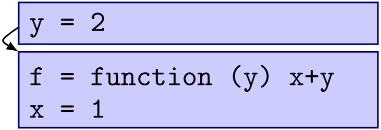
What are the environments?

- `x ← 1`
- `f ← function (y) x + y`

```
f = function (y) x+y
x = 1
```

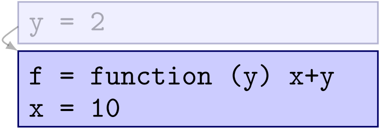
What are the environments?

- x ← 1
- f ← function (y) x + y
- f(2) ↦ 3



What are the environments?

- x ← 1
- f ← function (y) x + y
- f(2) ↦ 3
- x ← 10



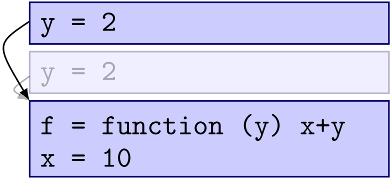
y = 2

f = function (y) x+y
x = 10

What are the environments?

- x ← 1
- f ← function (y) x + y
- f(2) ↦ 3

- x ← 10
- f(2) ↦ 12



y = 2

y = 2

f = function (y) x+y
x = 10

What are the environments?

- x ← 1

```
x = 1
```

What are the environments?

- `x ← 1`
- `f ← function (y) x + y`

```
f = function (y) x+y
x = 1
```

What are the environments?

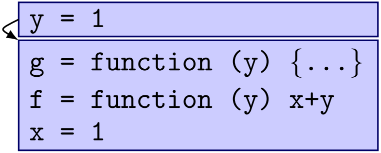
- `x ← 1`
- `f ← function (y) x + y`

-
  ```
  g ← function (y) {
      x ← 10
      x + y
  }
  ```

```
g = function (y) {...}
f = function (y) x+y
x = 1
```

What are the environments?

- `x ← 1`
- `f ← function (y) x + y`
-
  ```
  g ← function (y) {
      x ← 10
      x + y
  }
  ```
- `f(1) ↦ 2`



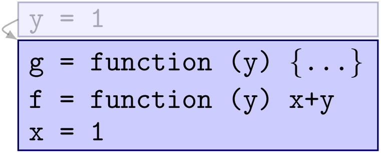
```
y = 1
```

```
g = function (y) {...}
f = function (y) x+y
x = 1
```

What are the environments?

- x ← 1
- f ← function (y) x + y
- g ← function (y) {

  ```
      x ← 10
      x + y
  }
  ```
- f(1) ↦ 2



y = 1

g = function (y) {...}

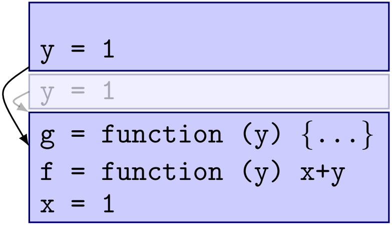
f = function (y) x+y

x = 1

What are the environments?

- `x ← 1`
- `f ← function (y) x + y`

-
  ```
  g ← function (y) {
      x ← 10
      x + y
  }
  ```
- `f(1) ↦ 2`
- `g(1) ↦ 11`



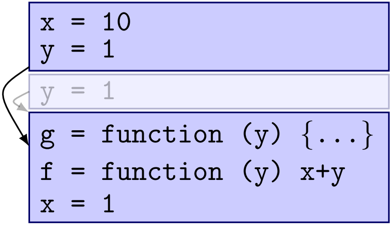
```
y = 1
```

```
y = 1
```

```
g = function (y) {...}
f = function (y) x+y
x = 1
```

What are the environments?

- x ← 1
- f ← function (y) x + y
- g ← function (y) {
  ```
      x ← 10
      x + y
  }
  ```
- f(1) ↦ 2
- g(1) ↦ 11



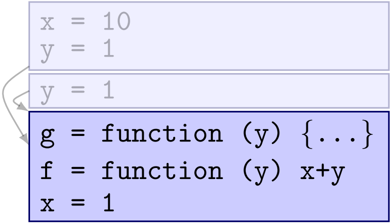
x = 10
y = 1

y = 1

g = function (y) {...}
f = function (y) x+y
x = 1

What are the environments?

- x ← 1
- f ← function (y) x + y
- g ← function (y) {
  ```
      x ← 10
      x + y
  }
  ```
- f(1) ↦ 2
- g(1) ↦ 11



x = 10
y = 1

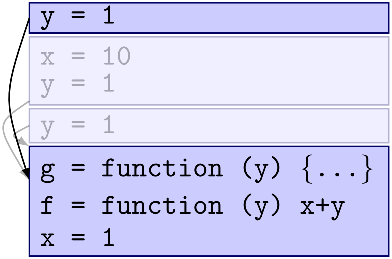
y = 1

g = function (y) {...}
f = function (y) x+y
x = 1

What are the environments?

- `x ← 1`
- `f ← function (y) x + y`

-
  ```
  g ← function (y) {
      x ← 10
      x + y
  }
  ```
- `f(1) ↦ 2`
- `g(1) ↦ 11`
- `f(1) ↦ 2`



```
y = 1
```

```
x = 10
y = 1
```

```
y = 1
```

```
g = function (y) {...}
f = function (y) x+y
x = 1
```

What are the environments?

- `x ← 1`

```
x = 1
```

What are the environments?

- `x ← 1`
- `f ← function (y) x + y`

```
f = function (y) x+y
x = 1
```

What are the environments?

- `x ← 1`
- `f ← function (y) x + y`

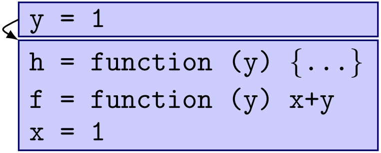
-
  ```
  h ← function (y) {
        x↞ 10
        x + y
  }
  ```

```
h = function (y) {...}
f = function (y) x+y
x = 1
```

What are the environments?

- x ← 1
- f ← function (y) x + y
- h ← function (y) {

  ```
      x<<- 10
      x + y
  }
  ```
- f(1) ↦ 2



y = 1

h = function (y) {...}

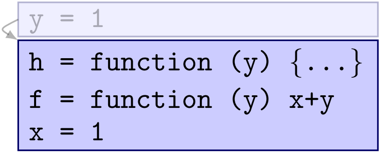
f = function (y) x+y

x = 1

What are the environments?

- x ← 1
- f ← function (y) x + y
- h ← function (y) {

  ```
      x↞ 10
      x + y
  }
  ```
- f(1) ↦ 2



y = 1

h = function (y) {...}

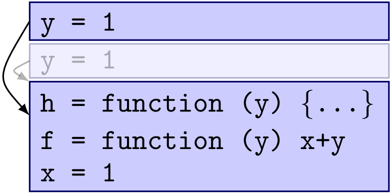
f = function (y) x+y

x = 1

What are the environments?

- x ← 1
- f ← function (y) x + y
- h ← function (y) {

      x↞ 10
      x + y
  }
- f(1) ↦ 2
- h(1) ↦ 11



y = 1

y = 1

h = function (y) {...}

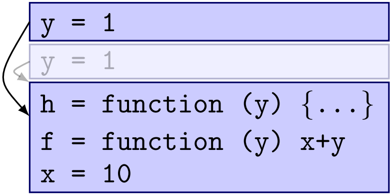
f = function (y) x+y

x = 1

What are the environments?

- x ← 1
- f ← function (y) x + y
- h ← function (y) {

  ```
      x↞ 10
      x + y
  }
  ```
- f(1) ↦ 2
- h(1) ↦ 11



y = 1

y = 1

h = function (y) {...}

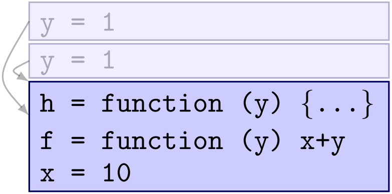
f = function (y) x+y

x = 10

What are the environments?

- `x ← 1`
- `f ← function (y) x + y`

-
  ```
  h ← function (y) {
        x↞ 10
        x + y
  }
  ```
- `f(1) ↦ 2`
- `h(1) ↦ 11`

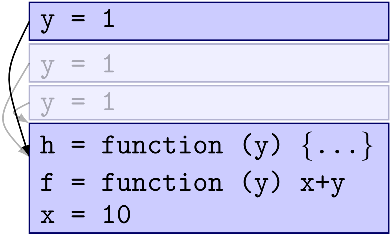


```
y = 1
y = 1
h = function (y) {...}
f = function (y) x+y
x = 10
```

What are the environments?

- x ← 1
- f ← function (y) x + y
- h ← function (y) {

  ```
      x<<- 10
      x + y
  }
  ```
- f(1) ↦ 2
- h(1) ↦ 11
- f(1) ↦ 11



y = 1

y = 1

y = 1

h = function (y) {...}
f = function (y) x+y
x = 10

What are the environments?

- `x` ← `1`

What are the environments?

- `x ← 1`

-
  ```
  f ← function (y,recurse) {
      if (recurse) {
          x ← 10
          f(y,FALSE)
      } else {
          x+y
      }
  }
  ```

What are the environments?

- `x ← 1`

-
  ```
  f ← function (y,recurse) {
           if (recurse) {
                x ← 10
                f(y,FALSE)
           } else {
                x+y
           }
  }
  ```
- `f(2,TRUE) ↦ 3`

What are the environments?

- `x` ← `1`

What are the environments?

- `x ← 1`

-
  ```
  F ← function (x) {
      f ← function (y) {
          x+y
      }
      f
  }
  ```

What are the environments?

- `x ← 1`

-
  ```
  F ← function (x) {
        f ← function (y) {
              x+y
        }
        f
  }
  ```
- `f ← F(2)`

What are the environments?

- `x ← 1`

-
  ```
  F ← function (x) {
      f ← function (y) {
          x+y
      }
      f
  }
  ```
- `f ← F(2)`
- `f(1) ↦ 3`

What are the environments?

- `x ← 1`
-
  ```
  F ← function (x) {
      f ← function (y) {
          x+y
      }
      f
  }
  ```
- `f ← F(2)`
- `f(1) ↦ 3`
- `x ← 10`

What are the environments?

- `x ← 1`

-
  ```
  F ← function (x) {
      f ← function (y) {
          x+y
      }
      f
  }
  ```
- `f ← F(2)`
- `f(1) ↦ 3`
- `x ← 10`
- `f(1) ↦ 3`

What are the environments?

- `x` ← `1`

What are the environments?

-
  ```
  x ← 1
  ```

-
  ```
  F ← function() {
      x ← 10
      f_ ← function(y) {
          x + y
      }
      f_
  }
  ```

What are the environments?

- `x ← 1`
-
  ```
  F ← function() {
        x ← 10
        f_ ← function(y) {
              x + y
        }
        f_
  }
  ```
- `f ← F()`

What are the environments?

- `x ← 1`

-
  ```
  F ← function() {
       x ← 10
       f_ ← function(y) {
            x + y
       }
       f_
  }
  ```
- `f ← F()`
- `f(1) ↦ 11`

What are the environments?

- `x ← 1`

-
  ```
  F ← function() {
       x ← 10
       f_ ← function(y) {
            x + y
       }
       f_
  }
  ```
- `f ← F()`
- `f(1) ↦ 11`
- `x ← 100`

What are the environments?

- `x ← 1`

-
  ```
  F ← function() {
        x ← 10
        f_ ← function(y) {
              x + y
        }
        f_
  }
  ```
- `f ← F()`
- `f(1) ↦ 11`
- `x ← 100`
- `f(1) ↦ 11`

What are the environments?

- `x` ← `1`

What are the environments?

- `x ← 1`

-
  ```
  F ← function() {
      x ← 10
          g ← function(y) {
          x + y
      }
      f ← function(z) {
          x ↞ z
      }
      list(getter=g,setter=f)
  }
  ```

What are the environments?

- `x ← 1`
-
  ```
  F ← function() {
    x ← 10
      g ← function(y) {
      x + y
    }
    f ← function(z) {
      x ↞ z
    }
    list(getter=g,setter=f)
  }
  ```
- `o = F()`

What are the environments?

- `x ← 1`

-
  ```
  F ← function() {
      x ← 10
          g ← function(y) {
          x + y
      }
      f ← function(z) {
          x ↞ z
      }
      list(getter=g,setter=f)
  }
  ```
- `o = F()`
- `o$getter(1) ↦ 11`

What are the environments?

- `x ← 1`

-
  ```
  F ← function() {
      x ← 10
          g ← function(y) {
          x + y
      }
      f ← function(z) {
          x ↞ z
      }
      list(getter=g,setter=f)
  }
  ```
- `o = F()`
- `o$getter(1) ↦ 11`
- `o$setter(100)`

What are the environments?

- `x ← 1`
-
  ```
  F ← function() {
      x ← 10
          g ← function(y) {
          x + y
      }
      f ← function(z) {
          x ↞ z
      }
      list(getter=g,setter=f)
  }
  ```
- `o = F()`
- `o$getter(1) ↦ 11`
- `o$setter(100)`
- `o$getter(1) ↦ 101`

Pass by value/reference (Python's case)

Pass by reference (lists)

Reference to the object is passed
Original object can be modified

-
  ```
  def f(l):
        l[2] = 0
  ```
- `l = [1,2,3]`
- `f(l)`
- `l` ↦ `[1,2,0]`

Pass by value (atoms)

Value of the object is copied
Original object cannot be modified

-
  ```
  def f(i):
        i = i+1
  ```
- `i = 2`
- `f(i)`
- `i` ↦ `2`

Variable passing in R

Semantically, pass by value for all basic data structures

- `f ← function(l) l[2] ← 0`
- `l = list(1,2,3)`
- `f(l)`
- `l ↦ 1 2 3`

Lazy evaluation

Pass by promise

Arguments are evaluated only and when needed

- `f ← function (x) {print(as.character(x)); x+10}`
- `g ← function (y,z) {print(as.character(y)); z}`
- `g(100,f(1))`

```
  ↧
"100"
"1"
11
```