Lecture 4: Advanced uses of functions

March 2021
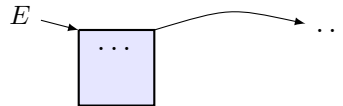
Remember that ...

In functional programming, functions are values:
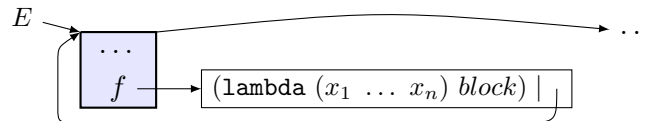
- The evaluation of a definition

  (**define** (f x_1 ... x_n) *block*)

  in an environment

  extends the top frame of E with the binding $f \mapsto \langle code, E \rangle$, where *code* is (**lambda** (x_1 ... x_n) *block*). The new environment E is


  The internal representation of the function is a pair $\langle code, E \rangle$, called **closure**. The pair stores the textual definition of the function together with a reference to the environment E where the function was created.

- The evaluation of a function call (f $expr_1$ ... $expr_n$) in an environment E proceeds as follows:

  1. First, the arguments $expr_1, \ldots, expr_n$ are reduced to values v_1, ..., v_n in E.

2. Next, we look up the value v of f in E. If

$$E(f) = \langle (\texttt{lambda } (x_1 \ \ldots \ x_n) \ block), E' \rangle.$$

we compute the value of *block* (the body of function f) in the environment
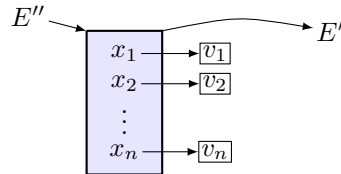
The top frame of this environment binds the formal parameters $x_1, \ldots, x_n$ of f to the input values $v_1, \ldots, v_n$.

3. After v is computed:

   - the top frame of E'' is garbage collected,
   - the evaluation environment is restored to be E, and
   - v is returned as value of the function call.

See Lecture 3 about how tail call optimization works.

1   Local definitions

An important programming principle is to avoid cluttering the global environment with useless or auxiliary definitions.

- In functional programming, we can make definitions local to the place where they are needed.

- We will illustrate how to apply this principle by improving Newton's method to compute numeric approximations of $\sqrt{a}$ for a number $a \in \mathbb{R}$, $a > 0$.

Newton noticed that, if a is a positive real number then $\sqrt{a}$ is the limit of the sequence $(x_n)_{n \in \mathbb{N}}$ where

$$x_0 = 1.0 \text{ and } x_{n+1} = (x_n + a/x_n)/2 \text{ for all } n \in \mathbb{N}.$$

1.1 Newton's method: version 1

Let's assume that $x \in \mathbb{R}$ is a good approximation of $\sqrt{a}$ if $|x^2 - a| < 0.000001$. For this purpose we define the auxiliary function

```
(define (good? x a) (< (abs (- (* x x) a)) 0.000001))
```

Also, if the approximation x is not good enough for $\sqrt{a}$, we define the following function to compute an improved approximation:

```
(define (improve x a) (/ (+ x (/ a x)) 2))
```

The following is a recursive implementation of Newton's method, starting from the initial approximation x = 1.0:

```
(define (newtonV1 a [x 1.0])
  (if (good? x a)
      x
      (newtonV1 a (improve x a)))))
```

If we define functions good?, improve and newtonV1 in a global environment E, the top frame of E is extended with 3 bindings:
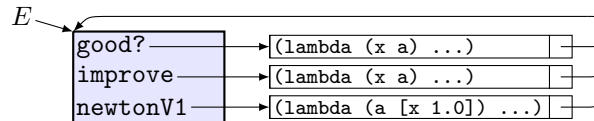
The disadvantage of this implementation is that the auxiliary functions good? and improve are globally visible. We wish to make them local to the body of Newton's method.

1.2 Newton's method: version 2

We can move the definitions of good? and improve in the body of function newton2:

```
(define (newtonV2 a [x 1.0])
  (define (good? x a) (< (abs (- (* x x) a)) 0.000001))
  (define (improve x a) (/ (+ x (/ a x)) 2))
  (if (good? x a) x (newtonV2 a (improve x a)))))
```

This definition can be improved if we notice that (newtonV2 a x) calls itself recursively as (newtonV2 a y) where y is some new input argument. Thus, the first argument of newtonV2 is invariant and need not be passed as argument to the local functions good? and improve:

3

```
(define (newtonV2 a [x 1.0])
  (define (good? x) (< (abs (- (* x x) a)) 0.000001))
  (define (improve x) (/ (+ x (/ a x)) 2))
  (if (good? x) x (newtonV2 a (improve x))))
```

This version adds a single binding to the top frame of E:
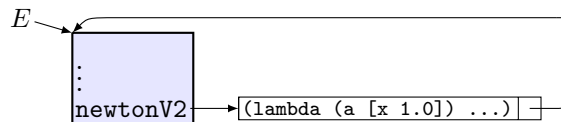
but has another disadvantage, which becomes visible when we call function
newtonV2. For example, the evaluation of (newtonV2 6) is reduced to the
evaluation of the body of newtonV2 in the extension of E with the temporary
frame depicted below:
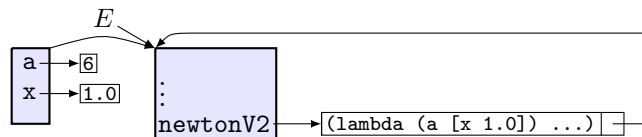
The body of newtonV2 has two local definitions, and its evaluation is reduced
to that of the recursive call (newtonV2 6 3.5) in the extended environment
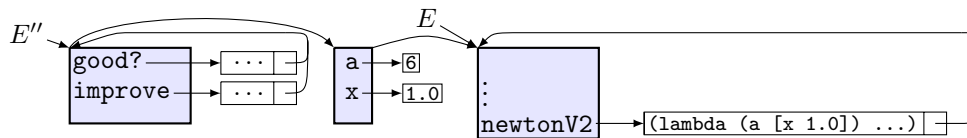
Because of tail recursion, the first two frames of E'' are garbage collected
and recreated between recursive calls of newtonV2. For example, the next
recursive call will be (newtonV2 6 2.60714) in the extended environment
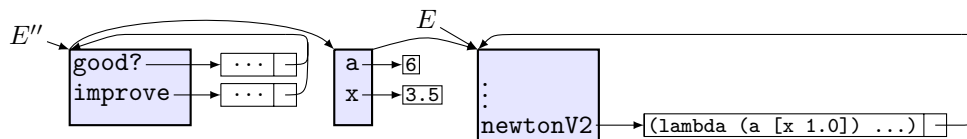
Note that the only binding that changes from a recursive call to another
is the binding for x. It would be more efficient to keep the bindings for a,
good? and improve, and not recreate them by every recursive call.

4

1.3 Newton's method: version 3

Version 2 is inefficient because recursive calls recreate invariant bindings for local variables. We can eliminate this problem as follows:

- We define a "wrapper", which is a function whose body contains all definitions which are invariant from one function call to another.

- In the "wrapper", we define an auxiliary function which performs the recursive computation, using the definitions in the "wrapper".

For Newton's method, we have

```
(define (newtonV3 a)        ; the wrapper
  (define (good? x) (< (abs (- (* x x) a)) 0.001))
  (define (improve x) (/ (+ x (/ a x)) 2))
  (define (sqrtIter x)        ; the auxiliary recursive function
    (if (good? x) x (sqrtIter (improve x))))
  (sqrtIter 1.0))
```

In this example, the variables with invariant definitions are good?, improve and a: they do not change from one recursive call to another.

Like version 2, the evaluation of this definition in an environment E adds a single binding to the top frame of E:
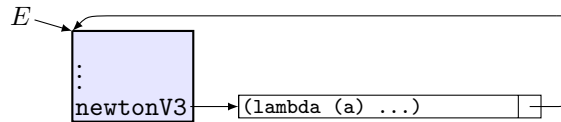
but the evaluation of function calls of newtonV3 is much more efficient. For example, the evaluation of (newtonV3 6) in E is reduced to the evaluation of (sqrtIter 1.0) in the environment
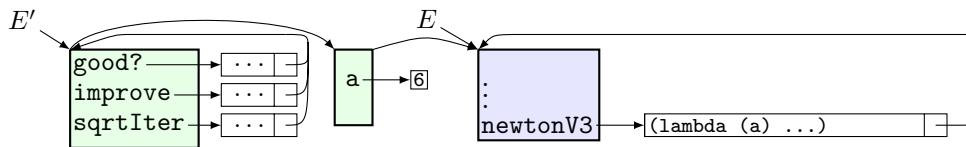
The definitions of good?, improve, sqrtIter and a are invariant and will persist in the first two frames of E' during the tail-optimized evaluation of (sqrt-iter 1.0). The evaluation of (sqrt-iter 1.0) returns the good approximation 2.4494943 of $\sqrt{6}$ after five recursive calls:

(sqrtIter 1.0) in $\qquad\qquad$ E'

(sqrtIter 3.5) in $\qquad\qquad\qquad$ E'

$\boxed{\text{x}} \longrightarrow \boxed{\text{1.0}}$

(sqrtIter 2.6071428) in $\qquad\qquad$ E'

$\boxed{\text{x}} \longrightarrow \boxed{\text{3.5}}$

(sqrtIter 2.4542563) in $\qquad\qquad$ E'

$\boxed{\text{x}} \longrightarrow \boxed{\text{2.6071428}}$

(sqrtIter 2.4494943) in $\qquad\qquad$ E'

$\boxed{\text{x}} \longrightarrow \boxed{\text{2.4542563}}$

2.4494943 in E
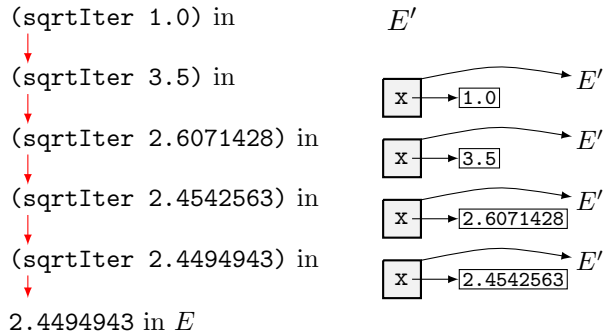
2 Functions with local state

Consider the function definition

```
(define (f x₁ ... xₙ)
   (lambda (y₁ ... yₘ) body))
```

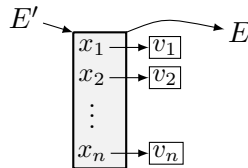The evaluation of this definition in an environment E adds the binding

$$f \mapsto \langle code, E \rangle$$

to the top frame of E. Note that f is a higher-order function because it returns a function as result. For example, the evaluation of

```
(define g (f expr₁ ... exprₙ))
```

in E has the following effect:

1. the values $v_1, \ldots, v_n$ of $expr_1, \ldots, expr_n$ are computed in E.

2. The body of f, which is (lambda (y_1 ... y_m) $body$), is evaluated in environment extended with a top frame that contains the bindings $x_i \mapsto v_i$ for $1 \leq i \leq n$. This is the environment

$E' \searrow \qquad\qquad E$

$\boxed{\begin{array}{l} x_1 \rightarrow \boxed{v_1} \\ x_2 \rightarrow \boxed{v_2} \\ \vdots \\ x_n \rightarrow \boxed{v_n} \end{array}}$

3. The binding $g \mapsto \langle code, E' \rangle$ is added to the top frame of E. Diagramatically, the environment E becomes
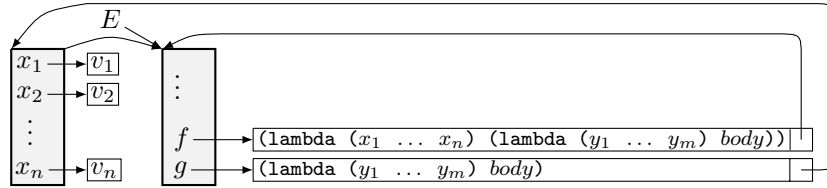
The top frame of E', which contains the bindings $x_i \mapsto v_i$ for $1 \le i \le n$ is not garbage collected because there is a reference to it, from the closure which is the value of g. The bindings $x_i \mapsto v_i$ from this frame are visible only from the body of g. They represent the **local state** of function g because g is the only function which can access them.

Functions as objects

Functions with local state can be used to model objects, like in OOP. Consider the function defined by

```
(define (make-student name id year study-field)
    (lambda (request  d)
      (cond [(eq? request 'name) name]
            [(eq? request 'id) id]
            [(eq? request 'year) year]
            [(eq? request 'study-field) study-field])))
```

This function is a constructor of objects which are function closures. For example, the function calls

```
> (define roy (make-student "Roy Wilson" 122016 1 'math))
> (define bill (make-student "Bill Cosby" 122021 2 'arts))
```

binds `roy` and `bill` to functions with local state for the variables `id`, `year` and `study-field`.

- `make-student` is a constructor of functions whose local state encapsulates information from the input arguments of the constructor call

- The closure produced by the constructor is a dispatch function f: if we call it with (f $'field$) we retrieve the value of the local variable $field$ stored in f. For example:

7

```
> (roy 'name)              > (bill name)
"Roy Wilson"               "Bill Cosby"
> (roy 'study-field)       > (bill 'study-field)
'math                      'arts
```

OOP with Racket

Racket is not a pure functional programming language: FP does not allow to
change the values of variables or the content of composite values, but Racket
has instructions that can change the value of a variable and the content of
a some composite values. These operations are destructive and specific to
imperative programming styles.

Note that:

1. Racket is intended to be used mostly for functional programming. The
   names of destructive operations end with ! to warn programmers that
   they have side effects.

2. Sometimes, these operations are useful.

We mention here only the assignment operation

(set! *name expr*)

which does the following when evaluated in an environment E: :

1. It computes the value v of *expr* in E

2. It looks up for a binding of *name* in E. If no binding is found, *name* is
   undefined and the assignment fails (it will display a warning message).
   Otherwise, the binding of *name* in E is replaced with *name* $\mapsto v$.

For example:

```
> (define x 1)      > (set! y 3)
> x                 set!: assignment disallowed;
1                   cannot set variable before its definition
> (set! x "abc")
> x
"abc"
```
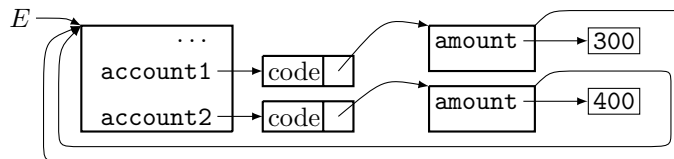
With set! we can define function objects that can change local state, e.g.,
account information:
```

8

```
(define (make-account amount)
  (lambda (request [value 0])
    (cond [(eq? request 'show) (display amount)]
          [(eq? request 'deposit) (set! amount (+ amount value))]
          [(eq? request 'withdraw)
           (if (> amount value)
               (set! amount (- amount value))
               (displayln "Insufficient money in account"))])))
```

The interpretation in an environment E of the definitions

```
> (define account1 (make-account 300))
> (define account2 (make-account 400))
```

creates two function values, for variables `account1` and `account2`, and each of them has its own local variable `amount`, as illustrated below.


```
> (account1 'deposit 50)      ; increase by 50 the amount of account1
> (account2 'withdraw 80)     ; decrease by 80 the amount of account2
> (account1 'show)
350
> (account2 'show)
320
> (account1 'withdraw 360)
Insufficient money in account
```

Note that

1. `makeAccount` is a constructor of function objects with local state for bank accounts.

2. The account objects created by `makeAccount` are dispatch functions which can be used to view or modify the state of an account.

3 Variadic functions

A variadic function is a function that can take an unlimited number of arguments. Typical examples are the predefined functions +, * and `append`.

A variadic function f can be defined as follows:

```
(define (f  x_1  ...  x_k  .  xs)  body)
```

9

Function f must be called with at least k input arguments. The values of the first k inputs are bound to variables $x_1, \ldots, x_k$ and the list of values of the other arguments is bound to parameter xs.

Examples:

- A function that computes the arithmetic mean $\dfrac{a_1 + a_2 + \ldots + a_n}{n}$ of the values $a_1, a_2, \ldots, a_n$ of its arguments:

```
(define (a-mean . lst)
  (if (null? lst)
      'no-average
      (/ (apply + lst) (length lst))))
```

```
> (a-mean)          > (a-mean 1 4 7)
'no-average         4
```

- A function that computes the harmonic mean $\dfrac{n}{\frac{1}{a_1} + \frac{1}{a_2} + \ldots + \frac{1}{a_n}}$ of the values $a_1, a_2, \ldots, a_n$ of its arguments:

```
(define (h-mean . lst)
  (if (null? lst)
      'no-harmonic-mean
      (/ (length lst)
         (apply +
                (map (lambda (x) (/ 1 x)) lst)))))
```

4   Problem-solving strategies

There are two well known problem-solving strategies: top-down and bottom-up.

Top-down

The top-down approach, also known as decomposition, solves a problem by starting from the most abstract specification that can describe succinctly the solution, and decomposing it incrementally into simpler sub-problems until we reach situations that can be implemented directly in RACKET. This approach is suitable for procedural programming and functional programming, because most procedures and functions have a compositional structure that can be identified by top-down refinement.

We used the top-down approach to define

- recursive functions on recursively defined datatypes.

- Newton's method to compute $\sqrt{a}$ for $a \in \mathbb{R}$, $a > 0$.

Bottom-up

The bottom-up approach resembles building with LEGO: It starts from elementary functions and operations whose behavior is specified in great detail (the basic building blocks), and repeatedly links them together into more complex functions, until we reach a solution of the given problem. This approach is suitable in programming environments with generic programming constructs that can be used as building blocks for a large variety of applications. For example, the list datatype and the higher-order functions `map`, `filter`, `apply`, `foldl`, `foldr` constitute very powerful collection of functionality that can be used for the bottom-up implementation of a large variety of applications. These higher-order functions behave as follows:

- (map f lst)

  takes as input a function f and list of values $v_1, \ldots, v_n$ and computes the list of values (f v_1), (f v_2), ..., (f v_n). More generally,

  (map f lst_1 ... lst_n)

  takes as input an n-ary function f and n lists of values, all of the same length, and returns the list of values

  $$(f \ v_{1,1} \ \ldots \ v_{1,n}), \ldots, (f \ v_{k,1} \ \ldots \ v_{k,n})$$

  if every list lst_i consists of values $v_{1,i}, \ldots, v_{k,i}$.

  Examples:

  ```
  > (map cons '(a b c) '(1 2 3))
  '((a . 1) (b . 2) (c . 3))
  > (map (lambda (x) (/ 1 x)) '(2 3 4))
  ```
  '($\frac{1}{2}$ $\frac{1}{3}$ $\frac{1}{4}$)

- (filter p lst)

  takes as inputs a predicate (=boolean function with 1 argument) p and a list lst and returns the list of values in lst for which p holds.

  Examples:

  ```
  > (filter symbol? '(a 1 () "abc" abc #t))
  '(a abc)
  ```

- (apply f lst)

  is reduced to the function call (f v_1 ... v_n) where $v_1, \ldots, v_n$ are the values of elements from lst.

- (foldl f v_0 lst)

  takes as inputs a function $f : A \times B \to B$, a value $v_0 \in B$ and a list `lst` of elements from A, and computes the value of $(f\ v_n\ \ldots (f\ v_1\ v_0)\cdots)$ where $v_1, \ldots, v_n$ are the values of elements from lst.

- (foldr f v_0 lst)

  takes as inputs a function $f : A \times B \to B$, a value $v_0 \in B$ and a list `lst` of elements from A, and computes the value of $(f\ v_1\ (f\ \ldots\ (f\ v_n\ v_0)\ \ldots))$ where $v_1, \ldots, v_n$ are the values of elements from lst.

Remarks: efficient implementations of these operations

- `foldl` has an efficient, tail-recursive definition:

  ```
  (define (foldl f v0 lst)
    (if (null? lst)
        v0
        (foldl f (f (car lst) v0) (cdr lst))))
  ```

  If the runtime complexity of `f` is $O(1)$, then the runtime complexity of

  ```
  (foldl f v0 lst)
  ```

  is $O(n)$ where n is the length of `lst`.

- (`reverse` has an efficient implementation with `foldl`:

  ```
  (define (reverse lst) (foldl cons null lst))
  ```

  Quiz: what is the runtime complexity of this implementation of `reverse`?

- A straightforward implementation of `foldr` is

  ```
  (define (foldr f v0 lst)
    (if (null? lst)
        v0
        (f (car lst) (foldr f v0 (cdr lst)))))
  ```

  but this is not tail-recursive. But `foldr` can be implemented efficiently with `reverse` and `foldl`.

Exercises:

1. Define `foldr` with `foldl` and `reverse`, and indicate the runtime complexity of this definition.

2. Define `filter` with `foldr`.

3. Define `length` with `foldl`.

4. Use `foldr` to define the variadic function

```
(compose f₁ ... fₙ)
```

which takes as inputs $n \geq 0$ unary functions $f_1, \ldots, f_n$ and returns the function that maps x to the value of

```
(f₁ ... (fₙ x)...)
```