What's new in Scheme: Lambda

Taken from Church's λ-calculus

```
(lambda (x) (+ x x))
```

"The function that maps x to x plus x"

At top level, just like `define`

In general, $\lambda x.E$, also written `(lambda (x) E)`

x is ***bound*** in E

other variables are ***free*** in E

Free variables make things interesting

```
(lambda (x) (+ x y))
```

Examples using Lambda

Higher-order functions

```
-> (val square (lambda (x) (* x x)))
-> (square 3)
9
-> (define twice (f) (lambda (y) (f (f y))))
-> (val 4th_power (twice square))
-> (4th_power 3)
81
-> ((twice square) 3)
81
```

History — nested scopes

Nesting of functions

```
procedure qsort(var a: array[1..1000] of integer);
    function partition(lo, hi: integer) : integer;
    begin
        // accesses array a[ ]
    end;

    procedure f(lo, hi: integer);
    var i: integer;
        begin if lo < hi then begin
            i := partition(lo, hi)
            f(lo, i-1);
            f(i+1, hi);
        end
    end

begin f(1, 1000)
end;
```

Why Lambda matters: Nested functions

Inner funs use parameters, variables of outer funs

- **cheap implementation uses "static links" or "displays"**
- **count difference in nesting depth**
- **maintain stack at run time**

  different from the call stack!
- **can identify at compile time which variables are used**

  hence "static scoping"

  (compile-time name resolution)

History — functions as arguments

Begin to treat functions as values

Example: general zero-finder

```
int findzero(int (*f)(int)) {
  int lo=0, hi=1000, k;
  while (lo + 1 < hi) {
    k = (lo + hi) / 2;
    if (f(k) < 0)  lo = k;
    else           hi = k;
  }
  return hi;
}
```

Finding roots

Nth root of k by finding zero of $x^n - k$

Can do for any n — use nested function, find its root:

```
-> (define power (x n)
     (if (= n 0) 1 (* x (power x (- n 1)))))
-> (define nth-root-of-k (n k)
     (let
       ((x-to-the-n-minus-k (lambda (x)
                              (- (power x n) k))))
       (findzero x-to-the-n-minus-k)))
-> (nth-root-of-k 3 27)
3
```

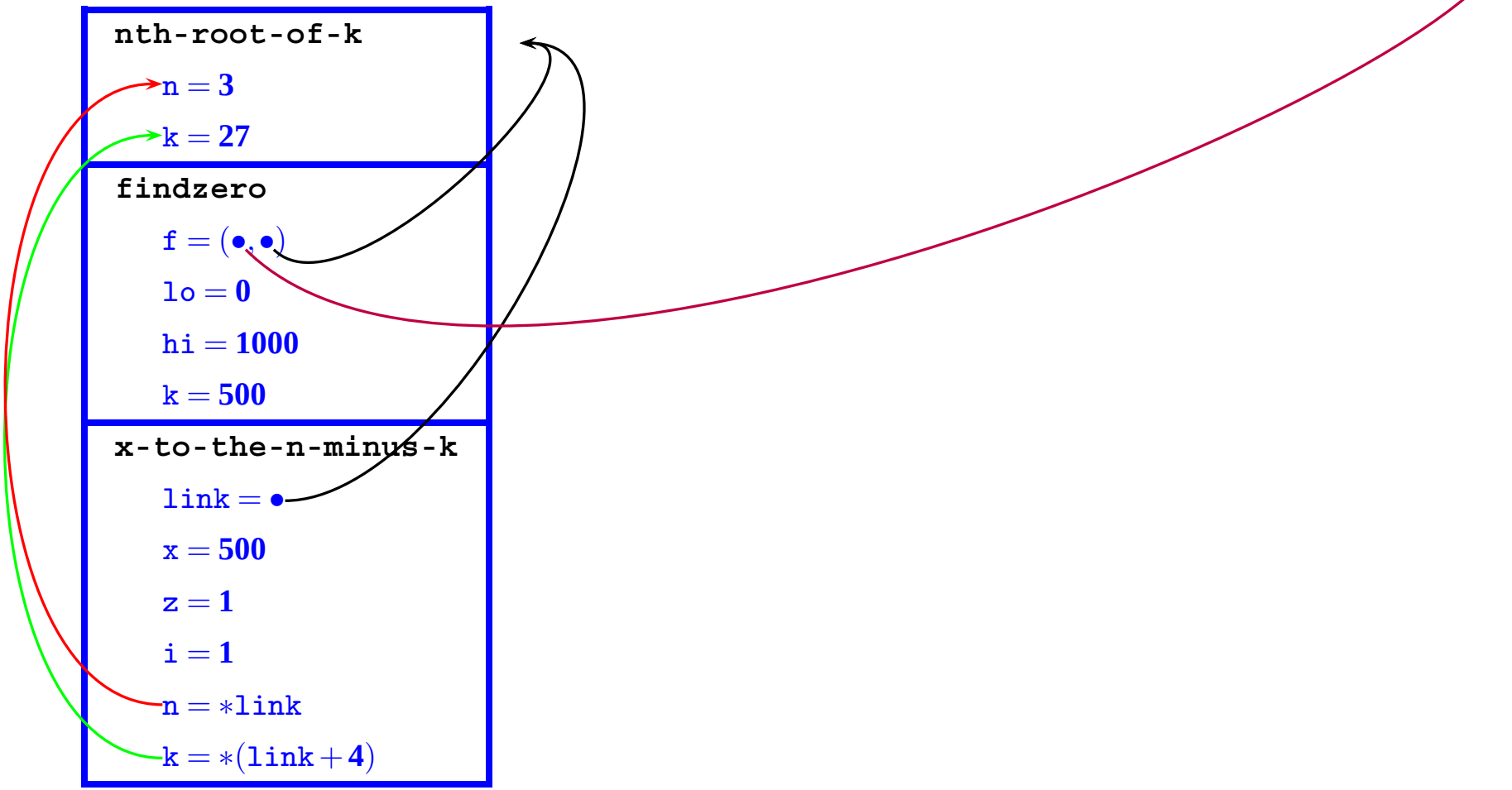
So-called “downward funargs” — down the call stack

free variables are in calling context, which is always live

Possible in Ada, Clu, Modula, Pascal...

Downward funargs

`nth-root-of-k` remains active during call to `x-to-the-n-minus-k`, so that `nth-root-of-k` can reach `n` and `k` on the call stack



More history — functions as results

Functions as values

suppose you don't want zero-finder mixed in?

```
-> (define to-the-n-minus-k (n k)
     (let
       ((x-to-the-n-minus-k (lambda (x)
                              (- (power x n) k))))
       x-to-the-n-minus-k))
-> (val x-cubed-minus-27 (to-the-n-minus-k 3 27))
-> (x-cubed-minus-27 2)
-19
```

x-to-the-n-minus-k "*escapes*" its original context

The “upward funarg problem”

How functions escape:

- **return a function**
- **assign function to global**
- **store in heap-allocated data structure**

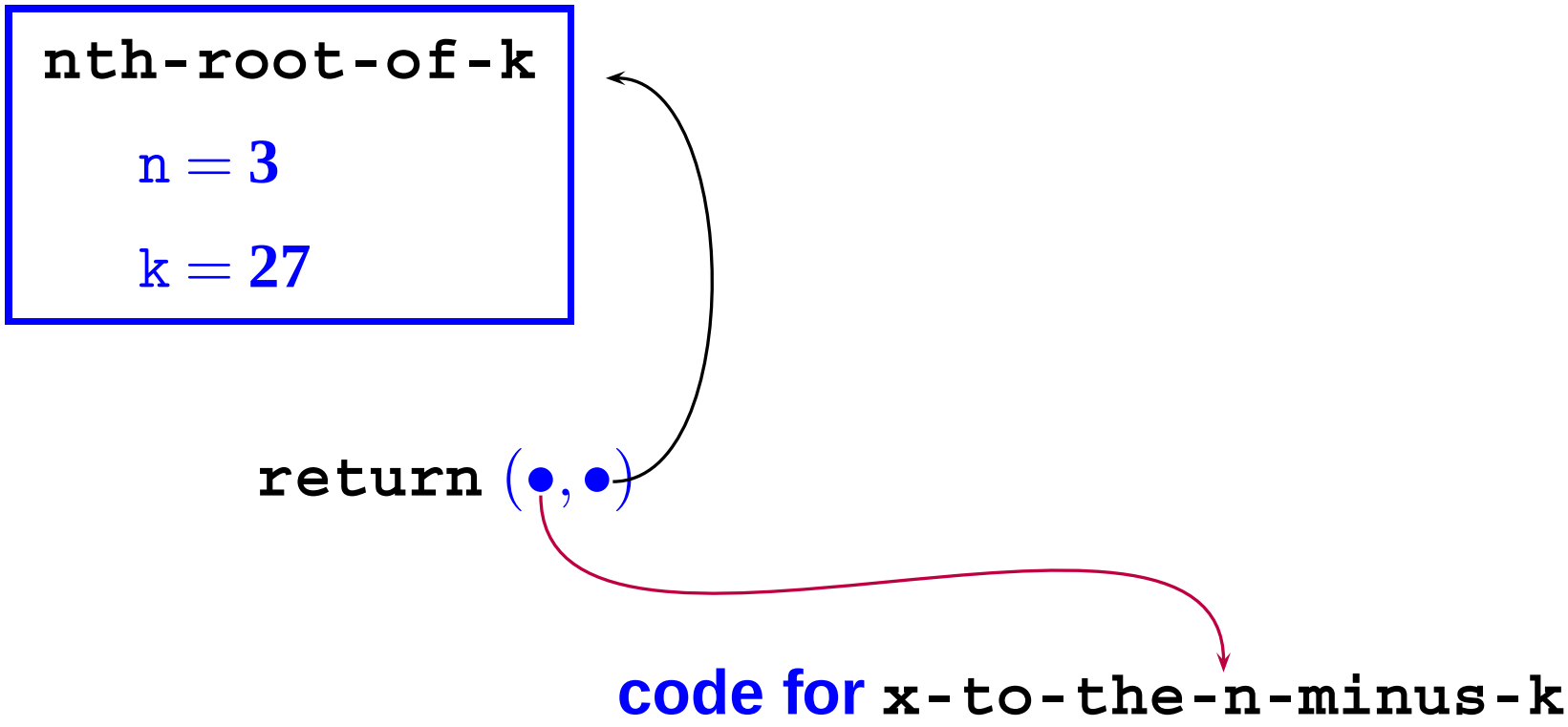
To see problem, imagine implementation:

when call to `to-the-n-minus-k` returns,

where are `n` and `k` ?

Upward funargs

`to-the-n-minus-k` returns; its parameters vanish



Closures

To have a function value, we need the equivalent of $\langle\langle \lambda x.e, \rho \rangle\rangle$, where ρ binds all the free variables of e

This agglutination is called a *closure*

In a compiled system, a record containing

- pointer to the code
- values of the free variables

Operational semantics of closures

$$\frac{}{\langle \textsf{LAMBDA}(\langle x_1,\ldots,x_n\rangle, e), \rho, \sigma\rangle \Downarrow \langle\langle \textsf{LAMBDA}(\langle x_1,\ldots,x_n\rangle, e), \rho\rangle\rangle, \sigma\rangle}\ (\textsc{MkClosure})$$

ρ: maps names to locations

σ: maps locations to values

$$\frac{\begin{array}{c}\ell_1,\ldots,\ell_n \notin \operatorname{dom}\sigma \\ \langle e,\rho,\sigma\rangle \Downarrow \langle\langle \textsf{LAMBDA}(\langle x_1,\ldots,x_n\rangle, e_c), \rho_c\rangle\rangle, \sigma_0\rangle \\ \langle e_1,\rho,\sigma_0\rangle \Downarrow \langle v_1,\sigma_1\rangle \\ \vdots \\ \langle e_n,\rho,\sigma_{n-1}\rangle \Downarrow \langle v_n,\sigma_n\rangle \\ \langle e_c, \rho_c\{x_1\mapsto\ell_1,\ldots,x_n\mapsto\ell_n\}, \sigma_n\{\ell_1\mapsto v_1,\ldots,\ell_n\mapsto v_n\}\rangle \Downarrow \langle v,\sigma'\rangle\end{array}}{\langle \textsf{APPLY}(e,e_1,\ldots,e_n),\rho,\sigma\rangle \Downarrow \langle v,\sigma'\rangle}\ (\textsc{ApplyClosure})$$

Closures change the game

Higher-order functions

- **Functions that take [existing] functions as arguments (mildly amusing)**
- **Functions that return [new] functions (surprisingly powerful)**

```
-> (define o (f g) (lambda (x) (f (g x))))
-> (define even? (n) (= 0 (mod n 2)))
-> (val odd? (o not even?))
-> (odd? 3)
#t
-> (odd? 4)
#f
```

Examples in Scheme: Currying

```
-> (val positive? (lambda (y) (< 0 y)))
-> (positive? 3)
#t
-> (val <-curried (lambda (x) (lambda (y) (< x y))))
-> (val positive? (<-curried 0))
-> (positive? 0)
#f
-> (val curry ; binary function -> value -> function
     (lambda (f)
       (lambda (x)
         (lambda (y) (f x y)))))
-> (val positive? ((curry <) 0))
-> (positive? -3)
#f
-> (positive? 11)
#t
```

λ as program structuring tool

Global variables are vulnerable:

```
-> (val seed 1)
-> (val rand (lambda ( )
       (set seed (mod (+ (* seed 9) 5) 1024))))
-> (rand)
14
-> (rand)
131
-> (set seed 1)
1
-> (rand)
14
```

λ: the ultimate protection

Idea: Hide internal variable `seed` inside λ
(nobody else can touch)

```
-> (val mk-rand (lambda (seed)
     (lambda ( )
       (set seed (mod (+ (* seed 9) 5) 1024))))))
-> (val rand (mk-rand 1))
-> (rand)
14
-> (rand)
131
-> (set seed 1)
error: set unbound variable seed
-> (rand)
160
```

λ instead of let?

`lambda` can replace `let`, but seems unnatural

```
(let ((x1 e1) ··· (xn en)) e)
```

is exactly equivalent to

```
((lambda (x1 x2 ... xn) e) e1 e2 ... en)
```

Standard higher-order functions

Common computations on lists:

`exists?`	**Does any element satisfy predicate?**
`filter`	**Select elements that satisfy predicate**
`map`	**Apply function to elements**
`fold`	**General list computations (two variations)**

List fun. `exists?`: does an element exist?

```
-> (define exists? (p? l)
     (if (null? l)
       #f
       (if (p? (car l))
         #t
         (exists? p? (cdr l)))))
-> (exists? pair? '(1 2 3))
#f
-> (exists? pair? '(1 2 (3)))
#t
-> (exists? ((curry =) 0) '(1 2 3))
#f
-> (exists? ((curry =) 0) '(0 1 2 3))
#t
```

List fun. `filter`: select some elements

```
-> (define filter (p? l)
     (if (null? l)
       '( )
       (if (p? (car l))
         (cons (car l) (filter p? (cdr l)))
         (filter p? (cdr l)))))
-> (filter (lambda (n) (<  0 n)) '(1 2 -3 -4 5 6))
(1 2 5 6)
-> (filter (lambda (n) (>= 0 n)) '(1 2 -3 -4 5 6))
(-3 -4)
-> (filter ((curry <)  0) '(1 2 -3 -4 5 6))
(1 2 5 6)
-> (filter ((curry >=) 0) '(1 2 -3 -4 5 6))
(-3 -4)
```

List filtering: composition revisited

```
-> (val positive? ((curry <) 0))
<procedure>
-> (filter positive?          '(1 2 -3 -4 5 6))
(1 2 5 6)
-> (filter (o not positive?) '(1 2 -3 -4 5 6))
(-3 -4)
```

List function `map`: apply function to list

```
-> (define map (f l)
     (if (null? l)
       '( )
       (cons (f (car l)) (map f (cdr l)))))
-> (map number? '(3 a b (5 6)))
(#t #f #f #f)
-> (map ((curry *) 100) '(5 6 7))
(500 600 700)
-> (val square* ((curry map) square))
<procedure>
-> (square* '(1 2 3 4 5))
(1 4 9 16 25)
```

Grand-daddy of list functions: fold

Idea is: $\lambda + . \lambda zero . x_1 + \cdots + x_n + zero$

Need the identity element of `+` (call it `zero`):

```
-> (define foldr (plus zero l)
     (if (null? l)
       zero
       (plus (car l) (foldr plus zero (cdr l)))))
-> (val sum  (lambda (l) (foldr + 0 l)))
-> (val prod (lambda (l) (foldr * 1 l)))
-> (sum '(1 2 3 4))
10
-> (prod '(1 2 3 4))
24
```

Another view of operator folding

```
'(1 2 3 4)   =   (cons 1 (cons 2 (cons 3 (cons 4 '( )))))
(foldr + 0 '(1 2 3 4))
             =   (+    1 (+    2 (+    3 (+    4   0 ))))
(foldr f z '(1 2 3 4))
             =   (f    1 (f    2 (f    3 (f    4   z ))))
```

`foldr` is a member of a class of transformations: catamorphisms

Works with *any* recursive datatype — often useful

Another catamorphism: `foldl` associates to left

Quick and dirty set implementation

Higher-order functions lead to compact code:

```
-> (val emptyset '( ))
-> (define member?      (x s)
     (exists? ((curry equal?) x) s))
-> (define add-element (x s)
     (if (member? x s) s (cons x s)))
-> (define union      (s1 s2) (foldl add-element s1 s2))
-> (define set-from-list (l) (foldl add-element '( ) l))
-> (union '(1 2 3 4) '(2 4 6 8))
(8 6 1 2 3 4)
```

A few higher-order functions go a long way

Example: Quicksort in 10 lines

Generalized equality for alists

“Built-in” equal won’t do for association lists

A-lists equal if each has same associations as the other:

$$al_1 = al_2 \text{ iff } al_1 \subseteq al_2 \wedge al_2 \subseteq al_1$$

```
(define sub-alist? (al1 al2)
  (not (exists?
         (lambda (pair)
           (not (equal? (cadr pair)
                        (find (car pair) al2))))
         al1)))
```

Equality from subset

```
-> (define =alist? (al1 al2)
     (if (sub-alist? al1 al2) (sub-alist? al2 al1) #f))
-> (=alist? '( ) '( ))
#t
-> (=alist? '((E coli)(I Magnin)(U Thant))
            '((E coli)(I Ching)(U Thant)))
#f
-> (=alist? '((U Thant)(I Ching)(E coli))
            '((E coli)(I Ching)(U Thant)))
#t
```