Scheme

Designed mid-70s, Guy Steele & Gerry Sussman, MIT

Strove for a clean, simple, elegant, expressive language
- few but powerful primitives which combine orthogonally

Practical implementation of the lambda calculus
- first-class, lexically-scoped, nestable procedure values (lambdas)
- first-class continuations (!)
- but eager evaluation of parameters

Other main characteristics:
- regular, parenthesized, prefix, expression-oriented syntax
- dynamically typed
- implicit pointers; garbage collection
- lists, list literals
- mostly-functional programming style
- (tail) recursion instead of iteration
- powerful arithmetic
- hygenic macros

Basics

Operations & control structures in prefix form:

```
(+ 3 4 5)
(if (>= i 0) i (- i))
(cond ((< x 0) "negative")
      ((= x 0) "zero")
      (else    "positive"))
```

- everything is an expression ⇒ everything returns a value

Top-level definitions through `define`:

```
(define **an-important-value** 3)
(define (foo a b c)
  (+ a (* b c)))
```

Local definitions through `let`, `let*`:

```
(let ((x (+ 3 4))
      (y (foo 2 x 6)))
  (let* ((x (+ x 1))
         (z (+ x 1)))
    (+ x y z)))
```

- everything an expression ⇒ everything admits local defs

Booleans

`#t`, `#f` are boolean constants
- `nil` ≠ `#f`

`and` and `or` use short-circuiting evaluation

Anything other than `#f` is true, as far as `if` is concerned
- allows predicate functions to return potentially useful data

Lists

Typical programming uses heterogenous variable-length lists

Constructors and destructors:

`(cons 3 (cons "hi" nil))` ⇒ `(3 "hi")`

`(cons 3 (cons "hi" 5))` ⇒ `(3 "hi" . 5)`

`(list 3 (list 4 5) 6)` ⇒ `(3 (4 5) 6)`

`(car (cons` *x y*`))` ⇒ *x* (a.k.a. `first`, `head`)

`(cdr (cons` *x y*`))` ⇒ *y* (a.k.a. `rest`, `tail`)

Constants:

`'(3 (4 5) 6)` ⇒ `(3 (4 5) 6)`

`'(+ (* 4 5) 6)` ⇒ `(+ (* 4 5) 6)`

`'()` ⇒ `()` (a.k.a. `nil`)

`'hi-there-bob` ⇒ `hi-there-bob` (a symbol)

Other operations:
- `length`, `append`, `reverse`, `member`, `eq?`, `equal?`

Example: association lists

Maintain list of phone numbers

```
(define phone-book
  '((bob 1234567)
    (sue 9876543)
    (jane (4568723 6542345))
    (steve unlisted)))

(define (add-number pb name numbers)
  (cons (list name numbers) pb))

(define (lookup-number pb name)
  (if (null? pb) #f
    (let* ((first-entry (car pb))
           (first-name (car first-entry)))
      (if (equal? first-name name)
          (cadr first-entry)
        (lookup-number (cdr pb) name)))))
```

Tail recursion: last thing to do before returning is recursive call

- defined to be implemented as efficiently as iteration

Lambdas

Lambda creates and returns
an anonymous, lexically-scoped function value

Examples:

```
(define square (lambda (x) (* x x)))
(square 3)

((lambda (x) (* x x))  3)

((if (< i j) + (lambda (x y) (+ x y 5)))  8 9)

(define (compose f g)
  ;; result of lambda remembers bindings of free vars f and g
  (lambda (x) (f (g x))))
(define double-square
  (compose (lambda (x) (+ x x)) square)
(define quad (compose square square))
(quad (double-square 10))
```


Functionals

Many operations over lists parameterized by function arguments

Map: takes a function and a list,
returns a list of applying function to each element of list

`(map square '(3 4 5))` ⇒ `(9 16 25)`

- generalized to *n*-ary functions over *n* list arguments

Reduce: takes a binary function `op`, an identity `id`, and a list `l`,
and returns the result of l_1 `op` l_2 `op ... op` l_n `op id`

```
(define (prod lst) (reduce * 1 lst))
(define (dot-product lst1 lst2)
  (reduce + 0 (map square lst1 lst2)))
```

Filter: takes a predicate function and a list, and returns a list
containing only those elements where predicate is true

```
(define (positives lst)
  (filter (lambda (x) (> x 0)) lst))
```

Find: takes a predicate and a list, and returns first element of list
where predicate is true

```
(define (first-positive lst)
  (find (lambda (x) (> x 0)) lst))
```


Currying

Can define functions that take a subset of their arguments, and
then return a function to take rest of arguments

- helpful if function called with some arguments fixed often,
  e.g. functionals

```
(define (mapc f) (lambda args (map f . args)))
(define square-all (mapc square))

(define (reducec op id)
  (lambda (lst) (reduce op id lst)))
(define dot-product
  (compose (reducec + 0) (mapc square)))

(define (filterc p)
  (lambda args (filter p . args)))
(define (findc p)
  (lambda args (find p . args)))

(define (positive? x) (> x 0))
(define positives (filterc positive?))
(define first-positive (findc positive?))
```

Delay and force: streams

`(delay expr)` returns a value that freezes evaluation of *expr*

`(force expr)` thaws *expr* and memoizes the result
so future thaws just return the thawed value

Can use these to implement streams and unbounded data structures:

```
(define (ints-from n)
  (cons n (delay (ints-from (+ n 1)))))

(define nats (ints-from 0))

(define (first-n n strm)
  (if (= n 0) nil
    (cons (car strm)
          (first-n (- n 1)
                   (force (cdr strm))))))

(define (map-stream f strm)
  (cons (f (car strm))
        (delay (map-stream f
                  (force (cdr strm))))))
```


Letrec

Support mutually-recursive bindings through `letrec`:

```
(letrec
  ((foo (lambda (x y) (... (bar y x) ...)))
   (bar (lambda (a b) (... (foo b a) ...))))
  ... (foo (bar 3 4) 5) ...)

(letrec ((first (cons 3 second))
         (second (cons 4 first)))
  first)
```


Continuations

Support non-local control flow through general continuations

```
(define (fast-prod lst)
  (call/cc (lambda (exit)
    (reduce
      (lambda (x y)
        (if (= x 0) (exit 0) (* x y)))
      1 lst))))
```

More flexible than just exits, since continuation procedures are first-class with unlimited extent

- simulate co-routines, scheduling, backtracking, etc. with first-class continuations

If first-class continuations,
first-class environments?
first-class stores?