First-class functions in data structures

Can put functions in data structures

E.g. a menu data structure

```
(define FileMenu
  (list (list 'Open... open-fn)
        (list 'Save save-fn)
        (list 'SaveAs... save-as-fn)
        (list 'Quit quit-fn)))

(define (click key)
  (let ((fn (assoc FileMenu key)))
    (fn)))
```


Streams using first-class functions

Goal: make a “list” that computes its next element on demand, rather than storing it

- **lazy evaluation**

Representation:
a pair of a value and a **function** to produce the next pair

```
(define (cons-stream car cdr-fn)
  (cons car cdr-fn))
(define (car-stream cons-stream)
  (car cons-stream))
(define (cdr-stream cons-stream)
  (let ((cdr-fn (cdr cons-stream)))
    (cdr-fn)))
```

Example of use: “infinite list” of ascending integers

```
(define (ints-from n)
  (cons-stream n
               (lambda () (ints-from (+ n 1)))))
(define nats (ints-from 0))
```


Functions on streams

Can define functions like `map-stream`:

- use `car-stream`, `cdr-stream`, `cons-stream` instead
- wrap tail argument *T* of `cons-stream` in `(lambda () T)`

```
(define (map-stream fn lst)
  (if (null? lst) ()
    (cons-stream
      (fn (car-stream lst))
      (lambda()
        (map-stream fn (cdr-stream lst))))))
```

Assume others like `map2-stream`, `filter-stream`, ...

- `reduce-stream` may not be useful....

Example: stream of sum of squares and doubles

```
(map2-stream + (map-stream square nats)
               (map-stream double nats))
```

→ `(0` *[conceptually:]* `2 8 15 24 35 ...)`

Control constructs

Languages support mechanisms for controlling execution flow:

Basic methods:

- procedure call & return
- conditional execution like `if`, `cond`

Advanced methods:

- looping (!)
- break, continue
- exception handling
- coroutines, threads
- ...

Continuations

Scheme supports all advanced control constructs with one special primitive: continuations

A continuation is a procedure that can be called (with a result value) to do "the rest of the program," exiting the current task

- enables parameterizing a procedure by "what to do next"
- enables having multiple return places, not just the one normal return, for different kinds of outcomes

Example:
`find` parameterized by success and failure continuations

```
(define (find pred lst if-found if-not-found)
  (cond ((null? lst) (if-not-found))
        ((pred (car lst)) (if-found (car lst)))
        (else (find pred (cdr lst)
                    if-found if-not-found))))


(find is-positive? '(...)
      (lambda(x) 'Yes)
      (lambda() 'No))
```


Current continuation

The normal return point is an implicit continuation: it takes the returned value and does the rest of the program

Scheme makes this continuation first-class upon request using `call-with-current-continuation` (a.k.a. `call/cc`)

`call/cc` takes an argument procedure of one argument, *P*, and invokes *P* passing the current continuation, *K*, as *P*'s argument

- if *P* returns *V* normally, `call/cc` returns *V*
- if *P* invokes *K*, passing one argument value, *V*, *P* quits and `call/cc` returns *V*

Example: computing products with an early exit

```
(define (prod lst)
  (call/cc (lambda (exit)
    ;; define a local helper procedure:
    (define (*-or-exit val rest)
      (if (= val 0)
          (exit 0) ;; break out of the reduce
          (* val rest)))
    (reduce *-or-exit 1 lst))))
```


Features of continuations

Continuations can be used to program exception handling, stack unwinding code, coroutines and thread packages, backtracking, etc.

- no special features needed other than continuations

Because continuations are first-class data values, can do things not possible in other more restricted settings

- e.g. using "exceptions" to implement threads or backtracking

Very powerful, which can be very confusing

Meta-circular interpreter

Exploit similarity between programs and data: write an interpreter for Scheme in Scheme

```
(process '((define x 5)
           (define fact (lambda (n)
              (if (<= n 1) 1
                  (* n (fact (- n 1))))))
           (fact x)))
   → 120
```

Key components

`process-helper`: processes commands one by one, passing off either to `process-define` or `evaluate`

`process-define`: processes `define` inputs

`evaluate`: evaluates expressions; a big case statement

- numbers, `()`, `#f`, `#t`: self-evaluating
- variables: do lookup in current vars environment
- `lambda`s: represented by itself [not a closure?]
- `if` special form: evaluate test and either then or else exprs
- calls: evaluate function & arg expressions, call `apply-op`

`apply-lambda`: extend environment with formal→actual bindings and evaluate body

`vars`: association list mapping variable names to their values

- initialized in `initial-env` to predefined functions
- extended in `process-define` to implement `define`
- extended in `extend-vars` when calling function with bindings from formals to actuals

Scheme evaluation

Goal: small set of powerful concepts, applied orthogonally, with good run-time support

- dynamic type safety, automatic garbage collection
- all computations are expressions with results
- list data structure, powerful side-effect-free operations, list-and-symbol constants
- first-class functions
- first-class continuations

No support for early error checking

Readability can be compromised by “elegant” code