CSE 505: Programming Languages

Professor: Craig Chambers

TAs: Vass Litvinov, Jonathan Aldrich

Goals:

- study of major concepts in programming languages
- programming experience in languages embodying concepts
- knowledge of key language design trade-offs, techniques
- exposure to formal semantics
- ideas of current research issues

Course outline

Functional languages (e.g. Scheme, ML, Haskell)

- side-effect-free programming
- recursive first-class functions, recursive data structures
- first-class control (continuations)
- eager vs. lazy evaluation
- polymorphic static type inference

Object-oriented languages (e.g. Smalltalk, Cecil)

- inheritance, subclass polymorphism
- models of dynamic dispatching
- polymorphic static type systems

Declarative languages (e.g. CLP(R))

- declarative programming, search algorithms
- constraints

Formal semantics

- lambda calculus
- operational & denotational semantics
- natural deduction rules

Coursework

Functional & OO sections:

- 1-2 homeworks each
- 1 programming project each
- 1 exam each

Declarative & semantics sections:

- 1 homework each

Course term paper

Final exam

Programming Language Design Principles & Goals

Programmers interact with program through language

- quickly learn programming language
- quickly express intent, model application domain
- easily modify/extend program
- read other people's code, understand intent
- reason about & ensure correctness; debug
- reason about performance trade-offs;
  ensure good performance

Tools interact with program through language

- compilers quickly analyze, optimize, and translate
- debuggers, program understanding tools
- machine program generators

Goals

Want language that makes these interactions easy as possible

W.r.t. programmers:

- learnable (simple, “natural”, no special cases)
- expressive, easy to write
  - for particular application domain? or in general?
- minimizes embedding of assumptions about current system
- readable, understandable; no “clever” encodings
- simple, clear feature interactions;
  no undefined combinations;
  no error-prone features;
  early, extensive automatic error checking
- simple, clear, efficient implementation model

W.r.t. tools:

- easy to automatically reason about & optimize
- easy to quickly extract important implementation info
- unambiguous
- regular enough to generate by machine

Design principles

Simplicity, regularity, orthogonality

- identify a small number of key concepts
- let them be combined in all combinations w/o restrictions

Provide abstraction mechanisms that enable
writing & checking (parameterized) code once,
reusing/instantiating code many times in wide range of ways

- anything interesting can be named and used by name
  (an abstraction), and vice versa
- anything namable can be parameterized

Be able to represent model of application as directly as possible

Fully define the semantics of programs, independent of
underlying implementation environment

Avoid error-prone constructs,
provide automatic checks on consistency for allowed
constructs

Models of typing

Strong typing, **type safety**: never apply operation to inappropriate data value w/o signalling an error

- includes array bounds checking? divide-by-0 checking?
- Scheme, ML, Haskell, Smalltalk, Cecil, Java, Prolog, safe subset of Modula-3

Weak typing: not strong

- C/C++, Pascal, Fortran, assembly

Static typing: type safety checked statically ("compile time")

- impossible for e.g. array bounds checking (value vs. type checking)
- ML, Haskell, Java, C/C++, Pascal, Fortran

Dynamic typing: not static

- Scheme, Smalltalk, Prolog

Some languages mix static and dynamic type checking

- CLU, Cecil; Java?

Typeless?

Static typing $\Rightarrow$ strong typing? Strong typing $\Rightarrow$ static typing?

Trade-offs

Strong typing safer, more reliable, more debuggable

Weak typing faster (than dynamically-checked strong typing), more flexible (in low-level systems code)

Static typing gives errors earlier, aids readability, improves efficiency

Dynamic typing is simpler, easier to write, more flexible

Scheme

A Lisp dialect:

- dynamically, strongly typed
- expression-oriented, largely side-effect-free
- functional
- list-oriented, garbage-collected heap-based
- **highly regular and expressive**

Lisp designed by McCarthy in late 50's

Scheme dialect introduced by Steele and Sussman in mid 70's, embodying an executable version of the lambda calculus

Typically used with interactive read-eval-print loop, with interpreted implementation

- rapid development of small programs
- good debugging support

Good compilers exist, too

Syntax

```
Program ::= { Definition | Expr }

Definition ::=
          (define id Expr)
        | (define (id_fn id_formal1 ... id_formalN)
            Expr)

Expr  ::= id
        | Literal
        | SpecialForm
        | (Expr_fn Expr_arg1 ... Expr_argN)

Literal ::= int | float | string
          | symbol | ...

SpecialForm ::=
          (if Expr_test Expr_then Expr_else)
        | ...
```

Examples

Identifiers:

```
x  x1_2  is-string?  make-string!  <=
```

Literals (self-evaluating expressions):

```
3  -4  0.34 -5.6e-7  "hi there"  ""  #t  #f
```

Definitions, expressions, and evaluation results:

```
(+ 3 4)          → 7
(define seven (+ 3 4))
seven            → 7
(+ seven 8)      → 15
(define (square n) (* n n))
(square seven)   → 49
(define (fact n)
  (if (<= n 0)
      1
      (* n (fact (- n 1)))))
(fact 20)        → 2432902008176640000
```

Everything is an expression, including “control structures”

Prefix operators & function calls is regular, but not “traditional”

Special forms

Regular call expressions evaluation all arguments then invoke procedure

- user-defined procedures work this way

Special forms are special “functions” where arguments aren’t all treated as expressions to be evaluated first

- can define new special forms using special macros

Example:

```
(define x 0)
(define y 5)
(if (= x 0) 0 (/ y x))        → 0
(define (my-if test then else)
  (if test then else))
(my-if (= x 0) 0 (/ y x))     → error!
```

Other special forms

`cond`: like `if-elseif-...-else` chain:

```
(cond ((> x 0) 1)
      ((= x 0) 0)
      (else -1))
```

Short-circuiting `and` and `or`:

```
(or (= x 0) (> (/ y x) 5) ...)
```

`let`: local variable bindings:

```
(define x 1) (define y 2) (define z 3)
(let ((x 5)
      (y (+ 3 4))
      (z (+ x y z)))
  (+ x y z))          → 5+7+(1+2+3)=18
```

`let*`: sequentially defined local variable bindings:

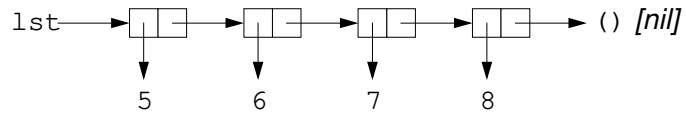
```
(let* ((x 5)
       (y (+ 3 4))
       (z (+ x y z)))
  (+ x y z))          → 5+7+(5+7+3)=27
```


Lists

Standard aggregate data structure is the list
(arrays are available, but less commonly used)

A list is a singly-linked chain of cons-cells (a.k.a. pairs):

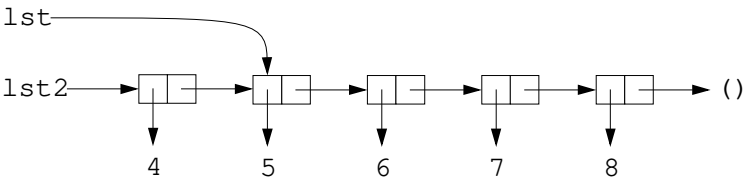
```
(define lst (list 5 6 7 8))
```




```
lst      → (5 6 7 8)
```

`cons` adds a new element to the front of a list, non-destructively:

```
(define lst2 (cons 4 lst))
```




```
lst      → (5 6 7 8)
lst2     → (4 5 6 7 8)
```

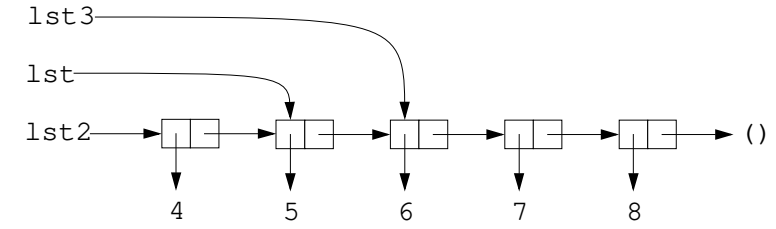
List accessors

`car` extracts the first element of the list:

```
(+ (car lst) (car lst2))    → 9
```

`cdr` extracts the rest of the list after the first, non-destructively:

```
(define lst3 (cdr lst))
```




```
lst      → (5 6 7 8)
lst2     → (4 5 6 7 8)
lst3     → (6 7 8)
```


List properties

`()` is the empty list literal, which is the result of `(list)`

- `#f` and `()` are the same in our implementation (yuck!)

Lists can be heterogenous, due to dynamic typing:

```
(list 3 4.5 () "hi" (list 3 5))
        → (3 4.5 () "hi" (3 5))
```

Lists can be nested within lists

E.g. an association list of key-value pairs:

```
(define Zips (list (list "Seattle" 98195)
                   (list "Boston" 02115)
                   (list "Reston" 22091)))
        →  (("Seattle" 98195)
            ("Boston" 02115)
            ("Reston" 22091))
```

Very simple, regular, flexible structure

Quoting

Can create list literals via `quote` or `'` special form:

```
(list 3 (list 4 5) 6)    → (3 (4 5) 6)
(quote (3 (4 5) 6))      → (3 (4 5) 6)
'(3 (4 5) 6)             → (3 (4 5) 6)
```

Quoted identifiers are symbol constants:

```
'positive                → positive
(car '(if (> a b) 3 4)) → if
```

Programs and data share same regular syntax

Equality

`eqv?`: test whether two things are the same
`equal?`: test whether two things are isomorphic

```
(eqv? 3 3)                          → #t

(eqv? (list 3 4) (list 3 4))        → #f
(let ((lst (list 3 4)))
  (eqv? lst lst))                   → #t
(equal? (list 3 4) (list 3 4))      → #t

(eqv? "hi" "hi")                    → #f
(equal? "hi" "hi")                  → #t

(eqv? 'hi 'hi)                      → #t

(eqv? () ())                        → #t
```

`eq?`: low-level and more precise than `eqv?`,
probably should be avoided

Other predicates

Numeric comparisons like =, <, >=

- no not-equal; use `not` function instead

Type-testing predicates, because of dynamic typing:

```
null?
pair?
symbol?
boolean?
number?  integer?  ...
string?
...
```

Note `?` naming convention

Binding

All variables *refer* to data values

`let` binding & parameter passing do not copy r.h.s. value
but introduce new name that refers & shares r.h.s. value

- "call by pointer value", "call by sharing"

```
(define lst (list 3 4 5))
(define (f lst2)
  (let ((lst3 lst2))
    (and (eqv? lst lst2) (eqv? lst2 lst3))))
(f lst)     → #t
```

Simple, regular model

Garbage collection

All data structures implicitly allocated in the global heap

- gives data structures unlimited potential lifetime (**indefinite extent**)
- allows all data to be passed around and returned from functions and stored in other data structures

System automatically reclaims memory for unused data structures

- programmer doesn't have to worry about freeing unused memory
- avoids freeing too early (dangling pointer bugs) and freeing too late (storage leak bugs)
- system can sometimes be *more* efficient than programmer

Indefinite extent is simple, expressive

Automatic storage reclamation is much less error-prone

Some small efficiency concerns

Recursion over lists

Lists are a recursive data type:

A **list** is `()` (base case)

or a pair of a value and another **list** (inductive case)

Well-suited for manipulation by recursive functions with same structure as the recursive data type definition:

```
(define (f lst ...)
  (if (null? lst) ...base-case...
      ... (car lst) ... (f (cdr lst) ...) ...
```

Examples:

```
(define (length lst)
  (if (null? lst) 0
      (+ 1 (length (cdr lst)))))
```

```
(define (sum lst)
  (if (null? lst) 0
      (+ (car lst) (sum (cdr lst)))))
```

Another example

;; find value associated with given `key` in `alist`
;; `alist` is a list of key-value pairs

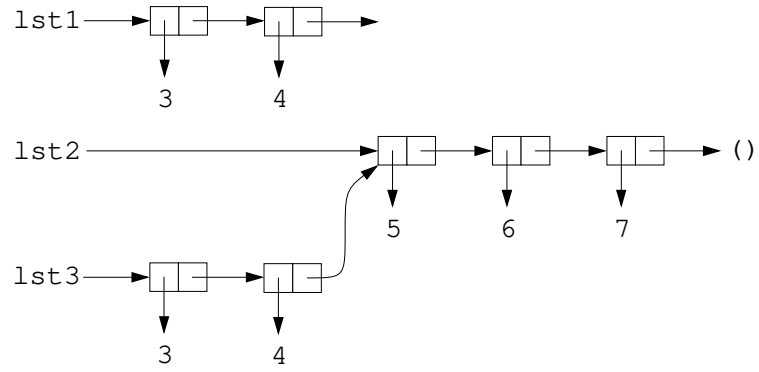
```
(define (assoc key alist)
  (if (null? lst) #f
    (let* ((pair (car alist))
           (k (car pair))
           (v (cdr pair)))
      (if (equal? key k) v
        (assoc key (cdr alist))))))
```

```
(assoc "Boston" Zips)    → 02115
(assoc "Tacoma" Zips)    → #f
```


Another example

;; append two lists, non-destructively

```
(define (append lst1 lst2)
  (if (null? lst1) lst2
    (cons (car lst1)
          (append (cdr lst1) lst2))))
(define lst1 '(3 4))
(define lst2 '(5 6 7))
(define lst3 (append lst1 lst2))
lst3                → (3 4 5 6 7)
```



Sharing list substructures is common,
and safe if no side-effects

Side effect special forms

`set!`: rebind a variable to refer to a different value

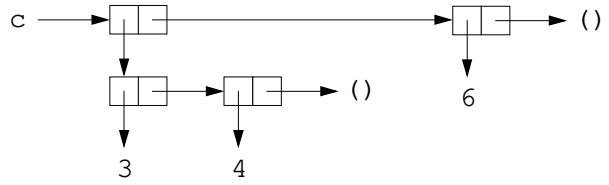
```
(define x 5)
(set! x '(6 7))
x                    → (6 7)

(define (test lst)
  (set! lst (cons 1 lst))
  lst)
(test '(2 3))        → (1 2 3)
```


Side effects on cons cells

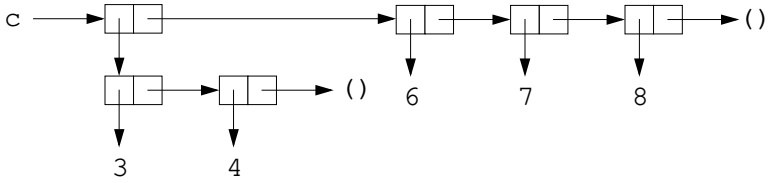
`set-car!`, `set-cdr!`: rebind head, tail of cons cell

```
(define c (list 5 6))
(set-car! c (list 3 4))
```




```
c                    → ((3 4) 6)

(set-cdr! (cdr c) (list 7 8))
```




```
c                    → ((3 4) 6 7 8)
```

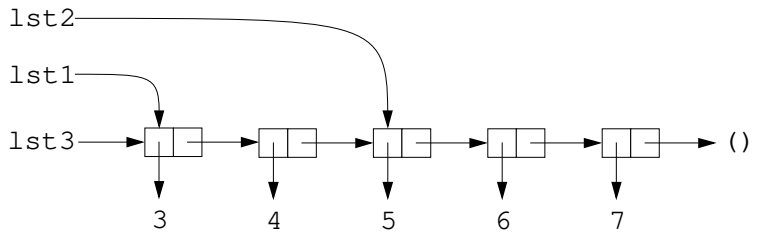
Example

`append!`: destructive `append`

```
(define (append! lst1 lst2)
  (cond ((null? lst1) lst2)
        ((null? (cdr lst1))
              (set-cdr! lst1 lst2)
              lst1)
        (else (append! (cdr lst1) lst2)
              lst1)))
```

```
(define lst1 '(3 4))
(define lst2 '(5 6 7))
(define lst3 (append! lst1 lst2))
```

`lst3` → `(3 4 5 6 7)`



`append!` more efficient than `append`, but
more complicated to use correctly in face of rampant sharing

Recursion vs. iteration

Recursion more general than iteration

- equivalent to iteration plus a stack

Recursion often considered less efficient (both time and space) than iteration

- procedure calls and stack allocation/deallocation on each "iteration"

Tail recursion: recursive call is last operation before returning

- Scheme requires tail recursion to be implemented just as efficiently as iteration

Examples:

```
(define (last lst)
  (if (null? (cdr lst)) (car lst)
      (last (cdr lst))))
```

```
(define (member x lst)
  (cond  ((null? lst) #f)
         ((eqv? x (car lst)) #t)
         (else (member x (cdr lst)))))
```

Converting to tail recursive form

This isn't tail recursive:

```
(define (factorial n)
  (if (= n 0) 1
      (* n (factorial (- n 1)))))
```

but we can define a tail-recursive version
using a helper procedure with an "accumulator" argument:

```
(define (factorial n)
  (fact-iter n 1))
(define (fact-iter n result)
  (if (= n 0) result
      (fact-iter (- n 1) (* n result))))
```


First-class functions

Functions are themselves data values that can be passed around and called later

- function to call is a full-fledged expression

Procedures can take as parameters other procedures (not just plain data), allowing code to be parameterized by code

- parameterized procedures sometimes called **functionals**

Example: a find functional with a comparison parameter function

```
(define (find pred-fn lst)
  (if (null? lst) #f
    (if (pred-fn (car lst)) (car lst)
      (find pred-fn (cdr lst)))))
```

To find first positive number in list:

```
(define (is-positive? n) (> n 0))
is-positive?            → #[compiled proc]
(find is-positive? '(-3 -4 5 7)) → 5
```

To find first nested list in list:

```
(find pair? '(3 (4 5) 6))       → (4 5)
```

More functionals

`map`: apply a function to each element of a list, collecting results into a new equal-length list

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

(map square '(3 4 5))  → (9 16 25)
(map is-positive? '(3 -4 5))  → (#t #f #t)
```

`map2`: take two lists and apply a binary function to corresponding elements, collecting results into a result list

```
(define (map2 fn lst1 lst2)
  (if (or (null? lst1) (null? lst2)) ()
    (cons (fn (car lst1) (car lst2))
          (map2 fn (cdr lst1) (cdr lst2)))))

(map2 + '(3 4 5) (6 7 8 9))  → (9 11 13)
(map2 list '("Seattle" "Boston")
           '(98195 02115))
     → (("Seattle" 98195) ("Boston" 02115))
```


Anonymous function literals

To exploit functionals better, can write a function in-line as an argument expression, without having to give it a global name

Syntax:

```
Literal ::= ...
          | (lambda (id_formal1 ... id_formalN)
              Expr)
```

Examples:

```
(map (lambda (n) (* n n)) '(3 4 5)) → (9 16 25)
(map2 (lambda (x y) (if (= y 0) 0 (/ x y)))
  '(3 4 5) '(-1 0 2))   → (-3 0 2)
```

Define for functions is "syntactic sugar" around a `lambda`

E.g.

```
(define (square n) (* n n))
```

is really just

```
(define square (lambda (n) (* n n)))
```

`cond`, `let`, `let*`, `and`, `or` are also syntactic sugars

More functionals

`reduce`: take a binary operator and apply it to pairs of list elements (like an infix operator) to compute a result

- left-to-right or right-to-left reduction?
- provide base value, or require non-empty input list and symmetric operator?

Right-to-left reduction, starting from base value:

```
(define (reduce fn base lst)
  (if (null? lst) base
    (fn (car lst)
        (reduce fn base (cdr lst)))))
```

Examples of use:

```
(define (sum lst) (reduce + 0 lst))
(define (prod lst) (reduce * 1 lst))
(define (length lst)
  (reduce (lambda (elem sum-rest)
            (+ 1 sum-rest))
          0 lst))
```


Another functional

`filter`: return a list of only those elements that pass a predicate

```
(define (filter pred lst)
  (reduce
     (lambda (val rest)
       (if (pred val) (cons val rest) rest)))
     nil lst))
```

Examples:

```
(filter is-positive? '(3 -4 5 -6))   → (3 5)
```

Returning functions from functions

If functions are first-class, then should be able to return them, store them in data structures, etc.

Example: function composition

```
(define (compose f g)
  (lambda (x) (f (g x))))

(define double-square (compose double square))
```


Currying

A curried function takes some of its arguments and returns a function which takes the remaining arguments

- the result function can be applied to many different arguments, without having to pass in the first arguments again

```
(define (mapc fn) (lambda (lst) (map fn lst)))

(define squarer (mapc square))
(squarer '(3 4 5))  → (9 16 25)
(define sum-of-squares (compose sum squarer))

(define (map2c fn) (lambda (lst1 lst2) ...))

(define (reducec fn base) (lambda (lst) ...))
(define sum (reducec + 0))
(define prod (reducec * 1))

(define (filterc pred) (lambda (lst) ...))
(define positives (filterc is-positive?))
(define negatives
  (filterc (lambda (x) (< x 0))))
```

Lexical scoping and block structure

Scheme has a hierarchy of nested scopes of name bindings

- global scope for all top-level names (from `define`)
- `lambda`, `let`, `let*` introduce nested scopes
  - `define` of a function introduces a nested scope, since it's sugar for `lambda`

The same name can be bound only once in a given scope, but in many different scopes

For a given reference to a name, which binding is meant?

Lexical scoping (a.k.a. static scoping):
search from use *outwards through static program text* to first matching declaration

Dynamic scoping:
search from use *upwards through call stack* to first matching declaration

Example

```
(define x 100)
(define bar (lambda (x)
  (foo 1 2)))
(define foo (lambda (y z)
  (let ((w (+ y 1)))
    (let ((y w))
      (+ x y)))))
```

Under lexical scoping (as in Scheme):

```
(bar 5)            → 102
```

Under dynamic scoping (as in some older Lisps):

```
(bar 5)            → 7
```

Easier to reason about non-local references with lexical scoping

- variables defined outside some scope: "free variables"

First-class procedures and scoping

Lexical scoping is interesting if returning a function with free variables

- how to remember bindings of free variables?

```
(define (compose f g)
  (lambda (x) (f (g x))))

(define double-square (compose double square))
(define square-double (compose square double))

(double-square 5)
(square-double 5)
```

Where do bindings for `f` and `g` come from?

How are these two calls distinguished?

All curried functions have free variables like this

Many anonymous lambda arguments (e.g. to `map` et al.) have free variables

Closures

To support lexically nested procedures which can be returned out of their enclosing scope, must represent as a **closure**: a pair of code address and **environment**

- environment records bindings of free variables
- closure no longer dependent on enclosing scope
- pair and environment must be heap-allocated
- e.g. Scheme, ML, Haskell, Smalltalk, Cecil

If only allow to pass nested procedures down, not return them, then can implement more cheaply

- environment can be stack-allocated, not heap-allocated
- e.g. Pascal, Modula-3

If allow nested procedures but not first-class procedures, then cheaper still

- do not need pair, just extra implicit environment argument
- e.g. Ada

If allow first-class procedures but no nesting, then can implement with just a code address

- e.g. C, C++

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)))
```


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," "where to return to"
- 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)
    (reduce
      (lambda (x accum)
        (if (zero? x)
          (exit 0) ;; break out of loop, return 0
          (* x accum) ;; continue multiplying
        ))
      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

Assessment of Scheme

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