Extended Example

```
(define (my-map func lst)
   (if (null? lst)
       '()
       (cons (func (car lst))
             (my-map func (cdr lst)))))

(define (const-func val)
   (lambda (x) val))

(define const1 (const-func 1))
(define const5 (const-func 5))

(my-map const1 '(1 2 3))
(my-map const5 '(1 2 3))
```


Variable Argument Lengths

Scheme has primitive functions `+`, `list`, `map`, etc. that accept a variable number of arguments. E.g.

```
(list)          => ()
(list 1)        => (1)
(list 1 2)      => (1 2)
```

Can we define our own?

You Bet!

These are identical:

```
(define (my-list . args) args)

(define my-list (lambda args args))
```

Note this “dot” notation is the same thing you get when you `cons` something to a non-list.

What if we want to require some arguments?

You Bet! (cont)

```
(define (my-map func . lists) ...)

(define my-map (lambda (func . lists) ...))
```

Bootstrapping

Creating something from a simpler version. For Example:

```
(define (my-map1 func lst)
   (if (null? lst)
       '()
       (cons (func (car lst)) (my-map1 func (cdr lst)))
   )
)

(define (my-map func . lists)
   (if (null? (car lists))
       '()
       (cons (apply func (my-map1 car lists))
             (apply my-map (cons func (my-map1 cdr lists)))
       )
   )
)
```


The Power of Lambda

Look for powerful constructs in a language.

Lambda is powerful.

We have already seen that it can be used in place of `let`.

What else can it do?

`cons` cells

Scheme has primitive procedures `cons`, `car`, and `cdr`.

Can we get by without them? Yes! Define abstract `cons` cell using `lambda` for implementation:

```
(define (my-cons the-car the-cdr)
   (lambda (which)
      (cond ((eq? which 'car) the-car)
            ((eq? which 'cdr) the-cdr)
            (else (error "cons cell usage error"))
      )
   )
)

(define (my-car the-pair)
   (the-pair 'car)
)

(define (my-cdr the-pair)
   (the-pair 'cdr)
)
```


Manipulating Functions

We have seen:

- Function creation (static).
- Function application (static and dynamic).
- Functions in variables

We can also:

- Manipulate functions to dynamically create new functions.

Dynamic Function Creation

Facilitated by:

- `lambda` to create new functions.
- First-class-ness of functions.
- Persistent environments.

Currying

Named for Haskell B. Curry

- Given n argument function f
- Specify k arguments to get $n-k$ argument function g.
- If $g = \text{curry}(f, a_1, \ldots, a_k)$ then $g(a_{k+1}, \ldots, a_n) = f(a_1, \ldots, a_n)$

Currying (cont)

A `curry` for a two argument function, f:

```
(define (curry f x)
   (lambda (y) (f x y))
)
```

But we can use variable argument lengths to write a general `curry`:

```
(define (curry f . first-args)
   (lambda second-args (apply f (append first-args second-args)))
)
```


Uses for `curry`

Lambda definitions are hard to read, curried functions are more clear.

```
(define add-lists (curry map +))
(add-lists '(1 2 3) '(4 5 6))

(map (curry + 5) '(1 2 3))

(define positive-only (curry filter positive?))

(define sum (curry foldr + 0))
(define sum (curry apply +))
```

Composing

Define `compose` to take two single argument functions f and g and return a function h that is f composed with g. That is

$$h(x) = f(g(x))$$

For example:

```
(define my-cadr (compose car cdr))
(define my-cadaar (foldr compose identity (list car cdr car car)))
```


Reversing Arguments

Define `reverse-args` that takes a function f and returns a new function g that takes the arguments in reverse. That is:

$$h(x, y) = f(y, x)$$

More Examples

```
(define my-reverse (curry foldl (reverse-args cons) '()))

(define compose-list (curry foldr compose identity))
(define my-cadaar (compose-list (list car cdr car car)))

;; from eval definition
(map (lambda (arg) (eval arg env)) expr)
;; becomes
(map (curry (reverse-args eval) env) expr)
```


Building Blocks

We have seen some useful higher-order building blocks: `map`, `foldr`, `curry`, `compose`, and `reverse-args`.

Scheme could provide these.

Instead Scheme provides the equipment to make them and more.

Side-Effects!

Side-effects are:

- Doing I/O (read or write).
- Changing a variable value.
- Crashing: `(/ 1 0)`.

So far we have only seen the purely functional subset of Scheme.

Input and Output

write/display outputs a scheme object. E.g. `(display "hello world")`.

newline outputs a newline.

write-char outputs a character.

read reads in a scheme object.

read-char reads in a character.

readline (Guile only) reads in a line as a string.

Setting Variable Values

In general commands that change values end in `!`:

set! sets variable value: `(set! x 10)`

set-car! sets car of existing `cons` cell.

set-cdr! sets cdr of existing `cons` cell.

Use these sparingly!

Sequencing Operations

Use `begin` to sequence expressions:

```
(define x 0)
(define y 0)
(begin
   (display "enter two numbers: ")
   (set! x (read))
   (set! y (read))
   (display "the sum is ")
   (display (+ x y))
   (newline)
)
```

Return value is value of last expression.

With no side-effects sequencing expressions is not useful:

```
(begin
   (+ 1 2 3)
   (car '(1 2 3))
)
```

Sequencing Operations (cont)

Special forms `lambda`, `let`, and `cond` allow for sequencing operations:

```
(define (welcome)
   (display "hello world")
   (newline)
)

(cond (#f (display "never") 0)
      (#t (display "always") 1)
)

(let ((x 5))
     (display "inside let\n")
     (+ x 10)
)
```

Value of sequence is value of last expression.

Examples of `set!`

```
(define x 10)
(define (x-plus y) (+ x y)

(x-plus 2)

(set! x 100)

(x-plus 2)
```


`fib` Example

Ever wonder how many times recursive `(fib 30)` calls `+`?

```
(define (fib n)
    (if (<= n 1)
        n
        (+ (fib (- n 1)) (fib (- n 2)))
    )
)

(define plus +)
(define count 0)
(define (+ . args)
   (set! count (plus 1 count))
   (apply plus args)
)

(fib 30)                    => 832040
count                       => 1346268
```


List Surgery

Can set `car` and `cdr` of a `cons` cell with `set-car!` and `set-cdr!`.

```
(define x '(1 2 3))
(define y '(a b c))
(define z (cdr x))
(define w (append x y))

(set-cdr! x '(200 300))
(set-cdr! (cdr y) '(ccc))
(set-car! z 99)

x
y
z
w
```

Note: cannot `set-c[ad]r!` the empty list. Why?

Circular Lists

```
(define x '(1 2 3))
(set-cdr! (cddr x) x)

x
```


set! vs set-car!

`set!` sets binding in environment.

`set-car!` sets binding in heap.

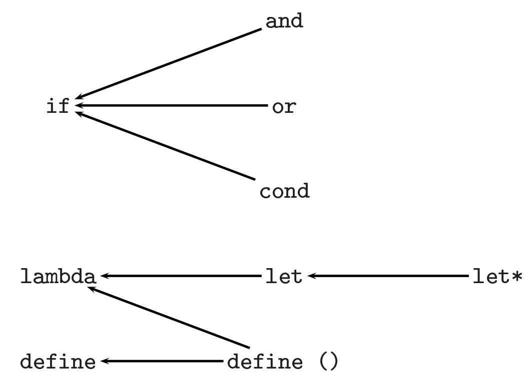
Compare:

```
(define (fix-car1 lst)
   (set! lst (cons 10 (cdr lst)))
)

(define (fix-car2 lst)
   (set-car! lst 10)
)

(define x '(1 2 3))
(define y '(1 2 3))

(fix-car1 x)
(fix-car2 y)

x
y
```


Derived Expressions

(aka Source-to-Source transformations)
These are special-forms that can be stated in terms of other special-forms.

Redundant in that Scheme is equally as expressive without them.



let->combination

Turn a symbolic `let` expression into a `lambda` and application. For example:

```
guile> (let->combination '(let ((x (+ 5 6)) (y 10)) (+ x y)))
((lambda (x y) (+ x y)) (+ 5 6) 10)

(define (let->combination expr)
   (let ((bindings (cadr expr))
         (body     (cddr expr))
        )
        (let ((vars (map car bindings))
              (exps (map cadr bindings))
             )
             (cons (cons 'lambda (cons vars body)) exps)
        )
   )
)
```

Recall Nested `let`s: `let*`

Derived form: `let*`

```
(let* ((<var-1> <expr-1>)        ⇒   (let ((<var-1> <expr-1>))
       (<var-2> <expr-2>)                 (let ((<var-2> <expr-2>))
       ...                                   ...
       (<var-n> <expr-n>)                      (let ((<var-n> <expr-n>))
      )                                             <body>
      <body>                                   )
)                                         )
                                     )
```


`and->if`

Convert `(and <e1> <e2> <e3>)` to an `if` expression