Self-Interpretation

- "language of abstract programming" (LAP)
  = core of *all* functional programming
  languages
  ... including LISP

- Any functional program can be expressed in this language as the form
  `letrec ... and ... and ... in expr`
  allows us to define arbitrary functions
  where "and" allows mutually recursive definitions

Eg: Reverse a list:

```
letrec app(X,Y) = if null(X) then Y
        else cons(car(X), app(cdr(X), Y))  and
 reverse(X) = if null(X) then nil
        else app(reverse(cdr(X)), cons(car(X),nil))
in reverse((1 2 3))
```


LAP Interpreter

- LAP interpreter will serve as basis for interpreting any functional programming language.

$\Rightarrow$ develop interpreter through number of stages, from simplest to more complex

- Use LAP to implement LAP
  so interpreter for LAP expressed as function called "`eval`" in LAP

- To avoid confusion between
  "()" of λ-calculus and
  parentheses that delimit parameter lists
  in our functions
  our subroutines will use square braces, "`eval[···]`"

LAP Interpreter Version 1

Just λ-calculus; no syntactic simplification:

$\langle expression \rangle$:= $\langle identifier \rangle$ | $\langle application \rangle$ | $\langle function \rangle$

$\langle application \rangle$:= ($\langle expression \rangle$ $\langle expression \rangle$)

$\langle function \rangle$:= (λ $\langle identifier \rangle$ "|" $\langle expression \rangle$)

$\langle identifier \rangle$:= a | b | c ...

Useful abstract functions:
E $\equiv$ arbitrary lambda expression:

Predicates: `is-id[E]`: true iff `E` is identifier
`is-function[E]`: true iff `E` is lambda function
`is-application[E]`: true iff `E` is an application

Eg: `is-id[x]` $\Rightarrow$ `T`
`is-id[(λx | x)]` $\Rightarrow$ `F`
`is-function[(λx | x)]` $\Rightarrow$ `T`
`is-function[((λx | x) A)]` $\Rightarrow$ `F`
`is-application[((λx | x) (λy | y))]` $\Rightarrow$ `T`
`is-application[(λx | x)]` $\Rightarrow$ `F`

Other Useful Functions

Selectors:
`get-fun[E]`: get function from application `E`
`get-arg[E]`: get argument from application `E`
`get-id[E]`: get identifier from function `E`
`get-body[E]`: get body expression from function `E`

Eg: `get-fun[(((λx | x) A) ((λy | y) B))]`
$\Rightarrow$ `((λx | x) A)`
`get-arg[(((λx | x) A) ((λy | y) B))]`
$\Rightarrow$ `((λy | y) B)`
`get-id[(λx | ((λy | y) A))]` $\Rightarrow$ `x`
`get-body[(λx | ((λy | y) A))]` $\Rightarrow$ `((λy | y) A)`

Constructors:
`create-fun[x, E]`: construct `(λx | E)`
`create-appl[A, B]`: construct `(A B)`
`new-id[]`: return a new identifier symbol

Eg: `create-fun[x, ((λy | y) B)]` $\Rightarrow$ `(λx | ((λy | y) B))`
`create-appl[((λx | x) A), ((λy | y) B)]`
$\Rightarrow$ `(((λx | x) A) ((λy | y) B))`
`new-id[]` $\Rightarrow$ `x`

Eval (aka Interpreter)

`eval[e]` evaluates its argument expression `e`

```
eval[e] =
 if is-id[e] then e
 elseif is-function[e]
   then create-fun[ get-id[e], eval[ get-body[e] ] ]
 else apply[ get-fun[e], get-arg[e] ]
```

`apply[f, a]`
applies "`f`" to "`a`"
using *applicative order reduction*

```
apply[f, a] =
 if is-id[f]
   then if is-id[a] then (f a)
       else create-appl[f, eval[a]]
 elsif is-application[f] then apply[eval[f], eval[a]]
   else eval[ subs[eval[a], get-id[f], get-body[f]] ]
```

`subs[a, x, e]` substitutes expression `a` for identifier `x` in expression `e`

. . . see Substitution Rules from above

Examples of Eval - I

```
eval[(λy | s)]  ⇒(eval func def)
 create-fun[get-id[(λy | s)],
       eval[get-body[(λy | s)]]]
  ⇒  create-fun[y, eval[s]]
  ⇒(eval id)  create-fun[y, s]  ⇒  (λy | s)
```

```
eval[((λy | s) x)]  ⇒(eval func app)
 apply[ get-fun[ ((λy | s) x) ],
       get-arg[ ((λy | s) x) ] ]
  ⇒  apply[ (λy | s), x ]
  ⇒(apply func def)
 eval[ subs[ eval[x], get-id[(λy | s)],
          get-body[(λy | s)]]]
  ⇒(eval id)  eval[subs[x, y, s]]  ⇒  eval[s]  ⇒  s
```


Examples of Eval - II

```
eval[ ((λy | s) ((λy | s) x)) ]
  ⇒(eval func app)
apply[ get-fun[((λy | s) ((λy | s) x))],
     get-arg[ ((λy | s) ((λy | s) x)) ] ]  ⇒
apply[(λy | s), ((λy | s) x)]
  ⇒(app func def)
eval[subs[eval[((λy | s) x)],
     get-id[(λy | s)], get-body[(λy | s)]]]
⇒  eval[ subs[s, y, s] ]  ⇒  eval[s]  ⇒  s
```

```
eval[ (λx | ((λy | s) ((λy | s) x))) ]
  ⇒(eval func def)
 create-fun[ get-id[(λx | ((λy | s) ((λy | s) x)))],
     Eeval[ get-body[(λx | ((λy | s) ((λy | s) x)))]] ]
  ⇒  create-fun[x, eval[((λy | s) ((λy | s) x))]]
  ⇒  ... ⇒  create-fun[x, s]  ⇒  (λx | s)
```


"Twice" Examples

Apply `twice` to constant function that returns `s`,
apply result to identifier `n`.

```
(((λf | (λx | (f(f x)))) (λy | s)) n)
```

```
eval[(((λf | (λx | (f(f x)))) (λy | s)) n)]  ⇒
 apply[ get-fun[ (((λf | (λx | (f(f x)))) ...))  ], ],
    get-arg[(((λf | (λx | (f(f x)))) (λy | s)) n)]]
 ⇒  apply[((λf | (λx | (f(f x)))) (λy | s)), n]
  ⇒(apply func app)
 apply[ eval[((λf | (λx | (f(f x)))) (λy | s))], eval[n] ]  ⇒
 apply[ apply[ get-fun[ ((λf | (λx | (f(f x)))) (λy | s))],
 get-arg[((λf | (λx | (f(f x)))) (λy | s))]], n]  ⇒
apply[ apply[(λf | (λx | (f(f x)))),
      (λy | s)], n]
 ⇒(apply func def)
 apply[ eval[ subs[eval[(λy | s)],
        get-id[(λf | (λx | (f(f x))))],
        get-body[(λf | (λx | (f(f x))))]]], n]
 ⇒  apply[ eval[ subs[ (λy | s), f, (λx | (f(f x)))]], n]
 ⇒  apply[ eval[(λx | ((λy | s)((λy | s) x)))], n]
 ⇒  apply[(λx | s), n]
 ⇒(apply func def)
 eval[ subs[ eval[n], get-id[(λx | s)],
        get-body[(λx | s)]] ]  ⇒
eval[subs[n, x, s]]  ⇒  eval[s]  ⇒  s
```

Normal Order Reduction

- To write abstract interpreter using *normal order reduction*: Just change definition of `apply`

```
apply[f, a] =
 if is-id[f] then
     if is-id[a] then (f a)
     else create-appl[f, eval[a]]
 elsif is-application[f] then apply[eval[f], a]
     else eval[subs[a, get-id[f], get-body[f]]]
```

- Difference: arg `a` is evaluated only after application performed

  ...and `f` is evaluated

Lambda Expressions as S-expressions

- Can convert λ-calculus expr's into Lisp s-expressions:

```
Lambda expression ⟨l-exp⟩  ⟹ s-expresion ⟨s-exp⟩

⟨identifier⟩  ⟹ ⟨identifier⟩

(λ⟨identifier⟩ | ⟨l-exp⟩)
    ⟹ (lambda (⟨identifier⟩) ⟨s-exp⟩)

(⟨l-exp⟩ ⟨l-exp⟩)  ⟹ (⟨s-exp⟩ ⟨s-exp⟩)
```

```
Eg: (λy | s)  ⟹ (lambda (y) s)
    (λx | (f(f x)))  ⟹ (lambda (x) (f (f x)))
    (λf | (λx | (f(f x))))
        ⟹ (lambda (f) (lambda (x) (f (f x))))
    (((λf | (λx | (f(f x)))) (λy | s)) n)  ⟹
     (((lambda (f) (lambda (x) (f (f x))))
           (lambda (y) s)) n)
```


Can Write Interpreter...

- If all λ-expr's represented by s-exp's, then can implement each interpreter fn using Lisp function:

```
is-id(e)  ⟹ atom(e)
is-function(e)  ⟹ eq( car(e), lambda )
is-application(e)  ⟹
    not( atom(e) ) and not( eq(car(e), lambda) )
get-fun(e)  ⟹ car(e)
get-arg(e)  ⟹ cdr(e)
...
```

```
Eg: is-id( (λf | (λx | (f x))) )  ⇒
     atom( (lambda (f) (lambda (x) (f x))) ) = false
    is-function( (λf | (λx | (f x))) )  ⇒
     eq( car((lambda (f) (lambda (x) (f x)))), lambda
    )
     = eq(lambda, lambda) = true
    is-application( (λf | (λx | (f x))) )  ⇒
     not( atom((lambda (f) (lambda (x) (f x)))) ) and
       not( eq( car((lambda (f) (lambda (x) (f x)))),
             lambda) )
    =  not(false) and not(eq(lambda, lambda))
    =  true and not true  =  false
```


Obvious Extensions

Extend to more general lambda expressions

- Multiple parameter lambda expressions:

```
(λ⟨id1 id2 ...  idn⟩ | ⟨l-exp⟩)  ⟹
(lambda (⟨id1 id2 ...  idn⟩) ⟨s-exp⟩)
```

- Can add built-in functions, constants:

```
(+ 3 4)  ⟹ (+ 3 4) = 7
(cons (+ 3 4) nil)  ⟹ (cons (+ 3 4) nil) = (7)
```

- Use s-expressions to represent other abstract constructs:

```
(let (⟨id1⟩ ⟨id2⟩ ...  ⟨idn⟩)
       (⟨s-exp1⟩ ⟨s-exp2⟩ ...  ⟨s-expn⟩)
   ⟨s-exp⟩)

(if ⟨s-exp⟩ ⟨s-exp⟩ ⟨s-exp⟩)

(letrec (⟨id⟩) (⟨s-exp⟩) ⟨s-exp⟩)
```

- special "function" `quote` to inhibit evaluation ...to represent data:

```
(quote ⟨s-exp⟩)
```

... re-write interpreter to support these extensions:

Take-home Msgs

- Abstract programming can be implemented by interpreter

  . . . and interpreter can be written by Abstract Programming

- Similar to interpreter/compiler of Procedural Programming Languages

- . . . basic idea behind LISP

- Need more. . .

LAP Interpreter Version 2

eval[e] evaluates its argument `e`

Case 1: if atom[e] $\Longrightarrow$ e

`e = x`

Otherwise, let `fn = car[e], args = cdr[e]`

Case 2: `fn` is list $\Longrightarrow$ apply[fn, args]

`e = ( (lambda (y) y) a )`

Case 3: `fn` is atom (symbolic name)

(a) `fn` $\neq$ `lambda`, built-in or special

`e = (x y)`
`e = (x ((lambda (y) y) a))`

$\Longrightarrow$ `cons[fn, map[eval, args]]`

(b) `fn = lambda`

`e = (lambda (f) ((lambda (y) y) a))`

$\Longrightarrow$ `list[ lambda, car[args],`
`eval[cadr[args]] ]`

(c) `fn` is built-in function

`e = (+ 3 4)`

$\Longrightarrow$ apply `fn` to `eval` of its args

(d1) `fn` is special form: `quote`

`e = (quote (a b))`

$\Longrightarrow$ car[args]

(d2) `fn` is special form: `if`

`e = (if (eq n 0) 1 (fact (- n 1)))`

$\Longrightarrow$
```
if eval[car[args]] = T
  then eval[cadr[args]]
  else eval[caddr[args]]
```

Otherwise, let

```
ids = car[args],
vals = cadr[args]
body = caddr[args]
```

if `fn = let` $\Longrightarrow$ `eval(subs2(vals, ids, body))`

"`subs2(v,n,e)`" substitutes v's for n's in `e`, simultaneously

if `fn = letrec` $\Longrightarrow$

```
apply[ list(lambda, ids, body),
   list(Y, list(lambda, ids, vals))]
```

where

```
Y = (lambda (y) ((lambda (x) y(xx))
         (lambda (x) y(xx))))
```

Full Example:

eval[((lambda (x y) (cons (car x) (cdr y)))
(quote (1 2)) (quote (3 4)))]

e1 = ((lambda (x y) (cons (car x) (cdr y)))
(quote (1 2)) (quote (3 4)))

1. atom[e1] = false

So, fn1 = car[e1] = (lambda (x y) (cons (car x) (cdr y)))
args1 = cdr[e1] = ((quote (1 2)) (quote (3 4)))

2. atom[fn1] = false

So...

apply[fn1, args1]
$\Longrightarrow$ eval[subs2[args1, cadr[fn1], caddr[fn1]]]

subs2[args1, cadr[fn1], caddr[fn1]] =
subs2[v2, n2, e2]

where
v2 = ((quote (1 2)) (quote (3 4)))
n2 = (x y)
e2 = (cons (car x) (cdr y))

... $\Longrightarrow$ eval[e3]

where e3 =(cons (car (quote (1 2))) (cdr (quote (3 4))))

1. atom[e3] = false

So...

2. fn3 = car[e3] = cons
args3 = cdr[e3] = ((car (quote (1 2))) (cdr (quote (3 4))))

atom[fn3] = atom[cons] = true

Is fn3 built-in?   Yes

So...

$\Longrightarrow$ apply-builtin[fn3, args3]
$\Longrightarrow$ cons[eval[(car (quote (1 2)))],
eval[(cdr (quote (3 4)))]]

eval[(car (quote (1 2)))] $\Longrightarrow$ car[eval[(quote (1 2))]]

e4 = (quote (1 2))

1. atom[e4] = false

So...

fn4 = car[e4] = quote args4 = cdr(e4) = ((1 2))

3. atom[fn4] = atom[quote] = true

fn4 is special "quote"

... $\Longrightarrow$ car[args4] = (1 2) $\Longrightarrow$ car[(1 2)] = 1

eval[(cdr (quote (3 4)))] ... $\Longrightarrow$ (4)

so... $\Longrightarrow$ cons[1, (4)] = (1 4)