$\boxed{\text{Abstract Programming}}$

- Lambda Calculus is difficult to use
  (notation, "()"s, . . .)

- **Abstract Programming**
  + more human-oriented syntax,
  + semantics precisely defined
  . . . in terms of λ- calculus

- Semantically: each symbol simply stands for equivalent λ-calculus definition:

  1. Repr'n of integers, booleans, and lists
     $1 \equiv (\lambda\, \mathtt{sz} \mid \mathtt{sz})$
     $2 \equiv (\lambda\, \mathtt{sz} \mid \mathtt{s(sz)})$
     $\top \equiv (\lambda\, \mathtt{xy} \mid \mathtt{x})$
  2. Primitive arithmetic functions
     add $\equiv (\lambda\, \mathtt{wzyx} \mid \mathtt{wy(zyx)})$
  3. Logic connectives
     and $\equiv (\lambda\, \mathtt{wz} \mid \mathtt{wz\ F})$
  4. Infix notation
     `1 + 2`

$\boxed{\texttt{let and where}}$

Given: $\texttt{inc} = \text{"}{+1}\text{"} = ((\lambda\,\texttt{wzyx} \mid \texttt{wy(zyx)})\ \texttt{1})$,

$\mapsto (\lambda\,\texttt{zyx} \mid \texttt{y(zyx)})$

$(\texttt{inc 2}) = (\lambda\,\texttt{zyx} \mid \texttt{y(zyx)})\ (\lambda\,\texttt{sz} \mid \texttt{s(sz)})$

$\mapsto (\lambda\,\texttt{yx} \mid \texttt{y(}\ (\lambda\,\underline{\texttt{s}}\overline{\texttt{z}} \mid \texttt{s(sz)})\ \underline{\texttt{y}}\overline{\texttt{x}}\texttt{)}\ )$

$\mapsto (\lambda\,\texttt{yx} \mid \texttt{y(y(y(x))))} \equiv 3$

- Can replace λ-calculus def'n by

$\boxed{\texttt{let inc = }(\lambda\,\texttt{x} \mid \texttt{x + 1})\texttt{ in (inc 2)}}$

$\boxed{\texttt{let inc(x) = x + 1 in inc(2)}}$

$\boxed{\texttt{inc(2) where inc(x) = x + 1}}$

In gen'l "`let f = A in E`"

has semantics: "$(\lambda\,\texttt{f} \mid \texttt{E})\ \texttt{A}$"

Similarly, "`let f(x) = B in E`"

has semantics: "$(\lambda\,\texttt{f} \mid \texttt{E})\ (\lambda\,\texttt{x} \mid \texttt{B})$"

Eg: `let inc =` $(\lambda\,\texttt{x} \mid \texttt{x + 1})$ `in (inc 2)`

or

`let inc(x) = (x + 1) in (inc 2)`

means

$(\lambda\,\texttt{inc} \mid \texttt{(inc 2)})(\lambda\,\texttt{x} \mid \texttt{x + 1}) \mapsto$

$((\lambda\,\texttt{x} \mid \texttt{x + 1})\ \texttt{2}) \mapsto \texttt{2 + 1} \mapsto 3$

Let Construction

- `let` construct $\equiv$ syntactic variation,
  to express function application

- Nested definitions are ok;
  ...correspond to nested lambda def'ns

Eg: `let x = 2 in let y = x*x in y+1`

$\equiv$

`(`λ`x | (`λ`y | y+1) x*x) 2`

Applicative order reduction

$\mapsto$ `let x = 2 in (x*x)+1` $\mapsto$ `(`λ`x | (x*x)+1) 2`

$\mapsto$ `(2*2)+1` $\mapsto$ `5`

Normal order reduction

$\mapsto$ `let y = 2*2 in y+1` $\mapsto$ `(`λ`y | y+1) 2*2`

$\mapsto$ `(2*2)+1` $\mapsto$ `5`

Name Conflicts

- Handle *name conflicts*. . .
  as in λ-calculus

Eg: `let x = 2 in let x = x*x in x+1`

$\equiv$

$(\lambda\,$`x | `$(\lambda\,$`x | x+1) x*x) 2`

$\mapsto$ `let x = 2*2 in x+1` $\mapsto$ $(\lambda\,$`x | x+1) (2*2)`

$\mapsto$ `(2*2)+1` $\mapsto$ `5`

letrec and whererec

- Can handle recursive functions:

```
let Rsum = (λf n | if (zeroP n) then 0 else (n + f (n-1)))
    in  ((Y Rsum) 3)
```

This expands to. . .

```
...= Rsum (Y Rsum) 3 =

(λf n | if (zeroP n) then 0 else (n + f (n-1)))
   (Y Rsum) 3  ↦
if (zeroP 3) then 0 else (3 + (Y Rsum) (3-1))  ↦
3 + ((Y Rsum) 2)  ↦ 3 + (Rsum (Y Rsum) 2)  ↦
3 + (λf n | if (zeroP n) then 0 else (n + f (n-1)))
   (Y Rsum) 2  ↦
3 + (if (zeroP 2) then 0 else (2 + (Y Rsum) (2-1)))
↦
3 + (2 + (Y Rsum) 1)  ↦ 3+2 + (Rsum (Y Rsum) 1)
↦
3+2 + (λf n | if (zeroP n) then 0 else (n + f (n-1)))
   (Y Rsum) 1  ↦
```

```
3+2 + (if (zeroP 1) then 0 else (1 + (Y Rsum) (1-1)))
↦
3+2 + 1 + ((Y Rsum) 0)  ↦ (3+2+1 + (Rsum (Y Rsum) 0))
↦
3+2+1 + (λf n | if (zeroP n) then 0 else (n + f (n-1)))
   (Y Rsum) 0  ↦
3+2+1+ (if (zeroP 0) then 0 else (0 + (Y Rsum) (0-1)))
↦ 3+(2+(1+0))  ↦ 6
```

“LetRec” Form

- Replace combinator lambda expression by self-referencing expression:

```
letrec sum(n) = if (zeroP n) then 0
      else n + sum(n-1)
    in  sum(3)
```

or

```
sum(3) wherec
sum(n) = if (zeroP n) then 0 else n + sum(n-1)
```

since

```
letrec sum(n) = if (zeroP n) then 0 else n + sum(n-1)
in sum(3)
↦ letrec sum(n) = if (zeroP n) then 0 else n +
sum(n-1) in 3+sum(2)
↦ letrec sum(n) = if (zeroP n) then 0 else n +
sum(n-1) in 3+2+sum(1)
↦ letrec sum(n) = if (zeroP n) then 0 else n +
sum(n-1) in 3+2+1+sum(0)
```

$\mapsto$ 3+2+1+0 $\mapsto$ 6

General LetRec

In gen'l: `letrec f(x) = B in f(A)`

has semantics: `(Y (λf x | B)) A`

Eg: `letrec sum(n) = if (zeroP n) then 0 else n + sum(n-1)`

`in sum(3)`

means

`(Y (λsum n | if (zeroP n) then 0 else n + sum(n-1))) 3`

- Note: Just like "`label`" !

Higher Order Functions

Def'n: A *higher order function* is a function
that has (at least one of):
1. ≥ 1 functions as arguments
2. A function as a result

- Lambda expr's are higher order functions, as all args and results are functions

Eg: Define function whose argument is function `f`,
which returns function
that takes value `x`
and returns result of applying `f` to `x` twice

```
twice = (λf | λx | f(f x))
```

Now use function `inc(y)` that returns `y+1`:

```
inc = (λy | y + 1)
```

```
twice( inc ) = (λf | (λx | f(f x))) (λy | y + 1)
↦ (λx | (λy | y + 1)((λy | y + 1) x))
↦ (λx | ((λy | y + 1) x) + 1)
↦ (λx | (x + 1) + 1)
```

... ≡ function (of `x`) that applies `inc` to `x` twice.

HigherOrder Function wrt Lists

- $\exists$ 3 general classes of
  higher order functions wrt lists:
  mapping, reduction, filtering

- Mapping

`map( unary function, list )` $\longrightarrow$ `(same size) list`

Eg: Increment all elements of list: `(15 24 17)`
Given function

```
   inc = (λy | y + 1)
incAll(L) = if null(L) then nil
   else cons(inc(car(L)), incAll(cdr(L)))
```

```
incAll((15 24 17))  ↦

cons(inc(car((15 24 17))), incAll(cdr((15 24 17))))  ↦

cons(inc(15)), incAll(( 24 17)))  ↦

cons(16, cons(inc(car((24 17))), incAll(cdr((24 17)))))

↦  cons(16, cons(inc(24), incAll((17))))  ↦

cons(16, cons(25, cons(inc(car((17))), incAll(cdr((17))))))

↦ cons(16, cons(25, cons(inc(17), incAll(nil))))  ↦

cons(16, cons(25, cons(18, nil)))  ↦

cons(16, cons(25, (18)))  ↦

cons(16, (25 18))  ↦ (16 25 18)
```

Another Example

Payroll represented by a list of dotted pairs:
(employee-name . employee-salary):
`((Peter . 23) (Mary . 54) ...)`

Want to raise salary of every employee by $1 K

```
raiseOne(Pair) = cons( car(Pair), cdr(Pair) + 1)
 (λp z | cons( first(p), rest(p)+1 ))
 ≡ (λp z | (λm n z | z m n)
     (λx | x T) p  (+ 1 (λx | x F) p) )
 ≡ (λp z | (λm n z | z m n)
     (p T)  (λzyx | y(zyx)) (p F))
 ≡ (λp z | (z (p T) (λyx | y((p F)yx))) )


raiseOne( (A . 3) )
 ≡ (λp z | (z (p T) (λyx | y((p F)yx))) )
       (λw | w A 3)
  ↦ (λz | (z ((λw | w A 3) T)
       (λyx | y(( (λw | w A 3) F)yx))) )
  ↦ (λz | (z (T A 3) (λyx | y( (F A 3) yx))) )
  ↦ (λz | (z A (λyx | y(3yx))))
  ↦ (λz | (z A (λyx | y( (λsx|s(s(s(x))) )yx))))
  ↦ (λz | (z A (λyx | y( y(y(y(x))) ) ))
  ↦ (λz | (z A 4) )
 ≡ [A, 4]
```

```
raiseAll(L) = if null(L) then nil
   else cons(raiseOne(car(L)), raiseAll(cdr(L)))

raiseAll( ((Peter .  23) (Mary .  54)) )
    ↦ ((Peter .  24) (Mary .  55))
```

Mapping Function

Note: `incAll`, `raiseAll` are similar,
even though `inc` and `raiseOne` are different

⇒ can write a function with args
   * a function (such as `inc`, `raiseOne`)
   * a list

that applies function to each element in list

```
map(f,L) = if null(L) then nil
   else cons( f(car(L)), map(f, cdr(L)))
```

Eg: `map( raiseOne, ((John . 23) (Mary . 54)) )` $\mapsto$

```
cons(raiseOne(car(((John . 23) (Mary . 54)))),
   map(raiseOne, cdr(((John . 23) (Mary . 54)))))
```
$\mapsto$

```
cons(raiseOne((John . 23)), map(raiseOne, ((Mary
. 54))))
```
$\mapsto$

```
cons((John . 24), map(raiseOne, ((Mary . 54))))
```
$\mapsto$

```
cons((John . 24), cons(raiseOne(car(((Mary . 54)))),
```

```
map(raiseOne, cdr(((Mary . 54)))))) ↦
cons((John . 24), cons(raiseOne((Mary . 54)),
map(raiseOne, nil))) ↦
cons((John . 24), cons((Mary . 55)), map(raiseOne,
nil))) ↦
cons((John . 24), cons((Mary . 55)), nil)) ↦
cons((John . 24), ((Mary . 55)))
≡ ((John . 24) (Mary . 55))
```

Unnamed Function

- If function only needed for one application
  just use lambda definition
  (ie, no need for name)

```
map( (λn | cons(car(n), cdr(n) + 1)),
   ((John . 23) (Mary . 54)))
```

- In Lisp, called `mapcar`
  as it uses the `car` of each element.

Eg:

```
→ (mapcar '(lambda (x) (+ x 1)) '(1 2 3 4 5))
(2 3 4 5 6)
→ (defun inc (x) (+ x 1))
INC
→ (mapcar 'inc '(1 2 3 4 5))
(2 3 4 5 6)
→ (defun raiseOne (n)
     (cons (car n) (+ 1 (cdr n))))
raiseOne
→ (mapcar 'raiseOne '((John . 23) (Mary . 54)))
((JOHN . 24) (MARY . 55))
```

Reduction

- Given 2-ary function `g`
  its identity `id`
  a list `L = (A1...An)`
  compute:
  `(g A1 (g A2 ...  (g An id) ...  ))`

Eg: function `+`, identity `0`, list of numbers `L = (5 3 2)`

`(+ 5 (+ 3 (+ 2 0))) = 10`

```
→ (defun plusReduction (X)
        (if (null X) 0
            (+ (car X) (plusReduction (cdr X)))))
→ (plusReduction '(5 3 2))
13
```

Eg: function `*`, identity `1`, list of numbers `L = (5 3 2)`

`(* 5 (* 3 (* 2 1))) = 30`

```
→ (defun timesReduction (X)
    (if (null X) 1
      (* (car X) (timesReduction (cdr X)))))
→ (timesReduction '(5 3 2))
30
```

Reduction Function

Def'n: `reduce(f, id, L) = if null(L) then id`
`else f(car(L), reduce(f, id, cdr(L)))`

Eg: `reduce(+, 0, (5 3 2))` $\mapsto$

`+(car(5 3 2), reduce(+, 0, cdr(5 3 2)))` $\mapsto$

`+(5, reduce(+, 0, (3 2)))` $\mapsto$

`+(5, +(car(3 2), reduce(+, 0, cdr(3 2))))` $\mapsto$

`+(5, +(3, reduce(+, 0, (2))))` $\mapsto$

`+(5, +(3, +(car(2), reduce(+, 0, cdr(2)))))` $\mapsto$

`+(5, +(3, +(2, reduce(+, 0, ()))))` $\mapsto$

`+(5, +(3, +(2, 0)))` $\mapsto$ `..` $\mapsto$ `+(5, 5)` $\mapsto$ `10`

- In Lisp: identity not used...
  `(g A1 (g A2...(g An id)...))` *becomes:*
  `(g A1 (g A2...(g An-1 An)...))`

Filtering

- Given predicate function `g`, list `L = (A1...An)` compute:

  $(A_{i1}\ A_{i2}\ \ldots\ A_{ik})$
  where $g(A_{im})$ = `true` for all $m = 1..k$

Eg: predicate `minusp`, list of numbers `L = (5 -3 -2 3)`
return `(-3 -2)`

```
(defun minusFilter (X)
 (cond ((null X) nil)
   ((minusp (car X)) (cons (car X) (minusFilter
(cdr X))) )
   (t (minusFilter(cdr X)))) )

→ (minusFilter '(5 -3 -2 3))

(-3 -2)
```

In gen'l:
```
filter(f, L) = if null(L) then nil
    else if f(car(L))
      then cons(car(L), filter(f, cdr(L)))
      else filter(f, cdr(L))
```

Eg: `filter(minusp, (5 -3 -2 3))` $\mapsto$ `(-3 -2)`