Lab08 & hw08

NJU SICP22 TAs

Scheme

- Writing executable math equation,
- … in linked list.

Lab08p1: over or under

```
(define (over-or-under a b)
    -1 if a is less than b
    0 if a is equal to b
    1 if a is greater than b
)
```

Lab08p1: over or under

```
(define (over-or-under a b)
    -1 if (< a b)
    0 if (= a b)
    1 else
)
```

Lab08p1: over or under

```
(define (over-or-under a b)
    (cond ((< a b) -1)
      ((= a b) 0)
      (else 1))
)
```

Lab08p2: make adder

- “closure”
- Your already know this well in python.

Lab08p3: composed

- “High-order function”
- Your already know this well in python.

Lab08p4: gcd

```
def gcd(a, b):
    if b == 0:
        return a
    return gcd(b, a % b)
```

Lab08p4: gcd

```
def gcd(a, b):
    if (= b 0):
        a
    (gcd (remainder a b))

expression
```

Lab08p4: gcd

```
def gcd(a, b):
    (if (= b 0)
        a
    (gcd (remainder a b)))

control
```

Lab08p5: make a list

```
(define lst
  'YOUR-CODE-HERE  ; not a comment!
)
```

Lab08p5: ordered

```
Math: ordered(L)
If L is nil: return True
If L is a :: nil: return True
If L is a :: b :: L’:
  return a < b /\ ordered (b :: L’)
```

Lab08p5: ordered

```
Math2Code: ordered(L)
If (null? L): #t
If (null? (cdr L)): #t
Else:
  return ((car L) < (car (cdr L)) /\ (ordered (cdr L))

; (car L) is a, (car (cdr L)) is b
```

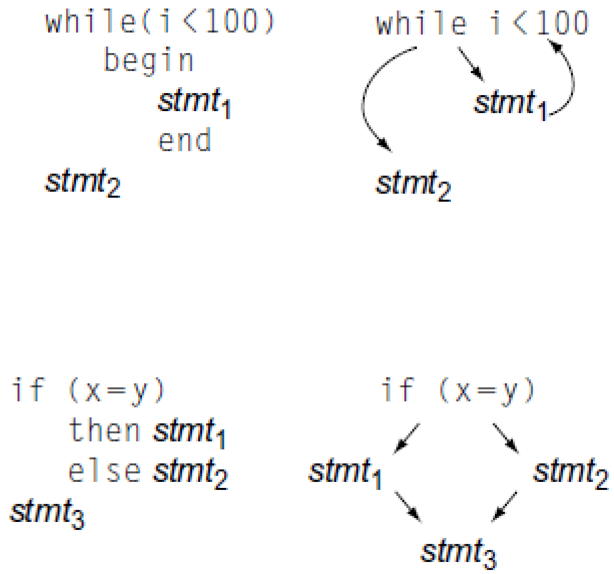
Lab08p5: ordered

```
Code: ordered(L)
(cond
  ((null? L) #t)
  ((null? (cdr L)) #t)
  (else
    ((car L) < (car (cdr L)) /\ (ordered (cdr L)))
)
; (car L) is a, (car (cdr L)) is b
```

Scheme

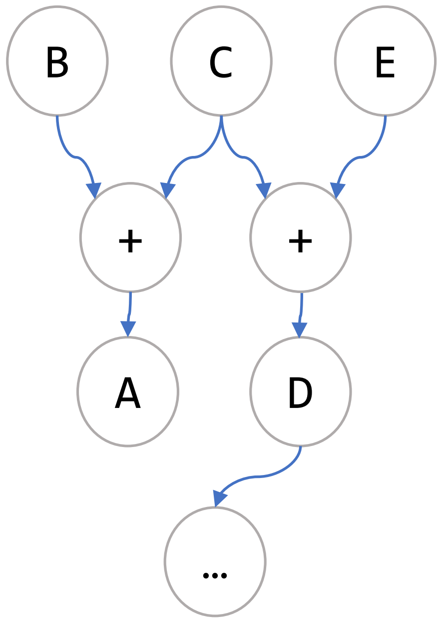
- Writing executable math equation,
- … in linked list.
- Functional & Imperative
- Dataflow & Control flow

```
...

A = B + C
D = C + E
return D
```



Control flow: how execution is performed?

```
...

A = B + C
D = C + E
return D
```



How data is mathematically related?

  “Data dependences”

Scheme

- Writing executable math equation,
- … in linked list.
- Functional & Imperative
- Dataflow & Control flow

- Functional programming is in somewhat “higher-level” than the imperative:
    - Let the compiler & machine decide the tedious control flow,
    - Human only needs to know the math.
        - And think the math.

Scheme

- Writing executable math equation,
- … in linked list.
- Functional & Imperative
- Dataflow & Control flow

You can visit last year's review for tygg's perspective.

- Functional programming is in somewhat "higher-level" than the imperative:
  - Let the compiler & machine decide the tedious control flow,
  - Human only needs to know the math.
    - And think the math.

Hw08p1: pow

```
Math: pow base exp =
If exp = 0: return 1
If exp % 2 = 0: return pow base (exp//2)
If exp % 2 = 1: return base * (pow base (exp//2))
```

$$x^{2y} = (x^y)^2$$

$$x^{2y+1} = x(x^y)^2$$

Hw08p1: pow

You definitely know how to do this!

```
Math: pow base exp =
If exp = 0: return 1
If exp % 2 = 0: return pow base (exp//2)
If exp % 2 = 1: return base * (pow base (exp//2))
```

$$x^{2y} = (x^y)^2$$

$$x^{2y+1} = x(x^y)^2$$

Hw08p2: filter-lst

```
Math: filter-list fn L
If L = nil: return nil
If L = a::L’:
    If (fn a): return a :: (filter-lst fn L’)
    else: return (filter-lst fn L’)
```

Hw08p3: no repeats

```
Math: no-repeats L
If L = nil: return nil
If L = a::L’:
    return a::(no-repeats (filter-lst (not eq a) L’))
```

Hw08p4: substitute

```
Math: substitute s old new
If s = nil: return s
If s = a::L’:   ;; a is not list
    If a = old: return new :: substitute L’ old new
    Else: return old :: substitute L’ old new
If s = L::L’:   ;; L is list
    return (substitute L old new) :: (substitute L’ old new)
```

Hw08p5: Sub All

```
Math: sub-all s olds news
If olds = nil: return s
If olds = old::olds’, news = new::news’:
    sub-all (substitute s old new) olds’ news’
```

Hw08p6: Tree in Scheme

- How to represent a tree with linked list?
- Node: (label, branches)

Hw08p7: Label Sum

```
Math: label-sum t
(label t) + (sum (map label-sum (branches t)))
```

Hw09p1: symbol differentiation, +

Math: $$\frac{\partial\big(f(x)+g(x)\big)}{\partial x} = \frac{\partial f(x)}{\partial x} + \frac{\partial g(x)}{\partial x}$$

```
(define (derive-sum expr var)
  (make-sum (derive (first-operand expr) var)
            (derive (second-operand expr) var)))
```

Hw09p1: symbol differentiation, *

Math: $$\frac{\partial\big(f(x)\times g(x)\big)}{\partial x} = \frac{\partial f(x)}{\partial x}\times g(x) + f(x)\times\frac{\partial g(x)}{\partial x}$$

```
(define (derive-product expr var)
  (make-sum
   (make-product (derive (first-operand expr) var)
                 (second-operand expr))
   (make-product (derive (second-operand expr) var)
                 (first-operand expr))))
```

Hw09p1: symbol differentiation, exp

Math:

$$\frac{\partial\left(f(x)^{g(x)}\right)}{\partial x} = g(x)\times f(x)^{g(x)-1}$$

```
(define (derive-exp exp var)
  (make-product (second-operand exp)
     (make-exp (first-operand exp)
         make-sum (second-operand exp) -1))))
```

Congratulation!

Now you know the *most* elegant language in the world ☺