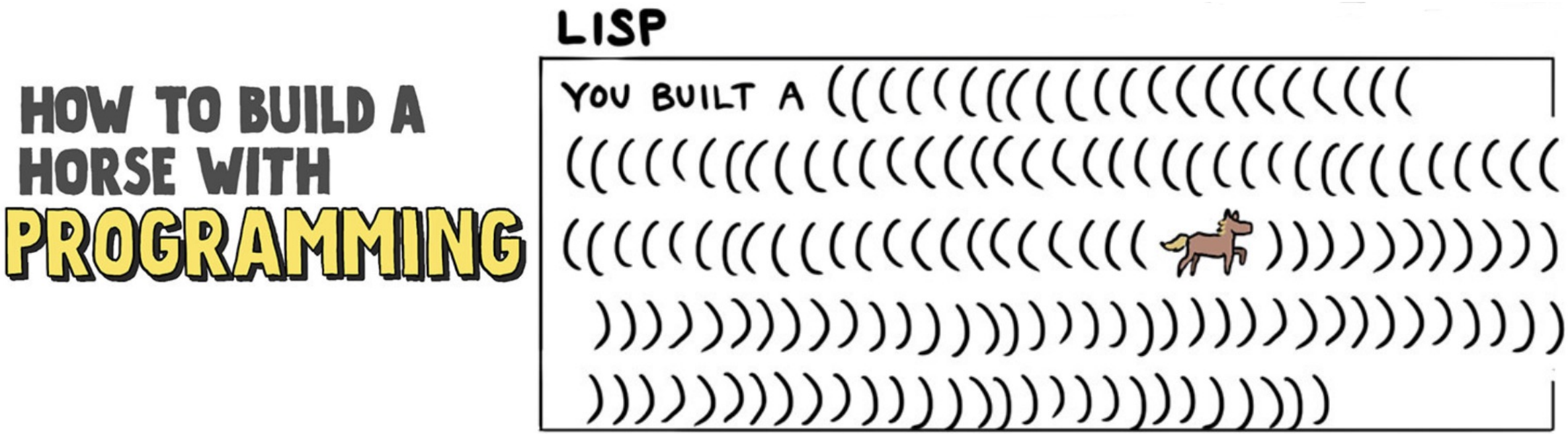
Review of Lab08 and Hw08



https://toggl.com/blog/build-horse-programming

Imperative versus Declarative

- Abstracting out control

Calculating Sum of a List

Imparative

```
total = 0
for i in range(len(lst)):
    total += lst[i]
```

Declaritive

```
sum(lst)
```

Imperative versus Declarative

- Abstracting out control

Calculating Filtered Elements of a List

Imparative

```
result = []
for i in range(len(lst)):
    if predicate(lst[i]):
        result.append(lst[i])
```

Declaritive

```
filter(predicate, lst)
```

Imperative versus Declarative

- Abstracting out control

  - Imperative:
    - Tell machines exactly what to do
    - Loops, conditions, returns, temporary variables
  - Declarative:
    - Declare what we want to accomplish
    - Functions, first-class functions, higher-order functions
    - Modularity and composability

  - Declarative programming is closer to how we think
  - Declarative programming is more understandable

Imperative versus Declarative

- Abstracting out control
- Immutability, no side effects

  - Declarative:
    - Avoid stateful interactions
    - Only input can affect output
    - No race conditions!
  - Imperative:
    - Too many bugs QAQ

  - Declarative programming is easier to reason about
  - Declarative programming is more understandable

Imperative versus Declarative

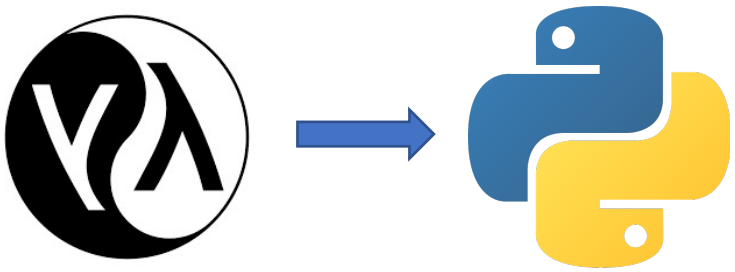
- Abstracting out control
- Immutability, no side effects

- Good reputation for academic
- Yet not commonly used
  - Hide implementation details
  - Extra cost to performance

Imperative versus Declarative

- Abstracting out control
- Immutability, no side effects

- Good reputation for academic
- Yet not commonly used, but learnt from



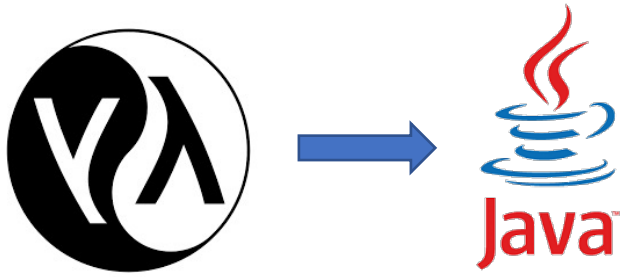
- Iterators, generators
- Lambdas, higher-order functions
- Built-in functions like map, reduce, filter, etc.

https://docs.python.org/3/howto/functional.html

Imperative versus Declarative

- Abstracting out control
- Immutability, no side effects

- Good reputation for academic
- Yet not commonly used, but learnt from

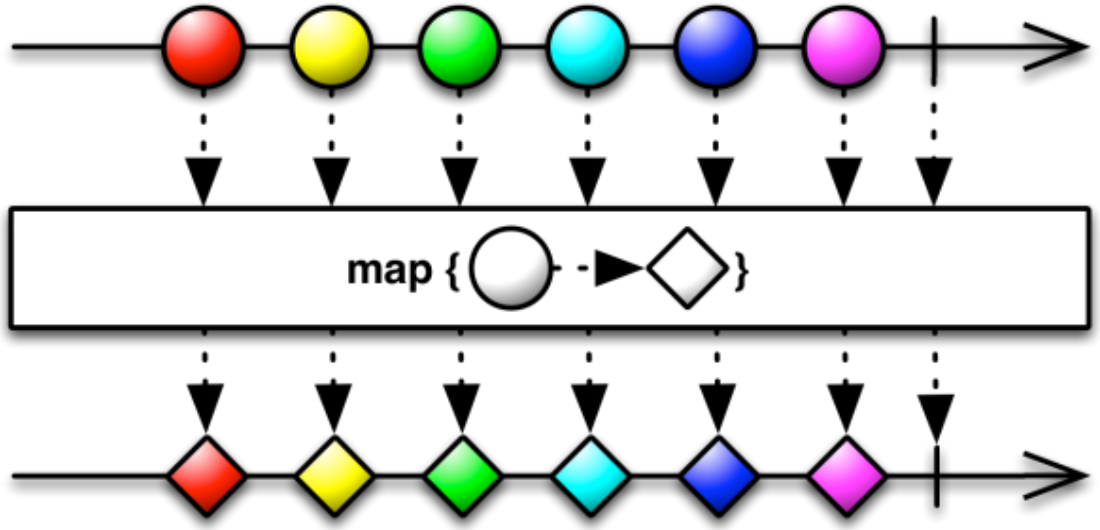


- Lambdas
- Stream API, map, reduce, etc.
- Future and completable future.

https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html

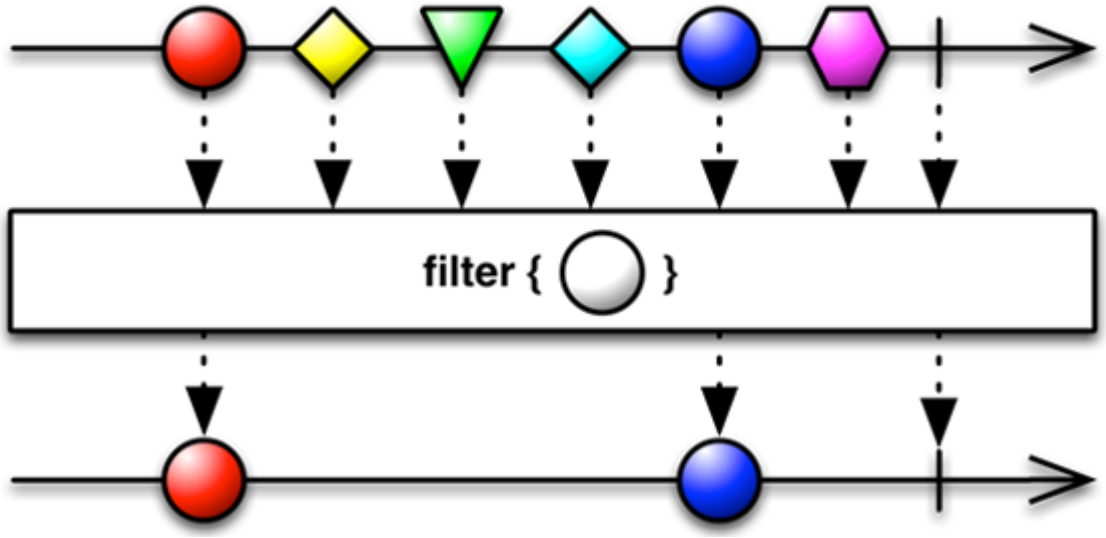
Useful Functions

- `map`



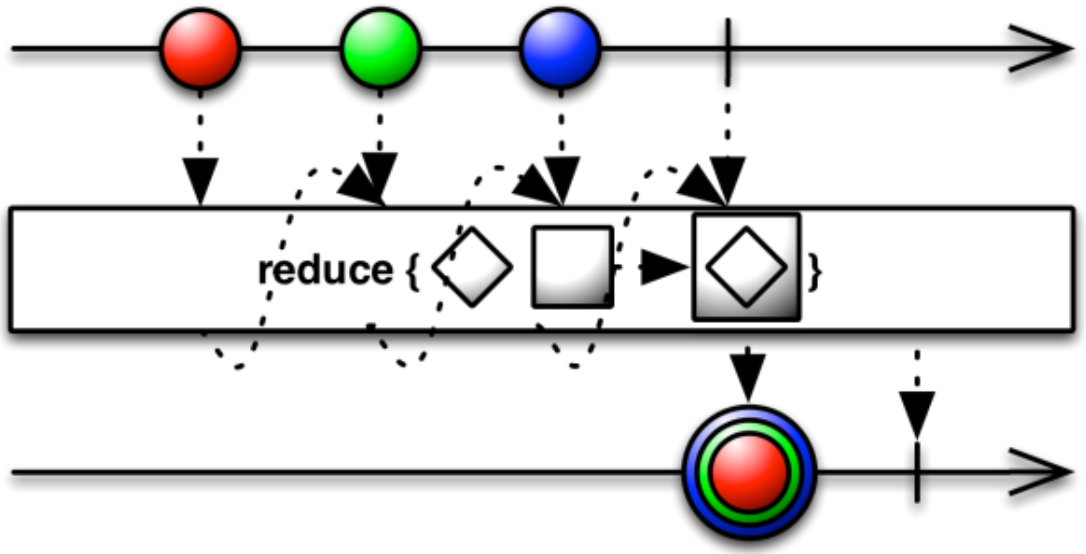
Useful Functions

- `map`
- `filter`



Useful Functions

- `map`
- `filter`
- `reduce`



lab08p1: over-or-under


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

lab08p1: over-or-under


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

lab08p2-3: lambdas


```
(define (make-adder n) (lambda (x) (+ n x)))

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

lab08p4: gcd


```
(define (gcd a b)
    (if (= b 0)
        a
        (gcd b (remainder a b))
    )
)
```

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

lab08p6: ordered


```
(define (ordered s)
    如果只有0个或者1个元素，则已经排序
    否则必须第1个元素<=第2个元素，并且后续部分是已经排序的
)
```

lab08p6: ordered


```
(define (ordered s)
    (if (只有0个或者1个元素)
        已经排序
        第1个元素<=第2个元素，并且后续部分是已经排序的
    )
)
```

lab08p6: ordered


```
(define (ordered s)
    (if (or (只有0个元素)(只有1个元素))
        已经排序
        第1个元素<=第2个元素，并且后续部分是已经排序的
    )
)
```

lab08p6: ordered


```
(define (ordered s)
    (if (or (null? s) (只有1个元素))
        已经排序
        第1个元素<=第2个元素，并且后续部分是已经排序的
    )
)
```

lab08p6: ordered


```
(define (ordered s)
    (if (or (null? s) (null? (cdr s)))
        已经排序
        第1个元素<=第2个元素，并且后续部分是已经排序的
    )
)
```

lab08p6: ordered


```
(define (ordered s)
    (if (or (null? s) (null? (cdr s)))
        #t
        第1个元素<=第2个元素，并且后续部分是已经排序的
    )
)
```

lab08p6: ordered


```
(define (ordered s)
    (if (or (null? s) (null? (cdr s)))
        #t
        (and
            第1个元素<=第2个元素
            后续部分是已经排序的
        )
    )
)
```

lab08p6: ordered


```
(define (ordered s)
    (if (or (null? s) (null? (cdr s)))
        #t
        (and
            (not (> (第一个元素) (第二个元素)))
            后续部分是已经排序的
        )
    )
)
```

lab08p6: ordered


```
(define (ordered s)
    (if (or (null? s) (null? (cdr s)))
        #t
        (and
            (not (> (car s) (car (cdr s))))
            后续部分是已经排序的
        )
    )
)
```

lab08p6: ordered


```
(define (ordered s)
    (if (or (null? s) (null? (cdr s)))
        #t
        (and
            (not (> (car s) (car (cdr s))))
            (ordered (后续部分))
        )
    )
)
```

lab08p6: ordered


```
(define (ordered s)
    (if (or (null? s) (null? (cdr s)))
        #t
        (and
            (not (> (car s) (car (cdr s))))
            (ordered (cdr s))
        )
    )
)
```

hw08p1: pow

```
(define (pow base exp)
    如果指数为0，结果为1
    如果指数是偶数，返回x的y次方的平方
    如果指数是奇数，返回x乘以x的y次方的平方
)
```

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

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

hw08p1: pow

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

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

```
(define (pow base exp)
    (if (= exp 0)
        1
        (
            如果指数是偶数，返回x的y次方的平方
            如果指数是奇数，返回x乘以x的y次方的平方
        )
    )
)
```

hw08p1: pow

```
(define (pow base exp)
    (if (= exp 0)
        1
        (if (even? exp)
            (x的y次方的平方)
            (x乘以x的y次方的平方)
        )
    )
)
```

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

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

hw08p1: pow

```
(define (pow base exp)
    (if (= exp 0)
        1
        (if (even? exp)
            (x的y次方的平方)
            (* base (x的y次方的平方))
        )
    )
)
```

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

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

hw08p1: pow

```
(define (pow base exp)
    (if (= exp 0)
        1
        (if (even? exp)
            (square (x的y次方))
            (* base (square (x的y次方)))
        )
    )
)
```

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

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

hw08p1: pow

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

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

```
(define (pow base exp)
    (if (= exp 0)
        1
        (if (even? exp)
            (square (pow base (quotient exp 2)))
            (* base (square (pow base (quotient exp 2)))))
        )
    )
)
```

hw08p1: pow

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

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

```
(define (pow base exp)
    (if (= exp 0)
        1
        (let ((xe2y (square (pow base (quotient exp 2)))))
            (if (even? exp)
                xe2y
                (* base xe2y)
            )
        )
    )
)
```

hw08p2: filter-lst


```
(define (filter-lst fn lst)
    如果列表为空，返回空列表
    否则先筛选第一个元素，然后筛选剩下的列表
)
```

hw08p2: filter-lst


```
(define (filter-lst fn lst)
    (if (null? lst)
        nil
        先筛选第一个元素，然后筛选剩下的列表
    )
)
```

hw08p2: filter-lst


```
(define (filter-lst fn lst)
    (if (null? lst)
        nil
        (if (fn (car lst))
            保留第一个元素，筛选剩下的列表
            删除第一个元素，筛选剩下的列表
        )
    )
)
```

hw08p2: filter-lst


```
(define (filter-lst fn lst)
    (if (null? lst)
        nil
        (if (fn (car lst))
            (cons (car lst) (筛选剩下的列表))
            (筛选剩下的列表)
        )
    )
)
```

hw08p2: filter-lst


```
(define (filter-lst fn lst)
    (if (null? lst)
        nil
        (if (fn (car lst))
            (cons (car lst) (filter-lst fn (cdr lst)))
            (filter-lst fn (cdr lst))
        )
    )
)
```

hw08p3: no-repeats


```
(define (no-repeats s)
      如果列表为空，返回空列表
      否则把第一个元素从后续列表中删除，然后继续处理剩下的内容
)
```

hw08p3: no-repeats


```
(define (no-repeats s)
    (if (null? s)
        nil
        (把第一个元素从后续列表中删除，然后继续处理剩下的内容)
    )
)
```

hw08p3: no-repeats


```
(define (no-repeats s)
    (if (null? s)
        nil
        (cons
            (car s)
            (继续处理从后续列表中删除第一个元素后剩下的内容)
        )
    )
)
```

hw08p3: no-repeats


```
(define (no-repeats s)
    (if (null? s)
        nil
        (cons
            (car s)
            (no-repeats
                从后续列表中删除第一个元素后剩下的内容
            )
        )
    )
)
```

hw08p3: no-repeats


```
(define (no-repeats s)
    (if (null? s)
        nil
        (cons
            (car s)
            (no-repeats
                (filter-lst
                    (条件：不等于s的第一个元素)
                    (输入：列表s的剩余内容)
                )
            )
        )
    )
)
```

hw08p3: no-repeats


```
(define (no-repeats s)
    (if (null? s)
        nil
        (cons
            (car s)
            (no-repeats
                (filter-lst
                    (lambda (x) (not (eq? x (s的第一个元素))))
                    (输入：列表s的剩余内容)
                )
            )
        )
    )
)
```

hw08p3: no-repeats


```
(define (no-repeats s)
    (if (null? s)
        nil
        (cons
            (car s)
            (no-repeats
                (filter-lst
                    (lambda (x) (not (eq? x (car s))))
                    (cdr s)
                )
            )
        )
    )
)
```

hw08p7: label-sum


```
(define (label-sum t)
    如果t是叶子，则结果为t的标签
    否则求t的标签加上所有分支的结果之和
)
```

hw08p7: label-sum


```
(define (label-sum t)
    (if (t是叶子)
        (t的标签)
        (t的标签加上所有分支的结果之和)
    )
)
```

hw08p7: label-sum


```
(define (label-sum t)
    (if (is-leaf t)
        (t的标签)
        (t的标签加上所有分支的结果之和)
    )
)
```

hw08p7: label-sum

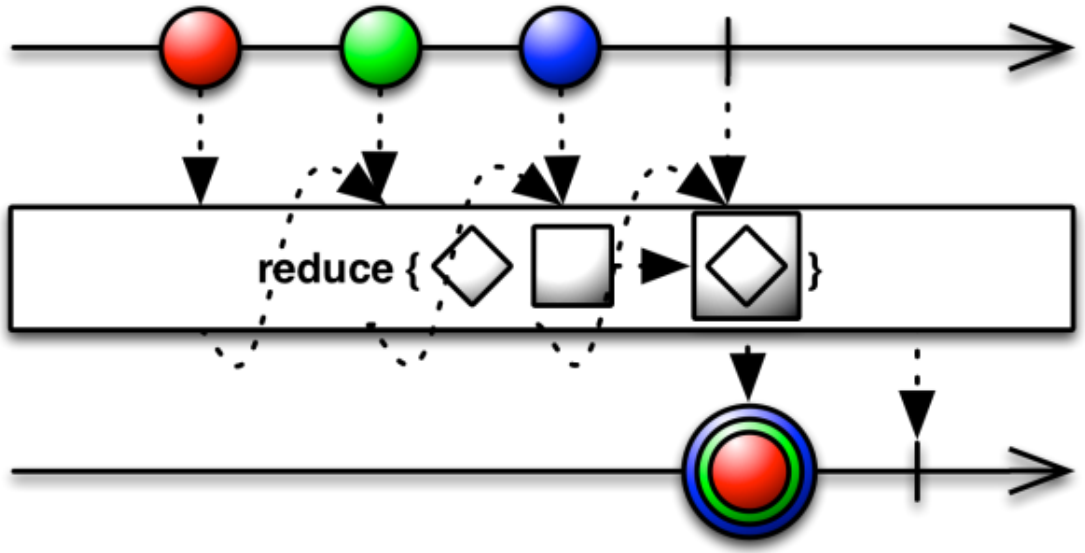

```
(define (label-sum t)
    (if (is-leaf t)
        (label t)
        (t的标签加上所有分支的结果之和)
    )
)
```

hw08p7: label-sum


```
(define (label-sum t)
    (if (is-leaf t)
        (label t)
        (+
            (t的标签)
            (所有分支的结果之和)
        )
    )
)
```

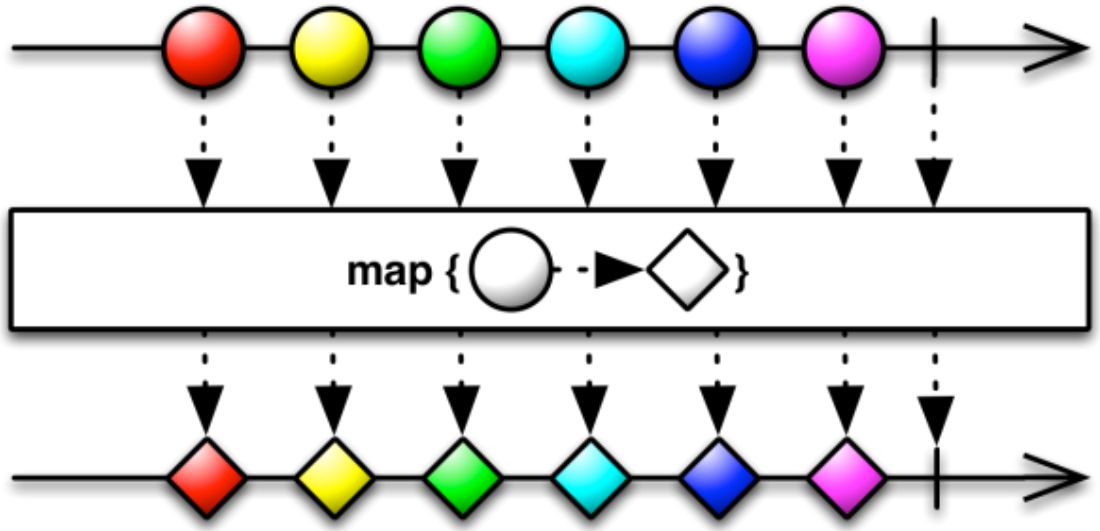
hw08p7: label-sum


```
(define (label-sum t)
    (if (is-leaf t)
        (label t)
        (+
            (label t)
            (所有分支的结果之和)
        )
    )
)
```



hw08p7: label-sum


```
(define (label-sum t)
    (if (is-leaf t)
        (label t)
        (+
            (label t)
            (reduce + (所有分支的结果))
        )
    )
)
```



hw08p7: label-sum


```
(define (label-sum t)
    (if (is-leaf t)
        (label t)
        (+
            (label t)
            (reduce + (map label-sum (t的分支)))
        )
    )
)
```

hw08p7: label-sum


```
(define (label-sum t)
    (if (is-leaf t)
        (label t)
        (+
            (label t)
            (reduce + (map label-sum (branches t))))
        )
    )
)
```

hw08p8: derive


```
(define (derive expr var)
  (cond ((number? expr) 0)
        ((variable? expr) (if (same-variable? expr var) 1 0))
        ((sum? expr) (derive-sum expr var))
        ((product? expr) (derive-product expr var))
        ((exp? expr) (derive-exp expr var))
        (else 'Error)))
```

$$\frac{\partial C}{\partial x} = 0 \qquad \frac{\partial x}{\partial x} = 1 \qquad \frac{\partial y}{\partial x} = 0 \qquad \frac{\partial \big(f(x) + g(x)\big)}{\partial x} = \ ?$$

hw08p8: derive

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

```
(define (derive-sum expr var)
      (∂f(x)/∂x + ∂g(x)/∂x)
)
```

hw08p8: derive

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

```
(define (derive-sum expr var)
    (make-sum
        (∂f(x)/∂x)
        (∂g(x)/∂x)
    )
)
```

hw08p8: derive

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

```
(define (derive-sum expr var)
    (make-sum
        (derive (f(x)) var)
        (derive (g(x)) var)
    )
)
```

hw08p8: derive

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

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

hw08p8: derive

$$\frac{\partial\left(f(x)\times g(x)\right)}{\partial x}=\frac{\partial f(x)}{\partial x}\times g(x)+f(x)\times\frac{\partial g(x)}{\partial x}$$

```
(define (derive-product expr var)
      (∂f(x)/∂x ×g(x) + f(x)× ∂g(x)/∂x)
)
```

hw08p8: derive

$$\frac{\partial\left(f(x)\times g(x)\right)}{\partial x}=\frac{\partial f(x)}{\partial x}\times g(x)+f(x)\times\frac{\partial g(x)}{\partial x}$$

```
(define (derive-product expr var)
    (make-sum
        (∂f(x)/∂x ×g(x))
        (f(x)× ∂g(x)/∂x)
    )
)
```

hw08p8: derive

$$\frac{\partial\left(f(x)\times g(x)\right)}{\partial x} = \frac{\partial f(x)}{\partial x}\times g(x) + f(x)\times\frac{\partial g(x)}{\partial x}$$

```
(define (derive-product expr var)
    (make-sum
        (make-product (∂f(x)/∂x) (g(x)))
        (make-product (f(x)) (∂g(x)/∂x))
    )
)
```

hw08p8: derive

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

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

hw08p8: derive

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

```
(define (derive-exp exp var)
    (g(x)×f(x)^(g(x)-1))
)
```

hw08p8: derive

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

```
(define (derive-exp exp var)
    (make-product
        (g(x))
        (f(x)^g(x)-1)
    )
)
```

hw08p8: derive

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

```
(define (derive-exp exp var)
    (make-product
        (second-operand exp)
        (f(x)^(g(x)-1))
    )
)
```

hw08p8: derive

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

```
(define (derive-exp exp var)
    (make-product
        (second-operand exp)
        (make-exp
            (f(x))
            (g(x) - 1)
        )
    )
)
```

hw08p8: derive

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

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

hw08p8: derive

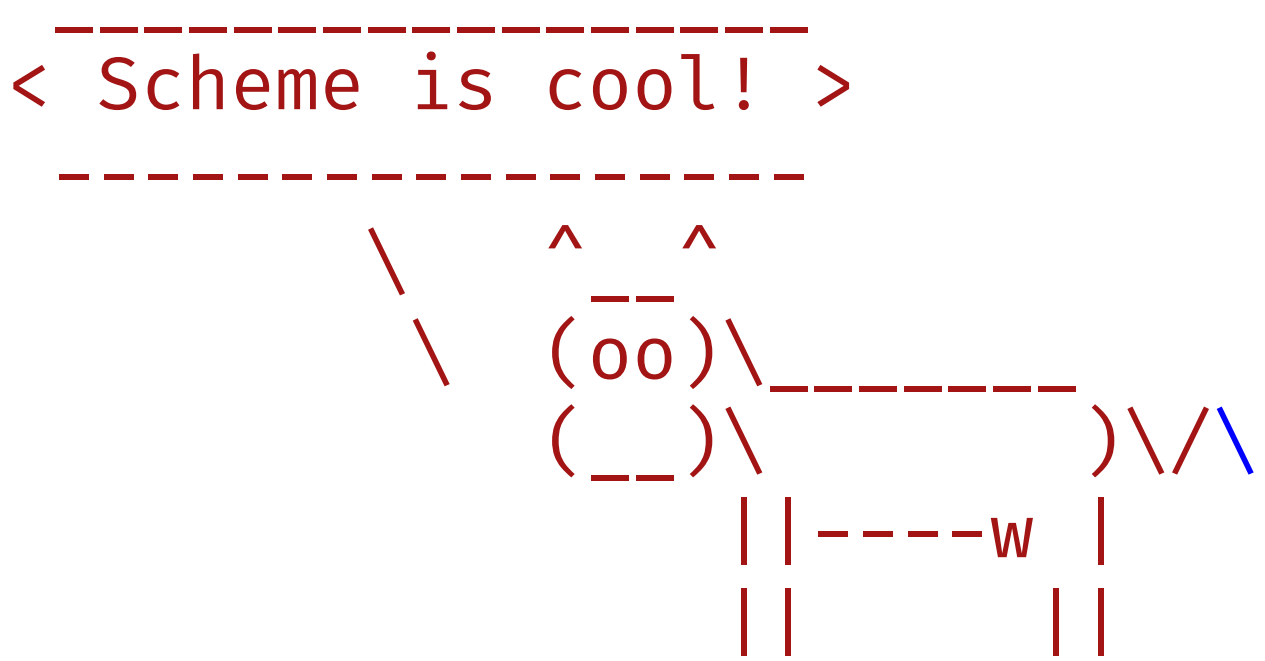
$$\frac{\partial\left(f(x)^{g(x)}\right)}{\partial x} = g(x) \times f(x)^{g(x)-1}$$

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

hw08p8: derive

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

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


Scheme is cool!