Programming in MUA

Weng Kai

- word <word> <word|number|bool>：将两个word合并为一个word，第二个值可以是word、number或bool
- sentence <value1> <value2>：将value1和value2合并成一个表，两个值的元素并列，value1的在value2的前面
- list <value1> <value2>：将两个值合并为一个表，如果值为表，则不打开这个表
- join <list> <value>：将value作为list的最后一个元素加入到list中（如果value是表，则整个value成为表的最后一个元素）
- first <word|list>：返回word的第一个字符，或list的第一个元素
- last <word|list>：返回word的最后一个字符，list的最后一个元素
- butfirst <word|list>：返回除第一个元素外剩下的表，或除第一个字符外剩下的字
- butlast <word|list>：返回除最后一个元素外剩下的表，或除最后一个字符外剩下的字

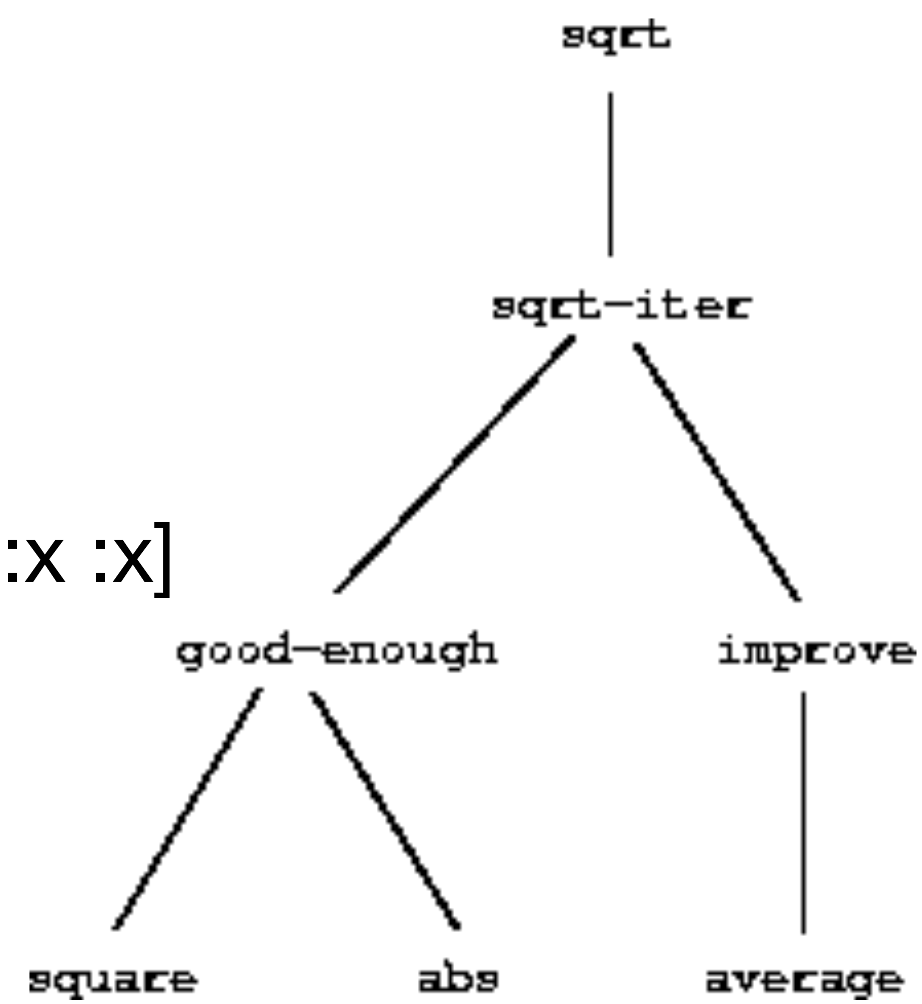
Square Roots by Newton's Method

$$\sqrt{x} \quad = \quad \text{the } y \text{ such that } y \geq 0 \text{ and } y^2 = x.$$

guess	Quotient	Average
1	2/1->2	(1+2)/2->1.5
1.5	2/1.5->1.33	(1.5+1.33)/2->1.4167
1.4167	2/1.4167->1.4118	(1.4167+1.4118)/2->1.4142

```
make "sqrt_it [
    [ x guess ]
    [
        if closeenough :x sq :guess
            [ return :guess ]
            [ return sqrt_it :x improve :guess :x ]
    ]
]
```

```
make "average [[a b] [
    return ((:a+:b)/2)
]]
make "sqrt [[x] [
    make "good_enough [[guess x] [
        return (abs (:guess * :guess - :x) < 0.001)
    ]]
    make "improve [[guess x] [
        return average :guess :x/:guess
    ]]
    make "sqrt_iter [[guess x] [
        if good_enough :guess :x
            [return :guess]
            [return sqrt_iter improve :guess :x :x]
    ]]
    return sqrt_iter 1.0 :x
]]
```



```
make "sq [
     [ x ]
     [ return mul :x :x ]
]
make "abs [
     [ x ]
     [ if lt :x 0
          [ return sub 0 :x ]
          [ return :x ]
     ]
]
make "closeenough [
     [ a b ]
     [ return lt abs sub :a :b 0.001 ]
]
make "improve [
     [ guess x ]
     [ return div add :guess div :x :guess 2 ]
]
make "sqrt_it [
     [ x guess ]
     [
          if closeenough :x sq :guess
               [ return :guess ]
               [ return sqrt_it :x improve :guess :x ]
     ]
]
make "sqrt [
     [x]
     [
          return sqrt_it :x 1
     ]
]
```

Linear Recursion and Iteration

```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

- $n! = n*(n-1)*(n-2)\dots1$
- $n! = n*[(n-1)*(n-2)\dots1] \longrightarrow n*(n-1)!$

```
make "factorial [[n][
    if lt :n 2
        [return 1]
        [return (:n * factorial (:n - 1))]
]]
```

More Productive

- Another rule for computing n! is that we first multiply 1 by 2, then multiply the result by 3, then by 4, and so on until we reach n. More formally, we maintain a running product, together with a counter that counts from 1 up to n.

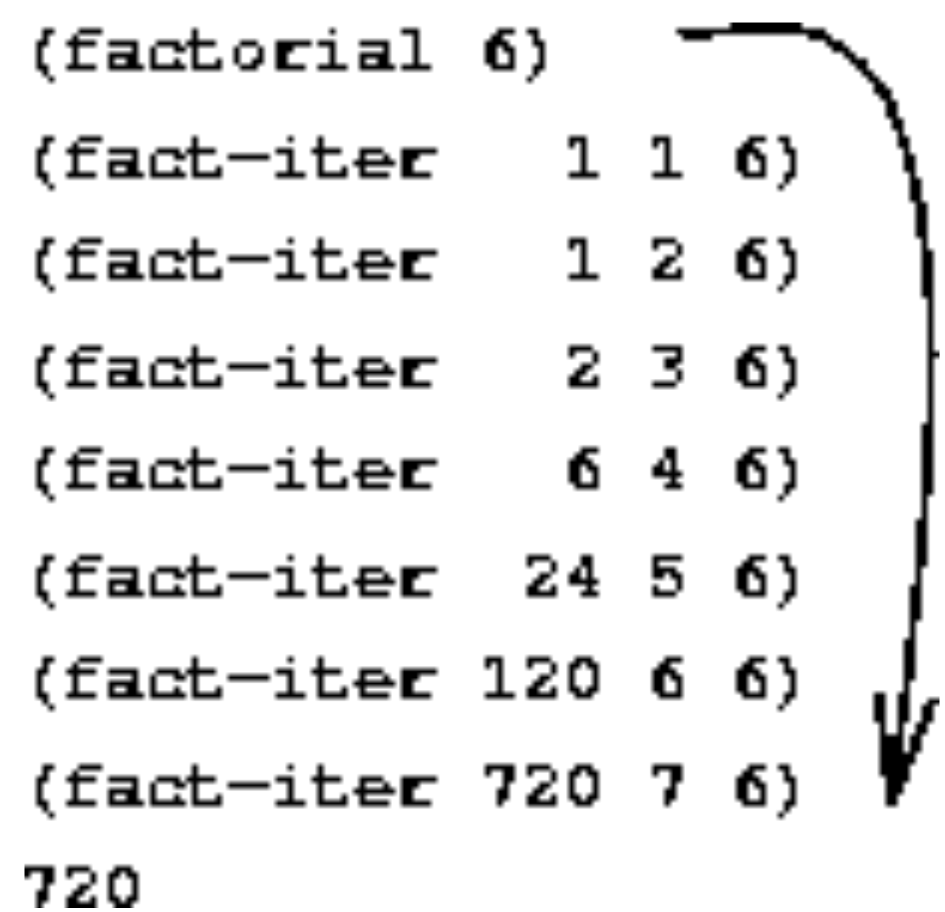
  - product = counter * product

  - counter = counter + 1

  - until the counter exceeds n, and then that n! is the value of the product

linear iterative

```
make "factorial_iter [[productor counter n][
     if gt :counter :n
          [return :productor]
          [return factorial_iter
               (:productor * :counter)
               (:counter + 1)
               :n
          ]
]]
```



```
(factorial 6)
(fact-iter   1 1 6)
(fact-iter   1 2 6)
(fact-iter   2 3 6)
(fact-iter   6 4 6)
(fact-iter  24 5 6)
(fact-iter 120 6 6)
(fact-iter 720 7 6)
720
```

iterative vs recursive

- In the iterative case, the program variables provide a complete description of the state of the process at any point. If we stopped the computation between steps, all we would need to do to resume the computation is to supply the interpreter with the values of the three program variables.

- In the recursive case, there is some additional “hidden” information, maintained by the interpreter and not contained in the program variables, which indicates “where the process is” in negotiating the chain of deferred operations.

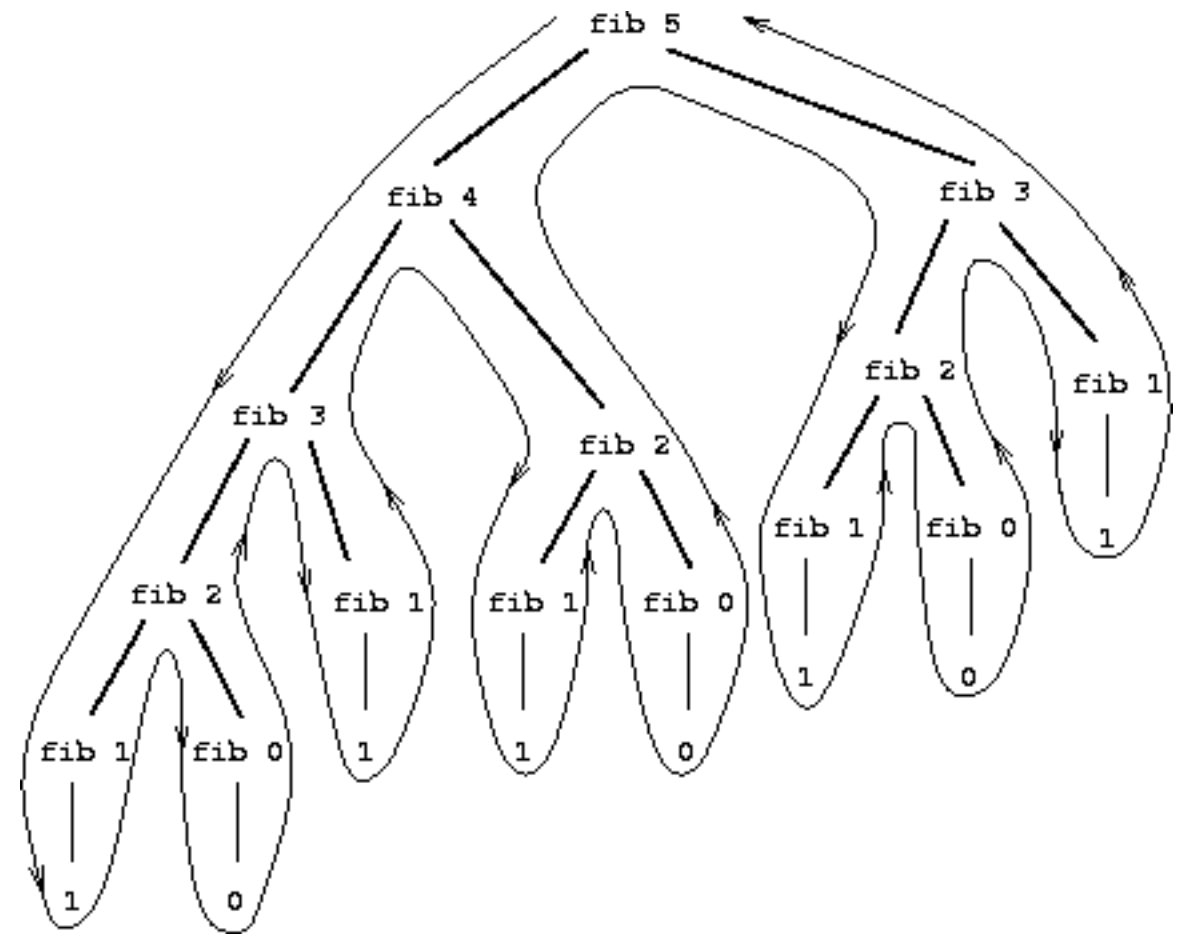
iterative vs recursive

```
(define (+ a b)
  (if (= a 0)
      b
      (inc (+ (dec a) b))))

(define (+ a b)
  (if (= a 0)
      b
      (+ (dec a) (inc b))))
```

Tree Recursion

$$\mathrm{Fib}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2) & \text{otherwise} \end{cases}$$




```
make "fib [[x][
    if lt :n 2
        [return 1]
        [return (fib (:n - 1) + fib (:n - 2))]
]]
```

iterative

- a = a+b
- b = original a
- counter = counter + 1
- until counter reaches n

The difference in number of steps required by the two methods -- one linear in n, one growing as fast as $Fib(n)$ itself -- is enormous, even for small inputs.

A tree-recursive process may be highly inefficient but often easy to specify and underst[illegible]nd has led people to propose that one [illegible] of both worlds by designing a “smart compiler” that could transform tree-recursive procedures into more efficient procedures that compute the same result.

memorization!

```
make "fib_iter [[a b counter] [
    if :counter = 0
        [return :b]
        [return fib_iter (:a + :b) :a (:counter - 1)]
]]
```

Pascal's triangle

```
        1
      1   1
    1   2   1
  1   3   3   1
1   4   6   4   1
       . . .
```

- The numbers at the edge of the triangle are all 1, and each number inside the triangle is the sum of the two numbers above it. Write a procedure that computes elements of Pascal's triangle by means of a recursive process.

```
make "yang_iter [ [lst lv result] [
    if isempty :lst
        [return join :result 1]
        [return yang_iter butfirst :lst first :lst join :result
add :lv first :lst ]
]]
print yang_iter [1 2 1] 0 []
```

Exponentiation Calculation

- $b^n = b * b^{n-1}$
- $b^0 = 1$

```
make "exp [[b n] [
    if lt :n 1
        [return 1]
        [return (:b * exp :b (:n - 1))]
]]
```

Optimization

$$b \cdot (b \cdot (b \cdot (b \cdot (b \cdot (b \cdot (b \cdot b))))))$$

$$\begin{aligned} b^2 &= b \cdot b \\ b^4 &= b^2 \cdot b^2 \\ b^8 &= b^4 \cdot b^4 \end{aligned}$$

$$\begin{aligned} b^n &= (b^{n/2})^2 && \text{if } n \text{ is even} \\ b^n &= b \cdot b^{n-1} && \text{if } n \text{ is odd} \end{aligned}$$

```
make "even? [[x] [
      if (:x % 2) = 0
            [return "true]
            [return "false]
]]
make "square [[x] [
      return (:x * :x)
]]
make "fast_exp [[b n] [
      if :n = 0
            [return 1]
            [if even? :n
                  [return square fast_exp :b (:n / 2)]
                  [return (:b * fast_exp :b (:n - 1))]
            ]
]]
```

Design a procedure that evolves an iterative exponentiation process that uses successive squaring and uses a logarithmic number of steps, as does `fast-expt`. (Hint: Using the observation that $(b^{n/2})^2 = (b^2)^{n/2}$, keep, along with the exponent n and the base b, an additional state variable a, and define the state transformation in such a way that the product $a\,b^n$ is unchanged from state to state. At the beginning of the process a is taken to be 1, and the answer is given by the value of a at the end of the process. In general, the technique of defining an *invariant quantity* that remains unchanged from state to state is a powerful way to think about the design of iterative algorithms.)

Searching for divisors

```
make “mallest_divisor [ [n] [
    return find_divisor :n 2
]]

make “find_divisor [ [n test_divisor] [
    if gt square :test_divisor :n
        [return :n]
        [if divides? :test_divisor :n
            [return :test_divisor]
            [find_divisor :n add :test_divisor 1]
]]]

make "divisor? [ [a b] [
    return eq mod :b :a 0
]]

make "prime? [ [n] [
    return eq smallest_divisor :n :n
]]
```

The Fermat test

- **Fermat's Little Theorem**: If *n* is a prime number and *a* is any positive integer less than *n*, then *a* raised to the *n*th power is congruent to *a* modulo *n*.

- The Fermat test is performed by choosing at random a number a between 1 and n - 1 inclusive and checking whether the remainder modulo n of the nth power of a is equal to a.

- if n ever fails the Fermat test, we can be certain that n is not prime. But the fact that n passes the test, while an extremely strong indication, is still not a guarantee that n is prime.

- There do exist numbers that fool the Fermat test.

- Numbers that fool the Fermat test are called Carmichael numbers, and little is known about them other than that they are extremely rare. There are 255 Carmichael numbers below 100,000,000. The smallest few are 561, 1105, 1729, 2465, 2821, and 6601.

费马小定理

- 如果n是一个素数，a是小于n的任意正整数，那么a的n次方与a模n同余

- 对于给定的整数n，随机任取一个a<n并计算出a^n取模n的余数。如果得到的结果不等于a，那么n就肯定不是素数。如果它就是a，那么n是素数的机会就很大。

- 通过检查越来越多的a值，就可以越来越相信n是素数。

```
make "expmod [ [base exp m] [
    if eq :exp 0
        [return 1]
        [if even? :exp
            [return mod square expmod :base div :exp 2 m m]
            [return mod mul :base expmod :base sub :exp 1 m m]
]]]

make “fermat_test [ [n] [
    make “try_it [ [a] [
        return eq expmod :a :n :n :a]
    ]
    return try_it add random sub :n 1 1
]]

make “fast_prime? [ [n times] [
    if eq :times 0
        [return true]
        [if fermat_test :n
            [return fast_prime? :n sub :times 1]
            [return false]
]]]
```

Formulating Abstractions with Higher-Order Procedures

- To construct procedures that can accept procedures as arguments or return procedures as values.
- Procedures that manipulate procedures are called higher-order procedures.

sum up

```
make "sum_int [[a b][
    if :a > :b
        [return 0]
        [return (:a + sum_int (:a + 1) :b))]
]]

make "pi_sum [[a b] [
    if :a > :b
        [return 0]
        [return  ((1.0 / (:a * (:a + 2))) + pi_sum (:a + 4) :b) ]
]]

make "name [[a b] [
    if :a > :b
        [return 0]
        [return  add <term> :a  name <next> :a: b
        ]
]]
```

$$\sum_{n=a}^{b} f(n) = f(a) + \cdots + f(b)$$

MUA

```
make "sum [[term a next b] [
    if :a > :b
        [return 0]
        [return (term :a + sum :term next :a :next :b)]
]]
make "pi_sum_term [[a] [
    return (1.0 / (:a * (:a + 2)))
]]
make "pi_sum_next [[a] [
    return (:a + 4)
]]
sum :pi_sum_term 1.0 :pi_sum_next 100
```

definite integral

- the definite integral of a function f between the limits a and b can be approximated numerically using the formula

$$\int_a^b f = \left[f\left(a + \frac{dx}{2}\right) + f\left(a + dx + \frac{dx}{2}\right) + f\left(a + 2dx + \frac{dx}{2}\right) + \ldots \right] dx$$

```
make "integral [ [f a b dx] [
    make "add-dx [ [x] [
        return add :x :dx
    ]]
    return mul sum :f add :a div :dx 2.0 :add-dx :b :dx
]]
```

iterative way

- The sum procedure above generates a linear recursion. The procedure can be rewritten so that the sum is performed iteratively. Show how to do this by filling in the missing expressions in the following definition:

```
make "sum [[term a next b] [
     make "iter [a result] [
          if <??>
               [output <??>]
               [output iter <??> <??>]
     ]
     output iter <??> <??>
]]
```

lamba

- MUA的过程就是一个两个子表的表
- 可以传递过程的地方也可以直接传递这个表

```
sum [[x][return (1.0 / (:x * (:x + 2)))]] 1.0 [[x][return (:x + 4)] 100
```

Finding roots of equations by the half-interval method

- To find roots of an equation $f(x) = 0$, where f is a continuous function. The idea is that, if we are given points a and b such that $f(a) < 0 < f(b)$, then f must have at least one zero between a and b. To locate a zero, let x be the average of a and b and compute $f(x)$. If $f(x) > 0$, then f must have a zero between a and x. If $f(x) < 0$, then f must have a zero between x and b. Continuing in this way, we can identify smaller and smaller intervals on which f must have a zero.

```
make "bs_root [[f left right] [
    make "mid average :left :right
    if close_enuogh? :left :right
        [return :mid]
        [
            make "test_value f :mid
            if gt :test_value 0
                [return bs_root :f :left :mid]
                [
                    if lt :test_value 0
                        [return bs_root :f :mid :right]
                        [return :mid]
                ]
        ]
]]
```

linear combination

- Consider the idea of forming a “linear combination” ax + by. We might like to write a procedure that would accept a, b, x, and y as arguments and return the value of ax +by

```
make "linear_combination [ [a b x y] [
    return add mul :a :x mul :b :y
]]
```

这样的代码能否用于其他的线性组合?

Rational Numbers

- make_rat n d returns the rational number whose numerator is the integer ⟨n⟩ and whose denominator is the integer ⟨d ⟩.
- numer x returns the numerator of the rational number ⟨x ⟩
- denom x returns the denominator of the rational number ⟨x ⟩.

```
make "add_rat [[x y] [
    return make_rat
        (numer :x * denom :y + numer :y * denom :x)
        (denom :x * denom :y)
]]
make "sub_rat [[x y] [
    return make_rat
        (numer :x * denom :y - numer :y * denom :x)
        (denom :x * denom :y)
]]
make "mul_rat [[x y] [
    return make_rat
        (numer :x * numer :y)
        (denom :x * denom :y)
]]
make "div_rat [[x y] [
    return make_rat
        (numer :x * denom :y)
        (denom :x * numer :y)
]]
make "eq_rat? [[x y] [
    return eq
        mul numer :x denom :y
        mul numer :y denom :x
]]
```

```
make "make_rat [[a b] [
    return sentence :a :b
]]

make "numer [[r] [
    return first :r
]]

make "denom [[r] [
    return last :r
]]
```