This Lecture

- Substitution model
- An example using the substitution model
- Designing recursive procedures
- Designing iterative procedures

Substitution model

- a way to figure out what happens during evaluation
  - not really what happens in the computer

•to apply a compound procedure:
  •evaluate the body of the procedure, with each parameter replaced by the corresponding operand

•to apply a primitive procedure: just do it

```
(define square (lambda (x) (* x x)))

1.            (square 4)
2.            (* 4 4)
3.            16
```


Substitution model details

```
(define square (lambda (x) (* x x)))
(define average (lambda (x y) (/ (+ x y) 2)))

(average 5 (square 3))
(average 5 (* 3 3))
(average 5 9)
```

first evaluate operands, then substitute (applicative order)

```
(/ (+ 5 9) 2)
(/ 14 2)
7
```

if operator is a primitive procedure, replace by result of operation

End of part 1

- how to use substitution model to trace evaluation

A less trivial procedure: factorial

- Compute n factorial, defined as n! = n(n-1)(n-2)(n-3)...1

•Notice that n! = n * [(n-1)(n-2)...] = n * (n-1)! if n > 1

```
(define fact
       (lambda (n)
          (if (= n 1)
              1
              (* n (fact (- n 1))))))
```

•predicate = tests numerical equality

```
(= 4 4) ==> #t     (true)
(= 4 5) ==> #f     (false)
```

•if special form

```
(if (= 4 4) 2 3) ==> 2
(if (= 4 5) 2 3) ==> 3
```

predicate consequent alternative


```
(define fact(lambda (n)
  (if (= n 1)1(* n (fact (- n 1))))))

(fact 3)
(if (= 3 1) 1 (* 3 (fact (- 3 1))))
(if #f 1 (* 3 (fact (- 3 1))))
(* 3 (fact (- 3 1)))
(* 3 (fact 2))
(* 3 (if (= 2 1) 1 (* 2 (fact (- 2 1)))))
(* 3 (if #f 1 (* 2 (fact (- 2 1)))))
(* 3 (* 2 (fact (- 2 1))))
(* 3 (* 2 (fact 1)))
(* 3 (* 2 (if (= 1 1) 1 (* 1 (fact (- 1 1))))))
(* 3 (* 2 (if #t 1 (* 1 (fact (- 1 1))))))
(* 3 (* 2 1))
(* 3 2)
6
```

The fact procedure is a recursive algorithm

- A recursive algorithm:
  - In the substitution model, the expression keeps growing

```
(fact 3)
(* 3 (fact 2))
(* 3 (* 2 (fact 1)))
```

  - Other ways to identify will be described next time

End of part 2

- how to use substitution model to trace evaluation
- how to recognize a recursive procedure in the trace

How to design recursive algorithms

- follow the general pattern:
  1. wishful thinking
  2. decompose the problem
  3. identify non-decomposable (smallest) problems

1. Wishful thinking

- Assume the desired procedure exists.
- want to implement fact? OK, assume it exists.
- BUT, only solves a smaller version of the problem.

2. Decompose the problem

- Solve a problem by
  1. solve a smaller instance (using wishful thinking)
  2. convert that solution to the desired solution
- Step 2 requires creativity!
  - Must design the strategy before coding.
  - n! = n(n-1)(n-2)... = n[(n-1)(n-2)...] = n * (n-1)!
  - solve the smaller instance, multiply it by n to get solution

```
(define fact
    (lambda (n) (* n (fact (- n 1)))))
```


3. Identify non-decomposable problems

- Decomposing not enough by itself
- Must identify the "smallest" problems and solve directly
- Define 1! = 1

```
(define fact
    (lambda (n)
      (if (= n 1) 1
          (* n (fact (- n 1))))))
```


General form of recursive algorithms

- test, base case, recursive case

```
(define fact
 (lambda (n)
   (if (= n 1)        ; test for base case
     1                ; base case
     (* n (fact (- n 1)))  ; recursive case
)))
```

- base case: smallest (non-decomposable) problem
- recursive case: larger (decomposable) problem

End of part 3

- Design a recursive algorithm by
  1. wishful thinking
  2. decompose the problem
  3. identify non-decomposable (smallest) problems

- Recursive algorithms have
  1. test
  2. recursive case
  3. base case

Iterative algorithms

- In a recursive algorithm, bigger operands => more space

```
(define fact (lambda (n)
   (if (= n 1) 1
        (* n (fact (- n 1))))))
(fact 4)
(* 4 (fact 3))
(* 4 (* 3 (fact 2)))
(* 4 (* 3 (* 2 (fact 1))))
(* 4 (* 3 (* 2 1)))
...
24
```

•An iterative algorithm uses constant space

Intuition for iterative factorial

- same as you would do if calculating 4! by hand:
  1. multiply 4 by 3 gives 12
  2. multiply 12 by 2 gives 24
  3. multiply 24 by 1 gives 24

•At each step, only need to remember:
previous product, next multiplier

•Therefore, constant space

•Because multiplication is associative and commutative:
1. multiply 1 by 2 gives 2
2. multiply 2 by 3 gives 6
3. multiply 6 by 4 gives 24

Iterative algorithm to compute 4! as a table

- In this table:
  - One column for each piece of information used
  - One row for each step

first row handles `0!` cleanly

product	counter	N
1	1	4
1	2	4
2	3	4
6	4	4
24	5	4

product * counter

counter +1

answer

- The last row is the one where counter > n
- The answer is in the product column of the last row

Iterative factorial in scheme

- (define ifact (lambda (n) (ifact-helper 1 1 n)))

initial row of table

(define ifact-helper (lambda (product counter n)

(if (> counter n)

product

compute next row of table

(ifact-helper (* product counter) (+ counter 1) n))))

answer is in product column of last row

at last row when counter > n

Partial trace for `(ifact 4)`

```
(define ifact-helper (lambda (product count n)
      (if (> count n) product
          (ifact-helper (* product count)
                    (+ count 1) n))))

(ifact 4)
(ifact-helper 1 1 4)
(if (> 1 4) 1 (ifact-helper (* 1 1) (+ 1 1) 4))
(ifact-helper 1 2 4)
(if (> 2 4) 1 (ifact-helper (* 1 2) (+ 2 1) 4))
(ifact-helper 2 3 4)
(if (> 3 4) 2 (ifact-helper (* 2 3) (+ 3 1) 4))
(ifact-helper 6 4 4)
(if (> 4 4) 6 (ifact-helper (* 6 4) (+ 4 1) 4))
(ifact-helper 24 5 4)
(if (> 5 4) 24 (ifact-helper (* 24 5) (+ 5 1) 4))
24
```

Iterative = no pending operations when procedure calls itself

- Recursive factorial:

```
(define fact (lambda (n)
    (if (= n 1) 1
        (* n (fact (- n 1)) )
    )))
```

pending operation

- `(fact 4)`

```
(* 4 (fact 3))
(* 4 (* 3 (fact 2)))
(* 4 (* 3 (* 2 (fact 1))))
```

- Pending ops make the expression grow continuosly

Iterative = no pending operations

- Iterative factorial:

```
(define ifact-helper (lambda (product count n)
    (if (> count n) product
        (ifact-helper (* product count)
                      (+ count 1) n))))
```

- `(ifact-helper 1 1 4)`

```
(ifact-helper 1 2 4)
(ifact-helper 2 3 4)
(ifact-helper 6 4 4)
(ifact-helper 24 5 4)
```

no pending operations

- Fixed size because no pending operations

End of part 4

- Iterative algorithms have constant space
- How to develop an iterative algorithm
  - figure out a way to accumulate partial answers
  - write out a table to analyze precisely:
    - initialization of first row
    - update rules for other rows
    - how to know when to stop
  - translate rules into scheme code

- Iterative algorithms have no pending operations when the procedure calls itself