

This Lecture

- Substitution model
- An example using the substitution model
- Designing recursive procedures
- Designing iterative procedures
- Proving that our code works

1/32

Substitution model

- A way to figure out what happens during evaluation
- **Not really what happens in the computer**



Rules of substitution model:

1. If **self-evaluating** (e.g. number, string, #t / #f), just return value
2. If **name**, replace it with value associated with that name
3. If **lambda**, create a procedure
4. If **special form** (e.g. if), follow the special form's rules for evaluating
5. If **combination** ($e_0 e_1 e_2 \dots e_n$):
 - Evaluate subexpressions e_i in any order to produce values ($v_0 v_1 v_2 \dots v_n$)
 - If v_0 is **primitive procedure** (e.g. +), just apply it to $v_1 \dots v_n$
 - If v_0 is **compound procedure** (created by lambda):
 - Substitute $v_1 \dots v_n$ for corresponding parameters in body of procedure, then repeat on body

2/32

Micro Quiz

```
(define average (lambda (x y) (/ (+ x y) 2)))  
(average (+ 3 4) 3)  
(5)
```

Rules of substitution model

1. If **self-evaluating** (e.g. number, string, #t / #f), just return value
2. If **name**, replace it with value associated with that name
3. If **lambda**, create a procedure
4. If **special form** (e.g. if), follow the special form's rules for evaluating
5. If **combination** ($e_0 e_1 e_2 \dots e_n$):
 - Evaluate subexpressions e_i in any order to produce values ($v_0 v_1 v_2 \dots v_n$)
 - If v_0 is **primitive procedure** (e.g. +), just apply it to $v_1 \dots v_n$
 - If v_0 is **compound procedure** (created by lambda):
 - Substitute $v_1 \dots v_n$ for corresponding parameters in body of procedure, then repeat on body

3/32

Substitution model – a simple example

```
(define square (lambda (x) (* x x)))  
  
(square 4)  
square → [procedure (x) (* x x)]  
4 → 4  
(* 4 4)  
16  
  
(define average (lambda (x y) (/ (+ x y) 2)))  
  
(average 5 (square 3))  
(average 5 (* 3 3))  
(average 5 9)  
(/ (+ 5 9) 2)  
(/ 14 2)  
7
```

4/32

A less trivial example: factorial

- Compute **n factorial**, defined as
$$n! = n(n-1)(n-2)(n-3)\dots 1$$
- How can we capture this in a procedure, using the idea of finding a common pattern?

5/32

How to design recursive algorithms

- Follow the general approach:
 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, it only solves a **smaller** version of the problem.
 - This is just like finding a common pattern: but here, solving the bigger problem involves the same pattern in a smaller problem

6/32

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 writing Scheme code.
 - $n! = n(n-1)(n-2)\dots = n[(n-1)(n-2)\dots] = n * (n-1)!$
 - solve the smaller instance, multiply it by n to get solution

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

7/32

Minor Difficulty

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

```
(fact 2)
(* 2 (fact 1))
(* 2 (* 1 (fact 0)))
(* 2 (* 1 (* 0 (fact -1)))) ... d'oh!
```

8/32

3. Identify non-decomposable problems

- Decomposing is not enough by itself
- Must identify the "smallest" problems and solve directly
- Define $1! = 1$ (or alternatively define $0! = 1$)

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

9/32

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
- more complex algorithms may have multiple base cases or multiple recursive cases (requiring more than one test)

10/32

Summary of recursive processes

- Design a recursive algorithm by
 1. wishful thinking
 2. decompose the problem
 3. identify non-decomposable (smallest) problems
- Recursive algorithms have
 1. test
 2. base case
 3. recursive case

11/32

```
(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
```

12/32

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

```
(fact 3)
```

Note the “**shape**” of this process

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

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

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

```
(* 3 2)
```

```
6
```

13/32

The fact procedure uses a recursive algorithm

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


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

14/32

Recursive algorithms use increasing space

- In a recursive algorithm, bigger operands consume more space

```
(fact 4)
(* 4 (fact 3))
(* 4 (* 3 (fact 2)))
(* 4 (* 3 (* 2 (fact 1))))
(* 4 (* 3 (* 2 1)))
...
24
```

```
(fact 8)
(* 8 (fact 7))
(* 8 (* 7 (fact 6)))
(* 8 (* 7 (* 6 (fact 5))))
...
(* 8 (* 7 (* 6 (* 5 (* 4 (* 3 (* 2 (fact 1))))))))
(* 8 (* 7 (* 6 (* 5 (* 4 (* 3 (* 2 1))))))
(* 8 (* 7 (* 6 (* 5 (* 4 (* 3 2))))))
...
40320
```

15/32

A Problem With Recursive Algorithms

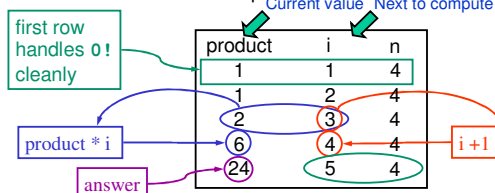
- Try computing 101!


```
101 * 100 * 99 * 98 * 97 * 96 * ... * 2 * 1
```
- How much space do we consume with pending operations?
- Better idea:
 - start with 1, remember that 2 is next
 - compute 1 * 2, remember that 3 is next
 - compute 2 * 3, remember that 4 is next
 - compute 6 * 4, remember that 5 is next
 - ...
 - compute 942594775983835942085162312448293674956231279470254376832788935341697759931622147650308786159180834691162349000354959958336970630260326400000000000000000000, and stop
- This is an **iterative algorithm** – it uses **constant space**

16/32

Iterative algorithm to compute 4! as a table

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



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

17/32

Iterative factorial in scheme

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

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

answer is in product column of last row at last row when $i > n$

18/32

Partial trace for (ifact 4)

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

19/32

Partial trace for (ifact 4)

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

(ifact 4)
(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)
24
```

Note the "shape" of this process

20/32

Recursive process = 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 operations make the expression grow continuously

21/32

Iterative process = no pending operations

- Iterative factorial:


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

no pending operations

```
(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)
```
- Fixed space because no pending operations

22/32

Summary of iterative processes

- Iterative algorithms use 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

23/32

Why is our code correct?

- How do we know that our code will always work?
 - Proof by authority** – someone with whom we dare not disagree says it is right!
 - For example
 - Proof by statistics** – we try enough examples to convince ourselves that it will always work!
 - E.g. keep trying, but bring sandwiches and a cot
 - Proof by faith** – we really, really, really believe that we always write correct code!
 - E.g. the Pset is due in 5 minutes and I don't have time
 - Formal proof** – we break down and use mathematical logic to determine that code is correct.



24/32

Proof by induction

- Proof by induction is a very powerful tool in predicate logic

$$\begin{array}{l} P(0) \\ \hline \forall n : P(n) \rightarrow P(n+1) \\ \hline \therefore \forall n : P(n) \end{array}$$

- Informally, if you can:
 - Show that some proposition P is true for $n=0$
 - Show that whenever P is true for some legal value of n , then it follows that P is true for $n+1$
 ...then you can conclude that P is true for all legal values of n

25/32

A simple example


$$\begin{array}{l} 1 = 1 \\ 1 + 2 = 3 \\ 1 + 2 + 4 = 7 \\ 1 + 2 + 4 + 8 = 15 \\ \dots \end{array}$$

$$\sum_{i=0}^n 2^i = 2^{n+1} - 1$$

26/32

An example of proof by induction


$$P(n) : \sum_{i=0}^n 2^i = 2^{n+1} - 1$$

Base case: $n = 0 : 2^0 = 2^1 - 1$ 

Inductive step: $\forall n : P(n) \rightarrow P(n+1)$

$$\sum_{i=0}^n 2^i = 2^{n+1} - 1 \quad P(n)$$

$$\sum_{i=0}^{n+1} 2^i = 2^{n+1} + 2^{n+1} = 2^{n+1} - 1 + 2^{n+1}$$

$$\sum_{i=0}^{n+1} 2^i = 2^{n+2} - 1 \quad P(n+1) \quad \text{}$$

27/32

Steps in proof by induction

- Define the predicate $P(n)$** (induction hypothesis)
 - Decide what the variable n denotes
 - Decide the universe over which n applies
- Prove that **$P(0)$ is true** (base case)
- Prove that **$P(n)$ implies $P(n+1)$** for all n (inductive step)
 - Do this by assuming that $P(n)$ is true, then trying to prove that $P(n+1)$ is true
- Conclude that **$P(n)$ is true for all n** by the principle of induction.

28/32

Back to factorial

- Induction hypothesis $P(n)$:

"our recursive procedure for **fact** correctly computes $n!$ for all integer values of n , starting at 1"

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

29/32

Proof by induction that fact works

- Base case:** does this work when $n=1$?
 - Note that this is $P(1)$, not $P(0)$ – we need to adjust the base case because our universe of legal values for n includes **only the positive integers**
- Yes – the IF statement guarantees that in this case we only evaluate the consequent expression: thus we return 1, which is 1!

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

30/32

Proof by induction that `fact` works

- Inductive step: We assume it works for some legal value of $n > 0$...

- so `(fact n)` computes $n!$ correctly

... and show that it works correctly for $n+1$

- What does `(fact n+1)` compute?

- Use the substitution model:

```
(fact n+1)
(if (= n+1 1) 1 (* n+1 (fact (- n+1 1))))
(if #f      1 (* n+1 (fact (- n+1 1))))
(* n+1 (fact (- n+1 1)))
(* n+1 (fact n))
(* n+1 n!)
(n+1)!
```

- By induction, `fact` will always compute what we expected, provided the input is in the right range ($n > 0$)

31/32

Lessons learned

- Induction provides the basis for supporting recursive procedure definitions
- In designing procedures, we should rely on the same thought process
 - Find the base case, and create solution
 - Determine how to reduce to a simpler version of same problem, plus some additional operations
 - Assume code will work for simpler problem, and design solution to extended problem

32/32