This Lecture

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

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 (e0 e1 e2 ... en):
   - Evaluate subexpressions ei in any order to produce values (v0 v1 v2 ... vn)
   - If vi is primitive procedure (e.g. +), just apply it to v1 ... vn
   - If vi is compound procedure (created by lambda):
     - Substitute v1 ... vn for corresponding parameters in body of procedure, then repeat on body

Micro Quiz

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

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 (e0 e1 e2 ... en):
   - Evaluate subexpressions ei in any order to produce values (v0 v1 v2 ... vn)
   - If vi is primitive procedure (e.g. +), just apply it to v1 ... vn
   - If vi is compound procedure (created by lambda):
     - Substitute v1 ... vn for corresponding parameters in body of procedure, then repeat on body

Substitution model – a simple example

(define square (lambda (x) (* x x)))
(square 4)

average (+ 3 4) 3

A less trivial example: factorial

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

- How can we capture this in a procedure, using the idea of finding a common pattern?

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
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)... = 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 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))))))

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!

General form of recursive algorithms

• test, base case, recursive case

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

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

(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
(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

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

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 942594775883859420851621244829367495623127947
    025437683278935341697759931922147650308786518083469
    1162349000354995953369796302632640000000000000000000000000,
    and stop
• This is an iterative algorithm – it uses constant space

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 first row handles 0! cleanly
  • The last row is the one where i > n
  • The answer is in the product column of the last row

<table>
<thead>
<tr>
<th>product</th>
<th>i</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

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)))
)
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)
(ifact-helper 24 5 4)
```

Note the “shape” of this process

```
(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
```

Recursive process = pending operations when procedure calls itself

- Recursive factorial:
  ```scheme
  (define fact (lambda (n)
      (if (= n 1) 1
       (* n (fact (- n 1)))))
  ```
  ```scheme
  (fact 4)
  (* 4 (fact 3))
  (* 4 (* 3 (fact 2)))
  (* 4 (* 3 (* 2 (fact 1))))
  ```
  Pending operations make the expression grow continuously

Iterative process = no pending operations

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

Summary of iterative processes

- Iterative algorithms use constant space
- How to develop an iterative algorithm
  1. Figure out a way to accumulate partial answers
  2. Write out a table to analyze precisely:
     - initialization of first row
     - update rules for other rows
     - how to know when to stop
  3. Translate rules into Scheme code
- Iterative algorithms have no pending operations when the procedure calls itself

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.
Proof by induction

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

\[
P(0) \\
\forall n: P(n) \rightarrow P(n+1) \\
\therefore \forall n: P(n)
\]

• Informally, if you can:
  1. Show that some proposition \( P \) is true for \( n=0 \)
  2. 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 \( n \)

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: \( \forall n: P(n) \rightarrow P(n+1) \)
\[
\sum_{i=0}^{n} 2^i + 2^{n+1} = (2^{n+1} - 1) + 2^{n+1} \\
\sum_{i=0}^{n+1} 2^i = 2^{n+1} - 1 \quad P(n+1)
\]

Steps in proof by induction

1. Define the predicate \( P(n) \) (induction hypothesis)

   • Decide what the variable \( n \) denotes
   • Decide the universe over which \( n \) applies

2. Prove that \( P(0) \) is true (base case)

3. 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

4. Conclude that \( P(n) \) is true for all \( n \) by the principle of induction.

Back to factorial

• Induction hypothesis \( P(n) \):

  “our recursive procedure for \texttt{fact} correctly computes \( n! \) for all integer values of \( n \), starting at 1”

\[
\text{(define fact} \\
(\text{lambda} \ (n) \\
\quad (\text{if} \ (= \ n \ 1) \\
\quad \quad 1 \\
\quad \quad (* \ n \ (\text{fact} \ (- \ n \ 1))))))
\]

Proof by induction that \texttt{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!

\[
\text{(define fact} \\
(\text{lambda} \ (n) \\
\quad (\text{if} \ (= \ n \ 1) \\
\quad \quad 1 \\
\quad \quad (* \ n \ (\text{fact} \ (- \ n \ 1))))))
\]
Proof by induction that fact works

- **Inductive step**: We assume it works for some legal value of \( n > 0 \)…
  - so \( \text{fact} \; n \) computes \( n! \) correctly
  - and show that it works correctly for \( n+1 \)
- What does \( \text{fact} \; n+1 \) compute?
- Use the substitution model:
  
  \[
  \begin{align*}
  \text{fact} \; n+1 &= \begin{cases}
  1 & \text{if } (n+1) \leq 1 \\
  (* \; n+1 \; (\text{fact} \; (- \; n+1 \; 1))) & \text{if } n \neq 1 \; (\text{fact} \; (- \; n+1 \; 1)))
  \end{cases}
  
  &\quad (* \; n+1 \; (\text{fact} \; n))
  
  &\quad (* \; n+1 \; n!)
  
  &\quad (n+1)! 
  \end{align*}
  \]

- By induction, \( \text{fact} \) will always compute what we expected, provided the input is in the right range \( (n > 0) \)

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