Structure and Interpretation of Computer Programs

COMP200
QUIZ

Write your name please!

Define your name as a variable whose value is your ID.
QUIZ
Write these procedures

\[ 1 + 2 + \ldots + 100 = \frac{(100 \times 101)}{2} \]

\[ 1^2 + 2^2 + \ldots + 100^2 = \frac{(100 \times 101 \times 201)}{6} \]
• Substitution model
• An example using the substitution model
• Designing recursive procedures
• Designing iterative procedures
- Substitution model
- An example using the substitution model
- Designing recursive procedures
- Designing iterative procedures
A way to figure out what happens during evaluation
A way to figure out what happens during evaluation

- not really what happens in the computer
SUBSTITUTION MODEL

Why?

Understand the process of evaluation

Work backwards from evaluation to design choices
SUBSTITUTION MODEL
Rules for Application

• If procedure is a **primitive procedure**, just do it.

• If procedure is a **compound procedure**, then:
  • *evaluate* the body of the procedure with each parameter replaced by the corresponding operand.
(define square (lambda(x)(* x x)))
SUBSTITUTION MODEL
Rules for Application

\[
\text{(define square (lambda(x)(* x x))})
\]

1. \text{(square 4)}
SUBSTITUTION MODEL
Rules for Application

\[
(\text{define square } (\text{lambda}(x)(\ast x x)))
\]

1. \((\text{square } 4)\)
2. \((\ast 4 4)\)
3. \(16\)
(define square (lambda (x) (* x x)))
(define average (lambda (x y) (/ (+ x y) 2))))
(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)
SUBSTITUTION MODEL
Rules for Application

(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)
SUBSTITUTION MODEL

Rules for Application

```
(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)

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

first evaluate operands, then substitute (applicative order)
SUBSTITUTION MODEL
Rules for Application

(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)

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

first evaluate operands, then substitute (applicative order)

if operator is a primitive procedure, replace by the result of operation
What have we learned?
SUBSTITUTION MODEL

What have we learned?

how to use substitution model to trace evaluation
• Substitution model
• An example using the substitution model
• Designing recursive procedures
• Designing iterative procedures
• Substitution model
• An example using the substitution model
• Designing recursive procedures
• Designing iterative procedures
Compute \( n \) factorial defined as:

\[ n! = n (n - 1) (n - 2) \cdots 1 \]
SUBSTITUTION MODEL

Factorial

Notice the pattern:

\[ n! = n \ (n - 1) \ (n - 2) \ \cdots \ 1 \]

\[ n! = n \ [(n - 1) \ (n - 2) \ \cdots \ 1] \]
Notice the pattern:

\[ n! = n \times (n-1)! \]

\[ n! = n \times [(n-1)!] \]

\[ n! = n \times (n-1) \]

\[ n! = n \times (n-1)! \]
SUBSTITUTION MODEL

Factorial

Careful!

\[
n! = n \times (n - 1) \times (n - 2) \cdots 1
\]

\[
n! = n \times [(n - 1) \times (n - 2) \cdots 1]
\]

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

procedure =
tests numerical equality

> (= 4 4)
#true

> (= 4 5)
#false
SUBSTITUTION MODEL

Factorial

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

*Factorial*

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

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

- **predicate**
- **consequence**
- **alternative**

*IF special form*
(define fact
  (lambda (n)
    (if (= n 1) 1
      (* n (fact (- n 1))))))
(define fact
  (lambda (n)
    (if (= n 1)
      1
      (* n (fact (- n 1)))))

SUBSTITUTION MODEL

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

(fact 3)
(define fact
  (lambda (n)
    (if (= n 1)
      1
      (* n (fact (- n 1))))))

(fact 3)
(if (= 3 1) 1 (* 3 (fact (- 3 1))))
(define fact
  (lambda (n)
    (if (= n 1)
        1
        (* n (fact (- n 1)))))

(fact 3)
(if (= 3 1) 1 (* 3 (fact (- 3 1))))
(if #false 1 (* 3 (fact (- 3 1))))
(define fact
  (lambda (n)
    (if (= n 1)
        1
        (* n (fact (- n 1))))))

(fact 3)
(if (= 3 1) 1 (* 3 (fact (- 3 1))))
(if #false 1 (* 3 (fact (- 3 1))))
(* 3 (fact (- 3 1)))
(define fact
  (lambda (n)
    (if (= n 1)
      1
      (* n (fact (- n 1))))))

(fact 3)
(if (= 3 1) 1 (* 3 (fact (- 3 1))))
(if #false 1 (* 3 (fact (- 3 1))))
(* 3 (fact (- 3 1)))
(* 3 (fact 2))
(define fact
  (lambda (n)
    (if (= n 1)
      1
      (* n (fact (- n 1))))))

(fact 3)
(if (= 3 1) 1 (* 3 (fact (- 3 1))))
(if #false 1 (* 3 (fact (- 3 1))))
(* 3 (fact (- 3 1)))
(* 3 (fact 2))
(* 3 (if (= 2 1) 1 (* 2 (fact (- 2 1)))))
(* 3 (if #false 1 (* 2 (fact (- 2 1))))))
(* 3 (* 2 (fact (- 2 1)))))
(* 3 (* 2 (fact 1)))
(define fact
  (lambda (n)
    (if (= n 1)
        1
        (* n (fact (+ - n 1))))))

(fact 3)
(if (= 3 1) 1 (* 3 (fact (+ - 3 1))))
(if #false 1 (* 3 (fact (+ - 3 1))))
(* 3 (fact (+ - 3 1)))
(* 3 (fact 2))
(* 3 (if (= 2 1) 1 (* 2 (fact (+ - 2 1)))))
(* 3 (if #false 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 #true 1 (* 1 (fact (+ - 1 1))))))
(* 3 (* 2 1))
(define fact
  (lambda (n)
    (if (= n 1)
        1
        (* n (fact (- n 1))))))

(fact 3)
(if (= 3 1) 1 (* 3 (fact (- 3 1))))
(if #false 1 (* 3 (fact (- 3 1))))
(* 3 (fact (- 3 1)))
(* 3 (fact 2))
(* 3 (if (= 2 1) 1 (* 2 (fact (- 2 1)))))
(* 3 (if #false 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 #true 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)
(if (= 3 1) 1 (* 3 (fact (- 3 1))))
(if #false 1 (* 3 (fact (- 3 1))))
(* 3 (fact (- 3 1)))
(* 3 (fact 2))
(* 3 (if (= 2 1) 1 (* 2 (fact (- 2 1))))))
(* 3 (if #false 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 #true 1 (* 1 (fact (- 1 1)))))))
(* 3 (* 2 1))
(* 3 2)
6
In the substitution model, the expression keeps growing:

\[
(fact\ 3) \\
(*\ 3\ (fact\ 2)) \\
(*\ 3\ (*\ 2\ (fact\ 1)))
\]
In the substitution model, the expression keeps growing:

\[
(fact \ 3) \\
(* \ 3 \ (fact \ 2)) \\
(* \ 3 \ (* \ 2 \ (fact \ 1)))
\]

More ways to identify.
SUBSTITUTION MODEL

What have we learned?

how to use substitution model to trace evaluation

???
What have we learned?

- how to use substitution model to trace evaluation
- how to recognise a recursive procedure in the trace
• Substitution model
• An example using the substitution model
• Designing recursive procedures
• Designing iterative procedures
TODAY
Outline

• Substitution model

• An example using the substitution model

• Designing recursive procedures

• Designing iterative procedures
RECURSIVE PROCEDURES
How to Design?

Follow the general pattern:

1. Wishful thinking
2. Decompose the problem
3. Identify non-decomposable (smallest) problems
RECURSIVE PROCEDURES
How to Design?

Follow the general pattern:

1. Wishful thinking
2. Decompose the problem
3. Identify non-decomposable (smallest) problems
RECURSIVE PROCEDURES

1. Wishful Thinking

- Assume the desired procedure exists.
RECURSIVE PROCEDURES

1. Wishful Thinking

- Assume the desired procedure exists.
- want to implement fact? OK, assume it exists.
RECURSIVE PROCEDURES

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.
RECURSIVE PROCEDURES

How to Design?

Follow the general pattern:

1. Wishful thinking
2. Decompose the problem
3. Identify non-decomposable (smallest) problems
RECURSIVE PROCEDURES

2. Decompose the Problem

• Solve the problem by
  • solving a smaller instance (using wishful thinking)
RECURSIVE PROCEDURES

2. Decompose the Problem

- Solve the problem by
  - solving a smaller instance (using wishful thinking)
  - converting that solution to the desired solution
RECURSIVE PROCEDURES

2. Decompose the Problem

- Solve the problem by
  - solving a smaller instance (using wishful thinking)
  - converting that solution to the desired solution

requires creativity!
RECURSIVE PROCEDURES

2. Decompose the Problem

converting that solution to the desired solution
design the strategy before coding:

\[ n! = n (n - 1) (n - 2) \cdots 1 \]
RECURSIVE PROCEDURES
2. Decompose the Problem

converting that solution to the desired solution

design the strategy before coding:

\[ n! = n \times (n - 1)! \quad \text{if } n > 1 \]
RECURSIVE PROCEDURES

2. Decompose the Problem

converting that solution to the desired solution
design the strategy before coding:

\[ n! = n \times (n - 1) \times (n - 2) \times \cdots \times 1 \]

\[ n! = n \times (n - 1)! \text{ if } n > 1 \]

solve the smaller instance
multiply it by n to get solution
RECURSIVE PROCEDURES

2. Decompose the Problem

converting that solution to the desired solution

design the strategy before coding:

\[ n! = n \ (n - 1) \ (n - 2) \ \cdots \ 1 \]

\[ n! = n \ [ (n - 1) \ (n - 2) \ \cdots \ 1] \]

\[ n! = n \times (n - 1)! \ \text{if} \ n > 1 \]

\[(\text{define fact} \ \lambda n (\lambda \ n (\text{fact} (- n 1))))\]

solve the smaller instance
multiply it by n to get solution
RECURSIVE PROCEDURES

2. Decompose the Problem

converting that solution to the desired solution

design the strategy before coding:

\[ n! = n \times (n-1) \times (n-2) \times \ldots \times 1 \]

\[ n! = n \times [(n-1) \times (n-2) \times \ldots \times 1] \]

\[ n! = n \times (n-1)! \quad \text{if} \quad n > 1 \]

(\text{define fact})

(\text{lambda} (n)

(* n (fact (- n 1)))))

It won’t work, why?
RECURSIVE PROCEDURES

How to Design?

Follow the general pattern:

1. Wishful thinking
2. Decompose the problem
3. Identify non-decomposable (smallest) problems
RECURSIVE PROCEDURES
Identify Non-decomposable Problems

• Decomposing alone is not enough.

• You must identify the smallest problems and solve them directly
RECURSIVE PROCEDURES
Identify Non-decomposable Problems

• Decomposing alone is not enough.
• You must identify the **smallest** problems and solve them directly

Define $1! = 1$
RECURSIVE PROCEDURES
Identify Non-decomposable Problems

• Decomposing alone is not enough.

• You must identify the **smallest** problems and solve them directly

Define \( 1! = 1 \)

\[
\text{(define fact}
\begin{array}{l}
\text{(lambda (n)}
\begin{array}{l}
\text{(if (= n 1)}
\begin{array}{l}
1
\end{array}
\text{1)}
\begin{array}{l}
(* \text{n (fact (- n 1)))})})
\end{array}
\end{array}
\end{array}
\)
RECURSIVE PROCEDURES
A General Form

test - base case - recursive case
RECURSIVE PROCEDURES
A General Form

test - base case - recursive case

(define fact
  (lambda (n)
    (if (= n 1)
      1
      (* n (fact (- n 1))))))
RECURSIVE PROCEDURES
A General Form

test - base case - recursive case

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

test for base case
RECURSIVE PROCEDURES
A General Form

test - base case - recursive case

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

test for base case
base case
RECURSIVE PROCEDURES
A General Form

test - base case - recursive case

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

test for base case
base case
recursive case
RECURSIVE PROCEDURES

A General Form

test - base case - recursive case

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

test for base case
base case
recursive case

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

What have we learned?
Design a recursive algorithm by

1. Wishful thinking
2. Decompose the problem
3. Identify non-decomposable (smallest) problems
What have we learned?

Design a recursive algorithm by

1. Wishful thinking
2. Decompose the problem
3. Identify non-decomposable (smallest) problems

Recursive algorithms have:

*test - base case - recursive case*
RECURSIVE PROCEDURES
The Space Issue

bigger operands => more space
bigger operands => more space

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

RECURSIVE ALGORITHMS
The Space Issue
RECURSIVE ALGORITHMS
The Space Issue

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
TODAY
Outline

• Substitution model
• An example using the substitution model
• Designing recursive procedures
• Designing iterative procedures
• Substitution model
• An example using the substitution model
• Designing recursive procedures
• Designing iterative procedures
An iterative algorithm uses constant space.
ITERATIVE ALGORITHMS

Intuition

Same as you’d do when calculating 4! by hand:

1. multiply 4 by 3 => 12
2. multiply 12 by 2 => 24
3. multiply 24 by 1 => 24
ITERATIVE ALGORITHMS

Intuition

Same as you’d do when calculating $4!$ by hand:

1. multiply 4 by 3 $\Rightarrow$ 12
2. multiply 12 by 2 $\Rightarrow$ 24
3. multiply 24 by 1 $\Rightarrow$ 24

At each step, remember previous product next multiplier
ITERATIVE ALGORITHMS

Intuition

Same as you’d do when calculating 4! by hand:

1. multiply 4 by 3  => 12
2. multiply 12 by 2  => 24
3. multiply 24 by 1  => 24

At each step, remember previous product
next multiplier

constant space
ITERATIVE ALGORITHMS

Intuition

Multiplication is associative and commutative:

1. multiply 1 by 2  => 2
2. multiply 2 by 3  => 6
3. multiply 6 by 4  => 24
## ITERATIVE ALGORITHMS

As a Table

<table>
<thead>
<tr>
<th>product</th>
<th>counter</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
ITERATIVE ALGORITHMS

As a Table

<table>
<thead>
<tr>
<th>product</th>
<th>counter</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>24</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

one column for for each piece of information used
## ITERATIVE ALGORITHMS

### As a Table

<table>
<thead>
<tr>
<th>product</th>
<th>counter</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
ITERATIVE ALGORITHMS

As a Table

<table>
<thead>
<tr>
<th>product</th>
<th>counter</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
**ITERATIVE ALGORITHMS**

*As a Table*

<table>
<thead>
<tr>
<th>product</th>
<th>counter</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

*handles 0! cleanly*

*product × counter*
ITERATIVE ALGORITHMS
As a Table

<table>
<thead>
<tr>
<th>product</th>
<th>counter</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
ITERATIVE ALGORITHMS
As a Table

<table>
<thead>
<tr>
<th>product</th>
<th>counter</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Product \times counter handles 0! cleanly.
### ITERATIVE ALGORITHMS

**As a Table**

<table>
<thead>
<tr>
<th>product</th>
<th>counter</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
ITERATIVE ALGORITHMS
Iterative Factorial in Scheme

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

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

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

(define ifact-helper
 (lambda (product counter n)
 (if (> counter n)
 product
 (ifact-helper(* product counter)(+ counter 1)n))))
Iterative Factorial in Scheme

\begin{verbatim}
(define ifact
  (lambda (n)(ifact-helper 1 1 n)))

(define ifact-helper
  (lambda (product counter n)
    (if (> counter n)
      product
      (ifact-helper(* product counter)(+ counter 1)n)))))
\end{verbatim}
(define ifact
  (lambda (n)(ifact-helper 1 1 n)))

(define ifact-helper
  (lambda (product counter n)
  (if (> counter n)
    product
    (ifact-helper(* product counter)(+ counter 1)n)))))
ITERATIVE ALGORITHMS

Partial Trace for (ifact 4)
(define ifact
  (lambda (n)(ifact-helper 1 1 n))))
(define ifact-helper
  (lambda (product counter n)
    (if (> counter n)
      product
      (ifact-helper(* product counter)(+ counter 1)n))))

(ifact 4)
(ifact-helper 1 1 4)
ITERATIVE ALGORITHMS

Partial Trace for \( (ifact \ 4) \)

\[
\text{(ifact 4)} \\
\text{(ifact-helper 1 1 4)} \\
\text{(if (> 1 4) 1 (ifact-helper (* 1 1) (+ 1 1) 4))}
\]
ITERATIVE ALGORITHMS

Partial Trace for (ifact 4)

(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))
ITERATIVE ALGORITHMS

Partial Trace for (ifact 4)

(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))
ITERATIVE ALGORITHMS

Partial Trace for (ifact 4)

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

(define ifact-helper
  (lambda (product counter n)
    (if (> counter n)
      product
      (ifact-helper(* product counter)(+ counter 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 ALGORITHMS
Partial Trace for (ifact 4)
ITERATIVE VS. RECURSIVE

Pending Operations

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

(define ifact-helper
  (lambda (product counter n)
    (if (> counter n)
      product
      (ifact-helper(* product counter)(+ counter 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)
ITERATIVE VS. RECURSIVE

Pending Operations

(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))))
ITERATIVE VS. RECURSIVE

Pending Operations

(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))))
ITERATIVE VS. RECURSIVE

Pending Operations

```
(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))))
```

Pending operations make the operation grow continuously!
ITERATIVE VS. RECURSIVE

Pending Operations

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

(define ifact-helper
  (lambda (product counter n)
    (if (> counter n)
      product
      (ifact-helper(* product counter)(+ counter 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
ITERATIVE VS. RECURSIVE

Pending Operations

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

(define ifact-helper
  (lambda (product counter n)
    (if (> counter n)
      product
      (ifact-helper(* product counter)(+ counter 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 there are no pending operations.
ITERATIVE ALGORITHMS

What have we learned?

- Iterative algorithms have constant space.
ITERATIVE ALGORITHMS

What have we learned?

• Iterative algorithms have constant space.
• How to develop an iterative algorithm
ITERATIVE ALGORITHMS

What have we learned?

• Iterative algorithms have constant space.

• How to develop an iterative algorithm
  1. figure out a way to accumulate partial answers
ITERATIVE ALGORITHMS

What have we learned?

• Iterative algorithms have constant space.

• How to develop an iterative algorithm
  1. figure out a way to accumulate partial answers
  2. write out a table to analyse precisely:
• Iterative algorithms have constant space.

• How to develop an iterative algorithm
  1. figure out a way to accumulate partial answers
  2. write out a table to analyse precisely:
     • initialization of first row
     • update rules for other rows
     • how to know when to stop
What have we learned?

• Iterative algorithms have constant space.

• How to develop an iterative algorithm
  1. figure out a way to accumulate partial answers
  2. write out a table to analyse precisely:
     • initialization of first row
     • update rules for other rows
     • how to know when to stop
  3. translate rules into scheme code
ITERATIVE ALGORITHMS

What have we learned?

• Iterative algorithms have constant space.

• How to develop an iterative algorithm

  1. figure out a way to accumulate partial answers

  2. write out a table to analyse 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.