Procedural abstraction and recursion

6.037 - Structure and Interpretation of Computer Programs

Mike Phillips, Benjamin Barenblat, Leon Shen, Ben Vandiver, Alex Vandiver, Arthur Migdal

Massachusetts Institute of Technology

Lecture 1

http://web.mit.edu/alexmv/6.037/

Class Structure

- TR, 7-9PM, through end of IAP
- `http://web.mit.edu/alexmv/6.037/`
- E-mail: `6.001-zombies@mit.edu`
- Five projects: due on the 10th, 15th, 17th, 24th, and 1st.
- Graded P/D/F
- Taking the class for credit is zero-risk!
- E-mail list sign-up on the website

Goals of the Class

- This is not a class to teach Scheme

Goals of the Class

- This is not a class to teach Scheme
- Nor really a class about programming at all

Goals of the Class

- This is not a class to teach Scheme
- Nor really a class about programming at all
- This is a course about Computer Science

Goals of the Class

- This is not a class to teach Scheme
- Nor really a class about programming at all
- This is a course about ~~Computer~~ Science
- ...which isn't about computers

Goals of the Class

- This is not a class to teach Scheme
- Nor really a class about programming at all
- This is a course about ~~Computer Science~~
- ...which isn't about computers
- ...nor actually a science

Goals of the Class

- This is not a class to teach Scheme
- Nor really a class about programming at all
- This is a course about ~~Computer Science~~
- ...which isn't about computers
- ...nor actually a science
- This is actually a class in computation

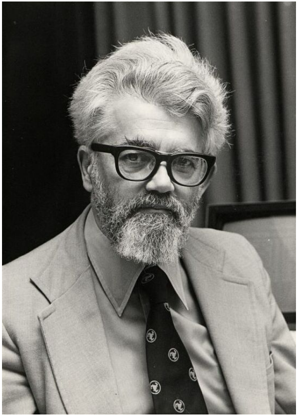
Prerequisites

- High confusion threshold
- Some programming clue
- A copy of Racket (Formerly PLT Scheme / DrScheme)
  `http://www.racket-lang.org/`
- Free time

Project 0

- Project 0 is out today
- Due on Thursday!
- Mail to `6.037-psets@mit.edu`
- Collaboration with current students is fine, as long as you note it

Some History

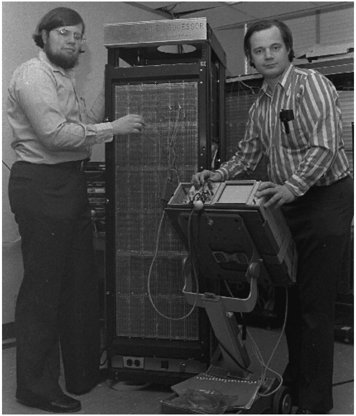


- Lisp invented in 1959 by John McCarthy (R.I.P. 2010)

Some History

- Lisp invented in 1959 by John McCarthy (R.I.P. 2010)
- Scheme invented in 1975 by Guy Steele and Gerald Sussman

Some History

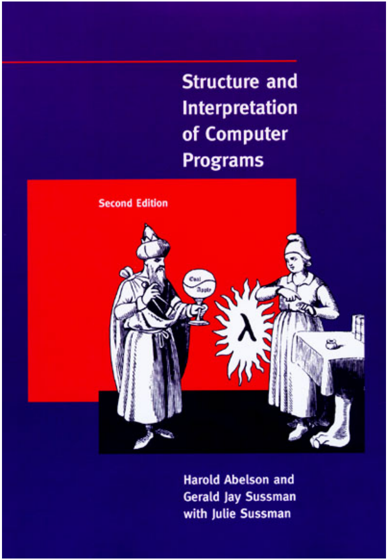


- Lisp invented in 1959 by John McCarthy (R.I.P. 2010)
- Scheme invented in 1975 by Guy Steele and Gerald Sussman
- Hardware Lisp machines, around 1978

Some History

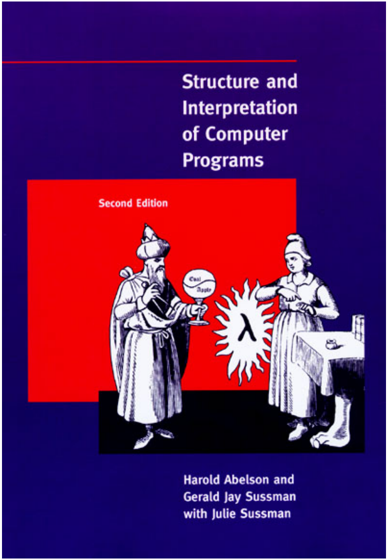
- Lisp invented in 1959 by John McCarthy (R.I.P. 2010)
- Scheme invented in 1975 by Guy Steele and Gerald Sussman
- Hardware Lisp machines, around 1978
- 6.001 first taught in 1980

Some History



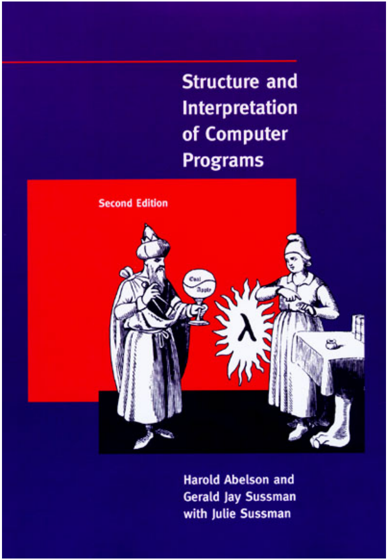
- Lisp invented in 1959 by John McCarthy (R.I.P. 2010)
- Scheme invented in 1975 by Guy Steele and Gerald Sussman
- Hardware Lisp machines, around 1978
- 6.001 first taught in 1980
- SICP published in 1984 and 1996

Some History

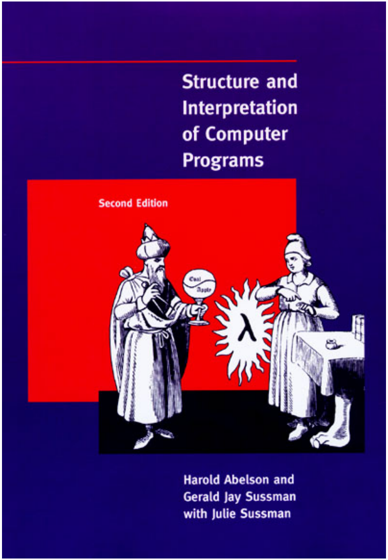


- Lisp invented in 1959 by John McCarthy (R.I.P. 2010)
- Scheme invented in 1975 by Guy Steele and Gerald Sussman
- Hardware Lisp machines, around 1978
- 6.001 first taught in 1980
- SICP published in 1984 and 1996
- R^6RS in 2007

Some History

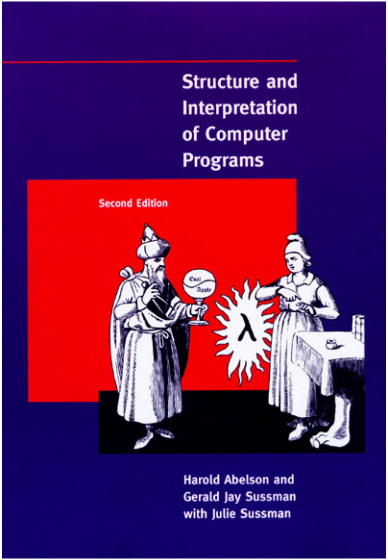


- Lisp invented in 1959 by John McCarthy (R.I.P. 2010)
- Scheme invented in 1975 by Guy Steele and Gerald Sussman
- Hardware Lisp machines, around 1978
- 6.001 first taught in 1980
- SICP published in 1984 and 1996
- R^6RS in 2007
- 6.001 last taught in 2007



- Lisp invented in 1959 by John McCarthy (R.I.P. 2010)
- Scheme invented in 1975 by Guy Steele and Gerald Sussman
- Hardware Lisp machines, around 1978
- 6.001 first taught in 1980
- SICP published in 1984 and 1996
- R^6RS in 2007
- 6.001 last taught in 2007
- 6.037 first taught in 2009

The Book ("SICP")



- Structure and Interpretation of Computer Programs
  by Harold Abelson and Gerald Jay Sussman
- `http://mitpress.mit.edu/sicp/`
- Not required reading
- Useful as study aid and reference
- Roughly one lecture per chapter

Key ideas

- Procedural and data abstraction
- Conventional interfaces & programming paradigms
  - Type systems
  - Streams
  - Object-oriented programming
- Metalinguistic abstraction
  - Creating new languages
  - Evaluators

Key ideas

- Procedural and data abstraction
- Conventional interfaces & programming paradigms
  - Type systems
  - Streams
  - Object-oriented programming
- Metalinguistic abstraction
  - Creating new languages
  - Evaluators

Lectures

1. Syntax of Scheme, procedural abstraction, and recursion
2. Data abstractions, higher order procedures, symbols, and quotation
3. Mutation, and the environment model
4. Interpretation and evaluation
5. Debugging
6. Language design and implementation
7. Continuations, concurrency, lazy evaluation, and streams
8. 6.001 in perspective, and the Lambda Calculus

Projects

0	Basic Scheme warm-up	Thursday 1/10
1	Higher-order procedures and symbols	Tuesday 1/15
2	Mutable objects and procedures with state	Thursday 1/17
3	Meta-circular evaluator	Thursday 1/24
4	OOP evaluator (The Adventure Game)	Friday 2/1*

Computation is Imperative Knowledge

- "How to" knowledge
- To approximate $\sqrt{x}$ (Heron's Method):
  - Make a guess G
  - Improve the guess by averaging G and $\frac{x}{G}$
  - Keep improving until it is good enough

Computation is Imperative Knowledge

- "How to" knowledge
- To approximate $\sqrt{x}$ (Heron's Method):
  - Make a guess G
  - Improve the guess by averaging G and $\frac{x}{G}$
  - Keep improving until it is good enough

$$x = 2 \qquad G \qquad = 1$$

Computation is Imperative Knowledge

- "How to" knowledge
- To approximate $\sqrt{x}$ (Heron's Method):
  - Make a guess G
  - Improve the guess by averaging G and $\frac{x}{G}$
  - Keep improving until it is good enough

$$
\begin{array}{lll}
x = 2 & G & = 1 \\
\frac{x}{G} = 2 & G = \frac{(1+2)}{2} & = 1.5
\end{array}
$$

Computation is Imperative Knowledge

- "How to" knowledge
- To approximate $\sqrt{x}$ (Heron's Method):
  - Make a guess G
  - Improve the guess by averaging G and $\frac{x}{G}$
  - Keep improving until it is good enough

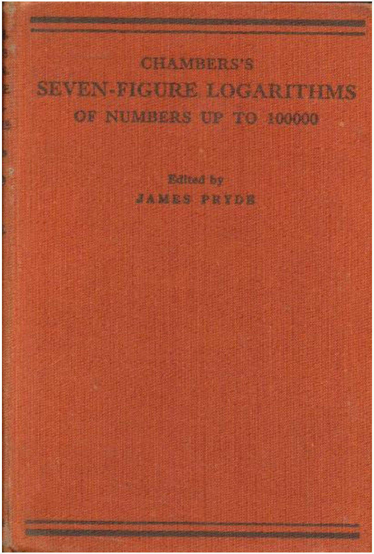
$$
\begin{array}{lll}
x = 2 & G & = 1 \\
\frac{x}{G} = 2 & G = \frac{(1+2)}{2} & = 1.5 \\
\frac{x}{G} = \frac{4}{3} & G = \frac{(\frac{3}{2}+\frac{4}{3})}{2} & = 1.4166
\end{array}
$$

Computation is Imperative Knowledge

- "How to" knowledge
- To approximate $\sqrt{x}$ (Heron's Method):
  - Make a guess G
  - Improve the guess by averaging G and $\frac{x}{G}$
  - Keep improving until it is good enough

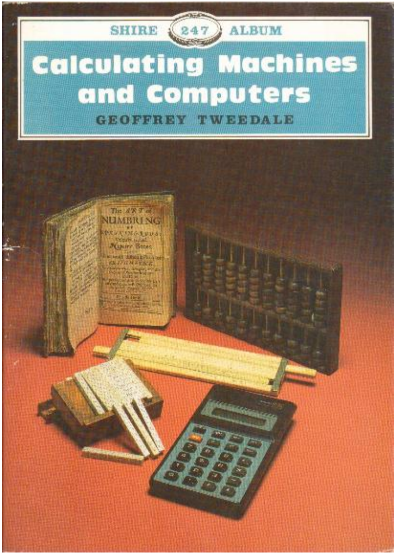
$$
\begin{array}{lll}
x = 2 & G & = 1 \\
\frac{x}{G} = 2 & G = \frac{(1+2)}{2} & = 1.5 \\
\frac{x}{G} = \frac{4}{3} & G = \frac{(\frac{3}{2}+\frac{4}{3})}{2} & = 1.4166 \\
\frac{x}{G} = \frac{24}{17} & G = \frac{(\frac{17}{12}+\frac{24}{17})}{2} & = 1.4142
\end{array}
$$

“How to” knowledge



- Could just store tons of “what is” information

“How to” knowledge



- Could just store tons of “what is” information
- Much more useful to capture “how to” knowledge – a series of steps to be followed to deduce a value – a procedure.

Describing “How to” knowledge

Need a language for describing processes:

- Vocabulary – basic primitives

Describing “How to” knowledge

Need a language for describing processes:

- Vocabulary – basic primitives
- Rules for writing compound expressions – syntax

Describing "How to" knowledge

Need a language for describing processes:

- Vocabulary – basic primitives
- Rules for writing compound expressions – syntax
- Rules for assigning meaning to constructs – semantics

Describing “How to” knowledge

Need a language for describing processes:

- Vocabulary – basic primitives
- Rules for writing compound expressions – syntax
- Rules for assigning meaning to constructs – semantics
- Rules for capturing process of evaluation – procedures

Representing basic information

- Numbers

Representing basic information

- Numbers
  - As floating point values

Representing basic information

- Numbers
  - As floating point values
  - In IEEE 754 format

Representing basic information

- Numbers
  - As floating point values
  - In IEEE 754 format
  - Stored in binary

Representing basic information

- Numbers
  - As floating point values
  - In IEEE 754 format
  - Stored in binary
  - In registers

Representing basic information

- Numbers
  - As floating point values
  - In IEEE 754 format
  - Stored in binary
  - In registers
  - Made up of bits

Representing basic information

- Numbers
  - As floating point values
  - In IEEE 754 format
  - Stored in binary
  - In registers
  - Made up of bits
  - Stored in flip-flops

Representing basic information

- Numbers
  - As floating point values
  - In IEEE 754 format
  - Stored in binary
  - In registers
  - Made up of bits
  - Stored in flip-flops
  - Made of logic gates

Representing basic information

- Numbers
    - As floating point values
    - In IEEE 754 format
    - Stored in binary
    - In registers
    - Made up of bits
    - Stored in flip-flops
    - Made of logic gates
    - Implemented by transistors

Representing basic information

- Numbers
  - As floating point values
  - In IEEE 754 format
  - Stored in binary
  - In registers
  - Made up of bits
  - Stored in flip-flops
  - Made of logic gates
  - Implemented by transistors
  - In silicon wells

Representing basic information

- Numbers
  - As floating point values
  - In IEEE 754 format
  - Stored in binary
  - In registers
  - Made up of bits
  - Stored in flip-flops
  - Made of logic gates
  - Implemented by transistors
  - In silicon wells
  - With electrical potential

Representing basic information

- Numbers
  - As floating point values
  - In IEEE 754 format
  - Stored in binary
  - In registers
  - Made up of bits
  - Stored in flip-flops
  - Made of logic gates
  - Implemented by transistors
  - In silicon wells
  - With electrical potential
  - Of individual electrons

Representing basic information

- Numbers
  - As floating point values
  - In IEEE 754 format
  - Stored in binary
  - In registers
  - Made up of bits
  - Stored in flip-flops
  - Made of logic gates
  - Implemented by transistors
  - In silicon wells
  - With electrical potential
  - Of individual electrons
  - With mass, charge, spin, and chirality

Representing basic information

- Numbers
  - As floating point values
  - In IEEE 754 format
  - Stored in binary
  - In registers
  - Made up of bits
  - Stored in flip-flops
  - Made of logic gates
  - Implemented by transistors
  - In silicon wells
  - With electrical potential
  - Of individual electrons
  - With mass, charge, spin, and chirality
  - Whose mass is imparted by interaction with the Higgs field

Representing basic information

- Numbers
    - As floating point values
    - In IEEE 754 format
    - Stored in binary
    - In registers
    - Made up of bits
    - Stored in flip-flops
    - Made of logic gates
    - Implemented by transistors
    - In silicon wells
    - With electrical potential
    - Of individual electrons
    - With mass, charge, spin, and chirality
    - Whose mass is imparted by interaction with the Higgs field
    - ...

Assuming a basic level of abstraction

- We assume that our language provides us with a basic set of data elements:
  - Numbers

Assuming a basic level of abstraction

- We assume that our language provides us with a basic set of data elements:
  - Numbers
  - Characters

Assuming a basic level of abstraction

- We assume that our language provides us with a basic set of data elements:
  - Numbers
  - Characters
  - Booleans

Assuming a basic level of abstraction

- We assume that our language provides us with a basic set of data elements:
  - Numbers
  - Characters
  - Booleans
- It also provides a basic set of operations on these primitive elements

Assuming a basic level of abstraction

- We assume that our language provides us with a basic set of data elements:
  - Numbers
  - Characters
  - Booleans
- It also provides a basic set of operations on these primitive elements
- We can then focus on using these basic elements to construct more complex processes

Rules for describing processes in Scheme

- Legal expressions have rules for constructing from simpler pieces – the syntax.

Rules for describing processes in Scheme

- Legal expressions have rules for constructing from simpler pieces – the syntax.
- (Almost) every expression has a value, which is “returned” when an expression is “evaluated.”

Rules for describing processes in Scheme

- Legal expressions have rules for constructing from simpler pieces – the syntax.
- (Almost) every expression has a value, which is “returned” when an expression is “evaluated.”
- Every value has a type.

Rules for describing processes in Scheme

- Legal expressions have rules for constructing from simpler pieces – the syntax.
- (Almost) every expression has a value, which is “returned” when an expression is “evaluated.”
- Every value has a type.
- The latter two are the semantics of the language.

Language elements – primitives

Self-evaluating primitives – value of expression is just object itself:

Numbers 29, -35, 1.34, $1.2e5$

Language elements – primitives

Self-evaluating primitives – value of expression is just object itself:

Numbers 29, -35, 1.34, $1.2e5$

Strings “this is a string” “odd #$@%#$ thing number 35”

Language elements – primitives

Self-evaluating primitives – value of expression is just object itself:

Numbers 29, -35, 1.34, $1.2e5$

Strings "this is a string" "odd #$@%#$ thing number 35"

Booleans `#t`, `#f`

Language elements – primitives

Built-in procedures to manipulate primitive objects:

Numbers	`+`, `−`, `*`, `/`, `>`, `<`, `>=`, `<=`, `=`
Strings	`string-length`, `string=?`
Booleans	`and`, `or`, `not`

Language elements – primitives

Names for built-in procedures

- $+, -, *, /, =, \ldots$
- What is the value of them?

Language elements – primitives

Names for built-in procedures

- +, −, ∗, /, =, ...
- What is the value of them?
- + $\rightarrow$ `#<procedure:+>`

Language elements – primitives

Names for built-in procedures

- `+`, `−`, `*`, `/`, `=`, ...
- What is the value of them?
- `+` $\rightarrow$ `#<procedure:+>`
- Evaluate by looking up value associated with the name in a special table – the environment.

Language elements – combinations

- How to we create expressions using these procedures?
- `(+ 2 3)`

Language elements – combinations

- How to we create expressions using these procedures?
- `(+ 2 3)`
    - Open paren

Language elements – combinations

- How to we create expressions using these procedures?
- `(+ 2 3)`
  - Open paren
  - Expression whose value is a procedure

Language elements – combinations

- How to we create expressions using these procedures?
- `(+ 2 3)`
  - Open paren
  - Expression whose value is a procedure
  - Other expressions

Language elements – combinations

- How to we create expressions using these procedures?
- `(+ 2 3)`
  - Open paren
  - Expression whose value is a procedure
  - Other expressions
  - Close paren

Language elements – combinations

- How to we create expressions using these procedures?
- `(+ 2 3)`
  - Open paren
  - Expression whose value is a procedure
  - Other expressions
  - Close paren
- This type of expression is called a combination
- Evaluate it by getting values of sub-expressions, then applying operator to values of arguments.

Language elements – combinations

- How to we create expressions using these procedures?
- `(+ 2 3)`
  - Open paren
  - Expression whose value is a procedure
  - Other expressions
  - Close paren
- This type of expression is called a <u>combination</u>
- Evaluate it by getting values of sub-expressions, then applying operator to values of arguments.
- You now know all there is to know about Scheme syntax!

Language elements – combinations

- How to we create expressions using these procedures?
- `(+ 2 3)`
  - Open paren
  - Expression whose value is a procedure
  - Other expressions
  - Close paren
- This type of expression is called a combination
- Evaluate it by getting values of sub-expressions, then applying operator to values of arguments.
- You now know all there is to know about Scheme syntax! (almost)

Language elements – combinations

- Note the recursive definition – can use combinations as expressions to other combinations:

```
(+ (* 2 3) 4)
```

$\rightarrow$

Language elements – combinations

- Note the recursive definition – can use combinations as expressions to other combinations:

`(+ (* 2 3) 4)` $\rightarrow$ `10`

Language elements – combinations

- Note the recursive definition – can use combinations as expressions to other combinations:

```
(+ (* 2 3) 4)           →    10
(* (+ 3 4) (- 8 2))     →
```

Language elements – combinations

- Note the recursive definition – can use combinations as expressions to other combinations:

```
(+ (* 2 3) 4)           →    10
(* (+ 3 4) (- 8 2))     →    42
```

Language elements – abstractions

- In order to abstract an expression, need a way to give it a name

Language elements – abstractions

- In order to abstract an expression, need a way to give it a name
  `(define score 23)`

Language elements – abstractions

- In order to abstract an expression, need a way to give it a name
  `(define score 23)`
- This is a special form
  - Does not evaluate the second expression
  - Rather, it pairs the name with the value of the third expression

Language elements – abstractions

- In order to abstract an expression, need a way to give it a name
  `(define score 23)`
- This is a special form
  - Does not evaluate the second expression
  - Rather, it pairs the name with the value of the third expression
- The return value is unspecified

Language elements – abstractions

- To get the value of a name, just look up pairing in the environment

```
(define score 23)
```
$\rightarrow$

Language elements – abstractions

- To get the value of a name, just look up pairing in the environment

```
(define score 23)
```
$\rightarrow$ `undefined`

Language elements – abstractions

- To get the value of a name, just look up pairing in the environment

`(define score 23)`	$\rightarrow$	`undefined`
`score`	$\rightarrow$	

Language elements – abstractions

- To get the value of a name, just look up pairing in the environment

`(define score 23)`	$\rightarrow$	`undefined`
`score`	$\rightarrow$	`23`

Language elements – abstractions

- To get the value of a name, just look up pairing in the environment

`(define score 23)`	$\rightarrow$	`undefined`
`score`	$\rightarrow$	`23`
`(define total (+ 12 13))`	$\rightarrow$	

Language elements – abstractions

- To get the value of a name, just look up pairing in the environment

`(define score 23)`	$\rightarrow$	`undefined`
`score`	$\rightarrow$	`23`
`(define total (+ 12 13))`	$\rightarrow$	`undefined`

Language elements – abstractions

- To get the value of a name, just look up pairing in the environment

`(define score 23)`	$\rightarrow$	`undefined`
`score`	$\rightarrow$	`23`
`(define total (+ 12 13))`	$\rightarrow$	`undefined`
`(* 100 (/ score total))`	$\rightarrow$	

Language elements – abstractions

- To get the value of a name, just look up pairing in the environment

`(define score 23)`	$\rightarrow$	`undefined`
`score`	$\rightarrow$	`23`
`(define total (+ 12 13))`	$\rightarrow$	`undefined`
`(* 100 (/ score total))`	$\rightarrow$	`92`

Language elements – common errors

```
(5 + 6)
```

Language elements – common errors

```
(5 + 6)
    => procedure application: expected procedure,
       given: 5; arguments were: #<procedure:+> 6
```

Language elements – common errors

```
(5 + 6)
    => procedure application: expected procedure,
       given: 5; arguments were: #<procedure:+> 6

((+ 5 6))
```

Language elements – common errors

```
(5 + 6)
    => procedure application: expected procedure,
       given: 5; arguments were: #<procedure:+> 6

((+ 5 6))
    => procedure application: expected procedure,
       given: 11 (no arguments)
```

Language elements – common errors

```
(5 + 6)
    => procedure application: expected procedure,
       given: 5; arguments were: #<procedure:+> 6

((+ 5 6))
    => procedure application: expected procedure,
       given: 11 (no arguments)

(* 100 (/ score totla))
```

Language elements – common errors

```
(5 + 6)
    => procedure application: expected procedure,
       given: 5; arguments were: #<procedure:+> 6

((+ 5 6))
    => procedure application: expected procedure,
       given: 11 (no arguments)

(* 100 (/ score totla))
    => reference to undefined identifier: totla
```

Scheme basics

Rules for evaluation:

- If self-evaluating, return value

Scheme basics

Rules for evaluation:

- If self-evaluating, return value
- If a name, return value associated with name in environment

Scheme basics

Rules for evaluation:

- If self-evaluating, return value
- If a name, return value associated with name in environment
- If a special form, do something special

Scheme basics

Rules for evaluation:

- If self-evaluating, return value
- If a name, return value associated with name in environment
- If a special form, do something special
- If a combination, then
  - Evaluate all of the sub-expressions, in any order

Scheme basics

Rules for evaluation:

- If self-evaluating, return value
- If a name, return value associated with name in environment
- If a special form, do something special
- If a combination, then
  - Evaluate all of the sub-expressions, in any order
  - Apply the operator to the values of the operands and return the result

Mathematical operators are just names

`(+ 3 5)` $\rightarrow$ 8

Mathematical operators are just names

`(+ 3 5)`	$\rightarrow$	8
`(define fred +)`	$\rightarrow$	undefined

Mathematical operators are just names

`(+ 3 5)`	$\rightarrow$	8
`(define fred +)`	$\rightarrow$	undefined
`(fred 3 6)`	$\rightarrow$	9

Mathematical operators are just names

`(+ 3 5)`	$\rightarrow$	8
`(define fred +)`	$\rightarrow$	undefined
`(fred 3 6)`	$\rightarrow$	9

- $+$ is just a name
- $+$ is bound to a value which is a procedure
- line 2 binds the name `fred` to that same value

All names are names

`(+ 3 5)` $\rightarrow$ 8

All names are names

`(+ 3 5)`	→	8
`(define + *)`	→	undefined

All names are names

`(+ 3 5)`	$\rightarrow$	8
`(define + *)`	$\rightarrow$	undefined
`(+ 3 5)`	$\rightarrow$	

All names are names

`(+ 3 5)`	$\rightarrow$	8
`(define + *)`	$\rightarrow$	undefined
`(+ 3 5)`	$\rightarrow$	15

All names are names

`(+ 3 5)`	→	8
`(define + *)`	→	undefined
`(+ 3 5)`	→	15

- There's nothing "special" about the operators you take for granted, either!

All names are names

`(+ 3 5)`	$\rightarrow$	8
`(define + *)`	$\rightarrow$	undefined
`(+ 3 5)`	$\rightarrow$	15

- There's nothing "special" about the operators you take for granted, either!
- Their values can be changed using `define` just as well

All names are names

`(+ 3 5)`	$\rightarrow$	8
`(define + *)`	$\rightarrow$	undefined
`(+ 3 5)`	$\rightarrow$	15

- There's nothing "special" about the operators you take for granted, either!
- Their values can be changed using `define` just as well
- Of course, this is generally a horrible idea

Making our own procedures

- To capture a way of doing things, create a procedure:

Making our own procedures

- To capture a way of doing things, create a procedure:
  `(lambda (x) (* x x))`

Making our own procedures

- To capture a way of doing things, create a procedure:
  `(lambda (x) (* x x))`
- `(x)` is the list of parameters

Making our own procedures

- To capture a way of doing things, create a procedure:
  `(lambda (x) (* x x))`
- `(x)` is the list of parameters
- `(* x x)` is the body

Making our own procedures

- To capture a way of doing things, create a procedure:
  `(lambda (x) (* x x))`
- `(x)` is the list of parameters
- `(* x x)` is the body
- `lambda` is a special form: create a procedure and returns it

Substitution

- Use this anywhere you would use a built-in procedure like $+$:
  `( (lambda (x) (* x x)) 5 )`

Substitution

- Use this anywhere you would use a built-in procedure like +:
  `( (lambda (x) (* x x)) 5 )`
- Substitute the value of the provided arguments into the body:
  `(* 5 5)`

Substitution

- Use this anywhere you would use a built-in procedure like +:
  `( (lambda (x) (* x x)) 5 )`
- Substitute the value of the provided arguments into the body:
  `(* 5 5)`
- Can also give it a name:
  `(define square (lambda(x) (* x x)))`
  `(square 5)` $\rightarrow$ `25`

Substitution

- Use this anywhere you would use a built-in procedure like $+$:
  `( (lambda (x) (* x x)) 5 )`
- Substitute the value of the provided arguments into the body:
  `(* 5 5)`
- Can also give it a name:
  `(define square (lambda(x) (* x x)))`
  `(square 5)` $\rightarrow$ `25`
- This creates a loop in our system, where we can create a complex thing, name it, and treat it as a primitive like $+$

Scheme basics

Rules for evaluation:

- If self-evaluating, return value
- If a name, return value associated with name in environment
- If a special form, do something special.
- If a combination, then
  - Evaluate all of the sub-expressions, in any order
  - Apply the operator to the values of the operands and return the result

Rules for applying:

- If primitive, just do it
- If a compound procedure, then substitute each formal parameter with the corresponding argument value, and evaluate the body

Interaction of define and lambda

```
(lambda (x) (* x x))
    => #<procedure>
```

Interaction of `define` and `lambda`

```
(lambda (x) (* x x))
    => #<procedure>
(define square (lambda (x) (* x x)))
    => undefined
```

Interaction of `define` and `lambda`

```
(lambda (x) (* x x))
    => #<procedure>
(define square (lambda (x) (* x x)))
    => undefined
(square 4)
```

Interaction of define and lambda

```
(lambda (x) (* x x))
    => #<procedure>
(define square (lambda (x) (* x x)))
    => undefined
(square 4)
    => (* 4 4)
```

Interaction of `define` and `lambda`

```
(lambda (x) (* x x))
    => #<procedure>
(define square (lambda (x) (* x x)))
    => undefined
(square 4)
    => (* 4 4)
    => 16
```

Interaction of `define` and `lambda`

```
(lambda (x) (* x x))
    => #<procedure>
(define square (lambda (x) (* x x)))
    => undefined
(square 4)
    => (* 4 4)
    => 16
```

"Syntactic sugar":

```
(define (square x) (* x x))
    => undefined
```

Lambda special form

- Syntax: `(lambda (x y) (/ (+ x y) 2))`

Lambda special form

- Syntax: `(lambda (x y) (/ (+ x y) 2))`
- 1st operand is the parameter list: `(x y)`
  - a list of names (perhaps empty)
  - determines the number of operands required

Lambda special form

- Syntax: `(lambda (x y) (/ (+ x y) 2))`
- 1st operand is the parameter list: `(x y)`
  - a list of names (perhaps empty)
  - determines the number of operands required
- 2nd operand is the body: `(/ (+ x y) 2)`
  - may be any expression
  - not evaluated when the lambda is evaluated
  - evaluated when the procedure is applied

Meaning of a lambda

```
(define x (lambda () (+ 3 2)))
```

$\rightarrow$

Meaning of a lambda

`(define x (lambda () (+ 3 2)))` $\rightarrow$ undefined

Meaning of a lambda

```
(define x (lambda () (+ 3 2)))
x
```

$\rightarrow$ undefined

$\rightarrow$

Meaning of a lambda

`(define x (lambda () (+ 3 2)))`	$\rightarrow$	undefined
`x`	$\rightarrow$	`#<procedure>`

Meaning of a lambda

`(define x (lambda () (+ 3 2)))`	$\rightarrow$	undefined
`x`	$\rightarrow$	`#<procedure>`
`(x)`	$\rightarrow$	

Meaning of a lambda

`(define x (lambda () (+ 3 2)))`	$\rightarrow$	undefined
`x`	$\rightarrow$	`#<procedure>`
`(x)`	$\rightarrow$	5

Meaning of a lambda

(define x (lambda () (+ 3 2)))	→	undefined
x	→	#<procedure>
(x)	→	5

The value of a lambda expression is a procedure

What does a procedure describe?

Capturing a common pattern:

- `(* 3 3)`
- `(* 25 25)`
- `(* foobar foobar)`

What does a procedure describe?

Capturing a common pattern:

- `(* 3 3)`
- `(* 25 25)`
- `(* foobar foobar)`

`(lambda (x) (* x x))`

Name for the thing that changes

What does a procedure describe?

Capturing a common pattern:

- `(* 3 3)`
- `(* 25 25)`
- `(* foobar foobar)`

```
(lambda (x) (* x x))
```

Common pattern to capture

Modularity of common patterns

Here is a common pattern:

Modularity of common patterns

Here is a common pattern:

- `(sqrt (+ (* 3 3) (* 4 4)))`

Modularity of common patterns

Here is a common pattern:

- `(sqrt (+ (* 3 3) (* 4 4)))`
- `(sqrt (+ (* 9 9) (* 16 16)))`

Modularity of common patterns

Here is a common pattern:

- `(sqrt (+ (* 3 3) (* 4 4)))`
- `(sqrt (+ (* 9 9) (* 16 16)))`
- `(sqrt (+ (* 4 4) (* 4 4)))`

Modularity of common patterns

Here is a common pattern:

- `(sqrt (+ (* 3 3) (* 4 4)))`
- `(sqrt (+ (* 9 9) (* 16 16)))`
- `(sqrt (+ (* 4 4) (* 4 4)))`

Modularity of common patterns

Here is a common pattern:

- `(sqrt (+ (* 3 3) (* 4 4)))`
- `(sqrt (+ (* 9 9) (* 16 16)))`
- `(sqrt (+ (* 4 4) (* 4 4)))`

Modularity of common patterns

Here is a common pattern:

- `(sqrt (+ (* 3 3) (* 4 4)))`
- `(sqrt (+ (* 9 9) (* 16 16)))`
- `(sqrt (+ (* 4 4) (* 4 4)))`

Here is one way to capture this pattern:

```
(define pythagoras
    (lambda (x y)
        (sqrt (+ (* x x) (* y y)))))
```

Modularity of common patterns

Here is a common pattern:

- `(sqrt (+ (* 3 3) (* 4 4)))`
- `(sqrt (+ (* 9 9) (* 16 16)))`
- `(sqrt (+ (* 4 4) (* 4 4)))`

Here is a better way to capture this pattern:

```
(define square (lambda (x) (* x x)))
(define pythagoras
    (lambda (x y)
        (sqrt (+ (square x) (square y)))))
```

Why?

- Breaking computation into modules that capture commonality

Why?

- Breaking computation into modules that capture commonality
- Enables reuse in other places (e.g. `square`)

Why?

- Breaking computation into modules that capture commonality
- Enables reuse in other places (e.g. `square`)
- Isolates (abstracts away) details of computation within a procedure from use of the procedure

Why?

- Breaking computation into modules that capture commonality
- Enables reuse in other places (e.g. `square`)
- Isolates (abstracts away) details of computation within a procedure from use of the procedure
- May be many ways to divide up:

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

```
(define pythagoras
    (lambda (x y)
        (sqrt (+ (square x) (square y)))))
```

Why?

- Breaking computation into modules that capture commonality
- Enables reuse in other places (e.g. `square`)
- Isolates (abstracts away) details of computation within a procedure from use of the procedure
- May be many ways to divide up:

```
(define square (lambda (x) (* x x)))
(define sum-squares
    (lambda (x y) (+ (square x) (square y))))
(define pythagoras
    (lambda (x y)
        (sqrt (sum-squares x y))))
```

A more complex example

To approximate $\sqrt{x}$:

1. Make a guess G

A more complex example

To approximate $\sqrt{x}$:

1. Make a guess G
2. Improve the guess by averaging G and $\frac{x}{G}$:

A more complex example

To approximate $\sqrt{x}$:

1. Make a guess G
2. Improve the guess by averaging G and $\frac{x}{G}$:
3. Keep improving until it is good enough

A more complex example

To approximate $\sqrt{x}$:

1. Make a guess G
2. Improve the guess by averaging G and $\frac{x}{G}$:
3. Keep improving until it is good enough

Sub-problems:

- When is “close enough”?

A more complex example

To approximate $\sqrt{x}$:

1. Make a guess G
2. Improve the guess by averaging G and $\frac{x}{G}$:
3. Keep improving until it is good enough

Sub-problems:

- When is “close enough”?
- How do we create a new guess?

A more complex example

To approximate $\sqrt{x}$:

1. Make a guess G
2. Improve the guess by averaging G and $\frac{x}{G}$:
3. Keep improving until it is good enough

Sub-problems:

- When is “close enough”?
- How do we create a new guess?
- How do we control the process of using the new guess in place of the old one?

Procedural abstractions

"When the square of the guess is within 0.001 of the value"

Procedural abstractions

"When the square of the guess is within 0.001 of the value"

```
(define close-enough?
    (lambda (guess x)
        (< (abs (- (square guess) x))
           0.001)))
```

Procedural abstractions

"When the square of the guess is within 0.001 of the value"

```
(define close-enough?
    (lambda (guess x)
        (< (abs (- (square guess) x))
           0.001)))
```

Note the use of the `square` procedural abstraction from earlier!

Procedural abstractions

```
(define average
    (lambda (a b) (/ (+ a b) 2)))
```

Procedural abstractions

```
(define average
    (lambda (a b) (/ (+ a b) 2)))

(define improve
    (lambda (guess x)
        (average guess (/ x guess))))
```

Why this modularity?

- `average` is something we are likely to want to use again
- Abstraction lets us separate implementation details from use
  - Originally:

```
(define average
     (lambda (a b) (/ (+ a b) 2)))
```

Why this modularity?

- `average` is something we are likely to want to use again
- Abstraction lets us separate implementation details from use
  - Originally:

    ```
    (define average
        (lambda (a b) (/ (+ a b) 2)))
    ```

  - Could redefine as:

    ```
    (define average
        (lambda (x y) (* (+ x y) 0.5)))
    ```

Why this modularity?

- `average` is something we are likely to want to use again
- Abstraction lets us separate implementation details from use
  - Originally:

    ```
    (define average
         (lambda (a b) (/ (+ a b) 2)))
    ```

  - Could redefine as:

    ```
    (define average
         (lambda (x y) (* (+ x y) 0.5)))
    ```

  - There's actually a difference between those in Racket (exact vs inexact numbers)

Why this modularity?

- `average` is something we are likely to want to use again
- Abstraction lets us separate implementation details from use
  - Originally:

    ```
    (define average
        (lambda (a b) (/ (+ a b) 2)))
    ```

  - Could redefine as:

    ```
    (define average
        (lambda (x y) (* (+ x y) 0.5)))
    ```

  - There's actually a difference between those in Racket (exact vs inexact numbers)
  - No other changes needed to procedures that use `average`

Why this modularity?

- `average` is something we are likely to want to use again
- Abstraction lets us separate implementation details from use
  - Originally:

    ```
    (define average
        (lambda (a b) (/ (+ a b) 2)))
    ```

  - Could redefine as:

    ```
    (define average
        (lambda (x y) (* (+ x y) 0.5)))
    ```

  - There's actually a difference between those in Racket (exact vs inexact numbers)
  - No other changes needed to procedures that use `average`
  - Also note that parameters are internal to the procedure – cannot be referred to by name outside of the lambda

Controlling the process

- Given `x` and `guess`, want `(improve guess x)` as new guess

Controlling the process

- Given `x` and `guess`, want `(improve guess x)` as new guess
- But only if the guess isn't good enough already

Controlling the process

- Given `x` and `guess`, want `(improve guess x)` as new guess
- But only if the guess isn't good enough already
- We need to make a decision – for this, we need a new special form

  `(if predicate consequent alternative)`

The `if` special form

```
(if predicate consequent alternative)
```

- Evaluator first evaluates the `predicate` expression
- If it returns a true value (`#t`), then the evaluator evaluates and returns the value of the `consequent` expression
- Otherwise, it evaluates and returns the value of the `alternative` expression

The `if` special form

`(if predicate consequent alternative)`

- Evaluator first evaluates the `predicate` expression
- If it returns a true value (`#t`), then the evaluator evaluates and returns the value of the `consequent` expression
- Otherwise, it evaluates and returns the value of the `alternative` expression
- Why must this be a special form? Why can't it be implemented as a regular `lambda` procedure?

Using `if`

- So the heart of the process should be:

```
(define (sqrt-loop guess x)
    (if (close-enough? guess x)
        guess
                    (improve guess x)    ))
```

- But somehow we need to use the value returned by `improve` as the new `guess`, keep the same `x`, and repeat the process

Using `if`

- So the heart of the process should be:

```
(define (sqrt-loop guess x)
    (if (close-enough? guess x)
        guess
                   (improve guess x)    ))
```

- But somehow we need to use the value returned by `improve` as the new `guess`, keep the same `x`, and repeat the process
- Call the `sqrt-loop` function again and reuse it!

Using `if`

- So the heart of the process should be:

```
(define (sqrt-loop guess x)
    (if (close-enough? guess x)
        guess
        (sqrt-loop (improve guess x) x)))
```

- But somehow we need to use the value returned by `improve` as the new `guess`, keep the same `x`, and repeat the process
- Call the `sqrt-loop` function again and reuse it!

Putting it together

Now we just need to kick the process off with an initial guess:

```
(define sqrt
    (lambda (x)
        (sqrt-loop 1.0 x)))

(define (sqrt-loop guess x)
    (if (close-enough? guess x)
        guess
        (sqrt-loop (improve guess x) x)))
```

Testing the code

- How do we know it works?

Testing the code

- How do we know it works?
- Fall back to rules for evaluation from earlier

Substitution model

Rules for evaluation:

- If self-evaluating, return value
- If a name, return value associated with name in environment
- If a special form, do something special.
- If a combination, then
  - Evaluate all of the sub-expressions, in any order
  - Apply the operator to the values of the operands and return the result

Rules for applying:

- If primitive, just do it
- If a compound procedure, then substitute each formal parameter with the corresponding argument value, and evaluate the body

Substitution model

Rules for evaluation:

- If self-evaluating, return value
- If a name, return value associated with name in environment
- If a special form, do something special.
- If a combination, then
  - Evaluate all of the sub-expressions, in any order
  - Apply the operator to the values of the operands and return the result

Rules for applying:

- If primitive, just do it
- If a compound procedure, then substitute each formal parameter with the corresponding argument value, and evaluate the body

The substitution model of evaluation

Substitution model

Rules for evaluation:

- If self-evaluating, return value
- If a name, return value associated with name in environment
- If a special form, do something special.
- If a combination, then
  - Evaluate all of the sub-expressions, in any order
  - Apply the operator to the values of the operands and return the result

Rules for applying:

- If primitive, just do it
- If a compound procedure, then substitute each formal parameter with the corresponding argument value, and evaluate the body

The substitution model of evaluation

. . . is a lie and a simplification, but a useful one!

```
(sqrt 2)
```

```
(sqrt 2)
((lambda (x) (sqrt-loop 1.0 x)) 2)
```

```
(sqrt 2)
((lambda (x) (sqrt-loop 1.0 x)) 2)
```

```
(sqrt 2)
((lambda (x) (sqrt-loop 1.0 x)) 2)
(sqrt-loop 1.0 2)
```

```
(sqrt 2)
((lambda (x) (sqrt-loop 1.0 x)) 2)
(sqrt-loop 1.0 2)
((lambda (guess x)
    (if (close-enough? guess x)
        guess
        (sqrt-loop (improve guess x) x))) 1.0 2)
```

```
(sqrt 2)
((lambda (x) (sqrt-loop 1.0 x)) 2)
(sqrt-loop 1.0 2)
((lambda (guess x)
    (if (close-enough? guess x)
        guess
        (sqrt-loop (improve guess x) x))) 1.0 2)
```

```
(sqrt 2)
((lambda (x) (sqrt-loop 1.0 x)) 2)
(sqrt-loop 1.0 2)
((lambda (guess x)
    (if (close-enough? guess x)
        guess
        (sqrt-loop (improve guess x) x))) 1.0 2)
(if (close-enough? 1.0 2)
    1.0
    (sqrt-loop (improve 1.0 2) 2))
```

```
(sqrt 2)
((lambda (x) (sqrt-loop 1.0 x)) 2)
(sqrt-loop 1.0 2)
((lambda (guess x)
    (if (close-enough? guess x)
        guess
        (sqrt-loop (improve guess x) x))) 1.0 2)
(if (close-enough? 1.0 2)
    1.0
    (sqrt-loop (improve 1.0 2) 2))
(sqrt-loop (improve 1.0 2) 2)
```

```
(sqrt 2)
((lambda (x) (sqrt-loop 1.0 x)) 2)
(sqrt-loop 1.0 2)
((lambda (guess x)
    (if (close-enough? guess x)
        guess
        (sqrt-loop (improve guess x) x))) 1.0 2)
(if (close-enough? 1.0 2)
    1.0
    (sqrt-loop (improve 1.0 2) 2))
(sqrt-loop (improve 1.0 2) 2)
(sqrt-loop ((lambda (a b) (/ (+ a b) 2)) 1.0 2) 2)
```

```
(sqrt 2)
((lambda (x) (sqrt-loop 1.0 x)) 2)
(sqrt-loop 1.0 2)
((lambda (guess x)
    (if (close-enough? guess x)
        guess
        (sqrt-loop (improve guess x) x))) 1.0 2)
(if (close-enough? 1.0 2)
    1.0
    (sqrt-loop (improve 1.0 2) 2))
(sqrt-loop (improve 1.0 2) 2)
(sqrt-loop ((lambda (a b) (/ (+ a b) 2)) 1.0 2) 2)
```

```
(sqrt 2)
((lambda (x) (sqrt-loop 1.0 x)) 2)
(sqrt-loop 1.0 2)
((lambda (guess x)
    (if (close-enough? guess x)
        guess
        (sqrt-loop (improve guess x) x))) 1.0 2)
(if (close-enough? 1.0 2)
    1.0
    (sqrt-loop (improve 1.0 2) 2))
(sqrt-loop (improve 1.0 2) 2)
(sqrt-loop ((lambda (a b) (/ (+ a b) 2)) 1.0 2) 2)
(sqrt-loop (/ (+ 1.0 2) 2) 2)
```

```
(sqrt 2)
((lambda (x) (sqrt-loop 1.0 x)) 2)
(sqrt-loop 1.0 2)
((lambda (guess x)
    (if (close-enough? guess x)
        guess
        (sqrt-loop (improve guess x) x))) 1.0 2)
(if (close-enough? 1.0 2)
    1.0
    (sqrt-loop (improve 1.0 2) 2))
(sqrt-loop (improve 1.0 2) 2)
(sqrt-loop ((lambda (a b) (/ (+ a b) 2)) 1.0 2) 2)
(sqrt-loop (/ (+ 1.0 2) 2) 2)
(sqrt-loop 1.5 2)
```

```
(sqrt 2)
((lambda (x) (sqrt-loop 1.0 x)) 2)
(sqrt-loop 1.0 2)
((lambda (guess x)
    (if (close-enough? guess x)
        guess
        (sqrt-loop (improve guess x) x))) 1.0 2)
(if (close-enough? 1.0 2)
    1.0
    (sqrt-loop (improve 1.0 2) 2))
(sqrt-loop (improve 1.0 2) 2)
(sqrt-loop ((lambda (a b) (/ (+ a b) 2)) 1.0 2) 2)
(sqrt-loop (/ (+ 1.0 2) 2) 2)
(sqrt-loop 1.5 2)
...
(sqrt-loop 1.4166 2)
...
```

A canonical example

- 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?

Recursive algorithms

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

Recursive algorithms

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

Wishful thinking

- Assume the desired procedure exists

Recursive algorithms

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

Wishful thinking

- Assume the desired procedure exists
- Want to implement `factorial`? Assume it exists.

Recursive algorithms

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

Wishful thinking

- Assume the desired procedure exists
- Want to implement `factorial`? Assume it exists.
- But, it only solves a smaller version of the problem

Recursive algorithms

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

Wishful thinking

- Assume the desired procedure exists
- Want to implement `factorial`? Assume it exists.
- But, it only solves a smaller version of the problem
- This is just finding the common pattern; but here, solving the bigger problem involves the same pattern in a smaller problem

Recursive algorithms

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

Decompose the problem

- Solve a smaller instance

Recursive algorithms

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

Decompose the problem

- Solve a smaller instance
- Convert that solution into desired solution
  $n! = n(n-1)(n-2)\ldots = n[(n-1)(n-2)\ldots] = n * (n-1)!$

Recursive algorithms

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

Decompose the problem

- Solve a smaller instance
- Convert that solution into desired solution
  $n! = n(n-1)(n-2)\ldots = n[(n-1)(n-2)\ldots] = n*(n-1)!$

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

Minor difficulty

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

Minor difficulty

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

(fact 2)
```

Minor difficulty

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

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

Minor difficulty

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

(fact 2)
(* 2 (fact 1))
(* 2 (* 1 (fact 0)))
```

Minor difficulty

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

(fact 2)
(* 2 (fact 1))
(* 2 (* 1 (fact 0)))
(* 2 (* 1 (* 0 (fact -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))))
(* 2 (* 1 (* 0 (* -1 (fact -2)))))
```

Minor difficulty

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

(fact 2)
(* 2 (fact 1))
(* 2 (* 1 (fact 0)))
(* 2 (* 1 (* 0 (fact -1))))
(* 2 (* 1 (* 0 (* -1 (fact -2)))))
⋮
```

Recursive algorithms

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

Identify non-decomposable problems

- Must identify the “smallest” problems and solve explicitly

Recursive algorithms

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

Identify non-decomposable problems

- Must identify the “smallest” problems and solve explicitly
- Define 1! to be 1

Recursive algorithms

- Have a test, a base case, and a recursive case

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

Recursive algorithms

- Have a test, a base case, and a recursive case

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

Recursive algorithms

- Have a test, a base case, and a recursive case

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

Recursive algorithms

- Have a test, a base case, and a recursive case

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

Recursive algorithms

- Have a test, a base case, and a recursive case

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

- More complex algorithms may have multiple base cases or multiple recursive cases

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

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

```
(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))))
```

```
(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)))
```

```
(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))))))
```

```
(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))))))
```

```
(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))
```

```
(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)
```

```
(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)
(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
```

Effects of recursive algorithms

Recursive algorithms consume more space with bigger operands!

```
(fact 4)
```

Effects of recursive algorithms

Recursive algorithms consume more space with bigger operands!

```
(fact 4)
(* 4 (fact 3))
```

Effects of recursive algorithms

Recursive algorithms consume more space with bigger operands!

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

Effects of recursive algorithms

Recursive algorithms consume more space with bigger operands!

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

Effects of recursive algorithms

Recursive algorithms consume more space with bigger operands!

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

Effects of recursive algorithms

Recursive algorithms consume more space with bigger operands!

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

Effects of recursive algorithms

Recursive algorithms consume more space with bigger operands!

```
(fact 8)
```

Effects of recursive algorithms

Recursive algorithms consume more space with bigger operands!

```
(fact 8)
(* 8 (fact 7))
```

Effects of recursive algorithms

Recursive algorithms consume more space with bigger operands!

```
(fact 8)
(* 8 (fact 7))
(* 8 (* 7 (fact 6)))
```

Effects of recursive algorithms

Recursive algorithms consume more space with bigger operands!

```
(fact 8)
(* 8 (fact 7))
(* 8 (* 7 (fact 6)))
(* 8 (* 7 (* 6 (fact 5))))
```

Effects of recursive algorithms

Recursive algorithms consume more space with bigger operands!

```
(fact 8)
(* 8 (fact 7))
(* 8 (* 7 (fact 6)))
(* 8 (* 7 (* 6 (fact 5))))
```

...

Effects of recursive algorithms

Recursive algorithms consume more space with bigger operands!

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

Effects of recursive algorithms

Recursive algorithms consume more space with bigger operands!

```
(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)))))))
```

Effects of recursive algorithms

Recursive algorithms consume more space with bigger operands!

```
(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))))))
```

Effects of recursive algorithms

Recursive algorithms consume more space with bigger operands!

```
(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))))))
```

...

Effects of recursive algorithms

Recursive algorithms consume more space with bigger operands!

```
(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
```

An alternative

- Try computing 101!
  $101 * 100 * 99 * 98 * 97 * 96 * \ldots * 2 * 1$
- How much space do we consume with pending operations?

An alternative

- Try computing 101!
  $101 * 100 * 99 * 98 * 97 * 96 * \ldots * 2 * 1$
- How much space do we consume with pending operations?
- Better idea: count up, doing one multiplication at a time
  - Start with 1 as the answer

An alternative

- Try computing 101!
  $101 * 100 * 99 * 98 * 97 * 96 * \ldots * 2 * 1$
- How much space do we consume with pending operations?
- Better idea: count up, doing one multiplication at a time
  - Start with 1 as the answer
  - Multiply by 2, store 2 as the current answer, remember we've done up to 2

An alternative

- Try computing 101!
  $101 * 100 * 99 * 98 * 97 * 96 * \ldots * 2 * 1$
- How much space do we consume with pending operations?
- Better idea: count up, doing one multiplication at a time
  - Start with 1 as the answer
  - Multiply by 2, store 2 as the current answer, remember we've done up to 2
  - Multiply by 3, store 6, remember we're done up to 3

An alternative

- Try computing 101!
  $101 * 100 * 99 * 98 * 97 * 96 * \ldots * 2 * 1$
- How much space do we consume with pending operations?
- Better idea: count up, doing one multiplication at a time
  - Start with 1 as the answer
  - Multiply by 2, store 2 as the current answer, remember we've done up to 2
  - Multiply by 3, store 6, remember we're done up to 3
  - Multiply by 4, store 24, remember we're done up to 4

An alternative

- Try computing 101!
  $101 * 100 * 99 * 98 * 97 * 96 * \ldots * 2 * 1$
- How much space do we consume with pending operations?
- Better idea: count up, doing one multiplication at a time
  - Start with 1 as the answer
  - Multiply by 2, store 2 as the current answer, remember we've done up to 2
  - Multiply by 3, store 6, remember we're done up to 3
  - Multiply by 4, store 24, remember we're done up to 4
  - ...

An alternative

- Try computing 101!
  $101 * 100 * 99 * 98 * 97 * 96 * \ldots * 2 * 1$
- How much space do we consume with pending operations?
- Better idea: count up, doing one multiplication at a time
  - Start with 1 as the answer
  - Multiply by 2, store 2 as the current answer, remember we've done up to 2
  - Multiply by 3, store 6, remember we're done up to 3
  - Multiply by 4, store 24, remember we're done up to 4
  - ...
  - Multiply by 101, get
    9425947759838359420851623124482936749562
    312794702543768327889353416977599316221476503087
    861591808346911623490003549599583369706302603264
    000000000000000000000000
  - Realize we're done up to the number we want, and stop

An alternative

- Try computing 101!
  $101 * 100 * 99 * 98 * 97 * 96 * \ldots * 2 * 1$
- How much space do we consume with pending operations?
- Better idea: count up, doing one multiplication at a time
  - Start with 1 as the answer
  - Multiply by 2, store 2 as the current answer, remember we've done up to 2
  - Multiply by 3, store 6, remember we're done up to 3
  - Multiply by 4, store 24, remember we're done up to 4
  - ...
  - Multiply by 101, get
    9425947759838359420851623124482936749562
    312794702543768327889353416977599316221476503087
    861591808346911623490003549599583369706302603264
    000000000000000000000000
  - Realize we're done up to the number we want, and stop
- This is an iterative algorithm – it uses constant space

Iterative algorithms as tables

product	done	max
1	1	5

- First row handles 1! cleanly

Iterative algorithms as tables

product	done	max
1	1	5
2	2	5

- First row handles 1! cleanly

Iterative algorithms as tables

product	done	max
1	1	5
2	2	5
6	3	5

- First row handles 1! cleanly
- `product` becomes `product * (done + 1)`

Iterative algorithms as tables

product	done	max
1	1	5
2	2	5
6	3	5
24	4	5

- First row handles 1! cleanly
- `product` becomes `product * (done + 1)`
- `done` becomes `done + 1`

Iterative algorithms as tables

product	done	max
1	1	5
2	2	5
6	3	5
24	4	5
120	5	5

- First row handles 1! cleanly
- `product` becomes `product * (done + 1)`
- `done` becomes `done + 1`
- The answer is `product` when `done = max`

```
(define (ifact-helper product done max)

                                  )
```

- The helper has one argument per column

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

(define (ifact-helper product done max)




                                  )
```

- The helper has one argument per column
- Which is called by `ifact`

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

(define (ifact-helper product done max)



                                  )
```

- The helper has one argument per column
- Which is called by `ifact`
- Which provides the values for the first row

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

(define (ifact-helper product done max)


        (ifact-helper (* product (+ done 1))
                      (+ done 1)
                     max) )
```

- The helper has one argument per column
- Which is called by `ifact`
- Which provides the values for the first row
- The recursive call to `ifact-helper`

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

(define (ifact-helper product done max)


        (ifact-helper (* product (+ done 1))
                      (+ done 1)
                      max) )
```

- The helper has one argument per column
- Which is called by `ifact`
- Which provides the values for the first row
- The recursive call to `ifact-helper` computes the next row

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

(define (ifact-helper product done max)
    (if (= done max)

        (ifact-helper (* product (+ done 1))
                      (+ done 1)
                      max)))
```

- The helper has one argument per column
- Which is called by `ifact`
- Which provides the values for the first row
- The recursive call to `ifact-helper` computes the next row
- And the `if` statement checks the end condition

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

(define (ifact-helper product done max)
    (if (= done max)
        product
        (ifact-helper (* product (+ done 1))
                      (+ done 1)
                      max)))
```

- The helper has one argument per column
- Which is called by `ifact`
- Which provides the values for the first row
- The recursive call to `ifact-helper` computes the next row
- And the `if` statement checks the end condition and output value

```scheme
(define (ifact-helper product done max)
    (if (= done max)
        product
        (ifact-helper (* product (+ done 1))
                      (+ done 1)
                      max)))
```

```
(define (ifact-helper product done max)
    (if (= done max)
        product
        (ifact-helper (* product (+ done 1))
                      (+ done 1)
                      max)))
(ifact 4)
```

```
(define (ifact-helper product done max)
    (if (= done max)
        product
        (ifact-helper (* product (+ done 1))
                      (+ done 1)
                      max)))
(ifact 4)
(ifact-helper 1 1 4)
```

```
(define (ifact-helper product done max)
    (if (= done max)
        product
        (ifact-helper (* product (+ done 1))
                      (+ done 1)
                      max)))
(ifact 4)
(ifact-helper 1 1 4)
(if (= 1 4) 1  (ifact-helper (* 1 (+ 1 1)) (+ 1 1) 4))
```

```
(define (ifact-helper product done max)
    (if (= done max)
        product
        (ifact-helper (* product (+ done 1))
                      (+ done 1)
                      max)))
(ifact 4)
(ifact-helper 1 1 4)
(if (= 1 4) 1  (ifact-helper (* 1 (+ 1 1)) (+ 1 1) 4))
(ifact-helper 2 2 4)
```

```
(define (ifact-helper product done max)
    (if (= done max)
        product
        (ifact-helper (* product (+ done 1))
                      (+ done 1)
                      max)))
(ifact 4)
(ifact-helper 1 1 4)
(if (= 1 4) 1  (ifact-helper (* 1 (+ 1 1)) (+ 1 1) 4))
(ifact-helper 2 2 4)
(if (= 2 4) 2  (ifact-helper (* 2 (+ 2 1)) (+ 2 1) 4))
```

```
(define (ifact-helper product done max)
    (if (= done max)
        product
        (ifact-helper (* product (+ done 1))
                      (+ done 1)
                      max)))
(ifact 4)
(ifact-helper 1 1 4)
(if (= 1 4) 1  (ifact-helper (* 1 (+ 1 1)) (+ 1 1) 4))
(ifact-helper 2 2 4)
(if (= 2 4) 2  (ifact-helper (* 2 (+ 2 1)) (+ 2 1) 4))
(ifact-helper 6 3 4)
```

```
(define (ifact-helper product done max)
    (if (= done max)
        product
        (ifact-helper (* product (+ done 1))
                      (+ done 1)
                      max)))
(ifact 4)
(ifact-helper 1 1 4)
(if (= 1 4) 1  (ifact-helper (* 1 (+ 1 1)) (+ 1 1) 4))
(ifact-helper 2 2 4)
(if (= 2 4) 2  (ifact-helper (* 2 (+ 2 1)) (+ 2 1) 4))
(ifact-helper 6 3 4)
(if (= 3 4) 6  (ifact-helper (* 6 (+ 3 1)) (+ 3 1) 4))
```

```
(define (ifact-helper product done max)
    (if (= done max)
        product
        (ifact-helper (* product (+ done 1))
                      (+ done 1)
                      max)))
(ifact 4)
(ifact-helper 1 1 4)
(if (= 1 4) 1  (ifact-helper (* 1 (+ 1 1)) (+ 1 1) 4))
(ifact-helper 2 2 4)
(if (= 2 4) 2  (ifact-helper (* 2 (+ 2 1)) (+ 2 1) 4))
(ifact-helper 6 3 4)
(if (= 3 4) 6  (ifact-helper (* 6 (+ 3 1)) (+ 3 1) 4))
(ifact-helper 24 4 4)
```

```
(define (ifact-helper product done max)
    (if (= done max)
        product
        (ifact-helper (* product (+ done 1))
                      (+ done 1)
                      max)))
```

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

```
(define (ifact-helper product done max)
    (if (= done max)
        product
        (ifact-helper (* product (+ done 1))
                      (+ done 1)
                      max)))
(ifact 4)
(ifact-helper 1 1 4)
(if (= 1 4) 1  (ifact-helper (* 1 (+ 1 1)) (+ 1 1) 4))
(ifact-helper 2 2 4)
(if (= 2 4) 2  (ifact-helper (* 2 (+ 2 1)) (+ 2 1) 4))
(ifact-helper 6 3 4)
(if (= 3 4) 6  (ifact-helper (* 6 (+ 3 1)) (+ 3 1) 4))
(ifact-helper 24 4 4)
(if (= 4 4) 24 (ifact-helper (* 24 (+ 4 1)) (+ 4 1) 4)
24
```

```
(define (ifact-helper product done max)
    (if (= done max)
        product
        (ifact-helper (* product (+ done 1))
                      (+ done 1)
                      max)))
(ifact 4)
(ifact-helper 1 1 4)
(if (= 1 4) 1  (ifact-helper (* 1 (+ 1 1)) (+ 1 1) 4))
(ifact-helper 2 2 4)
(if (= 2 4) 2  (ifact-helper (* 2 (+ 2 1)) (+ 2 1) 4))
(ifact-helper 6 3 4)
(if (= 3 4) 6  (ifact-helper (* 6 (+ 3 1)) (+ 3 1) 4))
(ifact-helper 24 4 4)
(if (= 4 4) 24 (ifact-helper (* 24 (+ 4 1)) (+ 4 1) 4)
24
```

Recursive algorithms have pending operations

- Recursive factorial:

```
(define (fact 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 expression grow continuously.

Iterative algorithms have no pending operations

- Iterative factorial:

```
(define (ifact n) (ifact-helper 1 1 n))
(define (ifact-helper product done max)
    (if (= done max)
        product
        (ifact-helper (* product (+ done 1))
                      (+ done 1)
                      max)))
```

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

- Fixed space because no pending operations

Iterative processes

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

Summary

- Lambdas allow us to create procedures which capture processes
- Procedural abstraction creates building blocks for complex processes
- Recursive algorithms capitalize on “wishful thinking” to reduce problems to smaller subproblems
- Iterative algorithms similarly reduce problems, but based on data you can express in tabular form

Recitation Time!

Reminders

- Project 0 is due Thursday
- Submit to `6.037-psets@mit.edu`
- `http://web.mit.edu/alexmv/6.037/`
- E-mail: `6.001-zombies@mit.edu`