Lecture 0

Prelude

Computational Processes

- Abstract beings that inhabit computers
- Manipulate data
- Directed by a program
- Written in a programming language

The Tool of this Course: Scheme

- Dialect of Lisp (1958)
- Proposed in 1975
- Extremely powerful and elegant
- Standardised into R^{n}Rs

R5Rs

- Many implementations available

I use DrRacket

actual goal of this course

- Enables you to “go meta”

Scheme???

```javascript
let balance = 100;

function withdraw(amount) {
   if (balance >= amount) {
      balance = balance - amount;
      return balance;
   } else {
      return "Insufficient funds";
   }
}

withdraw(25); // output: 75
```

```scheme
(define balance 100)

(define (withdraw amount)
  (if (>= balance amount)
      (begin (set! balance (- balance amount))
             balance)
      "Insufficient Funds"))

(withdraw 25) ; output: 75
```

Structure and Interpretation of Computer Programs -- JavaScript Adaptation
`https://sicp.comp.nus.edu.sg/index.html`

Racket

- `https://racket-lang.org/`
- language and IDE
- Racket language very similar *but not identical* to Scheme
- behave like Scheme: `#lang r5rs`

Multiple Variants!

- **Check that you are enrolled in correct variant!** (5 credits vs. 6 credits)

- Multiple variants, but same course

- One course page on Canvas

Lecture 1: Fundamentals of Higher Order Programming

1. Scheme S-expressions, Function definitions
2. Lexical scoping vs. dynamic scoping
3. Iteration as optimised tail recursion
4. Higher-order procedures and anonymous lambdas.

Lecture 1: Fundamentals of Higher Order Programming

$$\int_a^b f = \left[f\left(a + \frac{dx}{2}\right) + f\left(a + dx + \frac{dx}{2}\right) + f\left(a + 2dx + \frac{dx}{2}\right) + \dots \right] dx$$

```
(define (integral f a b dx)
  (define (add-dx x) (+ x dx))
  (* (sum f (+ a (/ dx 2.0)) add-dx b)
     dx))
```

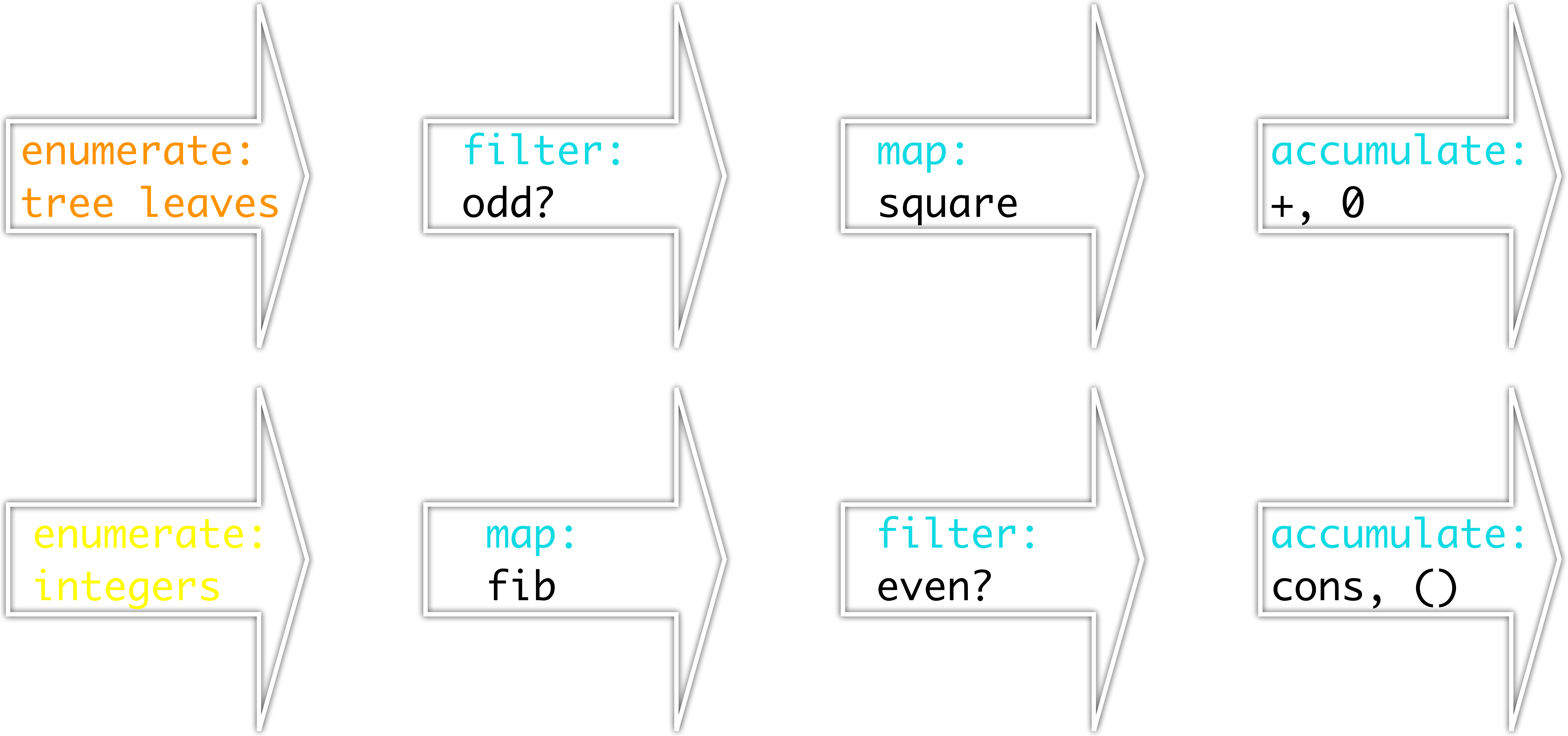
```
> (integral cube 0 1 0.01)
0.24998750000000042
```

Higher-Order Functions and Reuse

Lecture 2: Advanced Higher Order Programming

1. Cons-cells, lists and nested lists.
2. List processing and Higher Order List Procedures
3. Symbols and Homoiconicity: Quoting Lists
4. Homoiconicity for Meta-programming
5. Case Study: Symbolic derivation

Lecture 2: Advanced Higher Order Programming

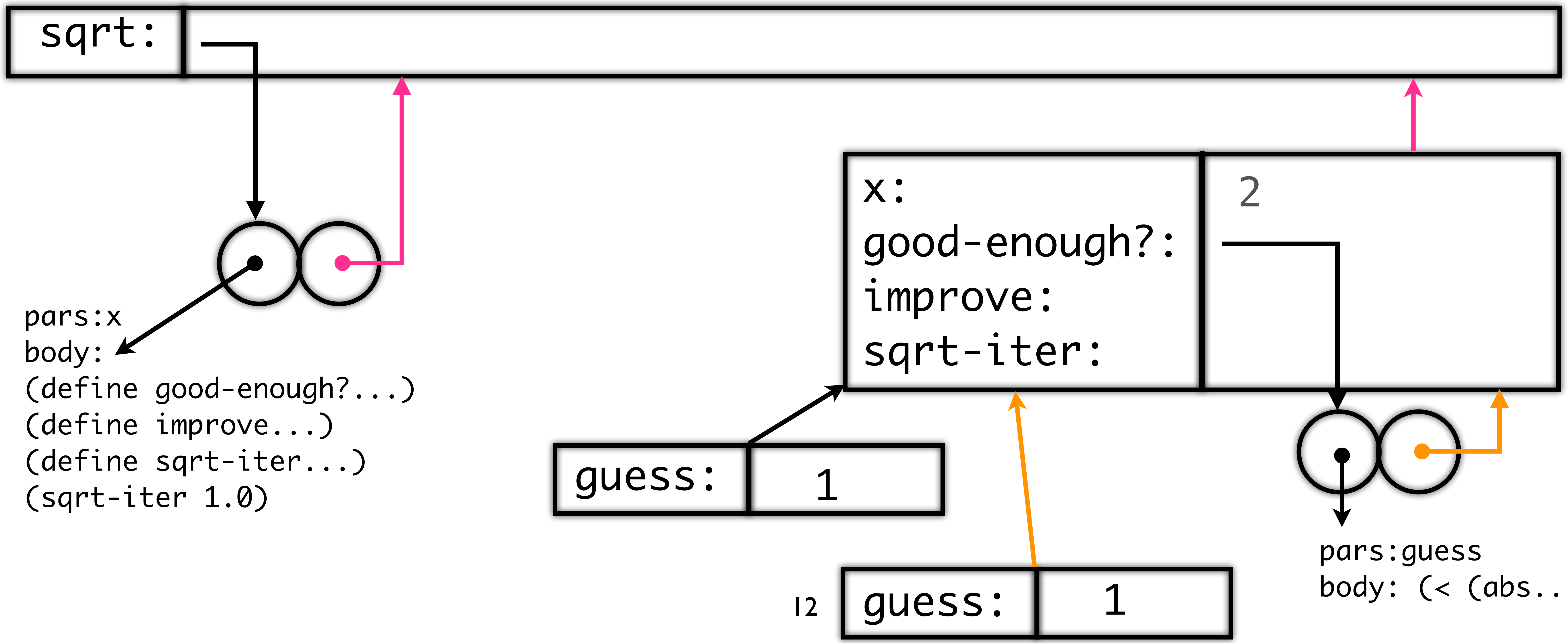


A Signal-Processing Engineer’s View

Lecture 3: Fundamental Concepts of State, Scoping and Evaluation Order

1. begin, set! and mutable state
2. Objects as closures
3. Environment diagrams, box-and-pointer diagrams
4. (Infinite) streams and lazy evaluation
5. `delay` and `force`

```
(define (sqrt x)
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess)
    (average guess (/ x guess)))
  (define (sqrt-iter guess)
    (if (good-enough? guess)
        guess
        (sqrt-iter (improve guess))))
  (sqrt-iter 1.0))
```



Lecture 4: Continuations and current-continuations

1. Continuations
2. call-with-current-continuation
3. An implementation of
   - goto,
   - yield,
   - coroutines
   - exception handling

Lecture 4: Continuations and current-continuations

```scheme
(define p1
  (new-process
   (lambda (input)
     (define (loop)
       (display "Tick 1 ")
       (display input)
       (newline)
       (set! input (transfer p2 (+ input 1)))
       (loop))
     (loop))))
```

```scheme
(define p2
  (new-process
   (lambda (input)
     (define (loop)
       (display "Tick 2 ")
       (display input)
       (newline)
       (set! input (transfer p3 (+ input 1)))
       (loop))
     (loop))))
```

```scheme
(define p3
  (new-process
   (lambda (input)
     (define (loop)
       (display "Tick 3 ")
       (display input)
       (newline)
       (set! input (transfer p1 (+ input 1)))
       (loop))
     (loop))))
```

```scheme
(transfer p1 0)
```

Initiate the computation in the REPL

Coroutines

Lecture 5: Semantics of Higher-Order Languages

1. Concrete vs. Abstract Syntax
2. Meta circular interpretation
3. The analyzing interpreter (i.e. compiler)
4. CPS interpretation and semantics of `call-with-current-continuation`

Lecture 5: Semantics of Higher-Order Languages

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Lecture 6: Variations on the Semantics

1. A lazy evaluation version of Scheme + thunkified interpreter
2. A nondeterministic version of Scheme + continuation-based interpreter

Lecture 6: Variations on the Semantics

```
(define (list-ref items n)
  ...)

(define (map proc items)
  ...)

(define (add-lists list1 list2)
  (cond ((null? list1) list2)
        ((null? list2) list1)
        (else (cons (+ (car list1) (car list2))
                    (add-lists (cdr list1) (cdr list2))))))


(define ones (cons 1 ones))
(define integers (cons 1 (add-lists ones integers)))
```

Chapter 2

No special stream procedures needed

Haskellish semantics

Lecture 6: Variations on the Semantics

"Difficult" People

Baker, Cooper, Fletcher, Miller, and Smith live on different floors of an apartment house that contains only five floors. Baker does not live on the top floor. Cooper does not live on the bottom floor. Fletcher does not live on either the top or the bottom floor. Miller lives on a higher floor than does Cooper. Smith does not live on a floor adjacent to Fletcher's. Fletcher does not live on a floor adjacent to Cooper's. Where does everyone live?

Lecture 6: Variations on the Semantics

```
(define (multiple-dwelling)
  (let ((baker (amb 1 2 3 4 5))
        (cooper (amb 1 2 3 4 5))
        (fletcher (amb 1 2 3 4 5))
        (miller (amb 1 2 3 4 5))
        (smith (amb 1 2 3 4 5)))
    (require
     (distinct? (list baker cooper fletcher miller smith)))
    (require (not (= baker 5)))
    (require (not (= cooper 1)))
    (require (not (= fletcher 5)))
    (require (not (= fletcher 1)))
    (require (> miller cooper))
    (require (not (= (abs (- smith fletcher)) 1)))
    (require (not (= (abs (- fletcher cooper)) 1)))
    (list (list 'baker baker)
          (list 'cooper cooper)
          (list 'fletcher fletcher)
          (list 'miller miller)
          (list 'smith smith))))
```

Lecture 7: Introduction to the λ-calculus

1. λ-expressions and β-reduction
2. Computability in λ-calculus:
   a construction of functional programming languages
3. Recursion and the Fixed-point Theorem.

Y

= λF. (λx. (F (x x)) λx. (F (x x)))

$=_\beta$ λF. (F (λx. (F (x x)) λx. (F (x x))))

$=_\beta$ λF. (F (F (λx. (F (x x)))) λx. (F (x x)))))

$=_\beta$ λF. (F (F (F (λx. (F (x x))))) λx. (F (x x))))))

$=_\beta$ λF. (F (F (F (F (λx. (F (x x)) λx. (F (x x)))))))

(fac c_3) = ((Y F) c_3)

= (λn. (if … ((F (λx. (F (x x)) λx. (F (x x)))) (dec n)) …) c_3)

$=_\beta$ (if … ((F (λx. (F (x x)) λx. (F (x x)))) c_2) …)

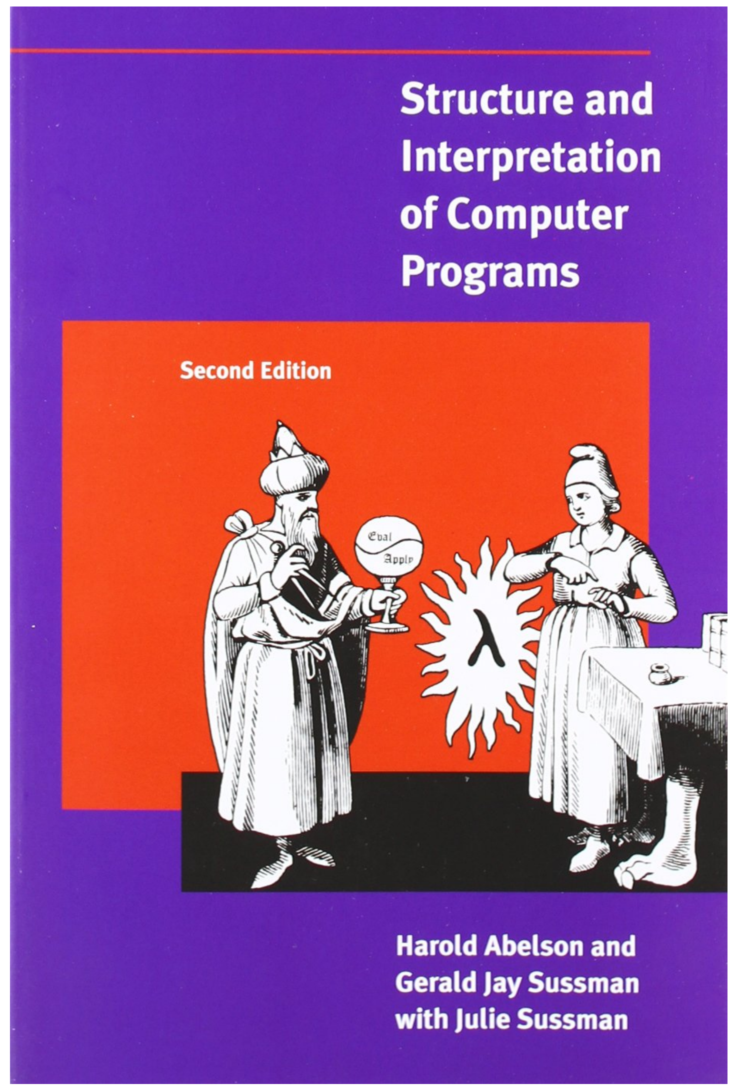
$=_\beta$ (if … (λn. (if … ((F (λx. (F (x x)) λx. (F (x x)))) (dec n)) …) c_2) …)

$=_\beta$ (if … (if … ((F (λx. (F (x x)) λx. (F (x x)))) c_1) …) c_2) …)

$=_\beta$ …

Operational Interpretation of Y

Study Material


Structure and Interpretation of Computer Programs
Second Edition
Harold Abelson and Gerald Jay Sussman with Julie Sussman

- Lecture 1, 2, 3, 5, 6:
  Ch. 1—4 of Structure and Interpretation of Computer Programs
  Gerald Jay Sussman and Hal Abelson
  Available online: `https://mitpress.mit.edu/sites/default/files/sicp/full-text/book/book.html`

- Lecture 4, 7:
  Slides + notes in classroom