Functional Programming in Education

George Wilson

Data61/CSIRO

george.wilson@data61.csiro.au

22nd May 2018

University

First year, first semester

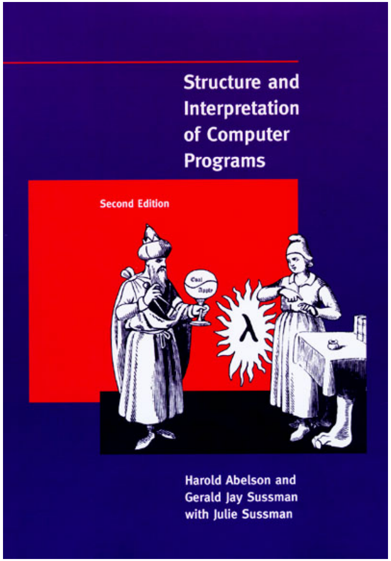
Which language?

```java
class Hello {

  public static void main(String[] args) {
    System.out.println("Hello, world!");
  }

}
```

	Content
Week 1	Basic expressions
Week 2	procedure declarations
Week 3	if-statement
Week 4	while-statement
Week 5	for-statement
...	



Structure and Interpretation of Computer Programs

Second Edition

Harold Abelson and
Gerald Jay Sussman
with Julie Sussman

```scheme
(define (map proc items)
  (if (null? items)
      nil
      (cons (proc (car items))
            (map proc (cdr items)))))
```

```scheme
(define (abs x)
  (cond ((> x 0) x)
        ((= x 0) x)
        ((< x 0) (- x))))
```

Evaluation by substitution

```
(define (sum-of-squares x y)
  (+ (sqr x) (sqr y)))
```

Evaluation by substitution

```
(define (sum-of-squares x y)
  (+ (sqr x) (sqr y)))
```

```
     (sum-of-squares 3 4)

=>   (+ (sqr 3) (sqr 4))
=>   (+ (* 3 3) (sqr 4))
=>   (+ 9 (sqr 4))
=>   (+ 9 (* 4 4))
=>   (+ 9 16)
=>   25
```

```scheme
(define (factorial n)
  (if (<= n 1)
      1
      (* n (factorial (- n 1)))))
```

```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

Incredible breadth of content

complexity analysis

symbolic computation with quotation

interpreters

object-oriented programming

logic programming

many other concepts

Criticisms

Examples are drawn from overly-technical domains

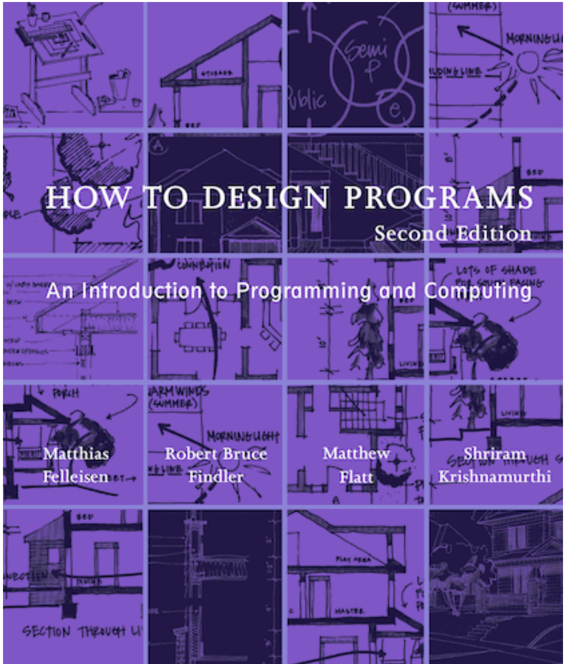
```
(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp)
         (if (same-variable? exp var) 1 0))
        ((sum? exp)
         (make-sum (deriv (addend exp) var)
                   (deriv (augend exp) var)))
        ((product? exp)
         (make-sum
           (make-product (multiplier exp)
                         (deriv (multiplicand exp) var))
           (make-product (deriv (multiplier exp) var)
                         (multiplicand exp))))
        (else
         (error "unknown expression type -- DERIV" exp))))
```

Criticisms

Lacks treatment of foundational problem-solving techniques

From an educational point of view, our experience suggests that undergraduate computer science courses should emphasize basic notions of modularity, specification, and data abstraction, and should not let these be displaced by more advanced topics, such as design patterns, object-oriented methods, concurrency, functional languages, and so on.

— Jackson and Chapin, 2000



HOW TO DESIGN PROGRAMS

Second Edition

An Introduction to Programming and Computing

Matthias Felleisen
Robert Bruce Findler
Matthew Flatt
Shriram Krishnamurthi

HR

received April 198

A critique of Abelson and Sussman
- or -
Why calculating is better than scheming

Philip Wadler
Programming Research Group
11 Keble Road
Oxford, OX1 3QD

Abelson and Sussman have written an excellent textbook which may start a revolution in the way programming is taught [Abelson and Sussman 1985a, b]. Instead of emphasizing a particular programming language, they emphasize standard engineering techniques as they apply to programming. Still, their textbook is intimately tied to the Scheme dialect of Lisp. I believe that the same approach used in their text, if applied to a language such as KRC or Miranda, would result in an even better introduction to programming as an engineering discipline. My belief has strengthened as my experience in teaching with Scheme and with KRC has increased.

—

```
perimeter shape = case shape of
  Square s        -> s * 4
  Rectangle w h -> 2*w + 2*h
  Circle r        -> 2 * pi * r
```

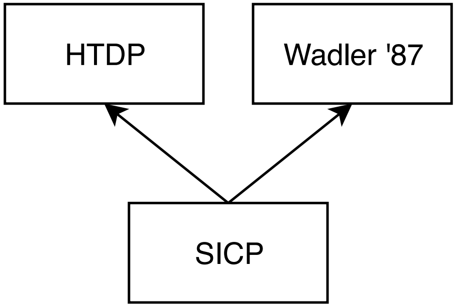
```scheme
(define (sum items)
  (cond ((null? items) 0)
        (else (+ (car items) (sum (cdr items))))))
```

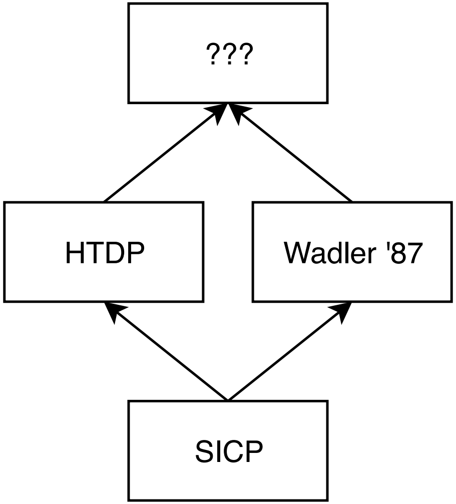
```scheme
(define (sum items)
  (cond ((null? items) 0)
        (else (+ (car items) (sum (cdr items))))))
```

```haskell
sum items = case items of
  []   -> 0
  x:xs -> x + sum xs
```

```
[] ++ ys       =  ys
(x:xs) ++ ys   =  x:(xs ++ ys)

(xs ++ ys) ++ zs = xs ++ (ys ++ zs)
```


HTDP
Wadler '87
SICP


???
HTDP
Wadler '87
SICP

3 + False

```
<interactive>:1:1: error:
    • No instance for (Num Bool) arising from a use of ‘+’
    • In the expression: 3 + False
      In an equation for ‘it’: it = 3 + False
```

Prelude.hs

```haskell
module Prelude
   ( N.Integer, (+)
   , (==), Bool (..)
   )
where

import Data.Bool (Bool (..))
import qualified Data.Eq as E
import GHC.Num (Integer)
import qualified GHC.Num as N

(+) :: Integer -> Integer -> Integer
(+) = (N.+)

(==) :: Integer -> Integer -> Bool
(==) = (E.==)
```

Thanks for listening!