

# Functional Programming in Education

George Wilson

Data61/CSIRO

`george.wilson@data61.csiro.au`

22nd May 2018



University

First year, first semester

Which language?

```
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

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

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)
```

Evaluation by substitution

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

---

```
      (sum-of-squares 3 4)  
=>    (+ (sqr 3) (sqr 4))
```

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))
```

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))
```

## 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))
```

## 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)
```

## 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))
```

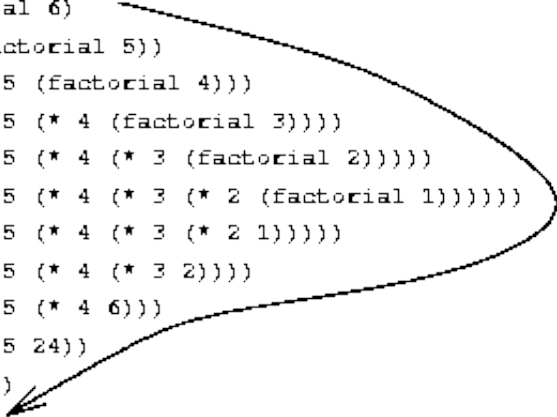
```
=> (+ 9 (sqr 4))
```

```
=> (+ 9 (* 4 4))
```

```
=> (+ 9 16)
```

```
=> 25
```

```
{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

Lacking coverage 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 (emphasis mine)



# HOW TO DESIGN PROGRAMS

Second Edition

An Introduction to Programming and Computing


Matthias  
Felleisen

Robert Bruce  
Findler

Matthew  
Flatt

Shriram  
Krishnamurthi

```
(require 2htdp/image)
(require 2htdp/universe)

(define rocket )

(define (picture-of-rocket height)
  (place-image rocket 50 height (empty-scene 100 60)))

(define (sign x)
  (cond ((> x 0) 1)
        ((= x 0) 0)
        ((< x 0) -1)))

(define (picture-of-rocket.v2 height)
  (cond
    [(<= height 60)
     (place-image 50 height
                  (empty-scene 100 60))]
    [(> height 60)
     (place-image 50 60
                  (empty-scene 100 60))]))
```

Welcome to [DrRacket](#), version 7.2 [3m].

Language: **Beginning Student**; memory limit: 128 MB.

> (picture-of-rocket.v2 5555)

*place-image: expects 4 arguments, but found only 3*

> |

Choose Language

☐ The Racket Language (ctl-R)

Start your program with #lang to specify the desired dialect. For example:

```
#lang racket           [docs]
#lang racket/base      [docs]
#lang typed/racket     [docs]
#lang scribble/base    [docs]
```

... and many more

☒ Teaching Languages (ctl-T)

**How to Design Programs**  
Beginning Student  
**Beginning Student with List Abbreviations**  
Intermediate Student  
Intermediate Student with lambda  
Advanced Student

**DeinProgramm**  
Die Macht der Abstraktion - Anfänger  
Die Macht der Abstraktion  
Die Macht der Abstraktion mit Zuweisungen  
Die Macht der Abstraktion - fortgeschritten

☐ Other Languages (ctl-O)

...

Show Details

OK

Cancel

### 1. From Problem Analysis to Data Definitions

Identify the information that must be represented and how it is represented in the chosen programming language. Formulate data definitions and illustrate them with examples.

### 2. Signature, Purpose Statement, Header

State what kind of data the desired function consumes and produces. Formulate a concise answer to the question *what* the function computes. Define a stub that lives up to the signature.

### 3. Functional Examples

Work through examples that illustrate the function's purpose.

### 4. Function Template

Translate the data definitions into an outline of the function.

### 5. Function Definition

Fill in the gaps in the function template. Exploit the purpose statement and the examples.


### 6. Testing

Articulate the examples as tests and ensure that the function passes all. Doing so discovers mistakes. Tests also supplement examples in that they help others read and understand the definition when the need arises—and it will arise for any serious program.

Figure 1: The basic steps of a function design recipe

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

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

  
received April 1980



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.

---

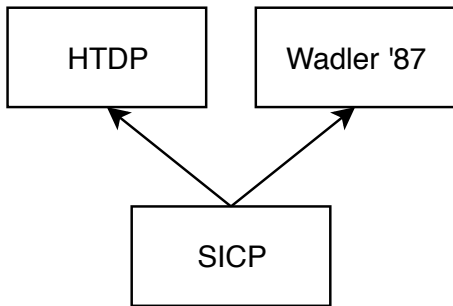
```
perimeter shape =  
  case shape of  
    Square s      -> s * 4  
    Rectangle w h -> 2*w + 2*h  
    Circle r      -> 2 * pi * r
```

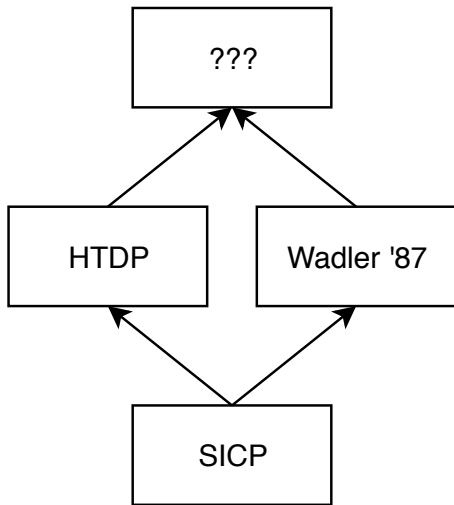
```
(define (sum items)
  (cond ((null? items) 0)
        (else (+ (car items) (sum (cdr items))))))
```

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

```
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)
```





haskell for mac screenshot



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

```
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.==)
```

custom type error machinery in GHC  
Helium?

Thanks for listening!