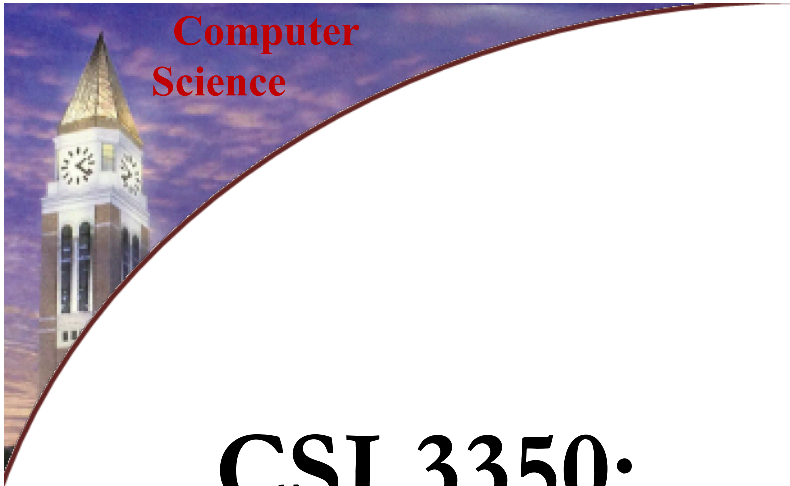
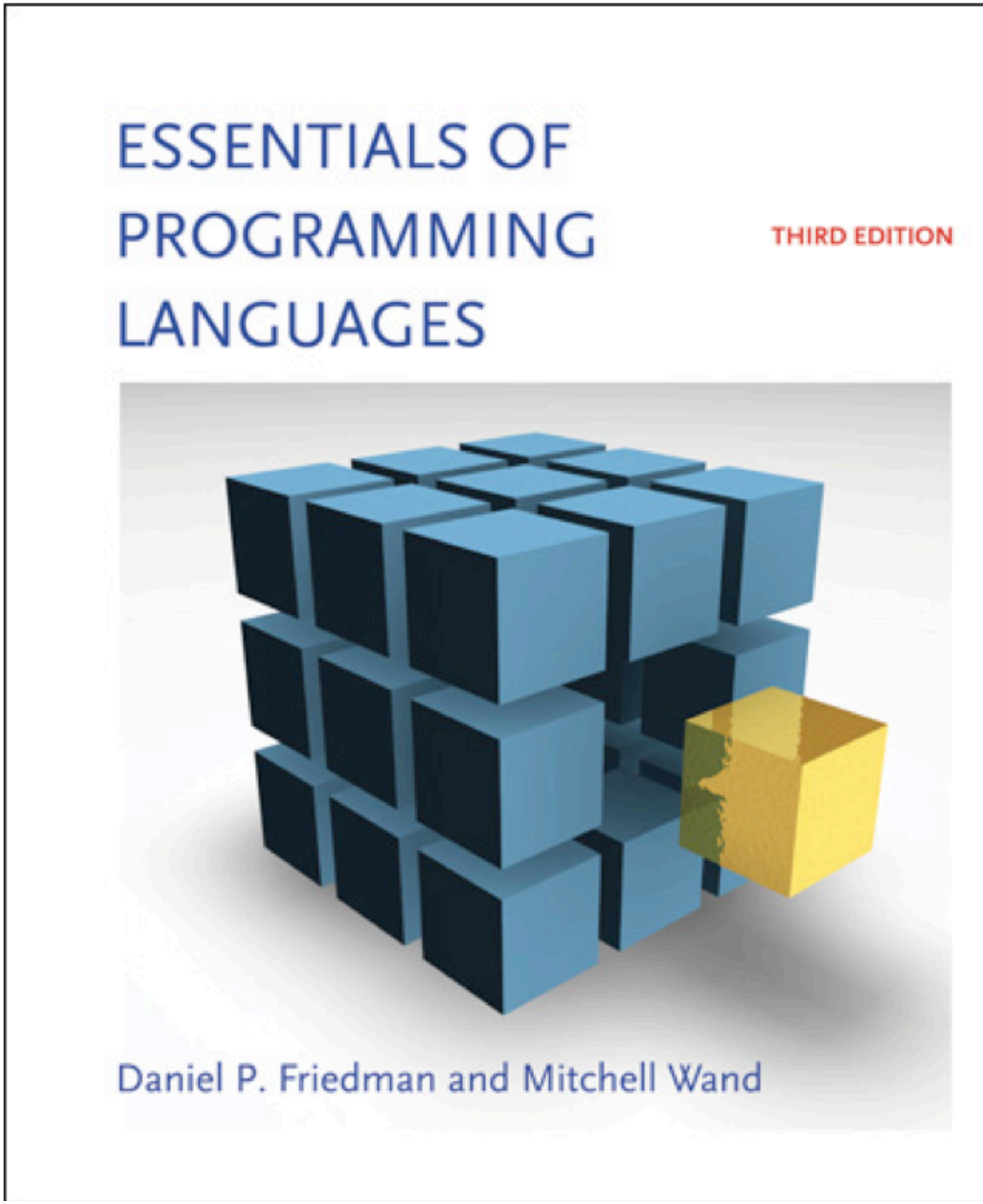
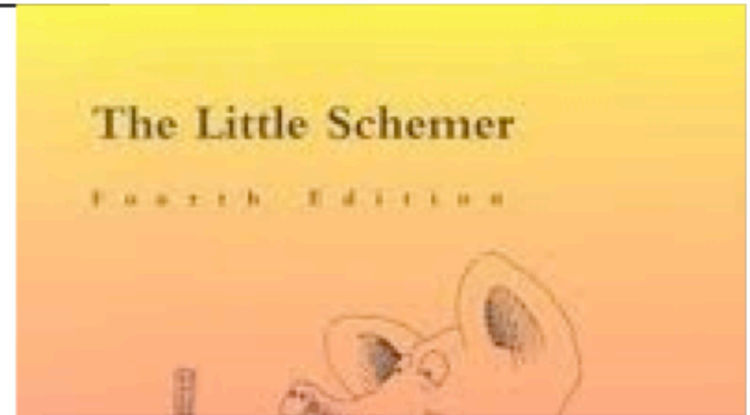


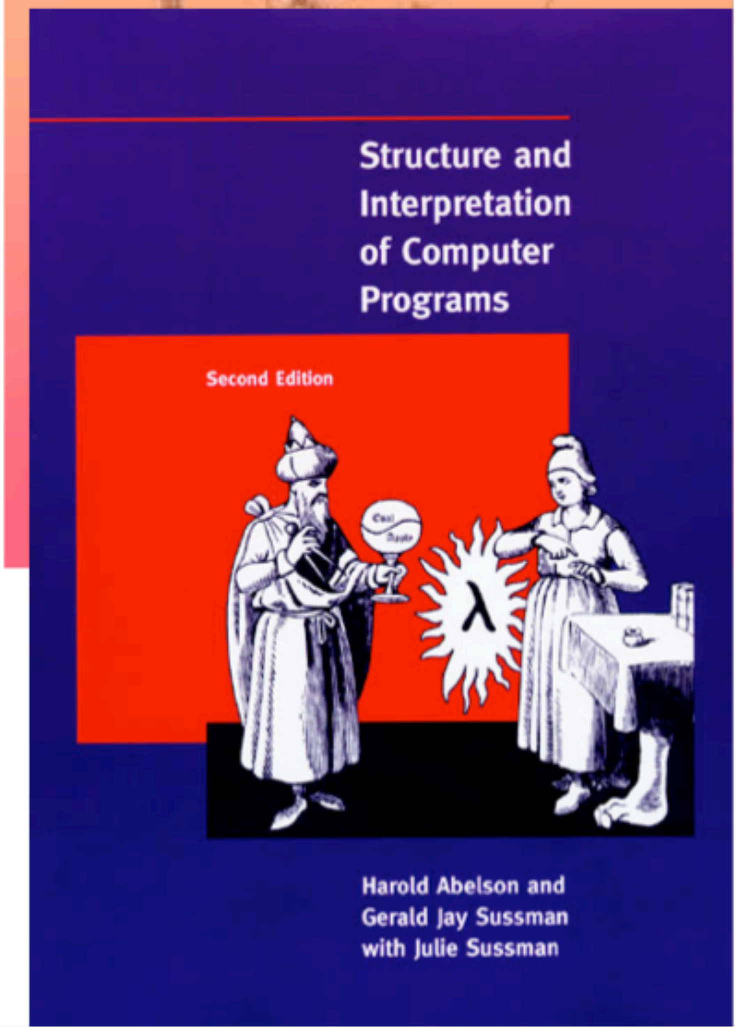
CSI 3350: PROGRAMMING LANGUAGES

Department of Computer Science & Engineering

Oakland University


ESSENTIALS OF
PROGRAMMING
LANGUAGES
THIRD EDITION
Daniel P. Friedman and Mitchell Wand


The Little Schemer
Fourth Edition


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

Required Books

1) ESSENTIALS OF PROGRAMMING LANGUAGES – 3rd Edition
Publisher: The MIT Press; (April 18, 2008)
ISBN: 978-0262062794
(URL: https://karczmarczuk.users.greyc.fr/TEACH/Doc/EssProgLan.pdf)

2) THE LITTLE SCHEMER – 4th Edition
Publisher: The MIT Press; (December 21, 1995)
ISBN: 978-0262560993
(URL: https://7chan.org/pr/src/The_Little_Schemer_4th_2.pdf)

3) STRUCTURE AND INTERPRETATION OF COMPUTER PROGRAMS – 2nd Edition
(URL: https://web.mit.edu/alexmv/6.037/sicp.pdf)

An update !

CSI3350 Course Objectives Fall 2019

- Be able to describe main quality criteria for the design of high level programming languages such as readability, writability etc.
- Be able to describe syntax of fundamental program components
- Be able to discuss fundamental concepts of semantics
- Be able to describe parameter passing and access to non-locals
- Be able to describe data types and type system
- Be able to apply major features of functional programming languages

Reading List

- SICP
  - Sections 1.1.1 ~ 1.1.6
  - Sections 2.2.1, 2.2.2 & 2.2.3
- The little Schemer
  - Preface p.xiii
  - Chap 1 ~ 3
- Revised Report on the Algorithmic Language Scheme
  - Section 1 [overview]
  - Section 6.1 – 6.3 [Standard Procedures]

Elements of Programming

- primitive *expressions*

Elements of Programming

- primitive *expressions*
- means of *combination*

Elements of Programming

- primitive *expressions*
- means of *combination*
- means of *abstraction*

Computer Science

Primitive Expressions

- Defined by basic data types

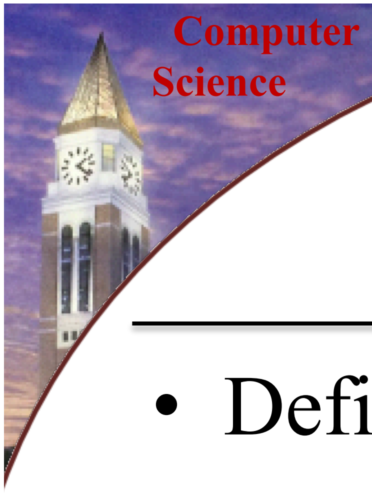
Java

- int, double, float
- 3,5, ...
- 3 + 5
- 3 + 5 + 100

Scheme / Racket

essential!

- number
- 3,5, ...
- (+ 3 5)
- (+ 3 5 100)



Primitive Expressions

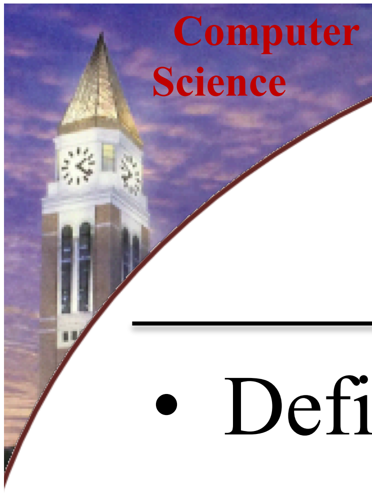
- Defined by basic data types

Java

- boolean
- true, false
- !false
- true && true && true

Scheme / Racket

- boolean
- #t, #f
- (not #f)
- (and #t #t #t)



Primitive Expressions

- Defined by basic data types

Java

- String
- “hello”
- “hello”.substring(2)
- “hello”.length()
- “hello” + “world”

Scheme / Racket

- String
- “hello”
- (substring “hello” 2)
- (string-length “hello”)
- (string-append “hello” “world”)

Means of Combination

- Compound elements are built from simpler ones

Means of Combination

- Compound elements are built from simpler ones

```
(* 2 4)
```

Means of Combination

- Compound elements are built from simpler ones

```
(* 2 4) (+ 3 5)
```

Means of Combination

- Compound elements are built from simpler ones

```
(+ (* 2 4) (+ 3 5))
```

Means of Combination

- Compound elements are built from simpler ones

```
(* 3 (+ (* 2 4) (+ 3 5)))
```

Means of Abstraction

- By which compound elements can be named and reused as units

Means of Abstraction

- By which compound elements can be named and reused as units

Java

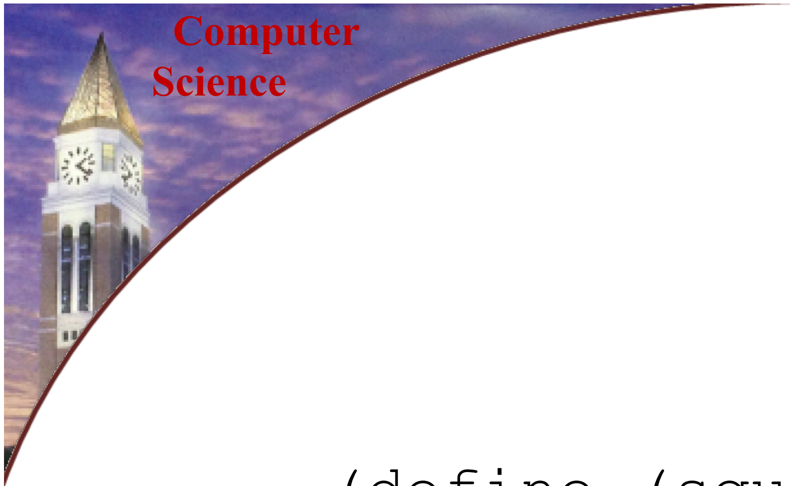
- method

```
class A {
 int square (int i) {
    return i * i;
 }
}
```

Scheme / Racket

- Function/Procedure

```
(define
    (square i )
      (* i  i )
)
```




```
(define (square i)    (*    i    i) )
```

Computer Science

```
(define (square i)    (*    i    i) )
```

To

```
(define (square i)    (*    i    i) )
```

To square

```
(define (square i)    (*    i    i) )
```

To square something

Computer Science

```
(define (square i)    (*    i    i) )
```

To square something ,

multiply

```
(define (square i)    (*     i    i) )
```

To square something ,

multiply it

```
(define (square i)   (*    i    i) )
```

To square something ,

multiply it by itself

```
(define (square i)   (*   i   i) )
```

To square something ,

```
(define (square i)    (*    i    i) )
```

```
(define (square i)   (*   i   i) )
```

Computer Science

function signature

```
(define (square i)   (*    i    i) )
```

function signature

```
(define (square i)   (*     i     i)  )
```

Computer Science

function signature

function body

```
(define (square i) (*    i    i) )
```

Means of Abstraction

- By which compound elements can be named and reused as units

Java

- method

```
class A {
 int square (int i) {
    return i * i;
 }
}
```

Scheme / Racket

- Function/Procedure

```
(define
    (square i )
     (* i  i )
)
```

Means of Abstraction

- By which compound elements can be named and reused as units

Java

- method

```
class A {
 int add2 (int i j) {
    return i + j;
 }
}
```

Scheme / Racket

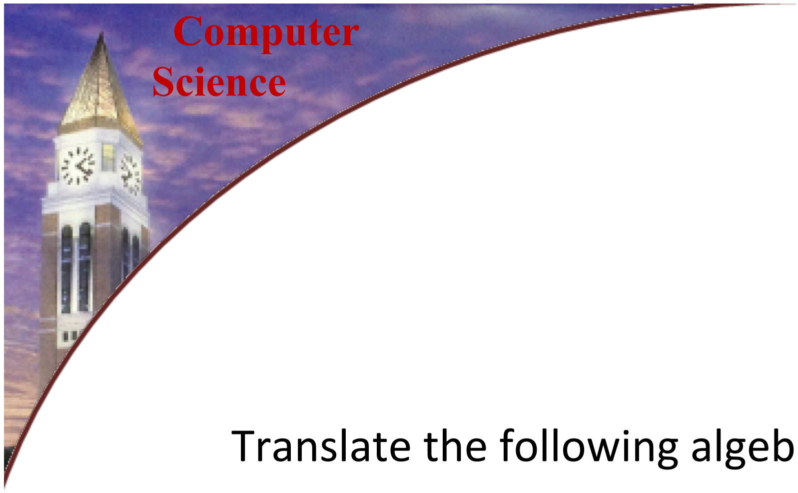
- Function/Procedure

```
(define
    (add2 i j )
     (+ i j )
)
```

Means of Abstraction

Scheme / Racket

```
(define
     (doit arg)
         (let   (  (v 300)
                      (t 42    )
                   )
          (add2 v t)
         )
)
```

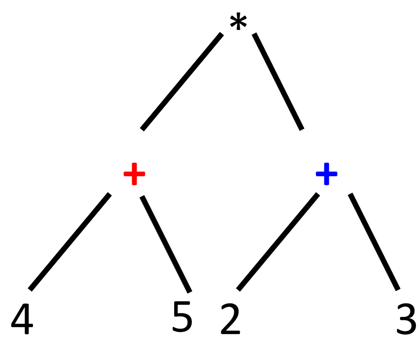


Translate the following algebraic formulas into **Scheme**'s notation:

```
( ( 3 + 3 ) * 9 )
( ( 6 * 9 ) / (( 4 + 2 ) + ( 4 * 3 )) )
( ( 6 * 9 ) / ( 4 + 2 ) + ( 4 * 3 ) )
(2 * ((20 - (91 / 7)) * (45 - 42)) )
```

Standard Scheme Expressions

- Prefix tree ((4 + 5) * (2 + 3))



(left-child-root left-left-child left-right-child)

```
( root  left-child  right-child  )
```

```
( * ( + 4 5 ) ( + 2 3) )
```

Making Use of Number Types

Factorial

```
(define
     (fact n )
      . . .
)
```

Making Use of Number Types

Factorial

```
(define
     (fact n )
       (if
          (= n 0)
          1
          (* n (fact (- n 1)))
       )
)
```