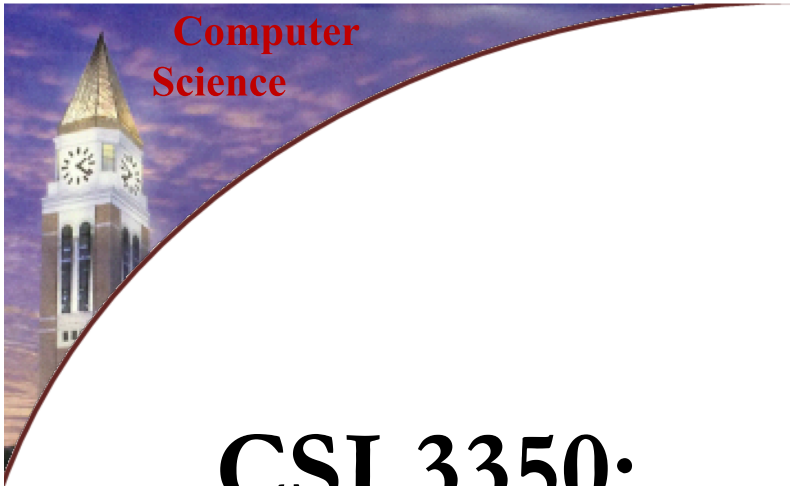
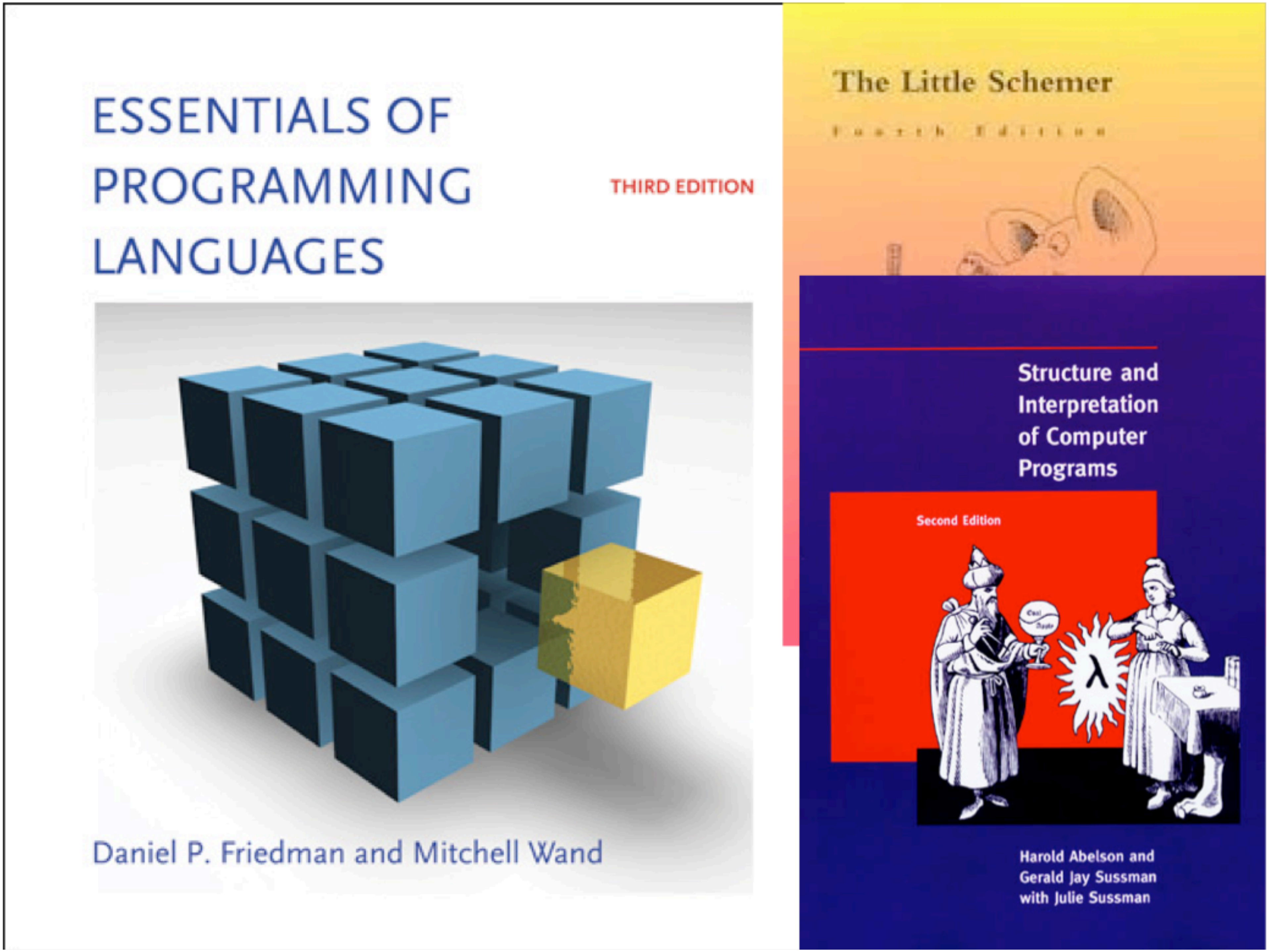


CSI 3350: PROGRAMMING LANGUAGES

Department of Computer Science & Engineering

Oakland University


ESSENTIALS OF PROGRAMMING LANGUAGES
THIRD EDITION
Daniel P. Friedman and Mitchell Wand
The Little Schemer
Structure and Interpretation of Computer Programs
Second Edition
λ
Harold Abelson and
Gerald Jay Sussman
with Julie Sussman

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

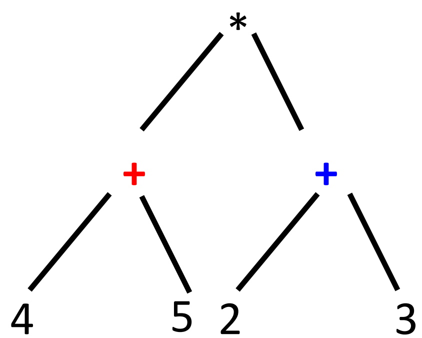
- SICP
  - Sections 1.1.1 ~ 1.1.6
  - Sections 2.2.1, 2.2.2 & 2.2.3
- The little Schemer
  - Preface p.xiii
  - Chap 1 ~ 3
- Revised Report on the Algorithmic Language Scheme
  - Section 1 [overview]
  - Section 6.1 – 6.3 [Standard Procedures]

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)))
       )
)
```

Computer Science

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

multiply it by itself

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

To square something ,

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

Computer Science

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

function signature

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

function signature

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

function signature

function body

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

λ : the Ultimate Abstraction in ALL Programming Languages

- Syntax of λ:

```
t ::= x          (variable)
    | λx.t       (abstraction)
    | t t        (application)
```

Different ways (paradigms) of programming

The following two programs of different paradigms are essentially doing the **SAME THING** !

```
(sumSq n) = sum ( map square [ 0 .. n ] )
```

Functional programming Paradigm (more this semester)

```
public static long sumSq (int n ){
        long sum = 0;
        int i = 0;
        while ( i <= n ){
                sum = sum + i * i;
                i += 1;
        }
        return sum;
}
```

Imperative programming Paradigm (like Java)

`if` and `cond`

(**if** condition
  consequent$_1$
  alternative
)

(**cond**
  (condition$_1$ consequent$_1$)
  (condition$_2$ consequent$_2$)
  . . .
  (condition$_n$ consequent$_n$)
  (**else** alternative)
)

`if` and **`cond`** are computationally equivalent expressions (functions in our functional language Scheme), your call to decide which to use. See the examples on the next slide.

write a function that takes one integer input `n`, and outputs "negative" is `n` is less than 0, "zero" if `n` is equal to 0, "one" if `n` is equal to 1, "two" if `n` is equal to 2, for all other cases simply output "etc.,"

```
(define (p n)
  (cond
    ( (< n 0) "negatie")
    ( (= n 0) "zero" )
    ( (= n 1) "one" )
    ( (= n 2) "two" )
    ( else "etc.," )

    )
  )
```

```
(define (pIf n)
  (if (< n 0)
      "negative"
      (if (= n 0)
          "zero"
          (if (= n 1)
              "one"
              (if (= n 2)
                  "two"
                  "etc.,") )
              )
          )
  )
```

`p` and **`pIf`** are doing the **same thing**, but –
which one is easier to you ?

Computer Science

Making Use of Number Types

Factorial

function signature

function body

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

Assume: `a` is not greater than `b`

```
(define (sum-integers-between a b) ...)

> (sum-integers-between 2 5)
14
```

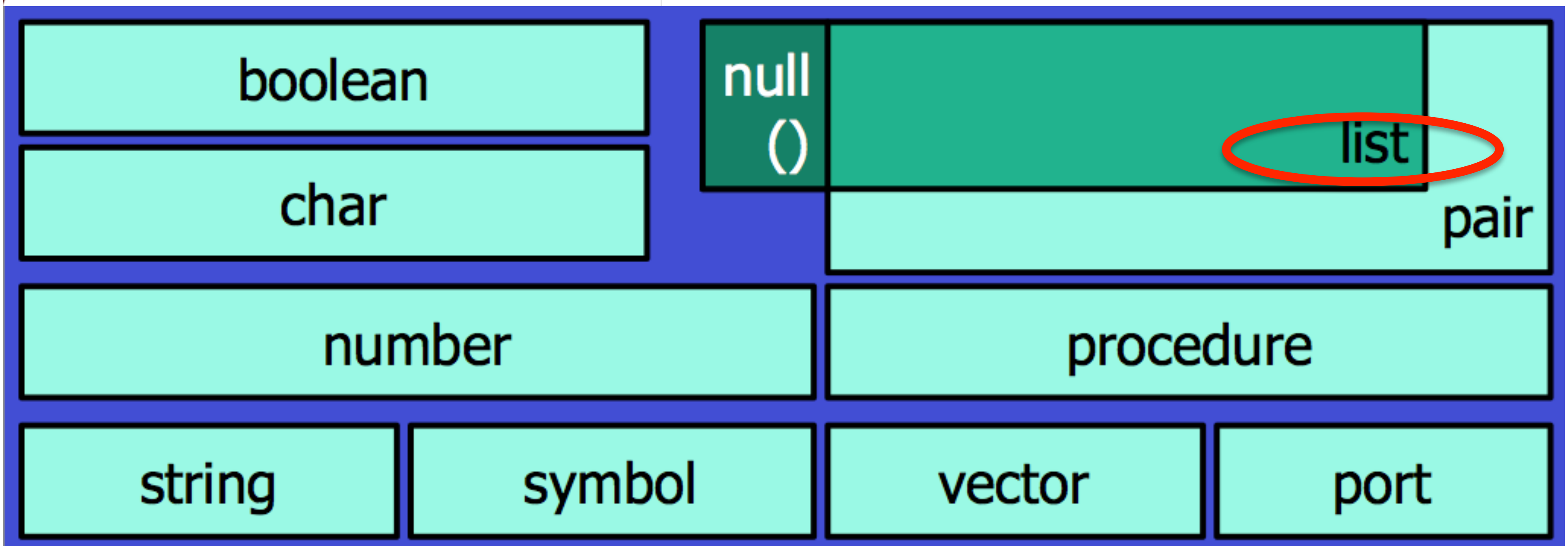
```scheme
(define (sum-integers-between a b)
  (if (= b a)
      a
      (+ b (sum-integers-between a (- b 1)))))
```

```
(define (sum-integers-between a b)
  (if (= b a)
      a
      (+ b (sum-integers-between a (- b 1)))))
```

Base case

Recursive case

Data Types in Scheme



List Manipulation

- list
- car, cdr, cddr, cadr etc
- first, second . . .
- length
- reverse
- append
- cons

List Manipulation

```
'( 1 2 3)

(car '( 1 2 3))  ➡  1
(cdr '( 1 2 3))  ➡  '(2 3)
```

List Manipulation

```
'( 1 2 3)

(car '( 1 2 3))  → 1
(cdr '( 1 2 3))  → '(2 3)
(cadr '( 1 2 3)) → 2
```

List Manipulation

```
'( 1 2 3)

(car '( 1 2 3))  →  1
(cdr '( 1 2 3))  →  '(2 3)
(cadr '( 1 2 3)) →  2
(cddr '( 1 2 3)) →  '(3)
```

List Manipulation

```
'( 1 2 3)

(car '( 1 2 3))  ➡ 1
(cdr '( 1 2 3))  ➡ '(2 3)
(cadr '( 1 2 3)) ➡ 2
(cddr '( 1 2 3)) ➡ '(3)


(cadr '( 1 (2 3)) ) ➡    ?
```

Computer Science

List Manipulation

```
(cadr `(1 (2 3)) )
```

List Manipulation

```
(cadr `(1 (2 3)) )
```

a compound function!

List Manipulation

```
(cadr `(1 (2 3)) )
```

Computer Science

List Manipulation

```
(cadr `(1 (2 3)) )
```

order of execution

```
(cadr `(1 (2 3))) ➡ (car(cdr `(1 (2 3))))
```

`(cadr lst)` → `(car(cdr)lst)`

`lst` above can refer to any list, like `(1 (2 3))

Computer Science

```
(cadr `(1 (2 3)))  ➡  (car(cdr `(1 (2 3))))
```

?

```
(cadr `(1 (2 3))) → (car(cdr `(1 (2 3))))
```

`(2 3)