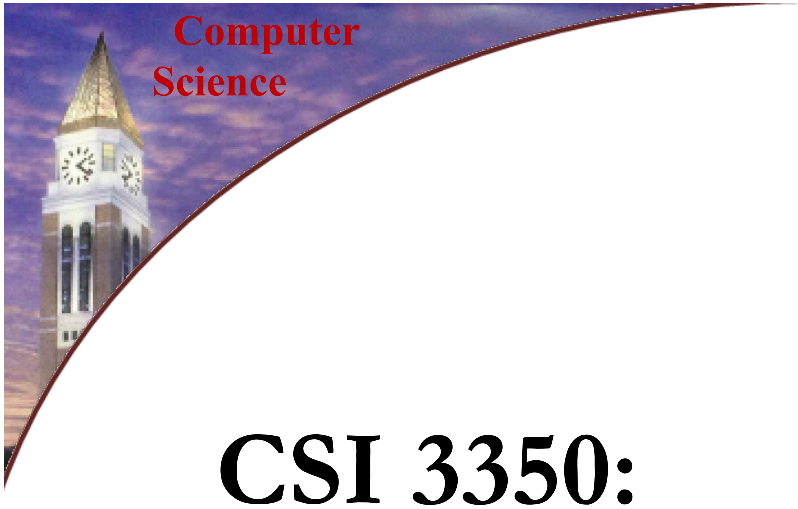
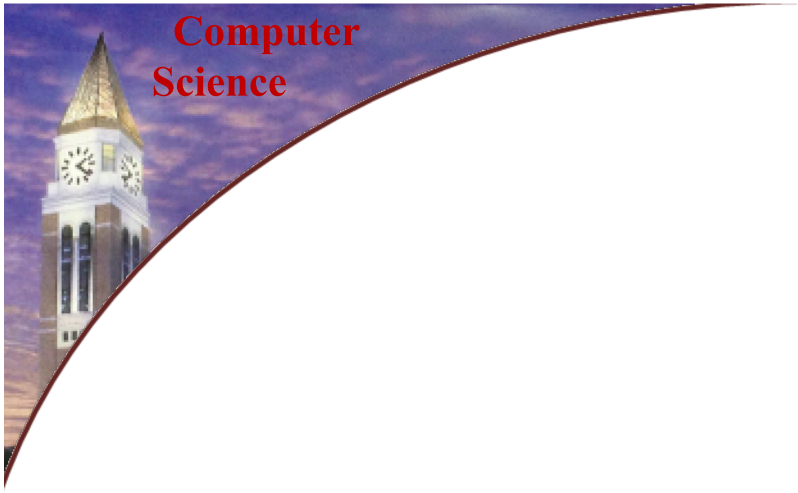


CSI 3350: PROGRAMMING LANGUAGES

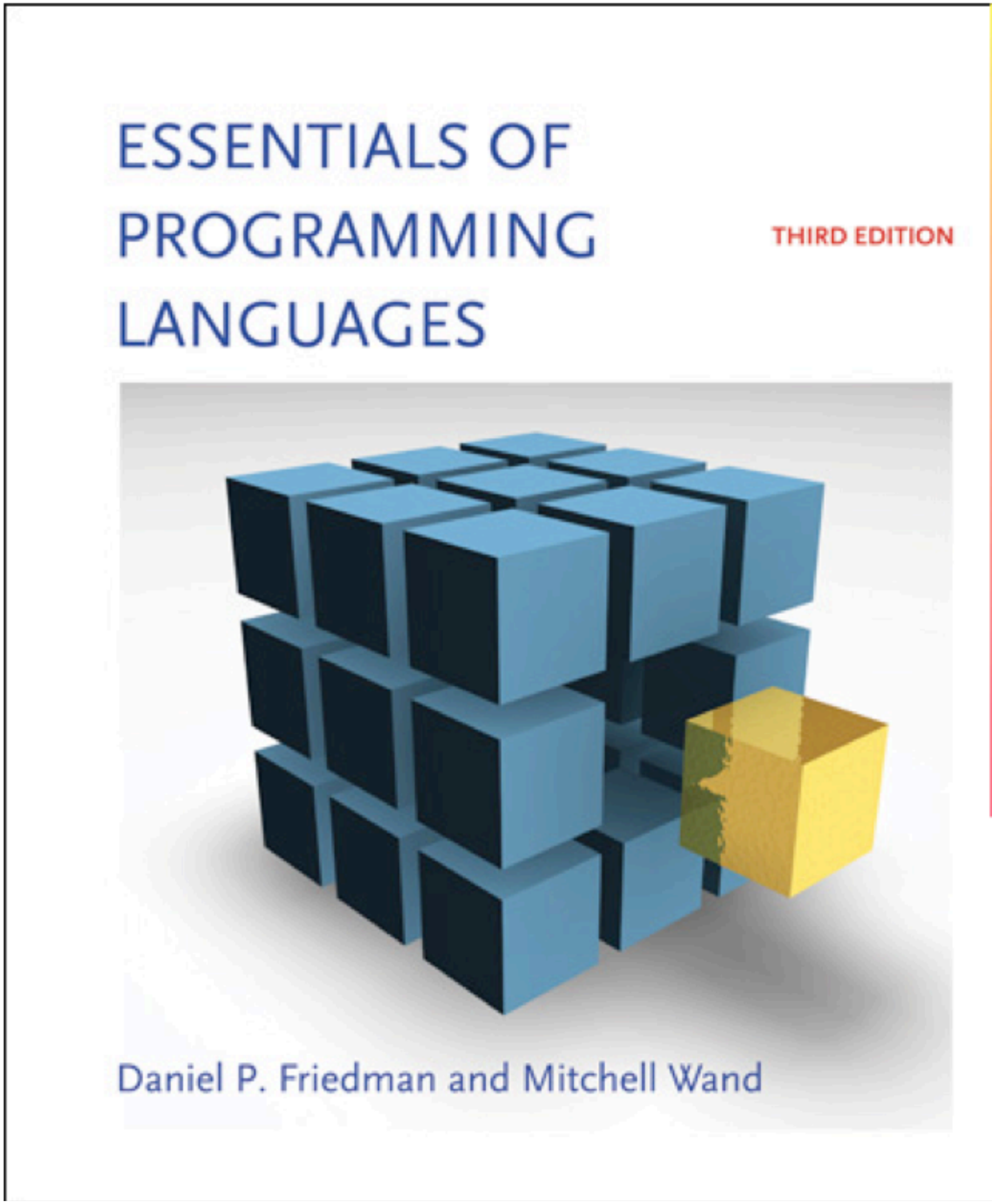
Department of Computer Science & Engineering

Oakland University



HW1 due this Friday (Sep 20) @ 11:55pm

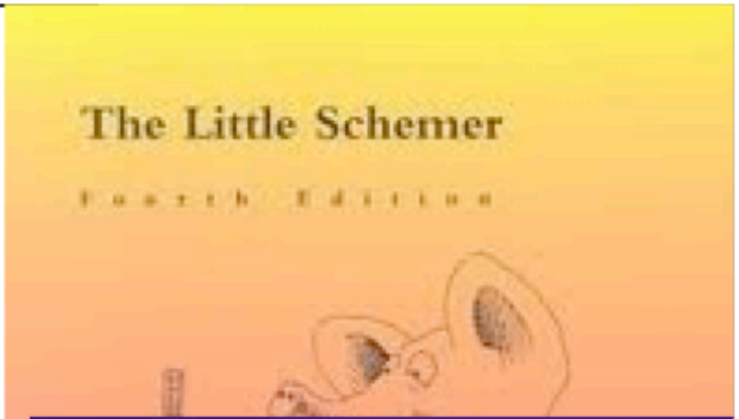
You can update your submission before the deadline



ESSENTIALS OF PROGRAMMING LANGUAGES

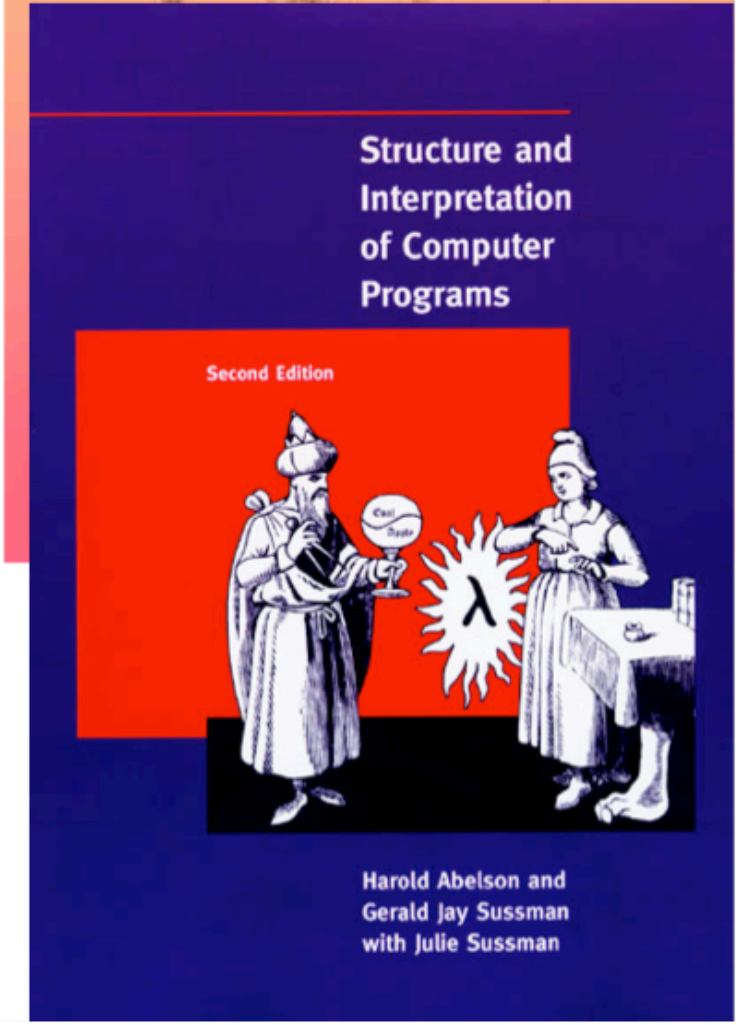
THIRD EDITION

Daniel P. Friedman and Mitchell Wand



The Little Schemer

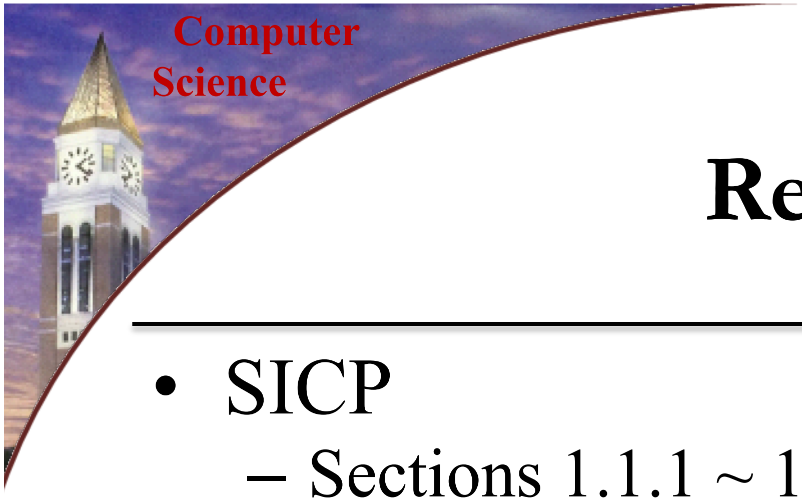
Fourth Edition



Structure and Interpretation of Computer Programs

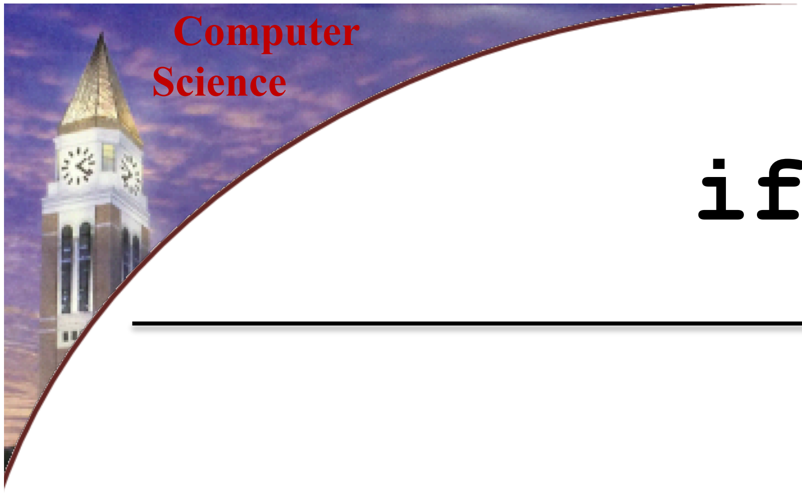
Second Edition

Harold Abelson and
Gerald Jay Sussman
with Julie Sussman



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]

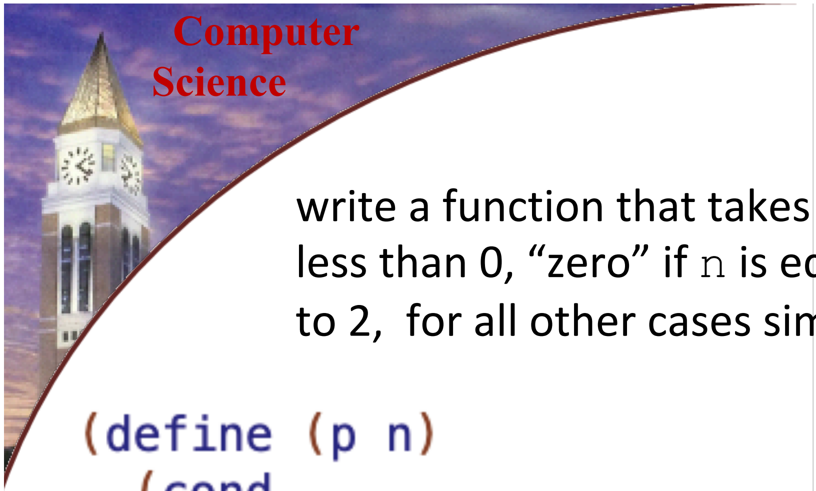


`if` and `cond`

(**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 ?

Making Use of Number Types

Factorial

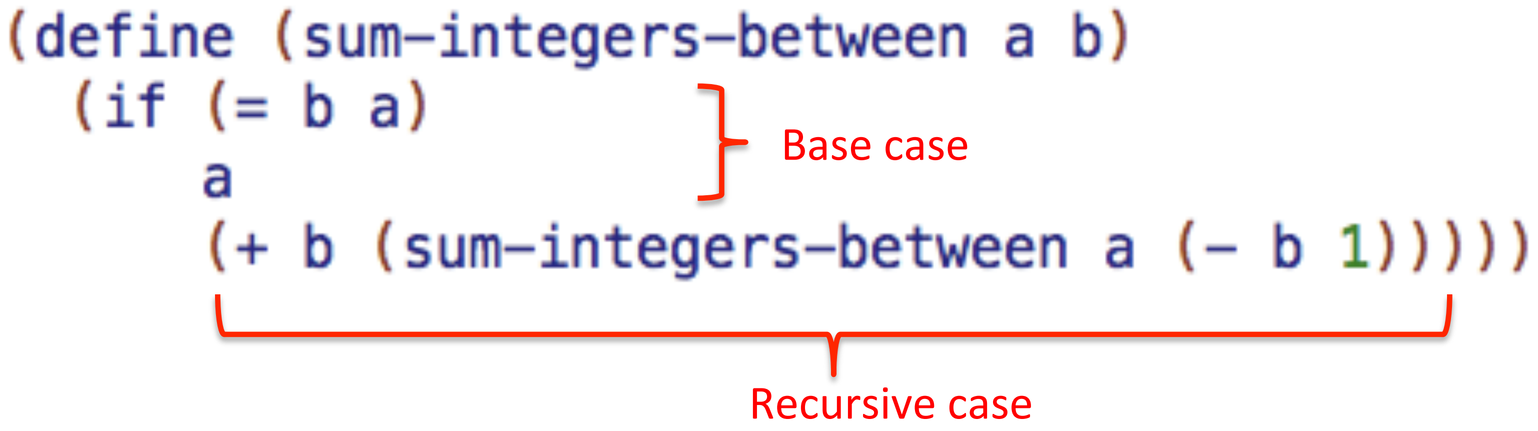
```
(define
     (fact n )
        (if
            (= n 0)
            1
            (* n (fact (- n 1)))
        )
)
```

function signature

function body

Assume: a is not greater than b

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

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

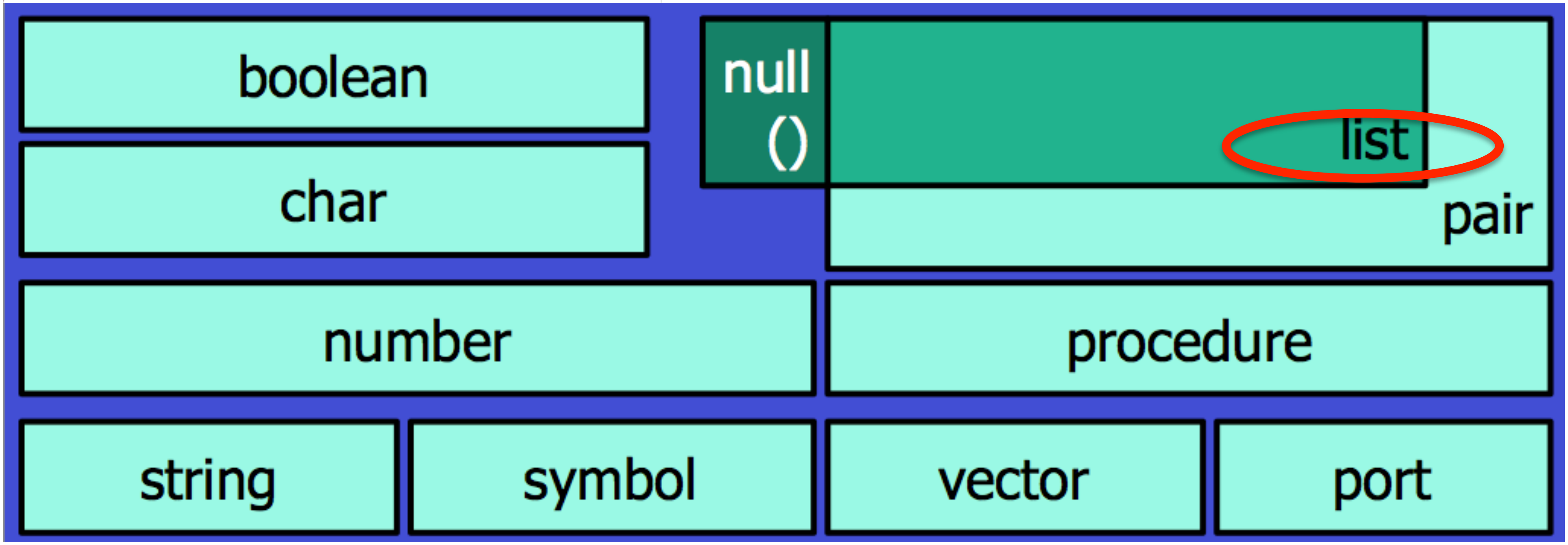


```
(define (sum-integers-between a b)
  (if (= b a)
      a
      (+ b (sum-integers-between a (- b 1)))))
```

Base case

Recursive case

Data Types in Scheme



List Manipulation

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

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)) ) → ?
```

List Manipulation

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

List Manipulation

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

a compound function!

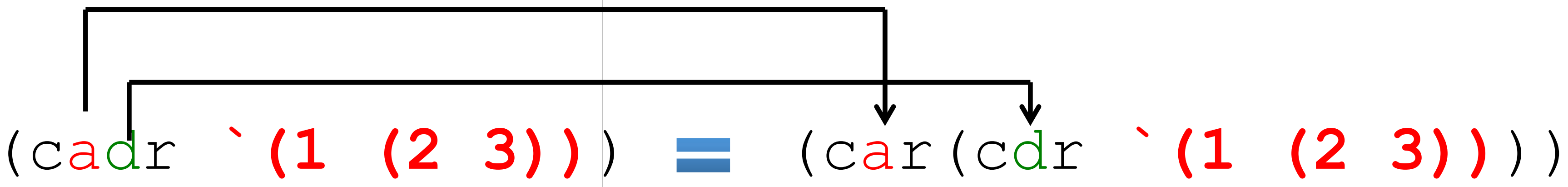
List Manipulation

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

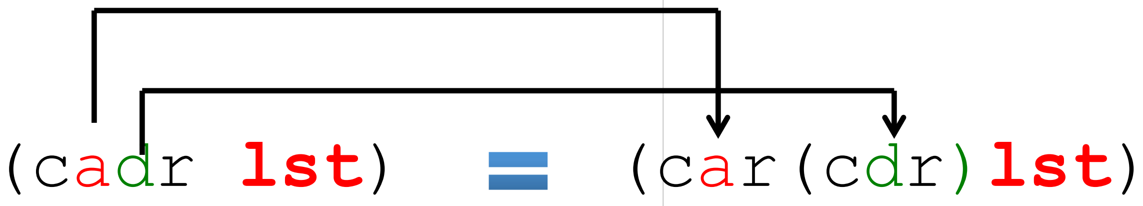
List Manipulation

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

order of execution

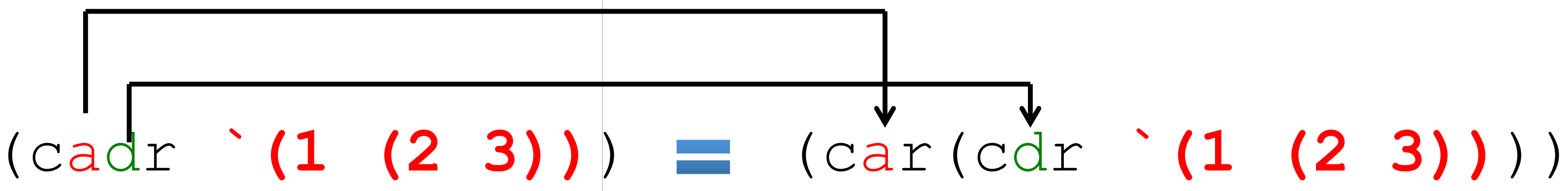


```
(cadr `(1 (2 3))) = (car(cdr `(1 (2 3))))
```



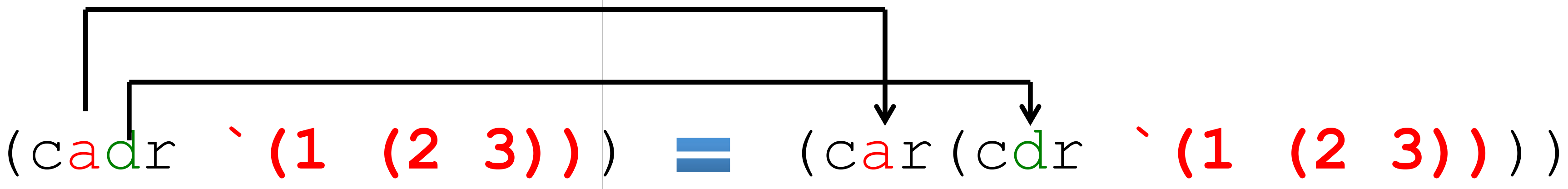
```
(cadr lst)   =   (car(cdr)lst)
```

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



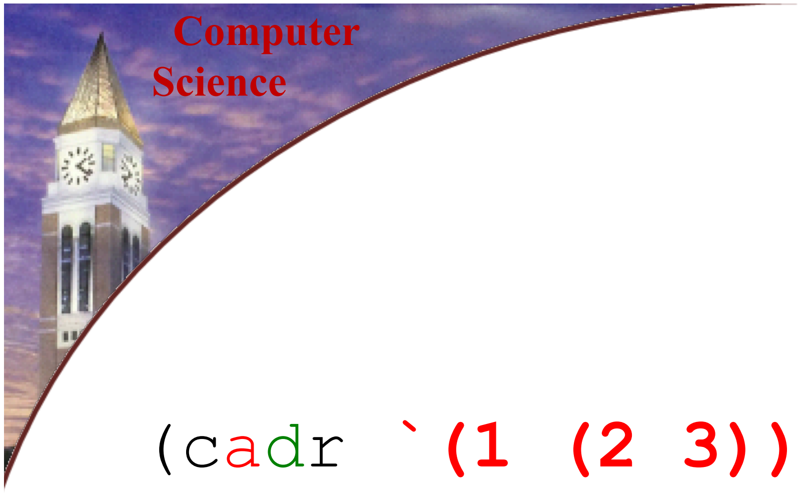
```
(cadr `(1 (2 3))) = (car(cdr `(1 (2 3))))
```

?



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

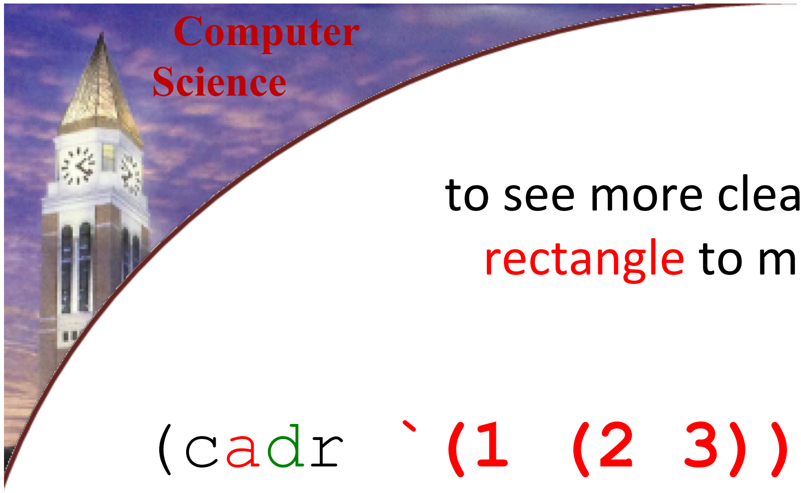
`(2 3)



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

to see more clearly how it works out, we use a red dotted rectangle to mark the immediate next computing step

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



to see more clearly how it works out, we use a red dotted rectangle to mark the immediate next computing step

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

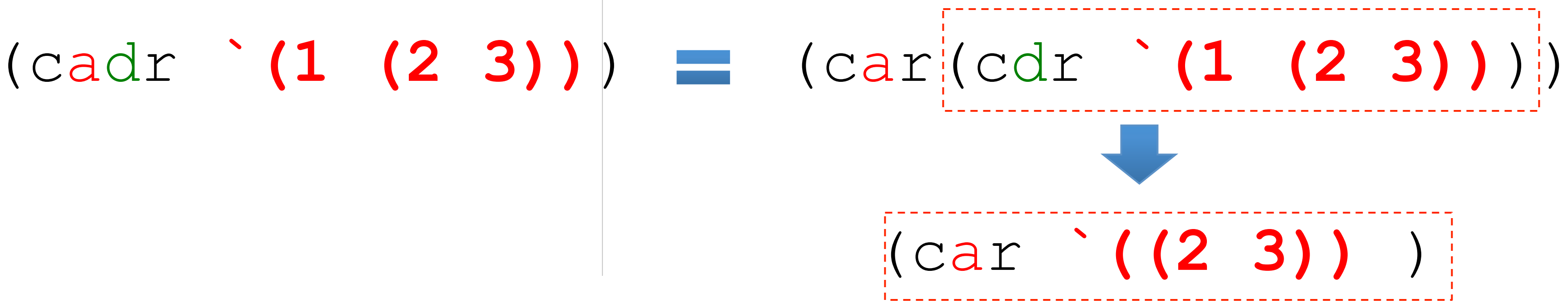
to see more clearly how it works out, we use a red dotted rectangle to mark the immediate next computing step

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

to see more clearly how it works out, we use a red dotted rectangle to mark the immediate next computing step

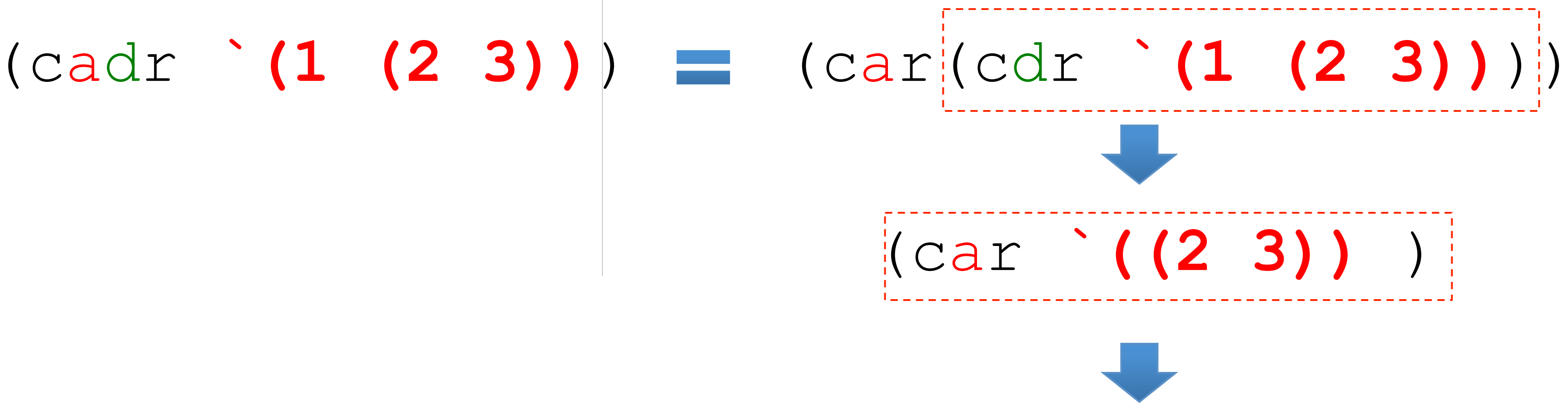
```
(cadr `(1 (2 3))) = (car (cdr `(1 (2 3))))
                             ↓
                     (car `((2 3)) )
```

to see more clearly how it works out, we use a red dotted rectangle to mark the immediate next computing step



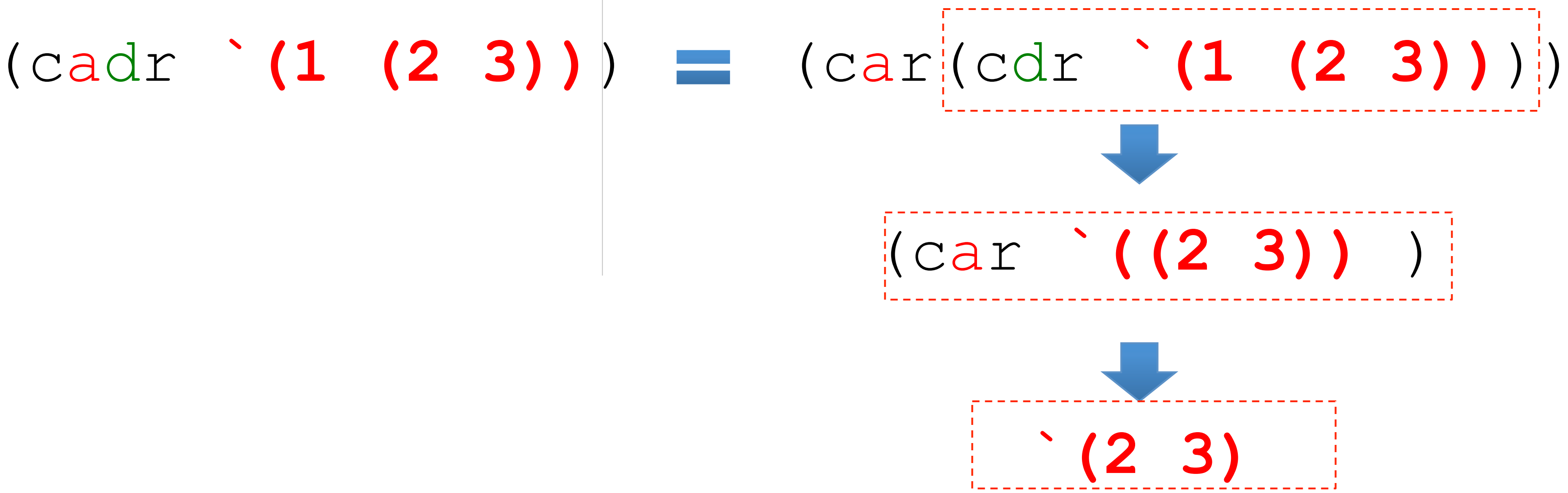
```
(cadr `(1 (2 3)))  =  (car (cdr `(1 (2 3))))

                      (car `((2 3)) )
```

to see more clearly how it works out, we use a red dotted rectangle to mark the immediate next computing step

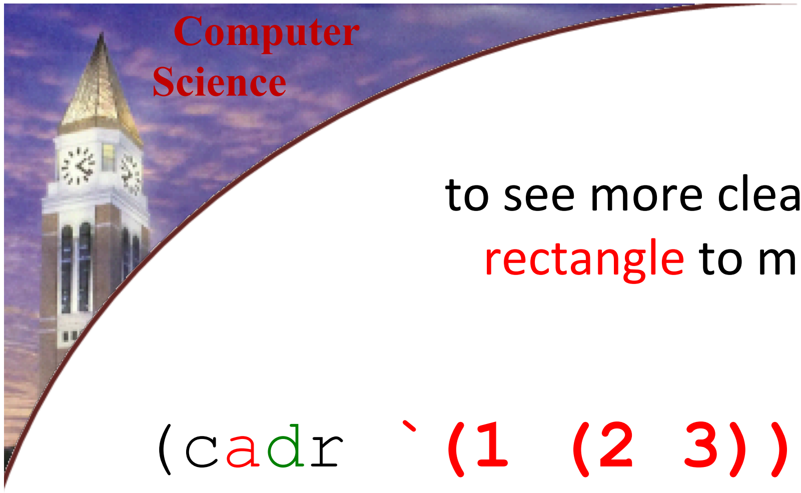


```
(cadr `(1 (2 3)))  =  (car (cdr `(1 (2 3))))

                      (car `((2 3)) )
```

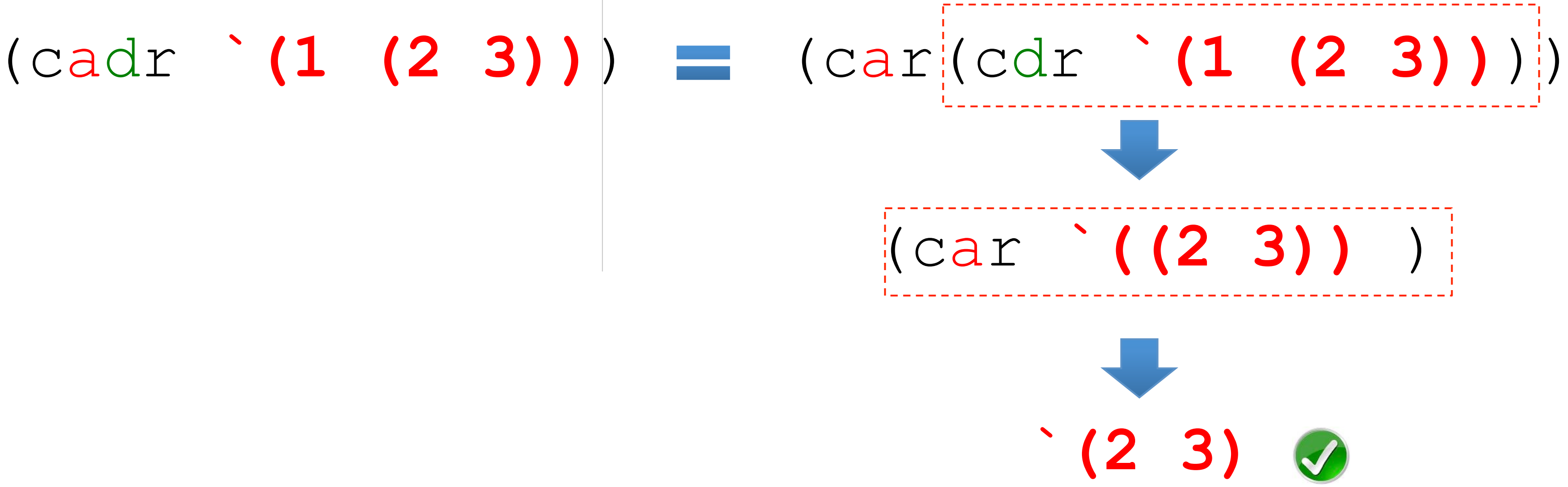
to see more clearly how it works out, we use a red dotted rectangle to mark the immediate next computing step



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

                       (car `((2 3)) )

                           `(2 3)
```



to see more clearly how it works out, we use a red dotted rectangle to mark the immediate next computing step

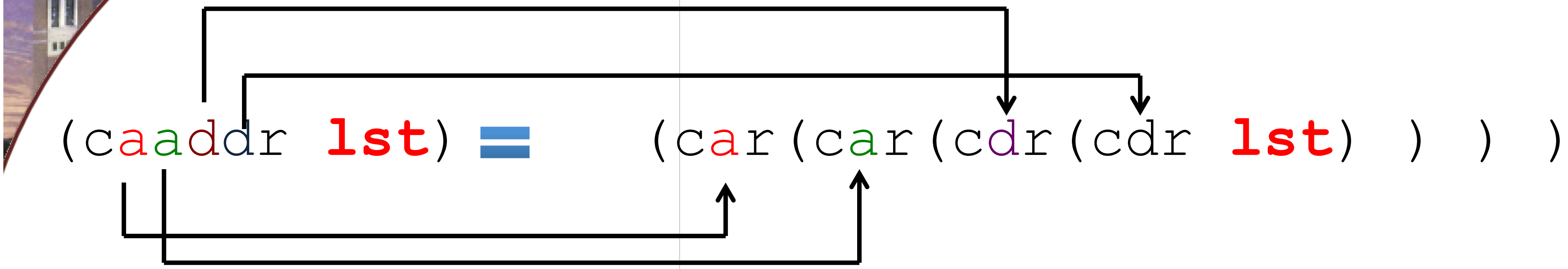


```
(cadr `(1 (2 3))) = (car (cdr `(1 (2 3))))

                  (car `((2 3)) )

                        `(2 3) ✓
```

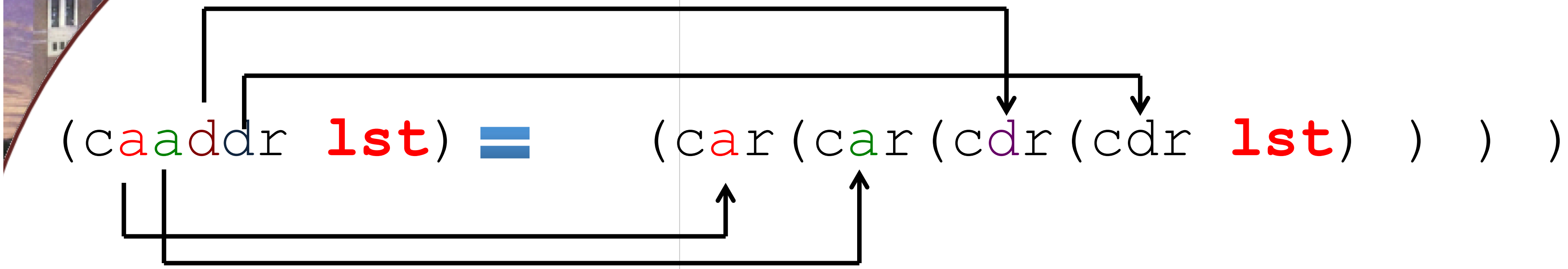
Computer Science

`(caaddr lst)` = ?

```
(caaddr lst) =    (car(car(cdr(cdr lst) ) ) )
```

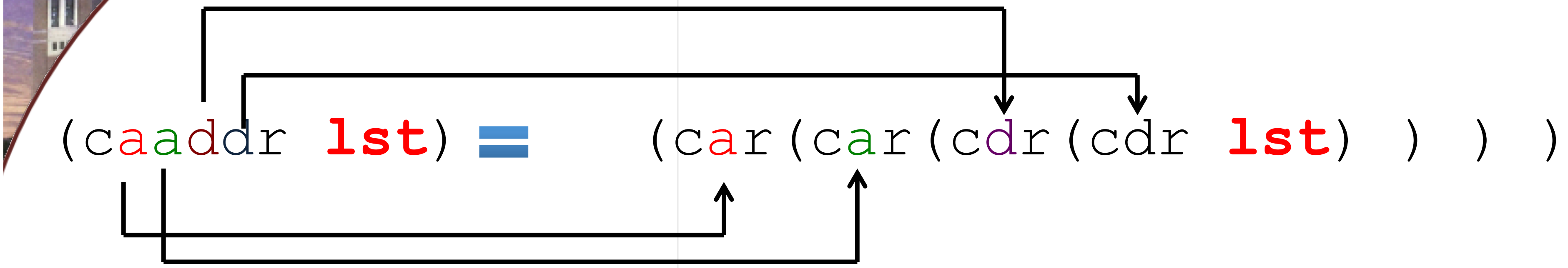


```
(caaddr lst) = (car(car(cdr(cdr lst) ) ) )
```



```
(caaddr lst) = (car(car(cdr(cdr lst) ) ) )
```

```
(caaddr '(1 2 3 4)) = ?
```



```
(caaddr lst) =    (car(car(cdr(cdr lst) ) ) )
```

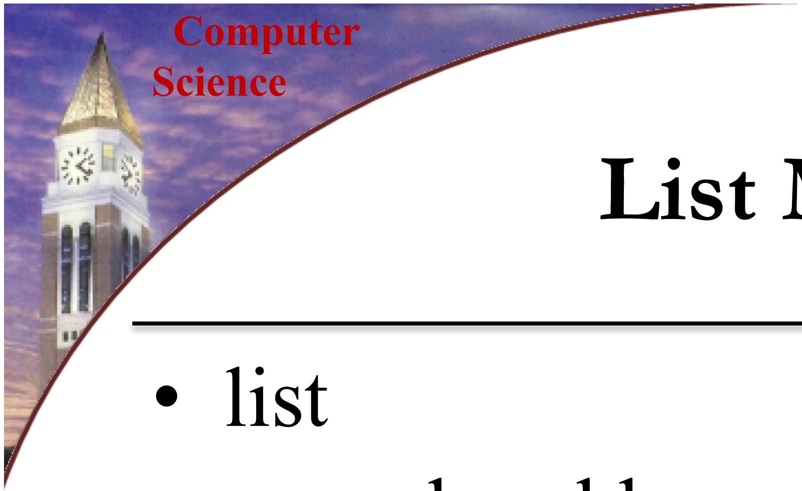
```
(caaddr '(1 2 (3) 4)) =  ?
```

List Manipulation

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

List Manipulation

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

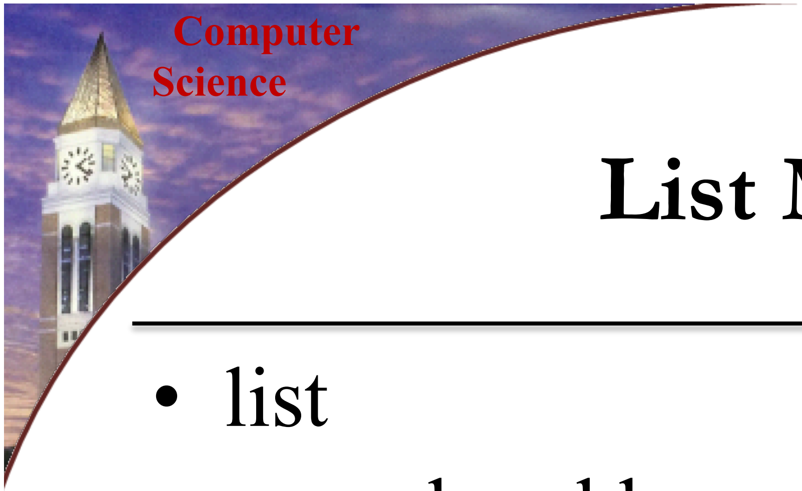


List Manipulation

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

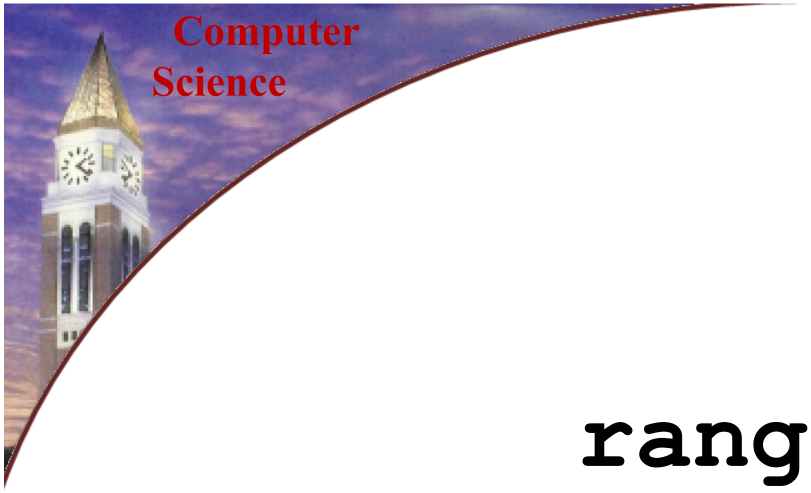
List Manipulation

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



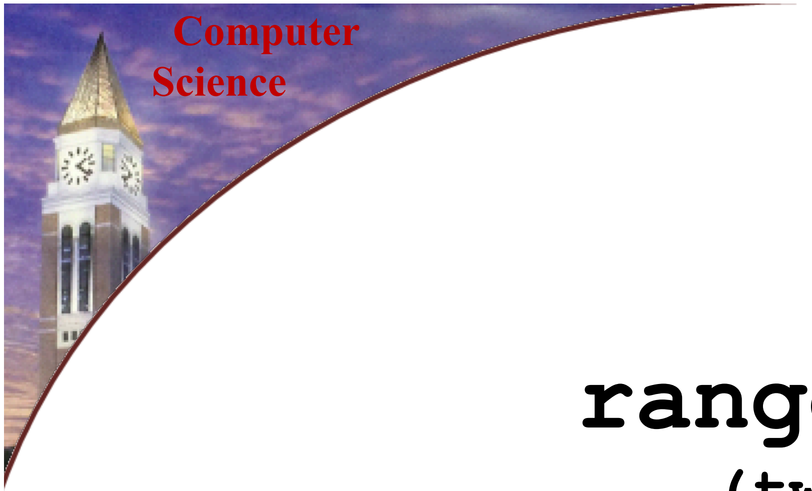
List Manipulation

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



range function

```
(range 1 3) =>  ‘( 1 2)
```



range function

(two versions)

```
(range 1 3) =>  ‘( 1 2)
(range 2) =>  ‘( 0 1)
```

zip function

zip function

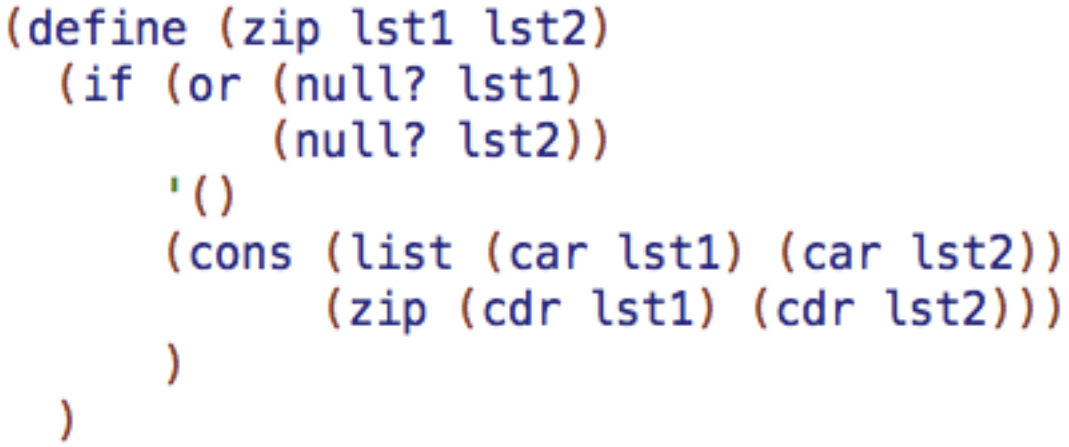
zip function

```
(zip `(3 4 2) `(5 9 7) )     ==> `((3 5) (4 9) (2 7))

(zip `(4 2) `(9 7) )         ==>  `((4 9) (2 7))

(zip `(2 3 1) `(9 2) )       ==>  `( (2 9) (3 2) )
```

zip function

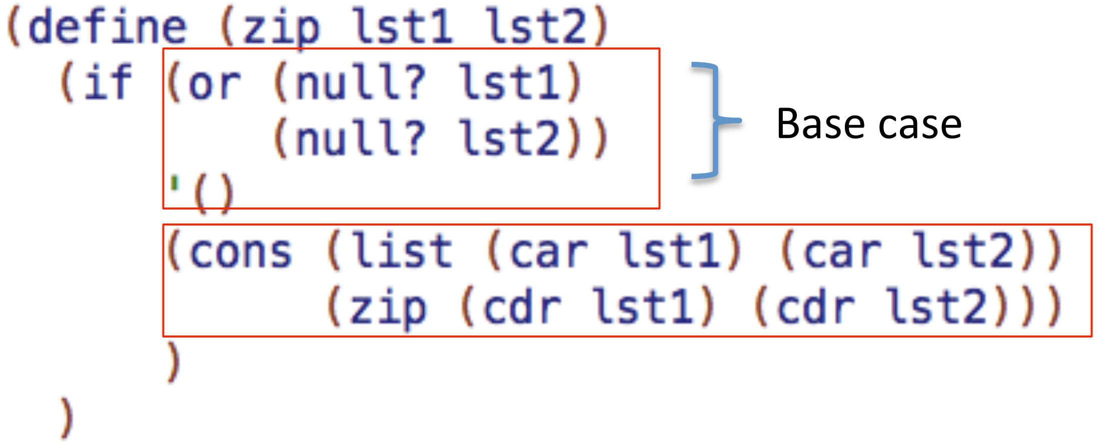
```
(zip `(3 4 2) `(5 9 7) )    ==> `((3 5) (4 9) (2 7))

(zip `(4 2) `(9 7) )        ==>  `((4 9) (2 7))

(zip `(2 3 1) `(9 2) )      ==>  `( (2 9) (3 2) )

(zip `() `(3 1 4 1 5 9) )   ==>  `()
```

zip function



```
(define (zip lst1 lst2)
  (if (or (null? lst1)
          (null? lst2))
      '()
      (cons (list (car lst1) (car lst2))
            (zip (cdr lst1) (cdr lst2)))
      )
  )
```

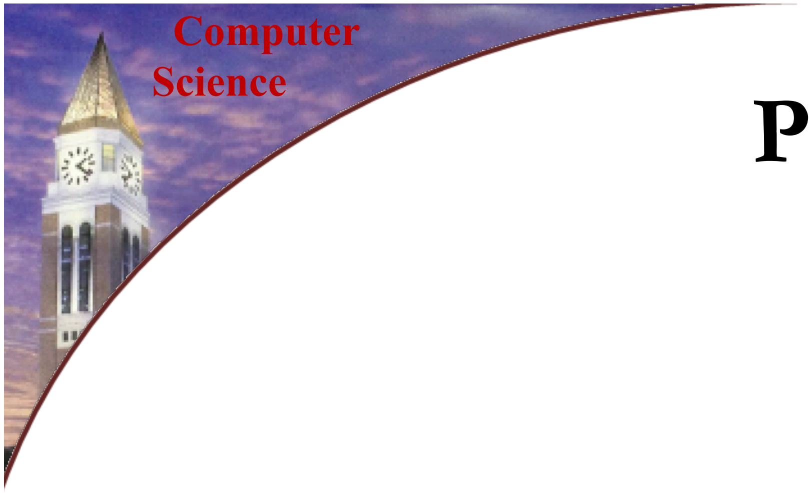
zip function



```
(define (zip lst1 lst2)
  (if (or (null? lst1)
          (null? lst2))
      '()
      (cons (list (car lst1) (car lst2))
            (zip (cdr lst1) (cdr lst2)))
      )
  )
```

Base case

Recursive case

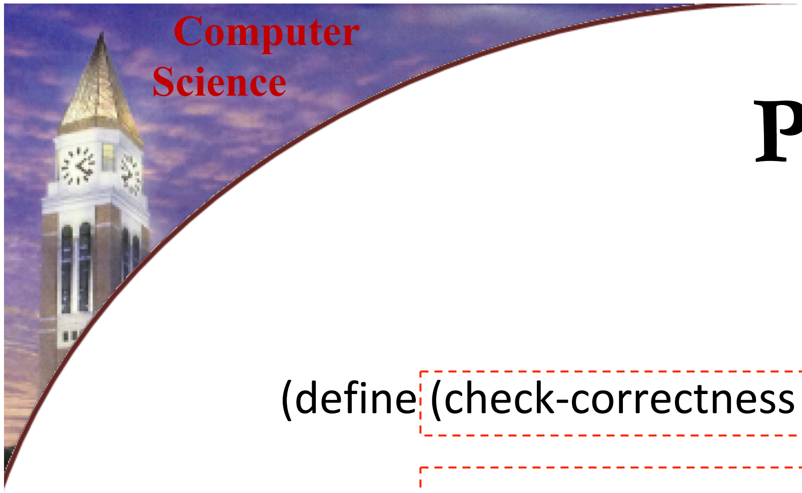


P15 of HW1

P15 of HW1

```
(define (check-correctness pair)

        (
           ; your code goes here !

        )
)
```

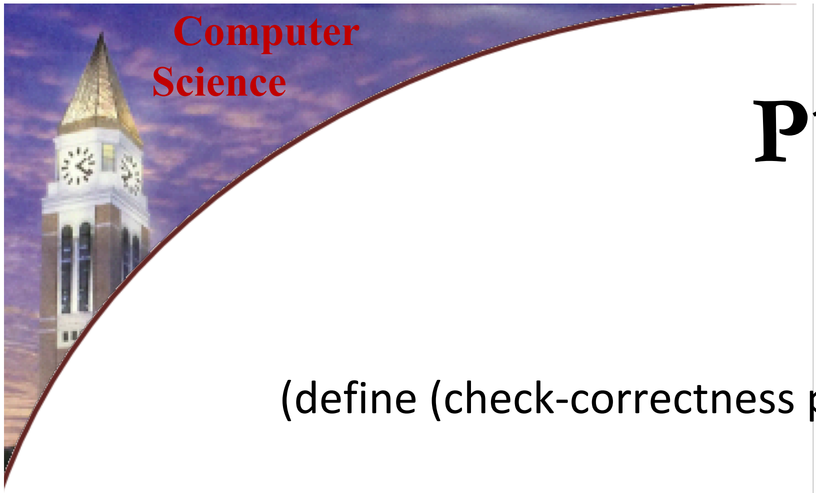
if the symbol is 'answer-to-everything but the number is not 42,
then raise an exception with the error message that you create with the
    function defined for the previous question;
your function returns #t only when the pair is exactly '(answer-to-everything 42);
for all other cases your function should return #f.



P15 of HW1

```
(define (check-correctness pair)

          (
               ; your code goes here !

          )
)
```

if the symbol is 'answer-to-everything but the number is not 42,
then raise an exception with the error message that you create with the
    function defined for the previous question;
your function returns #t only when the pair is exactly '(answer-to-everything 42);
for all other cases your function should return #f.



P15 of HW1

```
(define (check-correctness pair)

        (
           ; your code goes here !

        )
)
```

if the symbol is 'answer-to-everything but the number is not 42,
then raise an exception with the error message that you create with the
    function defined for the previous question;
your function returns #t only when the pair is exactly '(answer-to-everything 42);
for all other cases your function should return #f.

P15 of HW1

```
(define (check-correctness pair)

        (
           ; your code goes here !

        )
)
```

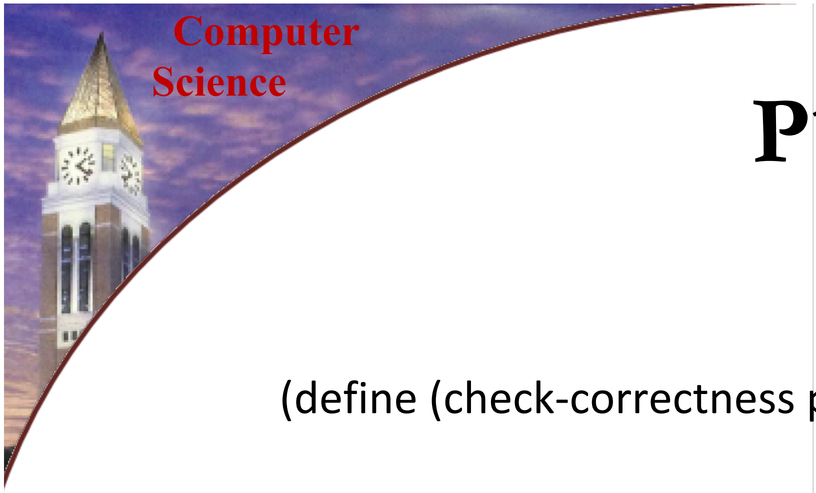
if the symbol is 'answer-to-everything but the number is not 42,
then raise an exception with the error message that you create with the
    function defined for the previous question;
your function returns #t only when the pair is exactly '(answer-to-everything 42);
for all other cases your function should return #f.

P15 of HW1

```
(define (check-correctness pair)

          (
              ; your code goes here !

          )
)
```

1

if the symbol is 'answer-to-everything but the number is not 42,
then raise an exception with the error message that you create with the
function defined for the previous question;
your function returns #t only when the pair is exactly '(answer-to-everything 42);
for all other cases your function should return #f.



P15 of HW1

```
(define (check-correctness pair)

        (
            ; your code goes here !

        )
)
```

1

if the symbol is 'answer-to-everything but the number is not 42,
then raise an exception with the error message that you create with the
function defined for the previous question;
your function returns **#t** only when the pair is exactly '(answer-to-everything 42);
for all other cases your function should return #f.

P15 of HW1

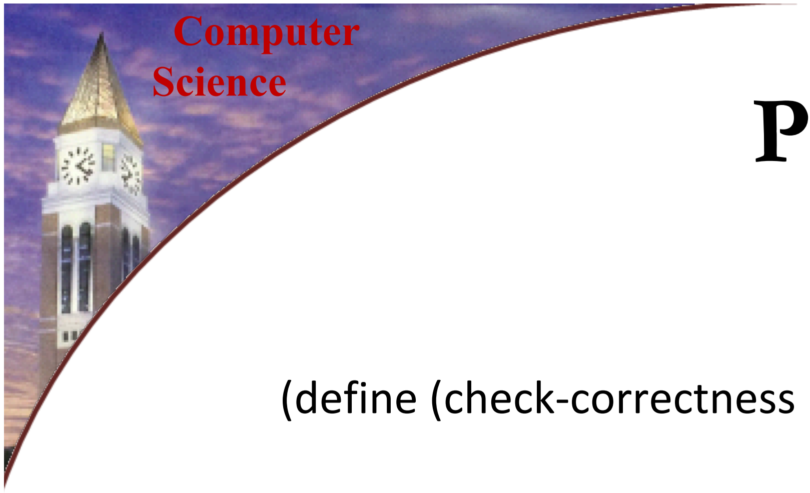
```
(define (check-correctness pair)

        (
            ; your code goes here !

        )
)
```

1

if the symbol is 'answer-to-everything but the number is not 42,
then raise an exception with the error message that you create with the
function defined for the previous question;

2

your function returns **#t** only when the pair is exactly '(answer-to-everything 42);
for all other cases your function should return #f.



P15 of HW1

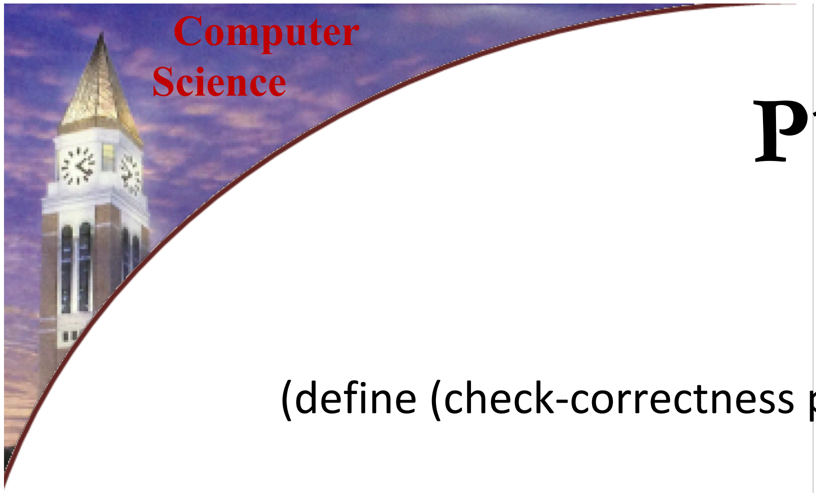
```
(define (check-correctness pair)

        (
            ; your code goes here !

        )
)
```

1

if the symbol is 'answer-to-everything but the number is not 42,
then **raise an exception** with the error message that you create with the
function defined for the previous question;

2

your function returns **#t** only when the pair is exactly '(answer-to-everything 42);
for all other cases your function should return #f.



P15 of HW1

```
(define (check-correctness pair)

        (
            ; your code goes here !

        )
)
```

1

if the symbol is 'answer-to-everything but the number is not 42, 3
then **raise an exception** with the error message that you create with the function defined for the previous question;

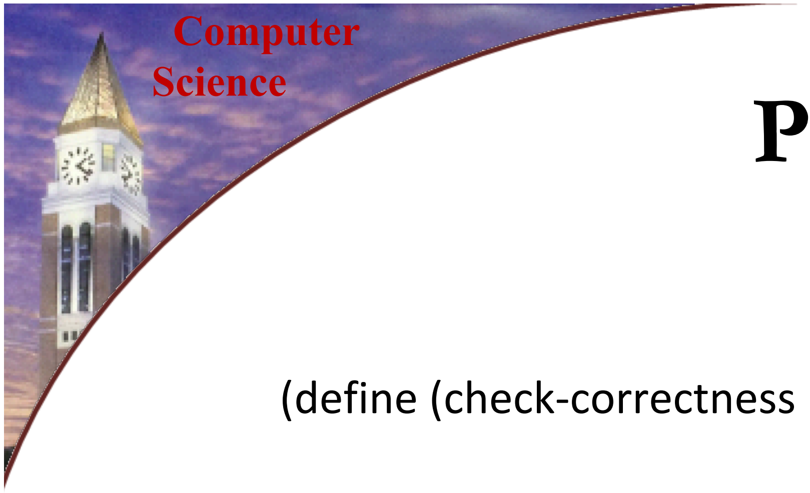
2

your function returns **#t** only when the pair is exactly '(answer-to-everything 42);
for all other cases your function should return #f.

P15 of HW1

```
(define (check-correctness pair)

          (
              ; your code goes here !

          )
)
```

if the symbol is 'answer-to-everything 1 but the number is not 42,
then raise an exception with the error message that you create with the 3
function defined for the previous question;
your function returns #t only when the pair is exactly '(answer-to-everything 42); 2
for all other cases your function should return #f.



P15 of HW1

```
(define (check-correctness pair)

        (
            ; your code goes here !

        )
)
```

1

if the symbol is 'answer-to-everything but the number is not 42, 3

then **raise an exception** with the error message that you create with the function defined for the previous question; 2

your function returns **#t** only when the pair is exactly '(answer-to-everything 42);

for all other cases your function should return **#f**. 4

P15 of HW1

```
(define (check-correctness pair)

          (
              ; your code goes here !

          )
)
```

if the symbol is 'answer-to-everything ① but the number is not 42,
then **raise an exception** with the error message that you create with the ③
function defined for the previous question;
your function returns **#t** only when the pair is exactly '(answer-to-everything 42); ②
for all other cases your function should return **#f**. ④