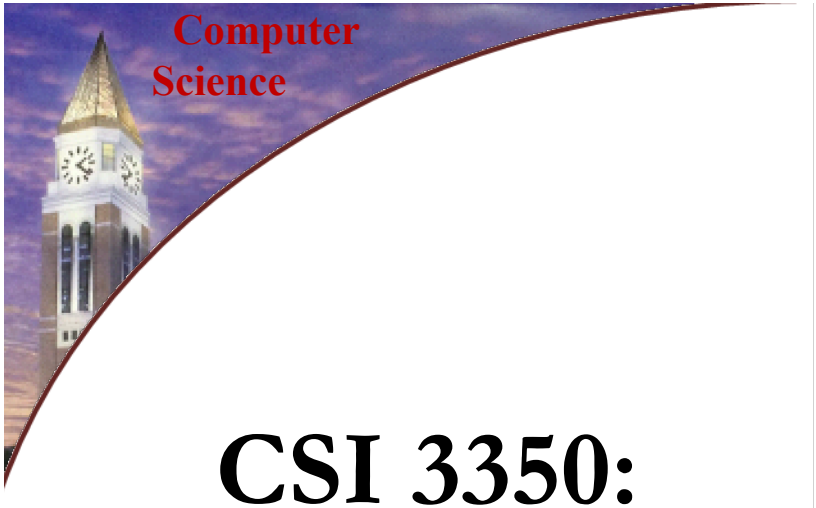
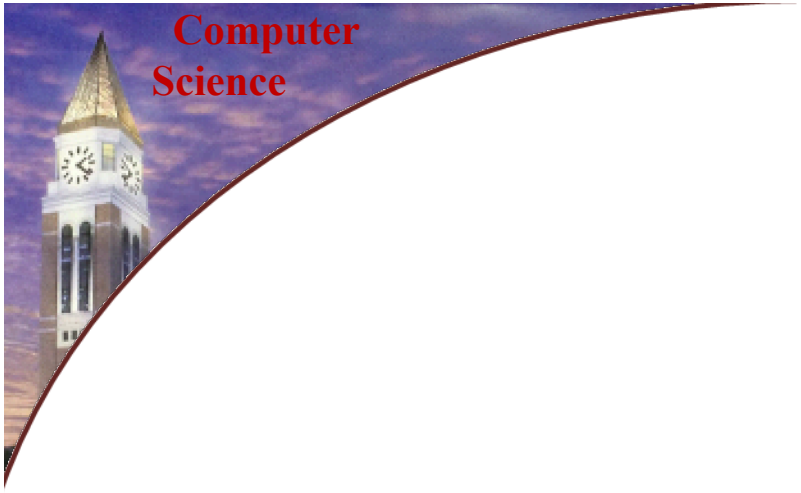


CSI 3350: PROGRAMMING LANGUAGES

Department of Computer Science & Engineering

Oakland University

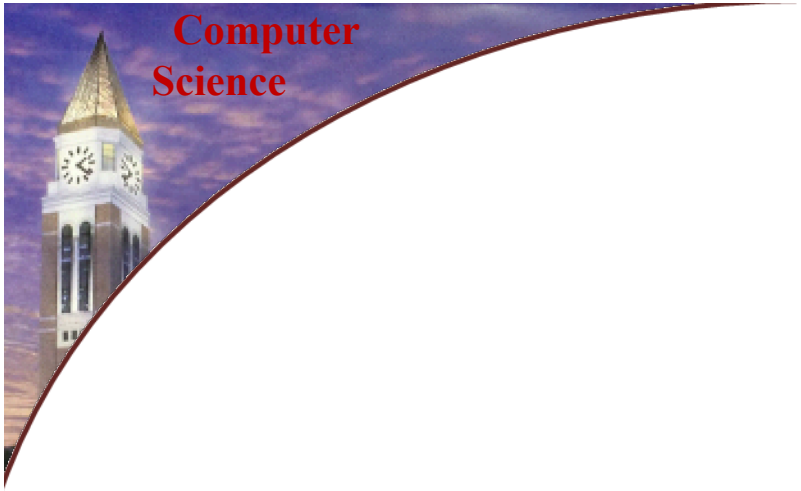


Exam 01

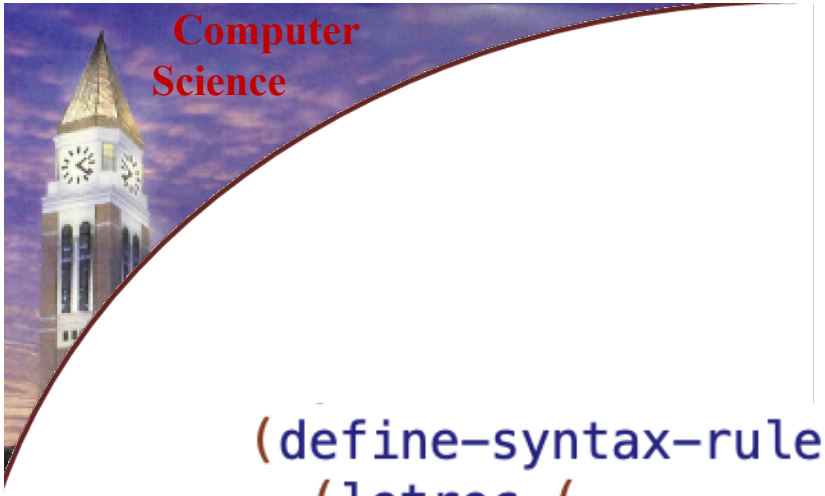
Oct 30 (in class)

(Exam 01 covers HW1~4)

t



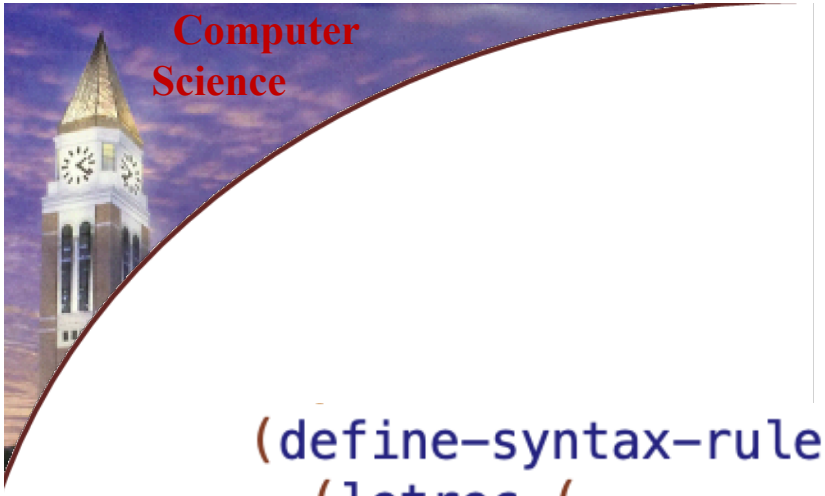
HW 4



HW 4

(Problem 2b)

```
(define-syntax-rule (while condition while-body)
  (letrec (
           (loop (lambda ()
                   (if (not condition)
                       0
                       ( seq while-body (loop) )
                       ))
                 )
           )
    (loop)
    )
  )
```

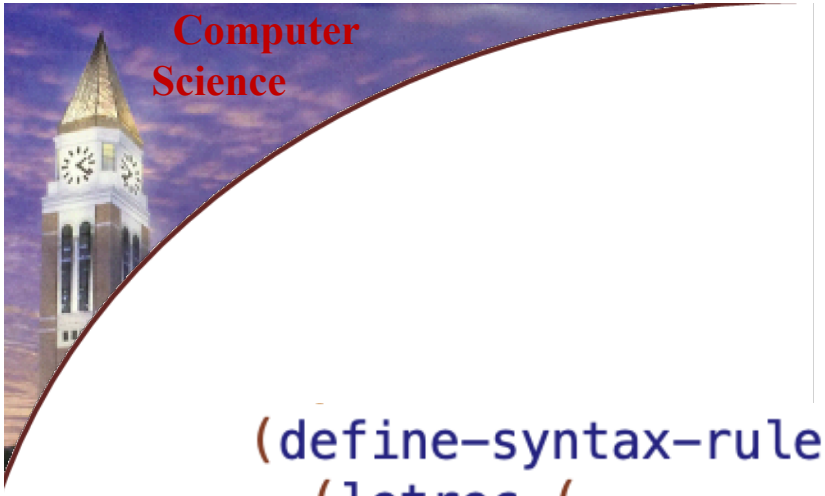


HW 4

(Problem 2b)

```
(define-syntax-rule (while condition while-body)
  (letrec (
           (loop (lambda ()
                   (if (not condition)
                       0
                       ( seq while-body (loop) )
                       ))
                 )
           )
    (loop)
    )
  )
```

(1)

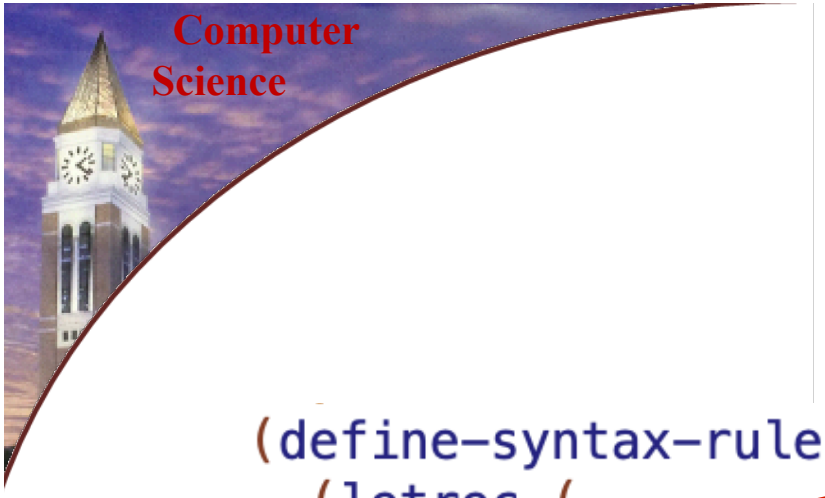


HW 4

(Problem 2b)

```
(define-syntax-rule (while condition while-body)
  (letrec (
           (loop (lambda ()
                   (if (not condition)
                       0
                       ( seq while-body (loop) )
                       ))
                 )
           )
    (loop)
    )
  )
```

(1)

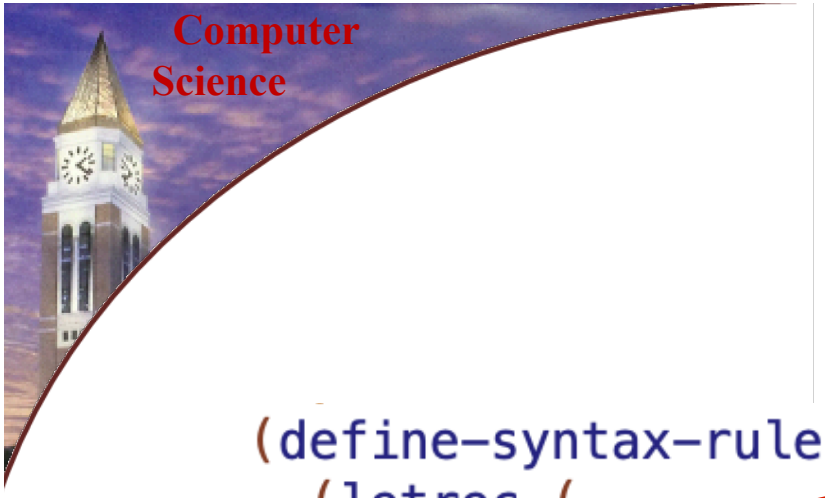


HW 4

(Problem 2b)

```
(define-syntax-rule (while condition while-body)
  (letrec (
           (loop (lambda ()
                   (if (not condition)
                       0
                       ( seq while-body (loop) )
                       ))
                 )
           )
    (loop)
    )
  )
```

(1)

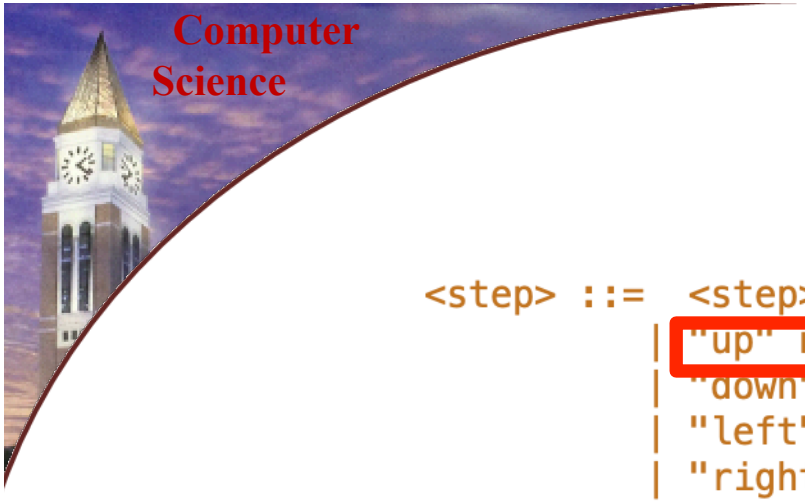


HW 4

(Problem 2b)

```
(define-syntax-rule (while condition while-body)
  (letrec (
           (loop (lambda ()
                   (if (not condition)
                       0
                       ( seq while-body (loop) )
                       ))
                 )
           )
    (loop)
    )
  )
```

(1)

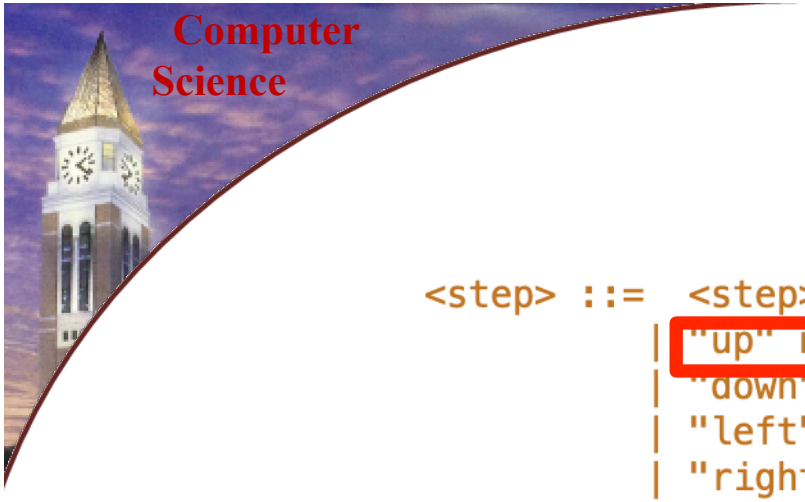


HW 4

(Problem 3)

Constructors

```
<step> ::=  <step>  <step>        "seq-step"
          | "up" number           "up-step"
          | "down" number         "down-step"
          | "left" number         "left-step"
          | "right" number        "right-step"
```

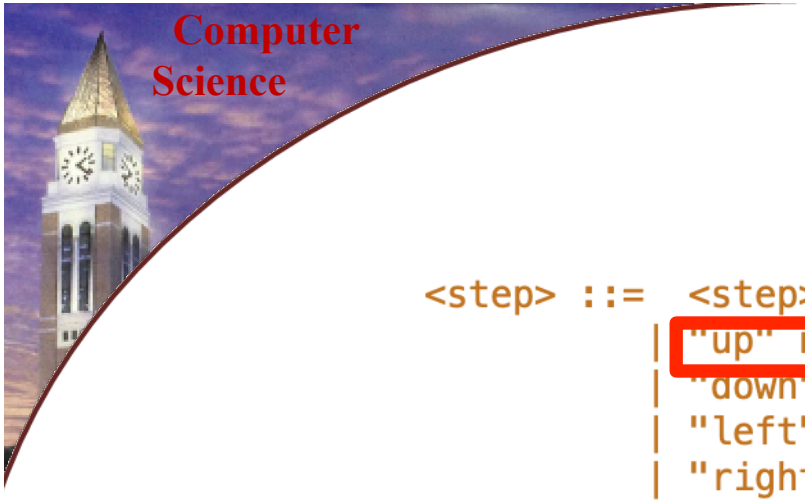


HW 4

(Problem 3)

Constructors
(Data Structure-based)

```
<step> ::=  <step>  <step>        "seq-step"
          | "up" number           "up-step"
          | "down" number         "down-step"
          | "left" number         "left-step"
          | "right" number        "right-step"
```



HW 4

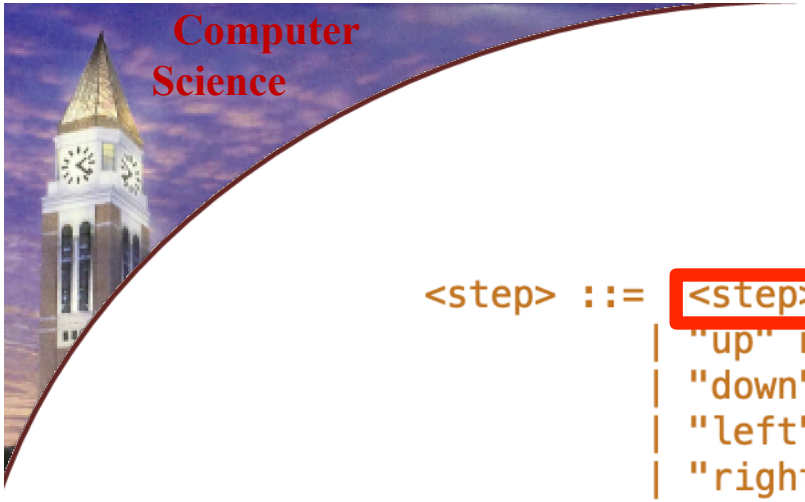
(Problem 3)

Constructors

(Data Structure-based)

```
<step> ::=  <step>  <step>        "seq-step"
          | "up" number           "up-step"
          | "down" number         "down-step"
          | "left" number         "left-step"
          | "right" number        "right-step"
```

```
(define (up-step n)
  (if (number? n)
      (list 'up n)
      (raise (invalid-args-msg "up-step" "number?" n))
      )
  )
```



HW 4

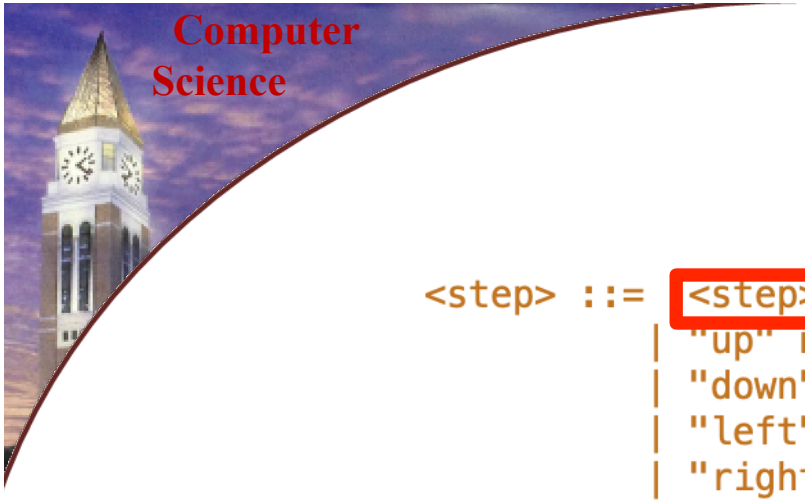
(Problem 3)

Constructors

(Data Structure-based)

```
<step> ::= <step>  <step>        "seq-step"
         | "up" number           "up-step"
         | "down" number         "down-step"
         | "left" number         "left-step"
         | "right" number        "right-step"
```

```
(define (seq-step st-1 st-2)
  (if (and (step? st-1) (step? st-2))
      (list 'seq st-1 st-2)
      (raise (invalid-args-msg "seq-step" "step?" st-1))
      )
  )
```



HW 4

(Problem 3)

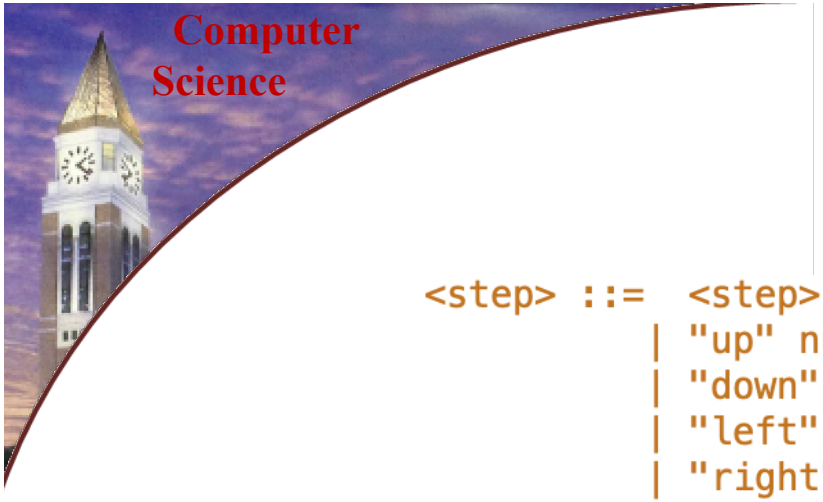
Constructors

(Data Structure-based)

```
<step> ::= <step>  <step>        "seq-step"
         | "up" number           "up-step"
         | "down" number         "down-step"
         | "left" number         "left-step"
         | "right" number        "right-step"
```

```
(define (seq-step st-1 st-2)
  (if (and (step? st-1) (step? st-2))
      (list 'seq st-1 st-2)
      (raise (invalid-args-msg "seq-step" "step?" st-1))
      )
  )
```

```
(define (step? st)
  (or (single-step? st)
      (seq-step? st)
      )
  )
```



HW 4

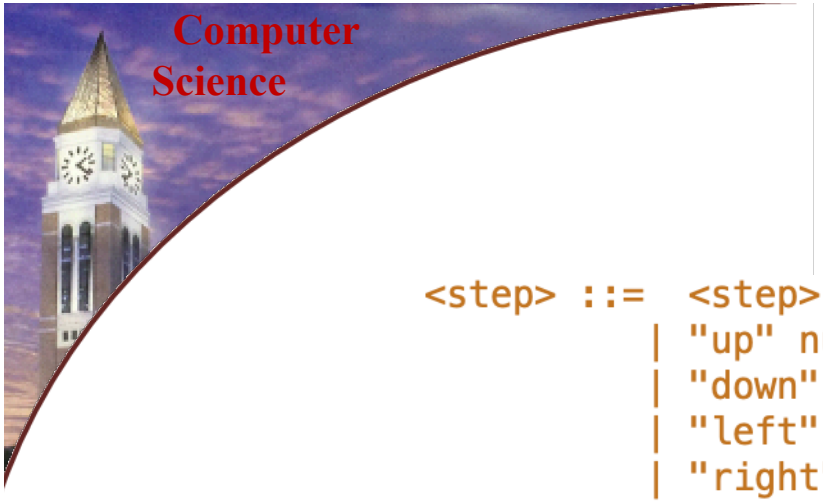
Predicates

(Data Structure-based)

(Problem 3)

```
<step> ::=  <step>  <step>        "seq-step"
          | "up" number           "up-step"
          | "down" number         "down-step"
          | "left" number         "left-step"
          | "right" number        "right-step"
```

```
(define (step? st)
  (or (single-step? st)
      (seq-step? st)
      )
  )
```



HW 4

Predicates
(Data Structure-based)

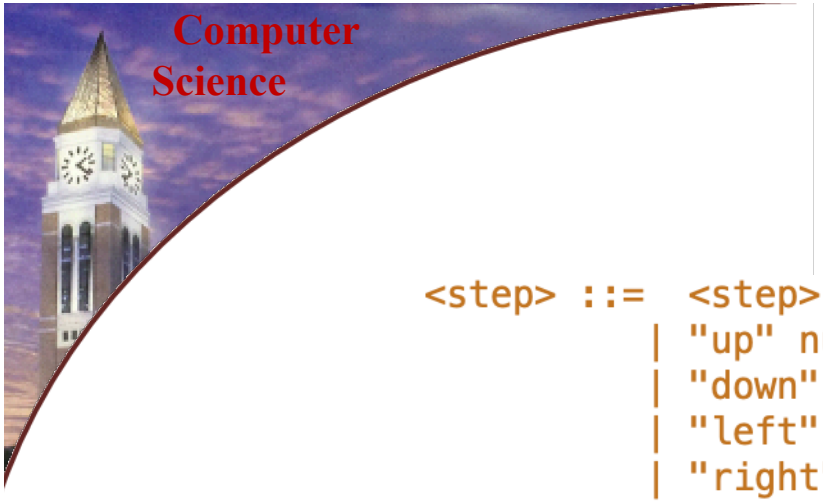
(Problem 3)

```
<step> ::=  <step>  <step>        "seq-step"
          | "up" number           "up-step"
          | "down" number         "down-step"
          | "left" number         "left-step"
          | "right" number        "right-step"
```

```
(define (step? st)
  (or (single-step? st)
      (seq-step? st)
      )
  )
```

```
(define (single-step? st)
  (or (up-step? st)
      (down-step? st)
      (right-step? st)
      (left-step? st)
      )
  )
```

```
(define (seq-step? st)
  (and (list? st)
       (equal? (length st) 3)
       (equal? (car st) 'seq)
       (step? (cadr st))
       (step? (caddr st))
    )
  )
```



HW 4

(Problem 3)

Predicates

(Data Structure-based)

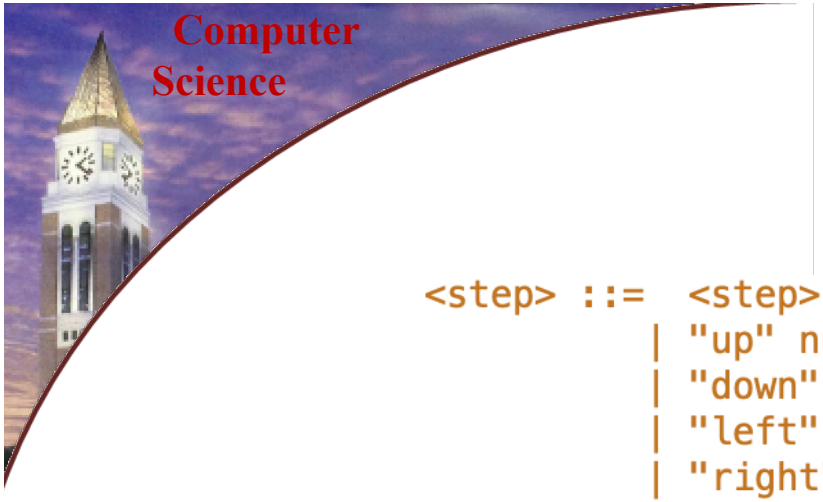
```
<step> ::=  <step>  <step>        "seq-step"
          | "up" number           "up-step"
          | "down" number         "down-step"
          | "left" number         "left-step"
          | "right" number        "right-step"
```

```
(define (step? st)
  (or (single-step? st)
      (seq-step? st)
      )
  )
```

```
(define (single-step? st)
  (or (up-step? st)
      (down-step? st)
      (right-step? st)
      (left-step? st)
      )
  )
```

```
(define (up-step? st)
  (valid-single-step? st 'up)
)
```

```
(define (seq-step? st)
  (and (list? st)
       (equal? (length st) 3)
       (equal? (car st) 'seq)
       (step? (cadr st))
       (step? (caddr st))
    )
)
```



HW 4

(Problem 3)

Predicates

(Data Structure-based)

```
<step> ::=  <step>  <step>        "seq-step"
          | "up" number           "up-step"
          | "down" number         "down-step"
          | "left" number         "left-step"
          | "right" number        "right-step"
```

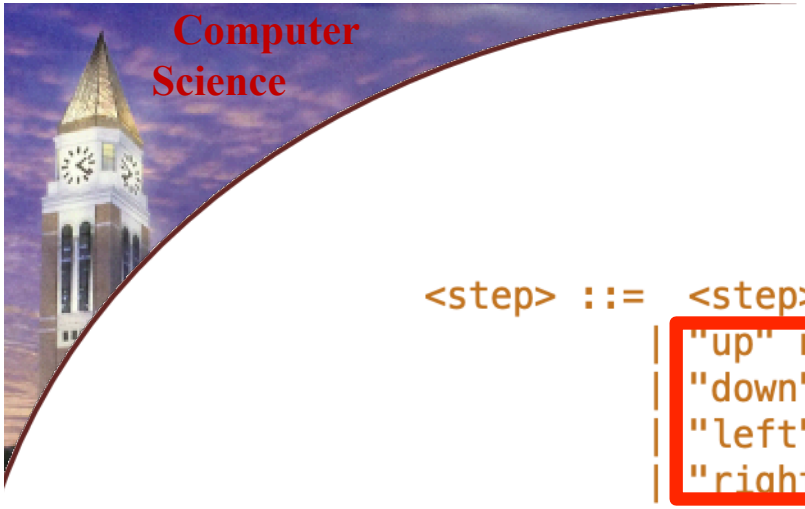
```
(define (step? st)
  (or (single-step? st)
      (seq-step? st)
      )
  )
```

```
(define (single-step? st)
  (or (up-step? st)
      (down-step? st)
      (right-step? st)
      (left-step? st)
      )
  )
```

```
(define (seq-step? st)
  (and (list? st)
       (equal? (length st) 3)
       (equal? (car st) 'seq)
       (step? (cadr st))
       (step? (caddr st))
    )
  )
```

```
(define (up-step? st)
  (valid-single-step? st 'up)
)
```

```
(define (valid-single-step? st marker)
  (and (list? st)
       (equal? (length st) 2)
       (equal? (car st) marker)
       (number? (cadr st))
       )
  )
```



HW 4

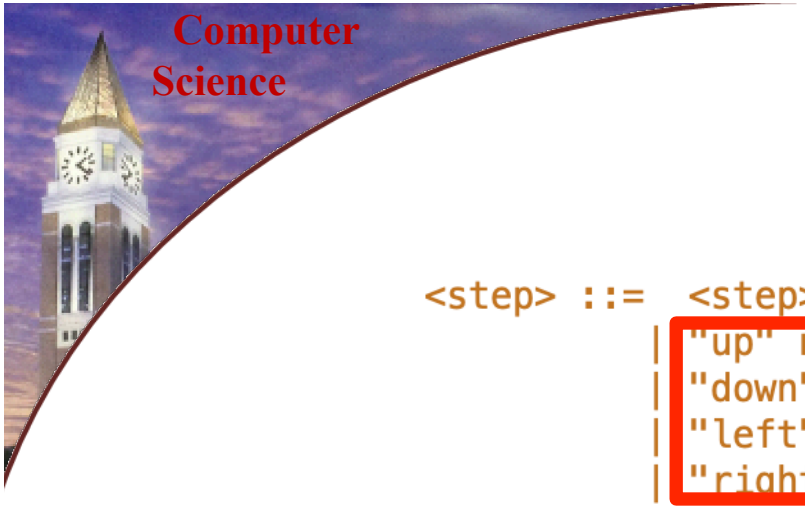
Extractors

(Data Structure-based)

(Problem 3)

```
<step> ::=  <step>  <step>        "seq-step"
          | "up" number           "up-step"
          | "down" number         "down-step"
          | "left" number         "left-step"
          | "right" number        "right-step"
```

```
(define (single-step->n st)
  (if (single-step? st)
      (cadr st)
      (raise (invalid-args-msg "single-step->n" "single-step?" st))
      )
  )
```



HW 4

Extractors

(Data Structure-based)

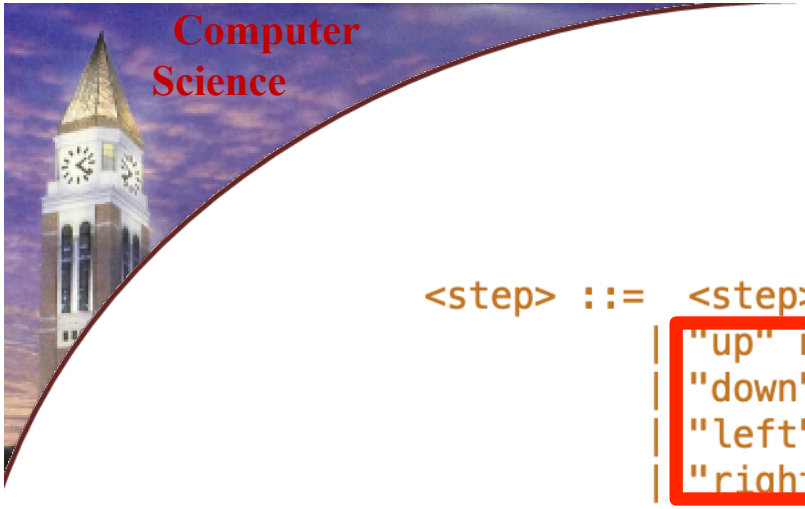
(Problem 3)

```
<step> ::=  <step>  <step>        "seq-step"
          | "up" number           "up-step"
          | "down" number         "down-step"
          | "left" number         "left-step"
          | "right" number        "right-step"
```

```
(define (single-step->n st)
  (if (single-step? st)
      (cadr st)
      (raise (invalid-args-msg "single-step->n" "single-step?" st))
      )
  )
```

4-in-1 !

(the same !)



HW 4

Extractors
(Data Structure-based)

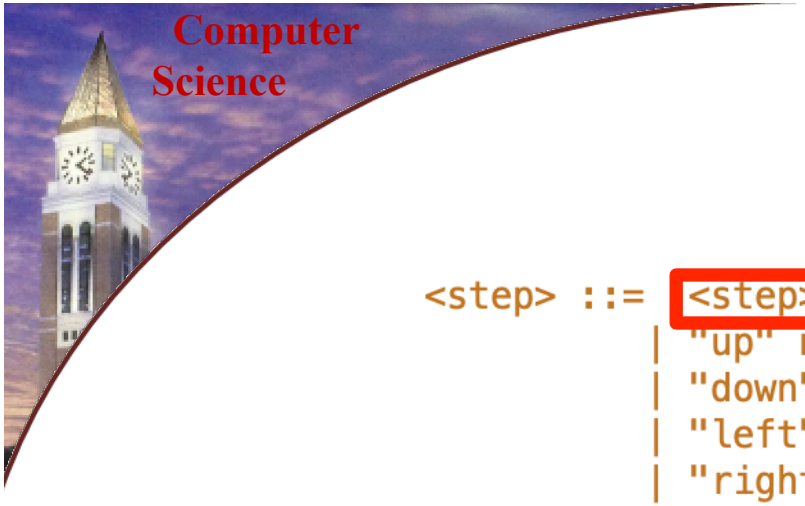
(Problem 3)

```
<step> ::=  <step>  <step>        "seq-step"
          | "up" number           "up-step"
          | "down" number         "down-step"
          | "left" number         "left-step"
          | "right" number        "right-step"
```

```
(define (single-step->n st)
  (if (single-step? st)
      (cadr st)
      (raise (invalid-args-msg "single-step->n" "single-step?" st))
      )
  )
```

4-in-1 !
(the same !)

```
(define (up-step n)
  (if (number? n)
      (list 'up n)
      (raise (invalid-args-msg "up-step" "number?" n))
      )
  )
```



HW 4

Extractors

(Data Structure-based)

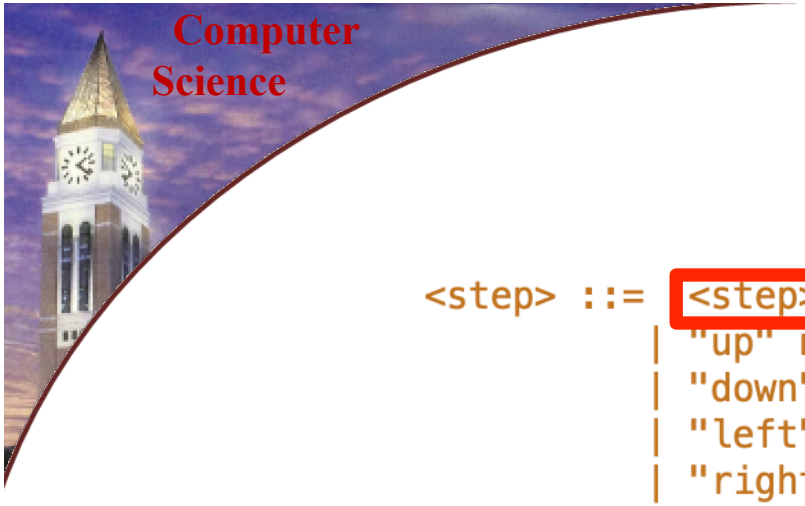
(Problem 3)

```
<step> ::= <step>  <step>        "seq-step"
         | "up" number           "up-step"
         | "down" number         "down-step"
         | "left" number         "left-step"
         | "right" number        "right-step"
```

```
(define (seq-step->st-1 st)
  (if (seq-step? st)
      (cadr st)
      (raise (invalid-args-msg "seq-step->st-1" "seq-step?" st))
      )
  )
```

```
;represented as the 3 tuple '(seq st-1 st-2)
(define (seq-step st-1 st-2)
  (if (and (step? st-1) (step? st-2))
      (list 'seq st-1 st-2)
      (raise (invalid-args-msg "seq-step" "step?" st-1))
      )
  )
```

```
(define (step? st)
  (or (single-step? st)
      (seq-step? st)
      )
  )
```



HW 4

Extractors

(Data Structure-based)

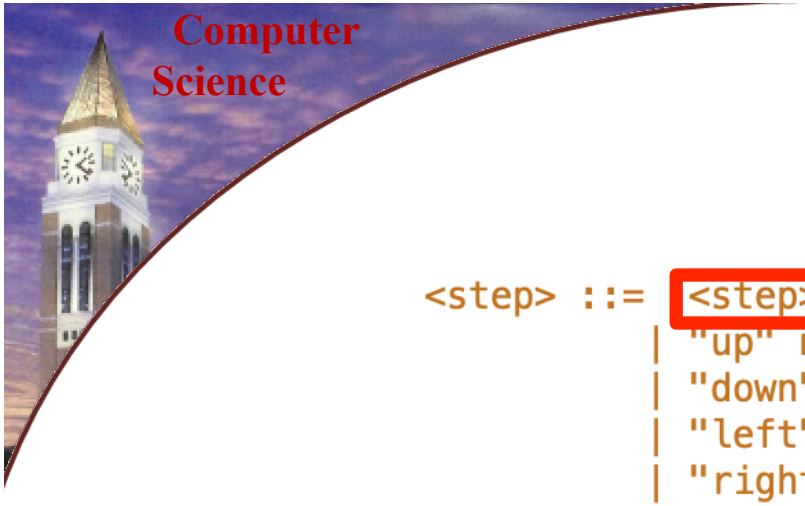
(Problem 3)

```
<step> ::= <step>  <step>        "seq-step"
         | "up" number           "up-step"
         | "down" number         "down-step"
         | "left" number         "left-step"
         | "right" number        "right-step"
```

```
(define (seq-step->st-2 st)
```

```
;represented as the 3 tuple '(seq st-1 st-2)
(define (seq-step st-1 st-2)
  (if (and (step? st-1) (step? st-2))
      (list 'seq st-1 st-2)
      (raise (invalid-args-msg "seq-step" "step?" st-1))
      )
  )
```

```
(define (step? st)
  (or (single-step? st)
      (seq-step? st)
      )
  )
```



HW 4

Extractors

(Data Structure-based)

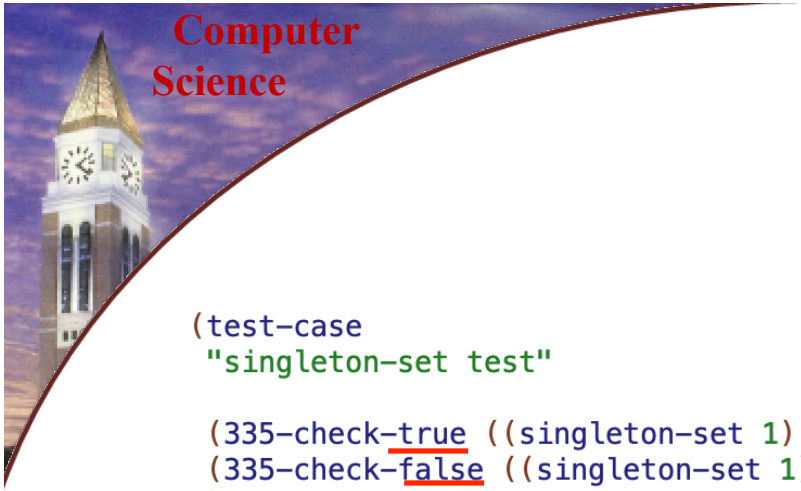
(Problem 3)

```
<step> ::= <step>  <step>        "seq-step"
         | "up" number           "up-step"
         | "down" number         "down-step"
         | "left" number         "left-step"
         | "right" number        "right-step"
```

```
(define (seq-step->st-2 st)
  (if (seq-step? st)
      (caddr st)
      (raise (invalid-args-msg "seq-step->st-2" "seq-step?" st))
      )
  )
```

```
;represented as the 3 tuple '(seq st-1 st-2)
(define (seq-step st-1 st-2)
  (if (and (step? st-1) (step? st-2))
      (list 'seq st-1 st-2)
      (raise (invalid-args-msg "seq-step" "step?" st-1))
      )
  )
```

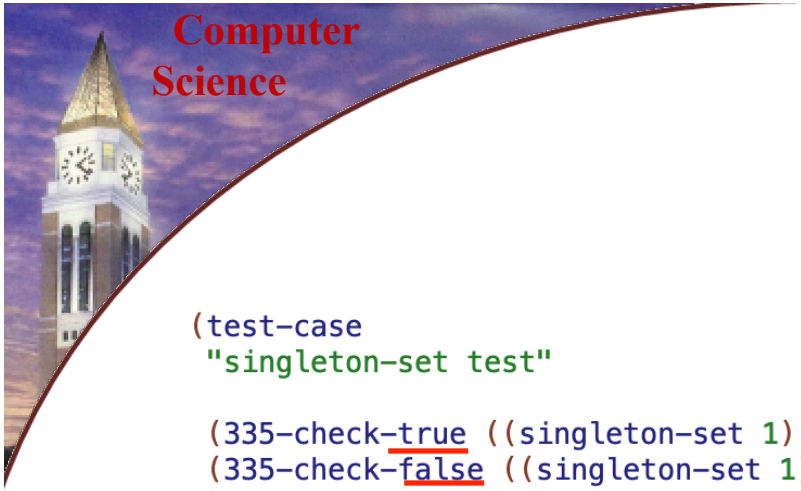
```
(define (step? st)
  (or (single-step? st)
      (seq-step? st)
      )
  )
```



HW 4

(Problem 4)

```
(test-case
 "singleton-set test"

 (335-check-true ((singleton-set 1) 1) "set containing 1, given 1")
 (335-check-false ((singleton-set 1) 2) "set containing 1, given 2")
 )
```

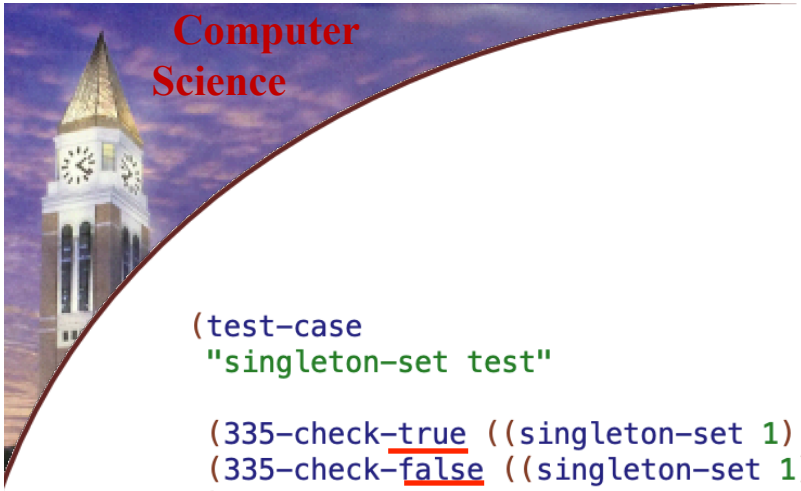


HW 4

(Problem 4)

```
(test-case
 "singleton-set test"

 (335-check-true ((singleton-set 1) 1) "set containing 1, given 1")
 (335-check-false ((singleton-set 1) 2) "set containing 1, given 2")
 )
```

```
(define (singleton-set x)
  (lambda (y)
    (equal? x y)
    )
  )
```



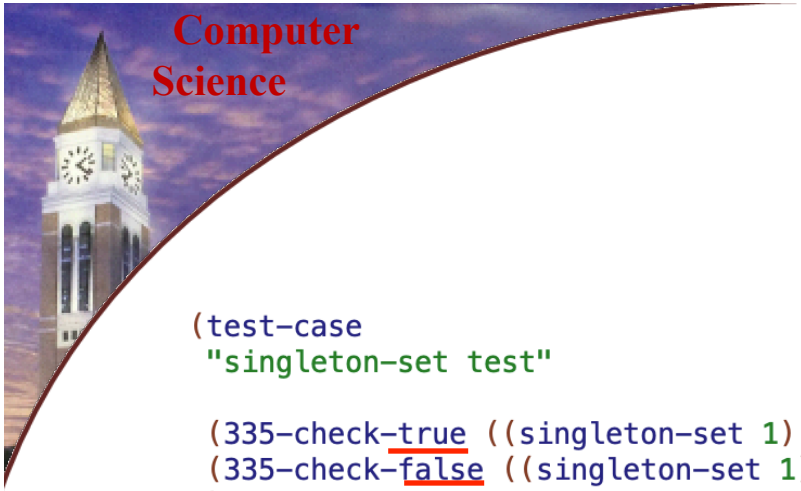
HW 4

(Problem 4)

```
(test-case
 "singleton-set test"

 (335-check-true ((singleton-set 1) 1) "set containing 1, given 1")
 (335-check-false ((singleton-set 1) 2) "set containing 1, given 2")
 )
```

```
(define (union s1 s2)
  (lambda (x)
    (or (s1 x) (s2 x))
    )
  )
```

```
(define (singleton-set x)
  (lambda (y)
    (equal? x y)
    )
  )
```



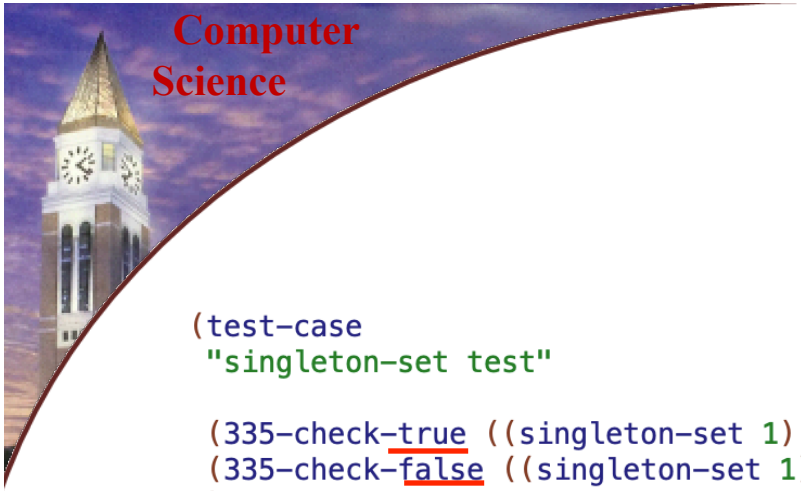
HW 4

(Problem 4)

```
(test-case
 "singleton-set test"

 (335-check-true ((singleton-set 1) 1) "set containing 1, given 1")
 (335-check-false ((singleton-set 1) 2) "set containing 1, given 2")
 )
```

```
(define (singleton-set x)
  (lambda (y)
    (equal? x y)
    )
  )
```

```
(define (union s1 s2)
  (lambda (x)
    (or (s1 x) (s2 x))
    )
  )

(define (intersection s1 s2)
  (lambda (x)
    (and (s1 x) (s2 x))
    )
  )
```



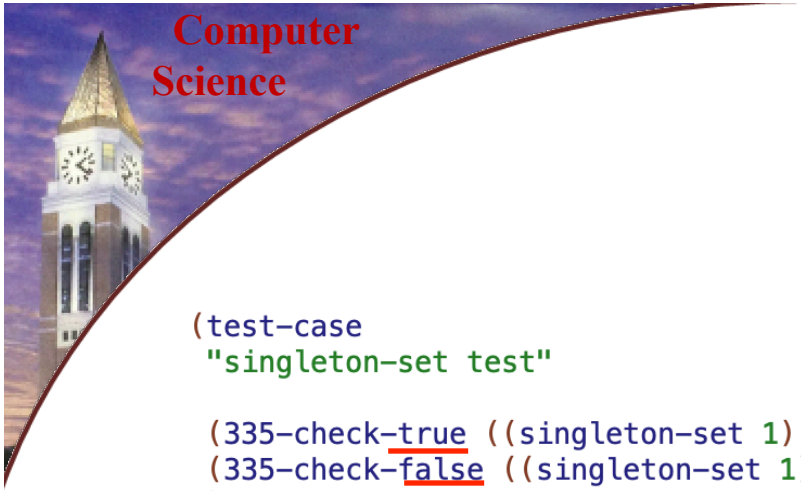
HW 4

(Problem 4)

```
(test-case
 "singleton-set test"

 (335-check-true ((singleton-set 1) 1) "set containing 1, given 1")
 (335-check-false ((singleton-set 1) 2) "set containing 1, given 2")
 )
```

```
(define (singleton-set x)
  (lambda (y)
    (equal? x y)
    )
  )
```

```
(define (union s1 s2)
  (lambda (x)
    (or (s1 x) (s2 x))
    )
  )

(define (intersection s1 s2)
  (lambda (x)
    (and (s1 x) (s2 x))
    )
  )

(define (diff s1 s2)
  (lambda (x)
    (and (s1 x) (not (s2 x)))
    )
  )
```



HW 4

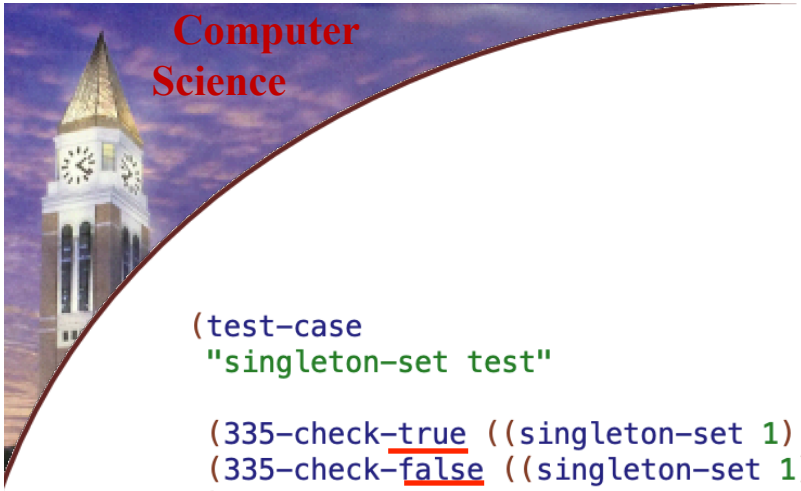
(Problem 4)

```
(test-case
 "singleton-set test"

 (335-check-true ((singleton-set 1) 1) "set containing 1, given 1")
 (335-check-false ((singleton-set 1) 2) "set containing 1, given 2")
 )
```

```
(define (singleton-set x)
  (lambda (y)
    (equal? x y)
    )
  )
```

```
(define (union s1 s2)
  (lambda (x)
    (or (s1 x) (s2 x))
    )
  )

(define (intersection s1 s2)
  (lambda (x)
    (and (s1 x) (s2 x))
    )
  )

(define (diff s1 s2)
  (lambda (x)
    (and (s1 x) (not (s2 x)))
    )
  )
```

```
(define (filter predicate s) ;s is a set
  (lambda (x)
    (and (predicate x) (s x))
    )
  )
```



HW 4

(Problem 4)

```
(test-case
 "singleton-set test"

 (335-check-true ((singleton-set 1) 1) "set containing 1, given 1")
 (335-check-false ((singleton-set 1) 2) "set containing 1, given 2")
 )
```

```
(define (singleton-set x)
  (lambda (y)
    (equal? x y)
    )
  )
```

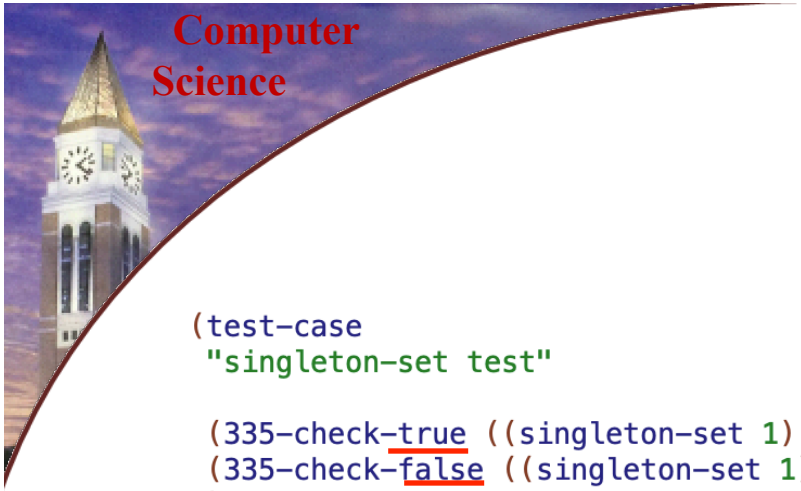
```
(define (union s1 s2)
  (lambda (x)
    (or (s1 x) (s2 x))
    )
  )
```

```
(define (intersection s1 s2)
  (lambda (x)
    (and (s1 x) (s2 x))
    )
  )
```

```
(define (diff s1 s2)
  (lambda (x)
    (and (s1 x) (not (s2 x)))
    )
  )
```

```
(define (filter predicate s) ;s is a set
  (lambda (x)
    (and (predicate x) (s x))
    )
  )
```

```
(define bound 1000)
(define (generate-range) (range (+ bound 1)))
```



HW 4

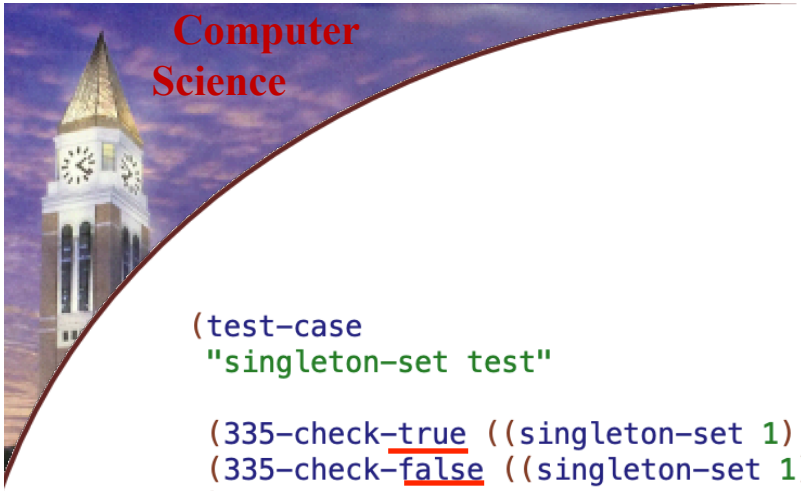
(Problem 4)

```
(test-case
 "singleton-set test"

 (335-check-true ((singleton-set 1) 1) "set containing 1, given 1")
 (335-check-false ((singleton-set 1) 2) "set containing 1, given 2")
 )
```

```
(define (singleton-set x)
  (lambda (y)
    (equal? x y)
    )
  )
```

```
(define (union s1 s2)
  (lambda (x)
    (or (s1 x) (s2 x))
    )
  )

(define (intersection s1 s2)
  (lambda (x)
    (and (s1 x) (s2 x))
    )
  )

(define (diff s1 s2)
  (lambda (x)
    (and (s1 x) (not (s2 x)))
    )
  )
```

```
(define (filter predicate s) ;s is a set
  (lambda (x)
    (and (predicate x) (s x))
    )
  )

(define bound 1000)
(define (generate-range) (range (+ bound 1)))

(define (exists? predicate s)
  (define (test? x)
    (if (s x)
        (predicate x)
        #f
        )
    )
  (ormap test? (generate-range))
  )
```



HW 4

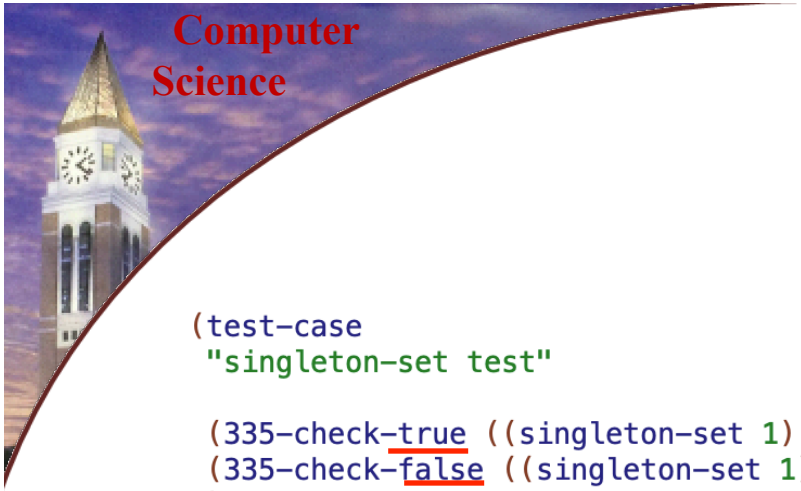
(Problem 4)

```
(test-case
 "singleton-set test"

 (335-check-true ((singleton-set 1) 1) "set containing 1, given 1")
 (335-check-false ((singleton-set 1) 2) "set containing 1, given 2")
 )
```

```
(define (singleton-set x)
  (lambda (y)
    (equal? x y)
    )
  )
```

```
(define (union s1 s2)
  (lambda (x)
    (or (s1 x) (s2 x))
    )
  )

(define (intersection s1 s2)
  (lambda (x)
    (and (s1 x) (s2 x))
    )
  )

(define (diff s1 s2)
  (lambda (x)
    (and (s1 x) (not (s2 x)))
    )
  )
```

```
(define (filter predicate s) ;s is a set
  (lambda (x)
    (and (predicate x) (s x))
    )
  )

(define bound 1000)
(define (generate-range) (range (+ bound 1)))

(define (all? predicate s)
  (define (test? x)
    (if (s x)
        (predicate x)
        #t
        )
    )
  (andmap test? (generate-range))
  )
```



HW 4

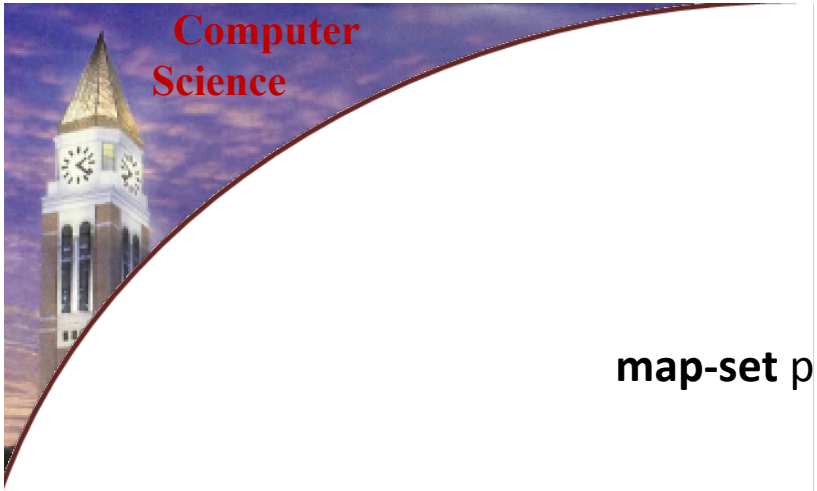
(Problem 4)

```
(test-case
 "singleton-set test"

 (335-check-true ((singleton-set 1) 1) "set containing 1, given 1")
 (335-check-false ((singleton-set 1) 2) "set containing 1, given 2")
 )
```

```
(define (singleton-set x)
  (lambda (y)
    (equal? x y)
    )
  )
```

```
(define (union s1 s2)
  (lambda (x)
    (or (s1 x) (s2 x))
    )
  )

(define (intersection s1 s2)
  (lambda (x)
    (and (s1 x) (s2 x))
    )
  )

(define (diff s1 s2)
  (lambda (x)
    (and (s1 x) (not (s2 x)))
    )
  )
```

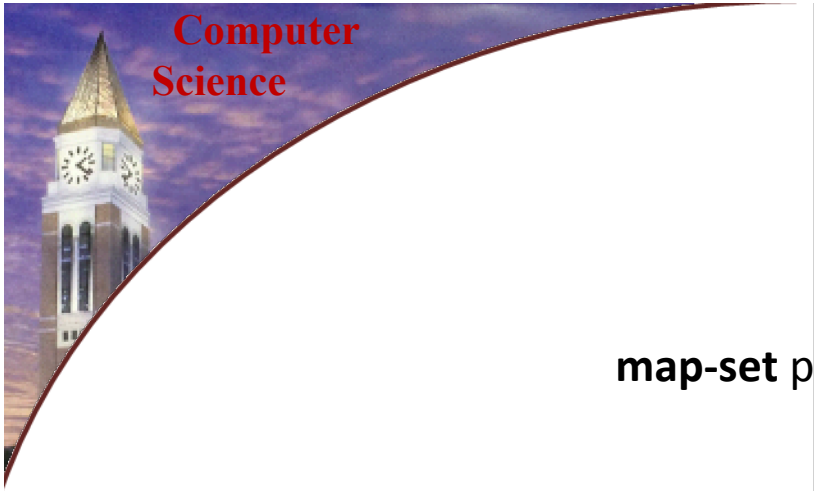
```
(define (filter predicate s) ;s is a set
  (lambda (x)
    (and (predicate x) (s x))
    )
  )

(define bound 1000)
(define (generate-range) (range (+ bound 1)))

(define (all? predicate s)
  (define (test? x)
    (if (s x)
        (predicate x)
        #t
        )
    )
  (andmap test? (generate-range))
  )
```



HW 4

(Problem 4)

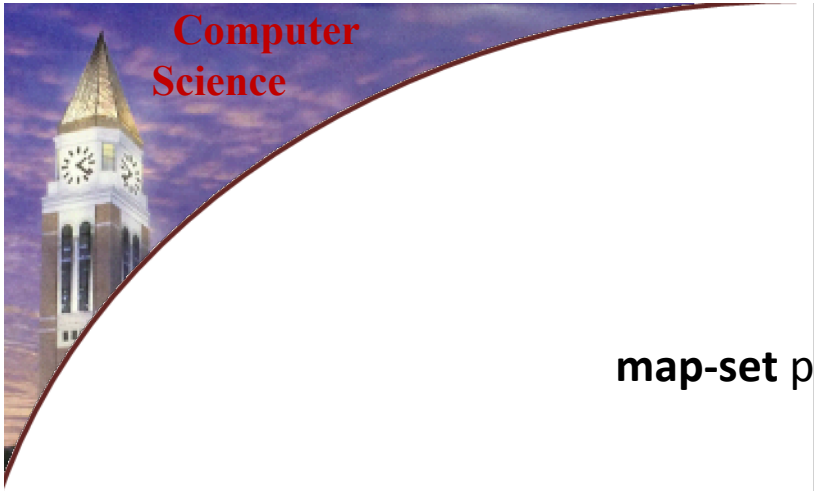
map-set problem



HW 4

(Problem 4)

map-set problem (let us go to our test file)

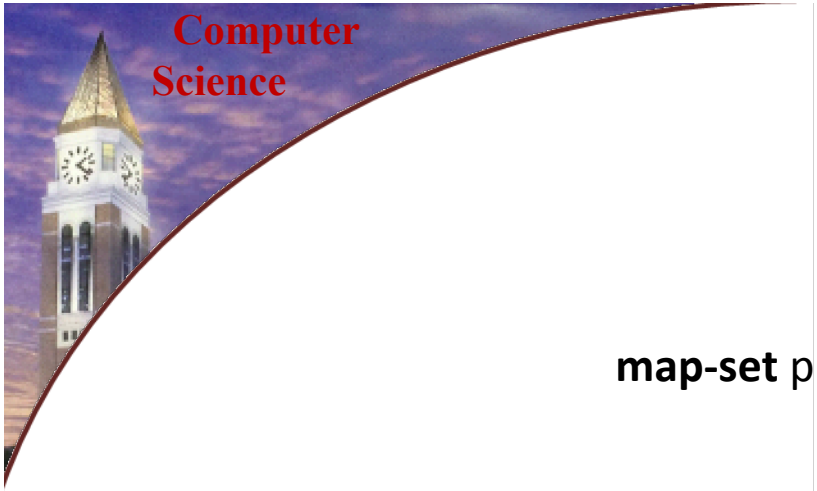


HW 4

(Problem 4)

Conceptually:
(map-set (lambda(x) (+ x 42)) { 3, 7, 13})
you get the set of
{45, 49, 55}

map-set problem (let us go to our test file)



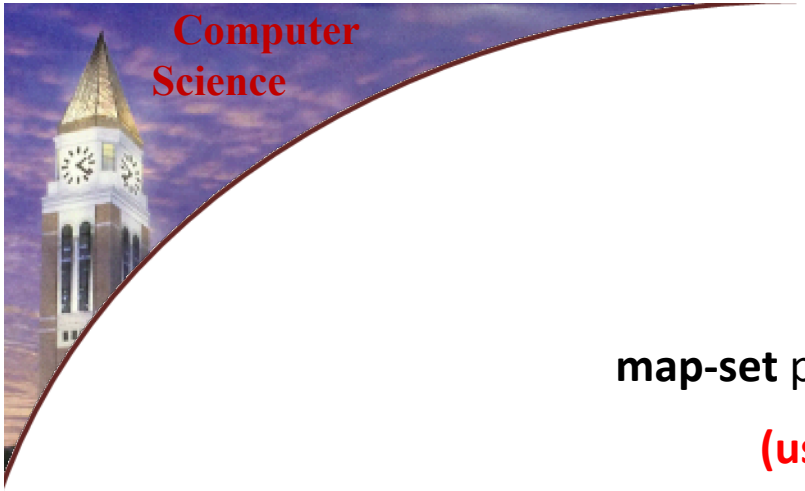
HW 4

(Problem 4)

Conceptually:
(map-set (lambda(x) (+ x 42)) { 3, 7, 13})
you get the set of
{45, 49, 55}

map-set problem (let us go to our test file)

here **{45, 49, 55}** should be represented as a function !



HW 4

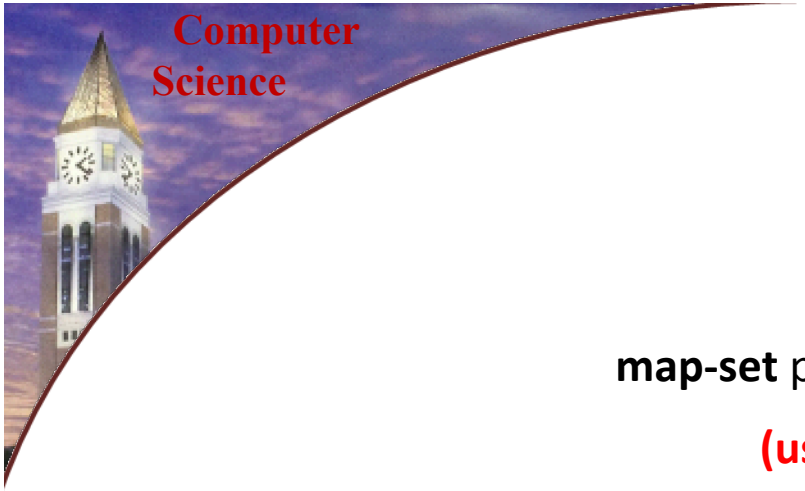
(Problem 4)

Conceptually:
(map-set (lambda(x) (+ x 42)) { 3, 7, 13})
you get the set of
{45, 49, 55}

map-set problem (let us go to our test file)

(use the `exist?` function)

here **{45, 49, 55}** should be represented as a function !



HW 4

(Problem 4)

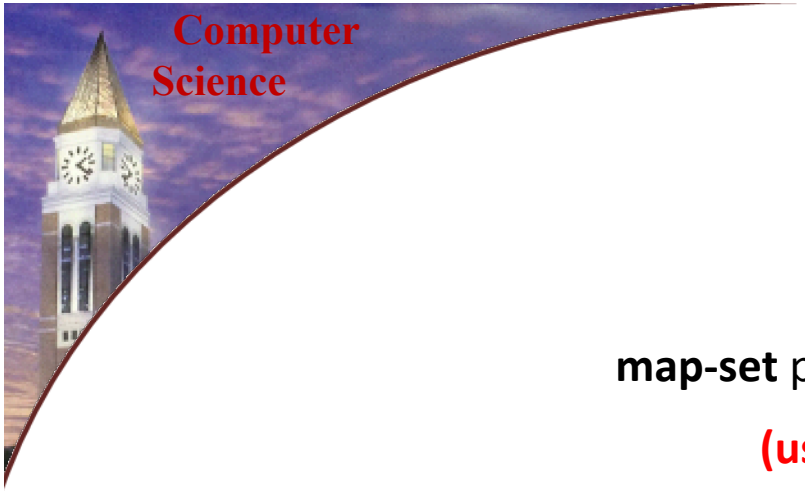
Conceptually:
(map-set (lambda(x) (+ x 42)) { 3, 7, 13})
you get the set of
{45, 49, 55}

map-set problem (let us go to our test file)

(use the `exist?` function)

here **{45, 49, 55}** should be represented as a function !

```
(define bound 1000)
(define (generate-range) (range (+ bound 1)))

(define (exists? predicate s)
  (define (test? x)
    (if (s x)
        (predicate x)
        #f
        )
    )
  (ormap test? (generate-range))
  )
```



HW 4

(Problem 4)

Conceptually:
(map-set (lambda(x) (+ x 42)) { 3, 7, 13})
you get the set of
{45, 49, 55}

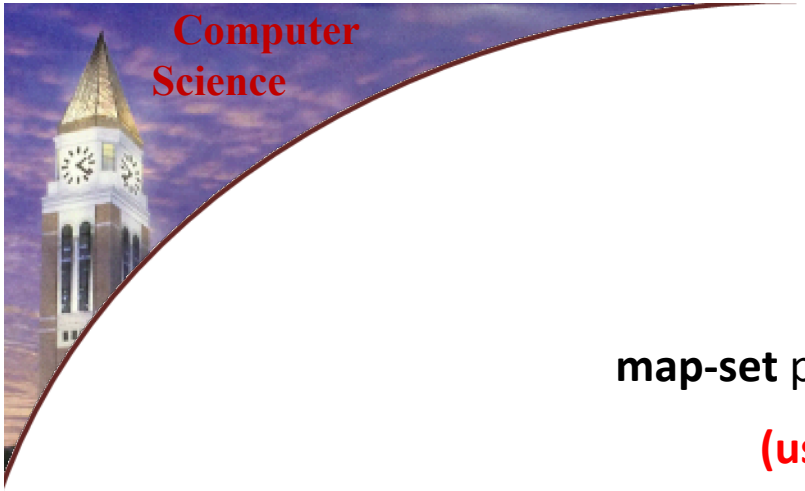
map-set problem (let us go to our test file)

(use the `exist?` function)

here **{45, 49, 55}** should be represented as a function !

to say **45** belongs to the set is the same to saying that there **exists** an number, say **i**, in **{3, 7, 13}** such that **i + 42** is 45

```
(define bound 1000)
(define (generate-range) (range (+ bound 1)))

(define (exists? predicate s)
  (define test? x)
    (if (s x)
        (predicate x)
        #f
        )
  )
  (ormap test? (generate-range))
  )
```



HW 4

(Problem 4)

Conceptually:
(map-set (lambda(x) (+ x 42)) { 3, 7, 13})
you get the set of
{45, 49, 55}

map-set problem (let us go to our test file)

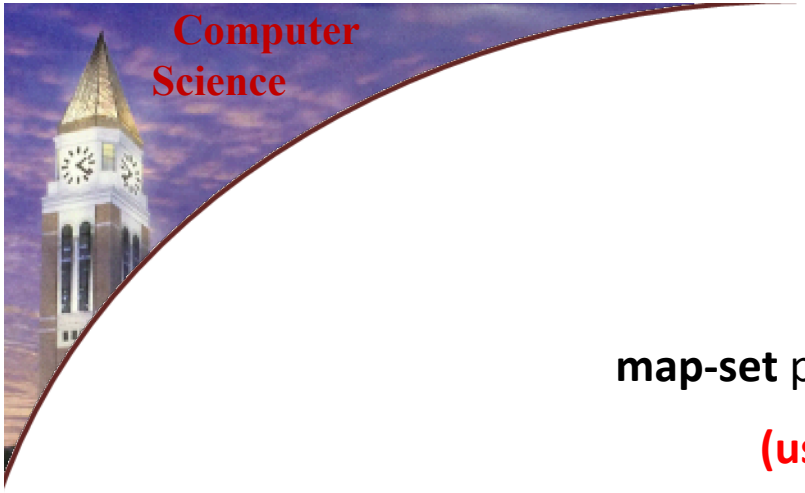
(use the `exist?` function)

here **{45, 49, 55}** should be represented as a function !

to say **45** belongs to the set is the same to saying that there **exists** an number, say **i**, in **{3, 7, 13}** such that **i + 42** is 45

to translate this idea using the exists? function seen on the right, is the following

```
(define bound 1000)
(define (generate-range) (range (+ bound 1)))

(define (exists? predicate s)
  (define test? x)
    (if (s x)
        (predicate x)
        #f
        )
    )
  (ormap test? (generate-range))
  )
```



HW 4

(Problem 4)

Conceptually:
(map-set (lambda(x) (+ x 42)) { 3, 7, 13})
you get the set of
{45, 49, 55}

map-set problem (let us go to our test file)

(use the `exist?` function)

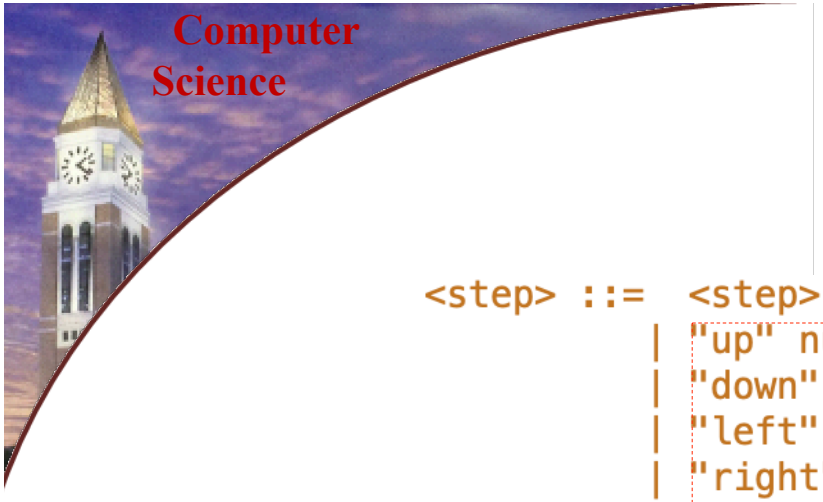
here **{45, 49, 55}** should be represented as a function !

to say **45** belongs to the set is the same to saying that there **exists** an number, say **i**, in **{3, 7, 13}** such that **i + 42** is 45

to translate this idea using the exists? function seen on the right, is the following

```
(define (map-set op s)
  (lambda (x)
    (exists? (lambda (i) (equal? (op i) x))
             s)
    )
  )
```

```
(define bound 1000)
(define (generate-range) (range (+ bound 1)))

(define (exists? predicate s)
  (define test? x)
    (if (s x)
        (predicate x)
        #f
        )
    )
  (ormap test? (generate-range))
  )
```



HW 4

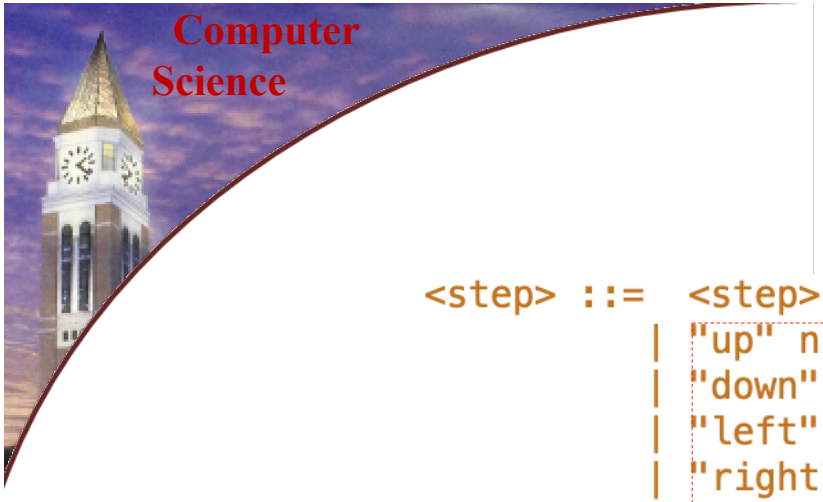
Constructors
(Procedural-based)

(Problem 5)

```
<step> ::=  <step>  <step>        "seq-step"
          | "up" number           "up-step"
          | "down" number         "down-step"
          | "left" number         "left-step"
          | "right" number        "right-step"
```

single steps

Pay special attention to the 4 kinds of single steps !



HW 4

Constructors (Procedural-based)

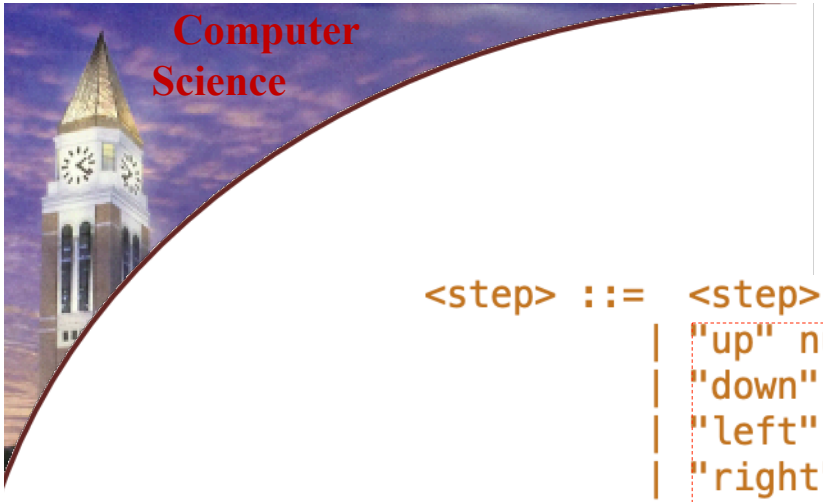
(Problem 5)

```
<step> ::=  <step>  <step>        "seq-step"
          | "up" number           "up-step"
          | "down" number         "down-step"
          | "left" number         "left-step"
          | "right" number        "right-step"
```

single steps

Pay special attention to the 4 kinds of single steps !

Rule of thumb: the **constructors** "receive all the heat" while the **predicate** and **extractors** are simply **one liner**!



HW 4

Constructors (Procedural-based)

(Problem 5)

```
<step> ::=  <step>  <step>        "seq-step"
          | "up" number           "up-step"
          | "down" number         "down-step"
          | "left" number         "left-step"
          | "right" number        "right-step"
```

single steps

Pay special attention to the 4 kinds of single steps !

Rule of thumb: the **constructors** "receive all the heat" while the **predicate** and **extractors** are simply **one liner**!

We did a general single-step constructor, i.e., **single-step-fun-representation**, for better code re-use purposes, for example -

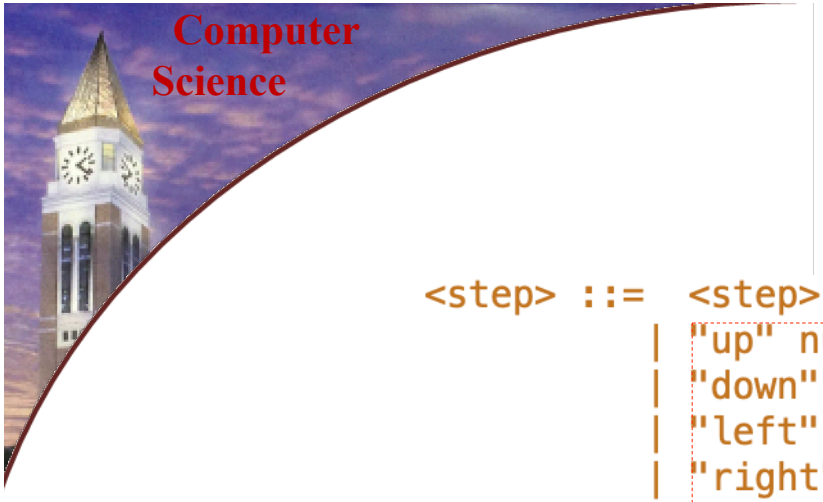
to create a **left step of step size 4**, just do

(**single-step-fun-representation** '**left-step 4**)

to create a **right step of step size 5**, just do

(**single-step-fun-representation** '**right-step 5**)

etc.,



HW 4

Constructors (Procedural-based)

(Problem 5)

```
<step> ::=  <step>  <step>        "seq-step"
          | "up" number           "up-step"
          | "down" number         "down-step"
          | "left" number         "left-step"
          | "right" number        "right-step"
```

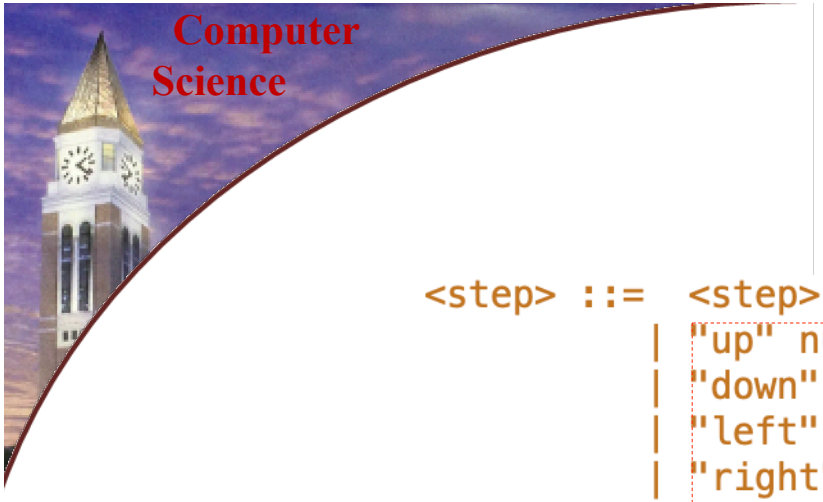
single steps

Pay special attention to the 4 kinds of single steps !

Rule of thumb: the **constructors** "receive all the heat" while the **predicate** and **extractors** are simply **one liner**!

the constructor receives all the heat!

```
(define (single-step-fun-representation type n)
  (if (number? n)
      (lambda (diff)
        (cond [(equal? diff op-data) n] ; extractor coded inside the contructor
              [(equal? diff type) #t]   ; predicate coded inside the constructor
              [else #f]
              )
        );end lambda
      (raise (get-constructor-error-msg-for-type type n))
      )
  )
```



HW 4

Constructors
(Procedural-based)

(Problem 5)

```
<step> ::=  <step>  <step>        "seq-step"
          | "up" number           "up-step"
          | "down" number         "down-step"
          | "left" number         "left-step"
          | "right" number        "right-step"
```

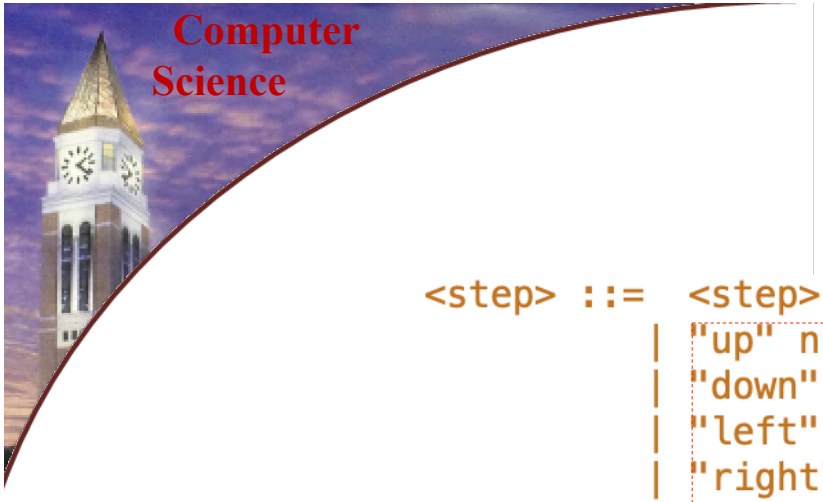
single steps

Pay special attention to the 4 kinds of single steps !

Rule of thumb: the **constructors** "receive all the heat" while the **predicate** and **extractors** are simply **one liner**!

the predicate is a one liner!
(using up step as an example!)

```
(define (up-step-proc? st)
  (and (procedure? st) (st type-up))
  )
```



HW 4

Constructors (Procedural-based)

(Problem 5)

```
<step> ::=  <step>  <step>        "seq-step"
          | "up" number           "up-step"
          | "down" number         "down-step"
          | "left" number         "left-step"
          | "right" number        "right-step"
```

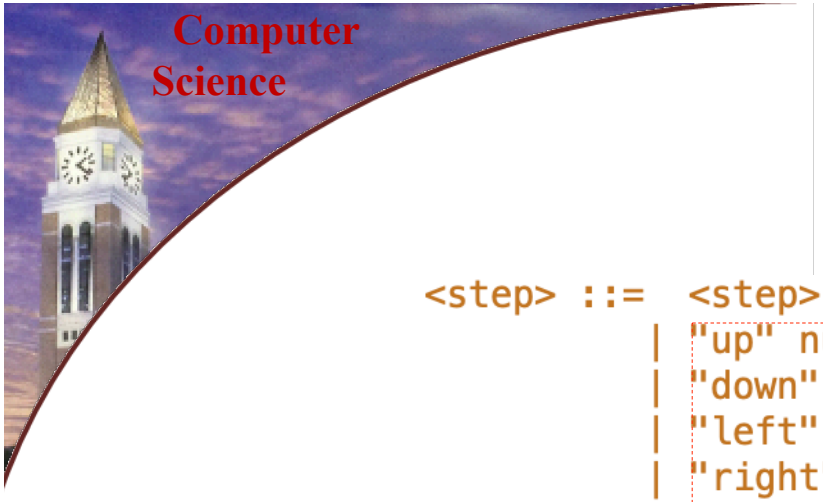
single steps

Pay special attention to the 4 kinds of single steps !

Rule of thumb: the **constructors** "receive all the heat" while the **predicate** and **extractors** are simply **one liner**!

the extractor is almost a one liner!
(using up step as an example!)

```
(define (single-step-proc->n st)
  (if (single-step-proc? st)
      (st 'single-step-data)
      (raise (invalid-args-msg "single-step-proc->n" "single-step-proc?" st))
      )
  )
```



HW 4

Constructors (Procedural-based)

(Problem 5)

```
<step> ::=  <step>  <step>        "seq-step"
          | "up" number           "up-step"
          | "down" number         "down-step"
          | "left" number         "left-step"
          | "right" number        "right-step"
```

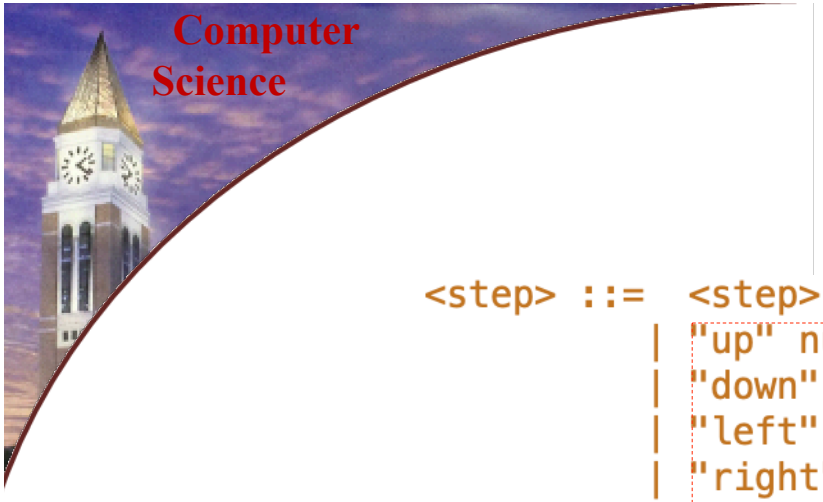
single steps

Pay special attention to the 4 kinds of single steps !

Rule of thumb: the **constructors** "receive all the heat" while the **predicate** and **extractors** are simply **one liner**!

the extractor is almost a one liner! (using up step as an example! in fact, left, right or down step all the same code for extractors, as explained in class !)

```
(define (single-step-proc->n st)
  (if (single-step-proc? st)
      (st 'single-step-data)
      (raise (invalid-args-msg "single-step-proc->n" "single-step-proc?" st))
      )
  )
```



HW 4

Constructors (Procedural-based)

(Problem 5)

```
<step> ::=  <step>  <step>        "seq-step"
          | "up" number           "up-step"
          | "down" number         "down-step"
          | "left" number         "left-step"
          | "right" number        "right-step"
```

single steps

Pay special attention to the 4 kinds of single steps !

Rule of thumb: the **constructors** "receive all the heat" while the **predicate** and **extractors** are simply **one liner**!

```
(define (single-step-proc->n st)
        (st 'single-step-data) )
```

the extractor is almost a one liner! (using up step as an example! in fact, left, right or down step all the same code for extractors, as explained in class !)