

Representing sets

Another data abstraction

Set operations:

- union-set---union of two sets
- intersection-set---intersection of two sets
- element-of-set?---test membership in a set
- adjoin-set---add an element to a set

Sets as unordered lists (without repetition)

```
(define (element-of-set? element set)
  (cond ((null? set) #f)
        ((equal? element (car set)) #t)
        (else (element-of-set? element
                                   (cdr set)))))

(define (adjoin-set element set)
  (if (element-of-set? element set)
      set
      (cons element set)))
```

Intersection

```
(define (intersection-set set1 set2)
  (cond ((or (null? set1) (null set2)) ())
        ((element-of-set? (car set1) set2)
         (cons (car set1)
               (intersection-set (cdr set1)
                                 set2)))
        (else (intersection-set (cdr set1)
                                 set2))))
```

Orders of growth for this representation

- element-of-set? --- $\theta(n)$
- adjoin-set --- $\theta(n)$
- intersection-set --- $\theta(n^2)$
- union-set --- $\theta(n^2)$

Sets as ordered lists (of numbers, ascending order)

```
(define (element-of-set? element set)
  (cond ((null? set) #f)
        ((= element (car set)) #t)
        ((< element (car set)) #f)
        (else (element-of-set? element
                                   (cdr set)))))
```

adjoin-set

```
(define (adjoin-set element set)
  (cond ((null? set) (list element))
        ((= element (car set)) set)
        (> element (car set))
          (cons (car set)
                (adjoin-set element
                            (cdr set))))
        (else (cons element set))))
```

intersection-set

```
(define (intersection-set set1 set2)
  (cond ((or (null? set1) (null? set2)) ())
        ((= (car set1) (car set2))
         (cons (car set1)
               (intersection-set
                (cdr set1)
                (cdr set2))))
        (intersection-set
         (cdr set1)
         (cdr set2))))
```

(continued)

```
((< (car set1) (car set2))
 (intersection-set (cdr set1) set2))
(else
 (intersection-set set1
                  (cdr set2))))
```

Orders of growth

- All four operations have order of growth equal to $\theta(n)$
- Operations `element-of-set?` and `adjoin-set` have been speeded up by a factor of 2

Sets as (labeled) binary trees

```
(define (make-tree entry
                  left-child
                  right-child)
  (list entry left-child right-child))
(define (entry tree) (car tree))
(define (left-branch tree) (cadr tree))
(define (right-branch tree) (caddr tree))
```

element of set

```
(define (element-of-set? element set)
  (cond ((null? set) #f)
        ((= element (entry set)) #t)
        ((< element (entry set))
         (element-of-set? element
                           (left-branch set)))
        (else
         (element-of-set?
          element
          (right-branch set)))))
```

adjoin set

```
(define (adjoin-set element set)
  (cond ((null? set)
         (make-tree element () ()))
        ((= element (entry set)) set)
        ((< element (entry set))
         (make-tree (entry set)
                     (adjoin-set
                      element
                      (left-branch set))
                     (right-branch set)))
```

(continued)

```
(else
 (make-tree
  (entry set)
  (left-branch set)
  (adjoin-set
   element
   (right-branch set))))))
```

Properties of tree representation

- If the trees are kept balanced, order of growth of `element-of-set?` and `adjoin-set` is $\theta(\log n)$
- Operations `intersection-set` and `union-set` can be implemented to have order of growth $\theta(n)$, but the implementations are complicated

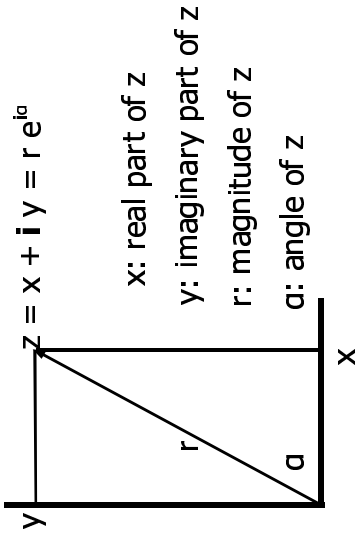
Comparison: orders of growth (θ)

<u>Operation</u>	<u>unordered</u>	<u>ordered</u>	<u>tree</u>
<code>element-of-set?</code>	n	n	$\log n$
<code>adjoin-set</code>	n	n	$\log n$
<code>intersection-set</code>	n^2	n	n
<code>union-set</code>	n^2	n	n

Multiple representations for abstract data

- Implementation of complex numbers as an example
- Illustrates how one representation can be better for one operation, but another representation might be better for another operation
- (Scheme already has complex numbers, but we'll pretend that it doesn't)

Complex numbers (math view)



Complex number arithmetic

$$\begin{aligned} z_1 + z_2 &= x_1 + iy_1 + x_2 + iy_2 \\ &= (x_1 + x_2) + i(y_1 + y_2) \end{aligned}$$

$$\begin{aligned} z_1 * z_2 &= r_1 e^{i\alpha_1} * r_2 e^{i\alpha_2} \\ &= (r_1 * r_2) e^{i(\alpha_1 + \alpha_2)} \end{aligned}$$

Two representations

- Rectangular
 - make-from-real-imag
 - real-part
 - imag-part
- Polar
 - make-from-mag-ang
 - magnitude
 - angle

Addition, subtraction

```
(define (add-complex z1 z2)
  (make-from-real-imag
    (+ (real-part z1) (real-part z2))
    (+ (imag-part z1) (imag-part z2))))
(define (sub-complex z1 z2)
  (make-from-real-imag
    (- (real-part z1) (real-part z2))
    (- (imag-part z1) (imag-part z2))))
```

Multiplication, division

```
(define (mul-complex z1 z2)
  (make-from-mag-ang
    (* (magnitude z1) (magnitude z2))
    (+ (angle z1) (angle z2))))
(define (div-complex z1 z2)
  (make-from-mag-ang
    (/ (magnitude z1) (magnitude z2))
    (- (angle z1) (angle z2))))
```

Rectangular representation

```
(define (make-from-real-mag x y) (cons x y))
(define (real-part z) (car z))
(define (imag-part z) (cdr z))
(define (make-from-mag-ang r a)
  (make-from-real-mag (* r (cos a))
    (* r (sin a))))
```

(continued)

```
(define (magnitude z)
  (sqrt (+ (square (real-part z))
    (square (imag-part z)))))
(define (angle z)
  (atan (imag-part z) (real-part z)))
```

Polar representation

```
(define (make-from-mag-ang r a) (cons r a))
(define (magnitude z) (car z))
(define (angle z) (cdr z))
(define (make-from-real-imag x y)
  (make-from-mag-ang
    (sqrt (+ (square x) (square y)))
    (atan y x)))
```

(continued)

```
(define (real-part z)
  (* (magnitude z) (cos (angle z))))
(define (imag-part z)
  (* (magnitude z) (sin (angle z))))
```

Tagged data

- Allow both representations to be used in the same program
- Tags will be "rectangular" and "polar"

Tagging as a data abstraction

```
(define (attach-tag type-tag contents)
  (cons type-tag contents))
(define (type-tag datum)
  (if (pair? datum)
      (car datum)
      (error "TYPE-TAG finds bad datum"
            datum)))
```

(continued)

```
(define (contents datum)
  (if (pair? datum)
      (cdr datum)
      (error "CONTENTS finds bad datum"
            datum)))
```

Complex numbers implementation using tags

```
(define (make-from-real-imag x y)
  (attach-tag 'rectangular (cons x y)))
(define (make-from-mag-ang r a)
  (attach-tag 'polar (cons r a)))
(define (rectangular? z)
  (eq? (type-tag z) 'rectangular))
(define (polar? z)
  (eq? (type-tag z) 'polar))
```

(continued)

```
(define (real-part z)
  (cond ((rectangular? z) (car (contents z)))
        ((polar? z)
         (* (car (contents z))
            (cos (cdr (contents z)))))
        (else
         (error
          "data type unknown to REAL-PART"
          z))))
```

imag-part

```
(define (imag-part z)
  (cond ((rectangular? z) (cdr (contents z)))
        ((polar? z)
         (* (car (contents z))
            (sin (cdr (contents z)))))
        (else
         (error
          "data-type unknown to IMAG-PART"
          z))))
```

magnitude

```
(define (magnitude z)
  (cond ((rectangular? z)
         (sqrt (+ (square (car (contents z)))
                  (square
                   (cdr (contents z))))))
        ((polar? z) (car (contents z)))
        (else
         (error
          "data-type unknown to MAGNITUDE"
          z))))
```

angle

```
(define (angle z)
  (cond ((rectangular? z)
        (atan (cdr (contents z))
              (car (contents z))))
        ((polar? z) (cdr (contents z)))
        (else
         (error "data-type unknown to ANGLE"
                z))))
```

The drama continues . . .

- What will happen if we allowed more representations of the same data type?
- How can we add more representations without rewriting all the implementation procedures all over again?
- How can we make the alternate representations more modular?
- (to be continued)