DATA TYPES IN RACKET

Abstract Data Types, from CSSE 230

- In, CSSE 230, we looked at data types.
- A data type specifies the way a data structure works.
- It is the sort of information one would find in the Java Docs.
- For example, the Java Docs for Stack specify:

> **peek**()
> Looks at the object at the top of this stack without removing it from the stack.
>
> **pop**()
> Removes the object at the top of this stack and returns that object as the value of this function.
>
> **push**(**E** item)
> Pushes an item onto the top of this stack.

Abstract Data Types, from CSSE 230

- Notice that the Java Docs do not provide code.
- This means that we may choose the manner in which to implement the ADT specified in the Java Docs.
- In principle, we could choose among: arrays, ArrayLists, Lists, Queues and others.

Abstract Data Types, from CSSE 230

- Additionally, notice that the documentation does not explicitly state runtime data.
- In this particular case, it is understood that push and pop are to run in constant time.
- This reduces the options for implementing the Stack ADT to LinkedList.

Abstract Datatypes

- An abstract data type consists of:
  - Interface (how the user sees it)
  - An indication of the runtime of operations
- A Data structure additionally:
  - The underlying representation
  - The implementation of the functionality

Number Abstract Data type

$$(\texttt{zero}) = \lceil 0 \rceil$$

$$(\texttt{is-zero?}\ \lceil n \rceil) = \begin{cases} \texttt{\#t} & n = 0 \\ \texttt{\#f} & n \neq 0 \end{cases}$$

$$(\texttt{successor}\ \lceil n \rceil) = \lceil n+1 \rceil \quad (n \geq 0)$$

$$(\texttt{predecessor}\ \lceil n+1 \rceil) = \lceil n \rceil \quad (n \geq 0)$$

A Note on Notation

- In EoPL notation, ⌈x⌉ means "the current representation of x".
- Example: non-negative integers.
  ⌈n⌉ means "the representation of the integer *n.*"

What is "four"?

Number Data type

```
;;; The number ADT, implemented through number of items in
;;; a list

(define zero
  (lambda () '()))

(define iszero? null?)

(define successor
  (lambda (n) (cons #t n)))

(define predecessor cdr)
```

$$(\texttt{zero}) = \lceil 0 \rceil$$

$$(\texttt{is-zero?}\ \lceil n \rceil) = \begin{cases} \texttt{\#t} & n = 0 \\ \texttt{\#f} & n \neq 0 \end{cases}$$

$$(\texttt{successor}\ \lceil n \rceil) = \lceil n+1 \rceil \quad (n \geq 0)$$

$$(\texttt{predecessor}\ \lceil n+1 \rceil) = \lceil n \rceil \quad (n \geq 0)$$

Number Data type

```
;;; The number ADT implemented through Scheme numbers

(define zero
  (lambda () 0))

(define is-zero? zero?)

(define successor
  (lambda (n) (+ n 1)))

(define predecessor
  (lambda (n) (- n 1)))
```

$$(\texttt{zero}) = \lceil 0 \rceil$$

$$(\texttt{is-zero?}\ \lceil n \rceil) = \begin{cases} \texttt{\#t} & n = 0 \\ \texttt{\#f} & n \neq 0 \end{cases}$$

$$(\texttt{successor}\ \lceil n \rceil) = \lceil n+1 \rceil \quad (n \geq 0)$$

$$(\texttt{predecessor}\ \lceil n+1 \rceil) = \lceil n \rceil \quad (n \geq 0)$$

Number Data type

```
(define plus
  (lambda (x y)
    (if (iszero? x)
        y
        (successor (plus (predecessor x) y)))))
```

$$(\texttt{zero}) = \lceil 0 \rceil$$

$$(\texttt{is-zero?}\ \lceil n \rceil) = \begin{cases} \texttt{\#t} & n = 0 \\ \texttt{\#f} & n \neq 0 \end{cases}$$

$$(\texttt{successor}\ \lceil n \rceil) = \lceil n+1 \rceil \quad (n \geq 0)$$

$$(\texttt{predecessor}\ \lceil n+1 \rceil) = \lceil n \rceil \quad (n \geq 0)$$

define-datatype

- **`define-datatype`** is a way of adding record types to Racket
- An extension develop by the authors of EoPL
- Implemented as a *syntactic extension* (using `define-syntax`).
-

Use define-datatype in your code

- Put **`chez-init.rkt`** in the same folder as your code.
- Add the following to the top of your file:

```
(require "chez-init.rkt")
```

Data Types

- Constructor
- Predicate to tell whether an object is of the desired type (instanceof in Java)
- Fields
- Types of the fields
- Sub-types and how to tell

Binary Trees

```
<bintree> ::= <number>
          ::= (<number> <bintree> <bintree>)
```

Binary Tree Data Type

```
(define-datatype bintree bintree?
  [leaf-node
    (datum number?)]
  [interior-node
    (datum number?)
    (left-tree bintree?)
    (right-tree bintree?)])

<bintree> ::= <number>
          ::= (<number> <bintree> <bintree>)
```

Binary Tree Data Type

```
;;; Let's create some sample data:
(define t1 (leaf-node 3))
(define t2 (leaf-node 4))
(define t3 (interior-node 5 t1 t2))
(define t4 (interior-node 6 t3 (leaf-node 7)))

;;; Let's calculate the sum of the numbers in the tree

(define sum
  (lambda (t)
    (cases bintree t
      [leaf-node (x) x]
      [interior-node (datum left right)
                     (+ datum (sum left) (sum right))])))
```

Exercise

- Develop a data-type for Number, i.e.

$$(\texttt{zero}) = \lceil 0 \rceil$$

$$(\texttt{is-zero?}\ \lceil n \rceil) = \begin{cases} \texttt{\#t} & n = 0 \\ \texttt{\#f} & n \neq 0 \end{cases}$$

$$(\texttt{successor}\ \lceil n \rceil) = \lceil n+1 \rceil \quad (n \geq 0)$$

$$(\texttt{predecessor}\ \lceil n+1 \rceil) = \lceil n \rceil \quad (n \geq 0)$$