# Data Abstraction

CS510

Data Abstraction in Scheme

An Extended Example: Environments

# Abstract Data Types

A data type that supports data abstraction

- Data abstraction: division of a data type into
  1. an interface
  2. an implementation
- Hidden Representation Principle: all access to the datatype implementation can only be performed through the interface
- This separation induces the capacity to modify the implementation without changing the code where the data type is used

# Inductive Sets as ADTs[1]

- Inductive sets may be seen as ADTs
- We may assume that they come equipped with the basic interface consisting of operations for:
  - constructing an element
  - testing whether an element belongs to the set
  - projecting the components of a composite element
- We'll next present an abstraction for representing these ADTs in Scheme
- This improves over last class' representation of inductive types in terms of pairs, lists, etc.
  - That representation did not respect the Hidden Representation Principle

[1] For this class, add `(require eopl/eopl)` just after "#lang racket"

# Primary Colors using `define-datatype`

$$\frac{}{Red \in PCol} \qquad \frac{}{Green \in PCol} \qquad \frac{}{Blue \in PCol}$$

- To represent it in Scheme we need:
  - constructors to build each color
  - predicate to test if a value is a representation of a color
  - observer to determine which color a value denotes (next slide)

```
(define-datatype primColor primColor?
           (red)
           (green)
           (blue))
```

- Data types defined like this are also called variant records

# Primary Colors using `define-datatype`

```
(define-datatype primColor primColor?
          (red)
          (green)
          (blue))
```

The declaration creates a datatype with interface:

- `red`:
  - 0 argument procedure for constructing color `red`.
- `green`:
  - 0 argument procedure for constructing color `green`.
- `blue`:
  - 0 argument procedure for constructing color `blue`.
- `primColor?`:
  - 1 argument predicate that only returns true if the value given is `red`, `green` or `blue`.
- Each constructor `red`, `green` and `blue` represents a variant of a `primColor`

# define-datatype and cases

```
(define-datatype primColor primColor?
          (red)
          (green)
          (blue))
```

- Note difference between `red` and `(red)`
- Constructors are procedures

```
> red
#<procedure:red>
> (red)
#(struct:red)
```

# Defining Functions Over Inductive Sets

```
(define nextPrimColor
       (lambda (pcolor)
         (cases primColor pcolor
            (red () (green))
            (green () (blue))
            (blue () (red))
            (else (error "bad primary color")))))
```

- Note the use of the `cases` construct

# Another Example

$$\frac{s \in S}{leaf(s) \in BTree(S)} \qquad \frac{l \in BTree(S) \quad r \in BTree(S)}{node(l, r) \in BTree(S)}$$

```
(define-datatype binTree binTree?
   (leaf-node
      (datum number?))
   (interior-node
      (left binTree?)
      (right binTree?)))
```

A binTree is either:

- a `leaf-node` or
- an `interior-node`

# Binary trees using `define-datatype`

```
(define-datatype binTree binTree?
   (leaf-node
      (datum number?))
   (interior-node
      (left binTree?)
      (right binTree?)))
```

The declaration creates a datatype with interface:

- a leaf-node:
  - 1 argument procedure for constructing a `leaf-node`.
  - this procedure tests if the argument is a number.
- an `interior-node`:
  - 2 argument procedure for constructing an `interior-node`.
- `binTree?`:
  - 1 argument predicate that only returns true if the value given is a `leaf-node` or an `interior-node`.

# Computer the sum of the nodes in a tree

```
(define-datatype binTree binTree?
     (leaf-node
        (datum number?))
     (interior-node
        (left binTree?)
        (right binTree?)))

;; binTree -> num
(define leaf-sum
  (lambda (tree)
    (cases binTree tree
       (leaf-node (datum) datum)
       (interior-node (left right)
          (+ (leaf-sum left) (leaf-sum right))))))
```

# Recursion Template

```
;; binTree -> ???
(define binTree-proc
  (lambda (t)
    (cases t tree
        (leaf-node (datum) ...)
        (interior-node (left right)
            ...(binTree-proc left)...
            ...(binTree-proc right)...))))
```

# Exercise

- Using this template write the procedure `binTree-flip` that produces a mirror image of a given binTree `t`

```
>(binTree-flip (interior-node
                             (leaf-node 3)
                             (interior-node
                              (leaf-node 7)
                              (leaf-node 10))))

(interior-node
   (interior-node (leaf-node 10) (leaf-node 7))
   (leaf-node 3))

>(binTree-flip (leaf-node 3))
(leaf-node 3)
```

Data Abstraction in Scheme

An Extended Example: Environments

# Another Data Abstraction Example: Environments

- An environment is a finite function $f$ that associates each symbol in a finite set with a value
- An example:

$$\begin{aligned} f(d) &= 6 \\ f(x) &= 7 \\ f(y) &= 8 \end{aligned}$$

# Environments as an Inductive Set

$$\frac{}{\textit{empty-env} \in \textit{Env}}$$

$$\frac{\textit{symbols} \in \textit{List}(\textit{Symbol}) \quad \textit{values} \in \textit{List}(\textit{Values}) \quad \textit{env} \in \textit{Env}}{\textit{extend-env symbols values env} \in \textit{Env}}$$

Environment as a function:

$$f(d) = 6, f(x) = 7, f(y) = 8$$

Environment as element of *Env*:

$$\textit{extend-env}\ '(d\ x)\ '(6\ 7)\ (\textit{extend-env}\ '(y)\ '(8)\ \textit{empty-env}))$$

# Defining Environments

- In order to define the environments ADT we have to provide:
  1. an interface
  2. an implementation
- We shall see two implementations
  1. In terms of the `define-datatype` construction (variant records)
  2. In terms of functions
- Before proceeding, a quick word on the `let` construct

# Local Variable Declaration

- `let` allows us to provide local definitions

```
> (let ((x 2) (y 3))
       (+ y x))
5
```

- `x` is declared to have value `2` and `y` value 3
- These variables are local in the sense that they may only be referenced inside the body of the let expression
  - The body of the let is the scope of the variables
- The names of the variables is irrelevant; Eg. an equivalent expression is

```
> (let ((u 2) (v 3))
       (+ u v))
5
```

# Implementation based on Variant Records

```
(define-datatype environment environment?
  (empty-env-record)
  (extended-env-record
    (syms (list-of symbol?))
    (vals (list-of scheme-value?))
    (env environment?)))

(define scheme-value? (lambda (v) #t))
```

- Two constructors
  - `empty-env-record`: constructs an empty environment
  - `extended-env-record`: extends a previous environment with a new association pair

# Operations that use the Variant Record representation

Constructing an empty environment

```
;; {} -> environment
(define empty-env
  (lambda ()
    (empty-env-record)))
```

- `environment?` is the name of a new type, so we can use it in a type expression (we drop the question mark though)

# Operations that use the Variant Record representation

Extending an environment with new associations

```
;; { [sym],[scheme-value?],environment } ->
    environment
(define extend-env
  (lambda (syms vals env)
    (extended-env-record syms vals env)))
```

Note: `sym` is our type expression for `symbol?`, just like we use `num` for `number?`

# Operations that use the Variant Record representation

Using environments to lookup the value of a symbol

```
;; { environment , sym } -> scheme-value
(define apply-env
  (lambda (env sym)
    (cases environment env
      (empty-env-record ()
        (error 'apply-env "No binding for ~s" sym))
      (extended-env-record (syms vals env)
        (let ((pos (list-find-position sym syms)))
          (if (>= pos 0)
              (list-ref vals pos)
              (apply-env env sym))))
  )))
```

- The auxiliary operation `list-find-position` is defined in the next slide

# Auxiliary Operations

```
;; {a, [a]} -> bool
(define list-find-position
  (lambda (sym los)
    (list-index
      (lambda (sym1) (eqv? sym1 sym)) los)))

;; {(a -> bool), [a]} -> num
(define list-index
  (lambda (pred ls)
    (cond
      ((null? ls) -1)
      ((pred (car ls)) 0)
      (else (let ((list-index-r
                   (list-index pred (cdr ls))))
                (if (>= list-index-r 0)
                    (+ list-index-r 1)
                    -1
                )
       )))))
```

# Functional Representation

- We now consider an alternative representation of environments
- It is based on functions
- Encoding datatypes in terms of functions is an interesting exercise
- We shall have two kinds of functions:
  - those that represent operations on environments
  - those that represent environments themselves
    - empty environment
    - extending environment

# Functional Representation

```
;; {} -> (sym -> scheme-value)
define empty-env
  (lambda ()
    (lambda (sym)
      (error 'apply-env "No binding for ~s" sym))))
```

- `empty-env` constructs an empty environment
- Its not an empty environment itself

# Functional Representation

```
;; { [sym],[scheme-value],(sym->scheme-value) } ->
    (sym -> scheme-value)
(define extend-env
  (lambda (syms vals env)
    (lambda (sym)
      (let ((pos (list-find-position sym syms)))
        (if (>= pos 0)
          (list-ref vals pos)
          (apply-env env sym))))))
```

- `extend-env` constructs a new enviornment
- Eg. `(extend-env '(x y) '(1 2) (empty-env))`

# Functional Representation

```scheme
;;  { sym -> scheme-value, sym } -> scheme-value
(define apply-env
  (lambda (env sym)
      (env sym)))
```

For example,

```scheme
> ((extend-env '(x y) '(1 2) (empty-env)) 'x)
1
> ((extend-env '(x y) '(1 2) (empty-env)) 'z)
apply-env: No binding for z
```

# Summary

- Variant record for encoding inductive sets in Scheme
  - Preferred over previous encodings seen in class since Hidden Representation Principle is upheld
- We saw various examples of variant records including trees and environments
- For environments we also presented an encoding in terms of functions