

Lecture 8 – Review

Data Abstraction

Interfaces & Representation



T. METIN SEZGIN





Lecture Nuggets



- A handful of key concepts in programming languages
 - Value
 - Abstraction
 - Interface
 - Representation
 - Implementation
- May have many implementations for an interface
- Representation of a value may take different forms
- The environment allows us to store variable value pairs

Interface vs. Implementation



- Teasing out the “interface” and the “implementation”
 - I don’t care how you manage it, but I’ll be happy as long as...
 - The particular way in which I accomplish my goal is by...

Representation vs. Value



Natural Numbers

$\lceil v \rceil$ “the representation of data v .”

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

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

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

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

Procedures manipulating the new data type



- How do we implement **plus**

```
(define plus
  (lambda (x y)
    (if (is-zero? x)
        y
        (successor (plus (predecessor x) y)))))
```

- Accomplish all you would like to accomplish through the **interface**
- And... $(\text{plus } [x] \ [y]) = [x + y]$

Back to Natural Numbers



- Constructors
- Observers

$$(\text{zero}) = [0]$$

$$(\text{is-zero? } [n]) = \begin{cases} \text{\#t} & n = 0 \\ \text{\#f} & n \neq 0 \end{cases}$$

$$(\text{successor } [n]) = [n + 1] \quad (n \geq 0)$$

$$(\text{predecessor } [n + 1]) = [n] \quad (n \geq 0)$$

Back to Natural Numbers



- Constructors
- Observers

constructor $(\text{zero}) = \lfloor 0 \rfloor$ *representation of zero in my implementation*

observer $(\text{is-zero? } \lfloor n \rfloor) = \begin{cases} \#t & n = 0 \\ \#f & n \neq 0 \end{cases}$ *manipulate the data and gives information*

constructor $(\text{successor } \lfloor n \rfloor) = \lfloor n+1 \rfloor \quad (n \geq 0)$

constructor $(\text{predecessor } \lfloor n+1 \rfloor) = \lfloor n \rfloor \quad (n \geq 0)$

implementing plus:

```
(define plus
  (lambda (x y)
    (if (is-zero? x)
        y
        (successor (plus (predecessor x) y)))))
```

$x-1+y$

$x-1$

$x-1+y+1 \rightarrow "x+y"$

$(\text{plus } [x] \ [y]) = [x+y],$

Ceren Tarim

Implementation of Natural Numbers



- Unary representation
 - Use #t's to represent numbers

$$\begin{aligned} [0] &= () \\ [n+1] &= (\text{\#t} . [n]) \end{aligned}$$

- Scheme implementation

```
(define zero (lambda () ' ()))  
(define is-zero? (lambda (n) (null? n)))  
(define successor (lambda (n) (cons #t n)))  
(define predecessor (lambda (n) (cdr n)))
```

Another implementation



- Scheme number representation
 - Use scheme numbers to represent numbers

- Scheme implementation

```
(define zero (lambda () 0))  
(define is-zero? (lambda (n) (zero? n)))  
(define successor (lambda (n) (+ n 1)))  
(define predecessor (lambda (n) (- n 1)))
```

o Data Structure Representation

o Procedural Representation

o Environment: Function that maps variables to values



o Interface

$(\text{empty-env}) = \lceil \emptyset \rceil$

$(\text{apply-env } \lceil f \rceil \text{ var}) = f(\text{var})$

$(\text{extend-env var } v \lceil f \rceil) = \lceil g \rceil$, where $g(\text{var}_1) = \begin{cases} v & \text{if } \text{var}_1 = \text{var} \\ f(\text{var}_1) & \text{otherwise} \end{cases}$

The environment interface



Environment interface:

`(empty-env)` = $[\emptyset]$
`(apply-env $[f]$ var)` = $f(var)$ *→ get the value*
`(extend-env var v $[f]$)` = $[g]$ *→ rep. of another environment*
where $g(var_1) = \begin{cases} v & \text{if } var_1 = var \\ f(var_1) & \text{otherwise} \end{cases}$

Implementation:

Grammar \rightarrow $Env = (\text{empty-env}) \mid (\text{extend-env } Var \text{ SchemeVal } Env)$
 $Var = Sym$

empty-env : $() \rightarrow Env$

```
(define empty-env  
  (lambda () (list 'empty-env)))
```

extend-env : $Var \times SchemeVal \times Env \rightarrow Env$

```
(define extend-env  
  (lambda (var val env)  
    (list 'extend-env var val env)))
```

apply-env : $Env \times Var \rightarrow SchemeVal$

```
(define apply-env  
  (lambda (env search-var)  
    (cond  
      [ ((eqv? (car env) 'empty-env)  
        (report-no-binding-found search-var))  
      [ ((eqv? (car env) 'extend-env)  
        (let ((saved-var (cadr env))  
              (saved-val (caddr env))  
              (saved-env (cadddr env)))  
          (if (eqv? search-var saved-var)  
              saved-val  
              (apply-env saved-env search-var))))  
      (else  
        (report-invalid-env env))))))
```

Lecture 9

Representation Strategies for Data Types



T. METIN SEZGIN

Lecture Nuggets



- We can represent data types using data structures
- We can represent data types using procedures
- Use the environment as an example
- We can automate mundane data type definitions

Nugget



The environment allows us to store
variable value pairs

Representation strategies



- Two strategies
 - Data Structure Representation
 - Procedural Representation
- Test case
 - Environment
 - ✦ Function that maps variables to values
 - List, function, hashtable...
 - Start with the interface
 - Introduce implementation

The Environment Interface



○ Environment

- ✦ Function that maps variables to values

$$\{(var_1, val_1), \dots, (var_n, val_n)\}$$

○ The interface

$$\begin{aligned} (\text{empty-env}) &= [\emptyset] \\ (\text{apply-env } [f] \text{ var}) &= f(\text{var}) \\ (\text{extend-env } \text{var } v [f]) &= [g], \\ &\text{where } g(\text{var}_1) = \begin{cases} v & \text{if } \text{var}_1 = \text{var} \\ f(\text{var}_1) & \text{otherwise} \end{cases} \end{aligned}$$

Data Structure Representation



- The interface
 - ✦ Constructors
 - ✦ Observers

```
(empty-env)           = []  
(apply-env [f] var)   = f(var)  
(extend-env var v [f]) = [g],  
                        where  $g(var_1) = \begin{cases} v & \text{if } var_1 = var \\ f(var_1) & \text{otherwise} \end{cases}$ 
```

- For example

```
(define e  
  (extend-env 'd 6  
    (extend-env 'y 8  
      (extend-env 'x 7  
        (extend-env 'y 14  
          (empty-env) ) ) ) ) )  
e(d) = 6, e(x) = 7, e(y) = 8
```

- The grammar

```
Env-exp ::= (empty-env)  
         ::= (extend-env Identifier Scheme-value Env-exp)
```

Implementation



Env = (empty-env) | (extend-env *Var* *SchemeVal* *Env*)
Var = *Sym*

Implementation



```
Env = (empty-env) | (extend-env Var SchemeVal Env)  
Var = Sym
```

empty-env : () \rightarrow *Env*

```
(define empty-env  
  (lambda () (list 'empty-env)))
```

extend-env : *Var* \times *SchemeVal* \times *Env* \rightarrow *Env*

```
(define extend-env  
  (lambda (var val env)  
    (list 'extend-env var val env)))
```

apply-env : *Env* \times *Var* \rightarrow *SchemeVal*

```
(define apply-env  
  (lambda (env search-var)  
    (cond  
      ((eqv? (car env) 'empty-env)  
       (report-no-binding-found search-var))  
      ((eqv? (car env) 'extend-env)  
       (let ((saved-var (cadr env))  
             (saved-val (caddr env))  
             (saved-env (cadddr env)))  
         (if (eqv? search-var saved-var)  
             saved-val  
             (apply-env saved-env search-var))))  
      (else  
       (report-invalid-env env))))))
```

Implementation



$Env = (\text{empty-env}) \mid (\text{extend-env } Var \text{ SchemeVal } Env)$
 $Var = Sym$

empty-env : $() \rightarrow Env$

```
(define empty-env  
  (lambda () (list 'empty-env)))
```

extend-env : $Var \times SchemeVal \times Env \rightarrow Env$

```
(define extend-env  
  (lambda (var val env)  
    (list 'extend-env var val env)))
```

apply-env : $Env \times Var \rightarrow SchemeVal$

```
(define apply-env  
  (lambda (env search-var)  
    (cond  
      ((eqv? (car env) 'empty-env)  
       (report-no-binding-found search-var))  
      ((eqv? (car env) 'extend-env)  
       (let ((saved-var (cadr env))  
             (saved-val (caddr env))  
             (saved-env (cadddr env)))  
         (if (eqv? search-var saved-var)  
             saved-val  
             (apply-env saved-env search-var))))  
      (else  
       (report-invalid-env env))))))
```

```
(define e  
  (extend-env 'd 6  
    (extend-env 'y 8  
      (extend-env 'x 7  
        (extend-env 'y 14  
          (empty-env))))))
```

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

$Env\text{-}exp ::= (\text{empty-env})$

$::= (\text{extend-env } Identifier \text{ Scheme-value } Env\text{-}exp)$

Nugget



We can represent data types using
procedures

Procedural Representation



$Env = Var \rightarrow SchemeVal$

empty-env : $() \rightarrow Env$

```
(define empty-env
  (lambda ()
    (lambda (search-var)
      (report-no-binding-found search-var))))
```

extend-env : $Var \times SchemeVal \times Env \rightarrow Env$

```
(define extend-env
  (lambda (saved-var saved-val saved-env)
    (lambda (search-var)
      (if (eqv? search-var saved-var)
          saved-val
          (apply-env saved-env search-var))))))
```

apply-env : $Env \times Var \rightarrow SchemeVal$

```
(define apply-env
  (lambda (env search-var)
    (env search-var)))
```

Nugget



We can automate mundane data
type definitions

(Racket is a powerful language that will simplify life for us)

Implementation



$Env = (\text{empty-env}) \mid (\text{extend-env } Var \text{ SchemeVal } Env)$
 $Var = Sym$

empty-env : $() \rightarrow Env$

```
(define empty-env  
  (lambda () (list 'empty-env)))
```

extend-env : $Var \times SchemeVal \times Env \rightarrow Env$

```
(define extend-env  
  (lambda (var val env)  
    (list 'extend-env var val env)))
```

apply-env : $Env \times Var \rightarrow SchemeVal$

```
(define apply-env  
  (lambda (env search-var)  
    (cond  
      ((eqv? (car env) 'empty-env)  
       (report-no-binding-found search-var))  
      ((eqv? (car env) 'extend-env)  
       (let ((saved-var (cadr env))  
             (saved-val (caddr env))  
             (saved-env (cadddr env)))  
         (if (eqv? search-var saved-var)  
             saved-val  
             (apply-env saved-env search-var))))  
      (else  
       (report-invalid-env env))))))
```

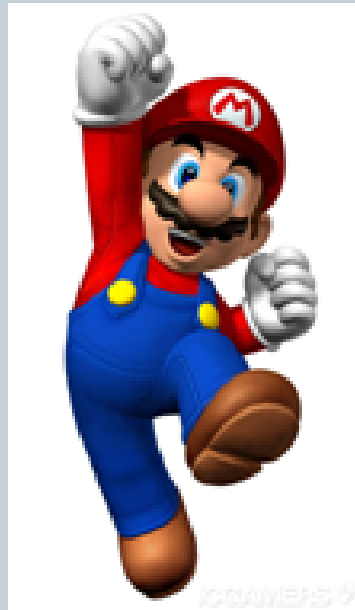
```
(define e  
  (extend-env 'd 6  
    (extend-env 'y 8  
      (extend-env 'x 7  
        (extend-env 'y 14  
          (empty-env))))))
```

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

$Env\text{-}exp ::= (\text{empty-env})$

$::= (\text{extend-env } Identifier \text{ Scheme-value } Env\text{-}exp)$

The general form of **define-datatype**



```
(define-datatype environment environment?  
  (empty-env)  
  (extend-env  
    (bvar symbol?)  
    (bval expval?)  
    (saved-env environment?))  
  (extend-env-rec  
    (id symbol?)  
    (bvar symbol?)  
    (body expression?)  
    (saved-env environment?)))
```

```
(define-datatype type-name type-predicate-name  
  { (variant-name { (field-name predicate) }*) }+ )
```

Example uses of `define-datatype`



- Lets define a “triple” structure using racket

Depending on how you look at it, **Racket** is

- a *programming language*—a dialect of Lisp and a descendant of Scheme;

See [Dialects of Racket and Scheme](#) for more information on other dialects of Lisp and how they relate to Racket.

- a *family* of programming languages—variants of Racket, and more; or
- a set of *tools*—for using a family of programming languages.

Where there is no room for confusion, we use simply *Racket*.

Racket’s main tools are

- **racket**, the core compiler, interpreter, and run-time system;
- **DrRacket**, the programming environment; and
- **raco**, a command-line tool for executing **Racket** commands that install packages, build libraries, and more.

Example uses of `define-datatype`



$S\text{-list} ::= (\{S\text{-exp}\}^*)$
 $S\text{-exp} ::= \text{Symbol} \mid S\text{-list}$

```
(define-datatype s-list s-list?
  (empty-s-list)
  (non-empty-s-list
   (first s-exp?)
   (rest s-list?)))
```

```
(define-datatype s-exp s-exp?
  (symbol-s-exp
   (sym symbol?))
  (s-list-s-exp
   (slst s-list?)))
```

Nugget



We can represent any data structure
easily using define-datatype