#### **Announcements**

- Midterm coming
- 2. PS'es announced earlier (+1 day)
- 3. Extra day (+1 day)

1

# Lecture 9 – Review Data Abstraction

Interfaces & Representation

T. METIN SEZGIN

ว

#### Lecture Nuggets

- A handful of key concepts in programming languages
  - o 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

3

#### Interface vs. Implementation

- Teasing out the "interface" and the "implementation"
  - o 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."

$$\begin{array}{l} (\texttt{zero}) = \left\lceil 0 \right\rceil \\ (\texttt{is-zero?} \left\lceil n \right\rceil) = \left\{ \begin{array}{l} \texttt{\#t} & n = 0 \\ \texttt{\#f} & n \neq 0 \end{array} \right. \\ (\texttt{successor} \left\lceil n \right\rceil) = \left\lceil n + 1 \right\rceil & (n \geq 0) \\ (\texttt{predecessor} \left\lceil n + 1 \right\rceil) = \left\lceil n \right\rceil & (n \geq 0) \end{array}$$

5

#### 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... (plus  $\lceil x \rceil \lceil y \rceil$ ) =  $\lceil x + y \rceil$

#### **Back to Natural Numbers**

- Constructors
- Observers

$$\begin{array}{l} (\texttt{zero}) = \lceil 0 \rceil \\ (\texttt{is-zero?} \lceil n \rceil) = \left\{ \begin{array}{l} \texttt{\#t} & n = 0 \\ \texttt{\#f} & n \neq 0 \end{array} \right. \\ (\texttt{successor} \lceil n \rceil) = \lceil n + 1 \rceil & (n \geq 0) \\ (\texttt{predecessor} \lceil n + 1 \rceil) = \lceil n \rceil & (n \geq 0) \end{array}$$

7

# Implementation of Natural Numbers

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

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

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
  - o 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)))
```

q

#### Yet another implementation

- Bignum representation
  - Use numbers in base N  $\lceil n \rceil =$

$$\lceil n \rceil = \begin{cases} () & n = 0 \\ (r \cdot \lceil q \rceil) & n = qN + r, \ 0 \le r < N \end{cases}$$

Such that

$$N = 16$$
, then  $\lceil 33 \rceil = (1 \ 2)$  and  $\lceil 258 \rceil = (2 \ 0 \ 1)$   
 $258 = 2 \times 16^0 + 0 \times 16^1 + 1 \times 16^2$ 

• Scheme implementation?

#### Lecture 10

Representation Strategies for Data Types

T. METIN SEZGIN

11

#### **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

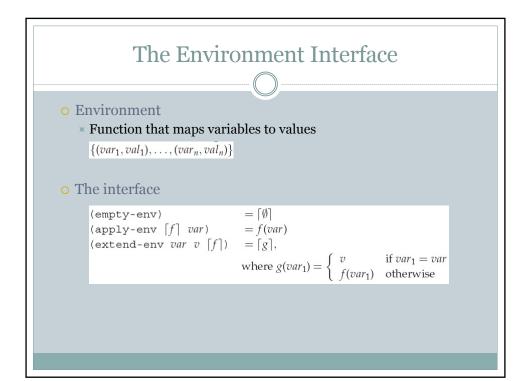
Nugget

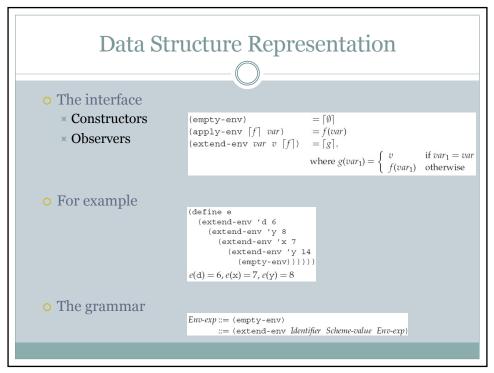
The environment allows us to store variable value pairs

13

### Representation strategies

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





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

17

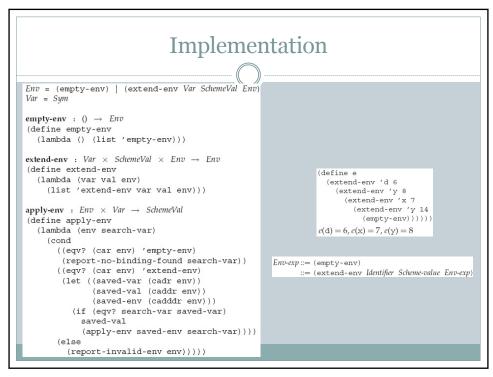
```
Implementation

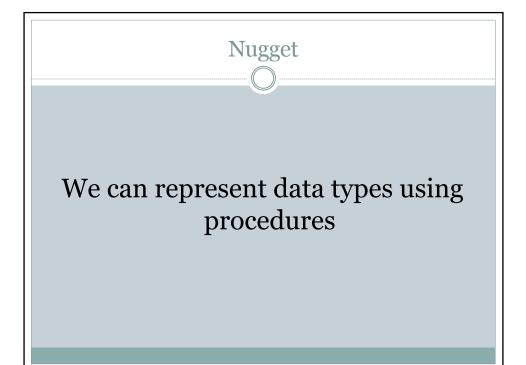
Env = (empty-env) | (extend-env Var SchemeVal Env)
Var = Sym

empty-env : () → Env
(define empty-env
(lambda () (list 'empty-env)))

extend-env : Var × SchemeVal × Env → Env
(define extend-env
(lambda (var val env)
(list 'extend-env var val env)))

apply-env : Env × Var → 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-val (caddr env))
(saved-var (caddr env))
```





# **Procedural Representation**

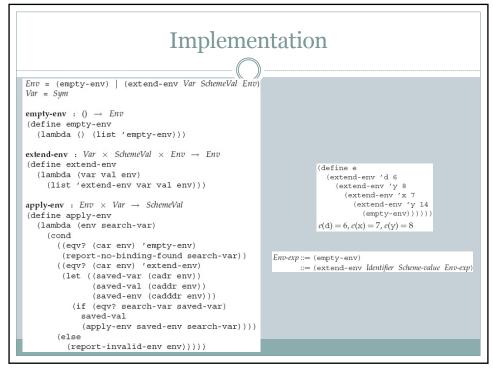
```
Env = Var → SchemeVal
empty-env : () \rightarrow Env
(define empty-env
  (lambda ()
    (lambda (search-var)
       (report-no-binding-found search-var)))
extend-env : Var \times SchemeVal \times Env \rightarrow Env
(define extend-env
  (lambda (saved-var saved-val saved-env)
    (lambda (search-var)
       (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)))
```

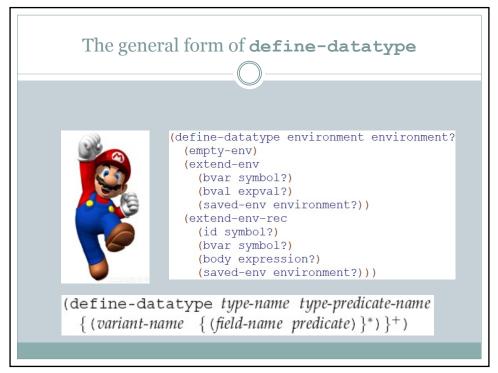
21

# Nugget

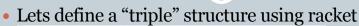
# We can automate mundane data type definitions

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





### Example uses of define-datatype



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.

25

#### Example uses of define-datatype

S-list ::= ( $\{S$ -exp $\}$ \*) S-exp ::= Symbol | S-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