

# Lecture 7

## Recursive Procedures & Auxiliary Procedures



T. METIN SEZGIN

# How do we go about the implementation?



- The grammar

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

$S\text{-list} ::= ()$   
 $\quad ::= (S\text{-exp} . S\text{-list})$   
 $S\text{-exp} ::= \text{Symbol} \mid S\text{-list}$

- The procedure

```
subst : Sym × Sym × S-list → S-list
(define subst
  (lambda (new old slist)
    (if (null? slist)
        '()
        (cons
         (subst-in-s-exp new old (car slist))
         (subst new old (cdr slist))))))
```

```
subst-in-s-exp : Sym × Sym × S-exp → S-exp
(define subst-in-s-exp
  (lambda (new old sexp)
    (if (symbol? sexp)
        (if (eqv? sexp old) new sexp)
        (subst new old sexp))))
```

# Take home message



## Follow the Grammar

More precisely:

- Write one procedure for each nonterminal in the grammar. The procedure will be responsible for handling the data corresponding to that nonterminal, and nothing else.
- In each procedure, write one alternative for each production corresponding to that nonterminal. You may need additional case structure, but this will get you started. For each nonterminal that appears in the right-hand side, write a recursive call to the procedure for that nonterminal.

```
S-list ::= ()  
         ::= (S-exp . S-list)  
S-exp ::= Symbol | S-list
```

# Lecture notes



Complicated Substitution Model:

•  $((\text{lambda}(x) (\text{lambda}(x) x)) a)$

bounded var.

NOT bounded to this x

bounded

PROBLEM

- be careful when modeling substitution
- concept of occurring freely

- Arda Poyraz

# Lecture notes



Occurs-free?:

| x             | exp | var |      |
|---------------|-----|-----|------|
| x             | y   |     | → #t |
| x             | x   |     | → #f |
| (lambda(x) x) | x   |     | → #f |
| (lambda(x) y) | y   |     | → #t |

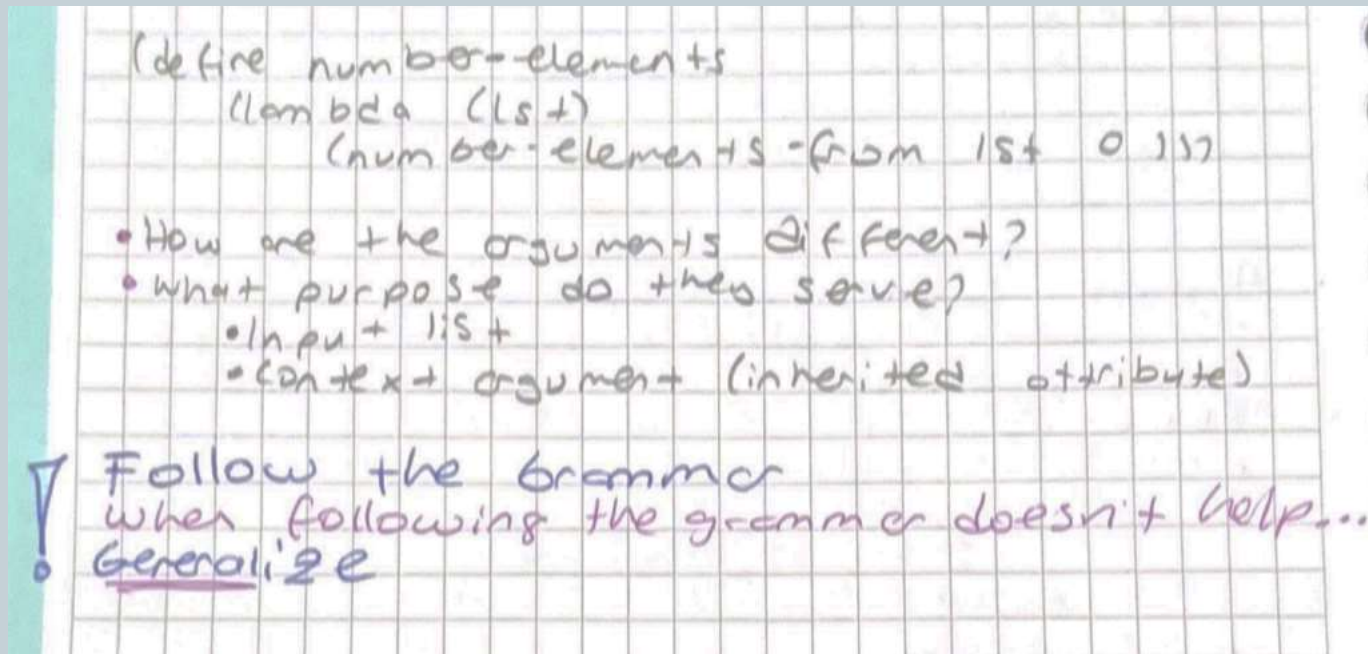
occurs-free?:

$Sym \times LExpr \rightarrow Bool$     TYPE

returns #t if the symbol var  
occurs free in exp, otherwise #f    USAGE

- Hasan Buhurcu
- Berat Karayilan

# Lecture notes



- Betul Korkmaz

The take home message



Follow the grammar

When following the grammar doesn't help...

Generalize

# Lecture 8

# 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

# Nugget



There are handful of key concepts in  
programming languages

# Data abstraction



- Value
- Representation
- Implementation
- Interface
- Abstraction

# 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...
- Examples of interface in real life

# Examples of Interface & Implementation



- Electricity
  - Interface
  - Implementation

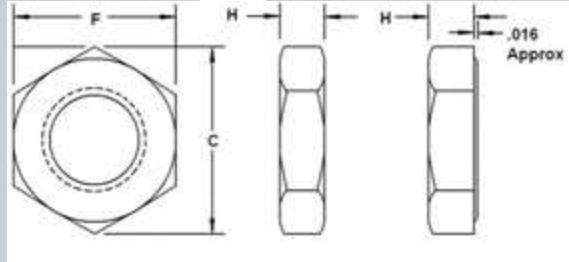


# Examples of Interface & Implementation



- Interface

- Nut



- Implementation

- Pipe wrench



- Adjustable spanner



- Wrench set



# 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...
- Examples of interface in real life
  - Electricity
    - ✦ Interface:
    - ✦ Implementation:
  - Nuts and bolts
    - ✦ Interface:
    - ✦ Implementation:
  - Cutlery
    - ✦ Interface:
    - ✦ Implementation:

# Nugget



May have many implementations for  
an interface (allows abstraction)



# Data Abstraction



- The ability to separate certain aspects of programming using interfaces and implementations.
- Advantages of Data Abstraction
  - Simplifies programming
  - Simplifies editing
  - Simplifies understanding
  - Hides unnecessary complexity

# Nugget



Representation of a value may take  
different forms

# Representation vs. Value



## Natural Numbers

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

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

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

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

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

# 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

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

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

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

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

# 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)))
```

# Yet another implementation



- **Bignum representation**

- Use numbers in base N

$$[n] = \begin{cases} () & n = 0 \\ (r \ . \ [q]) & n = qN + r, 0 \leq r < N \end{cases}$$

- Such that

$$N = 16, \text{ then } [33] = (1 \ 2) \text{ and } [258] = (2 \ 0 \ 1) \\ 258 = 2 \times 16^0 + 0 \times 16^1 + 1 \times 16^2$$

- Scheme implementation?



# 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

|  |                 |
|--|-----------------|
| <code>(empty-env)</code>   | $= [\emptyset]$ |
| <code>(apply-env <math>[f]</math> <math>var</math>)</code>                 | $= f(var)$      |
| <code>(extend-env <math>var</math> <math>v</math> <math>[f]</math>)</code> | $= [g],$        |

$\text{where } g(var_1) = \begin{cases} v & \text{if } var_1 = var \\ f(var_1) & \text{otherwise} \end{cases}$

# 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))))))
```

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

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

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