Lecture 7
Recursive Procedures & Auxiliary Procedures

T. METIN SEZGIN

How do we go about the implementation?

- The grammar

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

$$\begin{aligned} S\text{-}list &::= () \\ &::= (S\text{-}exp \;.\; S\text{-}list) \\ S\text{-}exp &::= Symbol \mid S\text{-}list \end{aligned}$$

- The procedure

subst : $Sym \times Sym \times S\text{-}list \rightarrow S\text{-}list$

```
(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 : $Sym \times Sym \times S\text{-}exp \rightarrow S\text{-}exp$

```
(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.

$$\begin{aligned} \textit{S-list} &::= (\,) \\ &::= (\textit{S-exp} \;\; . \;\; \textit{S-list}) \\ \textit{S-exp} &::= \textit{Symbol} \mid \textit{S-list} \end{aligned}$$

A more complex example

- Consider the procedure **`number-elements`**
- This procedure should take a list **`(`**$\mathbf{v_0}$ $\mathbf{v_1}$ $\mathbf{v_2}$ …**`)`** and return **`((0`** $\mathbf{v_0}$**`)`** **`(1`** $\mathbf{v_1}$**`)`** …**`))`**.

usage: `(number-elements-from '(`v_0 v_1 v_2 `...)` n`)`
`= ((`n v_0`)` `(`$n+1$ v_1`)` `(`$n+2$ v_2`)` `...)`

```
(define number-elements-from
  (lambda (lst n)
    (if (null? lst) '()
      (cons
        (list n (car lst))
        (number-elements-from (cdr lst) (+ n 1))))))
```

number-elements : $List \rightarrow Listof(List(Int, SchemeVal))$

```
(define number-elements
  (lambda (lst)
    (number-elements-from lst 0)))
```

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

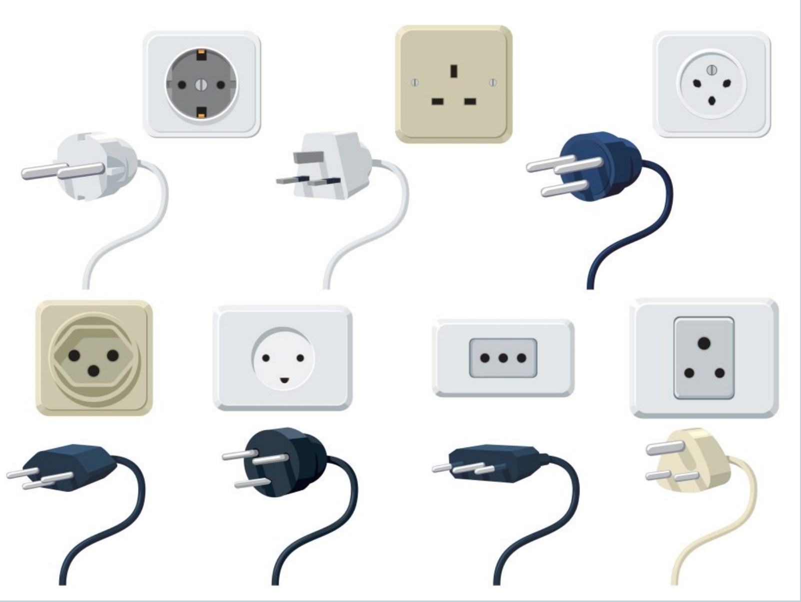
- Value
- Representation
- Implementation
- Interface
- Abstraction

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

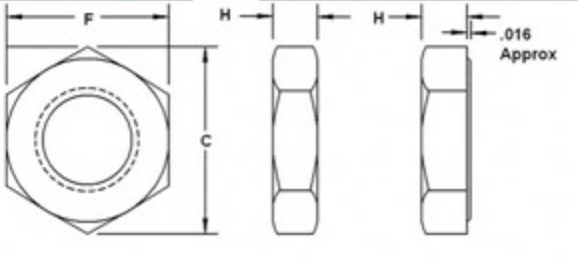
Examples of Interface & Implementation

- 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."

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

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

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

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

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… $(\texttt{plus}\ \lceil x \rceil\ \lceil y \rceil) = \lceil x + y \rceil$

Back to Natural Numbers

- Constructors
- Observers

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

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

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

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

Implementation of Natural Numbers

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

$$\lceil 0 \rceil = \texttt{()}$$
$$\lceil n+1 \rceil = \texttt{(\#t . } \lceil n \rceil \texttt{)}$$

  - 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

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

  - Such that

$N = 16$, then $\lceil 33 \rceil = \texttt{(1 2)}$ and $\lceil 258 \rceil = \texttt{(2 0 1)}$

$258 = 2 \times 16^0 + 0 \times 16^1 + 1 \times 16^2$

  - 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

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

- The interface

$$
\begin{aligned}
&\texttt{(empty-env)} && = \lceil \emptyset \rceil \\
&\texttt{(apply-env } \lceil f \rceil \ var\texttt{)} && = f(var) \\
&\texttt{(extend-env } var \ v \ \lceil f \rceil\texttt{)} && = \lceil g \rceil, \\
& && \quad \text{where } g(var_1) = \begin{cases} v & \text{if } var_1 = var \\ f(var_1) & \text{otherwise} \end{cases}
\end{aligned}
$$

Data Structure Representation

- The interface
  - Constructors
  - Observers

```
(empty-env)                  = ⌈∅⌉
(apply-env ⌈f⌉ var)          = f(var)
(extend-env var v ⌈f⌉)       = ⌈g⌉,
```

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

- For example

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

$e(\mathrm{d}) = 6, e(\mathrm{x}) = 7, e(\mathrm{y}) = 8$

- 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 : () → *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-env (cadddr env)))
         (if (eqv? search-var saved-var)
           saved-val
           (apply-env saved-env search-var))))
      (else
        (report-invalid-env env)))))
```

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-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))))))
```

$e(\mathrm{d}) = 6, e(\mathrm{x}) = 7, e(\mathrm{y}) = 8$

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