Data Abstraction

Gul Agha
CS 421
Data Abstraction

Data should be specified via interfaces

- *Interface*
  - What the data represents

- *Implementation*
  - Specific representation of the data and code for operations using the data

*Abstract Data Types*: the data type represented by the interface.

*The rest of program manipulates data only through operations specified by the interface.*
Example: Natural numbers

\[ [v] \quad \text{the representation of } v \]

\[
\text{zero} = [0]
\]

\[
\text{(iszero? } [v] \text{)} = \begin{cases} 
\#t & n = 0 \\
\#f & n \neq 0 
\end{cases}
\]

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

\[
\text{(pred } [n] \text{)} = [n - 1] \quad n \geq 0
\]
Representation Independence

- Client program should *not* rely on the representation of data or operations of an abstract data type.

```
(define plus
  (lambda (x  y)
    (if  (iszero?  x)
      y
      (succ (plus (pred x) y)))))
```
Unary Representation (Church’s encoding)

\[ [0] = ( ) \]
\[ [n + 1] = (\text{cons } \#t \ [n]) \]

(define zero `( ))
(define iszero? null?)
(define succ (lambda (n) (cons #t n)))
(define pred cdr)
Scheme’s Number Representation

(define zero 0)
(define iszero? zero?)
(define succ (lambda (n) (+ n 1)))
(define (pred (lambda (n) (- n 1))))
## Big Num Representation

Represent the numbers in base $N$:

$$\lceil n \rceil = \begin{cases} \text{() if } n = 0 \\ \text{(cons } r \text{ } \lceil q \rceil ) \quad n = qN + r \text{ where } 0 \leq r < N \end{cases}$$

**Example:** $N = 16$

$$\lceil 33 \rceil = (1 \ 2) \quad \lceil 258 \rceil = (2 \ 0 \ 1)$$
Abstraction to Represent Inductive Data Types

- **Constructors**
  - Build elements of inductive data type

- **Predicate**
  - To test if value is of
    - Inductively defined type
    - Subtypes used in definition

- **Extractor**
  - To deconstruct the data type value
Example Data Type

\[
\text{<bintree>} \rightarrow \\
\text{<number>} \\
| \text{<symbol>} \text{<bintree>} \text{<bintree>} \\
\]

bintree is either

- a number
- the cross-product of a symbol and two binary trees
(define-datatype bintree bintree?
    (leaf-node
        (datum number?))
    (interior-node
        (key symbol?)
        (left bintree?)
        (right bintree?)))
Interface of the Data Type

- A 1-argument procedure which constructs a leaf-node
  - leaf-node tests its arguments with number?

- A 3-argument procedure which constructs an interior-node
  - Tests its first argument with symbol?
  - Tests its second and third argument with bintree?
Syntax

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

)
```
Using the Bintree Datatype

(define datatype bintree
  (define leaf-node
    (datum number?))
  (define interior-node
    (key symbol?)
    (left bintree?)
    (right bintree?)))

(define leaf-sum
  (lambda (tree)
    (cases bintree tree
      (leaf-node (n) n)
      (interior-node
        (key left right)
        (+ (leaf-sum left)
          (leaf-sum right))))))

variables resolved lexically!
Creating a bin-tree

(define tree-a
  (interior-node 'a (leaf-node 2) (leaf-node 3)))

(define tree-b
  (interior-node 'b (leaf-node -1) tree-a))

(define tree-c
  (interior-node 'c tree-b (leaf-node 1)))
Concrete Syntax

- Particular representation of an inductive data type

\[
<\text{expression}>
\rightarrow <\text{identifier}>
\rightarrow (\text{lambda} (<\text{identifier}>)
\rightarrow (<\text{expression}>)
\rightarrow (<\text{expression}>, <\text{expression}>)
\]
Abstract Syntax

- Represents the actual structure
- Omits elements that are not necessary
- Represented as an ordered tree
  - Root is grammatical form (nonterminal)
  - Leaves are terminal
  - Tree represents the application of a rule
Example

\( (\text{lambda} \ (x) \ (f \ (f \ x))) \)
Abstract Syntax for Expressions

*May be represented by record datatype*

```
(define-datatype expression expression?
  (var-exp   (id symbol?))
  (lambda-exp
   (id symbol?)
   (body expression?))
  (app-exp
   (rator expression?)
   (rand expression?)))
```
Free Variable?

(define occurs-free?
(lambda (var exp)
  (cond
    ((symbol? exp) (eqv? var exp))
    ((eqv? (car exp) 'lambda)
     (and (not (eqv? (caadr exp) var))
         (occurs-free? var (caddr exp))))
    (else (or (occurs-free? var (car exp))
              (occurs-free? var (cadr exp)))))))

- The data representation is exposed..
Free Variable?

- In terms of the abstract data type
  (define occurs-free?
   (lambda (var exp)
     (cases expression exp
       (var-exp (id) (eqv? id var))
       (lambda-exp (id body)
         (and (not (eqv? id var))
           (occurs-free? var body)))
       (app-exp (rator rand)
         (or (occurs-free? var rator)
           (occurs-free? var rand))))))
(define parse-expression
  (lambda (datum)
    (cond
      ((symbol? datum) (var-exp datum))
      ((pair? datum)
       (if (eqv? (car datum) 'lambda)
           (lambda-exp (caadr datum)
                        (parse-expression (caddr datum)))
           (app-exp
            (parse-expression (car datum))
            (parse-expression (cadr datum))))))
  ))
A Richer Language

<expression>
→ <number> lit-exp (datum)
→ <var-exp> var-exp (id)
→ (if (expression) (expression) (expression)) if-exp (test-exp true-exp false-exp)
→ (lambda (<identifier>) <expression>) lambda-exp (id body)
→ (<expression> <expression>) app-exp (rator rand)
Digression: More useful higher order functions

- Reduce
- Zip

Reference

Also has fun web programming package in Scheme (go up in webpage)
Reduce

Successive reductions by a binary operator

Reduce Left

Reduce Right
Example

- (reduce-left - '(1 2 3 4 5))
  -13
- (reduce-right - '(1 2 3 4 5))
  3
- (reduce-left append
  (list (list 1 2 3) (list 'a 'b 'c)))
  (1 2 3 a b c)
reduce-right defined

(define reduce-right
  (lambda (f lst)
    (if (null? (cdr lst))
      (car lst)
      (f (car lst)
        (reduce-right f (cdr lst))))))
Trace of call to reduce right

> (reduce-right - '(1 2 3 4 5))
| (reduce-right #<primitive:-> (1 2 3 4 5))
| (reduce-right #<primitive:-> (2 3 4 5))
| | (reduce-right #<primitive:-> (3 4 5))
| | (reduce-right #<primitive:-> (4 5))
| | | (reduce-right #<primitive:-> (5))
| | | | 5
| | | -1
| | | 4
| | -2
| 3
3
Iterative Procedure for reduce-left

(define reduce-left
  (lambda (f lst)
    (reduce-help-left f (cdr lst) (car lst)))))

(define reduce-help-left
  (lambda (f lst res)
    (if (null? lst)
      res
      (reduce-help-left f (cdr lst)
        (f res (car lst)))))
Sample execution of reduce-left

```
> (reduce-left - '(1 2 3 4 5))
|(reduce-help-left #<primitive:-> (2 3 4 5) 1)
|(reduce-help-left #<primitive:-> (3 4 5) -1)
|(reduce-help-left #<primitive:-> (4 5) -4)
|(reduce-help-left #<primitive:-> (5) -8)
|(reduce-help-left #<primitive:-> ( ) -13)
|-13
-13
```
Zip

- Composes two lists of equal length to a single list by means of zipping their elements pair-wise.
Example of zip

- (zip cons '(1 2 3) '(a b c))
  ((1 . a) (2 . b) (3 . c))

- (zip + '(1 2 3) '(4 5 6))
  (5 7 9)
zip implementation

(define zip
  (lambda (z lst1 lst2)
    (if (null? lst1)
        '()
        (cons (z (car lst1) (car lst2))
            (zip z (cdr lst1) (cdr lst2))))))