Part II: Data Abstraction

Marco T. Morazán

Seton Hall University

Outline

1. Interface and Implementation

Definition and Implementation

- Representing a set means defining a new data type

Definition and Implementation

- Representing a set means defining a new data type
- Values are the representations
- Operations are procedures that manipulate them

Definition and Implementation

- Representing a set means defining a new data type
- Values are the representations
- Operations are procedures that manipulate them
- Data abstraction divides a data type in two pieces: interface and implementation

Definition and Implementation

- Representing a set means defining a new data type
- Values are the representations
- Operations are procedures that manipulate them
- Data abstraction divides a data type in two pieces: interface and implementation
- Interface
    - what the data type represents
    - what the operations on the data are
    - what properties these operations have

Definition and Implementation

- Representing a set means defining a new data type
- Values are the representations
- Operations are procedures that manipulate them
- Data abstraction divides a data type in two pieces: interface and implementation
- Interface
  - what the data type represents
  - what the operations on the data are
  - what properties these operations have
- Implementation
  - provides a specific representation of the data
  - provides code for the operations

Definition and Implementation

- Representing a set means defining a new data type
- Values are the representations
- Operations are procedures that manipulate them
- Data abstraction divides a data type in two pieces: interface and implementation
- Interface
  - what the data type represents
  - what the operations on the data are
  - what properties these operations have
- Implementation
  - provides a specific representation of the data
  - provides code for the operations
- Using abstract data types
  - Program manipulates data only through the defined operations
  - Changing the representation only requires changing how the implementation of the operations in the interface; program that use the data types are unchanged

Definition and Implementation

- Idea is well-known to you
  - numbers: add, subtract, multiply, etc.
  - files: open, close, save, etc.

Definition and Implementation

- Idea is well-known to you
  - numbers: add, subtract, multiply, etc.
  - files: open, close, save, etc.
- OOP
  - Classes implement interfaces
  - Programs use classes to declare and manipulate objects

Definition and Implementation

- Idea is well-known to you
  - numbers: add, subtract, multiply, etc.
  - files: open, close, save, etc.
- OOP
  - Classes implement interfaces
  - Programs use classes to declare and manipulate objects
- The *most important* part of an implementation is the *specification of how data is represented*
- $|v|$ denotes the representation of data v

Definition and Implementation

Nonnegative Integers

- Interface
  - zero $= \lceil 0 \rceil$
  - (isZero? $\lceil n \rceil$), #t if $n = 0$ and #f if $n \neq 0$
  - (succ $\lceil n \rceil$) $= \lceil n + 1 \rceil$, $n \geq 0$
  - (pred $\lceil n + 1 \rceil$) $= \lceil n \rceil$, $n \geq 0$
  - (dec2nnint $\lceil n \rceil_{10}$) $= \lceil n \rceil$
  - (nnint2dec $\lceil n \rceil$) $= \lceil n \rceil_{10}$
- No details on how nonnegative integers are represented
- Only requires that procedures produce the desired behavior

Definition and Implementation

Nonnegative Integers

- Interface
  - zero $= \lceil 0 \rceil$
  - (isZero? $\lceil n \rceil$), #t if $n = 0$ and #f if $n \neq 0$
  - (succ $\lceil n \rceil$) $= \lceil n + 1 \rceil$, $n \geq 0$
  - (pred $\lceil n + 1 \rceil$) $= \lceil n \rceil$, $n \geq 0$
  - (dec2nnint $\lceil n \rceil_{10}$) $= \lceil n \rceil$
  - (nnint2dec $\lceil n \rceil$) $= \lceil n \rceil_{10}$
- No details on how nonnegative integers are represented
- Only requires that procedures produce the desired behavior
- Write a program to add to nonnegative integers

Definition and Implementation

Nonnegative Integers

- Interface
  - zero $= |0|$
  - (isZero? $|n|$), #t if $n = 0$ and #f if $n \neq 0$
  - (succ $|n|$) $= |n + 1|$, $n \geq 0$
  - (pred $|n + 1|$) $= |n|$, $n \geq 0$
  - (dec2nnint $|n|_{10}$) $= |n|$
  - (nnint2dec $|n|$) $= |n|_{10}$
- No details on how nonnegative integers are represented
- Only requires that procedures produce the desired behavior
- Write a program to add to nonnegative integers
-
```
;; nnint nnint → nnint
;; Purpose: Add the given nnints
(define (plus x y)
  (if (isZero? x)
      y
      (succ (plus (pred x) y))))

(check-equal? (nnint2dec (plus (dec2nnint 0) (dec2nnint 0))) 0)
(check-equal? (nnint2dec (plus (dec2nnint 2) (dec2nnint 0))) 2)
(check-equal? (nnint2dec (plus (dec2nnint 0) (dec2nnint 1))) 1)
(check-equal? (nnint2dec (plus (dec2nnint 3) (dec2nnint 2))) 5)
```

Definition and Implementation

Nonnegative Integers

- Unary Representation

Definition and Implementation

Nonnegative Integers

- Unary Representation
- `;; An nnint is either: 1. '()  2. (cons #t nnint)`

Definition and Implementation

Nonnegative Integers

- Unary Representation
- `;; An nnint is either: 1. '()  2. (cons #t nnint)`
- ```
  ;; Constructors
  ;;  → nnint
  ;; Purpose: Construct zero
  (define (zero) '())
  ```
  ```

Definition and Implementation

Nonnegative Integers

- Unary Representation
- `;; An nnint is either: 1. '()  2. (cons #t nnint)`
- ```
  ;; Constructors
  ;;  → nnint
  ;; Purpose: Construct zero
  (define (zero) '())
  ```
- ```
  ;; number → nnint
  ;; Purpose: Construct nnint for the given decimal nnint
  (define (dec2nnint n) (build-list n (lambda (i) #t)))
  ```

Definition and Implementation

Nonnegative Integers

- Unary Representation
- `;; An nnint is either: 1. '()  2. (cons #t nnint)`
- ```
  ;; Constructors
  ;;  → nnint
  ;; Purpose: Construct zero
  (define (zero) '())
  ```
- ```
  ;; number → nnint
  ;; Purpose: Construct nnint for the given decimal nnint
  (define (dec2nnint n) (build-list n (lambda (i) #t)))
  ```
- ```
  ;; nnint → nnint
  ;; Purpose: Construct the successor of the given nnint
  (define (succ n) (cons #t n))
  ```
  ```

Definition and Implementation

Nonnegative Integers

- Unary Representation
- `;; An nnint is either: 1. '()  2. (cons #t nnint)`
- ```
  ;; Constructors
  ;;  → nnint
  ;; Purpose: Construct zero
  (define (zero) '())
  ```
- ```
  ;; number → nnint
  ;; Purpose: Construct nnint for the given decimal nnint
  (define (dec2nnint n) (build-list n (lambda (i) #t)))
  ```
- ```
  ;; nnint → nnint
  ;; Purpose: Construct the successor of the given nnint
  (define (succ n) (cons #t n))
  ```
- ```
  ;; Observers
  ;; nnint → Boolean
  ;; Purpose: Determine if given nnint is zero
  (define (isZero? n) (null? n))
  ```

Definition and Implementation

Nonnegative Integers

- Unary Representation
- `;; An nnint is either: 1. '()  2. (cons #t nnint)`
- ```
  ;; Constructors
  ;;  → nnint
  ;; Purpose: Construct zero
  (define (zero) '())
  ```
- ```
  ;; number → nnint
  ;; Purpose: Construct nnint for the given decimal nnint
  (define (dec2nnint n) (build-list n (lambda (i) #t)))
  ```
- ```
  ;; nnint → nnint
  ;; Purpose: Construct the successor of the given nnint
  (define (succ n) (cons #t n))
  ```
- ```
  ;; Observers
  ;; nnint → Boolean
  ;; Purpose: Determine if given nnint is zero
  (define (isZero? n) (null? n))
  ```
- ```
  ;; nnint → nnint
  ;; Purpose: Return the predecessor of given nnint
  (define pred cdr)
  ```
  ```

Definition and Implementation

Nonnegative Integers

- Unary Representation
- ;; An nnint is either: 1. '() 2. (cons #t nnint)
- ```
  ;; Constructors
  ;;   → nnint
  ;; Purpose: Construct zero
  (define (zero) '())
  ```
- ```
  ;; number → nnint
  ;; Purpose: Construct nnint for the given decimal nnint
  (define (dec2nnint n) (build-list n (lambda (i) #t)))
  ```
- ```
  ;; nnint → nnint
  ;; Purpose: Construct the successor of the given nnint
  (define (succ n) (cons #t n))
  ```
- ```
  ;; Observers
  ;; nnint → Boolean
  ;; Purpose: Determine if given nnint is zero
  (define (isZero? n) (null? n))
  ```
- ```
  ;; nnint → nnint
  ;; Purpose: Return the predecessor of given nnint
  (define pred cdr)
  ```
- ```
  ;; nnint → number
  ;; Purpose: Return decimal representation of given nnint
  (define (nnint2dec n) (length n))
  ```

Definition and Implementation

Nonnegative Integers

- Racket Numbers Representation

Definition and Implementation

Nonnegative Integers

- Racket Numbers Representation
- `;; A nnint is a nonnegative Racket integer`

Definition and Implementation

Nonnegative Integers

- Racket Numbers Representation
- `;; A nnint is a nonnegative Racket integer`
- `;; Constructors`

```
;;  → nnint    Purpose: Construct zero
(define (zero) 0)
```

Definition and Implementation

Nonnegative Integers

- Racket Numbers Representation
- `;; A nnint is a nonnegative Racket integer`
-
```
;; Constructors
;;  → nnint    Purpose: Construct zero
(define (zero) 0)
```
-
```
;; number → nnint
;; Purpose: Construct nnint for the given decimal nnint
(define (dec2nnint n) n)
```

Definition and Implementation

Nonnegative Integers

- Racket Numbers Representation
- `;; A nnint is a nonnegative Racket integer`
- ```
  ;; Constructors
  ;;  → nnint    Purpose: Construct zero
  (define (zero) 0)
  ```
- ```
  ;; number → nnint
  ;; Purpose: Construct nnint for the given decimal nnint
  (define (dec2nnint n) n)
  ```
- ```
  ;; nnint → nnint
  ;; Purpose: Construct the successor of the given nnint
  (define succ add1)
  ```
  ```

Definition and Implementation

Nonnegative Integers

- Racket Numbers Representation
- `;; A nnint is a nonnegative Racket integer`
- ```
  ;; Constructors
  ;;  → nnint    Purpose: Construct zero
  (define (zero) 0)
  ```
- ```
  ;; number → nnint
  ;; Purpose: Construct nnint for the given decimal nnint
  (define (dec2nnint n) n)
  ```
- ```
  ;; nnint → nnint
  ;; Purpose: Construct the successor of the given nnint
  (define succ add1)
  ```
- ```
  ;; Observers
  ;; nnint → Boolean  Purpose: Determine if given nnint is zero
  (define isZero? zero?)
  ```

Definition and Implementation

Nonnegative Integers

- Racket Numbers Representation
- `;; A nnint is a nonnegative Racket integer`
- ```
  ;; Constructors
  ;;  → nnint    Purpose: Construct zero
  (define (zero) 0)
  ```
- ```
  ;; number → nnint
  ;; Purpose: Construct nnint for the given decimal nnint
  (define (dec2nnint n) n)
  ```
- ```
  ;; nnint → nnint
  ;; Purpose: Construct the successor of the given nnint
  (define succ add1)
  ```
- ```
  ;; Observers
  ;; nnint → Boolean  Purpose: Determine if given nnint is zero
  (define isZero? zero?)
  ```
- ```
  ;; nnint → nnint
  ;; Purpose: Return the predecessor of given nnint
  (define (pred n)
    (if (= n 0)
        (eopl:error 'pred "Zero does not have a predecessor.")
        (sub1 n)))
  ```
  ```

Definition and Implementation

Nonnegative Integers

- Racket Numbers Representation
- ```
  ;; A nnint is a nonnegative Racket integer
  ```
- ```
  ;; Constructors
  ;;  → nnint    Purpose: Construct zero
  (define (zero) 0)
  ```
- ```
  ;; number → nnint
  ;; Purpose: Construct nnint for the given decimal nnint
  (define (dec2nnint n) n)
  ```
- ```
  ;; nnint → nnint
  ;; Purpose: Construct the successor of the given nnint
  (define succ add1)
  ```
- ```
  ;; Observers
  ;; nnint → Boolean  Purpose: Determine if given nnint is zero
  (define isZero? zero?)
  ```
- ```
  ;; nnint → nnint
  ;; Purpose: Return the predecessor of given nnint
  (define (pred n)
    (if (= n 0)
        (eopl:error 'pred "Zero does not have a predecessor.")
        (sub1 n)))
  ```
- ```
  ;; nnint → number
  ;; Purpose: Return decimal representation of given nnint
  (define (nnint2dec n) n)
  ```
  ```

Definition and Implementation

Nonnegative Integers

- Bignum representation

Definition and Implementation

Nonnegative Integers

- Bignum representation
- Numbers are represented in base N (large N)
- The representation uses a list of numbers such that each number is in 0..N-1 (called bigits)

Definition and Implementation

Nonnegative Integers

- Bignum representation
- Numbers are represented in base N (large N)
- The representation uses a list of numbers such that each number is in 0..N-1 (called bigits)
- Inductive Definition of |n|

$$f(n) = \begin{cases} \texttt{'()} & \text{if } n = 0 \\ (cons\ \ r\ \ |q|) & \text{otherwise}, where n = q * N + r, 0 \leq r < N \end{cases}$$

Definition and Implementation

Nonnegative Integers

- Bignum representation
- Numbers are represented in base N (large N)
- The representation uses a list of numbers such that each number is in 0..N-1 (called bigits)
- Inductive Definition of |n|

$$f(n) = \begin{cases} \text{'}() & \text{if } n = 0 \\ (cons\ \ r\ \ |q|) & \text{otherwise}, where n = q * N + r, 0 \leq r < N \end{cases}$$

- N = 100
- |131| = (31 1)
- |87| = (87)
- |6874| = (74 68)

Definition and Implementation

Nonnegative Integers

- Bignum representation
- Numbers are represented in base N (large N)
- The representation uses a list of numbers such that each number is in 0..N-1 (called bigits)
- Inductive Definition of |n|

$$f(n) = \begin{cases} \text{'}() & \text{if } n = 0 \\ (cons\ \ r\ \ |q|) & \text{otherwise}, where n = q * N + r, 0 \leq r < N \end{cases}$$

- N = 100
- |131| = (31 1)
- |87| = (87)
- |6874| = (74 68)
- N = 500
- |6874| = (374 13) = 13*500 +374

Definition and Implementation

- QUIZ: 2.1, include nnint2dec and dec2nnint (Due in 1 week)

Environments

- An environment (env) associates a value with each element of a finite set of vars

Environments

- An environment (env) associates a value with each element of a finite set of vars
- Given a var an env returns its value
- It's a function!

Environments

- An environment (env) associates a value with each element of a finite set of vars
- Given a var an env returns its value
- It's a function!
- Interface (constructors and observers)
  - (empty-env) = $\lceil \emptyset \rceil$, constructor
  - (apply-env $\lceil$e$\rceil$ var) = e(var), observer
  - (extend-env var v $\lceil$e$\rceil$) = $\lceil$g$\rceil$, constructor

$$g(var1) = \begin{cases} v & \text{if } var = var1 \\ f(var1) & \text{otherwise} \end{cases}$$

Environments

- An environment (env) associates a value with each element of a finite set of vars
- Given a var an env returns its value
- It's a function!
- Interface (constructors and observers)
  - (empty-env) = $\lceil\emptyset\rceil$, constructor
  - (apply-env $\lceil$e$\rceil$ var) = e(var), observer
  - (extend-env var v $\lceil$e$\rceil$) = $\lceil$g$\rceil$, constructor

$$g(var1) = \begin{cases} v & \text{if } var = var1 \\ f(var1) & \text{otherwise} \end{cases}$$

- (define e (extend-env 'x 2 (extend-env 'y 1 (empty-env))))
- (apply-env e 'y) is ?
- (apply-env e 'x) is ?
- (apply-env e 'z) is ?

Environments

- Data Structure Representation
- <env> ::= (empty-env)
  ::= (extend-env <symbol> <Racket-value> <env>)

Environments

- Data Structure Representation
- $\langle$env$\rangle$ ::= (empty-env)
  ::= (extend-env $\langle$symbol$\rangle$ $\langle$Racket-value$\rangle$ $\langle$env$\rangle$)
- ```
  ;;  → env
  ;; Purpose: Construct the empty env
  (define (empty-env) '(empty-env))
  ```
  ```

Environments

- Data Structure Representation
- <env> ::= (empty-env)
  ::= (extend-env <symbol> <Racket-value> <env>)
- ```
  ;;  → env
  ;; Purpose: Construct the empty env
  (define (empty-env) '(empty-env))
  ```
- ```
  ;; symbol X env → env
  ;; Purpose: Add a binding to the given env
  (define (extend-env var val e)
    (list 'extend-env var val e))
  ```

Environments

- Data Structure Representation
- <env> ::= (empty-env)
  ::= (extend-env <symbol> <Racket-value> <env>)
- ```
  ;;  → env
  ;; Purpose: Construct the empty env
  (define (empty-env) '(empty-env))
  ```
- ```
  ;; symbol X env → env
  ;; Purpose: Add a binding to the given env
  (define (extend-env var val e)
    (list 'extend-env var val e))
  ```
- ```
  ;; env symbol → X
  ;; Purpose: Get the value of given var in given env
  (define (apply-env e var)
    (cond [(eq? (car e) 'empty-env)
            (eopl:error 'apply-env "No binding for ~s" var)]
          [(eq? (cadr e) var) (caddr e)]
          [else (apply-env (cadddr e) var)]))
  ```
  ```

Environments

- Procedural Representation
- env instances must offer all the observers (cf. an object)

Environments

- Procedural Representation
- env instances must offer all the observers (cf. an object)
-
```
(define (empty-env)
  (lambda (search-var)
    (eopl:error 'apply-env "No binding for ~s" search-var)))
```

Environments

- Procedural Representation
- env instances must offer all the observers (cf. an object)
-
```
(define (empty-env)
  (lambda (search-var)
    (eopl:error 'apply-env "No binding for ~s" search-var)))
```
-
```
(define (extend-env var val e)
  (lambda (search-var)
    (cond [(eqv? search-var var) val]
          [else (apply-env e search-var)])))
```

Environments

- Procedural Representation
- env instances must offer all the observers (cf. an object)
- ```
  (define (empty-env)
    (lambda (search-var)
      (eopl:error 'apply-env "No binding for ~s" search-var)))
  ```
- ```
  (define (extend-env var val e)
    (lambda (search-var)
      (cond [(eqv? search-var var) val]
            [else (apply-env e search-var)])))
  ```
- ```
  (define (apply-env e var) (e var))
  ```
  ```

Environments

- HOMEWORK: 2.7-2.9

Environments

- HOMEWORK: 2.7-2.9
- QUIZ: 2.12 (Due in 1 week)

Abstraction for Recursive Data

<bintree>	::=	<number>
	::=	(<symbol> <bintree> <bintree>)
- Defines the elements of the set as Racket values
- This is a *representation choice*

Abstraction for Recursive Data

<bintree>	::=	<number>
	::=	(<symbol> <bintree> <bintree>)
- Defines the elements of the set as Racket values
- This is a *representation choice*
- What should the interface look like?
- Constructors to build each kind of <bintree>
- a predicate to test if a value is a <bintree>
- a way of distinguishing between a leaf and an internal node
- a way of extracting its components

Abstraction for Recursive Data

- We will use: define-datatype

Abstraction for Recursive Data

- We will use: define-datatype
- Syntax and Semantics

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

- can only be used at the top level
- Creates a variant record data type named type-name
- Each variant has a name and 0 or more fields
- Each field has a name and associated predicate
- No two types may have the same name
- No two variants may have the same name even if they belong to different types
- Each field predicate must be a one-argument function used to assure that the field's values are valid

Abstraction for Recursive Data

- <bintree> ::= <number>
  ::= (<symbol> <bintree> <bintree>)

Abstraction for Recursive Data

- <bintree> ::= <number>
  ::= (<symbol> <bintree> <bintree>)
- `(define-datatype bintree bintree?`

Abstraction for Recursive Data

- <bintree> ::= <number>
  ::= (<symbol> <bintree> <bintree>)
- `(define-datatype bintree bintree?`
- ```
   (leaf-node
     (datum number?))
  ```
  ```

Abstraction for Recursive Data

- <bintree> ::= <number>
  ::= (<symbol> <bintree> <bintree>)
- `(define-datatype bintree bintree?`
-
```
  (leaf-node
    (datum number?))
```
-
```
  (interior-node
    (key symbol?)
    (left bintree?)
    (right bintree?)))
```

Abstraction for Recursive Data

- <bintree> ::= <number>
  ::= (<symbol> <bintree> <bintree>)
- `(define-datatype bintree bintree?`
- ```
  (leaf-node
    (datum number?))
  ```
- ```
  (interior-node
    (key symbol?)
    (left bintree?)
    (right bintree?)))
  ```
- Creates 1-argument function, bintree?, to test if something is a bintree
- Creates a 1-argument function, leaf-node, to create a leaf; Argument is tested with number?. If it fails an error is reported
- Creates a 3-argument procedure, interior-node, to create an interior node; First argument tested with symbol? and other 2 with bintree?

Abstraction for Recursive Data

- $$\begin{array}{lll} \langle\text{slist}\rangle & ::= & \langle(\{\text{sexp}\}^{*}\ \rangle) \\ \langle\text{sexp}\rangle & ::= & \langle\text{symbol}\rangle \\ & ::= & \langle\text{slist}\rangle \end{array}$$

Abstraction for Recursive Data

- $$\begin{array}{lll} \langle\text{slist}\rangle & ::= & \langle(\{\text{sexp}\}^* \rangle) \\ \langle\text{sexp}\rangle & ::= & \langle\text{symbol}\rangle \\ & ::= & \langle\text{slist}\rangle \end{array}$$
- ```
  (define-datatype s-list s-list?
    (empty-s-list)
  ```
  ```

Abstraction for Recursive Data

<slist>	::=	<({sexp}* >)
<sexp>	::=	<symbol>
	::=	<slist>

- ```
  (define-datatype s-list s-list?
    (empty-s-list)
  ```

- ```
    (non-empty-s-list
      (first s-exp?)    creates its own list representation
      (rest s-list?)))
  ```

Abstraction for Recursive Data

-
<slist>	::=	<({sexp}* >)
<sexp>	::=	<symbol>
	::=	<slist>

- ```
  (define-datatype s-list s-list?
    (empty-s-list)
  ```
- ```
    (non-empty-s-list
      (first s-exp?)    creates its own list representation
      (rest s-list?)))
  ```
- ```
  (define-datatype s-exp s-exp?
  ```
  ```

Abstraction for Recursive Data

<slist>	::=	<({sexp}* >)
<sexp>	::=	<symbol>
	::=	<slist>

- ```
  (define-datatype s-list s-list?
    (empty-s-list)
  ```
- ```
    (non-empty-s-list
      (first s-exp?)
      (rest s-list?)))
  ```
  creates its own list representation
- ```
  (define-datatype s-exp s-exp?
  ```
- ```
    (symbol-symbol-exp (data symbol?))
  ```

Abstraction for Recursive Data

- $\langle$slist$\rangle$ ::= $\langle(\{$sexp$\}^* \rangle)$
  $\langle$sexp$\rangle$ ::= $\langle$symbol$\rangle$
  ::= $\langle$slist$\rangle$
- `(define-datatype s-list s-list?`
  `(empty-s-list)`
- `(non-empty-s-list`
  `(first s-exp?)` creates its own list representation
  `(rest s-list?)))`
- `(define-datatype s-exp s-exp?`
- `(symbol-symbol-exp (data symbol?))`
- `(s-list-symbol-exp (data s-list?)) )`

Abstraction for Recursive Data

- <slist> ::= <({sexp}* >)
  <sexp> ::= <symbol> | <slist>

Abstraction for Recursive Data

<slist>	::=	<({sexp}* >)
<sexp>	::=	<symbol> | <slist>

- ```
  (define-datatype s-list s-list?
    (an-s-list  (data  (list-of symbol-exp?))))
  (define (symbol-exp? e)
    (or (symbol? e) (and (pair? e) ((list-of symbol-exp?) e)))
  ```
  ```

Abstraction for Recursive Data

- $\langle$slist$\rangle$ ::= $\langle(\{$sexp$\}^{*}\rangle)$
  $\langle$sexp$\rangle$ ::= $\langle$symbol$\rangle$ | $\langle$slist$\rangle$
- ```
  (define-datatype s-list s-list?
    (an-s-list  (data  (list-of symbol-exp?))))
  (define (symbol-exp? e)
    (or (symbol? e) (and (pair? e) ((list-of symbol-exp?) e)))
  ```
- ```
  (define-datatype s-exp s-exp?
  ```

Abstraction for Recursive Data

- <slist> ::= <({sexp}* >)
  <sexp> ::= <symbol> | <slist>
- ```
  (define-datatype s-list s-list?
    (an-s-list  (data  (list-of symbol-exp?))))
  (define (symbol-exp? e)
    (or (symbol? e) (and (pair? e) ((list-of symbol-exp?) e)))
  ```
- `(define-datatype s-exp s-exp?`
- `  (symbol-symbol-exp (data symbol?))`
  ```

Abstraction for Recursive Data

- $$\begin{array}{lcl} \langle\text{slist}\rangle & ::= & \langle(\{\text{sexp}\}^* \rangle) \\ \langle\text{sexp}\rangle & ::= & \langle\text{symbol}\rangle \mid \langle\text{slist}\rangle \end{array}$$
- ```
  (define-datatype s-list s-list?
    (an-s-list  (data  (list-of symbol-exp?))))
  (define (symbol-exp? e)
    (or (symbol? e) (and (pair? e) ((list-of symbol-exp?) e)))
  ```
- `(define-datatype s-exp s-exp?`
- `  (symbol-symbol-exp (data symbol?))`
- `  (s-list-symbol-exp (data s-list?)) )`
  ```

Abstraction for Recursive Data

- Write a program to sum the numbers in a bintree

```
(define-datatype bintree bintree?
  (leaf-node (data number?))
  (interior-node (key symbol?)
                 (lst bintree?)
                 (rst bintree?)))
```

Abstraction for Recursive Data

- Write a program to sum the numbers in a bintree

```
(define-datatype bintree bintree?
  (leaf-node (data number?))
  (interior-node (key symbol?)
                 (lst bintree?)
                 (rst bintree?)))
```

- ```
  ;; Sample bintree
  (define BT0 (leaf-node 4))
  (define BT1 (interior-node 'T (leaf-node 2) (leaf-node -2)))
  (define BT2 (interior-node 'T
                             (interior-node
                               'L (leaf-node 10) (leaf-node 20))
                             (interior-node
                               'L (leaf-node 30) (leaf-node 40))))
  ```
  ```

Abstraction for Recursive Data

- Write a program to sum the numbers in a bintree

```
(define-datatype bintree bintree?
  (leaf-node (data number?))
  (interior-node (key symbol?)
                 (lst bintree?)
                 (rst bintree?)))
```

- ```
  ;; Sample bintree
  (define BT0 (leaf-node 4))
  (define BT1 (interior-node 'T (leaf-node 2) (leaf-node -2)))
  (define BT2 (interior-node 'T
                             (interior-node
                               'L (leaf-node 10) (leaf-node 20))
                             (interior-node
                               'L (leaf-node 30) (leaf-node 40))))
  ```

- ```
  ;; bintree → number
  ;; Purpose: Add nums in given bintree
  (define (leaf-sum t)
  ```

Abstraction for Recursive Data

- Write a program to sum the numbers in a bintree

```
(define-datatype bintree bintree?
  (leaf-node (data number?))
  (interior-node (key symbol?)
                 (lst bintree?)
                 (rst bintree?)))
```

- ```
  ;; Sample bintree
  (define BT0 (leaf-node 4))
  (define BT1 (interior-node 'T (leaf-node 2) (leaf-node -2)))
  (define BT2 (interior-node 'T
                             (interior-node
                               'L (leaf-node 10) (leaf-node 20))
                             (interior-node
                               'L (leaf-node 30) (leaf-node 40))))
  ```

- ```
  ;; bintree → number
  ;; Purpose: Add nums in given bintree
  (define (leaf-sum t)
  ```

- ```
  (check-equal? (leaf-sum BT0) 4)
  (check-equal? (leaf-sum BT1) 0)
  (check-equal? (leaf-sum BT2) 100)
  ```
  ```

Abstraction for Recursive Data

- Write a program to sum the numbers in a bintree

```
(define-datatype bintree bintree?
  (leaf-node (data number?))
  (interior-node (key symbol?)
                 (lst bintree?)
                 (rst bintree?)))
```

- ```
  ;; Sample bintree
  (define BT0 (leaf-node 4))
  (define BT1 (interior-node 'T (leaf-node 2) (leaf-node -2)))
  (define BT2 (interior-node 'T
                             (interior-node
                               'L (leaf-node 10) (leaf-node 20))
                             (interior-node
                               'L (leaf-node 30) (leaf-node 40))))
  ```
- ```
  ;; bintree → number
  ;; Purpose: Add nums in given bintree
  (define (leaf-sum t)
  ```
- ```
    (cases bintree t
      (leaf-node (val) val)
      (interior-node (k l r)
        (+ (leaf-sum l) (leaf-sum r)))))
  ```
- ```
  (check-equal? (leaf-sum BT0) 4)
  (check-equal? (leaf-sum BT1) 0)
  (check-equal? (leaf-sum BT2) 100)
  ```

Abstraction for Recursive Data

<**exp**>	::=	<number>
	::=	<Boolean>
	::=	<id>
	::=	(lambda (<id>*) <exp>)
	::=	(<exp> <exp>*)
- A BNF grammar specifies a particular representation using specific strings and values
- Called: *Concrete Syntax* or External Representation

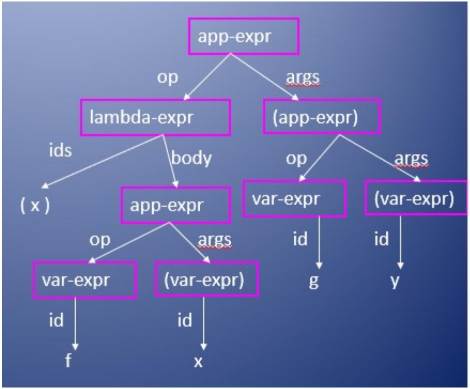
Abstraction for Recursive Data

- $\langle \textbf{exp} \rangle$ ::= $\langle$number$\rangle$
  ::= $\langle$Boolean$\rangle$
  ::= $\langle$id$\rangle$
  ::= (lambda ($\langle$id$\rangle^*$) $\langle$exp$\rangle$)
  ::= ($\langle$exp$\rangle$ $\langle$exp$\rangle^*$)
- A BNF grammar specifies a particular representation using specific strings and values
- Called: *Concrete Syntax* or External Representation
-
```
(define-datatype expr expr?
  (var-expr (id symbol?))
  (num-expr (num number?))
  (bool-expr (b boolean?))
  (lambda-expr
   (params (list-of symbol?))
   (body expr?))
  (app-expr
   (op expr?)
   (args (list-of expr?))))
```
- Called *Abstract Syntax* or Internal Representation
- Terminal symbols are not stored (e.g., keywords, (and))

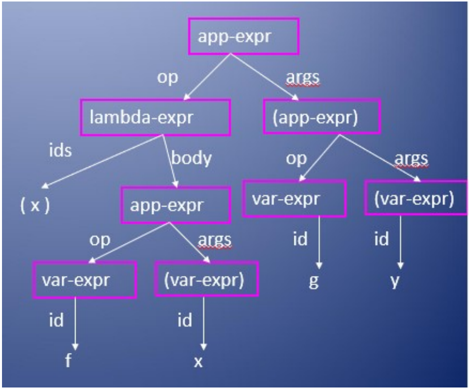
Abstraction for Recursive Data

- An abstract syntax tree is created from an instance of concrete syntax through a process called *parsing*
- Each node corresponds to a step in the syntactic derivation
- Internal nodes are named with a syntactic category
- Edges are labeled with the names occurring in a syntactic category
- Leaves correspond to terminal strings

Abstraction for Recursive Data

- An abstract syntax tree is created from an instance of concrete syntax through a process called *parsing*
- Each node corresponds to a step in the syntactic derivation
- Internal nodes are named with a syntactic category
- Edges are labeled with the names occurring in a syntactic category
- Leaves correspond to terminal strings
- ((lambda (x) (f x)) (g y))
- 

Abstraction for Recursive Data

- An abstract syntax tree is created from an instance of concrete syntax through a process called *parsing*
- Each node corresponds to a step in the syntactic derivation
- Internal nodes are named with a syntactic category
- Edges are labeled with the names occurring in a syntactic category
- Leaves correspond to terminal strings
- ((lambda (x) (f x)) (g y))
  - 

- Transforming an abstract syntax tree back to concrete syntax is called *unparsing*

Abstraction for Recursive Data

- Abstract syntax trees are important
- Programs that manipulate other programs are syntax oriented compilers and interpreters (like Watson's natural language parser and interpreter)
- Transforming a program into a abstract syntax tree makes manipulating programs significantly easier

Environments

- ```
  ;; Sample exp (concrete syntax)
  (define E0 'x)
  (define E1 100)
  (define E2 #t)
  (define E3 '(lambda (x y) (+ x y)))
  (define E4 '((lambda (x y) (+ x z)) 2 3))

  (check-equal? (unparse-lc-expr (parse-lc-exp E0)) E0)
  (check-equal? (unparse-lc-expr (parse-lc-exp E1)) E1)
  (check-equal? (unparse-lc-expr (parse-lc-exp E2)) E2)
  (check-equal? (unparse-lc-expr (parse-lc-exp E3)) E3)
  (check-equal? (unparse-lc-expr (parse-lc-exp E4)) E4)
  ```
  ```

Definition and Implementation

- ```
  ;; exp → expr
  ;; Purpose: Parse the give LC exp
  (define (parse-lc-exp e)
  ```
  ```

Definition and Implementation

- ```
  ;; exp → expr
  ;; Purpose: Parse the give LC exp
  (define (parse-lc-exp e)
  ```
- ```
    (cond [(symbol? e) (var-expr e)]
          [(number? e) (num-expr e)]
          [(boolean? e) (bool-expr e)]
  ```

Definition and Implementation

```
• ;; exp → expr
  ;; Purpose: Parse the give LC exp
  (define (parse-lc-exp e)
•   (cond [(symbol? e) (var-expr e)]
          [(number? e) (num-expr e)]
          [(boolean? e) (bool-expr e)]
•         [(eq? (car e) 'lambda)
           (lambda-expr (cadr e)
                        (parse-lc-exp (caddr e)))]
```

Definition and Implementation

- ```
  ;; exp → expr
  ;; Purpose: Parse the give LC exp
  (define (parse-lc-exp e)
  ```
- ```
    (cond [(symbol? e) (var-expr e)]
          [(number? e) (num-expr e)]
          [(boolean? e) (bool-expr e)]
  ```
- ```
          [(eq? (car e) 'lambda)
           (lambda-expr (cadr e)
                        (parse-lc-exp (caddr e)))]
  ```
- ```
          [else (app-expr
                  (parse-lc-exp (car e))
                  (map (lambda (aexp) (parse-lc-exp aexp))
                       (cdr e)))]))
  ```

Definition and Implementation

```
;; exp → expr
;; Purpose: Parse the give LC exp
(define (parse-lc-exp e)
  (cond [(symbol? e) (var-expr e)]
        [(number? e) (num-expr e)]
        [(boolean? e) (bool-expr e)]
        [(eq? (car e) 'lambda)
         (lambda-expr (cadr e)
                      (parse-lc-exp (caddr e)))]
        [else (app-expr
               (parse-lc-exp (car e))
               (map (lambda (aexp) (parse-lc-exp aexp))
                    (cdr e)))]))
```

```
;; expr → exp
;; Purpose: Unparse the given LC expr
(define (unparse-lc-expr er)
```

Definition and Implementation

- ```
  ;; exp → expr
  ;; Purpose: Parse the give LC exp
  (define (parse-lc-exp e)
  ```
- ```
    (cond [(symbol? e) (var-expr e)]
          [(number? e) (num-expr e)]
          [(boolean? e) (bool-expr e)]
  ```
- ```
          [(eq? (car e) 'lambda)
           (lambda-expr (cadr e)
                        (parse-lc-exp (caddr e)))]
  ```
- ```
          [else (app-expr
                  (parse-lc-exp (car e))
                  (map (lambda (aexp) (parse-lc-exp aexp))
                       (cdr e)))]))
  ```
- ```
  ;; expr → exp
  ;; Purpose: Unparse the given LC expr
  (define (unparse-lc-expr er)
  ```
- ```
    (cases expr er
      (var-expr (s) s)
      (num-expr (n) n)
      (bool-expr (b) b)
  ```

Definition and Implementation

```
• ;; exp → expr
  ;; Purpose: Parse the give LC exp
  (define (parse-lc-exp e)
•   (cond [(symbol? e) (var-expr e)]
          [(number? e) (num-expr e)]
          [(boolean? e) (bool-expr e)]
•         [(eq? (car e) 'lambda)
           (lambda-expr (cadr e)
                        (parse-lc-exp (caddr e)))]
•         [else (app-expr
                (parse-lc-exp (car e))
                (map (lambda (aexp) (parse-lc-exp aexp))
                     (cdr e)))]))
• ;; expr → exp
  ;; Purpose: Unparse the given LC expr
  (define (unparse-lc-expr er)
•   (cases expr er
      (var-expr (s) s)
      (num-expr (n) n)
      (bool-expr (b) b)
•     (lambda-expr (params body)
                   (list 'lambda params (unparse-lc-expr body)))
```

Definition and Implementation

```
• ;; exp → expr
  ;; Purpose: Parse the give LC exp
  (define (parse-lc-exp e)
•   (cond [(symbol? e) (var-expr e)]
          [(number? e) (num-expr e)]
          [(boolean? e) (bool-expr e)]
•         [(eq? (car e) 'lambda)
           (lambda-expr (cadr e)
                        (parse-lc-exp (caddr e)))]
•         [else (app-expr
                 (parse-lc-exp (car e))
                 (map (lambda (aexp) (parse-lc-exp aexp))
                      (cdr e)))]))
• ;; expr → exp
  ;; Purpose: Unparse the given LC expr
  (define (unparse-lc-expr er)
•   (cases expr er
      (var-expr (s) s)
      (num-expr (n) n)
      (bool-expr (b) b)
•     (lambda-expr (params body)
             (list 'lambda params (unparse-lc-expr body)))
•     (app-expr (op args)
            (cons (unparse-lc-expr op)
                  (map (lambda (exr) (unparse-lc-expr exr))
```

Definition and Implementation

- HOMEWORK: 2.21, 2.22, 2.24, 2.25, 2.27, 2.28, 2.29