Data-directed Design

Corky Cartwright

Department of Computer Science

Rice University

From last lecture: List template

```
;; (define (f ... a-list ...)
;;   (cond
;;     [(empty? a-list) ...]
;;     [(cons? a-list)  ... (first a-list) ...
;;       ... (f ... (rest a-list) ...) ...]))
```

Template does not depend on element type. It applies to ***`alpha`*****`-list`** where ***`alpha`*** is any type. In fact, some functions like **`length`**, **`reverse`**, **`append`**, **`first`**, **`rest`** work for all types ***`alpha`*****`-list`** (also written **`(list-of`** ***`alpha`*****`)`**).

Plan for Today

- List abbreviations
- More discussion of the list template
- Data-directed design with numbers
- Strong structural recursion
- Another ubiquitous self-referential data type: trees

List Abbreviations

- Abbreviations
  - Let ***c1***, ***c2***, **...**, ***cn*** be constants (including quoted symbols).
    (list *c1 c2* ... *cn*) abbreviates
    (cons *c1* (cons *c2* ... (cons *cn* empty))...)
  - Let ***s1***, ***s2***, **...**, ***sn*** be symbols , constants (excluding symbols) or lists constructed of such atoms.
  - **'(*s1* ... *sn*)** abbreviates **(list 's1 ... 'sn)**
  - Examples (all equal)
  - `'((1 2) (3 four))`
  - `(list (list 1 2) (list 3 'four))`
  - `(cons (cons 1 (cons 2 empty)) (cons (cons 3 (cons ‘four empty)) empty))`
- Do not nest quoted notation.
- Do not use **true**, **false**, **empty** inside quotation.

A simple list function that takes 2 list arguments

- The append function that concatenates lists is built-in to Racket. We will define this function

```
; app: list-of-alpha list-of-alpha -> list-of-alpha
; purpose: (app a b) concatenates the lists a and b.

; Examples
(check-expect (app '(a b) '(c d)) '(a b c d))
(check-expect (app empty '(c d)) '(c d))
(check-expect (app '(a b) empty) = '(a b)

; Instantiated template (on which argument do we recur?)
|#
  (define (app x y)
    (cond [(empty x?) ...]
          [(cons? x?) ... (first x) ...
               (app (rest x) y) ... ]))
#|
```

app cont.

- ```
  ; Code:
   (define (app x y)
     (cond [(empty x?) y]
           [(cons? x?) (cons (first x) (app (rest x) y))]))

  ; Test?  Already done!
  ```

- Would recurring on the second argument work?
  ```

Using append as an auxiliary function

- **`append`** is included in the Racket library
- concatenation is the common string (a form of list of char) “construction” operation
- *Problem:* cost of operation is not constant; it is proportional to size of first argument (or, in case of strings, size of constructed list)
- Example of function that when simply coded uses append to construct its result: **`flatten`**

Defining Deep Lists and **`flatten`**

```
;; A deepList is either:
;; * empty, or
;; * (cons s adl) where a is a symbol or a deepList and adl is a deepList
;; Examples
(define dl1 '((())))
(define dl2 '((a) ((b))))
(define dl3 '((a b c d (e)) ((f) ((g)))))
;;
;; Template for deepList
#|
(define (f ... dl ... )
  (cond [(empty? dl) ... ]
        [(symbol? dl) ... (flatten (rest dl)) ... ]
        [(cons? dl)
         (cond [(symbol? (first dl)) ... (first dl) ... (flatten (rest dl)) ...)]
               [(cons? (first dl)) ... (flatten (first dl)) ... (flatten (rest dl)) ... ])]))
|#
;; flatten: deepList -> symbol-list
;; Purpose: (flatten dl) consumes a deepList dl and concatenates all of
;; the symbols embedded in dl into a symbol-list where the symbols appear
;; in the same order as when dl is printed as string.
;; input to form a list of elements
```

Defining Deep Lists and **`flatten`** (cont.)

```
;; Examples:
(check-expect (flatten dl1) empty)
(check-expect (flatten dl2) '(a b))
(check-expect (flatten dl3) '(a b c d e f g))

;; Template Instantiation for flatten:
#|
(define (flatten dl)
  (cond [(empty? dl) ... ]
        [(cons? dl)
         (cond [(symbol? (first dl)) ... (first dl) ... (flatten (rest dl)))]
               [(empty? (first dl)) ... (flatten (rest dl)) ... ]
               [(cons? (first dl)) ... (flatten (first dl)) ... (flatten (rest dl)) ... ])]))
|#
;; Code:
(define (flatten dl)
  (cond [(empty? dl) empty ]
        [(cons? dl)
         (cond [(symbol? (first dl)) (cons (first dl) (flatten (rest dl)))]
               [(cons? (first dl)) (append (flatten (first dl)) (flatten (rest dl)))])]))
```

Defining `flatten`

```
;; Tests Done!
```

Improving **`flatten`**? Need a help function with an accumulator; next lecture.

Algebraic Data I

Given a set of constructor symbols $\mathbb{C}$ (with associated arities > 0) and a set of primitive data values $\mathbb{P}$, the domain of *values generated by* $\mathbb{C}$ and $\mathbb{P}$ is inductively defined as follows:

1. Every primitive value $p \in \mathbb{P}$ is a *value*; and
2. For every constructor $c \in \mathbb{C}$ of arity n, and values $v_1, \ldots, v_n$, $c(v_1, \ldots, v_n)$ is a *value.*

If $\mathbb{P}$ is the set of primitive values of basic Racket (which excludes strings, functions, vectors [arrays], and other more complex built-in forms of data) and $\mathbb{C}$ is the set of primitive constructors (only `cons`) plus the constructors defined in a Racket program P, the domain of value available in P is simply the set of values generated by $\mathbb{C}$ and $\mathbb{P}$. In this domain, every data value has the form $p \in \mathbb{P}$ or $c(e_1, e_2, \ldots e_n)$ where $c \in \mathbb{C}$ with arity n and $e_1, e_2, \ldots e_n$ are data values.

Observation From this perspective, every value in a Racket program is a tree. Recall that we are not yet including functions as data values.

Algebraic Data II

- Note that we have explicitly excluded functions from $\mathbb{P}$. Why?
- In a typical functional program, the domain $\mathbb{V}$ includes an enormous amount of "junk" because no restrictions are placed on the value arguments $v_1, v_2, \ldots v_n$ in a construction $c(v_1, v_2, \ldots, v_n)$.
- The ML family of languages, which we will see in Haskell, follows a different conceptual path and imposes type restrictions on the operands of constructions and functions
  - Every (mono)type is disjoint from every other (mono)type.
  - Every value belongs to a unique monotype.

  Haskell has added some interesting workarounds to support a form of subtyping.

Algebraic Data Types I

In the documentation framework that we use for Racket, we introduce type definitions for the purpose of precise program documentation. Note that our types are subsets of the program domain of values $\mathbb{V}$ that can overlap (as in Java). The framework is not designed to support static type checking.

A type definition in a program P has the form

$$T := S_1 \mid S_2 \mid \ldots \mid S_n$$

where

- T is a new name (identifier);
- each S_i is either
  - a recursive subset of $\mathbb{P}$,
  - a type T defined elsewhere in the program
  - an expression $c(T_1, \ldots, T_k)$ denoting the set $\{ c(v_1, \ldots, v_k) \mid v_i \in T_i \}$ where c is a defined constructor (possibly primitive) and T_i is defined elsewhere in the program.

The sets S_i must be disjoint.

We often write these definitions out in prose rather than using the := notation.

There is an obvious structural induction scheme for reasoning about an algebraic type.

Algebraic Data Types II

Our data definition framework is very expressive. Essentially any data domain consisting of freely constructed finite trees can be formulated as algebraic data. Some examples include:

- Files on your computer (at least in Linux)
  - Simple File (an array of characters), or
  - Folder, which contains a list of pairs (string, file)
- XML
  - Baroque format for representing algebraic data as ASCII text
- Internet domain names
- Structurally well-formed programs (abstract syntax)

In some cases, the domain of interest must be embedded in a larger “freely constructed domain”. For the domain of ascending integer-lists must be embedded in a larger domain such as all integer-lists. The former is not an algebraic type but the latter is.

On the other hand, some forms of data are best characterized as quotients of algebraic types. I am not aware of a mainstream functional language that directly supports data definitions that construct quotients of algebraic types. In contrast, this form of data definition is easily done in many class-based OO languages.

Inductive Structure of $\mathbb{N}$

- Standard definition from mathematics

```
;; A natural-number (natural for short) is either
;;      0, or
;;      (add1 n)
;; where n is a natural-number (natural)
```

- We often use the symbol $\mathbb{N}$ to denote this domain.
- Comments:
  - In mathematics, add1 is usually called succ, suc, or S, for *successor.*
  - Principle of mathematical induction for the natural numbers is based on this definition:

$$\frac{P(0), \quad \forall x\, [P(x) \rightarrow P(add1(x))]}{\forall x\, P(x)}$$

- Is there an analogous induction principle for other forms of inductively defined data? Yes!

Basic Operations on Naturals

- Examples (using constructors)
  - Zero: **`0`**
  - One: **`(add1 0)`**
  - Four: **`(add1 (add1 (add1 (add1 0))))`**
- Accessors:
  - **`sub1 : N -> N`**

    Note: **`sub1`** is typically called **`pred`** or P in mathematical logic; in Racket (**`sub1`** **0)** is not an error (for reasons explained later).
- Recognizers:
  - **`zero? : Any -> bool`**
  - **`positive? : Any -> bool ;; not called add1?`**

Basic Laws (Reductions) for Natural Numbers

- The rules for primitive or auto-generated (for **`define-struct`**) operation for a (typically infinite) table
- Recall the ones for lists:
  - For all values `v`, and list values `l`, we have
    - **`(empty? empty) = true`** ;; recognizer
    - **`(empty? (cons v l)) = false`**
    - **`(rest (cons v l)) = l`** ;; accessor
    - **`(first (cons v l)) = v`**
- Basic laws:
  - For all natural numbers `n`, we have
    - **`(zero? 0) = true`** ;; recognizer
    - **`(zero? (add1 n)) = false`**
    - **`(positive? (add1 n)) = true`**
    - **`(positive? 0) = false`**
    - **`(sub1 (add1 n)) = n`** ;; accessor
- Similar rules exist for **all** inductively-defined data types
- What about laws for **`(equal? ...)`**

Natural Numbers: Template

- Template for **`natural`** is very similar to lists:

```
;; f : natural-number -> ...
;; (define (f n)
;;   (cond [(zero? n) ...]
;;         [(positive? n)
;;          ...(f (sub1 n)) ...]))
```

Example

- Write a function that repeats a symbol s several (n) times

- Examples

```
 (repeat 'Rabbit 0) = empty
 (repeat 'Rabbit (add1 (add1 0)))
     =  '(Rabbit 'Rabbit)
```

- Code:

```
;; repeat : symbol natural -> symbol-list
(define (repeat s n)
 (cond [(zero? n) empty]
       [else (cons s (repeat s (sub1 n)))]))
```

Generalization: Full Structural Recursion

- Corresponds to "strong induction" on natural numbers

$$\frac{P(0),\ \forall n\ [\forall n'<n\ P(n)\ ] \rightarrow P(S(x))]}{\forall n\ P(n)}$$

- Template instantiation includes recursive calls on deeper "predecessors" than the immediate ones; the instantiation must anticipate what predecessors are required.

Example of Full Structural Recursion

```
;; fib: natural -> natural
;; Template instantiation
;;(define (fib n)
;;  (cond [(< n 2) ...]
;;        [(positive? n) .. (fib (- n 1))
;;            .. (fib (- n 2)) .. )]))
;;)
;; Code:
  (define (fib n)
   (cond [(< n 2) 1]
         [(positive? N) (+ (fib (- n 1)) (fib (- n 2)))]))
```

Defining Add

```
(define (add m n)
 (cond
  [(zero? m) n]
  [(positive? m) (add1 (add (sub1 m) n))]))

(define (right-add m n)
 (cond
  [(zero? n) m]
  [(positive? n) (add1 (right-add m (sub1 n)))]))
```

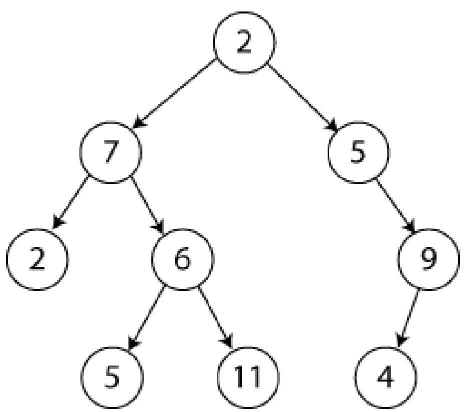
Defining Integers

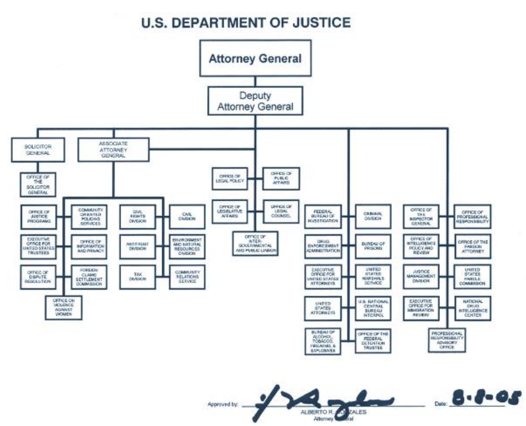
- An integer is either:
  - `0`; or
  - **`(add1 n)`** where **`n`** has the form **`0`** or **`(add1 …)`** [non-negative]; or
  - **`(sub1 n)`** where **`n`** has the form **`0`** or **`(sub1 …)`** [non-positive].
- Recognizers:
  - **`zero?:`** `any -> bool`
  - **`positive?:`** `any -> bool`
  - **`negative?:`** `any -> bool`
- In Racket, **`add1`** and **`sub1`** have been extended to all integers by defining for all integers **`n`** :
  - **`(add1 (sub1 n))`** `=` **`n`**
  - **`(sub1 (add1 n))`** `=` **`n`**
- Hence, **`(add1 -1)`** **and** **`(sub1 0)`** are not errors.

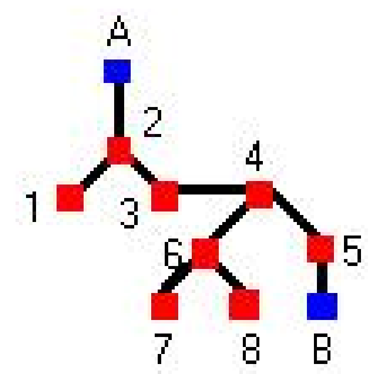
Another Inductive Type: Trees

- Labeled trees
- Organizational charts
- Decision trees
- Search trees

and many more!







From Lists to Trees

Example of a List Data Definition

```
;; Given the built-in two argument constructor cons with
;; fields first and rest:
;; An alpha-list is
;; * empty, or
;; * (cons s los)
;; where s is an alpha and los is a alpha-list
```

Example of a Tree Data Definition

```
;; Given the struct definition
(define-struct person (name mother father))
; An ancestryTree is
; * empty    (representing "unknown origin" or "none")
; * (make-person n m f)    (with two self-references)
; where n is a symbol, m is a person and f is a person
```

Examples of ancestryTree

```
(make-person 'Bob
  (make-person 'Jane empty
    (make-person 'Tom
      (make-person 'Cat empty empty) empty))
  (make-person ’Rob empty
    (make-person ’Sue empty
      (make-person 'Ray empty
        (make-person ’Johny empty empty)))))
```

Template for ancestryTree

- In non-empty trees, we anticipate accessing each child of the tree:

```
; f : ancestryTree -> ...
; (define (f ... at ...)
;  (cond
;     [(empty? at) ...]
;     [else ... (person-name at) ...
;       ...  (person-mother c) ...
;       ...  (person-father c) ...)
```

Template for Processing a Tree

- Recursion in type → recursion in template

```
; f : person -> ...
; (define (f ... c ...)
;  (cond
;    [(empty? c) ...]
;    [else ... (person-name c) ...
;     ... (f (person-mother c)) ...
;     ... (f (person-father c))...)
```

Example: Tree Depth

- Consider the following problem
  - Given an ancestry tree, compute the maximum number of generations for which we know something about this person.
- Type Contract: **`person -> natural`**
- Examples (next slide)
- Template?

Tree Depth Examples

```
(define cat (make-person 'Cat empty empty))
(define tom (make-person 'Tom cat empty))
(define jane (make-person empty tom))
(define johnny (make-person 'Johnny empty empty))
(define ray (make-person 'Ray empty johnny))
(define sue (make-person 'Sue empty ray))
(define rob (make-person 'Rob empty sue))
(define bob (make-person 'Bob jane rob))

(check-expect (max-depth cat) 1)
(check-expect (max-depth tom) 2)
(check-expect (max-depth jane) 3)
(check-expect (max-depth johnny) 1)
(check-expect (max-depth ray) 2)
(check-expect (max-depth sue) 3)
(check-expect (max-depth rob) 4)
(check-expect (max-depth bob) 5)
```

Tree Depth Template Instantiation

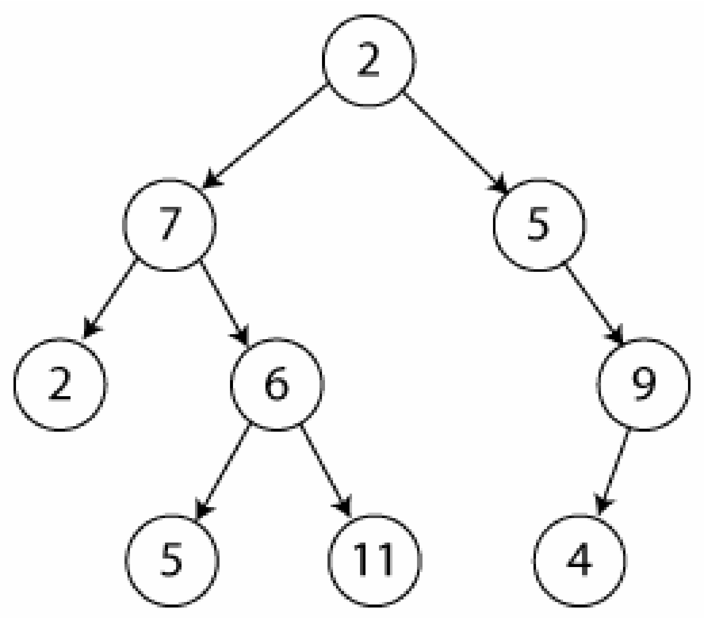
```
;; max-depth : person -> natural
;; (define (max-depth c)
;;   (cond
;;     [(empty? c) ...]
;;     [else ...
;;       ... (max-depth (person-mother c)) ...
;;       ... (max-depth (person-father c)) ...]))
```

Tree Depth

```
;;max-depth : person -> natural
(define (max-depth c)
 (cond
   [(empty? c) 0]
   [else (add1
    (max (max-depth (person-mother c))
         (max-depth (person-father c)))]))
;; Tests  Done!
```

Examples (tests) can help in writing code.

Binary Search Trees



Binary Search Trees

```
(define-struct BTNode (num left right))
;;  A binary-tree (BT) is either
;;  * false, or
;;  * (make-BTNode n l r)
;;  where n is a number, l and r are BTs.

;;  A binary-tree bt is is ordered iff either
;;  * bt is empty, or
;;  * bt has the form (make-BTNode n l r) where
;; Invariants:
;;  1. Numbers in l are less than or equal to n
;;  2. Numbers in r are greater than n

;; A BST is a binary-tree abt that is ordered.
```