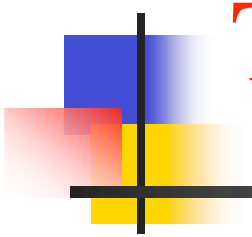
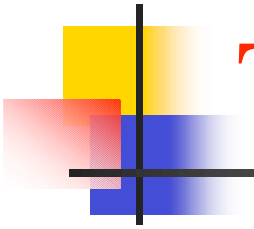


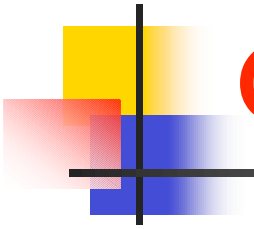
Trees

Corky Cartwright
Department of Computer Science
Rice University



Today's Goals

- Loose Ends
  - Catching mistakes and raising errors
  - `and` & `or` operations
- Trees
  - Significantly more expressive type
  - "Lists with many tails"
- Examples:
  - Family tree
  - Binary search tree

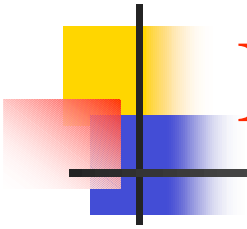


Checked Contracts

- If we use type recognizers for every clause in a template, we can easily add an error reporting clause so that ours functions that check the input type (contract in HTDP terminology) for the primary argument. Example:

```
(define (len aloa)
  (cond [(empty? aloa) 0]
        [(cons? aloa) (add1 (len (rest aloa)))]
        [else (error "length: expected <list>; given" aloa)]))
```

- Questions:
  - Is error reporting a good idea.
  - Should error behavior be documented?
- Answers to question are surprising subtle and lacking in consensus. In the case above, it is probably a good idea but it is generally not done. (DrScheme libraries perform these checks.)

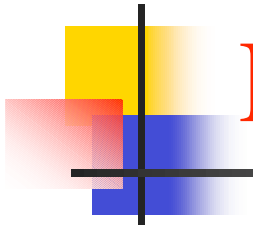


Error Reporting

- To report an error in Scheme invoke:

  ```
  (error msg v1 … vn)
  ```

  where `msg` is a string enclosed in quotation marks and `v1`, …, `vn` are arbitrary Scheme values.
- Error checking should be done:
  - only where the overhead is tolerable;
  - only in basic operations like data constructors when static type checking does not already enforce the contract
- Introductory texts grossly over-promote error checking. Excessive error-checking makes code ponderous.
- Error checking should not be included in a contract (HTDP: purpose statement) unless the client code can depend on it and use it (by "catching" the error). Java.



Reductions for Errors

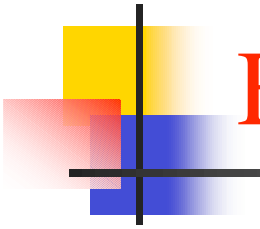
- First, note how errors work for functions you already know. In any context

  `(/ 1 0) => /: divide by zero`

  at top level. This is unique among our rules.
- The error construct generalizes this mechanism. /In any context

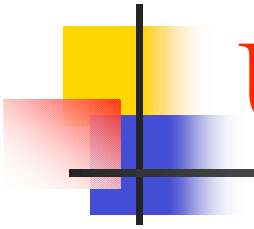
  `(error "Rabbit Late!") => Rabbit: Late!`

  at top level
- Use errors **only** as required by the problem or recipe
- Tip: in HWK, put hand evaluations in comments



Error Reporting in Data Construction

- Constructing ill-formed data is a very serious error. Why? Because the error generally won't be discovered until the data is used, which may not give any clue as to where it was created. Hence, some extra error checking overhead is justified in this situation.
- This issue is the subject of recent and ongoing research. The problem is hard because the requisite legality tests are parametric. Think about the `cons` operations in `append`.
- In our programs, ignore this issue unless I raise it.



Using `and` & `or`

- Scheme and abbreviates a conditional and takes an arbitrary number of arguments (our student dialects require at least 2)

  `(and arg1 ... argn)` abbreviates

```
(cond [(not arg1) false]
      ...
      [else argn])
```

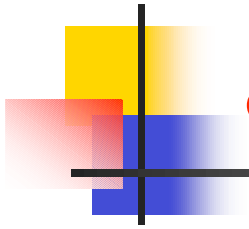
  Hence,

```
(and true true false true (zero? (/ 2 0)) ...)
 => false
```

- This behavior is called "short-circuit" or "non-strict
- What does `or` do?

  `(or arg1 ... argn)` abbreviates

```
(cond [arg1 true]
      ...
      [else argn])
```



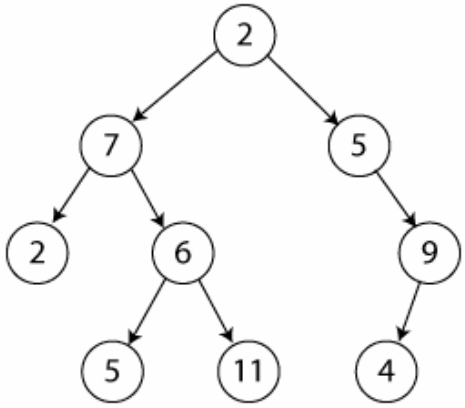
`and` & `or` cont.

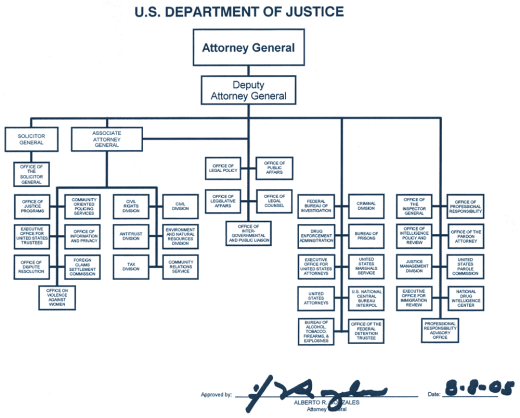
- What are the reduction rules (laws) for `and`?
  - `(and false ... argn) => false`
  - `(and true arg2 ... argn) => (and arg2 ... argn)`
  - `(and v) => v`
- What are the reduction rules (laws) for `or`?
  - `(or true ... argn) => true`
  - `(or false arg2 ... argn) => (or arg2 ... argn)`
  - `(or v) => v`

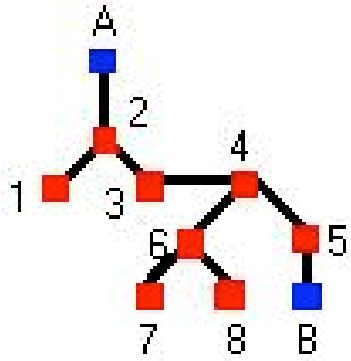
Another Inductive Type: Trees

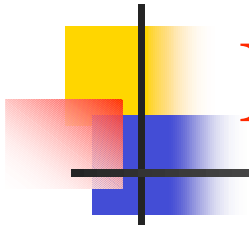
- Labeled trees
- Organizational charts
- Decision trees
- Search trees

and many more!





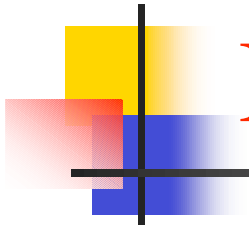




From Lists to Trees

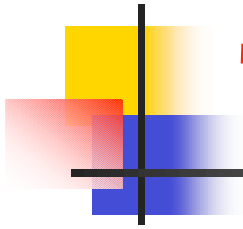
```
; A list-of-symbols (los for short) is
;       empty, or
;       (cons s l)
; where s is a symbol and l is a los

; A person is
;      empty                          // Represents "unknown"
;      (make-person n m d)            // Two self-references
; where n is a symbol, and m and d are persons
(define-struct person (name mom dad))
```



Example Tree

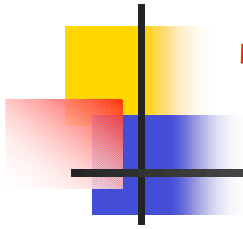
```
(make-person 'Bob
  (make-person 'Jane empty
    (make-person 'Tom
      (make-person 'Cat empty empty) empty))
  (make-person 'Rob empty
    (make-person 'Sue empty
      (make-person 'Ray empty
        (make-person 'Johny empty empty)))))
```



Template for a Tree (1/3)

- We first test for which variety

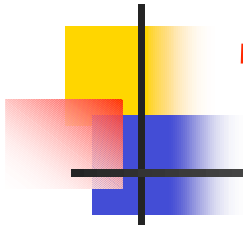
```
; f : person -> …
; (define (f x)
;   (cond
;     [(empty? x) …]
;     [else …
```



Template for a Tree (2/3)

- We make sure that each field is available

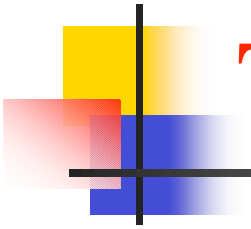
```
; f : person -> …
; (define (f x)
;  (cond
;    [(empty? x) …]
;    [else … (person-name x)
;      …     (person-mom x) …
;      …     (person-dad x) …)
```



Template for a Tree (3/3)

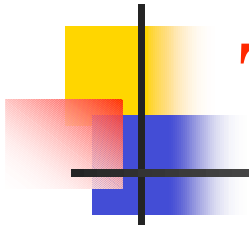
- Recursion in type -> recursion in template

```
; f : person -> …
; (define (f x)
;  (cond
;    [(empty? x) …]
;    [else … (person-name x)
;     … (f (person-mom x))…
;     … (f (person-dad x))…)
```



Tree Depth (in class ex.)

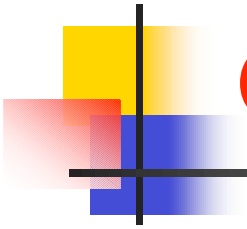
- Consider the following problem
  - "Given a person tree, compute the maximum number of generations for which we know something about this person."
- Contract (or "type") is
  - `person -> natural`
- Examples (from above)
- Template?



Tree Depth

```
max-depth : person -> natural
(define (max-depth x)
 (cond
   [(empty? x) 0]
   [else (+ 1
    (max (max-depth (person-mom x))
         (max-depth (person-dad x))))))
```

Examples were really helpful for filling in the code!



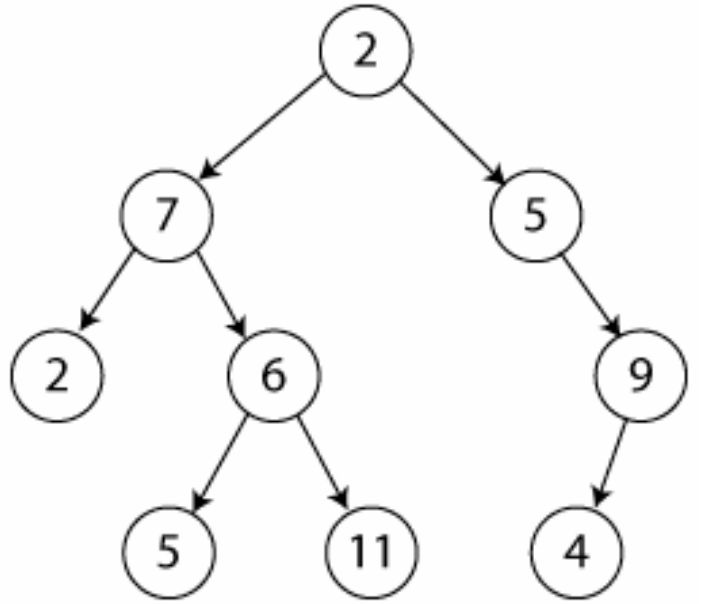
Conventions for type definitions

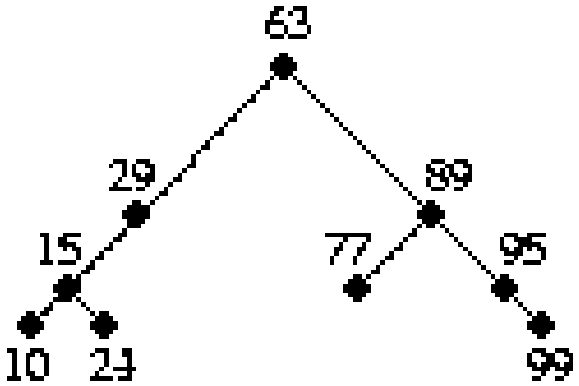
- Our type definitions have a very specific form. Does this definition fit in?

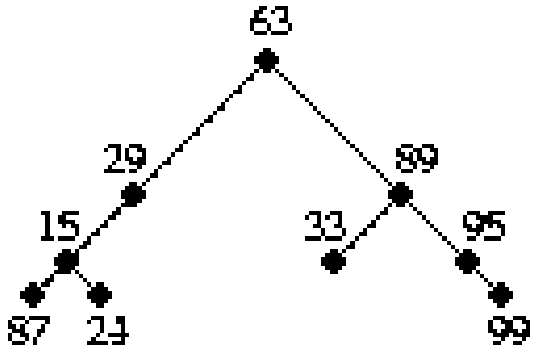
```
; A child node is (make-child f m na da ec)
; where f and m are either
;     empty or
;     child nodes
```

- Our convention:
  - The new type is a variety of values (forms)
  - The type of each field in each struct is a name

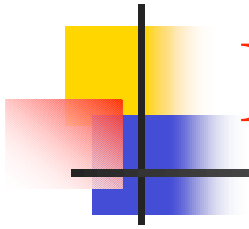
Binary Search Trees





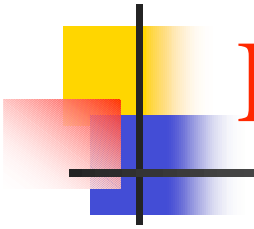


Which tree obeys a simple rule regarding the ordering of node values?



Binary Search Trees

```
;  A binary-search-tree (BST) is either
;      false, or
;      (make-node n l r)
;  where n is a number, l and r are BTs.
;  Invariants:
;       1. Numbers in l are less than or equal to n
;       2. Numbers in r are greater than n
(define-struct node (num left right))
```



For Next Class

- Homework due Monday
- Midterm:
  - Wednesday, in class
  - Chapters 1-13
- No quizzes until midterm