List Abbreviations

- Move to the language:
  - "Beginning Student with List Abbreviations"!

never type `cons` *again!*

BUT: remember what a list is! It's always a `cons` *of something with a list (or the empty list).*

The `list` operation

Instead of this:

```
(cons 'a (cons 'b (cons 'c (cons 'd empty))))
```

you can do this:

```
(list 'a 'b 'c 'd)
```

`first` and `rest`

- `first` and `rest` still work the same way:

```
> (first (list 'a 'b 'c))
'a
> (rest (list 'a 'b 'c))
(list 'b 'c)
> (rest (list 'x))
empty
```

Items are evaluated!

```
(list (+ 1 2) 'b (number? 22 )) =>
(list 3 'b true)


(list 12 (list 'a 'b) (= 1 2)) =>
(list 12 (list 'a 'b) false)
```

nested list!

But wait, it gets better!

- You can use the shorthand notation `'(` instead of `(list`:

```
'( 1 2 3 ) => (list 1 2 3)
```

- BUT: this is actually somewhat different!
  - '(tells scheme that within the list being defined, all left parens `(` should be treated as if they were `(list`
  - `'(1 2 (3 4) 5) =>`

```
(list 1 2 (list 3 4) 5)
```

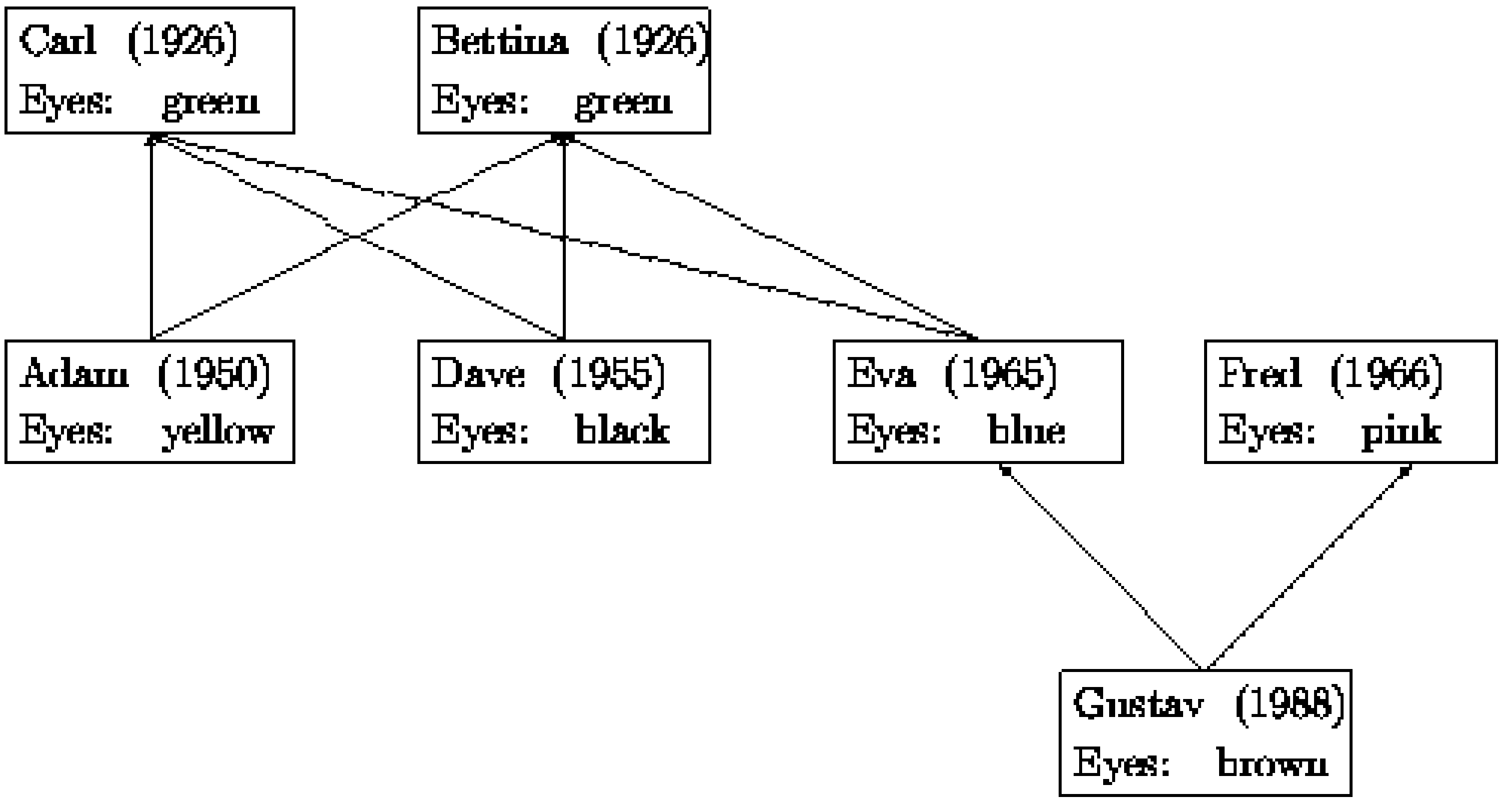
Inside `'(`

- Scheme doesn't evaluate expressions inside a list created with `'(`
- anything that is not a number is treated as a symbol.

```
'(Hi Fred) => (list 'Hi 'Fred)
'(1 (+ 2 3) 4) =>
        (list 1 (list '+ 2 3) 4)
```

Structures contains structures

- Example: family tree



Example Structure

```
(define-struct child
    (father mother name date eyes))
```

Data Definition: A `child` is a structure:

```
(make-child f m na da ec)
```

where `f` and `m` are `child` structures; `na` and `ec` are symbols; and `da` is a number.

Child Data Definition Issue

- The data definition of child means that it is impossible for all child structures to be valid!
  - some structure doesn't have a mother or father field that is a child structure.
- It is also impossible to actually create such a structure:

```
(make-child (make-child (make-child ...
```

Revised Child Data Definition

A child node is:

`(make-child f m na da ec)` where

1. `f` and `m` are either
   1. `empty` or
   2. child nodes;
2. `na` and `ec` are symbols;
3. `da` is a number.

Better Data Definition

A family tree node is either:

1. `empty` or
2. `(make-child f m na da ec)` where

`f` and `m` are family tree nodes, `na` and `ec` are symbols and `da` is a number.

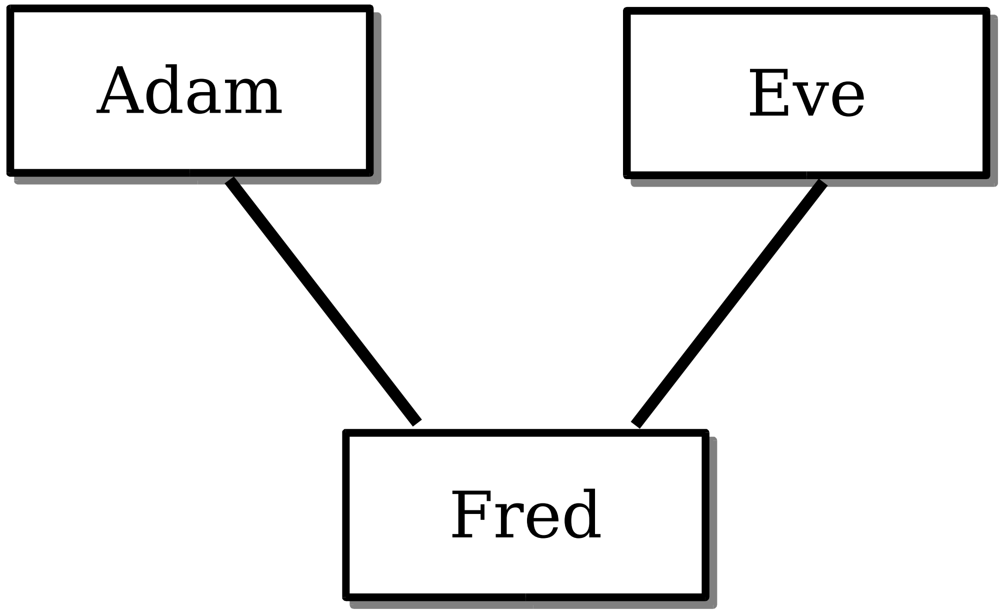
Examples

```
(define adam
  (make-child empty empty 'Adam 0 'blue))

(define eve
  (make-child empty empty 'Eve 1 'rose))

(define fred
  (make-child adam eve 'Fred 1921 'red))
```

Family Tree of Nodes



Processing trees

- General strategy is similar to processing lists.
- Different *base case*.
  - node has no parent.
  - possibly node is empty
- Generally want to process both parent nodes recursively.

Simple Function

```
(define (fathername chld)
  (cond
    [(empty? (child-father chld)) empty]
    [else
     (child-name (child-father chld))]))

(fathername fred) => 'Adam
(fathername adam) => empty
```

More interesting function

- Write a function that will determine whether there are any ancestors that have blue eyes.
- Strategy: under any of the following conditions a child node does have an ancestor with blue eyes:
  - the child has blue eyes.
  - the mother or any of her ancestors have blue eyes.
  - the father or any of his ancestors have blue eyes.

More on blue eyes.

- Under either of the following conditions we should report that there are not blue eyed ancestors.
  - if child node is empty
  - or both of
    - child doesn't have blue eyes.
    - neither parent has ancestor with blue eyes -or- there are no parent(s).

Template

```
(define (blue-eyed-ancestor? chld)
  (cond
    [ (empty? chld) false ]
    [ ...chld has blue eyes... true]
    [ ...no parents...   false ]
    [ ...(blue-eyed-ancestor? mother)...  true]
    [ ...(blue-eyed-ancestor? father)...  true]
    [ else false ]))
```

Refinement

```
[ ...chld has blue eyes... true]
```

becomes:

```
[ (symbol=? (child-eyes chld) 'Blue) true]
```

no parent nodes

```
[ ...no parents...   false ]
```

becomes

```
[ (and (empty? (child-mother chld))
       (empty? (child-father chld))) false ]
```

mother or father?

```
[ ...(blue-eyed-ancestor? mother)..   true]
[ ...(blue-eyed-ancestor? father)..   true]
```

become:

```
[ (blue-eyed-ancestor? (child-mother chld))
   true ]
[ (blue-eyed-ancestor? (child-father chld))
   true ]
```

Better mother or father?

```
[ ...(blue-eyed-ancestor? mother)..   true]
[ ...(blue-eyed-ancestor? father)..   true]
```

become:

```
[ (or
   (blue-eyed-ancestor? (child-mother chld))
   (blue-eyed-ancestor? (child-father chld)))
   true ]
```

Everything

```
(define (blue-eyed-ancestor? chld)
  (cond
    [ (empty? chld) false ]
    [ (symbol=? (child-eyes chld) 'Blue) true]
    [ (and (empty? (child-mother chld))
           (empty? (child-father chld))) false ]
    [ (or  (blue-eyed-ancestor? (child-mother chld))
           (blue-eyed-ancestor? (child-father chld)))
           true ]
    [ else false ]))
```

Too much?

```
(define (blue-eyed-ancestor? chld)
  (cond
    [ (empty? chld) false ]
    [ (symbol=? (child-eyes chld) 'Blue) true]
    [ (and (empty? (child-mother chld))
           (empty? (child-father chld))) false ]
    [ (or  (blue-eyed-ancestor? (child-mother chld))
           (blue-eyed-ancestor? (child-father chld)))
           true ]
    [ else false ]))
```

Is this necessary?

Two Too much?

```
(define (blue-eyed-ancestor? chld)
  (cond
    [ (empty? chld) false ]
    [ (symbol=? (child-eyes chld) 'Blue) true]
    [ (and (empty? (child-mother chld))
           (empty? (child-father chld))) false ]
    [ (or  (blue-eyed-ancestor? (child-mother chld))
           (blue-eyed-ancestor? (child-father chld)))
           true ]
    [ else false ]))
```

How about this?

Simple is better

```
(define (blue-eyed-ancestor? chld)
  (cond
    [ (empty? chld) false ]
    [ else
      (or
        (symbol=? (child-eyes chld) 'Blue)
        (blue-eyed-ancestor? (child-mother chld))
        (blue-eyed-ancestor? (child-father chld)))]))
```

Tree data structures

- Many uses
  - represent relationships
    - family trees – parents.
    - taxonomy, hierarchical classification
    - file systems (folders/directories and files)
  - establish relationships
    - search trees
    - game trees

Binary Search

- Assume an ordered list of things.
- We want to find out if *x* is in the list.
- Start in the middle
  - only need to search one side of the list, as *x* is either smaller than the middle or larger.
- Rinse and repeat (recursively with the side that could contain *x*).

Simple Example of binary search strategy

- I'm thinking of a number between 1 and 100 (inclusive).
- Each time you guess: I will tell you whether you are right, or whether your guess is too high or too low.
- How many guesses will it take you?
  - it is possible to determine an upper bound on the number of necessary guesses.

Scheme search on a list

trying to find out if a number is in an ordered list.

```
'( 1 4 12 86 92 93 94 95)
```

```
(define (list-contains? alist x)
  ... ? ... )
```

consumes a list, produces a boolean

Inefficient strategy

```
(define (list-contains? alist x)
  (cond
    [(empty? alist) false]
    [(= (first alist) x) true]
    [ else
      (list-contains? (rest alist) x)]))
```

Slightly better (on average)

```
(define (list-contains? alist x)
  (cond
    [(empty? alist) false]
    [(= (first alist) x) true]
    [(> (first alist) x) false]
    [ else
      (list-contains? (rest alist) x)]))
```

New Code

Metric: number of comparisons

- The *expensive* operation is:

```
(= (first alist) x)
```

- We want to minimize the number of times this is executed.
- For this *algorithm*, the worst case is that we compare to every item in the list.
  – as the list doubles in size, we should expect this algorithm to take twice as long.

The problem with list-contains?

- It doesn't take full advantage of the ordering of the elements in the list.
- Binary search will require fewer comparisons
  - a lot fewer if the list size is large!
  - it takes advantage of the ordering much better.
  - worst case is now about $\log_2$(size of the list)
  - Consider if the list holds 1,000,000 items...

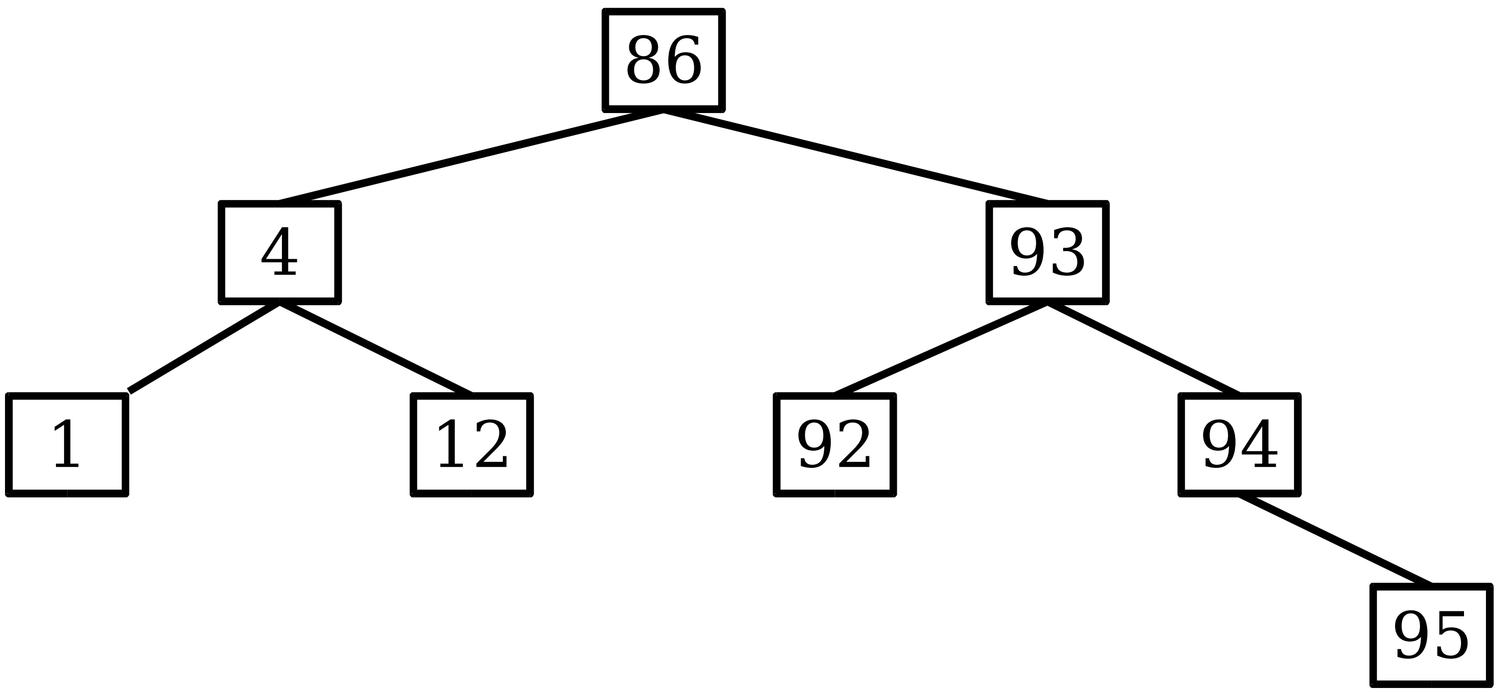
1,000,000 vs. 20 comparisons!

Binary search in scheme?

- If we process a list, we have a problem:
  - it's hard to find the middle of the list.
    - it takes a long time
- We can arrange the items in a *binary search tree*
- Scheme tree based on structures.

Binary Search Tree Example

`'( 1 4 12 86 92 93 94 95)`



Binary Trees

- A Binary tree is a tree in which every node has 0, 1 or 2 *children*.
- Not all binary trees are *binary search trees*.
- A *Binary Search Tree* has the properties:
  - all items in the left child tree have smaller value than the parent
  - all items in the right child tree have values greater than the parent.

Using a binary search tree to search for some value *x*

1) Start at the top of the tree (current node is the top of the tree).
2) If the current node == x done (true).
3) If the current node < x
   - if left child exists - move to the left child
   - if no left child report false.
4) If the current node > x
   - if right child exists - move to the right child
   - if no right child report false.
5) Go back to step 2.

Scheme Example:
binary search tree node

```
(define-struct bsnode
   (num left right))
```

Creating our tree

```
(define stree
  (make-bsnode 86
   (make-bsnode 4
    (make-bsnode 1 empty empty)
    (make-bsnode 12 empty empty))
   (make-bsnode 93
    (make-bsnode 92 empty empty)
    (make-bsnode 94 empty
     (make-bsnode 95 empty empty)))))
```

Scheme tree search

```
(define (tree-contains? nd x)
  (cond
    [(empty? nd) false]
    [...found x... true]
    [... node > x ... (tree-contains? left)...]
    [... node < x ... (tree-contains? right)...]))
```

Checking for node == x

```
[(= found x... true]
```

becomes:

```
[(= (bsnode-num nd) x) true]
```

current node > x ?

we must search to the left of this node.

```
[... node > x ... (tree-contains? left)...]
```

becomes

```
 [(> (bsnode-num nd) x)
       (tree-contains? (bsnode-left nd) x)]
```

current node < x ?

we must search to the right of this node.

```
[... node < x ... (tree-contains? right)...]
```

becomes

```
 [else
     (tree-contains? (bsnode-right nd) x)]
```

Entire binary search function

```
(define (tree-contains? nd x)
  (cond
    [(empty? nd) false]
    [(= (bsnode-num nd) x) true]
    [(> (bsnode-num nd) x)
      (tree-contains? (bsnode-left nd) x)]
    [else
     (tree-contains? (bsnode-right nd) x)]))
```