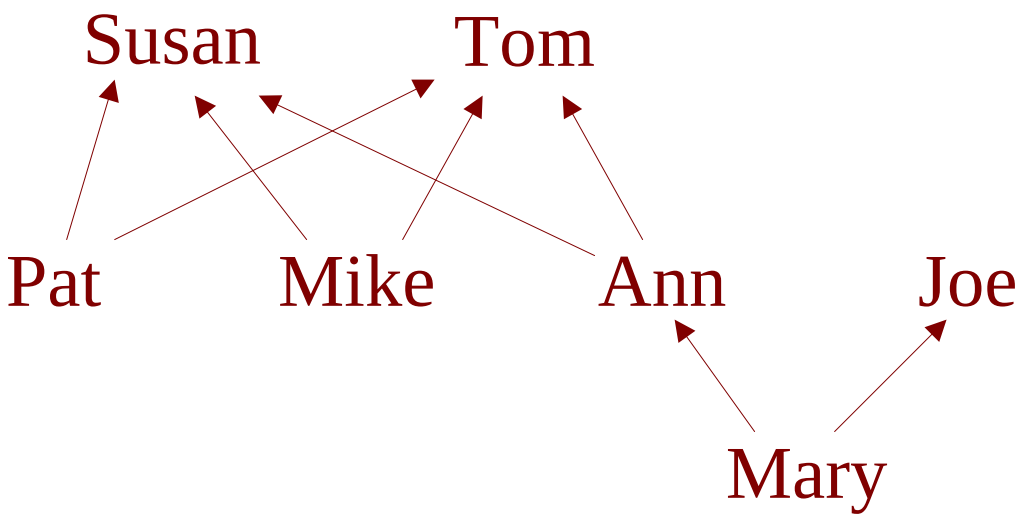
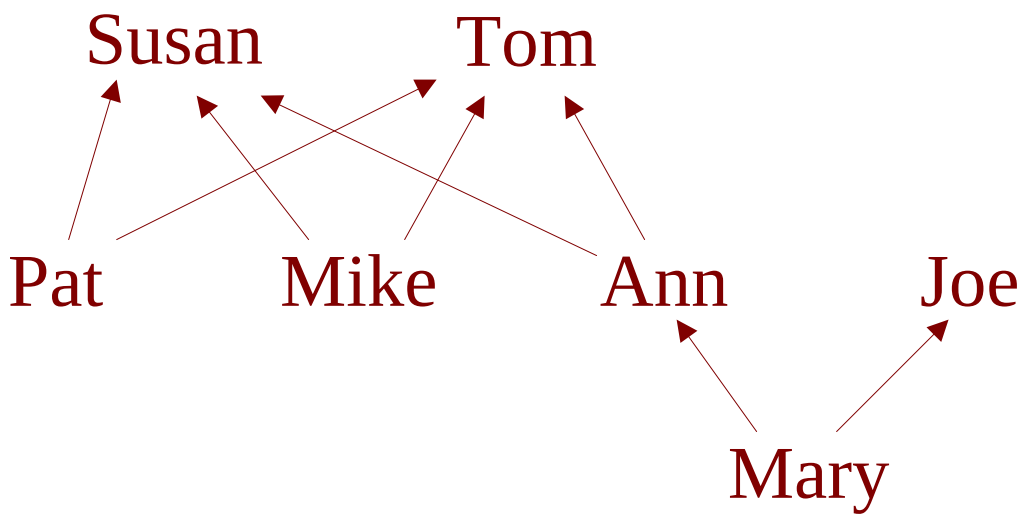
Developing Programs for Family Trees

Consider the following family tree:



Assuming we want to write programs to query who is in the tree or to count how many generations are in the tree, what data model could we use?

Consider the following family tree:

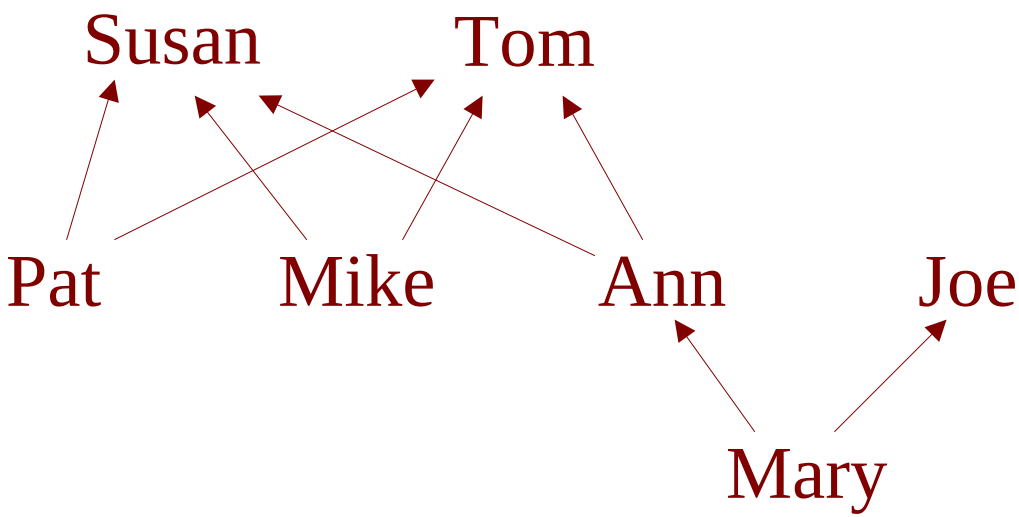


Need to represent:

- Names
- Info about people (name, father, mother)

(Will ignore other info, like birthday, for now)

Consider the following family tree:



Need to represent:

- Names (use symbols)
- Info about people (name, father, mother)

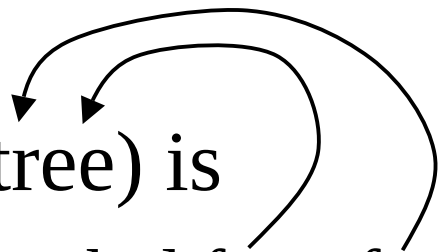
(use structures)

Data Model for Family Trees

How about:

- A family tree (ftree) is

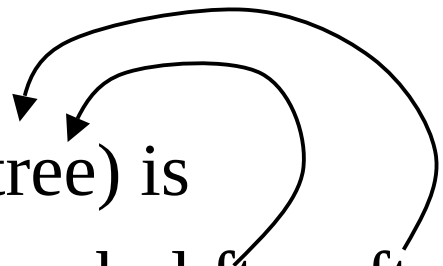
  (make-person symbol ftree ftree)



(define-struct person (name father mother))

[Try making a family tree with this definition]

Making Family Trees



- A family tree (ftree) is

  (make-person symbol ftree ftree)

(define-struct person (name father mother))

(make-person ‘Mary
(make-person ‘Joe ____ ____)
(make-person ‘Ann
(make-person ‘Tom …)
(make-person ‘Susan …)))

What goes here?

Making Family Trees

Follow the data definition! Must use a make-person

- A family tree (ftree) is

  (make-person symbol ftree ftree)

(define-struct person (name father mother))

(make-person ‘Mary
(make-person ‘Joe ____ ____)
(make-person ‘Ann
(make-person ‘Tom …)
(make-person ‘Susan …)))

What goes here?

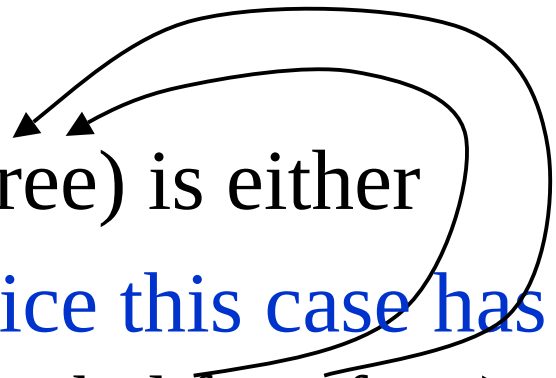
A Broken Family Tree Data Model

- Definition requires each parent to be a whole person, with a father and mother (who are also make-persons …)

- The definition doesn't allow finite trees!

General rule: every recursive data definition needs at least one non-recursive case (ie, a case with no arrows or with a finite chain of arrows)

A New Data Model for Family Trees



- A family tree (ftree) is either
  - ‘unknown [notice this case has no arrows]
  - (make-person symbol ftree ftree)

(define-struct person (name father mother))

[Try making a family tree with this definition]

Sample Family Trees

- ‘Hallie

- (make-person ‘Mary
  ‘unknown
  (make-person ‘Ann ‘unknown ‘unknown))

- (make-person ‘Bernie
  (make-person ‘Fred
  (make-person ‘Bubba
  ‘unknown
  ‘unknown)
  ‘unknown))
  (make-person ‘Lisa ‘unknown ‘unknown))

Programs on Family Trees

Suppose we want to write a program on family trees, but I don't tell you which one …

```
;; ftree-func : ftree → ???
(define (ftree-func aftree)
  …)
```

How much of this program can you write based on the data definition? [Try it]

Template on Family Trees

```
;; ftree-func : ftree → ???
(define (ftree-func aftree)
    …)
```

What kind of data definition does ftree have?

Template on Family Trees

```
;; ftree-func : ftree → ???
(define (ftree-func aftree)
   (cond […        …]
         […        …]))
```

What kind of data definition does ftree have?

one based on (two) cases …

Template on Family Trees

```
;; ftree-func : ftree → ???
(define (ftree-func aftree)
   (cond […        …]
         […        …]))
```

What questions differentiate the cases?

Template on Family Trees

```
;; ftree-func : ftree → ???
(define (ftree-func aftree)
   (cond [(symbol? aftree)        …]
         [(person? aftree)        …]))
```

What questions differentiate the cases?

Template on Family Trees

```
;; ftree-func : ftree → ???
(define (ftree-func aftree)
   (cond [(symbol? aftree)        …]
         [(person? aftree)        …]))
```

What other information is available in each case?

Template on Family Trees

```
;; ftree-func : ftree → ???
(define (ftree-func aftree)
   (cond [(symbol? aftree)        …]
         [(person? aftree)        …]))
```

What other information is available in each case?

none in the symbol? case

Template on Family Trees

```
;; ftree-func : ftree → ???
(define (ftree-func aftree)
   (cond [(symbol? aftree)        …]
         [(person? aftree)
          (person-name aftree) …
          (person-father aftree) …
          (person-mother aftree) …]))
```

What other information is available in each case?

selectors in the person? case

Template on Family Trees

```
;; ftree-func : ftree → ???
(define (ftree-func aftree)
   (cond [(symbol? aftree)        …]
         [(person? aftree)
          (person-name aftree) …
          (person-father aftree) …
          (person-mother aftree) …]))
```

What about the arrows in the data definition?

Template on Family Trees

```
;; ftree-func : ftree → ???
(define (ftree-func aftree)
   (cond [(symbol? aftree)        …]
         [(person? aftree)
          (person-name aftree) …
          (ftree-func (person-father aftree))…
          (ftree-func (person-mother aftree)) …]))
```

What about the arrows in the data definition?

add recursive calls

Template on Family Trees

```
;; ftree-func : ftree → ???
(define (ftree-func aftree)
   (cond [(symbol? aftree)        …]
         [(person? aftree)
          (person-name aftree) …
          (ftree-func (person-father aftree))…
          (ftree-func (person-mother aftree)) …]))
```

This is the full template for programs over family trees

Practice Problems on Family Trees

- ;; count-generations : ftree → number
  ;; return the number of generations in a tree

- ;; in-family? : ftree name → boolean
  ;; determines whether name appears in tree

[try each in turn]

count-generations : Solution

```
;; count-gen : ftree → number
;; counts generations in a family tree
(define (count-gen aftree)
   (cond [(symbol? aftree) 0]
         [(person? aftree)
          (+ 1
             (max  (count-gen (person-father aftree))
                   (count-gen
                      (person-mother aftree))))]))
```

[the blue text is what we added to the template]

in-family? : Solution

```
;; in-family? : ftree name → boolean
;; determines whether name appears in tree
(define (in-family? aftree aname)
   (cond [(symbol? aftree) false]
         [(person? aftree)
          (or (symbol=? (person-name aftree) aname)
              (in-family? (person-father aftree) aname)
              (in-family? (person-mother aftree) aname))]))
```

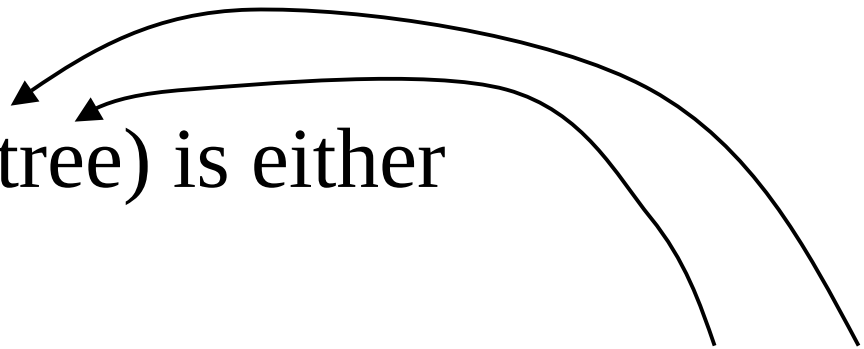
Augmenting the Model

Programmers often augment their initial data models as the problem evolves.

We want to augment our family tree model with information on birth-year and eye-color

How would you change the model?

Revised Model, Version 1

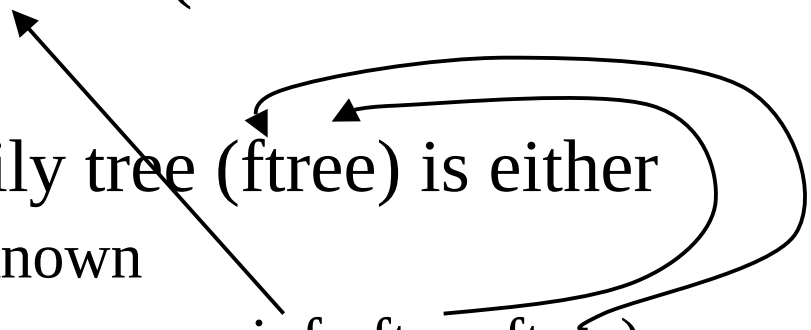


- A family tree (ftree) is either
  - ‘unknown
  - (make-person name number symbol ftree ftree)

(define-struct person (name year eye father mother))

Revised Model, Version 2

- An info is a (make-info name number symbol)
- A family tree (ftree) is either
  - ‘unknown
  - (make-person info ftree ftree)



(define-struct person (data father mother))
(define-struct info (name year eye))

Which Model is Better?

- Model 1 is a little simpler, because it has fewer data definitions (and fewer arrows)

- Model 2 is more flexible, because we can add new info about a person without changing the data definition or template for people

Model 2 is probably a better choice in the long run

[develop a template for model 2]

Template on Revised Family Trees

```
;; ftree-func : ftree → ???
(define (ftree-func aftree)
   (cond [(symbol? aftree)  … ]
         [(person? aftree)
          (info-func (person-data aftree)) …
          (ftree-func (person-father aftree))…
          (ftree-func (person-mother aftree)) …]))
```

```
;; info-func : info → ???
(define (info-func an-info)
   (info-name an-info) …
   (info-year an-info) …
   (info-eye an-info) )
```

Notice we now have two template functions, because we have two complex data definitions

Template on Revised Family Trees

```
;; ftree-func : ftree → ???
(define (ftree-func aftree)
    (cond [(symbol? aftree)  … ]
          [(person? aftree)
           (info-func (person-data aftree)) …
           (ftree-func (person-father aftree))…
           (ftree-func (person-mother aftree)) …]))
```

```
;; info-func : info → ???
(define (info-func an-info)
    (info-name an-info) …
    (info-year an-info) …
    (info-eye an-info) )
```

Notice that the recursive calls match the arrows in the data definition! (3 arrows, 3 calls)

Templates become truly useful (even invaluable) when data definitions get long or refer to each other (ie, have many arrows crossing between them)

We expect you to use them.

Practice Problems on Family Trees 2

- ;; count-blue-eyed : ftree → number
  ;; return number of blue-eyed people in tree

- ;; has-old-and-blue? : ftree number → boolean
  ;; determines whether tree contains a blue-eyed
  ;; person born before given year

- ;; gather-green-eyed : ftree → list[name]
  ;; construct list of names of green-eyed people

[try each in turn]

has-old-and-blue? : Solution

```
;; has-old-and-blue? : ftree num → boolean
(define (has-old-and-blue? aftree yr)
   (cond [(symbol? aftree)  false ]
         [(person? aftree)
          (or (old-and-blue? (person-data aftree) yr)
              (has-old-and-blue? (person-father aftree) yr)
              (has-old-and-blue? (person-mother aftree) yr)))]))

;; old-and-blue? : info number → boolean
;; true if person has blue eyes and was born before given year
(define (old-and-blue? an-info born-before)
   (and (< (info-year an-info) born-before)
        (symbol=? ‘blue (info-eye an-info)))))
```

Descendant Family Trees

- Current model is *ancestor*-based : each person refers to her parents.
- Hard to access information about someone's children
- Let's create a new model in which parents refer to their children instead of the other way around

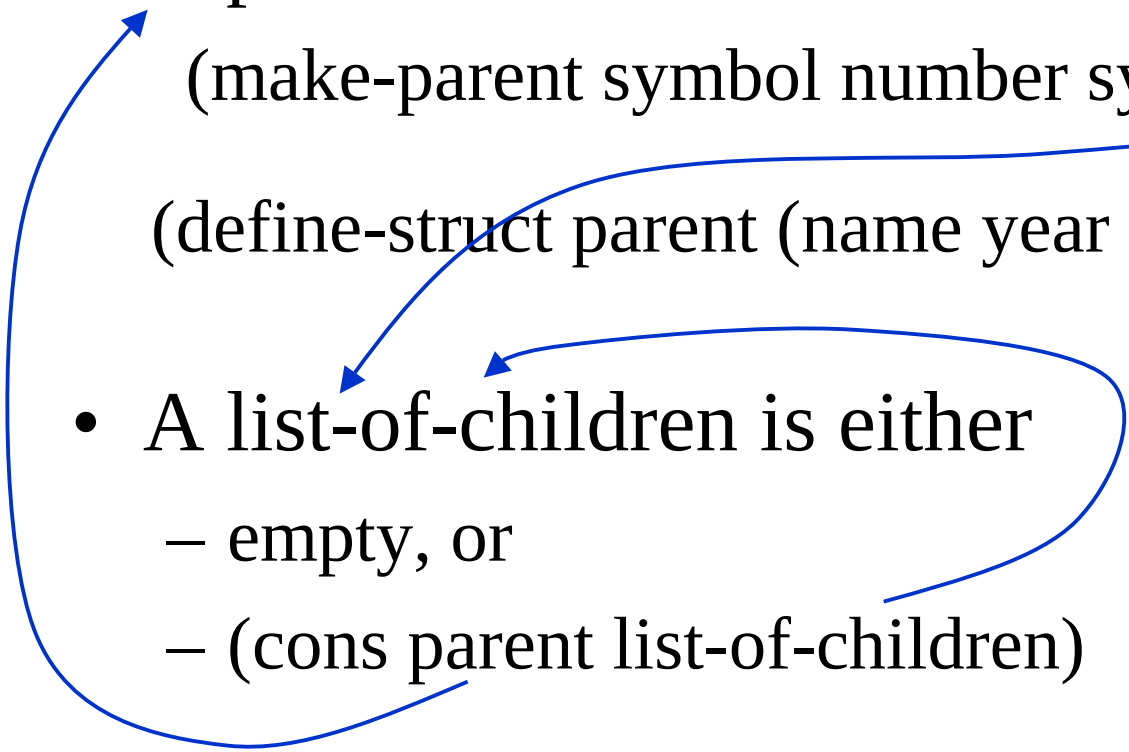
Descendant Family Trees: Data Defn

- A parent is a structure

  (make-parent symbol number symbol list-of-children)

  (define-struct parent (name year eye children))

- A list-of-children is either
  - empty, or
  - (cons parent list-of-children)



[where do we need arrows?]

Descendant Family Trees: Data Defn

- A parent is a structure

  (make-parent symbol number symbol list-of-children)

  (define-struct parent (name year eye children))

- A list-of-children is either
  - empty, or
  - (cons parent list-of-children)

[try writing examples from this data defn]

Descendant Family Trees: Examples

- (define Marypar (make-parent ‘Mary 1975 ‘blue empty))

- (make-parent
  ‘Susan
  1925
  ‘green
  (cons (make-parent
  ‘Ann 1943 ‘blue (cons Marypar empty))
  (cons (make-parent ‘Pat 1949 empty)
  empty)))

Descendant Family Trees: Data Defn

- A parent is a structure

  (make-parent symbol number symbol list-of-children)

  (define-struct parent (name year eye children))

- A list-of-children is either
  - empty, or
  - (cons parent list-of-children)

[try writing the template for this data defn]

Descendant Family Trees : Template

```
;; pfunc : parent → ???
(define (p-func a-parent)
    (parent-name a-parent) ...
    (parent-year a-parent) ...
    (parent-eye-color a-parent) ...
    (loc-func (parent-children a-parent)) ... )

;; loc-func : list-of-children → ???
(define (loc-func a-loc)
    (cond [(empty? a-loc) ...]
          [(cons? a-loc) ...
              (p-func (first a-loc)) ...
              (loc-func (rest a-loc)) ... ]))
```

Again, one call to a template function per arrow in the data defn

Practice on Desc. Family Trees

- ;; count-blue-eyed : parent → number
  ;; return number of blue-eyed desc from parent

- ;; has-old-and-blue? : ftree number → boolean
  ;; determines whether tree contains a blue-eyed
  ;; person born before given year

[try each in turn]

count-blue-eyed : Solution 1

```
;; count-blue-eyed : parent → number
(define (count-blue-eyed a-parent)
   (cond [(symbol=? 'blue (parent-eye-color a-parent))
          (+ 1 (count-blue-kids (parent-children a-parent)))]
         [else (count-blue-kids (parent-children a-parent))]))

;; count-blue-kids : list-of-children → number
(define (count-blue-kids a-loc)
   (cond [(empty? a-loc) 0]
         [(cons? a-loc)
          (+  (count-blue-eyed (first a-loc))
              (count-blue-kids (rest a-loc)))]))
```

count-blue-eyed : Solution 2

```
;; count-blue-eyed : parent → number
(define (count-blue-eyed a-parent)
   (cond [(symbol=? 'blue (parent-eye-color a-parent))
          (+ 1 (count-blue-kids (parent-children a-parent)))]
         [else (count-blue-kids (parent-children a-parent))]))

;; count-blue-kids : list-of-children → number
(define (count-blue-kids a-loc)
   (foldr 0 (lambda (kid result-rest)
              (+ (count-blue-eyed kid)
                 (count-blue-kids (rest a-loc))))
            a-loc))
```

Why Did We Create All These Variations on Models?

- This is how real world program development works.
  - You develop a simple model,
  - figure out how to solve your problem there,
  - then augment the model with new features
  - or redesign the model if necessary.

  Many programs on the simple models can be reused with just a little modification. And you get to think out your problem in stages, rather than all at once.

- We will be doing this model-refinement routine on languages later in the course.

Summary: where are we in the course?

We are done learning Scheme. We have covered how to write programs using

- conditionals
- structures
- lists
- trees

These, plus data definitions and templates, give you all of the Scheme programming tools that you need for the course.

Next week, we start studying programming languages.