More Data Structures

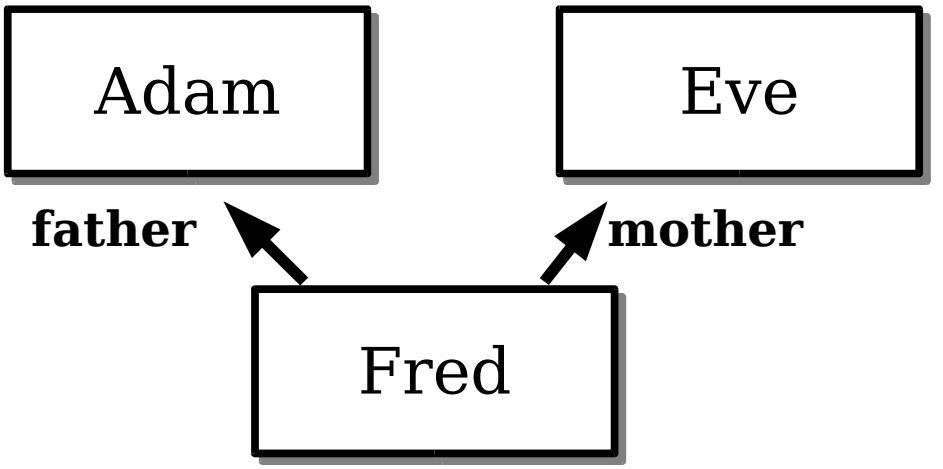
- Lists of lists
  - problem set today - permutations.
- Lists of structures
  - List based animation from quiz
  - alternative representation for family tree.

Family Tree revisited

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.

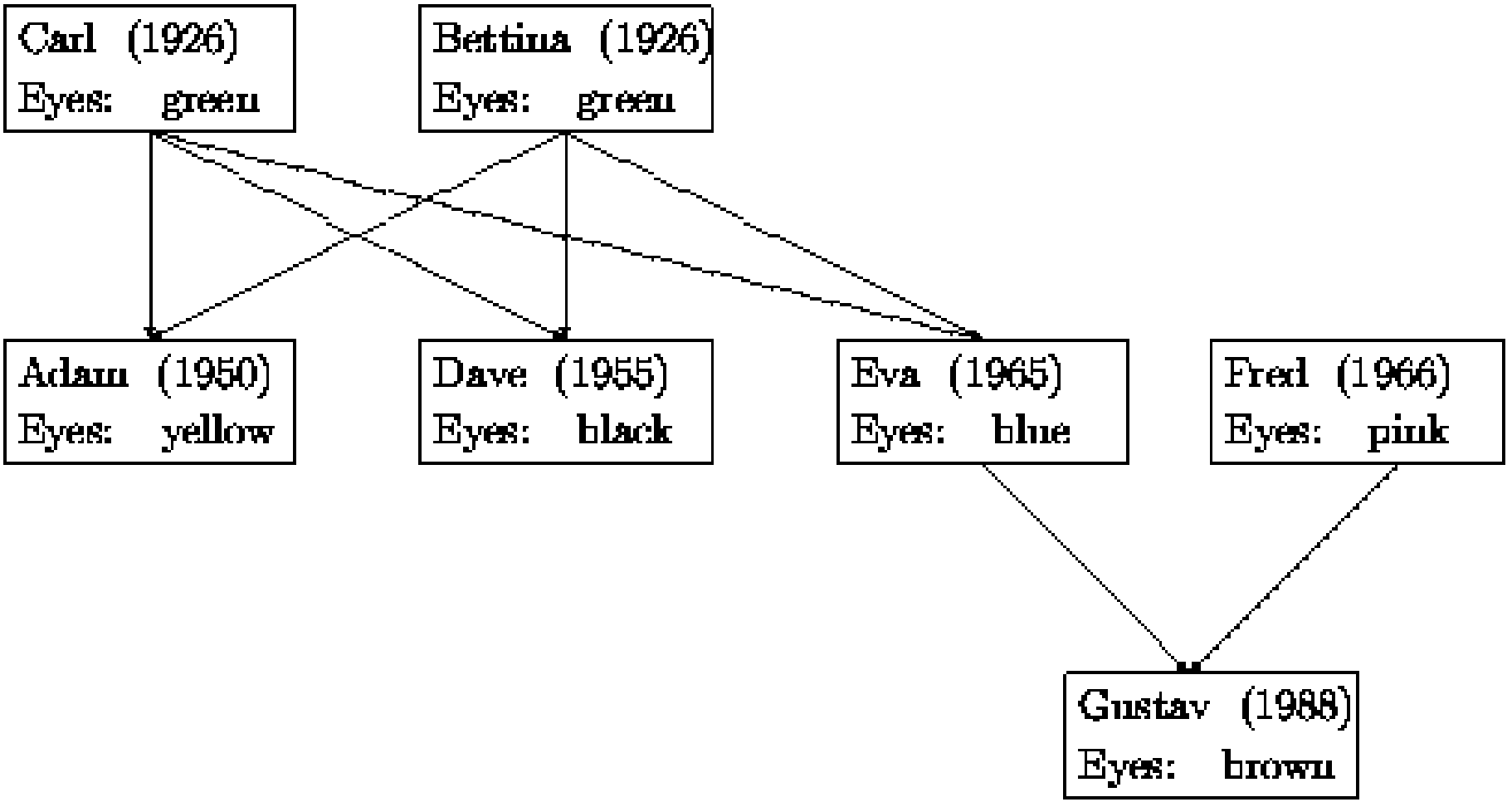


Alternative model

- parent structure instead of child structure.
- each parent can have many children
  - need a list of children.

```
(define-struct parent
        (children name date eyes))
```

Revised Family Tree



An Issue

- With new approach, if we have a node: it is not possible to determine the mother (or the father).
  - it is possible to determine who is the parent of a node.

Data Definitions

- A *parent* is a structure:

```
(make-parent loc n d e)
```

where `loc` is a *list of children*, `n` and `e` are symbols and `d` is a number.

- A *list of children* is either:
  - `empty` or
  - `(cons p loc)` where `p` is a parent and `loc` is a list of children.

Mutually Referential Data Definitions

- In general, we try to avoid using something that has not been defined yet when constructing a data definition.
- In some cases it cannot be avoided (like in this case).
- The definition of *parent* depends on the definition of *list of children* which depends on the definition of *parent.*

Example family tree

```
(define fred
  (make-parent empty 'Fred 1921 'red))

(define adam
  (make-parent (list fred)'Adam 0 'blue))

(define eve
  (make-parent (list fred) 'Eve 1 'rose))
```

Processing the new family tree.

- functions will now start with a parent and traverse the children.

- We can consider writing a function to determine whether a parent has any descendant that has blue eyes.

blue-eyed-descendant?

- base case – does this node have blue eyes?
- recursion: does anyone in the list of children have blue eyes?
  - we can stop as soon as we find a "yes"

```
(define (blue-eyed-descendant? p)
  (cond
    [(symbol=? (parent-eyes p) 'Blue) true]
    [ else ... check the children of p ...]
```

`blue-eyed-children?`

- We need a function that can traverse a *list of children* and return true if it finds a child that has blue eyes.
  - or if the child has a child that has blue eyes.
    - or if the child has a child that has a child that has blue eyes.
      - or if the child has a child that has a child that has a child ...
  - for each child on the list we need to call `blue-eyed-descendant!`
    - hey! this sort-of matches our data definition!

blue-eyed-children?

```
(define (blue-eyed-children? aloc)
  (cond
    [(empty? aloc) false]
    [else
      (cond
        [(blue-eyed-descendant?
                (first aloc)) true]
        [else (blue-eyed-children?
                (rest aloc))]))]))
```

Alternative `blue-eyed-children?`

```
(define (blue-eyed-children? aloc)
  (cond
    [(empty? aloc) false]
    [else
      (or
        (blue-eyed-descendant? (first aloc))
        (blue-eyed-children? (rest aloc)))]))
```

`or` instead of nested `cond`

Finishing `blue-eyed-descendant?`

```
(define (blue-eyed-descendant? p)
  (cond
    [(symbol=? (parent-eyes p) 'blue) true]
    [else
       (blue-eyed-children?
           (parent-children p))]))
```

could use an `or` here instead of `cond`

Important Issue

- The structure of the code matches the structure of the data definition.
- Creating a (formal) data definition is useful
  - in many cases necessary.
- Quiz 6 will involve creating data definitions!
  - and functions that operate on the data.

Exercises to think about

- How would you describe a data structure that could represent a file system?
  - hierarchy of folders and files.
  - Chapter 16 examines this problem.
- How would you describe a data structure that could represent a scheme program?
- How about course at RPI?
  - instructor(s), students, textbooks, TAs, grades, etc....