Another Function Decomposition Example
(which might use anonymous functions)

- Construct a function that will tell whether
  a directed graph, represented as a list of
  arcs, is acyclic.

```
cyclic: a -- b -- c
acyclic: a -- c
```

“Pruning” Method

- Rosalind B. Marimont
  A new method of checking the consistency of precedence matrices
  Journal of the ACM 6, 164-171, 1959

- Pruning away any arcs that point to leaves does not
  change the cyclic/acyclic nature of the graph.
- Pruning such arcs may produce additional leaves.
- Prune until no further pruning is possible:
  - If the result is empty, the original graph was acyclic.
  - If not, it was cyclic.

Examples of Pruning
(leaves shown in green)

```
example 1:
```

```
example 2:
```

Pruning with Graphs as Lists

- Example 1:
  - [{a, b}, {a, c}, {b, d}, {c, d}, {c, e}, {e, a}]
  - [{a, b}, {a, c}, {e, a}]
  - [{a, c}, {e, a}]

- Example 2:
  - [{a, b}, {a, c}, {b, d}, {c, d}, {c, e}]
  - [{a, b}]
  - []

Note

- We are assuming that every node in the graph is
  on one or the other end of an arc, i.e. there are no
  isolated nodes, as in the graph below.
- Otherwise, we’d have to represent the graph with
  two lists: one of nodes and one of arcs.

```
graph
```

Functional Code

- Basic idea:
  - As long as there is a leaf:
    Remove leaves and their attached arcs
- Translation:
  - isAcyclic(Graph) =
    null(iterate(removeLeaves, hasLeaf, Graph));
hasLeaf

- A Graph has a leaf iff isLeaf is true for one of its nodes.
- hasLeaf(Graph) = 
  \( \text{some}(\text{Node} \Rightarrow \text{isLeaf}(	ext{Node}, \text{Graph}), \text{nodes}(\text{Graph})); \)

true when Node is a leaf of this Graph

isLeaf

- A node is a leaf if it is not the first of any arc in the graph.
- isLeaf(Node, Graph) = 
  \( \text{\neg member}(\text{Node}, \text{map}(\text{first}, \text{Graph})); \)

not boolean, is true for only one element of second arg

nodes(Graph)

- nodes(Graph) = 
  \( \text{remove_duplicates}(\text{append}(\text{map}(\text{first}, \text{Graph}), \text{map}(\text{second}, \text{Graph}))); \)

all nodes of Graph that begin some arc

remove_leaves

- To remove the leaves:
  remove any arc that points to a leaf
- removeLeaves(Graph) = 
  \( \text{drop}(\text{Arc} \Rightarrow \text{isLeaf}(	ext{second}(\text{Arc}), \text{Graph}), \text{Graph}); \)

the node to which Arc points

iterate

- iterate(action, continue, State) = 
  continue(State) ?
  iterate(action, continue, action(State))
  : State;

conditional expression (as in C++, Java)
\( P \Rightarrow A : B \)
means if \( P \) is true then the value of the expression is \( A \);
otherwise it is \( B \).
What's “Low-Level” About This?

- "low-level" refers to the construction of functions by explicitly composing and decomposing lists.
- Previously we used higher-order functions to do most of the non-trivial work in a functional decomposition.
- Now we are going to use pattern matching rules, recursion, etc.

Fundamental List Dichotomy

- A list is either:
  - empty, [ ] or
  - non-empty, in which case it has both a first and rest.
- Most list definitions deal with these cases separately.
- Definitions are typically a form of inductive definition, in which [ ] is the basis.

List Decomposition Notation

- When a list is non-empty, it has a first element and the rest of the elements form a list.
- The general form of a non-empty list will be represented:
  
  \[ \{ F \mid R \} \]

  Here F is a variable represents the first element, and R is a variable representing the rest of the elements (R has a list as its value, even though brackets aren't around R).

List Decomposition Example

- Consider a defining equation:
  
  \[ \{ F \mid R \} = [1, 2, 3, 4] \]

  F is a variable represents the first element, so:
  
  \[ F = 1 \]

  R is a variable representing the rest of the elements, so:
  
  \[ R = [2, 3, 4] \]

List Decomposition Clarified

- A defining equation:
  
  \[ \{ F \mid R \} = \text{some list} \]

can only be valid when the RHS list is non-empty.

  Thus
  
  \[ \{ F \mid R \} = [ ] \]
can never be a valid equation.

Defining Functions by Rules

- Suppose we want to define a function taking an arbitrary list as an argument.
- It is sufficient to:
  
  - define the function on the empty list, and
  
    - define the function on a general non-empty list.
Define the function `halve_all`, which divides every element in a list by 2.

- `halve_all([]) => [];`
- `halve_all([F | R]) => [F/2 | halve_all(R)];`

This can be read:
- "halving all of the empty list is the empty list."
- "halving all of a non-empty list is half of the first element followed by halving all of the rest."

**Computed by "Rewriting"**

- `halve_all([2, 4, 6]) ➨ [1 | halve_all([4, 6])]
  ➨ [1, 2 | halve_all([6])]
  ➨ [1, 2, 3 | halve_all([])]
  ➨ [1, 2, 3]`

**Extended Notation for Greater Readability**

- The first so-many, rather than just the first, element, can be shown separated by commas:
  - `[a, b, c, d | R]` means a list with at least 4 elements, `a, b, c, d`, followed by the elements in list `R` (which could be empty).

  In the extended notation:
  - `halve_all([2, 4, 6])` ➨
  - `[1 | halve_all([4, 6])]
  - `[1, 2 | halve_all([6])]
  - `[1, 2, 3 | halve_all([])]`
  - `[1, 2, 3]`

**A Way of Remembering**

- The combination `|[ .. ]` inside a list "melts away" into `[..]` unless `..` is empty, then it just melts away

  Examples:
  - `[1 | [2, 3, 4]]` ➨ `[1, 2, 3, 4]`
  - `[1, 2 | [3, 4]]` ➨ `[1, 2, 3, 4]`
  - `[1, 2, 3 | [4]]` ➨ `[1, 2, 3, 4]`
  - `[1, 2, 3, 4 | []]` ➨ `[1, 2, 3, 4]`

**Alternate**

- Of course, we could have just used `map` in this particular case:
  - `halve(A) = A/2;`
  - `halve_all(X) = map(halve, X);`

- Use higher-order functions such as `map` when possible; resort to lower-order ones when you think you need to.

- Higher-order functions can often tell the story more succinctly.

**Example**

- Define the function `member` which tests whether the first argument is an element of the list in the second argument.

  ```plaintext
  member(X, []) => 0;
  member(X, [F | R]) =>
  \( X == F ? 1 : member(X, R); \)
  ```

  Conditional expression (as in C++, Java)
Alternate

Instead of using a conditional expression, use a third rule with pattern matching:

- member(X, [ ]) => 0;
- member(X, [X| R]) => 1;
- member(X, [F| R]) => member(X, R)

The rule used is always the first (from top to bottom) applicable one.

Rule Matching

Consider evaluating

- member(3, [1, 2, 3, 4]) ➨ rule 3 is the first to apply
- member(3, [2, 3, 4]) ➨ rule 3 is the first to apply
- member(3, [3, 4]) ➨ rule 2 is the first to apply
- 1

Second Alternate
(less desirable)

Use a conditional guard:

- member(X, [ ]) => 0;
- member(X, [X| R]) => (X == F) ? 1;
- member(X, [F| R]) => member(X, R);

The condition is tested after any other matching is applied.
If the condition fails, then subsequent rules are tried.

Matching with Two or More List Arguments

Some functions have more than one list argument.
Induction might, or might not, use rules that dichotomize both lists.

Example: List Equality
First Rule

Two lists are equal if they both are empty:
equals([], []) => 1:
List Equality:
Second Rule

- Two lists are equal if they are both non-empty and
  - the first elements of each are the same, and
  - the lists of the rest of the elements of each are equal.

\[ \text{equals}([A | L], [A | M]) \Rightarrow \text{equals}(L, M) \]

List Equality:
Third Rule

- Otherwise, the two lists are not equal:
  \[ \text{equals}(X, Y) \Rightarrow 0; \]

Summary of Equality Rules

1. \[ \text{equals}([], []) \Rightarrow 1; \]
2. \[ \text{equals}([A | L], [A | M]) \Rightarrow \text{equals}(L, M); \]
3. \[ \text{equals}(X, Y) \Rightarrow 0; \]

Example of List Equality

- Revisit our earlier example:
  - Are these lists equal: \([1, 2, 3]\) vs. \([1, 2]\)?
  - Try the rules:
    - \[ \text{equals}([1, 2, 3], [1, 2]) \Rightarrow (\text{rule 2}) \]
    - \[ \text{equals}([2, 3], [2]) \Rightarrow (\text{rule 2}) \]
    - \[ \text{equals}([3], []) \Rightarrow (\text{rule 3}) \]
    - 0
  - i.e. the lists are not equal.

Example: Low-Level Version of Nim

- Recall:
  - \(\text{magic}(L, s) = \) a new list similar to \(L\), except that the first element, \(p\), where \(\text{xor}(p, s) < p\), is replaced with \(\text{xor}(p, s)\).

- Translation to rules:
  - \(\text{magic}([], []) \Rightarrow [];\)
  - \(\text{magic}([p | R], s) \Rightarrow \text{xor}(p, s) < p ? [\text{xor}(p, s) | R] \)
  - \(\text{magic}([p | R], s) \Rightarrow [p | \text{magic}(R, s)];\)

Improving Efficiency

- Recall the \(\text{isAcyclic}\) example.
  - There might be occasion to compute the same thing multiple times, for example,
    \(\text{isLeaf}(\text{Node}, \text{Graph})\) may be called multiple times for a given \(\text{Graph}\):
    - \(\text{isLeaf}(\text{Node}, \text{Graph}) = \) \(\text{some}(\text{Node}) = \text{isLeaf}(\text{Node}, \text{Graph}), \text{nodes}(\text{Graph})\)\)
  - Each time \(\text{isLeaf}\) is called, \(\text{map}([\text{first}, \text{Graph}])\) is recomputed:
    - \(\text{isLeaf}(\text{Node}, \text{Graph}) = \text{ismember}(\text{Node}, \text{map}([\text{first}, \text{Graph}]))\)
  - It may be better to compute \(\text{map}([\text{first}, \text{Graph}])\) “up front” and pass it to \(\text{isLeaf}\)
Improving Efficiency

- Computing up front means an extra argument to isLeaf, which may clutter the meaning of a given function:
- Below we “promote” map(first, Graph) out of isLeaf

  `hasLeaf(Graph) = some((Node) => isLeaf(Node, map(first, Graph)), nodes(Graph));`

- However, it is still may be called once for each node.
- In order to avoid recomputation, we need to promote it out of the call to `some`.
- This can be done with a local equation, or “equational guard”:

  `hasLeaf(Graph) = Firsts = map(first, Graph), some((Node) => isLeaf(Node, Firsts), nodes(Graph));`

  *equational guard*

- If we prefer not to use an equational guard, we can pass Firsts as an argument to `hasLeaf`:

  `hasLeaf2(Graph, Firsts) = some((Node) => isLeaf(Node, Firsts), nodes(Graph));`

- This will necessitate introduction of a new definition for the original 1-argument `hasLeaf`:

  `hasLeaf(Graph) = hasLeaf2(Graph, map(first, Graph));`

- Alas, we overlooked at least one little detail:
- isLeaf is used in removeLeaves as well as in hasLeaf, so we’ll similarly have to adjust its use there.

  `removeLeaves(Graph) = removeLeaves2(Graph, map(first, Graph));`

  `removeLeaves2(Graph, Firsts) = drop((Arc) => isLeaf(second(Arc), Firsts)), Graph);`

- There is still one obvious inefficiency:
  - `map(first, Graph)` is computed in both `hasLeaf` and `removeLeaves`. We’d prefer to compute it only once.
  - The low-level definition (using recursion instead of `iterate`) then might be:

    `isAcyclic(Graph) = Firsts = map(first, Graph), hasLeaf2(Graph, Firsts) ? isAcyclic(removeLeaves2(Graph, Firsts)) : null(Graph));`

- Alternatively, we could construct a different version of `iterate`, but it would seem to be rather special purpose.

Closing Notes on Efficiency

- In previous slides, we used the property of referential transparency of functional languages (that expressions can be substituted for equivalent expressions) to improve efficiency.
- It would not generally be possible to do such substitutions in an imperative language; procedures that have side-effects cannot be substituted so freely.
- Transforming a program in the manner shown may impair its clarity and readability, so it is better to maintain a perhaps less-efficient reference version apart from the “production” version.

Mixed Functional Programming Examples

- Use low-level or high-level, whatever fits best
- Maybe start with low-level, and the use high-level retrospectively
- Radix conversion
- Tail recursion
- Tree searching
Convert Number to Binary

- Example:
  - toBinary(37) \(\Rightarrow\) \([1, 0, 0, 1, 0, 1]\)
  \[32 + 0*16 + 0*8 + 4 + 0*2 + 1\]
- First try: use method discussed in class earlier:
  - divide by 2, record remainder, continue with quotient
  - until 0

Rules:
- toBinary1(0) \(\Rightarrow\) \([\ ]\)
- toBinary1(N) \(\Rightarrow\) \([N\%2 \mid toBinary1(N/2)]\)
  
Problems with this definition?

Another try:
- toBinary(N) \(\Rightarrow\) toBinary2(N, \([\ ]\))
- toBinary2(0, Acc) \(\Rightarrow\) Acc;
- toBinary2(N, Acc) \(\Rightarrow\) toBinary2(N/2, \([N\%2 \mid Acc]\))

Why is this definition better?
What is still lacking?

Accumulators and Tail Recursion

- From previous slide:
  - toBinary2(0, Acc) \(\Rightarrow\) Acc;
  - toBinary2(N, Acc) \(\Rightarrow\) toBinary2(N/2, \([N\%2 \mid Acc]\));
  
Acc is called an "accumulator" argument:
  - It "accumulates" the result until the basis case is reached, then "unloads" it.

This type of recursion is called "tail recursion":
  - There is no "cleanup" to be done after the recursive call to toBinary2, and therefore no need to "stack" calls
  - We can effectively "turn over control" to the subordinate call: giving it a form of iteration.

Can similarly convert to any given base.
Can pass the base as an argument.
Can convert back (from numeral list to number).
Exercise

- Construct fromBinary, e.g.
  - fromBinary([1, 0, 0, 1, 0, 1]) => 37
- Considerations:
  - Do we need an accumulator?
  - Can it be done with tail-recursion?
  - Try it and see.

An Approach

- Write iterative pseudo-code, then construct recursive equivalent.
- L = ... list to be converted ...
  - Result = 0;
  - while(L != [])
    - Result = 2*Result + first(L);
    - L = rest(L);
  - ... answer is in Result ...
- Defining fromBinary3(L, Result):
  - fromBinary3([], Result) => Result;
  - fromBinary3([F | R], Result) => fromBinary3(R, 2*Result+F);
  - fromBinary3(L) = fromBinary3(L, 0);
- fromBinary3([1, 0, 0, 1, 0, 1], 0) ➔ fromBinary3([0, 0, 1, 0, 1], 1) ➔ fromBinary3([0, 1, 0, 1], 2) ➔ fromBinary3([1, 0, 1], 9) ➔ fromBinary3([1], 18) ➔ 37

Exercise

- What if the list were least-significant bit first?
- Can you do construct the function?
- Can you construct a tail-recursive implementation?
- Construct the xor function used in the nim move function.
  - This could entail the essence of toBinary and fromBinary in one definition.

Exercises

- Compare "obvious" and tail-recursive forms of:
  - factorial function (fac(n) = 1*2*3*...*n)
  - length function
  - sum of a list
  - reduce
  - reverse

Essential Non-Tail Recursions

- Some functions don’t admit a tail-recursive version (unless reverse is used before or after):
  - Examples:
    - map, keep, drop
    - append

append Elimination

- When maximum efficiency is desired, uses of append should be avoided.
- It is often possible to get rid of append by defining versions of functions with an extra accumulator argument.
- Example:
  - nodes(Graph) = remove_duplicates(append(map(first, Graph), map(second, Graph)));
- Show how to avoid append by generalizing map to take an accumulator.