Function Decomposition

- To solve problems using functions, we typically:
  - Express the problem informally as a function, with input and output.
  - Break down the function as a composition of simpler functions.
  - Repeat this process, until we are using only functions that are built-in.

Function Decomposition Example

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

```
cyclic: a -> b -> c -> d -> e
acyclic: a -> b -> 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:
  - Example 2:

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.
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));

  test for empty list
  iterate first arg.
  as long as
  second arg. true
  remove all
  leaves
  and their
  arcs
  test
  whether
  there is a
  leaf
  starting
  graph

- A Graph has a leaf iff isLeaf is true for one of its nodes.

- hasLeaf(Graph) =
  - some((Node) => isLeaf(Node, Graph), nodes(Graph));

- nodes(Graph) =
  - remove_duplicates(append(map(first, Graph), map(second, Graph)));

- remembering our assumption: 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.

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

  conditional expression (as in C++, Java)
  P ? A : B
  means if P is true then the value of the expression is A; otherwise it is B.
Improving Efficiency

- Recall the isAcyclic example.
- There might be occasion to compute the same thing multiple times, for example
  - isLeaf(Node, Graph) may be called multiple times for a given Graph:
    - hasLeaf(Graph) = some(Node) isLeaf(Node, Graph, nodes(Graph));
  - Each time isLeaf is called, map(first, Graph) is recomputed
    - isLeaf(Node, Graph) = !member(Node, map(first, Graph));
  - It may be better to compute map(first, Graph) “up front” and pass it to 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));
  - hasLeaf(Node, Firsts) = !member(Node, map(first, 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));

If we prefer not to use an equational guard, we can pass Firsts as an argument to hasLeaf:
- Below we “promote” map(first, Graph) out of isLeaf
  - 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));

Improving Efficiency

- 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);

Improving Efficiency

- 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.