Review: Curried Functions

- Given the definition
  \[ \text{scaleBy}(Y) = (X) \Rightarrow Y \times X; \]
  we have
  \[ \text{map}(\text{scaleBy}(5), [1,2,3]) \Rightarrow [5,10,15] \]
  \[ \text{scaleby}(4)(7) \Rightarrow 28 \]
- We could have defined
  \[ \text{scale}(Y, L) = \text{map}(\text{scaleBy}(Y), L); \]

A Useful Example

- An **association list** is a representation of a function (with a finite domain):
  \[ \{ \text{"Jan", 31}, \text{"Feb", 28}, \text{"Mar", 31}, \text{"Apr", 30} \} \]
- But we cannot simply apply this list as a function; we have to use `assoc` instead:
  \[ \text{assoc("Feb", \ldots list above \ldots)} \Rightarrow \{\text{"Feb", 28}\} \]
Making Fun of Association Lists

- We want a function
  \[
  \text{makeFun}(\text{Alist})
  \]
  that converts an association list \text{Alist} to an ordinary function that can be applied, e.g.,

  \[
  f = \text{makeFun}([[\text{"Jan"},31],[\text{"Feb"},28],[\text{"Mar"},31]]);
  f(\text{"Feb"}) \Rightarrow 28
  \]

Defining \text{makeFun}

- How to define \text{makeFun}?

Composing Functions

- The composition of two functions,
  \[
  f: T \to U
  \quad \text{and} \quad
  g: S \to T
  \]
  is the function, call it \text{h} for now, such that
  \[
  h: S \to U
  \quad \text{and for every} \quad x \in S, \quad h(x) = f(g(x))
  \]
- The composition is sometimes written using an operator \text{o}:
  \[
  f \circ g
  \]

Composing Functions

- If anonymous functions are supported, we don’t need the “call it \text{h}” aspect:
  \[
  \text{compose}(f,g) = (x) \Rightarrow f(g(x))
  \]
  or, equivalently,
  \[
  \text{compose}(f,g)(x) = f(g(x))
  \]
Why compose?

• Composing can make code more efficient in some cases

\[
\text{map}(f, \text{map}(g, L)) \Rightarrow \]

Generating Curried Functions

• Define

\[
\text{curry}(f)(X)(Y) = f(X, Y)
\]

• If \( f \) is a two-argument function then \( \text{curry}(f) \)
  is a curried version of this function.
  - Computes the same result as the given \( f \), but \( \text{curry}(f) \)
    expects its argument to be given in separate applications rather than simultaneously.
  - \( \text{curry}(f)(X) \) is like "\( f(X, \cdot) \)" in engineering or mathematics books.

• Example: \( \text{scale}(K, L) = \text{map}((\times)(K), L); \)
  the 2-argument multiply function

Searching a Graph

• Suppose we want to find whether there is a node with a certain property \( P \) reachable from a node in a graph.
• Rather than assume a specific representation such as an arc-list, we’ll use abstraction:
  • Assume we have a function \( \text{targets} \) such that
    \( \text{targets}(\text{Graph}, \text{Node}) \)
    is the list of targets of the node.

Finding a Node in a Graph

• Want to define

\[
\text{findNode}(\text{Graph}, \text{Node}, P)
\]

where \( \text{Node} \) is the starting node, \( P \) is a predicate on nodes.

• Return value:
  - If a node is found, return a list containing just that node.
  - If no such node is found, return the empty list.
Simplifying the Problem

\[
\text{findNode}(\text{Graph}, \text{Node}, \text{P}) \Rightarrow \\
\quad \text{P}((\text{Node}) \ ? \ [\text{Node}]); \\
\text{findNode}(\text{Graph}, \text{Node}, \text{P}) \Rightarrow \\
\quad \text{findFromList}(\text{Graph}, \\
\quad \quad \text{targets}(\text{Graph}, \text{Node}), \\
\quad \quad \text{P});
\]

Starting From a List of Nodes

\[
\text{findFromList}(\text{Graph}, [], \text{P}) \Rightarrow [ ]; \\
\text{findFromList}(\text{Graph}, [\text{Target}|\text{Targets}], \text{P}) \Rightarrow \\
\quad \text{Found} = \text{findNode}(\text{Graph}, \text{Target}, \text{P}), \\
\quad \quad (\text{Found} \neq []) \ ? \\
\quad \quad \text{Found} : \\
\quad \quad \quad \text{findFromList}(\text{Graph}, \text{Targets}, \text{P});
\]

Mutual Recursion

- The relationship between \text{findNode} and \text{findFromList} is that of \textbf{mutual recursion}.
  - \text{findNode} delegates work to \text{findFromList}
  - \text{findFromList} delegates work to \text{findNode}

- This approach seems natural in this problem.

Correctness

- The previous solution will work if the graph is acyclic.
  - If not acyclic, it may work in some cases, and loop infinitely in others.

- So it doesn't \textit{really} work.

- How can we fix this?
Handling the Cyclic Case

- If our graph is finite, infinite looping can only occur when the same node recurs on a path.

- By keeping track of the nodes on the path from the starting point, we can check whether a node recurs.

**findNode revised**

```prolog
findNode(Graph, Node, P) =>
    findNode(Graph, Node, P, []);

findNode(Graph, Node, P, Seen) =>
    P(Node) ? [Node];

findNode(Graph, Node, P, Seen) =>
    member(Node, Seen) ? [];

findNode(Graph, Node, P, Seen) =>
    findFromList(Graph,
                 targets(Graph, Node),
                 P, [Node|Seen]);
```

**findFromList revised**

```prolog
findFromList(Graph, [], P, Seen) => [ ];

findFromList(Graph, [Target|Targets], P, Seen) =>
    Found = findNode(Graph, Target, P, Seen),
    (Found != []) ?
        Found :
        findFromList(Graph, Targets, P,
                     [Target | Seen]);
```

- We include Target in the list above so that we never do a recursive findNode from a node more than once.
- However, we may still search from the same node more than once. Why? Is this preventable?

Varieties of Search

- The version of find presented previously is called depth-first search.

- The other prevalent form of find is breadth-first search.
Search Orders

depth-first:

breadth-first:

Search Orders

depth-first:

breadth-first:

Search Orders

depth-first:

breadth-first:

Search Orders

depth-first:

breadth-first:
Comparative Strengths

- Breadth-first has the advantage of finding the shortest path to a node with the desired property.
- Depth-first is easier to implement and is more space-efficient.