Lower-Level Functional Programming
September 17, 2001

Review

• Last week we talked about coding using some very powerful techniques
  - Particularly, built-in higher-order functions (map, reduce, keep, some)

• But these aren't always enough
  - rex cannot include every useful higher-order function
  - This week: how to implement these sorts of functions ourselves

Conditionals

• The rex expression

\[ C \ ?\ A \ :\ B \]

means exactly the same as it does in Java.
  - Returns the result of A if the expression C is non-zero, and the result of B otherwise.
  - Only one of A and B will be evaluated. (Why?)

Exercise

• Given a list of lists, return the longest one.

\[ \text{longerList}(A, B) = \]

\[ \text{longestList}(L) = \]
Exercise

- Assume we have predicates \( p \) and \( q \).

\[
!p(X) == p(X) \land q(Y) == p(X) \lor q(Y) == p(X) \land q(Y) == p(X) \lor q(Y) ==
\]

Recursion

- We have seen many examples of functions calling other functions.
  - A function is said to be recursive if it may call itself

\[
\text{factorial}(N) =
\begin{cases} 
  1 & \text{if } N \leq 0 \\
  N \times \text{factorial}(N-1) & \text{otherwise}
\end{cases}
\]

- Key idea: we can solve a hard problem by using the solutions to simpler problems!

Tracing a Recursive Computation

\[
\text{factorial}(3) ==
\begin{align*}
3 & \times \text{factorial}(2) == \\
3 & \times (2 \times \text{factorial}(1)) == \\
3 & \times (2 \times (1 \times \text{factorial}(0))) == \\
3 & \times (2 \times (1 \times 1)) == \\
3 & \times (2 \times 1) == \\
3 & \times 2 == \\
6.
\end{align*}
\]

Fundamental List Dichotomy

A list is either:
- empty, or
- non-empty, in which case it has a first element and a list of the remaining elements.
List Decomposition Notation

- When a list is non-empty, it has a first element and the rest of the elements form a list.

- A **non-empty list** can be written in rex as:
  
  \[ F \mid R \]
  
  - Here \( F \) is a variable represents the first element, and \( R \) is a variable representing the rest of the elements.
  - Remember, \( R \) is a list!

List Decomposition Example

- Many ways to write the same list:

  \[
  [1, 2, 3] == \\
  [1 \mid \{2, 3\}] == \\
  [1 \mid 2 \mid \{3\}] == \\
  [1 \mid 2 \mid 3 \mid \{\}]
  \]

More List Notation

- If we want to talk about more than the first element, we can:

  \[
  [1, 2, 3] == \\
  [1 \mid [2, 3]] == \\
  [1, 2 \mid [3]] == \\
  [1, 2, 3 \mid []]
  \]

Patterns

- Generic representations of values.
  - May involve constants, variables, and list notation.

- Examples:

  \[
  L \]
  
  \[
  [F \mid R] \\
  [F, S \mid R] \\
  [F, S, T] \\
  [[F], S] \\
  [[[F], S] \mid R] \\
  [F, S, T \mid R]
  \]


Pattern Matching

- We say that a value matches a pattern if it has the same form as the pattern.
  - That is, if there is a way to give variables in the pattern values so as to equal the value.

- Example: The pattern \([F \mid R]\)
  matches the value \([1, 2, 3, 4]\)
  because we can take \(F = 1\) and \(R = [2, 3, 4]\).

Exercise

- Which of these patterns match which of these lists?

<table>
<thead>
<tr>
<th>Pattern</th>
<th>List</th>
</tr>
</thead>
<tbody>
<tr>
<td>([F \mid R])</td>
<td>([1, 2, 3])</td>
</tr>
<tr>
<td>([F, S \mid R])</td>
<td>([1, [2,3]])</td>
</tr>
<tr>
<td>([F, S, T \mid R])</td>
<td>([1, 2, 3])</td>
</tr>
<tr>
<td>([F], S)</td>
<td>([1, 2, [3]])</td>
</tr>
<tr>
<td>([F, S, T])</td>
<td>([1, 2, [3,4]])</td>
</tr>
<tr>
<td>([F, S, T \mid R])</td>
<td>([1, 2, [3,4]])</td>
</tr>
</tbody>
</table>

Variable Definitions

- As our first use of patterns, we can use pattern matching to define variables!

```haskell
rex > X = 3;
1
rex > [F \mid R] = [1,2,3];
1
rex > X + F;
4
```

More Variable Definitions

```haskell
rex > [F, S, T \mid R] = [1, 2, [3,4]];
1
rex > F + S + length(R);
3
rex > [A, B, C \mid D] = [1, 2];
0
rex > A;
*** warning: unbound symbol A
*** aborting to top-level
```
Equality for Lists

- Two lists are equal if they have the same number of elements, and their elements occur in the same order.
- Equivalently:
  - Two lists are equal if, and only if:
    - They are both empty, or
    - They are both non-empty and the first elements of each are the same, and the lists containing the rest of the elements are equal.
- Note that this check is recursive! It uses the equality check to perform an equality check.

List Equality Formalized

- Let’s re-cast our list equality check as a set of rules, to be applied sequentially whenever the question of equality is asked.

First Equality Rule

- Two lists are equal if they both are empty:
  \[
  \text{equals}(\text{[]}, \text{[]}) \Rightarrow 1;
  \]
- Here \(\Rightarrow\) is read “rewrite as” or “can be replaced with”

Second Equality Rule

- Two lists are equal if they are both non-empty and the first elements of each are the same, and the lists containing the rest of the elements are equal.
  \[
  \text{equals}([A|L], [A|M]) \Rightarrow \text{equals}(L,M);
  \]
Third Equality Rule

- Otherwise, the two lists are not equal:

\[
equals(X, Y) \Rightarrow 0;
\]

(implied by “and only if”)

Example using Equality Rules

- Are the lists \([1, 2, 3]\) and \([1, 2]\) equal?

- Try the rules:

  \[
equals([1, 2, 3], [1, 2]) \Rightarrow \text{(rule 2)}
  
equals([2, 3], [2]) \Rightarrow \text{(rule 2)}
  
equals([3], []) \Rightarrow \text{(rule 3)}
  
  0
  \]

- So (surprise!) the answer is no.

Summary of Equality Rules

\[
equals([], []) \Rightarrow 1;
equals([A|L],[A|M]) \Rightarrow equals(L,M);
equals(X, Y) \Rightarrow 0;
\]

Two Ways to Define Functions

- All at once, by a single equation:

\[
F(X_1, \ldots, X_n) = \text{some expression}
\]

- By cases, by a collection of rules:

\[
F(\text{pattern}_1) \Rightarrow \text{some expression}
\]

\[\ldots\]

\[
F(\text{pattern}_k) \Rightarrow \text{some other expression}
\]

- Next class we'll see a lot more examples of functions defined by cases.