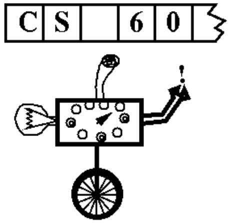

Lower-Level Functional Programming

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.

```
longerList(A,B) =


longestList(L) =
```

Exercise

- Assume we have predicates p and q.

```
!p(X)  ==


p(X) && q(Y) ==


p(X) || 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

```
factorial(N) =
    (N <= 0) ? 1 : N*factorial(N-1)
```

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

Tracing a Recursive Computation

```
factorial(3) ==
   3 * factorial(2) ==
   3 * (2 * factorial(1)) ==
   3 * (2 * (1 * factorial(0))) ==
   3 * (2 * (1 * 1)) ==
   3 * (2 * 1) ==
   3 * 2 ==
   6.
```

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 | 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 | [2,3] ] ==
[1 | [2 | [3]] ] ==
[1 | [2 | [3 | []] ] ]
```

More List Notation

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

```
[1,2,3] ==
[1 | [2,3]] ==
[1,2 | [3]] ==
[1,2,3 | []]
```

Patterns

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

```
L
[F | R]
[F, S | R]
[F, S, T]
[[F], S]
[[F, S] | R]
[F, S, T | 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 | 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?

```
L                    [1, 2, 3]
[F | R]              [1, [2,3]]
[F, S | R]           [[1], 2, 3]
[F, S, T]            [1, 2 | [3]]
[[F], S]             [[1,2], 3]
[[F, S] | R]         [1, 2, [3,4]]
[F, S, T | R]        7
```

Variable Definitions

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

```
rex > X = 3;
1

rex > [F | R] = [1,2,3];
1

rex > X + F;
4
```

More Variable Definitions

```
rex > [F, S, T | R] = [1, 2, [3,4]];
1

rex > F + S + length(R);
3

rex > [A, B, C | 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:

```
equals([ ], [ ]) => 1;
```

- Here => 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.

```
equals([A|L], [A|M]) => equals(L,M);
```

Third Equality Rule

- Otherwise, the two lists are not equal:

```
equals(X, Y) => 0;
```

(implied by “and only if”)

Summary of Equality Rules

```
equals([],    []    ) => 1;
equals([A|L],[A|M]) => equals(L,M);
equals(X,     Y     ) => 0;
```

Example using Equality Rules

- Are the lists `[1,2,3]` and `[1,2]` equal?
- Try the rules:

```
equals([1, 2, 3], [1, 2]) =>      (rule 2)
equals([2, 3], [2]) =>            (rule 2)
equals([3], []) =>                (rule 3)
0
```

- So (surprise!) the answer is no.

Two Ways to Define Functions

- All at once, by a single equation:
  $F(X_1,...,X_n) = \textit{some expression}$
- By cases, by a collection of rules:
  $F(\textit{pattern}_1) \Rightarrow \textit{some expression}$
  ...
  $F(\textit{pattern}_k) \Rightarrow \textit{some other expression}$
- Next class we'll see a lot more examples of functions defined by cases.