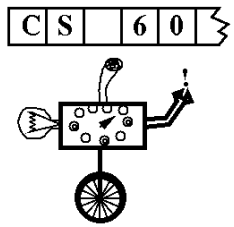

Higher-Order Functions

September 12, 2001

Review

- In the last class you saw a number of built-in rex functions.
  - `first`, `rest`, `reverse`, `cons`, `append`, `length`, `member`, `assoc`, ...
- Warmup Exercises:
  - `reverse(cons(F,R)) ==` ?
  - Define `second` (which is already built-in)

`range`

- `range` produces a "range" of numbers

```
range(1, 10) ==> [1,2,3,4,5,6,7,8,9,10]
```

- There is also a 3-argument version, in which the increment can be specified:

```
range(1, 4.5, 0.5) ==> [1,1.5,2,2.5,3,3.5,4,4.5]
```

`scale`

- `scale` multiplies the values in a list by a common factor

```
scale(3, [2, 4, 6, 8]) ==> [6, 12, 18, 24]
```

`remove_duplicates`

- `remove_duplicates` returns a new list with the 2nd, 3rd, ... of any element removed

```
remove_duplicates([2, 3, 4, 5, 2, 6, 5, 4])
                    ==>
              [2, 3, 4, 5, 6]
```

sort

- Sorts a list into "increasing" order
  - Numbers are ordered numerically
  - Strings are ordered alphabetically
  - Lists are ordered lexicographically

```
sort([3, 1, 4, 2]) ==> [1, 2, 3, 4]
```

- Relationships

```
length(sort(L)) ==
sort(reverse(L)) ==
sort(append(sort(L), sort(M))) ==
```

Predicates

- A ***predicate*** is a function that returns one of two values, for purposes of discrimination among arguments.
- In rex, the two values are:
  - 1, for true
  - 0, for false
- Some built-in rex predicates follow

`null` Predicate

- `null` tests a list for being empty:

```
null([ ]) ==> 1

null([1]) ==> 0
```

`member` predicate

- `member(X, L)` tells whether or not `X` occurs in list `L`

```
member(11, [5, 7, 11, 13]) ==> 1

member(12, [5, 7, 11, 13]) ==> 0
```

`even` and `odd` Predicates

- `even(X)` tells whether or not `X` is evenly divisible by 2.

```
even(11) ==> 0
even(12) ==> 1
```

- `odd(X)` tells whether or not `X` divided by 2 has a remainder of 1.

```
odd(11) ==> 1
odd(12) ==> 0
```

- Note: The arguments must be an integers.

`is_prime` predicate

- `is_prime(X)` tells whether or not `X` is prime (has any even divisors other than itself and 1)

```
is_prime(11) ==> 1

is_prime(12) ==> 0
```

- Note: The argument must be an integer.

Equality

- Even the equality operator `==` is a predicate with two arguments

```
'a' == 'a'  ==>  1
'a' == 'b'  ==>  0
```

New Predicates

- Predicates are just functions, so we can define others

```
is_zero(X) = (X == 0);
```

- Logical operators include `||` (or), `&&` (and), and `!` (not)

```
non_zero(X) = !is_zero(X)
is_two(X) = is_prime(X) && even(X)
```

"satisfy"

- When an argument value makes a predicate return value 1 (true), the argument is said to **satisfy** the predicate.
- This is useful in constructing sentences where the argument to the predicate is treated as active and the predicate is passive.

"satisfy" Example

- The predicate `is_prime` is satisfied by each of 2, 3, 5, 7, 11, ...
- It is not satisfied by 4, 6, 8, 9, 10, ...

Type Distinction

- We will use `T` to represent some arbitrary data type.
  - T^* means the type of lists of elements of type `T`.
- Here are some *type specifications*:

```
cons   : T × T* → T*
append : T* × T* → T*
first  : T* → T
rest   : T* → T*
```

- Here $\times$ means the *pairing* of arguments.

Higher-Order Functions

- By a higher-order function, we mean one that either:
  - takes a function as an argument, or
  - returns a function as a value
- Examples follow.

`map`

- `map` is an extremely useful function.
- Its first argument is a function of one argument.
- Its second argument is a list of valid arguments for this function.
- It applies the first argument to all of the elements in the list, giving a list of the results.

`map` Examples

```
map(odd, [2, 3, 4, 5, 6, 7, 8, 9])
     ==> [0, 1, 0, 1, 0, 1, 0, 1]

map(is_prime, [2, 3, 4, 5, 6, 7, 8, 9])
     ==> [1, 1, 0, 1, 0, 1, 0, 0]

square(X) = X*X;

  map(square, [2, 3, 4, 5, 6, 7, 8, 9])
   ==> [4, 9, 16, 25, 36, 49, 64, 81]
```

Exercise

- Give a type signature for map.
- (Hint: Let T stand for the type of elements in the list.)

3-argument map

- This version of map is defined similarly, but
  - The first argument is a binary (2-argument) function;
  - The 2nd and 3rd arguments are lists of equal length.
- The function argument is applied to pairs of corresponding elements, one from each list.
- The function returns a list of the results.

3-argument map

- General scheme:

$$\texttt{map(F, [}x_1,x_2,x_3,\ldots,x_n\texttt{], [}y_1,y_2,y_3,\ldots,y_n\texttt{])}$$
$$\texttt{==>}$$
$$\texttt{[F(}x_1\texttt{, }y_1\texttt{), F(}x_2\texttt{, }y_2\texttt{), …, F(}x_n\texttt{, }y_n\texttt{)]}$$

- Examples:

```
map(+, [1, 2, 3], [4, 5, 6]) ==> [5, 7, 9]
map(*, [1, 2, 3], [4, 5, 6]) ==> [4, 10, 18]
map(list, [1, 2, 3], [4, 5, 6])
            ==> [[1, 4], [2, 5], [3, 6]]
```

Exercise

- Give a type signature for the 3-argument map.
- (Note: The lists don't have to have the same type of element as each other.)

keep

- keep has a first argument that is a predicate and a second argument that is a list.
- It returns the list of values that satisfy the first argument.

keep(odd, [3, 4, 6, 5, 11, 12, 22, 31])
==>
[3, 5, 11, 31]

drop

- drop is like keep, except that it returns the list of values that do not satisfy the predicate argument.

```
drop(odd, [3, 4, 6, 5, 11, 12, 22, 31])
            ==> [4, 6, 12, 22]

is_zero(X) = (X == 0);

  drop(is_zero, [4, 6, 2, 0, 1, -5, 0])
           ==> [4, 6, 2, 1, -5]
```

Exercise

- *keep* and *drop* both have the same type signature; what is it?

reduce

- reduce takes three arguments:
  - a binary function op
    - op should be associative:
      x op (y op z) == (x op y) op z
  - a base value b
  - a list L = [x_1, x_2, x_3, …, x_n]
- It returns a single value:
  - If L is empty, then the value returned is b.
  - If L is not empty, the value is
    b op x_1 op x_2 op x_3 op … op x_n

Units

- Frequently (but not always!) the base value b is the *unit* for the binary function op.
- A unit u has the property that for any X,
  u op X == X op u == X.
- What are the units for +, *, and append?

reduce Examples

```
reduce(+, 0, [6, 7, 8, 9]) ==> 30

reduce(*, 1, [6, 7, 8, 9]) ==> 3024

reduce(append, [], [[1, 2, 3],[4, 5],[6]])
                    ==>
            [1, 2, 3, 4, 5, 6]
```