High-Level Functional Programming

High-Level Functional Programming

- By ***high-level*** we mean that we are only going to construct functions by composing together (usually powerful) built-in functions.
- We place the construction of functions based on the list dichotomy, for example, under *low-level*.

Some Built-in Functions in rex

- We already saw examples:
  - **length**: returns the length of a list
  - **sort**: returns a sorted version of a list
  - **reverse**: returns the reverse of a list
  - **append**: appends together two lists
- Other functions follow

zip

- **zip** “zips together” two lists:
  - zip([3, 5, 7], [11, 13, 17]) ➪

    [3, 11, 5, 13, 7, 17]

first

- **first** returns the first element of a non-empty list:
  - first([3, 5, 7, 11, 13]) ➪ 3
  - first([[3, 5, 7], 11, 13]) ➪ [3, 5, 7]
- first([ ]) doesn’t make sense; it returns an **error value**
- Be sure that the argument to first is not [ ].

rest

- **rest** returns a list of all but the first element of a non-empty list:
  - rest([3, 5, 7, 11, 13]) ➪ [5, 7, 11, 13]
  - rest([[3, 5, 7], 11, 13]) ➪ [11, 13]
- rest([ ]) doesn’t make sense; it returns an **error value**
- Be sure that the argument to rest is not [ ].

cons

- **cons** creates a list from a first element and another list:
  - cons(3, [5, 7, 11, 13]) ➪ [3, 5, 7, 11, 13]
  - cons([3, 5, 7], [11, 13]) ➪ [[3, 5, 7], 11, 13]
- IMPORTANT: cons is not *append*:
  - append([3, 5, 7], [11, 13]) ➪ [3, 5, 7, 11, 13]

Type Signature

- Suppose T is some data type
- Let T* mean the type of lists of elements of type T. Here are some **type signatures**:
- cons: $T \times T^* \to T^*$
- append: $T^* \times T^* \to T^*$
- first: $T^* \to T$
- rest: $T^* \to T^*$
- Here $\times$ means the ***pairing*** of arguments.

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]
- Type signature of range?

scale

- **scale** multiplies the values in a list by a common factor
- scale(3, [2, 4, 6, 8]) ➪ [6, 12, 18, 24]
- Type signature of scale?

assoc

- **assoc** "looks up" a value in an association list.
  - If found, the entire pair is returned.
  - If not found, [ ] is returned.
- assoc("c", [["a", 3], ["b", 5], ["c", 7]]) ➪ ["c", 7]
- assoc("d", [["a", 3], ["b", 5], ["c", 7]]) ➪ [ ]
- Type signature of assoc?

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]

Predicates

- A ***predicate*** is a function that returns one of two values, for purposes of discrimination among arguments.
- In rex, the two values of interest 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
- Type signature of null?

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 predicate

- **even**(X) tells whether or not X is evenly divisible by 2.
- even(11) ➪ 0
- even(12) ➪ 1
- Note: The argument must be an integer.

odd predicate

- **odd**(X) tells whether or not X divided by 2 has a remainder of 1.
- odd(11) ➪ 1
- odd(12) ➪ 0
- Note: The argument must be an integer.

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.

“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, ...

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
- Predicates are special cases of functions.

map

- **map** is an extremely useful function.
- Its first argument is a function of one argument.
- Its second argument is a list of values of the same type as the argument to the first argument.
- It applies the first argument to all of the elements in the list, giving a list as the result.

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]

In rex, we can define functions by equations this way.

Exercise

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

3-argument map in rex

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

3-argument map

- map(F, [x_1, x_2, x_3, ..., x_n], [y_1, y_2, y_3, ..., y_n]) ➪
  [F(x_1, y_1), F(x_2, y_2), ..., F(x_n, y_n)]

- 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])
  ➪ [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 operator, say b, of type $V \times V \rightarrow V$; b should be associative: b(x, b(y, z)) = b(b(x, y), z)
  - a value u of type V
  - a list L = [x_1, x_2, x_3, …, x_n] of values of type V
- It returns a single value of type V:
  - If L is empty, then the value returned is u.
  - If L is not empty, the value is
    b(…b(b(u, x_1), x_2), x_3), …, x_n)

Units

- If the first argument of reduce is an algebraic operator, then
- Normally the second argument is the ***unit*** for that operator.
- A unit has the property that for any X,
  b(u, X) = b(X, u) = X.
- 0 is the unit for +, 1 is the unit for *,
  [] is the unit for append.

Exercise

- What is an appropriate unit for:
  - max
  - min

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]

Anonymous Functions

- Sometimes it may be regarded as inconvenient to **name** functions such as isZero.
- Another problem arises when we want to **fix** one or more arguments to a function, leaving the remainder to vary.
- Both are solved by *anonymous* functions.

Anonymous Functions

- Functions have a **meaning** independent of the **names** we give them.
- We want a way to use a function without giving it a name.
- Notation:
  (X) => … some expression …
  means “the function that, with argument X, returns the value of … some expression ….

Example

- The function isZero, defined by:

    isZero(X) = X == 0;

  can also be written *anonymously*:

    (X) => X == 0

  read "the function that, with argument X, returns the value of X == 0".

Precedent

- This notation for talking about a function goes back to (at least) Bourbaki (French Mathematics Group), where the symbol

  $\mapsto$

  was used instead of

  =>
- Alonzo Church used the idea extensively, but with a different symbol λ as a *prefix*.

More Anonymous Functions

- (X) => X+5 — The function that adds 5
- (X) => X*5 — The function that multiplies by 5
- (X) => X*X — The function that squares
- (X, Y) => Y/X — The function that divides its second argument by its first.

Sample Usage

- map((X)=>X+5, [1, 2, 3, 4])

  ➪ [6, 7, 8, 9]
- map((X)=>X*X, [1, 2, 3, 4])

  ➪ [1, 4, 9, 16]

Exercise

- Give an equation defining *scale* using *map*, where

    scale(F, L) multiplies each element of L by a factor F.

Anonymous Functions with "Imported" Values

- drop_multiples(X, L) =

    drop((Y) => (Y%X == 0), L)

    The predicate that tests divisibility by X.
- Here X is **imported** to the anonymous function; it is not an argument to it.
- This form of usage is **VERY IMPORTANT.**

Exercises

- Give an equation defining pairWith, such that
  pairWith(X, L) creates a list in which each element of L is paired with X:
  pairWith(3, [1, 2, 3])
  ➪ [[3, 1], [3, 2], [3, 3]]

Exercises

- Can you give an equation defining pairs, such that
  pairs(L, M) creates a list in which each element of L is paired with each element of M, e.g.
  pairs([1, 2, 3], [4, 5, 6])
  ➪ [[1, 4], [1, 5], [1, 6],
  [2, 4], [2, 5], [2, 6],
  [3, 4], [3, 5], [3, 6]]

find function

- find(P, L) returns the longest suffix of L that begins with an element satisfying P.
- Example:
  - find(odd, [2, 4, 6, 7, 9, 10, 12])
    ➪ [7, 9, 10, 12]
- As with map, etc., find is often used with anonymous functions.

find_index function

- find_index(P, L) returns the index of the first element L that begins with an element satisfying P.
- Example:
  - find_index(odd, [2, 4, 6, 7, 9, 10, 12])
    ➪ 3
- Indices start with 0 as for the first element of the list.

find_indices function

- find_indices(P, L) returns the list of indices of elements of L that satisfy P.
- Example:
- find_indices(odd, [2, 4, 6, 7, 9, 8, 12, 13])
  ➪ [3, 4, 7]