Functional Programming
Functional Programming

- Functional programming is one of the major fundamental programming paradigms.

- It means programming only by composing functions, not using assignment statements.

- It can be used in conjunction with other paradigms, such as object-oriented programming.
Functional Programming is “Complete”

- There is a certain well-defined sense in which a programming language can be called “complete”:
  - The language is capable of representing any computable function.
  - Most languages of significance, including most functional ones, are complete in this sense.
  - More on the definition of “computable” and “complete” later.
A Functional Programming Language

- We will use the language rex to exemplify functional programming.
- rex is interactive:
  - Definitions are entered.
  - Expressions are evaluated to get results.
- You may run rex on turing:
  - turing > rex
  - rex >
rex usage examples
(user input is shown in bold)

rex > length([[1, 2], [3, 4], [5, 6]]);
3

rex > sort([3, 9, 1, 2, 8, 7, 5, 6, 4]);
[1, 2, 3, 4, 5, 6, 7, 8, 9]

rex > sort(['oats', 'peas', 'beans', 'barley']);
[barley, beans, oats, peas]

rex >
more rex usage examples
(define variables to avoid re-entry)

rex > x = [3, 9, 1, 2, 8, 7, 5, 6, 4];
1

This 1 means true, the definition was accepted.

rex > x;
[3, 9, 1, 2, 8, 7, 5, 6, 4]

rex > sort(x);
[1, 2, 3, 4, 5, 6, 7, 8, 9]

rex > x;
[3, 9, 1, 2, 8, 7, 5, 6, 4]
more rex usage examples
(previous session continued)

rex > length(x);
9

rex > reverse(x);
[4, 6, 5, 7, 8, 2, 1, 9, 3]

rex > append(x, x);
[3, 9, 1, 2, 8, 7, 5, 6, 4,
  3, 9, 1, 2, 8, 7, 5, 6, 4]
Load files to prevent re-typing

contents of file text.rex, prepared with a text editor, such as Emacs:

```
// This is a set of rex definitions, with comments

// x is a list of some small random numbers.
x = [3, 9, 1, 2, 8, 7, 5, 6, 4];

// y is a list of some grains.
y = sort(["oats", "peas", "beans", "barley"]);

// z is a list of pairs
z = [ [1, 2], [3, 4], [5, 6] ];

/*
   Above you see comments to end-of-line.
   You can also have multi-line comments such as this one,
   just like Java or C++.
*/
```
At least two ways to load a file:

Method 1: Include the file name on the UNIX command line:

```
unix > rex test.rex
test.rex loaded
rex > x;
[3, 9, 1, 2, 8, 7, 5, 6, 4]

rex > y;
[barley, beans, oats, peas]

rex > z;
[[1, 2], [3, 4], [5, 6]]
```

You can re-run the command without retyping, e.g.

```
unix > !r
```
At least two ways to load a file:

Method 2: Include the file from a rex command line

```plaintext
unix > rex
rex > *i test.rex
read file test.rex
rex > x;
[3, 9, 1, 2, 8, 7, 5, 6, 4]
rex > y;
[barley, beans, oats, peas]
rex > z;
[[1, 2], [3, 4], [5, 6]]
```
Split-screen editing in Emacs
(what I use most of the time)

In Emacs:
control-x 2 to split window
escape-x shell to get shell
Can cut/paste using only keystrokes

---

UNIX shell in emacs window

Your rex file for editing
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
  - **member**: tells whether something is in 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]`
Suppose $T$ is some data type

Let $T^*$ mean the type of lists of elements of type $T$. Here are some type signatures:

- $\text{cons}: T \times T^* \rightarrow T^*$
- $\text{append}: T^* \times T^* \rightarrow T^*$
- $\text{first}: T^* \rightarrow T$
- $\text{rest}: T^* \rightarrow T^*$

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])** \(\rightarrow\) [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]) \(\rightarrow\) [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

- \( \text{member}(X, L) \) tells whether or not \( X \) occurs in list \( L \)
- \( \text{member}(11, [5, 7, 11, 13]) \equiv 1 \)
- \( \text{member}(12, [5, 7, 11, 13]) \equiv 0 \)
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) \equiv 1`
- `is_prime(12) \equiv 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

- \(\text{map}(F, [x_1, x_2, x_3, \ldots, x_n], [y_1, y_2, y_3, \ldots, y_n]) \Rightarrow [F(x_1, y_1), F(x_2, y_2), \ldots, F(x_n, y_n)]\)

- **Examples:**
  - \(\text{map}(+, [1, 2, 3], [4, 5, 6]) \Rightarrow [5, 7, 9]\)
  - \(\text{map}(*, [1, 2, 3], [4, 5, 6]) \Rightarrow [4, 10, 18]\)
  - \(\text{map}(\text{list}, [1, 2, 3], [4, 5, 6]) \Rightarrow [[[1, 4], [2, 5], [3, 6]]]\)
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])
  \(\Rightarrow\) [3, 5, 11, 31]
**drop**

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

- \text{drop}(\text{odd}, [3, 4, 6, 5, 11, 12, 22, 31]) \Rightarrow [6, 12, 22]

- \text{is\_zero}(X) = X == 0;
drop(\text{is\_zero}, [4, 6, 2, 0, 1, -5, 0]) \Rightarrow [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 \( V \sqcap V \sqcap V \);
    - \( b \) 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
    \[ b(...b(b(b(u, x_1), x_2), x_3), ..., x_n) \]
Units

- If the first argument of reduce is an algebraic operator, then
- Normally the second argument is the \textit{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 $\ast$, $[]$ is the unit for append.
Exercise

What is an appropriate unit for:

- max
- min
reduce Examples

- $\text{reduce}(+, 0, [6, 7, 8, 9]) \Rightarrow 30$

- $\text{reduce}(*, 1, [6, 7, 8, 9]) \Rightarrow 3024$

- $\text{reduce}(\text{append}, [], [[1, 2, 3], [4, 5], [6]]) \Rightarrow [1, 2, 3, 4, 5, 6]$
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:
  \[(X) \Rightarrow \ldots \text{some expression} \ldots\]
  means “the function that, with argument \(X\), returns the value of \(\ldots\) some expression \ldots”.
Example

- The function isZero, defined by:
  
isZero(X) = X == 0;

  can also be written anonymously:

  \[(X) \Rightarrow X == 0\]

  read “the function that, with argument \(X\), returns the value of \(X == 0\)”.
This notation for talking about a function goes back to (at least) Bourbaki (French Mathematics Group), where the symbol \( \rightarrow \) was used instead of \( \Rightarrow \).

Alonzo Church used the idea extensively, but with a different symbol \( \overline{\rightarrow} \) as a prefix. This notation requires some acclimation.
More Anonymous Functions

- \((X) \Rightarrow X+5\)  The function that adds 5
- \((X) \Rightarrow X\times5\)  The function that multiplies by 5
- \((X) \Rightarrow X\times X\)  The function that squares
- \((X, Y) \Rightarrow Y/X\)  The function that divides its second argument by its first.
Sample Usage

- map((X)=>X+5, [1, 2, 3, 4])
  \[\Rightarrow [6, 7, 8, 9]\]

- map((X)=>X*X, [1, 2, 3, 4])
  \[\Rightarrow [1, 4, 9, 16]\]
Give an equation defining \textit{scale} using \textit{map}, where
\[
\text{scale}(F, L) \text{ multiplies each element of } L \text{ by a factor } F.
\]
Anonymous Functions with “Imported” Values

- \( \text{drop}\_\text{multiples}(X, L) = \)
  \[
  \text{drop}((Y) \Rightarrow (Y \% X == 0), \ L)
  \]

  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**.
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])

\[\Rightarrow [ [3, 1], [3, 2], [3, 3] ]\]
Can you give an equation defining a function \texttt{pairs}, such that \texttt{pairs}(L, M) creates a list in which each element of \texttt{L} is paired with each element of \texttt{M}, e.g.

\texttt{pairs([[1, 2, 3], [4, 5, 6]])} \\
\Rightarrow [ [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])
    - $\Rightarrow [7, 9, 10, 12]$

- 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])
  \[\Rightarrow [3, 4, 7]\]
Function Decomposition

- This means: Implement a complex function in terms of simpler ones.
- This idea is of universal importance.
- Those simpler functions can be implemented in terms of still-simpler ones, and so on, until we get down to built-in functions.
Construct a function that will tell whether a directed graph, represented as a list of arcs, is acyclic.

cyclic:

acyclic:
“Pruning” Method

- Rosalind B. Marimont
  A new method of checking the consistency of precedence matrices
  Journal of the ACM 6, 164-171, 1959

- Pruning away any arcs that point to leaves does not change the cyclic/acyclic nature of the graph.

- Pruning such arcs may produce additional leaves.

- Prune until no further pruning is possible:
  - If the result is empty, the original graph was acyclic.
  - If not, it was cyclic.
Examples of Pruning
(Leaves shown in green)

Example 1:

Example 2:

Empty (no arcs)
Pruning with Graphs as Lists

- **Example 1:**
  - $[ [a, b], [a, c], [b, d], [c, d], [c, e], [e, a] ]\Rightarrow$
  - $[ [a, b], [a, c], [c, e], [e, a] ]\Rightarrow$
  - $[ [a, c], [c, e], [e, a] ](no\ leaves)$

- **Example 2:**
  - $[ [a, b], [a, c], [b, d], [c, d], [c, e]]\Rightarrow$
  - $[ [a, b], [a, c]]\Rightarrow$
  - $[ ]\Rightarrow$
Note

- We are assuming that every node in the graph is on one or the other end of an arc, i.e. there are no isolated nodes, as in the graph below.
- Otherwise, we’d have to represent the graph with two lists: one of nodes and one of arcs.
Functional Code

- Basic idea:
  - As long as there is a leaf:
    Remove leaves and their attached arcs

- Translation:
  - `isAcyclic(Graph) = null(iterate(removeLeaves, hasLeaf, Graph));`

  - test for empty list
  - iterate first arg. as long as second arg. true
  - remove all leaves and their arcs
  - test whether there is a leaf
  - starting graph
hasLeaf

- A Graph has a leaf iff isLeaf is true for one of its nodes.
- hasLeaf(Graph) = 
  some((Node) => isLeaf(Node, Graph), nodes(Graph));

  test whether first arg. is true for some element of second arg.
  true when Node is a leaf of this Graph
  list of nodes of Graph
A node is a leaf if it is not the first of any arc in the graph.

\[
isLeaf\text{(Node, Graph)} = \neg \text{member\text{(Node, map\text{(first, Graph)})}};
\]
nodes(Graph)

- nodes(Graph) =
  remove_duplicates(append(map(first, Graph),
                      map(second, Graph)));

- remembering our assumption: that every node in the graph is on
  one or the other end of an arc, i.e. there are no isolated nodes,
  as in the graph below.
To remove the leaves:

- remove any arc that points to a leaf

\[
\text{removeLeaves(Graph)} = \text{drop}((\text{Arc}) \Rightarrow \text{isLeaf}((\text{second(Arc)}, \text{Graph}), \text{Graph}));
\]

- the node to which Arc points
- the list of arcs in the graph
iterate

iterate(action, continue, State) =
continue(State) ?
  iterate(action, continue, action(State))
: State;

conditional expression (as in C++, Java)
P ? A : B
means if P is true then the value of the expression is A; otherwise it is B.