Low-Level Functional Programming

What's "Low-Level" About This?

- "low-level" refers to the construction of functions by explicitly creating and decomposing lists a few elements at a time.
- Previously we used higher-order functions to do most of the non-trivial work in a functional decomposition.
- Now we are going to use pattern matching rules, recursion, etc.

Fundamental List Dichotomy

- A list is either:
  - **empty**, [ ]   or
  - **non-empty**, in which case it has both a
    - first
    - rest
- Most list definitions deal with these cases separately.
- Definitions are typically a form of **inductive definition**, in which [ ] is the basis.

List Decomposition Notation

- When a list is non-empty, it has a first element and the rest of the elements form a list.
- The general ***form*** of a **non-empty list** will be represented:

  [F | R]

  read "F followed by" R.
- Here **F** is a variable represents the **first** element, and **R** is a variable representing the **rest** of the elements (**R** has a **list** as its value, even though brackets aren't around R).

List Decomposition Example

- Consider a defining equation:

  [F | R] = [1, 2, 3, 4]

  F is a variable represents the first element, so:

  F == 1

  R is a variable representing the rest of the elements, so:

  R == [2, 3, 4]

List Decomposition Clarified

- A defining equation:

  [F | R] = *some list*

  can ***only*** be valid when the RHS list is ***non-empty***.

  Thus

  [F | R] = [ ] can *never* be a valid equation.

Defining Functions by Rules

- Suppose we want to define a function taking an arbitrary list as an argument.
- It is sufficient to:
  - define the function on the empty list, and
  - define the function on a general non-empty list.

called the "basis"

called the "induction step" or "recursion"

Example

- Define the function **halve_all**, which divides every element in a list by 2.
  - halve_all([ ]) => [ ];
  - halve_all([F | R]) => [F/2 | halve_all(R)];
- This can be read:
  - "halving all of the empty list is the empty list."
  - "halving all of a non-empty list is half of the first element ***followed by*** halving all of the rest."

Computation by "Rewriting"

- halve_all([2, 4, 6]) ==>
- [1 | halve_all([4, 6])] ==>
- [1 | [2 | halve_all([6])]] ==>
- [1 | [2 | [3 | halve_all([ ])]]] ==>
- [1 | [2 | [3 | [ ]]]] ==
- [1 | [2 | [3]]] ==
- [1 | [2, 3]] ==
- [1, 2, 3]

Extended Notation for Greater Readability

- The first so-many, rather than just the first, element, can be shown separated by commas:
  - [a, b, c, d | R] means a list with at least 4 elements, a, b, c, d, followed by the elements in list R (which could be empty).
- In the extended notation:
  - halve_all([2, 4, 6]) ==>
  - [1 | halve_all([4, 6])] ==>
  - [1, 2 | halve_all([6])] ==>
  - [1, 2, 3 | halve_all([ ])] ==>
  - [1, 2, 3]

A Way of Remembering

- The combination
  | […]
  inside a list "melts away" into
  , …
  If … is empty, then it just melts away, period.
- Examples:
  - [1 | [2, 3, 4]] == [1, 2, 3, 4]
  - [1, 2 | [3 , 4]] == [1, 2, 3, 4]
  - [1, 2, 3 | [4]] == [1, 2, 3, 4]
  - [1, 2, 3, 4 | [ ]] == [1, 2, 3, 4]

Alternate

- Of course, we could have just used ***map*** *in this particular case*:
  - halve(A) = A/2;
  - halve_all(X) = map(halve, X);
- Use higher order functions such as *map* when possible; resort to lower-order ones when you think you need to.
- Higher-order functions can often tell the story more succinctly.

Define from a low-level:

- map
- reduce

Example

- Define the function **member** which tests whether the first argument is an element of the list in the second argument.
  - member(X, [ ]) => 0;
  - member(X, [F | R]) =>
    (X == F) ? 1 : member(X, R);

    conditional expression (as in C++, Java)

Alternate

- Instead of using a conditional expression, use a third rule with **pattern matching**:
  - member(X, [ ]) => 0;
  - member(X, [X| R]) => 1; Note: X's values must match.
  - member(X, [F| R]) => member(X, R);
- The rule used is always the **first** (from top to bottom) applicable one.

Rule Matching

For reference:
member(X, [ ]) => 0; // rule 1
member(X, [X| R]) => 1; // rule 2
member(X, [F| R]) => member(X, R); // rule 3

- Consider evaluating
  - member(3, [1, 2, 3, 4]) ==> rule 3 is the first that matches
  - member(3, [2, 3, 4]) ==> rule 3 is the first that matches
  - member(3, [3, 4]) ==> rule 2 is the first that matches
  - 1

Rule Matching

- Consider evaluating
  - member(5, [1, 2, 3]) ==> rule 3 is the first that matches
  - member(5, [2, 3]) ==> rule 3 is the first that matches
  - member(5, [3]) ==> rule 3 is the first that matches
  - member(5, []) ==> rule 1 is the first that matches
  - 0

Using Conditional Guards

- Alternatively, can use a **conditional guard**:
  - member(X, [ ]) => 0;
  - member(X, [F| R]) => (X == F) ? 1;

    conditional guard
  - member(X, [F| R]) => member(X, R);
- The condition is tested after any other matching is applied.
- If the condition fails, then subsequent rules are tried.

Define from a low-level:

- keep
- range

Matching with Two or More List Arguments

- Some functions have more than one list argument.
- Induction might, or might not, use rules that dichotomize **both** lists.

Example: append

Example: List Equalty First Rule

- Two lists are equal if they both are empty:

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

List Equalty: Second Rule

- Two lists are equal if they are both non-empty and
  - the first elements of each are the same, and
  - the lists of the rest of the elements of each are equal.

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

List Equalty: Third Rule

- Otherwise, the two lists are not equal:

  equals(X, Y) => 0;

Summary of Equality Rules

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

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

3 equals(X, Y) => 0;

Example of List Equality

- Revisit our earlier example:
  - Are these lists equal:

    [1, 2, 3] vs. [1, 2] ?
- Try the rules:
  - equals([1, 2, 3], [1, 2]) ==> (rule 2)
  - equals([2, 3], [2]) ==> (rule 2)
  - equals([3], [ ]) ==> (rule 3)
  - 0
- i.e. the lists are not equal.

The Merge Pattern

- This **important** pattern arises many times in different applications.
- Construct a function that merges two lists of numbers:
  - The argument lists are already in ascending order.
  - The result is to be in ascending order as well.

Merge Example

- merge([3, 6, 7, 9, 13, 15], [2, 5, 8, 16, 20])

  ==> [2, 3, 5, 6, 7, 8, 9, 13, 15, 16, 20]

- Notes:
  - merge is built into rex
  - There are many similar functions that aren't.

Merge by Induction

- merge([ ], M) => M;
- merge(L, [ ]) => L;
- merge([A | L], [B | M]) =>

  A < B ?

  [A | merge(L, [B | M])]

  : [B | merge([A | L], M)];

Similar Example

- Consider the representation of numbers as lists of prime factors with multiplicity, e.g.

  200 represented as [[2, 4], [5, 2]]

  with factors in **increasing** order.
- Want to define functions gcd and lcm on this representation:
  - gcd(R, S) = representation of greatest common divisor of R and S
  - lcm(R, S) = representation of least common multiple of R and S
  - gcd([[2, 4], [5, 2]], [[2, 2], [3, 4], [5, 7]]) ==> ?

Using Auxiliary Functions

- Often the function to be defined is not directly definable in a natural or efficient way using recursion. A **helper** or **auxiliary** function may be necessary.
- Example: reverse

Naïve Reverse

- reverse([]) => [];
- reverse([A | L]) => append(reverse(L), [A]);

- This is "naïve" because:
  - It's the first thing that comes to mind.
  - It's very inefficient:
    - The time taken will be proportional to the **square** of the length of the list.

Use auxiliary to make non-naive reverse

- reverse(L) = reverse(L, [ ]);

- reverse([ ], M) => M;
- reverse([A | L], M) => reverse(L, [A | M]);

- Example of function-name overloading.
- reverse(L, M) "appends M to the reverse of L".
- reverse(L, [ ]) therefore reverses L.

Mixed Functional Programming Examples

- Use low-level **or** high-level, whatever fits best
  - Maybe start with low-level, and the use high-level retrospectively
- Radix conversion
  - Tail recursion
- Tree and graph searching (later slides)

Convert Number to Binary

- Example:
  - toBinary(37) ⇨

[1,	0,	0,	1,	0,	1]
1*32 +	0*16 +	0*8 +	1*4 +	0*2 +	1*1
2^5	2^4	2^3	2^2	2^1	2^0

- First try:
  - divide by 2, record remainder, continue with quotient
  - until 0

Convert Numbers to Binary

- Rules:
  - toBinary1(0) => [ ];
  - toBinary1(N) => [$\underbrace{\text{N\%2}}_{\text{remainder}}$ | toBinary1($\underbrace{\text{N/2}}_{\text{quotient}}$)];

- Any problems with this definition?

Convert Number to Binary

- Another try:
  - toBinary(N) = toBinary2(N, []);
  - toBinary2(0, Acc) => Acc;
  - toBinary2(N, Acc) =>
    toBinary2(N/2, [N%2 | Acc]);
    quotient remainder
- Why is this definition better?
- What is still lacking?

Accumulators and Tail Recursion

- From previous slide:
  - toBinary2(0, Acc) => Acc;
  - toBinary2(N, Acc) =>
    toBinary2(N/2, [N%2 | Acc]);
- Acc is called an "accumulator" argument:
  - It "accumulates" the result until the basis case is reached, the "unloads" it.
- This type of recursion is called "tail recursion":
  - There is no "cleanup" to be done after the recursive call to toBinary2, and therefore no need to **"stack"** calls.
  - We can effectively "turn over control" to the subordinate call; giving a form of **iteration**.

Accumulators and Tail Recursion

Version with accumulator

- toBinary2(37, []) ==>
- toBinary2(18, [1]) ==>
- toBinary2(9, [0, 1]) ==>
- toBinary2(4, [1, 0, 1]) ==>
- toBinary2(2, [0, 1, 0, 1]) ==>
- toBinary2(1, [0, 0, 1, 0, 1]) ==>
- toBinary2(0, [1, 0, 0, 1, 0, 1]) ==>
- [1, 0, 0, 1, 0, 1]

Version without accumulator

- toBinary1(37) ==>
- [1 | toBinary1(18)] ==>
- [1, 0 | toBinary1(9)] ==>
- [1, 0, 1 | toBinary1(4)] ==>
- [1, 0, 1, 0 | toBinary1(2)] ==>
- [1, 0, 1, 0, 0 | toBinary1(1)] ==>
- [1, 0, 1, 0, 0, 1| toBinary1(0)] ==>
- [1, 0, 1, 0, 0, 1| [ ]] ==
- [1, 0, 1, 0, 0, 1]

Notes:

- Can similarly convert to any given base,
- by passing the base as an argument.
- Can convert back (from numeral list to number).

Exercise

- Construct fromBinary, e.g.
  - fromBinary([1, 0, 0, 1, 0, 1]) ==> 37
- Considerations:
  - Do we need an accumulator?
  - Can it be done with tail-recursion?
  - Try it and see.

An Approach

- Write **iterative pseudo-code**, then construct recursive equivalent.
- L = ... list to be converted ...;

```
Result = 0;
while( L != [ ] )
      {
      Result = 2*Result + first(L);
      L = rest(L);
      }
... answer is in Result ...
```

- Defining fromBinary3(L, Result):
- fromBinary3([], Result) => Result;
- fromBinary3([F | R], Result) =>
  fromBinary3(R, 2*Result+F);
- fromBinary(L) = fromBinary3(L, 0);

- fromBinary3([1, 0, 0, 1, 0, 1], 0) ==>
- fromBinary3([0, 0, 1, 0, 1], 1) ==>
- fromBinary3([0, 1, 0, 1], 2) ==>
- fromBinary3([1, 0, 1], 9) ==>
- fromBinary3([0, 1], 9) ==>
- fromBinary3([1], 18) ==>
- fromBinary3([], 37) ==>
- 37

Exercise

- What if the list were least-significant bit first?
  - Can you do construct the function?
  - Can you construct a tail-recursive implementation?

Exercises

- Compare "obvious" and tail-recursive forms of:
  - factorial function (fac(n) = 1*2*3*…*n)
  - length function
  - sum of a list
  - reduce
  - reverse

Essential Non-Tail Recursions

- Some functions don't admit a tail-recursive version (unless *reverse* is used before or after):
  - Examples:
    - map, keep, drop
    - append

append Elimination

(aka "appendectomy")

- When maximum efficiency is desired, uses of append should be avoided.
- It is often possible to get rid of append by defining versions of functions with an extra accumulator argument.
- Example:

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

- Show how to avoid *append* by generalizing map to take an accumulator.

reverse elimination

- Some functions naturally build lists in reverse.
- Rather than immediately reversing the result, consider leaving it as is (in reversed form) and exploiting this fact at a later stage of the function "pipeline".
- Some functions, such as map, keep, drop, … work equally well whether or not the data is in reverse order.