Low-Level Functional Programming
What’s “Low-Level” About?

- “low-level” refers to the construction of functions by explicitly composing and decomposing lists.

- 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:
  \[ [ F \mid R ] \]
  Here \( F \) is a variable represents the first element, and \( R \) is a variable representing the rest of the elements
  (Note: \( R \) has a list as its value, even though brackets aren’t shown around \( R \)).
List Decomposition Example

- Consider a defining equation:
  
  \[ [ F \mid R ] = [1, 2, 3, 4] \]

  \( F \) is a variable representing 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:
  \[ [ F \mid R ] = \text{some list} \]
  can only be valid when the RHS list is non-empty.

Thus
  \[ [ F \mid R ] = [ ] \text{ can never be a valid equation.} \]
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”
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.”

This is our first example of a recursive definition.
Computation by “Rewriting”

- \( \text{halve\_all}([2, 4, 6]) \Rightarrow \)
- \( [1 \mid \text{halve\_all}([4, 6])] \Rightarrow \)
- \( [1 \mid [2 \mid \text{halve\_all}([6])] ] \Rightarrow \)
- \( [1 \mid [2 \mid [3 \mid \text{halve\_all}([ \ ])] ] ] \Rightarrow \)
- \( [1 \mid [2 \mid [3 \mid [ \ ] ] ] ] == \)
- \( [1 \mid [2 \mid [3] ] ] == \)
- \( [1 \mid [2, 3] ] == \)
- \( [1, 2, 3] \)
The first so-many, rather than just the first, element, can be shown separated by commas:

\[[a, b, c, d \mid R]\] 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:

- \text{halve_all}([2, 4, 6]) ➞ 
- \[1 \mid \text{halve_all}([4, 6])\] ➞ 
- \[1, 2 \mid \text{halve_all}([6])\] ➞ 
- \[1, 2, 3 \mid \text{halve_all}([\ ])]\] ➞ 
- \[1, 2, 3\]
A Way of Remembering

- The combination
  \[ [ \text{...} ] \]

*inside* a list “melts away” into

, ...

unless ... is empty, then it just melts away

- Examples:
  - \([1 \mid [2, 3, 4]] = [1, 2, 3, 4] \)
  - \([1, 2 \mid [3, 4]] = [1, 2, 3, 4] \)
  - \([1, 2, 3 \mid [4]] = [1, 2, 3, 4] \)
  - \([1, 2, 3, 4 \mid [\;\;]] = [1, 2, 3, 4] \)
Of course, we could have just used \textit{map in this particular case}:

- \textit{halve}(A) = A/2;
- \textit{halve\_all}(X) = \textit{map}(\textit{halve}, X);

Use higher order functions such as \textit{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 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)*
Instead of using a conditional expression, use a third rule with **pattern matching**:

- `member(X, [ ])) => 0;`
- `member(X, [X| R])) => 1;`
- `member(X, [F| R])) => member(X, R);`

The rule used is always the **first** (from top to bottom) applicable one.
Rule Matching

- Consider evaluating
  - \texttt{member(3, [1, 2, 3, 4])} ➔ rule 3 is the first to apply
  - \texttt{member(3, [2, 3, 4])} ➔ rule 3 is the first to apply
  - \texttt{member(3, [3, 4])} ➔ rule 2 is the first to apply
  - 1

\begin{center}
\begin{tabular}{l}
member(X, [ ]) => 0; \quad // rule 1 \\
member(X, [X| R]) => 1; \quad // rule 2 \\
member(X, [F| R]) => member(X, R); \quad // rule 3
\end{tabular}
\end{center}
Rule Matching

- Consider evaluating
  - member(5, [1, 2, 3]) ➔ rule 3 is the first to apply
  - member(5, [2, 3]) ➔ rule 3 is the first to apply
  - member(5, [3]) ➔ rule 3 is the first to apply
  - member(5, []) ➔ rule 1 is the first to apply
  - 0
Second Alternate
(less desirable)

- Use a **conditional guard**:
  - member(X, [ ]) => 0;
  - member(X, [F| R]) => (X == F) ? 1;
  - 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.
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: List Equality
First Rule

- Two lists are equal if they both are empty:
  equals([], []) => 1;
List Equality:
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.

\[ \text{equals}([A \mid L], [A \mid M]) \Rightarrow \text{equals}(L, M); \]
List Equality:
Third Rule

- Otherwise, the two lists are not equal:
  {equals(X, Y) => 0;
Summary of Equality Rules

1. $\text{equals}([\ ], [\ ]) \Rightarrow 1$;
2. $\text{equals}([A \mid L], [A \mid M]) \Rightarrow \text{equals}(L, M)$;
3. $\text{equals}(X, Y) \Rightarrow 0$;
Example of List Equality

- Revisit our earlier example:
  - Are these lists equal:
    
    \[ [1, 2, 3] \text{ vs. } [1, 2] \] ?
  
- Try the rules:
  - \( \text{equals}([1, 2, 3], [1, 2]) \) ➔ (rule 2)
  - \( \text{equals}([2, 3], [2]) \) ➔ (rule 2)
  - \( \text{equals}([3], [\ ]) \) ➔ (rule 3)
  - 0

- i.e. the lists are not equal.
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 searching
Convert Number to Binary

- Example:
  - `toBinary(37) ➞ [1, 0, 0, 1, 0, 1]`
    - $32 + 0*16 + 0*8 + 4 + 0*2 + 1$

- First try:
  - divide by 2, record remainder, continue with quotient
  - until 0
Convert Number to Binary

- **Rules:**
  - `toBinary1(0) => [ ];
  - `toBinary1(N) => [N%2 | toBinary1(N/2)];`

- **Problems with this definition?**
Another try:

- `toBinary(N) = toBinary2(N, [])`;

- `toBinary2(0, Acc) => Acc`;

- `toBinary2(N, Acc) => toBinary2(N/2, [N%2 | Acc])`;

Why is this definition better?
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

- `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]`

- `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.
- Can pass the base as an argument.
- Can convert back (from numeral list to number).
Exercise

- Construct from\texttt{Binary}, e.g.
  \[
  \text{fromBinary}([1, 0, 0, 1, 0, 1]) \equiv 37
  \]

- 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.

- \( L = \ldots \text{list to be converted} \ldots; \)  
  \[ \text{Result} = 0; \]
  \[ \text{while}(L \neq [ \ ]){} \]
  \[ \quad \text{Result} = 2*\text{Result} + \text{first}(L); \]
  \[ \quad L = \text{rest}(L); \]
  \[ \} \]
  \[ \ldots \text{answer is in Result} \ldots \]

- Defining \( \text{fromBinary3}(L, \text{Result}): \)
  
  \( \text{fromBinary3}([\ ], \text{Result}) \Rightarrow \text{Result}; \)
  
  \( \text{fromBinary3}([F \mid R], \text{Result}) \Rightarrow \)
  \[ \text{fromBinary3}(R, 2*\text{Result}+F); \]
  
  \( \text{fromBinary}(L) = \text{fromBinary3}(L, 0); \)
  
  \( \text{fromBinary3}([1, 0, 0, 1, 0, 1], 0) \Rightarrow \)
  
  \( \text{fromBinary3}([0, 0, 1, 0, 1], 1) \Rightarrow \)
  
  \( \text{fromBinary3}([0, 1, 0, 1], 2) \Rightarrow \)
  
  \( \text{fromBinary3}([1, 0, 1], 9) \Rightarrow \)
  
  \( \text{fromBinary3}([0, 1], 9) \Rightarrow \)
  
  \( \text{fromBinary3}([1], 18) \Rightarrow \)
  
  \( \text{fromBinary3}([\ ], 37) \Rightarrow \)
  
  \( 37 \)
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`
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.