Goals

• Contrast type synonyms with new types
• Investigate why accumulator-style recursion can be more efficient
• See pattern-matching for built-in “one of” types (not really a concept, but important for homework)
• See the elegance of “deep patterns” and a generalization of what bindings are
  – What we have been doing is just a special case
Type synonyms

You can bind a *type name* to a type. Example:

    type intpair = int * int

(We call something else a *type variable*.)

In ML, this creates a *synonym*, also known as a *transparent* type definition. Recursion not allowed.

So a type name is *equivalent* to its definition.

We’ll have much more to say about equivalence and *abstract* types later.

To contrast, the type a datatype binding introduces is not equivalent to any other type (until possibly a later type binding).
Recursion

You should now have the hang of recursion:

- It’s no harder than using a loop (whatever that is)
- It’s much easier when you have multiple recursive calls (e.g., with functions over ropes or trees)

But there are idioms you should learn for *elegance, efficiency, and understandability.*

Today: using an *accumulator.*
Accumulator lessons

- Accumulators can avoid data-structure copying
- Accumulators can reduce the depth of recursive calls that are not tail calls
- Key patterns:
  - Non-accumulator: compute recursive results and combine
  - Accumulator: use recursive result as new accumulator
  - The base case becomes the initial accumulator

You will use recursion in non-functional languages—this lesson still applies.

Note: We spent considerable time investigating how to_list_1 and to_list_2 work using the overhead projector.
Back to patterns

We saw that the case expression was how to test variants and extract values from datatype values. Advantages:

- exhaustiveness and redundancy checked for us
- more concise syntax for binding local variables to extracted values

In fact, case expressions are the preferred way to test variants and extract values from all ML’s “one-of” types, including predefined ones.

So: Do not use functions hd, tl, null, isSome, val0f

Teaser: These functions are useful for passing as values

Note:

- You could define all these functions yourself
- [] and :: are just funny-looking constructors; NONE and SOME aren’t even funny-looking
Tuple patterns

You can also use patterns to extract fields from tuples and records.

This is better style than #1 and #foo, and it means you do not (ever) need to write function arguments.

Instead of a case with one pattern, you can put a pattern directly in a val binding.

Next time we'll see patterns and bindings are much more general.