Goals

- Contrast type synonyms with new types
- See pattern-matching for built-in “one of” types (not really a concept, but important for ML programming) and “each of” types
- Investigate why accumulator-style recursion can be more efficient
Type synonyms

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

```ml
    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.

To contrast, the type a datatype binding introduces is not equivalent to any other type (until possibly a later type binding).
Review: datatypes and pattern-matching

Evaluation rules for datatype bindings and case expressions:

```
datatype t = C1 of t1 | C2 of t2 | ... | Cn of tn
```

Adds constructors Ci where Ci v is a value (and Ci has type ti->t).

```
  case e of p1 => e1 | p2 => e2 | ... | pn => en
```

- Evaluate e to v
- If pi is the first pattern to match v, then result is evaluation of ei in environment extended by the match.
- If C is a constructor of type t1 * ... * tn -> t, then
  C(x1,...,xn) is a pattern that matches C(v1,...,vn) and the match extends the environment with x1 to v1 ... xn to vn.
- Coming soon: many more pattern forms.
Why patterns?

Even without more pattern forms, this design has advantages over functions for “testing and destructing” (e.g., null, hd, and tl):

• easier to check for missing and redundant cases
• more concise syntax by combining “test, destruct, and bind”
• you can easily define testing and destructing in terms of pattern-matching

In fact, case expressions are the preferred way to test variants and extract values from all ML’s “one-of” types, including predefined ones ([] and :: just funny syntax).

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

Teaser: These functions are useful for passing as values
Tuple/record patterns

You can also use patterns to extract fields from tuples and records:
pattern \{f_1=x_1, \ldots, f_n=x_n\} (or \(x_1, \ldots, x_n\)) matches
\{f_1=v_1, \ldots, f_n=v_n\} (or \(v_1, \ldots, v_n\)).

For record-patterns, field-order does not matter.

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

Instead of a case with one pattern, better style is a pattern directly in
a val binding.

Next time: “deep” (i.e., nested) patterns.
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 idioms:
  – 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.

Let’s investigate the evaluation of to_list_1 and to_list_2.
Tail calls

If the result of $f(x)$ is the result of the enclosing function body, then $f(x)$ is a tail call.

More precisely, a tail call is a call in tail position:

- In `fun f(x) = e`, e is in tail position.
- If `if e1 then e2 else e3` is in tail position, then e2 and e3 are in tail position (not e1). (Similar for case).
- If `let b1 ... bn in e end` is in tail position, then e is in tail position (not any binding expressions).
- Function arguments are not in tail position.
- ...

So what?

Why does this matter?

• Implementation takes space proportional to depth of function calls ("call stack" must "remember what to do next")

• But in functional languages, implementation must ensure tail calls eliminate the caller’s space

• Accumulators are a systematic way to make some functions tail recursive

• "Self" tail-recursive is very loop-like because space does not grow.