Where are we

- Features so far: functions, tuples, lists, options, local bindings
- Concepts so far: syntax vs. semantics, environments
- Today’s features: record types, datatypes, pattern-matching
- Today’s concepts: types, constructors/deconstructors, case-coverage
Base types and compound types

Languages typically provide a small number of “built-in” types and ways to build compound types out of simpler ones:

- Base types examples: int, bool
- Type builder examples: tuples, lists, records

Base types clutter a language definition? better to make them libraries when possible?

- ML does this to a remarkable extent (e.g., we will soon define away bool and conditionals)

Good to let programmers bind types to type names, just like we bind values to variables.
Compound-type flavors

Conceptually, just a few ways to build compound types:

1. “Each-of”: A t contains a t1 and a t2
2. “One-of”: A t contains a t1 or a t2
3. “Self-reference”: The definition of t may refer to t

Examples:

- int * bool
- int option
- int list

Fact: A lot of data can be described this way.
Convenient to think of as trees.

(optional) jargon: Product types, sum types, recursive types
User-defined types

There are many reasons to define your own types:

1. Using a tuple with 12 fields is incomprehensible

2. Writing down large types is unpleasant, error-prone; computers can help

3. Large programs can use abstract types to be robust to change
   - A couple weeks ahead

4. So the language doesn’t have to “build in” lists and options and … that aren’t always needed
Datatype

One-of types are less similar across languages

- We’ll discuss OO’s approach to one-of in a few weeks

In ML, we use make a new type with a datatype binding, e.g.:

datatype exp = Const of int  
            | Negate of exp  
            | Add of exp * exp  
            | Mul of exp * exp

Semantics: Extend the environment with four constructors (in part, functions/constants that produce values of type exp)

val e = Add(Const(42), Negate(Mul(Const(7), Const(6))))

So we have a way to build them... what’s missing?
The old way

For lists, we had a way to:

- Test which *variant* a value was (*null*)
- Extract the values from *value-carrying* variants (*hd/tl*)
  - Makes no sense if you have the *wrong* variant

What would this look like for *exp*?
The new way

Rather than add *variant-tests* and *variant-deconstructors*, ML has a *case expression* that uses *pattern-matching*.

In its simplest form, case has one pattern for each constructor in a datatype and binds one variable for each value carried. Example:

```ocaml
fun eval e =
  case e of
    Const i => i
  | Negate e2 => ~ (eval e2)
  | Add(e1,e2) => (eval e1) + (eval e2)
  | Mul(e1,e2) => (eval e1) * (eval e2)

val z = eval Add(Const(42),Negate(Mul(Const(7),Const(6))))
```

What are the typing rules?
What are the evaluation rules?
Type-checking case

In addition to binding local variables and requiring branches to have the same type, the typing rules for case prevent some run-time errors:

- Exhaustiveness: No test can “fail” (a warning)
- Redundancy: No test can be “impossible” (an error)

So far, case gives us what we need to use datatypes:

- A (combined) way to test variants and extract values (deconstruct)
- Powerful enough to define our own tests and deconstructors

In fact, pattern-matching is far more general and elegant:

- Can use it for datatypes already in the top-level environment
- Can use it for *any* type (later)
- Can have deep patterns (later)