

# CSE 341: Programming Languages

Dan Grossman

Spring 2008

Lecture 4— Records, Datatypes

## Where are we

---

- Done features: functions, tuples, lists, local bindings, options
- Done concepts: syntax vs. semantics, environments, mutation-free
- Today features: records, datatypes, case expressions (pattern-matching)
- Today concepts: “One-of” types, constructors/extractors, 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`, `string`, `unit`
- Type builder examples: `tuples`, `lists`, *records* (see code)

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)

# Compound-type flavors

---

Conceptually, just a few ways to build compound types:

1. “Each-of”: A  $t$  contains a  $t_1$  *and* a  $t_2$
2. “One-of”: A  $t$  contains a  $t_1$  *or* a  $t_2$
3. “Self-reference”: The definition of  $t$  may refer to  $t$

Examples:

- `int * bool` (*syntactic sugar* for a record type in ML)
- `int option`
- `int list`

Remarkable: A *lot* of data can be described this way.

(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; we have computers for that
3. Large programs can use *abstract types* to be robust to change
  - A couple weeks ahead
4. So the language doesn't have to “bake in” lists and options and ...

# 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 make a *new type* with a datatype binding, e.g.:

```
datatype mytype = TwoInts of int*int
                | Str of string
                | Pizza
```

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

- TwoInts has type `int*int->mytype`
- Str has type `string->mytype`
- Pizza has type `mytype`.

So we have a way to build them... what's missing?

## The old way

---

For lists and options, we had a way to:

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

What would this look like for `mytype`?

## The new way

---

Rather than add *variant-tests* and *data-extractors* (non-standard jargon), 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:

```
case e of
  TwoInts(i1,i2) => e1
| Str s => e2
| Pizza => e3
```

What are the typing rules?

What are the evaluation rules?

*Patterns* are not types nor expressions (despite syntactic similarity)

## 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)

# Expression trees

---

```
datatype arith_exp = Constant of int
                  | Negate of arith_exp
                  | Add of arith_exp * arith_exp
```

Think of values of type `arith_exp` as trees where nodes are

- Constant with one `int` child
- Negate with one child that can be any `arith_exp` tree.
- Add with two children that can be any `arith_exp` trees.

In general, a type describes a set of values, which are often trees.

One-of types give you different variants for nodes.

Constructors evaluate arguments to values (trees) and create bigger values (i.e., taller trees).

## Where we're going

---

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

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

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

- Can use it for datatypes already in the top-level environment (e.g., lists and options)
- Can use it for *any* type (Wednesday; also tail recursion)
- Can have deep patterns (Friday; also course motivation)