CSE 341, Winter 2008, Lecture 4 Summary

Standard Disclaimer: These comments may prove useful, but certainly are not a complete summary of all the important stuff we did in class. They may make little sense if you missed class, but hopefully will help you organize and process what you have learned.

Programming languages need ways to describe data so that we can write programs that operate on that data. We need *base types* such as `int`, `string`, `bool`, and `unit`. We also need ways to build *compound types* from simpler types, such as `tuples` (pairs are 2-tuples, triples are 3-tuples, etc.), `lists`, and even function types (`->`). This lecture is about record types, which are much like tuples, and `datatype` bindings, which are probably unlike anything you have seen before. Conceptually, this lecture is about describing data in terms of *and* ("each-of" types), *or* ("one-of" types), and *self-reference* ("recursive" types).

Syntactically, an ML record is constructed via `{f1=e1, ..., f2=en}` where `f1`, ... `fn` are *field names* (sequences of letters) and `e1`, ..., `en` are expressions. Exactly like tuples, the evaluation rules are to evaluate the expressions to values and a record with values in all fields is itself a value. The only difference from tuples is that the order of the fields does not matter; we use field names instead of "position" to determine which field is which. Unlike in many languages, we do not need any sort of type declaration before we use a record; we can just use whatever field names we want. The type of a record describes what fields it has and what types those fields have. Again, field order does not matter; in fact, the read-eval-print loop always alphabetizes field names before it prints a type. To extract a field `foo` from a record, we can use `#foo e` where `e` evaluates to a record.

Tuples are, in fact, so much like records that they actually *are* records. When you write `(7,"hi")`, that is exactly the same as writing `{1=7, 2="hi"}`. The type `int*string` is just another way of writing `{1:int, 2:string}`. Now, using the tuple syntax is cleaner and better style, but it is still an elegant language design to have it really just be another language feature (records). That way, the designer and implementor of the language has less work to do; we can completely define how tuples behave by explaining how they are really just a different syntax for particular records. This is our first example of the idea of *syntactic sugar*, a piece of the language that is just a prettier way of writing something already in the language. It is syntactic because it is just a different way of writing something, and it is sugar because it makes the language sweeter.

Let us now return to building compound types in terms of "each of", "one of", and "self reference". Each-of types are often the simplest for people to understand. They are like records, or Java classes with only fields. Given types `t1`, `t2`, `t3`, we make some new type `t` that "has" each of `t1`, `t2`, *and* `t3`. "One of" types are just as important; it is very common to have a type `t` that "has" either `t1`, `t2`, *or* `t3`. As we will discuss near the end of the course, in object-oriented languages like Java, one uses subclassing to implement "one of" types. Finally, self-reference is necessary to describe recursive data structures like lists and trees.

We have actually seen examples of each kind of compound type in earlier lectures. `int * bool` is an each-of type; each value of this type has an `int` and a `bool`. `int option` is a one-of type; each value of this type either has an `int` or it does not. `int list` is a compound type using all three notions: a value of type `int list` has either no data (the empty list) *or* it has an `int` *and* another `int list` (notice the self-reference).

In ML, you use a `datatype` binding to create a new one-of type that comes with *constructors* and *patterns* for building and accessing values of the new type, respectively. A silly example of such a binding is:

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

This binding creates a new type `mytype` with 3 *variants*. A value of type `mytype` is created in one of 3 ways: (1) By using the constructor `TwoInts` on a pair of ints, (2) By using the constructor `Str` on a string, or (3) By using the constructor `Pizza` on nothing. In this sense, constructors are functions (if they take arguments) or constants (if they do not). In our example, the datatype binding adds to the environment/context: (1) `TwoInts` of type `int * int -> mytype`, (2) `Str` of type `string -> mytype`, and (3) `Pizza` of type `mytype`. As with any other function, an argument to a constructor can be any expression of the correct type.

So constructors give us a way to make values of a datatype such as `mytype`, but we also need a way to use such values after we make them. Doing so requires two kinds of operations: (1) tests to see which variant

we have and (2) operations that extract the values that a particular variant has. We have already seen such operations for options and lists. The functions `null` and `isSome` are variant tests. The functions `valOf`, `hd`, and `tl` are the value extractors. Notice they raise exceptions if applied to a value of an unexpected variant.

A `datatype` binding does not directly create variant tests and value extractors. Instead, ML uses *pattern matching*, an elegant and convenient feature that combines the variant test and value extraction. By combining them, the type-checker can check that you do not forget any cases or duplicate any cases.

Pattern-matching is done with a case-expression. The syntax is

```
case e0 of
  p1 => e1
| ...
| pn => en
```

where `e0`, `e1`, ..., `en` are expressions and `p1`, ..., `pn` are *patterns*. We have not seen patterns before. They look a lot like expressions syntactically, but they are more restrictive and their semantics is very different.

Here is a function using pattern matching and our previous datatype example:

```
fun f2 x =  (* f2 has type mytype -> int *)
    case x of
        Pizza => 3
      | TwoInts(i1,i2) => i1 + i2
      | Str s => 8
```

The semantics of a case-expression is to evaluate the expression between the `case` and `of` to some value `v` and then proceed through the patterns in order, finding the first one that *matches*. For now, a pattern-matches if the constructor in it is the same as the constructor for the value `v`. For example, if the value if `TwoInts(7,9)`, then the first pattern would not match and the second one would. We evaluate the expression to the right of the matching pattern (on the other side of the `=>`) and that result is the result of the whole case-expression. The other expressions are not evaluated.

We have not yet explained the value-extraction part of pattern-matching. Since `TwoInts` has two values it "carries", a pattern for it can (and, for now, must) use two variables (the `(i1,i2)`). As part of matching, the corresponding parts of the value `v` (continuing our example, the 7 and the 9) are bound to `i1` and `i2` in the environment used to evaluate the corresponding right-hand side (the `i1+i2`). In this sense, pattern-matching is like a let-expression, it binds variables in a local scope.

It turns out case-expressions are a more general and powerful form of conditional expressions. Like `if e1 then e2 else e3`, they use their first expression to decide which other expression is used to produce the answer. Type-checking is also similar: the different branches must all have the same type and that is the type of the whole expression. The key additional power is that patterns can bind variables, extending the context/environment for the corresponding branch.

In fact, conditional expressions are really just syntactic sugar for case-expressions and a predefined datatype. Among the bindings evaluated before your program starts is:

```
datatype bool = true | false
```

We can then treat `if e1 then e2 else e3` as a syntactic shortcut for
`case e1 of true => e2 | false => e3`.

As we will see in the next two lectures, pattern-matching can be used for much more than just datatype values. However, its basic semantics of comparing a pattern against a value and binding variables for the corresponding branch will not change.

Let us consider two less silly examples of datatype bindings. This first one defines identities as either a social-security number or (presumably for when someone does not have a number), a first name, optional middle name, and last name:

```
datatype id = SSN of int
            | Name of string * (string option) * string
```

Whenever a data definition has a clear "or" in it, datatypes are the way to go. Novice programmers often abuse "each of" types when they want "one of types". For example you might see bad code like this (in any language; we just use ML for the example):

```
(* If ssn is -1, then the other fields are the name, otherwise ssn is
   the identity and the other fields should be ignored *)
type bad_id = {ssn: int, first : string, middle : string option, last : string}
```

On the other hand, if a name record is supposed to have an optional social-security number *and* a (non-optional) name, then an "each of" type is the right thing:

```
type name_record = {ssn: int option, first : string, middle : string option, last : string}
```

Our second example is a data definition for arithmetic expressions containing constants, negations, and additions.

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

Thanks to the self-reference, what this data definition really describes is a *tree* where the leaves are integers and the internal nodes are either negations with one child or additions with two children. We can write a function that takes an `exp` and evaluates it:

```
fun eval e =
    case e of
        Constant i => i
      | Negate e2  => ~ (eval e2)
      | Add(e1,e2) => (eval e1) + (eval e2)
```

So this function call evaluates to 15:

```
eval (Add (Constant 19, Negate (Constant 4)))
```

Notice how constructors are just functions that we call with other expressions (often other values built from constructors).

For practice, other functions you could write that process `exp` values could compute:

- The largest constant in an expression.
- A list of all the constants in an expression (use list append).
- How many addition expressions are in an expression.