CSci 4223
Principles of Programming Languages

Lecture 5
Prof. Clarkson
Spring 2013

Review
• Features learned: functions, tuples, lists, local bindings, options, records
• Compound types:
  – Each-of
  – One-of
  – Self-reference
• Syntactic sugar: syntax, no (new) semantics
  – Tuples are records
  – andalso, orelse are if expressions
• Today, two new features:
  – Datatype bindings
  – Case expressions

Datatype bindings
• New sort of binding (variable binding, function binding):
  - \texttt{datatype mybool = Mytrue | Myfalse}
    - Creates a one-of type named mybool
    - Creates two constructors named Mytrue and Myfalse
    - Those are also values of type mybool
  
  - In fact, that’s exactly how Booleans are defined in ML:
    - \texttt{datatype bool = false | true}
      - Compiler “includes” this definition before your program

Datatype for Days

(* similar to an enum in Java or C *)
\[
\texttt{datatype day = Sun | Mon | \ldots | Fri | Sat}
\]

fun day_to_int (d : day) =
  if d=Sun then 1
  else if d=Mon then 2
  else if d=Tue then 3
  else if d=Wed then 4
  else if d=Thu then 5
  else if d=Fri then 6
  else (* d=Sat *) 7

Datatype for Days

(* similar to an enum in Java or C *)
\[
\texttt{datatype day = Sun | Mon | \ldots | Fri | Sat}
\]

fun day_to_int (d : day) =
  case d of
    Sun => 1
  | Mon => 2
  | Tue => 3
  | Wed => 4
  | Thu => 5
  | Fri => 6
  | Sat => 7

Datatype for Days

(* similar to an enum in Java or C *)
\[
\texttt{datatype day = Sun | Mon | \ldots | Fri | Sat}
\]

fun int_to_day (i : int) =
  case i of
    1 => SOME Sun
  | 2 => SOME Mon
  | 3 => SOME Tue
  | 4 => SOME Wed
  | 5 => SOME Thu
  | 6 => SOME Fri
  | 7 => SOME Sat
  | _ => NONE
But wait, there's more...

A "strange" (?) and totally awesome (!) way to make one-of types:

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

- Adds a new type `mytype` to the environment
- Adds constructors to the environment: `TwoInts`, `Str`, and `Pizza`
- A constructor is (among other things), a function that makes values of the new type (or is a value of the new type):
  - `TwoInts`: `int * int -> mytype`
  - `Str`: `string -> mytype`
  - `Pizza`: `mytype`

Any value of type `mytype` is made from one of the constructors

The value contains:
- A "tag" for "which constructor" (e.g., `TwoInts`)
- The corresponding data (e.g., `(7,9)`)

Examples:
- `TwoInts(3+4,5+4)` evaluates to `TwoInts(7,9)`
- `Str(if true then "hi" else "bye")` evaluates to `Str("hi")`
- `Pizza` is a value

Using them

So we know how to build `datatype` values; need to access them

There are two aspects to accessing a `datatype` value:
1. Check what variant it is (what constructor made it)
2. Extract the data (if that variant has any)

Notice how our other one-of types used functions for this:
- `null` and `isSome` check variants
- `hd`, `tl`, and `valOf` extract data (raise exception on wrong variant)

CT could have done the same for `datatype` bindings:
- For example, functions like "isStr" and "getStrData"
- Instead it did something better

Case

ML combines the two aspects of accessing a one-of value with a case expression and pattern-matching

- Pattern-matching much more general/powerful (next week)

Example:
```
fun f x = (* f has type mytype -> int *)
  case x of
    Pizza => 3
  | TwoInts(i1,i2) => i1+i2
  | Str s => String.size s
```

Why this way is better

0. You can use pattern-matching to write your own testing and data-extractions functions if you must
  - But don't do that on your homework
1. You can't forget a case (inexhaustive pattern-match a warning)
2. You can't duplicate a case (a type-checking error)
3. You won't forget to test the variant correctly and get an exception (like `hd [[]]`)
4. Pattern-matching can be generalized and made more powerful, leading to elegant and concise code

Patterns

In general the syntax is:
```
case e0 of
  p1 => e1
  | p2 => e2
  ...
  | pn => en
```

For today, each pattern is a constructor name followed by the right number of variables (i.e., `C x C x C(x,y) y` or `...`)
- Syntactically most patterns (all today) look like expressions
- But patterns are not expressions
  - We do not evaluate them
  - We see if the result of `e0` matches them
1/30/13

Useful examples

That last datatype was silly...

- Enumerations, including containing other data

```plaintext
datatype suit = Club | Diamond | Heart | Spade
datatype rank = Jack | Queen | King | Ace | Num of int
```

- Alternative ways of representing data

```plaintext
(* Every student either has an id number * or (temporarily) is identified by name. *)
datatype student_id = IdNum of int | FullName of string
```

That said...

If every person has a name and optionally an id number, then each-of is what you mean:

```plaintext
{ id_num : int option, full_name : string }
```

Don’t do this

```plaintext
(* If id_num is ~1, then use the name field, * otherwise ignore the name field. *)
{ id_num : int, full_name : string }
```

- Why is this bad?
  - Using each-of where you mean one-of
  - Compiler can’t ensure only one field is in use
  - Compiler can’t ensure you test id_num at every use
  - Makes code harder to understand

- Why do programmers do it anyway?
  - Bad training...they didn’t study PL! ;)
  - Forced to use languages that make one-of types inconvenient

Datatype for Binary Trees

```plaintext
datatype tree = Leaf of int | Node of tree * int * tree
def val t = Node(Node(Leaf 3, 2, Leaf 4), 1, Node(Leaf 6, 5, Leaf 7))
```

You’ve seen this before in CS 1112...

```plaintext
class TreeNode {
  int data;
  TreeNode left;
  TreeNode right;
}
class BinaryTreeNode {
  TreeNode root;
  public boolean contains (int k) {
    return recursiveSearch(root, k);
  }
  boolean static recursiveSearch(TreeNode node, int k) {
    if (node == null) {
      return false;
    }
    return (k == node.data ||
    recursiveSearch(node.left, k) ||
    recursiveSearch(node.right, k));
  }
}
```

And after CS 2113 you could do better...

```plaintext
interface Tree {
  boolean contains (int k);
}
class Leaf implements Tree {
  int data;
  boolean contains (int k) {
    return k == data;
  }
}
class Node extends Leaf {
  Tree left;
  Tree right;
  boolean contains (int k) {
    return (k == data | left.contains(k) | right.contains(k));
  }
}
And now you can do MUCH better...

```ml
datatype tree =
  Leaf of int
| Node of tree * int * tree

fun contains (t:tree, i:int) =
  case t of
    Leaf() => i=j
  | Node(l,j,r) => i=j orelse contains(l,i) orelse contains(r,i)
```

- Simple, concise, elegant, beautiful

Beauty is our Business

When we recognize the battle against chaos, mess and unmastered complexity as one of computing science's major callings, we must admit that "Beauty is our Business."

– Edsger W. Dijkstra (d. 2002)

7th recipient of the Turing Award

"The precious gift that this Turing Award acknowledges is Dijkstra's style: his eloquent insistence and practical demonstration that programs should be composed correctly, and not just debugged into correctness...We have come to value good programs in much the same way as we value good literature. And at the center of this movement, creating and reflecting patterns no less beautiful than useful, stands E.W. Dijkstra."

Expression Trees

Represent ML expressions using ML:

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

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

Recursion

Not surprising:

Functions over recursive datatypes are usually recursive