Review

Datatype bindings and pattern-matching so far:

\[
\text{datatype } t = \text{C1 of } t_1 \mid \text{C2 of } t_2 \mid \ldots \mid \text{Cn of } t_n
\]

- Adds type \( t \) and constructors \( \text{Ci} \) of type \( t_i \rightarrow t \)
- \( \text{Ci v} \) is a value

\[
\text{case } e \text{ of } p_1 \Rightarrow e_1 \mid p_2 \Rightarrow e_2 \mid \ldots \mid p_n \Rightarrow e_n
\]

- Evaluate \( e \) to a value
- If \( p_i \) is the first pattern to match the value, then result is evaluation of \( e_i \) in environment extended by the match
- Pattern \( \text{Ci}(v_1, \ldots, v_n) \) matches value \( \text{Ci}(v_1, \ldots, v_n) \) and extends the environment with \( x_1 \) to \( v_1 \), \( x_2 \) to \( v_2 \), etc.
- This lecture: many more kinds of patterns and ways to use them

Recursive datatypes

Datatype bindings can describe recursive structures
- Arithmetic expressions from last lecture
- Linked lists, for example:

\[
\text{datatype } \text{my_int_list} = \text{Empty} \mid \text{Cons of } \text{int} \times \text{my_int_list}
\]

\[
\text{val } x = \text{Cons}(4, \text{Cons}(23, \text{Cons}(2008, \text{Empty})))
\]

\[
\text{fun append_my_list } (xs, ys) =
\begin{align*}
\text{case } xs \text{ of} \\
\text{Empty} & \Rightarrow ys \\
\text{Cons}(x, xs') & \Rightarrow \text{Cons}(x, \text{append_my_list}(xs', ys))
\end{align*}
\]

Options are datatypes

Options are just a predefined datatype binding
- \text{NONE} and \text{SOME} are constructors, not just functions
- So use pattern-matching not \text{isSome} and \text{valOf}

\[
\text{fun inc_or_zero intoption =}
\begin{align*}
\text{case intoption of} \\
\text{NONE} & \Rightarrow 0 \\
\text{SOME } i & \Rightarrow i+1
\end{align*}
\]

Lists are datatypes

Don’t use \text{hd}, \text{tl}, or \text{null} either
- \([\ ]\) and \(::\) are constructors too
- (strange syntax, particularly \text{infix})

\[
\text{fun sum_list intlist =}
\begin{align*}
\text{case intlist of} \\
[\ ] & \Rightarrow 0 \\
\text{head::tail} & \Rightarrow \text{head + sum_list tail}
\end{align*}
\]

\[
\text{fun append } (xs, ys) =
\begin{align*}
\text{case } xs \text{ of} \\
[\ ] & \Rightarrow ys \\
x::xs' & \Rightarrow x :: \text{append}(xs', ys)
\end{align*}
\]

Why pattern-matching

- Pattern-matching is better for options and lists for the same reasons as for all datatypes
  - No missing cases, no exceptions for wrong variant, etc.
- We just learned the other way first for pedagogy
- So why are \text{null} and \text{tl} predefined then?
  - For passing as arguments to other functions (next week)
  - Because sometimes they’re really convenient
  - But not a big deal: could define them yourself with case
Each-of types

So far have used pattern-matching for one of types because we needed a way to access the values.

Pattern matching also works for records and tuples:
- The pattern \((x_1, \ldots, x_n)\) matches the tuple value \((v_1, \ldots, v_n)\)
- The pattern \(\{f_1=x_1, \ldots, f_n=x_n\}\) matches the record value \(\{f_1=v_1, \ldots, f_n=v_n\}\)
  (and fields can be reordered)

Example

This is poor style, but based on what I told you so far, the only way to use patterns:
- Works but poor style to have one-branch cases

\[
\text{fun sum_triple \ triple =} \\
\text{case \ triple \ of} \\
\text{\ (x, y, z) => x + y + z}
\]

\[
\text{fun sum_stooges \ stooges =} \\
\text{case stooges \ of} \\
\text{\ \{larry=x, moe=y, curly=z\} => x + y + z}
\]

Better example

This is reasonable style
- Though we will improve it one more time next
- Semantically identical to one-branch case expressions

\[
\text{fun sum_triple \ triple =} \\
\text{let \ val \ (x, y, z) = triple \ in} \\
\text{\ x + y + z \ end}
\]

\[
\text{fun sum_stooges \ stooges =} \\
\text{let \ val \ \{larry=x, moe=y, curly=z\} = stooges \ in} \\
\text{\ x + y + z \ end}
\]

A new way to go

- For homework 2:
  - Do not use the \# character
  - You won’t need to write down any explicit types
- These are related
  - Type-checker can use patterns to figure out the types
  - With just \#foo it can’t “guess what other fields”

Function-argument patterns

A function argument can also be a pattern
- Match against the argument in a function call

\[
\text{fun \ f \ p = e}
\]

Examples:

\[
\text{fun \ sum_triple \ (x, y, z) =} \\
\text{\ x + y + z}
\]

\[
\text{fun \ sum_stooges \ \{larry=x, moe=y, curly=z\} =} \\
\text{\ x + y + z}
\]
A function that takes one triple of type `int*int*int` and returns an `int` that is their sum:

```ml
fun sum_triple (x, y, z) = x + y + z
```

A function that takes three `int` arguments and returns an `int` that is their sum:

```ml
fun sum_triple (x, y, z) = x + y + z
```

See the difference? (Me neither.) 😊

---

**The truth about functions**

- In ML, every function takes exactly one argument (*)
- What we call multi-argument functions are just functions taking one tuple argument, implemented with a tuple pattern in the function binding
  - Elegant and flexible language design
- Enables cute and useful things you can’t do in Java, e.g.,

```ml
fun rotate_left (x, y, z) = (y, z, x)
fun rotate_right t = rotate_left(rotate_left t)
```

* “Zero arguments” is the unit pattern () matching the unit value ()

---

**One-of types in function bindings**

As a matter of taste, I personally have never loved this syntax, but others love it and you’re welcome to use it:

```ml
fun f p1 = e1
  | f p2 = e2
  ...
  | f pn = en
```

As a matter of semantics, it’s syntactic sugar for:

```ml
fun f x = el
  case x of
    p1 => e1
  | p2 => e2
  ...
```

---

**More sugar**

By the way, conditionals are just a predefined datatype and if-expressions are just syntactic sugar for case expressions

```ml
datatype bool = true | false
if e1 then e2 else e3
```

```ml
case e1 of true => e2 | false => e3
```

---

**Nested patterns**

- We can nest patterns as deep as we want
  - Just like we can nest expressions as deep as we want
  - Often avoids hard-to-read, wordy nested case expressions
- So the full meaning of pattern-matching is to compare a pattern against a value for the “same shape” and bind variables to the “right parts”
  - More precise recursive definition coming after examples
- Examples:
  - Pattern `a::b::c::d` matches all lists with >= 3 elements
  - Pattern `a::b::c::[]` matches all lists with 3 elements
  - Pattern `((a,b), (c,d))::e` matches all non-empty lists of pairs of pairs

---

**Useful example: zip/unzip 3 lists**

```ml
fun zip3 lists =
  case lists of
    ([],[],[]) => []
  | (hd1::tl1,hd2::tl2,hd3::tl3) =>
      (hd1,hd2,hd3)::zip3(tl1,tl2,tl3)
  | _ => raise ListLengthMismatch

fun unzip3 triples =
  case triples of
    [] => ([],[],[])
  | (a,b,c)::tl =>
      let val (l1, l2, l3) = unzip3 tl
      in
        (a::l1,b::l2,c::l3)
      end
```

More examples in the code for the lecture
(Most of) the full definition

The semantics for pattern-matching takes a pattern $p$ and a value $v$ and decides (1) does it match and (2) if so, what variable bindings are introduced.

Since patterns can nest, the definition is elegantly recursive, with a separate rule for each kind of pattern. Some of the rules:

- If $p$ is a variable $x$, the match succeeds and $x$ is bound to $v$
- If $p$ is _, the match succeeds and no bindings are introduced
- If $p$ is $(p_1, \ldots, p_n)$ and $v$ is $(v_1, \ldots, v_n)$, the match succeeds if and only if $p_1$ matches $v_1$, ..., $p_n$ matches $v_n$. The bindings are the union of all bindings from the submatches
- If $p$ is $C\, p_1$, the match succeeds if $v$ is $C\, v_1$ (i.e., the same constructor) and $p_1$ matches $v_1$. The bindings are the bindings from the submatch.
- ... (there are several other similar forms of patterns)