Let bindings

Motivation: Functions without local variables can be poor style and/or really inefficient.

Syntax: \texttt{let } b_1 \ b_2 \ldots \ b_n \texttt{ in e end} where each \texttt{b_i} is a \textit{binding}.

Typing rules: Type-check each \texttt{b_i} and \texttt{e} in context including previous bindings. Type of whole expression is type of \texttt{e}.

Evaluation rules: Evaluate each \texttt{b_i} and \texttt{e} in environment including previous bindings. Value of whole expression is result of evaluating \texttt{e}.

Elegant design worth repeating:

- Let-expressions can appear anywhere an expression can.
- Let-expressions can have any kind of binding.
  - Local functions can refer to any bindings \textit{in scope}.

More than style

Exercise: hand-evaluate \texttt{bad\_max} and \texttt{good\_max} for lists $\langle 1, 2 \rangle$, $\langle 1, 2, 3 \rangle$, and $\langle 3, 2, 1 \rangle$.

Extra Credit Exercise: As a function of $n$, how long will it take to calculate

- $\texttt{bad\_max}([1, 2, \ldots, n])$?
- $\texttt{bad\_max}([n, n-1, \ldots, 1])$?

Summary and general pattern

Major progress: recursive functions, pairs, lists, let-expressions

Each has a syntax, typing rules, evaluation rules.

Functions, pairs, and lists are very different, but we can describe them in the same way:

- How do you create values? (function definition, pair expressions, empty-list and ::)
- How do you use values? (function application, #1 and #2, \texttt{null}, \texttt{hd}, and \texttt{tl})
Boolean operations

In ML the “and” and “or” operations are named andalso and orelse.

Example:

```ml
val x = 19;
val y = 0;
val z = if x>2 andalso y>2 then 3.0 else 4.0;
val w = if x>2 orelse y>2 then 3.0 else 4.0;
```

Patterns – Sneak Preview

In ML patterns provide a useful way of defining functions, often more readable than using conditionals. (You can use them for HW 1 if you like!)

```ml
(* return the result of reversing a list *)
fun reverse(xs) = if xs=[] then []
    else reverse(tl(xs)) @ [hd(xs)]

(* definition of reverse using patterns to test for
    the empty list, and also to pick the list apart *)
fun preverse([]) = []
| preverse(x::xs) = preverse(xs) @ [x]
```

Options

Options provide a way of representing a value that might or might not be present.

- Create a `option` with `NONE` or `SOME e` where `e` has type `t`.
- Use a `t` `option` with `isSome` and `valOf`

Why not just use a list with zero or one element? An interesting style trade-off:

- Options better express purpose, enforce invariants on callers, maybe faster.
- But cannot use functions on options with lists that are already constructed for some other purpose.