MIDTERM 1
In class, Thursday, Oct 10
last time

- functions as values
- higher-order functions
- mapping and folding
map/foldr fusion

For all suitably typed $f$, $g$, $z$ and $L$

$$\text{foldr } g \ z \ (\text{map } f \ L)$$

$$= \text{foldr } (\text{fn } (x, y) \Rightarrow g(f \ x, y)) \ z \ L$$

*a map followed by a fold can be replaced by a single fold*

*avoids building an extra list*
a map/foldr example

\[(x_1,y_1),\ldots,(x_n,y_n)\] → \[x_1 y_1,\ldots,x_n y_n\] → map (op * ) → \[x_1 y_1,\ldots,x_n y_n\] → foldr (op + ) 42 → \[x_1 y_1,\ldots,x_n y_n\] → foldr (fn ((x,y),u) => x*y + u) 42 → \[x_1 y_1,\ldots,x_n y_n\] → \[x_1 y_1 + \ldots + x_n y_n + 42\]
foldr and @

For all suitably typed g, z, L₁ and L₂

\[ \text{foldr } g \ z \ (L₁ @ L₂) = \text{foldr } g \ (\text{foldr } g \ z \ L₂) \ L₁ \]

Proof: induction on length of L₁

This shows the right-left combination order used by foldr
**foldl**

```
fun foldl F z [ ] = z
    | foldl F z (x::L) = foldl F (F(x, z)) L
```

```
foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

```
foldl F z [x₁,...,xₙ] = F(xₙ, F(xₙ₋₁,..., F(x₁,z)...))
```

(combines from left to right)

**foldl is tail recursive, unlike foldr**
using foldl

fun sum L = foldl (op +) 0 L

fun largest (x::R) = foldl Real.max x R

foldl (op ^) "are belong to us"
  ["all ","your ","base "]

  = "base your all are belong to us"
In general, when is
\( \text{foldr} \ g = \text{foldl} \ g \)?

\[
\begin{align*}
\text{foldr} \ (\text{op} \ @) \ [ ] \ [[1,2], [ ], [3,4]] &= [1,2,3,4] \\
\text{foldl} \ (\text{op} \ @) \ [ ] \ [[1,2], [ ], [3,4]] &= [3,4,1,2]
\end{align*}
\]
foldl blend

[ cherry, banana, ice cream ]

= Contrast with foldr
foldr vs foldl

Suppose for all x, y and u
\[ g(x, g(y, u)) = g(y, g(x, u)). \]

Then for all L and z
\[ \text{foldr } g \ z \ L = \text{foldl } g \ z \ L \]

\[ \text{foldr } g = \text{foldl } g \]

(op +) and (op ^) have this property
(op @) and (op ^) do not

this holds when \( g \) is associative and commutative
foldr vs foldl

For all suitably typed $g$, $z$ and $L$

$$\text{foldr } g \ z \ L = \text{foldl } g \ z \ (\text{rev } L)$$

Proof: by induction on length of $L$
foldl and @

For all suitably typed g, z, L₁ and L₂

\[ \text{foldl } g \ z \ (L₁ \@ L₂) \]
\[ = \text{foldl } g \ (\text{foldl } g \ z \ L₁) \ L₂ \]

Proof: by induction on length of L₁

This shows the left-right combination order used by foldl

\[ \text{foldr } g \ z \ (L₁ \@ L₂) \]
\[ = \text{foldr } g \ (\text{foldr } g \ z \ L₂) \ L₁ \]
invariance

Let $g : t_1 \times t_2 \rightarrow t_2$ and $p : t_2 \rightarrow \text{bool}$ be total functions.

Say $g$ preserves $p$ if for all $z : t_2$ and $x : t_1$,

$$p(z) \text{ implies } p(g(x,z))$$

**Invariance Theorem**

If $g$ preserves $p$, and $p(z)$ holds, then for all $L$, $p(\text{foldr } g \ z \ L)$ holds.

Similarly, for foldl

$$\text{ins : int } \times \text{ int list } \rightarrow \text{ int list}$$

preserves $\text{sorted}$

... so foldr ins $[\ ]$ $L$ is sorted
A case study

Developing an *abstract* and *general* solution to a *family* of problems

The benefits of *polymorphic types* and *higher-order functions*

Advantages of tasteful *currying*
A type of data, with a comparison function. Want to sort lists and trees of data.
examples

<table>
<thead>
<tr>
<th>type</th>
<th>comparison</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>usual</td>
<td>3 &lt; 4</td>
</tr>
<tr>
<td>int * int</td>
<td>lexicographic</td>
<td>(3,2) &lt; (4,1), (3,2) &lt; (3,3)</td>
</tr>
<tr>
<td>string</td>
<td>dictionary</td>
<td>“car” &lt; “card” &lt; “kardashian”</td>
</tr>
</tbody>
</table>

obviously in ML we can’t use `<` for all of these at once!
A **sorting algorithm** ... puts elements of a **list** in a certain **order**. The most-used orders are numerical ... and **lexicographical**.

Efficient **sorting** is important ... for optimizing **search** and **merge** algorithms and for producing human-readable output.
A **comparison** for type $t$ is a *total* function

$$\text{cmp} : t \times t \rightarrow \text{order}$$

such that

$$\text{cmp}(x,y) = \text{LESS} \quad \text{iff} \quad \text{cmp}(y,x) = \text{GREATER}$$

$$\text{cmp}(x,y) = \text{EQUAL} \quad \text{iff} \quad \text{cmp}(y,x) = \text{EQUAL}$$

and

$$\text{cmp}(x,y) = \text{LESS} \land \text{cmp}(y,z) \not\equiv \text{GREATER} \implies \text{cmp}(x,z) = \text{LESS}$$

$$\text{cmp}(x,y) = \text{GREATER} \land \text{cmp}(y,z) \not\equiv \text{LESS} \implies \text{cmp}(x,z) = \text{GREATER}$$

$$\text{cmp}(x,y) = \text{EQUAL} \land \text{cmp}(y,z) = \text{EQUAL} \implies \text{cmp}(x,z) = \text{EQUAL}$$

“the obvious properties”
For all values \( x, y : \text{int} \)

\[
\text{Int.compare}(x, y) = \text{LESS} \quad \text{if } x < y \\
= \text{EQUAL} \quad \text{if } x = y \\
= \text{GREATER} \quad \text{if } x > y
\]

\( x < y \) iff \( y > x \)

\( x < y \) and \( y \leq z \) implies \( x < z \)

etc.

This is a comparison function!
pairs of integers

lexcompare : (int * int) * (int * int) -> order

fun lexcompare(((x₁, y₁), (x₂, y₂))) =
  case Int.compare(x₁, x₂) of
    LESS => LESS
    | GREATER => GREATER
    |      EQUAL => Int.compare(y₁, y₂)

This is a comparison function
flipping

flip: ('a * 'a -> order) -> ('a * 'a -> order)

fun flip cmp (x, y) = cmp (y, x)

If `cmp` is a comparison, so is `flip(cmp)`.

(flip Int.compare) (2,3) = GREATER
**lex on pairs**

\[
\text{lex : ('a * 'a -> order) * ('b * 'b -> order) -> ('a * 'b) * ('a * 'b) -> order)}
\]

\[
\text{fun lex (cmp1, cmp2) ((x1, y1), (x2, y2)) =}
\]

\[
\text{case cmp1(x1, x2) of}
\]

\[
\text{LESS => LESS}
\]

| GREATER => GREATER
| EQUAL => cmp2(y1, y2)

If \(\text{cmp1}\) is a comparison for \(t_1\) and \(\text{cmp2}\) is a comparison for \(t_2\) then \(\text{lex(cmp1, cmp2)}\) is a comparison for \(t_1 * t_2\)

\[
\text{lexcompare = lex(Int.compare, Int.compare)}
\]

\[
: (\text{int * int}) * (\text{int * int}) -> \text{order}
\]
Define a function

\[ \text{listlex} : (\text{'a} \times \text{'a}) \to (\text{'a list} \times \text{'a list}) \to \text{order} \]

such that

\text{when cmp is a comparison for } t, \text{ listlex cmp is a comparison for } t \text{ list}

Hint:

\[
\begin{align*}
\text{listlex cmp ([ ], [ ])} &= \text{EQUAL} \\
\text{listlex cmp ([ ], y::R)} &= \text{LESS} \\
\text{listlex cmp (x::L, [ ])} &= \text{GREATER} \\
\text{listlex cmp (x::L, y::R)} &= \text{cmp(x,y)} \quad \text{if cmp(x,y)\neq\text{EQUAL}} \\
\text{listlex cmp (x::L, y::R)} &= \text{listlex cmp (L, R)} \quad \text{if cmp(x,y)=\text{EQUAL}}.
\end{align*}
\]
less, eq, lesseq

less, eq, lesseq : ('a * 'a -> order) -> ('a * 'a -> bool)

fun less cmp (x, y) = (cmp(x, y) = LESS)

fun eq cmp (x, y) = (cmp(x, y) = EQUAL)

fun lesseq cmp (x, y) = (cmp(x, y) <> GREATER)
sorted

sorted : ('a * 'a -> order) -> 'a list -> bool

fun sorted cmp [ ] = true
| sorted cmp [x] = true
| sorted cmp (x::y::L) =
  case cmp(x, y) of
  GREATER => false
| _             => sorted cmp (y::L)

L is cmp-sorted iff
sorted cmp L = true

Every element is cmp-≤ all later elements
**insertion**

\[
\text{ins : (}'a \times 'a -> \text{order}) -> (}'a \times 'a \text{ list}) -> 'a \text{ list}
\]

\[
\text{fun ins cmp (x, [ ])} = [x]
\]

\[
\begin{align*}
\text{ins cmp (x, y::L)} &= \\
\text{case cmp(x, y) of} & \\
\text{GREATER} & \Rightarrow y :: \text{ins cmp (x, L)} \\
\text{_} & \Rightarrow x :: (y::L)
\end{align*}
\]

If \text{cmp} is a comparison and \text{L} is \text{cmp-sorted}, then

\[
\text{ins cmp (x, L)} = \text{a cmp-sorted permutation of x::L}.
\]
note

If

cmp(x,y) = EQUAL,

then

ins cmp (x, y::L) = x::y::L

(Remember this!)
insertion
(uncurried version)

ins : ('a * 'a -> order) ★ ('a * 'a list) -> 'a list

fun ins (cmp, (x, [ ])) = [x]
| ins (cmp, (x, y::L)) =
  case cmp(x, y) of
  GREATER => y :: ins (cmp, (x, L))
  _        => x :: (y::L)
why curry?

We prefer the *curried* insertion function

\[
\text{ins} \colon (\alpha \times \alpha \to \text{order}) \to (\alpha \times \alpha \text{ list}) \to \alpha \text{ list}
\]

to the *uncurried* version

\[
\text{ins} \colon (\alpha \times \alpha \to \text{order}) \times (\alpha \times \alpha \text{ list}) \to \alpha \text{ list}
\]

The curried function can be partially applied...

\[
\text{ins Int.compare} \colon \text{int} \times \text{int list} \to \text{int list}
\]

\[
\text{ins String.compare} \colon \text{string} \times \text{string list} \to \text{string list}
\]
ordering curry

We chose

\[ \text{ins : (}'a \times 'a -> \text{order}) \rightarrow (]'a \times 'a \text{ list}) \rightarrow 'a \text{ list} \]

Why not

\[ \text{ins : (}'a \times 'a \text{ list}) \rightarrow (]'a \times 'a -> \text{order}) \rightarrow 'a \text{ list} ? \]

Why not curry again…

\[ \text{ins : (}'a \times 'a -> \text{order}) \rightarrow 'a \rightarrow 'a \text{ list} \rightarrow 'a \text{ list} ? \]
folding

foldl, foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

• Standard ways to combine a list

fun foldl g z [ ] = z
  | foldl g z (x::L) = foldl g (g(x,z)) L

fun foldr g z [ ] = z
  | foldr g z (x::L) = g(x, foldr g z L)
**isortl and isortr**

isortl, isortr : ('a * 'a -> order) -> 'a list -> 'a list

```ml
fun isortl cmp L = foldl (ins cmp) [ ] L
fun isortr cmp L = foldr (ins cmp) [ ] L
```

If cmp is a comparison, then for all lists L,

- **isortl** cmp L = a cmp-sorted permutation of L
- **isortr** cmp L = a cmp-sorted permutation of L
connection

• Int.compare : int * int -> order  (usual <)

• isortl Int.compare = isortr Int.compare

  = isort : int list -> int list
  (as defined previously)

Follows from the (proven) specs,

since an integer list

has only ONE <-sorted permutation
examples

isortl Int.compare [3,1,2,1] = [1,1,2,3]

isortr lexcompare [(1,2),(2,2),(1,1),(2,1)]

= [(1,1),(1,2),(2,1),(2,2)]

isortl String.compare ["all", "your", "base", "are", "belong", "to", "us"]

val it = ["all", "are", "base", "belong", "to", "us", "your"] : string list
• For *integer* data we get
  \[ \text{isortl\ Int.compare} = \text{isortr\ Int.compare}. \]

• Is it true that for *all* types of data and *all* comparisons \( \text{cmp} \) we get
  \[ \text{isortl\ cmp} = \text{isortr\ cmp} \]
“algebraic” specs

If \( g \) is total, then for all \( z \) and \([x_1, \ldots, x_n]\),

\[
\text{foldr} \; g \; z \; [x_1, \ldots, x_n] = g(x_1, g(x_2, \ldots g(x_n,z)\ldots))
\]

\[
\text{foldl} \; g \; z \; [x_1, \ldots, x_n] = g(x_n, g(x_{n-1}, \ldots g(x_1,z)\ldots))
\]
so...

Let \( i = \text{ins} \; \text{cmp}. \)

\[
isortr \; \text{cmp} \; [x_1, \ldots, x_n] = i(x_1, i(x_2, \ldots i(x_n,[ \ ]))\ldots))\]

inserts “equal” items in
the same order as they occur in the list

\[
isortl \; \text{cmp} \; [x_1, \ldots, x_n] = i(x_n, i(x_{n-1}, \ldots i(x_1,[ \ ]))\ldots))\]

inserts “equal” items in
the opposite order
stability

A sorting function is *stable* if and only if it preserves the relative ordering of “equal” items.

**isortr cmp** is stable

**isortl cmp** is not
fun leftcompare ((x,y), (x’,y’)) = Int.compare(x,x’)

leftcompare is a comparison for int * bool

isortl leftcompare [(1,true),(1,false)] = [(1,false),(1,true)]

isortr leftcompare [(1,true),(1,false)] = [(1,true),(1,false)]

isortl ≠ isortr
special case

- If all items in L are \texttt{cmp}-equal

\begin{align*}
isortl \texttt{cmp} \ L &= \text{rev} \ L \\
isortr \texttt{cmp} \ L &= \ L
\end{align*}
reflection

• isortl and isortr are not extensionally equal

• What’s special about Int.compare that makes isortl Int.compare = isortr Int.compare?

\[
\text{Int.compare}(x,y) = \text{EQUAL} \quad \text{if and only if} \quad x = y
\]
stability as an equational property

(*) same : ('a * 'a -> order) -> 'a -> 'a -> bool *
fun same cmp x y = (cmp(x,y) = EQUAL)

fun filter p [ ] = [ ]
| filter p (x::L) = if (p x) then x::filter p L else filter p L

A function s : ('a * 'a -> order) -> 'a list -> 'a list is STABLE iff for all comparisons cmp, and all x and L,
filter (same cmp x) L = filter (same cmp x) (s cmp L).
going further

• Easy to generalize *mergesort*

• Easy to generalize from *lists* to *trees*

\[ \text{msort : ('a * 'a -> order) -> ('a list -> 'a list)} \]

\[ \text{Msort : ('a * 'a -> order) -> ('a tree -> 'a tree)} \]
cmp-sorted trees

Let \texttt{cmp} be a comparison for type \texttt{t}. Consider values of type \texttt{t} tree.

- \texttt{Empty} is a \texttt{cmp-sorted} tree
- \texttt{Node(A, x, B)} is a \texttt{cmp-sorted} tree iff
  
  (i) Everything in \texttt{A} is \texttt{cmp-\leq} to \texttt{x}
  
  (ii) Everything in \texttt{B} is \texttt{cmp-\geq} to \texttt{x}
  
  (iii) \texttt{A} and \texttt{B} are \texttt{cmp-sorted} trees
A **polymorphic** sorting function can be used with different **types** and **comparisons**

- **One** type, **many** instances
- **One** specification, **many** special cases
- **One** function, **many** uses
- **One** correctness proof, **many** consequences
We can rewrite isortr to make ins a \textit{local} function...

\begin{verbatim}
fun isortr cmp L =
  let
    fun ins cmp (x, [ ]) = [x]
    |     ins cmp (x, y::R) = (case cmp(x, y) of
        GREATER => y::ins cmp(x, R)
      | _               => x::y::R)
  in
    foldr (ins cmp) [ ] L
  end;
\end{verbatim}
The local function doesn’t need a cmp argument

```haskell
fun isortr cmp L = let
  fun ins (x, [ ]) = [x]
  |     ins (x, y::R) = (case cmp(x, y) of GREATER => y::ins(x, R)
    | _               => x::y::R)
in
  foldr ins [ ] L
end;
```

`isortr : ('a * 'a -> order) -> 'a list -> 'a list`