1 Type-based design

- You can often figure out the design of a function based on the types. This is especially true of conversion functions, like curry and uncurry, and the first part of the homework below, but is in general very useful for figuring out the general flow of a function.

- Below is a list of introduction and elimination expressions, which allow you create and utilize types.

<table>
<thead>
<tr>
<th>Type</th>
<th>Introduction</th>
<th>Elimination</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a \to b$</td>
<td>$\text{fun} x \to e$</td>
<td>$f \ x$</td>
</tr>
<tr>
<td>$a \times b$</td>
<td>$(e_1, e_2)$</td>
<td>$\text{fst} \ e, \text{snd} \ e, \text{fun} \ (x, y) \to e, \text{match} \ ewith(e_1, e_2)$</td>
</tr>
<tr>
<td>$&lt; a &gt;$</td>
<td>$&lt; e &gt;$</td>
<td>$e$</td>
</tr>
</tbody>
</table>

2 Reminders from last class.

1. We want to stage the following function, with type $\text{int} \to (\text{int} \times \text{int}) \to (\text{int} \times \text{int})$

   ```
   let rec f n s =
   if n = 0 then s
   else let x = fst s
    y = snd s
    s' = (x-y, x+y)
    in f (n-1) s'
   ```

   The code we would like to produce is something on the lines of

   ```
   let x' = x - y
   y' = x + y
   x'' = x' - y'
   y'' = x' + y'
   ...
   ```

2. Our first attempt at staging had the type $\text{int} \to< (\text{int} \times \text{int}) \to< (\text{int} \times \text{int})$
let rec f n s =  
  if n = 0 then s  
  else let (x = fst s)  
          y = snd s  
          s' = (x-y, x+y)  
          in ~f (n-1) <s'>

The code it produces is, at best, of the form

let x = fst (1,2)  
  y = snd (1,2)  
  s' = (x-y, x+y)  
  x' = fst s'  
...

And that only happens when we apply the let-in fix discussed later on.

3. Our second attempt at staging had the type \(\text{int} \rightarrow (\text{< int >} \times \text{< int >}) \rightarrow (\text{< int >} \times \text{< int >})\)

let rec f n s =  
  if n = 0 then s  
  else let x = fst s  
           y = snd s  
           s' = (<~x-~y>, <~x+~y>)  
           in f (n-1) s'

No fst's, snd's, or pair introduction.  
But code explosion, because \(x\) and \(y\) appear twice in the code.

3 Code explosion

1. Consider the power function.

let sqr x = x * x

let rec power (n, x) =  
  if n = 0 then 1  
  else if even n then sqr (power (n/2, x))  
           else x * (power (n-1, x))
2. We stage it like so

```plaintext
let rec power (n, x) =
  if n = 0 then 1
  else if even n then <sqr ~(power (n/2, x))>
  else <x * ~(power (n-1, x))>
```

And get code that looks like

```plaintext
< fun a -> ~(power(3, <a>))> =>
<fun a -> a * (sqr(a*(1)))>
```

3. Can we get rid of the call to `sqr`? Our ideal code looks something like this for $x^5$

```plaintext
fun a -> a *
  (let a' =
    (let a'' = a * 1 in a'' * a'')
    in a' * a')
```

Which is pretty much the same as

```plaintext
fun a -> let a'' = a * 1 in
  let a' = a'' * a'' in
  a' * a' * a
```

where we use let bindings to gather squares.

4. So let us try to stage square. To a first approximation, this new `sqr` is `< float >→< float >`

```plaintext
let sqr2 x = <~x * ~x>

let rec power2 (n, x) =
  if n = 0 then 1
  else if even n then sqr ~(power (n/2, x))
  else <~x * ~(power (n-1, x))>
```

This generates code that looks like

```plaintext
< fun a -> ~(power(3, <a>))> =>
<fun a -> a * ((a*1) * (a*1))>
```
OH NO! It’s code duplication. That \( a \times 1 \) occurs twice. The code will be exponential in \( n! \).

Aha! The square function! We splice code in twice, and thus duplicate the code. \( x \) comes in, and gets put in two places.

So

\[ <\sim x \times \sim x> \]

yields precisely

\( (a*1) \times (a*1) \)

5. To avoid duplicating \( e_1 \) we bind \( e_1 \) to a variable using a let: \( let \ x = e_1 \ in \ x \times x \)

\[
\text{let sqr3 x = } \langle \text{let b = } \sim x \text{ in } b * b \rangle
\]

Generates code that looks like

\[ \langle \text{fun a } \rightarrow \sim (\text{power}(3, \langle a \rangle)) \rangle \Rightarrow \langle \text{fun a } \rightarrow a * (\text{let x = a*1 in } x * x) \rangle \]

As a general pattern, replace

\[ <\ldots \sim z \ldots \sim z \ldots \ldots > \]

with

\[ <\ldots \text{let } x = \sim z \text{ in } \ldots x \ldots x \ldots \ldots > \]

4 Homework

1. Define \( f \) and \( g \), where they have the types

\[
f : (\langle int \rangle \times \langle int \rangle) \rightarrow \langle \text{int } \times \text{int} \rangle\]
\[
g : \langle \text{int } \times \text{int} \rangle \rightarrow (\langle \text{int } \rangle \times \langle \text{int } \rangle)
\]

2. Write a detailed explanation of why we cannot just insert a let-statement in the function discussed in the "Reminders from last class" section in its current state.

3. Look at the ideal code we would like to generate and try to write a staged program to generate it, independently of the function we were staging today.