Tail recursion (continued)

Review: tail recursive version of sum

```
sum = sum' 0
sum' z [] = z
sum' z (x:xs) z = sum' (z + x) xs   -- nothing to postpone
```

Run `sum' 0 [1,2,3]` using Python "interpreter" above.

```
    z = 0, l = [1,2,3], done = False, result = None
==> z = 1, l = [2,3],   done = False, result = None
==> z = 3, l = [3],     done = False, result = None
==> z = 6, l = [],      done = False, result = None
==> z = 6, l = [],      done = True,  result = 6
```

Note how in each step a list element is being added to `f` 's argument `z` ,

The argument `z` is "accumulating" the result.

The tail recursive sum gets evaluated like the following Python program

```
z = 0
for x in [1,2,3]:
  z = x + z
```

The equational view illustrates what's going on.

```
sum [1,2,3]
= sum' 0 [1,2,3]
= sum' (0 + 1) [2,3]
= sum' 1 [2,3]
= sum' (1 + 2) [3]
= sum' 3 [3]
= sum' (3 + 3) []
= sum' 6 []
= 6
```

Aside: why is the above crappy in Haskell? Hint: consider the above with lazy evaluation.

General definition of tail recursion

A recursive function is *tail recursive*

if the value of a call to the function is the same as the value of its recursive call.

OR

if each recursive call in the program is surrounded only by conditional (if etc) and let statements.

Understanding tail recursive programs

Usual recursion pattern: assume recursive calls work (if the main argument is smaller).

Using "normal" recursion with pattern matching:

- when computing *(sum (x:xs))*, assume *(sum xs)* is the sum of the numbers in *xs*, so *sum (x:xs) = x + sum(xs)* is the sum of the numbers in *x:xs*.

Using the helper sum':

- when computing *(sum' z (x:xs))*, what can we assume about *(sum' (z+x) xs)*?

How can we specify what *sum'* is supposed to do?

Specifying sum'.

```
sum' :: Int -> ([Int] -> Int)
sum' z [] = z
sum' z (x:xs) = sum' (z + x) xs
```

This function is only used by *sum* with *z=0*. What does it mean for a general *z*?

Specification: For all *z*, *sum' z xs* adds *z* to the sum of the numbers in *xs*.

This avoids thinking operationally about *z* being an *accumulator*, i.e. an "running" sum of list elements seen "so far".

Instead, *z* is any number.

Tail recursion and induction

Prove that the tail-recursive sum computes the sum of all the list elements. I.e.
For all xs, *sum xs = Σ xs*, where *Σ xs* is the sum of the elements in *xs*.

Proof by induction on *n = length(xs)*.

Base case. *sum [] = sum' 0 [] = 0*

Inductive case. Assume *sum xs = Σ xs* and show that *sum (x:xs) = Σ (x:xs)*.

> *sum (x:xs) = sum' 0 (x:xs) = sum' (0+x) xs*

We're stuck: can't apply induction hypothesis; no way to get *sum* back.

Solution from math: Generalize the induction hypthesis.

Proof with generalized induction hypothesis.

Proof by induction on $n = length(xs)$ that for all z, $sum' \; z \; xs \equiv z + \Sigma xs$.

Base case. $sum' \; z \; [] \equiv z = z + \Sigma \; []$

Inductive case. Assume, for all z, that $sum' \; z \; xs \equiv z + \Sigma xs$.

> $sum' \; z \; (x:xs) = sum' \; (z+x) \; xs = (z+x) + \Sigma \; xs = z + \Sigma \; (x:xs)$

QED

Another tail recursion example: reverse

Denote by ↻(xs) the reverse of the list xs.

```
rev xs = rev' [] xs

rev' z [] = z
rev' z (x:xs) = rev' (x:z) xs
```

```
rev [1,2,3]                  -- For any z:
= (rev' []) [1,2,3]          (rev' z) [1,2,3]
= (rev' [1]) [2,3]           = (rev' (1:z)) [2,3]
= (rev' [2,1]) [3]           = (rev' (2:1:z)) [3]
= (rev' [3,2,1]) []          = (rev' (3:2:1:z)) []
= [3,2,1]                    = 3:2:1:z
```

Spec of rev': *rev' z xs = ↻xs ++ z*.

Emulating loops in Haskell

Consider a "loopy" program using, say, three variables x y z.

```
def loopy():
    x = x0; y = y0; z = z0;
    while e do:  # e is some boolean expression using x,y,z
        x = e1   # update x using expression e1
        y = e2   # update y using expression e2
        z = e3   # update z using expression e3
    return r     # r some expression using x, y, z
```

In Haskell:

```
loopy = loopy' x0 y0 z0
loopy' x y z = if e then loopy' e1 e2 e3 else r
```

```
loopy = loopy' x0 y0 z0
loopy' x y z = if e then loopy' e1 e2 e3 else r
```

Input/Output

LYaH:

- Chapter 9, *Input/Output*, section 1 ("Hello World").

IO in other languages

Most programming languages, e.g. Python, Java, C/C++:

1. Allow free mix of computation and IO.
2. Give no way to ensure imported methods are side-effect-free.

This makes parallelization and reasoning difficult.
Consider a function/method call `f(e1,e2)`.

1. Evaluate arguments e1 and e2 in parallel? Race condition possible.
2. Order of evaluation matters: can't reason by just plugging in argument expressions for variables in the body of `f`.

Haskell's approach to IO

In a nutshell: use the type system to prevent all IO within normal computation, forcing it to the "top level" of the program.

- `IO a` is the type of *IO actions* that produce a value of type `a` when *run*.
- computing value of type `IO a` does *not* do any IO.
- E.g. `putStrLn:: String -> IO()` is a "built-in" that prints a line.
  Evaluating `putStrLn "foo"` creates an IO action value but does *not* do any IO.
- What can we do with an IO action besides combine it with other ones?

    i. Run it as the *main* program.

    ii. Run it at the `ghci` prompt.

What is in an action?

An action

1. Specifies some IO to do, input and/or output, when the action is run.
2. Specifies what *value* should be produced when the action is run. E.g. for a "read" action, the input value is produced.

Key idea: *evaluating* an expression of type `IO a` produces an *action* which can be *run* only as *main* or at the `ghci` prompt.

Some built-in actions:

```
readFile :: String -> IO String  -- value of (readFile f) is the contents
putStrLn :: String -> IO ()      -- () is for "don't care" values
```

Building IO actions

We use a special *do* notation.

```
do
  -- line 1
  -- line 2
  -- ...
  -- line n
```

where each line is one of the following.

1. `e` where `e` is any expression of type `IO a` .
2. `x <- e` where `e` is any expression of type `IO a` .
3. `let x = e` where `e` is an expression of any type.

The `do` expression, when *evaluated*, evaluates to an action.

When the action is *run*, as `main` or at the `ghci` prompt, each line is *run* in order.

1. `e`
   gets run as an IO action
2. `x <- e`:
   The action `e` is run, and the value produced by running it is named `x`
3. `let x = e`: the value of *e* is a "normal" value and is named `x`

A degenerate IO action

return *e* is an action for any expression *e*. When run, it

1. does no IO, and
2. produces the value of *e* as its value

Example: interactive stack-based calculator.

Program maintains a stack. User enters an "operator", resulting stack is displayed.

Op	Action
<number>	push the number onto the stack
*	replace top two elements with their product
+	replace top two elements with their sum
/	replace top two elements with the result of dividing
-	negate the top element
c	clear the stack

Example interaction.

User Input	Displayed stack
c	
17	17
3	3:17
1	1:3:17
-	-1:3:17
+	2:17
*	34