Functional Programming

Functional programming is one type of declarative programming.

In functional programs, it still *feels* like you're giving explicit instructions, but the language/compiler has more freedom in interpreting details.

Start by thinking of imperative programming, but…

- There is no “state” (global variables, etc.).
- No variables at all can be stored.
- No variables means no loops (because we need variables as `for` loop counters or `while` conditions).

How can we get anything done without variables? The emphasis is on evaluating calculations, not executing steps.

We still get function arguments. e.g. `x` is a function argument, not a variable in these:

```
def half_of(x):
    return x/2
```

.py

```
half_of x = x/2
```

.hs

Recursion can take the place of loops.

What we express is “calculation we need the result of” instead of “follow these steps”. That is what gives the compiler more freedom.

Having no variables means (should mean?) easier debugging: results of a function are determined *entirely* by its arguments.

Haskell

Haskell is a functional programming language.

It's *very* functional: *purely functional*. It doesn't allow you to sneak imperative tricks in.

Some other functional languages allow some imperative things to happen (Lisp, Scheme, OCaml, Erlang). They are arguably more practical, but [I claim] not as good at helping you learn functional style programming.

So, we're using Haskell.

We will use *GHC* (the *Glasgow Haskell Compiler*) as our compiler. It's the standard toolchain for Haskell.

Also, it includes the GHCi interactive environment where you can test expressions/code, as well as interact with programs/modules.

```
Prelude> 2-4
-2
Prelude> 2*3 /= 6
False
```

GHCI

```
Prelude> reverse [2,3,4,5]
[5,4,3,2]
```

To load code from a `.hs` file, use "`:l`":

```
Prelude> :l mycode.hs
[1 of 1] Compiling Main             ( mycode.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

GHCI

To reload changes to the `.hs` file, use "`:r`":

```
*Main> :r
Ok, modules loaded: Main.
```

GHCI

Haskell Basics

Basic data types (incomplete):

- numbers: `4`, `6.3`, `(-4)`
- strings: `"abc"`, `"6.3"`
- lists: `[1, 2, 3, 4]`, `["abc", "def"]`
- boolean: `True`, `False`

Numeric expressions work like you probably expect:

Expression	Result
`3 + 4`	`7`
`3 + 4 * 2`	`11`
`2 ^ 8`	`256`
`3.0 + 4`	`7.0`
`8.0 / 2`	`4.0`

... and boolean expressions:

Expression	Result
`not True`	`False`
`False || True`	`True`
`False && True`	`False`
`6 == 4`	`False`
`6 /= 4`	`True` (≠ comparison)
`6 <= 4`	`False`

One basic list operation: *concatenate*:

Expression	Result
`[1,2,3] ++ [4,5]`	`[1,2,3,4,5]`
`[] ++ [1,2]`	`[1,2]`

Another: *cons* (short for "construct"):

Expression	Result
`3 : [4,5]`	`[3,4,5]`

Expression	Result
`2 : (3 : (4 : []))`	`[2,3,4]`
`2 : 3 : 4 : []`	`[2,3,4]`

Note: The `(:)` operator has a list **element** on the left, and a **list** on the right. The element is prepended to the start of the list. The operator is right-associative, so the last two examples are equivalent.

Characters are indicated with single quotes and strings with double quotes (like C):

Expression	Result
`'a'`	a character
`"a"`	a string
`"abc"`	a string
`'abc'`	an error
`"abc" ++ "def"`	`"abcdef"`
`'a' ++ "bcd"`	an error

Strings are just shorthand for lists of characters (like C):

Expression	Result
`"ab" ++ ['c', 'd']`	`"abcd"`
`'a' : "bc"`	`"abc"`
`['X', 'Y'] == "XY"`	`True`

Haskell lists are conceptually singly-linked lists: imagine a head with pointer to the rest of the list. *

Not only does `1:2:[] == [1,2]` but you could think of the syntax `[1,2]` as just a shorthand for `1:2:[]`.

* but what the compiler does is another story.

Haskell Functions

Functions are (obviously) important in functional languages. They are the primary unit of code (as opposed to classes, statements, etc).

Calling functions in Haskell looks different than you might expect: no parens around arguments; just spaces separate arguments. It may take some getting used to.

Most Languages	Haskell Equivalent
`f(a, b)`	`f a b`
`f(a, b+1)`	`f a (b+1)`

`f(a, b) + 1`	`f a b + 1`
	`(f a b) + 1`

`f(g(a), b)`	`f (g a) b`

Haskell knows what is a function and how many arguments it takes: it expects the right number of arguments to follow (except when it doesn't: more later).

So, if `f` is a two-argument function, this is sensible:

```
f 1 2
```

... and these are incomplete and have and extra argument, respectively (but more later):

```
f 1
f 1 2 3
```

Defining functions (at least for simple expressions) is easy (filename `functions1.hs`):

```
half x = x/2
isBig a = a > 100
listify x y = [x, y]
```

And then you can use them:

```
Prelude> :l functions1.hs
[1 of 1] Compiling Main             ( functions1.hs, interpreted )
Ok, modules loaded: Main.
*Main> half 5
2.5
*Main> isBig 123
True
*Main> listify 'a' 'b'
"ab"
```

Pattern Matching

A simple expression doesn't let us get much logic into a function.

The most Haskell-ish way to express a condition is *pattern matching* where we can express “if the arguments look like *this*, the result is...”.

For example, a function for a logical “and”:

```
myAnd False _ = False
myAnd True a = a
```

Things happening there:

- The first definition is used if the first argument is `False`.
- The _ matches any value (but ignores it).
- The second is used if the first argument is `True`
- The named argument matches anything captures its value.

The behaviour is an “and” operation:

```
*Main> :r
Ok, modules loaded: Main.
*Main> myAnd True True
True
*Main> myAnd True False
False
*Main> myAnd False True
False
*Main> myAnd False False
False
```

Rule: the *first* definition that matches applies.

```
isZero 0 = True
isZero _ = False
```

```
*Main> isZero 0
True
*Main> isZero 1
False
```

Recursion works and we can match a *cons pattern* (first and rest of a list). We could define something like:

```
myLength [] = 0
myLength (_:xs) = 1 + myLength xs
```

```
*Main> myLength [1, 2, 3]
3
*Main> myLength "Hello World"
11
```

The cons pattern is very common for recursing over a list.

```
processList [] = …                        -- base case
processList (x:xs) = … processList xs …   -- recursive case
```

Another example: are two lists equal?

```
listEqual [] [] = True
listEqual (x:xs) (y:ys) = x == y && listEqual xs ys
```

```
*Main> listEqual [1,2,3] [1,2,3]
True
*Main> listEqual [1,2,3] [1,2,4]
False
```

Problem?

The patterns in the function definition miss the case of non-equal length lists.

```
*Main> listEqual [1,2,3] [1,2,3,4]
*** Exception: functions1.hs:(16,1)-(17,51): Non-exhaustive patterns in function l
```

We can add these cases to cover those inputs:

```
listEqual [] (_:_) = False
listEqual (_:_) [] = False
```

(or `listEqual _ _ = False` and use the first-match-only behaviour.)

Rules for pattern matching:

- plain identifier (argument name, like `x`): matches anything and captures the value.
- `_`: matches anything but we don't need the value.
- literal value (like `1`, `[]`, etc): matches if equal.

Matching lists:

- `[]`: matches an empty list.
- `(a:tail)`: matches a list with **at least one element**. First is `a`, rest of list is `tail` (and could be an empty list).

- `(a:b:tail)`: matches a list with **at least two elements**. First is `a`, second is `b`, remainder of list is `tail` (could be empty).

For example, if we have the pattern:

```haskell
(a:b:c)
```

... then these lists match as follows:

List	a	b	c
`[6,7,8,9]`	`6`	`7`	`[8,9]`
`['a', 'b']`	`'a'`	`'b'`	`[]`
`"ab"`	`'a'`	`'b'`	`[]` (`== ""`)
`[100]`	no match		

Conditionals

Pattern matching produces nice-looking code, but it can't do everything. There are a few ways to write more flexible conditional code.

There is an `if`/`else` expression, but I'm banning it: it's not Haskell-ish, and often used by students to avoid learning the functional style.

Much nicer: *guarded expressions*

```haskell
mySignum x
  | x>0          = 1
  | x<0          = -1
  | otherwise    = 0
```

Analgous to the mathematical function:

$$signum(x) = \begin{cases} 1 & \text{if } x > 0 \\ -1 & \text{if } x < 0 \\ 0 & \text{otherwise} \end{cases}$$

There is also a `case` expression to give multiple alternatives in a convenient way:

```haskell
word n = case n of
  1 -> "one"
  2 -> "two"
  3 -> "three"
  _ -> "???"
```

The `case` is an expression: it can be used just like any other Haskell expression:

```haskell
wordWithX n = (case n of
  1 -> "one"
  2 -> "two"
  3 -> "three"
  _ -> "???") ++ "X"
```

All of the “values” for the cases are patterns, just like function definitions:

```
describeList lst = "The list is " ++ case lst of
  _:_:_:_:_  -> "fairly long"
  _:_        -> "short"
  []         -> "empty"
```

```
*Main> describeList []
"The list is empty"
*Main> describeList [3,4]
"The list is short"
*Main> describeList "hello"
"The list is fairly long"
```

Repetition

Since there are no variables, we can't write loops like we're used to in other languages.

Loops (as we think of them) need a variable (`i` here) that determines when the loop ends: no variable = no way to end the loop.

```
for ( int i=0; i<10; i++ ) {
    cout << i << '\n';
}
```

```
int i = 100;
while ( i>1 ) {
    cout << i << '\n';
    i /= 2;
}
```

We have recursion, and know (I hope) that recursion can be used instead of any loops.

```
myLength [] = 0
myLength (_:xs) = 1 + myLength xs
```

Recursion is going to be the biggest way we repeat logic: time to get comfortable with it.

With recursions and conditions as we have them, the language is Turing complete. Everything else is just convenience.

We will also have implicit loops [my term]: an expression that implies repeated application of a function...

List Comprehensions

Recall the ***set builder*** notation from math:

$$\{x^2 \mid x \in \mathbb{N}, x \text{ is even}\} = \{4, 16, 36, \ldots\}$$

It's a very efficient notation to describe a set. Haskell has a very similar notation to build a list...

A ***list comprehension*** are a syntax in Haskell to describe a list, similar to the set builder notation.

[Python borrowed list comprehensions: you may have seen them there. Or LINQ in .NET is like list comprehensions and SQL had a baby.]

First part: ***generators***, like “$x \in \mathbb{N}$” in the set notation. The generator creates the “source” values.

Use the `<-` operator with an existing list on the right. e.g.

```
x <- [1,2,3,4]
```

The generator can be used with any expression to build a list. Each generator value is used as an “argument” to the expression.

```
Prelude> [ x^2+1 | x <- [1, 2, 3, 4, 5, 6, 7] ]
[2,5,10,17,26,37,50]
Prelude> [ succ c | c <- "hello" ]
"ifmmp"
```

[`succ` is “successor”: the next one.]

You can use more than one generator in a comprehension: *every pair* of values is produced.

```
Prelude> [ 10*x + y | x <- [3,4,5], y <- [6,7,8,9]]
[36,37,38,39,46,47,48,49,56,57,58,59]
```

A second piece we can add: ***guards***, like “x **is even**” in the set notation.

A guard is a boolean expression that decides if a value should be included in the result or not.

```
Prelude> [ x^2+1 | x <- [1, 2, 3, 4, 5, 6, 7] ]
[2,5,10,17,26,37,50]
Prelude> [ x^2+1 | x <- [1, 2, 3, 4, 5, 6, 7], even x ]
[5,17,37]
```

[`even` is built in.]

List comprehensions are often used with the `..` shorthand for arithmetic sequences.

Expression	Result
`[1..6]`	`[1,2,3,4,5,6]`
`[2,4..15]`	`[2,4,6,8,10,12,14]`
`['a'..'m']`	`"abcdefghijklm"`

The “`..`” can be used anywhere you need a list, not just in a comprehension. But it makes it easy to write a list comprehension that builds a list of arbitrary size:

```
firstSquares n = [ i*i | i <- [1..n] ]
firstEvenSquares n = [ i*i | i <- [1..n], even i]
firstEvenSquares' n = [ i*i | i <- [2,4..n]]
```

```
*Main> firstSquares 10
[1,4,9,16,25,36,49,64,81,100]
*Main> firstEvenSquares 10
[4,16,36,64,100]
*Main> firstEvenSquares' 10
[4,16,36,64,100]
```

The notation we have lets us be *very* expressive in Haskell.

```
qs [] = []
qs (x:xs) = smaller ++ [x] ++ larger
```

```
  where smaller = qs [a | a<-xs, a<=x]
        larger = qs [a | a<-xs, a>x]
```

… a Quicksort in four lines.

Let and Where

It's easy to create unreadable code in Haskell: any function that does anything interesting probably requires a long expression. Just because it works doesn't mean it's good code.

The `let` and `where` clauses let you alias values so you can use them by-name or multiple times.

e.g. in the definition of the Quicksort, the `where` let me (1) turn what could have been one long, ugly expression into three short ones; (2) give the sub-expressions names so they are easier to understand.

```
qs [] = []
qs (x:xs) = smaller ++ [x] ++ larger
  where smaller = qs [a | a<-xs, a<=x]
        larger = qs [a | a<-xs, a>x]
```

The “`let` *name*=… `in` …” structure creates an expression and can be used anywhere you can use any other expression.

```
Prelude> let x=4 in x*2
8
Prelude> (let x=4 in x*2) ^ 3
512
```

But it's more useful to simplify things in longer definitions:

```
qs' [] = []
qs' (x:xs) =
  let smaller = qs' [a | a<-xs, a<=x]
      larger = qs' [a | a<-xs, a>x]
  in smaller ++ [x] ++ larger
```

The `where` must be part of a function definition (or similar block):

```
somePowers x = [x, sq x, sq (sq x)]
  where sq n = n*n
```

In either case, multiple sub-expressions can be defined, separated by linebreak (or semicolon).

The choice between `let` and `where` is often one of style: does it make more sense to define the sub-expressions before or after the “main” expression? It will depend on the context. Do the more readable thing.

```
qs (x:xs) = smaller ++ [x] ++ larger
  where smaller = qs [a | a<-xs, a<=x]
        larger = qs [a | a<-xs, a>x]
```

```
qs' (x:xs) =
  let smaller = qs' [a | a<-xs, a<=x]
      larger = qs' [a | a<-xs, a>x]
  in smaller ++ [x] ++ larger
```

You should use `let` or `where` if your code is getting complicated. Break up big expressions; give the sub-parts meaningful names.

[If I see a single expression more than one moderate-length line, I'm going to refuse to help you. The TAs should take off marks for coding style too.]

Recursion

The idea of recursion shouldn't be new. But maybe a little review wouldn't hurt...

Function definitions will have two cases (which may be sub-divided further).

Base case: small case(s) where the result is obvious.

Recursive case: you find a smaller subproblem, another version of the same problem that will help you find the “real” solution.

Make a recursive call to solve the subproblem.

Use that sub-result to return the “real” result.

e.g. Argument `n` becomes `n-1` or `n/2`. Base case is `0` or `1`.

e.g. Argument `x:xs` becomes `xs`. Base case is `[]`

An example: calculate powers (x^y for x, y integers, $y \geq 0$).

```haskell
myPower _ 0 = 1
myPower x y = x * myPower x (y-1)
```

This uses the facts $x^0 = 1$ and $x^y = x \cdot x^{y-1}$.

Like with any other iteration, we have the opportunity to be more or less efficient.

```haskell
myPower' x y
  | y==0      = 1
  | even y    = half*half
  | odd y     = x*half*half
  where half = myPower' x (div y 2)
```

This uses the facts $x^0 = 1$ and $x^y = \left(x^{y/2}\right)^2$ (and fixes integer division rounding).

Running times?

`myPower` is $O(y)$ and `myPower` is $O(\log y)$.

More recursion: *Hutton* chapter 6 (especially §6.6) and *Learn you a Haskell* chapter 5.