Functional Programming

Imperative programs:

- Write values to data structures.
- Because the order of writes matters, the order of statements matters. Hidden side effects make program difficult to write and modify. Write reordering can give different answers, so there are serial dependencies in the code.

Functional programs:

- Evaluate "pure" expressions without side effects, like mathematics.
- Because the order of evaluation does not matter, implementations can use lazy evaluation, or parallel evaluation. The lack of side effects makes code more modular, more reliable, and easier to run in parallel, understand, and modify.
- Often use "higher-order functions": functions that take other functions as arguments. Type declarations involving functions tend to rapidly get unwieldy (e.g., recursive function taking itself as an argument), so functional programming languages normally use type inference (OCaml, Haskell, C++11 templates or auto) or dynamic typing (JavaScript, Erlang).

"Functional programming combines the flexibility and power of abstract mathematics with the intuitive clarity of abstract mathematics."

- Randall Munroe, XKCD #1270 alt text

See what Haskell programmers think functional programming means.

The functional style can be used in any language--even assembly--by ensuring functions do not have side effects. Several key abstractions like Zippers are useful in any language.

Drawbacks of the functional style include a mismatch with real-world I/O operations such as writing bytes to the network or serial port, or splatting pixels on the screen. It is also more complex to create functional (read-only) data structures (see readable summary of this and later functional data structures).

(Illustration of Lambda calculus as being like alien technology.)

Compact Lambda Calculus

In 1932 Alonzo Church discovered lambda calculus, a powerful functional representation that is mathematically fascinating but not easily understood. (Examples, which are most easily read with a Greek alphabet cheat sheet.)

In analyzing theoretical computation, we need a very simple model of computation, so we often use a Turing machine. In mathematics we need a very simple model of functions, so we normally use lambda calculus (readable intro to lambda calculus for C++ programmers) which uses simple single-variable replacement, a simpler structure that makes for very simple proofs.

Everyone seems to use a slightly different syntax for defining and applying lambda calculus functions. Here we're defining a function F that takes one argument x, and calls the function G with it.

Source	*Anonymous Function Notation*
1932 Alonzo Church paper	F: λ x . [G x]
Haskell	`F = \x -> G x`
Python	`F = lambda x: G(x)`
JavaScript	`F = x => G(x)`
C++	`auto F=[](auto x) { return G(x); };`
English text	`Define a function as taking some arguments, and doing some work on them.`

Functional programming term	*Lambda calculus*	*Python*	*Description*
Lambda **abstraction**	λ x . [_]	`lambda x: _`	Define a function.
Function **application** (apply the function G to the value x)	G x	G(x)	Call (apply) a function with an argument.
(Beta) β-**reduction**	(λ x . x) F = F	Run the function	Run a function

Functional Programming in Python

Python functions normally use the "def" keyword:

```
def dadd(a,b):
    return a+b
dadd(3,10)
```

13

Python also supports the keyword "lambda" that creates a function taking a list of arguments, like "lambda *arguments*: *body*":

```
fadd = lambda a,b: a+b
fadd(3,10)
```

Result: 13

"Currying" is the conversion of multi-argument functions into a chain of nested simpler single-argument functions that give the same result. Lambda expressions support currying more naturally than multi-argument functions, and they give the opportunity for higher-order programming such as only binding one of the arguments now and saving the other for later.

```
cadd = lambda a: lambda b: a+b
cadd(3)(10)
```

Result: 13

The terse lambda syntax above is equivalent to defining cadd as an outer function that takes a, with an inner function that takes b:

```
def cadd2(a):
    def innerC(b):
        return a+b
    return innerC

cadd2(3)(10)
```

Result: 13

Key to functional programming: you can pass functions as arguments to other functions, or return a function instead of a value:

```
bindleft = lambda f: lambda L: f(L)
add3 = bindleft(cadd)(3)
add3(10)
```

Result: 13

It's common to use nested functions in functional programming, which is the easiest way to do things like arbitrary nesting. For example, this function takes N curried arguments, and adds them all together:

```
# Add up the next n curried arguments
def addN(n):
    sum=0
    def inner(next):
        nonlocal sum   # access parent's sum
        sum = sum+next
        nonlocal n   # access parent's n
        n=n-1
        if n==0:
            return sum  # done
        else:
            return inner  # more coming!
    return inner

addN(3)(1)(5)(10)
```

Result: 16

The most natural way to write recursive functions in a pure functional language is by passing the name of the function to itself as an argument(!), so the function can call itself. Here's the Fibonacci sequence:

```
fib=lambda f: lambda i:  i  if (i<2)  else  f(f)(i-1)+f(f)(i-2)
fib(fib)(6)
```

Result: 8

This fibonacci is purely functional, but it's a little clunky to call because we need to call fib(fib). Luckily, you can define a simple higher-order function called the "U combinator" to fix the call side:

```
U = lambda f: f(f)
fibEasy = U(fib)
fibEasy(6)
```

Result: 8

It's still a little clunky to define fib, because to express the recursion, we have to call f(f)(i-1). You can fix this with a fixed-point combinator like the famous Y combinator (not the company).

```
Y = lambda F: F(lambda x:Y(F)(x))
fibY = lambda f: lambda i: i if (i<2) else f(i-1) + f(i-2)
fibEasier = Y(fibY)
fibEasier(6)
```

Result: 8

This Python intro to Lambda Calculus builds up all of computation from nothing but functions, and includes the Y combinator definition above. This is an intellectually bracing exercise, and excellent practice with the founding ideas in functional programming, even if it's poor for writing actual programs!

Here's our 2024 attempt at Python lambda calculus.

Once you're comfortable with those, Stephen Wolfram's The Ruliology of Lambdas has some interesting observations.

Functional Programming in JavaScript

JavaScript makes heavy use of functional programming ideas in the 'callbacks' used in every network access. This means you may need to do fairly heavy functional programming for normal tasks like fetching a list of network files–your callback after getting each file needs to use itself as a (recursive) callback when it grabs the next file, which continues until a network error or the end of the list. (Today you can finally do this with a normal loop, using the async await syntax, but there are still some advantages to doing it via functional programming recursion.)

JavaScript supports a terse "argument => body" syntax for declaring anonymous functions:

```
f = (a,b) => a+b
print(f(3,10));
```

(Try this in NetRun now!)

JavaScript also lets you declare inner functions anywhere, and they can directly access local variables of any of their parent functions.

```
function f(a,b) {
      var x=function(c) { // declare inner function
            return a+c; // access parent function's variable
      }
      return x(b);
}
print(f(3,7));
```

(Try this in NetRun now!)

In the terse style, this inner function can just be a parenthesized subexpression:

```
f = (a,b) => ( c => a+c ) (b)
print(f(3,10));
```

(Try this in NetRun now!)

JavaScript also has first-class functions: you can pass functions as arguments, as used for callbacks in many APIs. Here's how we'd pass a function itself as an argument, here for simple recursion:

```
var fib=function(recurse,i) {
      if (i<2) return i;
      else return recurse(recurse,i-1) + recurse(recurse,i-2);
}
print(fib(fib,6));
```

(Try this in NetRun now!)

It's a bit clunky to pass the function name twice, so we might write a "higher order" function that takes as an argument the function that needs itself as an argument:

```
var heal_recursion=function(f,arg) { return f(f,arg); }
```

This looks a little better if you curry the arguments into separate functions:

```
var curry_recursion=function(f) {
   return function(arg) { return f(f,arg); }
}
```

You'd use this like:

```
var f=curry_recursion(fib);
return f(10);
```

One place you might use this stuff is in building read-only data structures:

```
function no_students(s) { return false; };
function add_student(next_student,old_students) {
      return function (s) {
            if (s==next_student) return true;
            else return old_students(s);
      }
};

function is_student() {
      return add_student("A",add_student("B",no_students));
};

print(is_student()("A"));
```

(Try this in NetRun now!)

```
/* Make a new function that represents a tree of data.
    left and right can be data or functions.
  Returns a node function that can be visited.
*/
var tree_node=function (left,right) {
  return function (visit) {
   if (left) { visit(left); }
   if (right) { visit(right); }
  }
};

/* Visit a tree node */
var visit=function(node) {
  if (typeof node == "function")
    node(visit);
  else
    console.log(node);
};

var tree = tree_node(123,tree_node("hello","madness"));

visit(tree);
```

(Try this in NetRun now!)

I wrote a walkthrough of lambda calculus in JavaScript.

Functional Programming in C++

See several functional approaches to C++.

One big limitation of statically typed languages like C++ is it's difficult to express recursive higher order functions: this direct translation of the javascript above works fine right up until you try to call it--the specialization we want is fib<typeof(fib<typeof(fib… forever!

```
template <typename FN>
long fib(FN recurse,long i)
{
	if (i<2) return i;
	else return recurse(recurse,i);
}

long foo(void)
{
	return fib(fib,6);
}
```

(Try this in NetRun now!)

There is a pre-C++11 solution to this, (the fixed point combinator) but the types get really ugly. In C++14, lambdas can take an "auto" parameter for calling themselves, and it's much more natural:

```
auto f=[](auto self,int i) {
    if (i<=1) return 1;
    else return i*self(self,i-1);
};
```

Functional Programming in Haskell

Haskell is nearly unique in being a "pure" functional language: most primarily functional languages such as OCaml allow imperative code blocks.

https://tryhaskell.org/

The syntax is very compact and purely functional.

```
main = putStrLn "Hello"
```

The above defines the main function as an application of the print function "putStrLn" to a string.

```
main = (\x -> putStrLn x) "Hello"
```

This defines an anonymous lambda that takes an argument x, then applies the lambda to the string.

```
main = (\f -> (\x -> f x) "Hello") putStrLn
```

This defines two nested lambdas. The string gets applied first, and we bring in the function later. (Parenthesis are critical here, because space will apply arguments it's easy to apply the wrong thing, which will fail type check.)

Monads in Disguise

There's an abstraction used for state in functional programming called a "monad", taken from category theory. The main utility with monads is being able to rearrange and reason about code with side effects. (Think about a complex program with global variables, it can be difficult to rearrange function calls without breaking the global variables.)

"For a monad **m**, a value of type **m a** represents having access to a value of type **a** within the context of the monad." —C. A. McCann

The two operations in a Monad interface are:

- wrap: bring a value into a monad.
    - In Haskell, this is the "return" operation.
- bind: call a function on the monad value, effectively changing it.
    - In Haskell, this is the bind operator ">>=", which is basically just function application but for monads.

Many stateful things like data structures have a natural representation as monads.

Stateful Thing	*Monad operations*	*Description*
Variable	wrap: declare the variable bind: change the variable	Wrap brings a plain value into a monad context. Bind changes the variable's value (usually written with operators).
JavaScript Promise	wrap: Promise.resolve bind: .then()	Wrap is how you bring a raw value into the Promise. Bind is how we chain additional things onto the promise.
Rust Option	wrap: Some(x) or None bind: map	Wrap creates an Option type. map applies a function *if* there is a value inside.

The three Monad laws are:

- Left identity: wrap and then bind is equivalent to just calling the function.
    - In Haskell: (return a) >>= f == f a
    - Here a is any value, and f is a function that takes a as input.
    - (return a) wraps a
    - >>= f binds f
    - The combination is equivalent to just f applied to a
- Right identity: binding wrap does nothing.
    - In Haskell: (m >>= return) == m
- Associativity: binding functions can happen in either order.
    - In Haskell: (m >>= f) >>= g == m >>= (\x -> f x >>= g)
    - Left side: start with m, bind f, then bind g.

- Right side: start with m, bind a lambda that first applies f, then binds g.

Homework

"Currying" is the conversion of multi-argument functions into a chain of simpler single-argument functions that give the same result.

Write a Python function curry2 that takes a single argument: a function, that in turn takes two arguments. Curry2 should return a chain of higher-order functions that take one argument at a time. For example, curry2(f)(a)(b) should give the same result as f(a,b).

Now write a Python or Javascript function curryN that takes the number of arguments to curry. For example, curryN(f,2)(a)(b) should give the same result as f(a,b).

- In Python you can call a function with N arguments in a [] list with a star syntax
- In JavaScript you can call a function with an array of arguments using *fn*.apply(*fn*,*argArray*) , or set each parameter one at a time using *fn*.bind(*fn*,*firstArg*)

Appendix: Tromp Diagrams

If you don't find functional programs easy to understand as text (and I don't), then this video shows John Tromp's visual lambda calculus notation. (Sadly, I still don't find these easy to understand as diagrams!)

Lucas Suss's annotated lambda diagrams at least have readable labels, although they're rotated 90 degrees relative to Tromp diagrams to give space for the labels.

Justine Tunney wrote a ridiculously compact lambda calculus interpreter (a 383 *byte* executable) with some highly compressed but very opaque programs.