More on
Functional Programming

It’s still programming

- Lots in common with imperative programming, of course
  - Issues of naming, scoping, types, expressions, control flow still arise
  - All languages must be scanned, parsed, etc
  - Functional languages make very heavy use of subroutines
  - Concurrency & nondeterminacy apply to these languages as much as imperative languages (concurrency is often much easier, for reasons we’ll see later)
- Boundaries tend to blur a bit
  - Functional features being added to imperative languages
    - Python can be written in a largely functional style
    - C# supports monads, a key feature of functional programming.
    - Functional features being added to VB, Java 8, even C++
    - Third-party library adds some functional features to FORTRAN
  - The most common logic language (Prolog) offers some imperative features
  - So do most LISP descendants
  - It’s straightforward to build a logic programming system in most functional languages

Pure functions

- In most functional languages, functions are *pure*—they have no side effects, always return the same value for a given set of parameters, and do not depend on anything that isn't passed as a parameter. This leads to some useful features:
  - Lazy evaluation
    - If it's more convenient to leave an expression unevaluated, I can do that; it'll have the same value whether I evaluate it now, or next Tuesday. Thus I can wait until I need it (if ever).
  - Memoization
    - Likewise, once I've evaluated it for a given set of parameters, I can store the value and look it up next time; it's not going to change if the server slows down, the vendor changes, etc.
  - Concurrency is trivial
    - If I have 100 function evaluations pending, and 100 processors, I can parallelize them completely; none depend on the others or can affect the others.
    - This allows divide-and-conquer, partial accumulation, associativity

Higher-order functions: Example

- Map function
  - `(map f L)` applies function f to each member of list L
  - `(map sqr '(1 2 3))` returns the list `(1 4 9)`
- Function as parameter to handle errors:
  - ```
    (define (safe-div numer denom if-error)
    #error handling function as parameter
        (if (= denom 0)
           (if-error) # divide by 0, call error function
          (quotient numer denom)))
    # otherwise it's safe, return int quotient
    ```
    ```

Customizing functions

- We can have complex library functions and customize them to what we need:
- ```
  (define (generic-safe-div numer denom if-err if-success)
     (if (= denom 0)
         (if-err)
         (if-success (div numer denom)))
  ```
- ```
  (define (my-safe-div numer denom)
      (generic-safe-div numer denom my-if-err my-if-succ))
  ```
- This is an example of *partial application.* We apply some of the parameters of a more complex function to produce a simpler function.
- Obviously, we can define multiple versions of my-if-err and my-if-succ, producing the exact behavior we want from my-safe-div. Or several different versions, depending on our needs.

Currying

- Named after the mathematician Haskell Curry
- Allows a multi-parameter function call to be treated as a series of 1-parameter calls.

```
(define (F x y z)
   # function body
 )
 # Suppose x = 5, y = 3.2, z = "ABC"
(define (f1 y z)
   (F 5 y z))  # f1 is a function of y & z
(define (f2 z)
   (f1 3.2 z)  # f2 is a function of z
(f2 "ABC")
```

- Obviously, we could have started with currying z, then y, leaving us with a function of x.
- This is useful in building a ‘custom’ version of a function
- Also, the map function requires a 1-parameter function passed to it. This lets us build one.

Other higher-order functions

- Among the most common:
- `(fold f s L):` f is a function that takes 2 parameters. The value s and the first item in L are passed to f; the result is passed with the second element of L, and so on. Typical uses: sum, min, max, etc.
  - ```
    (define (sum L) (fold + 0 L))
    (define (product L) (fold * 1 L))
    ```
- `(filter f L):` Return a new list, containing all elements of L for which f returns true.
    ```

Evaluation of function parameters

- Some languages use *normal order* - this is also known as *lazy evaluation.*
  - Function parameters are not evaluated until they are needed; once evaluated, they are memoized.
  - If a parameter is undefined, but also not needed—no harm, no foul.
  - This also allows defining a potentially infinite data structure—a list of all integers, for example. It only grows as big as the number of times the ‘next’ item is requested.
- Other (most) languages use *applicative order*—all parameters are evaluated before the function begins. Undefined parameters are a runtime error.
- Some languages allow a choice—generally evaluation can be forced if a language uses lazy evaluation, and a normal-order (lazy evaluation) version of Racket is available, though it usually uses applicative order.

Strictness & lazy evaluation

- Evaluation order can affect execution speed, but also program correctness
  - A program encountering a dynamic semantic error or infinite regression in an ‘unneeded’ subexpression under applicative-order evaluation may run successfully under normal-order evaluation
- A side-effect-free function is said to be strict if it is undefined (doesn’t terminate or encounters an error) if any of its parameters are undefined. Such a function can evaluate all of its arguments, so can safely use applicative order (results won’t depend on evaluation order)
- A function is nonstrict if it doesn’t impose this requirement –if it is sometimes defined even if some arguments aren’t.
- A language is strict if it’s defined such that functions are always strict; a language is nonstrict if it allows nonstrict functions
- If a language always evaluates in applicative order, then it is strict, since a call with an undefined argument will result in an error.
- Contrapositively, a nonstrict language cannot use applicative order; it must use normal order to avoid evaluating unneeded arguments
  - Standard ML, OCaml, and Scheme are strict; Miranda and Haskell are nonstrict

Implementation issues

- Trivial Update Problem
  - Suppose we have a data structure with 100K+ items in it. We need to change one of them.
  - In a procedural language, this is no problem; we identify the item & update it; or delete the old value & insert the new one. These change the data structure.
  - But in a functional language, data (including aggregated data) is immutable!
  - To handle this functionally, we return a new struct with the item added. How can this be done in a memory-efficient manner?
  - In practice, some functional languages (such as Racket) do allow procedural-style interactions with the (set! Item collection) function. But this loses the benefit of functional-style programming.
  - This can be mitigated significantly by careful choice of data structures, but this must be considered at language implementation time

Dealing with the real world

- In functional languages, functions have no side effects and always return the same result from the same parameters.
- But the real world is messy.
  - Each call to read_data() is expected to return a new item, and it doesn't have to be the same as the one before it; and sooner or later, it'll probably fail because of end-of-file.
  - Side effects include things such as:
    - Mark the invoice as paid
    - Update the database
    - Display the new score
    - Play the sound effect
    - Send the email
    - Print the document
    - Our clients just call these *effects,* and consider them to be the point of the program.
- Thus, we have to make some compromises to deal with the real world. But we can contain the state within specific types of containers.

Real world operations

- We may have:
  - An operation that might fail (Maybe)
  - An operation that might return different types of data on success or failure (e.g., either a record, or an error message). (Either)
  - An operation that we might need to wait to complete (Async)
  - Retrieving data that the user hasn’t typed in yet, or that we have to retrieve from a drive (Reader)
  - Data that needs to be sent to a display, which is a side effect (Writer)
  - A global context that a function might affect (State)
  - Or many, many other things.
- We’re going to look at the simplest 2: Maybe and Either.
  - Some languages use the name Option instead of Maybe.

Why do we do this?

- This allows our functions to remain pure.
- Instead of “this function returns a number, unless an error happens, in which case it raises an exception,” we have “this function returns a Maybe.”
- Instead of “this function returns the requested customer record, or an error message, or an exception from the database,” we have “this function returns an Either: On success, the requested customer record; on failure, a list of one or more error messages (strings).”
- Thus, we have a consistent interface, and can sequence our operations reliably.
- State and side effects are encapsulated in our container

Elevated worlds

- We are *not* going to go into the mathematical theory behind these.
- They're called *monads.* For programming-language purposes*, a monad requires 3 things:
  - A container to hold the data
  - A function (often called *lift*, or *pure*, or *return*) that moves data into those containers
  - A function (usually called *bind* or *chain*) for combining functions that take 'normal' data and return 'elevated' data (that is, data in a container)
- Our running example will be integer division.
- **Mathematically, a monad is a monoid in the category of endofunctors. What's the problem?*

Maybe

- Think of *Maybe* as a box. It might contain a value, it might contain nothing.
- We need a way to put data into the box. If x is a number, it's just the value of x; otherwise it's nothing:
```
(define (lift-maybe x)
     (if (number? x)
            (just x)
            nothing))
```
- We can rewrite our division function to return a Maybe:
- ```
  (define (safe-div x y)
    (if (= y 0)
        nothing
        (just (quotient x y))))
  ```
- And yes, we could add logic to check they're both numbers (or both integers) first, and return nothing if either test fails
- What's the advantage? It's a pure function again. It no longer returns perhaps a number, perhaps nothing. It always returns a Maybe.
  ```

‘world-crossing’ functions

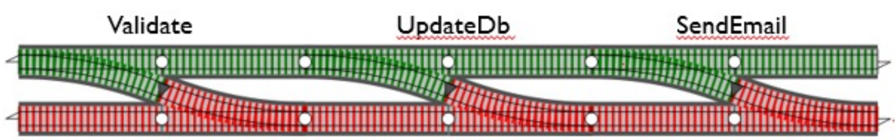
- But now we have a problem.
- Safe-div takes in a number and returns a Maybe. Suppose we’re going to pass the result to some other function. That function expects a number, not a Maybe. Do we have to modify that function to take a Maybe as well? If we’ll have to rewrite all our functions, it’s not worth it.
- No, we need a function to *bind* or *chain* function calls together:
-
```
(define (chain f m) ; m = a maybe value
  (if (nothing? m)
      nothing
      (f (from-just 0 m))))
```
- Let’s unpack this….

What’s happening here?

- If an earlier operation failed (m is nothing), then we can’t continue; return nothing at once.
- If we have a value, we need to remove it from *just* (which is needed to indicate this is a Maybe value).
- But, when you’re writing general code, it’s possible you’re calling from-just on nothing, or something that isn’t just a value.
- So we have to specify what to return in that case.
  - Here we return 0. Returning #f is another popular choice.
- If m is just a value, pass it to f; otherwise pass 0 to f.
  - Yes, in this case, we know m isn’t nothing, but the function is written for the general case.

Parallel tracks

- In effect, we have 2 parallel tracks: A success track and a failure track.
- If we ever get back nothing, for whatever reason, the remaining chained functions are bypassed and never called.



- Most languages have syntactic sugar to avoid nesting multiple function calls. For example, suppose x and y in safe-div could be either numbers or expressions, in some convenient notation. We have a function called eval that returns a numeric value (Maybe).

A more powerful division routine

- ```
  (define (safe-div-2 x y)
    (do
        [m <- (eval x)]
        [n <- (eval y)]
        (safe-div m n)))
  ```
- This syntax masks the nested calls to chain. If either call to eval returns nothing, safe-div-2 returns nothing at once; the remaining actions are skipped.
  ```

The Either type

- An Either is similar to a Maybe, except instead of holding just a value or nothing, it definitely holds something.
- That something is labeled as a success or failure. The data type it holds can differ based on that label:
- ```
  (define (safe-div x y)
    (if (and (number? x) (number? y))
        (if (= y 0)
            (failure "division by 0 error")
            (success (quotient x y)))
        (failure "safe-div: x or y not a number")))
  ```
- Likewise, we can define a failure to hold a list of strings (error messages) if we prefer, or anything else we feel like.
  ```

Bind with Either

- ```
  (define (chain f e) ; e = an either value
    (if (failure? e)
        e  ; return the failure we got, bypass f
        (f (from-success 0 e))))
  ```
- In this case, if we’re calling f, we know e isn’t a failure, so it’s a success, so the 0 parameter won’t be needed, but the function has the parameter and we have to specify a value, even if it’s never used.
- Using these tools, we can move our error handling inside the functions, and so don’t need try/except, or elaborate if/else logic to deal with possible errors.
  ```

Lists as monads

- Likewise, lists are
  - A container
  - With functions that move data into a list
  - And functions that allow applying single-parameter functions to them (map, filter, etc)
- Therefore lists are monads
- These can be used as building blocks to build more abstract data structures
  - OCaml's popularity in the financial services industry is largely due to its type system, allowing complex behavior & context to be managed reliably

Application: Parser Combinators

- A common application is to build a parser.
- We can use higher-order functions to build up combinations
- So, for example, begin with a function that can parse a single character.
  - Pass in the character to be matched, and the string to be parsed.
  - If the first character matches, return success, the matched character as a singleton list, and the rest of the input string
  - Otherwise return a failure, error message in a singleton list, and the unchanged input.
  - Note the pattern: Success or failure, and a list consisting of a singleton sublist, and the remaining input as a string

Parse one character

```
(define (parse-specific-char ch input-str)
   (let
      ([chars (string->list input-str)])
     (if (eq? ch (first chars))
       (success (list (list (first chars)) (list->string(rest chars))))
       (failure (list (list "not a match with " ch) input-str)))))
```

- But of course, we want to be able to match more than one character. We need to be able to match any of a group:

```
(define (any opt-list input-str)
       ; returns which character in opt-list, if any, first char of input-str matches
   (let
     ([chars (string->list input-str)]) ; turn input-str into list of chars

         (define (iter opt-list ch)
         (if (empty? opt-list)
               (failure (list '("no match") input-str))
               (if (eq? (first opt-list) ch)
                    (success (list (list ch) (list->string (rest chars))))
                    (iter (rest opt-list) ch))))

    (iter opt-list (first chars))))
```

Parsing character types

```
(define (parse-decimal-digit input-str)
   (any (string->list "1234567890")
```

- We can also combine types; another higher-level function will make that easier:

```
(define (either-or f1 f2 input)
   (let
       ([result (f1 input)])
       (if (success? result)
          result
          (f2 input))))
```

- Apply f1; return result if successful; if not successful, apply f2.
- So once we have (parse-alphabetic) and (parse-numeric) functions,

```
(define (parse-alphanumeric input)
     (either-or parse-alphabetic parse-numeric input))
```

Sequencing operations

- We may have some constructs that we expect to be something followed by a slightly different something
  - "An identifier is an alphabetic character or underscore, followed by any number of alphanumeric or underscore characters."
- For this we need a higher-order function (and-then f1 f2 input)
  - Apply f1. If f1 fails, return failure immediately.
  - If f1 succeeds, apply f2
  - If f2 succeeds, return success
  - If f1 succeeds and f2 fails, is that good enough? Sometimes yes (an identifier can be a single alphabetic character), sometimes no (in Pascal, a pointer dereference operator ^ must be followed by an identifier).
  - Add parameter to the function whether success on f2 is required.

Combining operations = clear code

- Once we have the tools to connect things, the code becomes simple:

```
(define (parse-alphanumeric-char input)
  (either-or parse-alphabetic-char parse-numeric-char input))
(define (alphanumeric-string input-str)
  (many parse-alphanumeric-char input-str))
(define (alpha-string input-str)
  (many parse-alphabetic-char input-str))
(define (add-op input-str)
  (any (string->list "+-") input-str))
(define (nonzero-digit input-str)
  (any (string->list "123456789") input-str))
(define (decimal-digit input-str)
  (any (string->list "0123456789") input-str))
(define (hex-digit input-str)
  (any (string->list "0123456789abcdefABCDEF") input-str))
(define (digit-string input-str)
  (many decimal-digit input-str))
```

Concurrency in Functional Languages

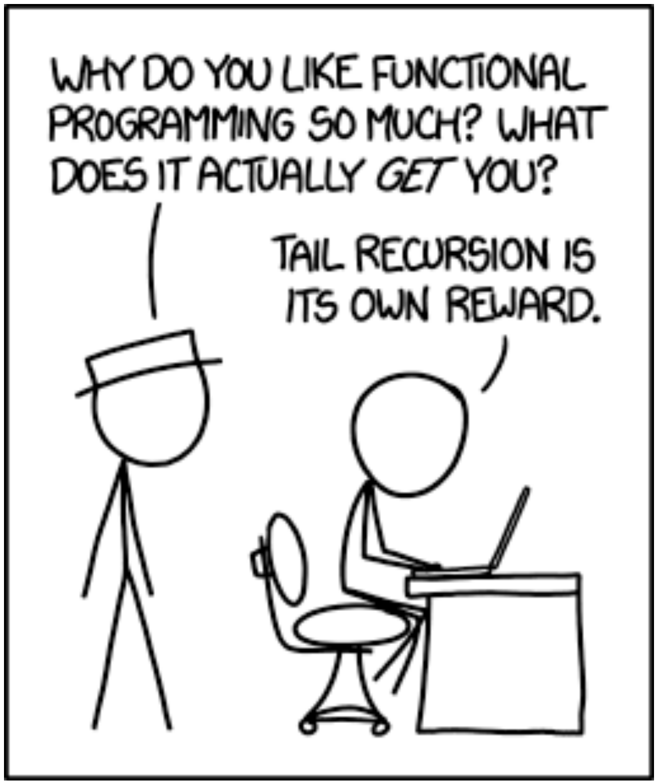
- Because data is immutable, we can’t rely on mutexes, semaphores, etc.
- Most functional languages use an explicit *message-passing* protocol (set of functions) to allow inter-thread communication
- But because data is immutable, the most common synchronization problem—1 thread modifying data just as another thread is reading it, or 2 threads trying to write data at the same time—simply doesn’t occur.

Why use functional programming?

- Functions can be composed and treated as modules; no side effects, results are always the same for a particular input.
  - As long as rules on type compatibility are met, this is always safe.
- Easier to reason about program behavior & prove correctness
- Functional programs tend to be more compact than procedural code
- Since there's no shared mutable state, there's no interaction between parts of a program except what's defined via function calls.
  - So undocumented side effects, misordered updates, dangling or uninitialized references simply don't occur.

So why aren't we using functional programming all the time?

- Most programmers start out learning procedural languages, so a functional style *looks* hard.
- A lot of online tutorials use Haskell, which most programmers aren't familiar with and has an unusual syntax.
- Functional languages, while easier to write & debug, can be harder to read.
- Many functional languages aren't fully portable, sometimes lacking in library packages, or (especially) debugging & profiling tools.
  - Though this is getting better—Rust is a particular example.
- Some problems (e.g. user interaction) map more directly to procedural style
- If raw execution speed is a criterion, functional languages will lag
  - Though again, this is getting better, and the ease of maintenance of functional languages is a plus. F#, Rust, Racket offer good performance.



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

- More seriously, though, it lets you reason equationally about your code. Build correct functions, compose them correctly, and you have a provably correct program.