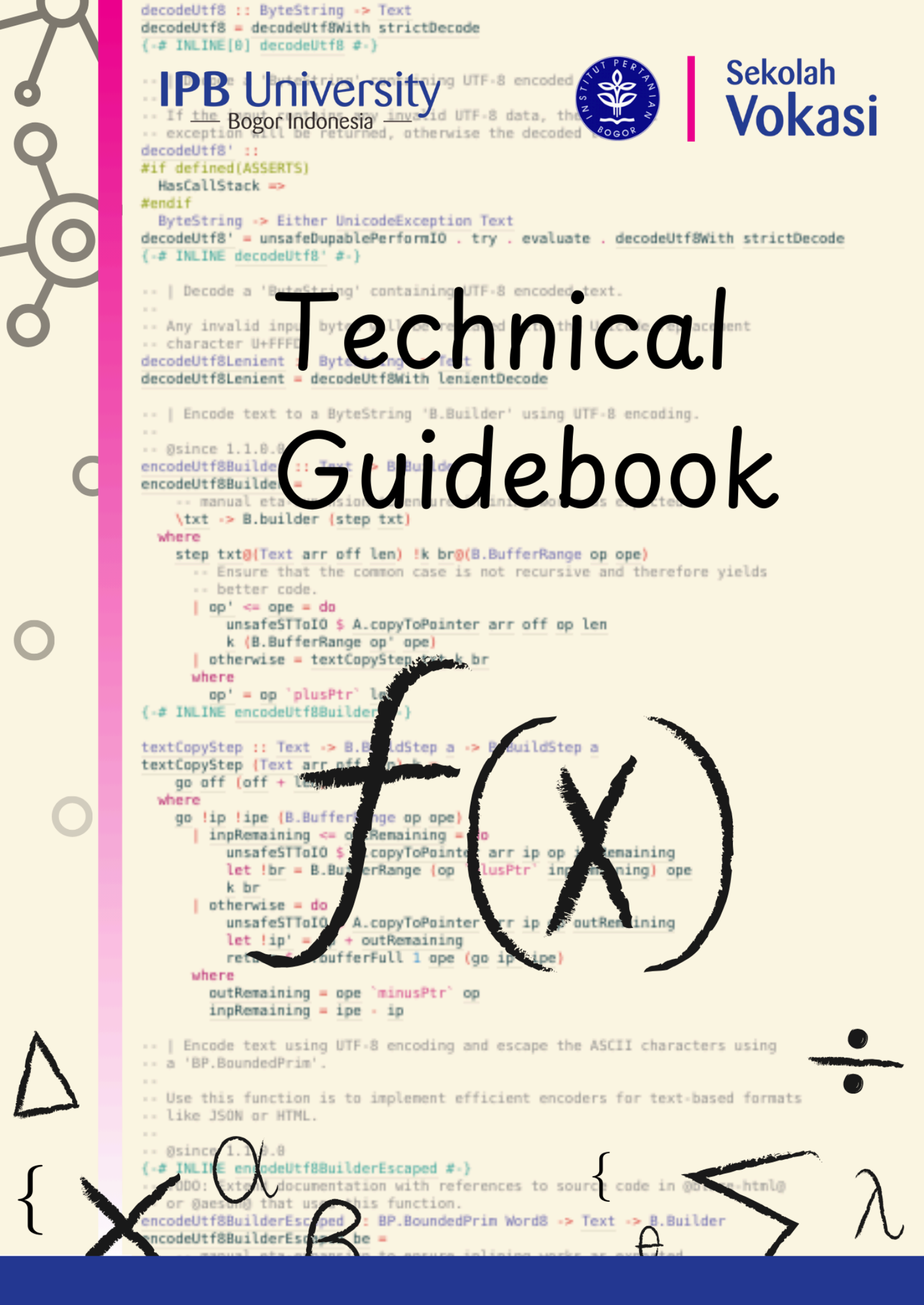
IPB University
Bogor Indonesia

Sekolah Vokasi

Technical Guidebook



Functional Programming Concepts

Table of Contents

Abstract

Functional programming is a programming paradigm in which programs are made by applying and composing functions. A functional programmer writes programs by writing small, single-purpose functions and composing them into larger ones. This
paradigm emphasizes immutability, pure functions, and composition of higher-order functions that promote greater modularity than conventional methods. This guide is written to educate young programmers on an alternative to the prevalent imperative or object oriented paradigms and lead them to learn more themselves.

Keywords: functional programming, concepts, guide

The tools we use have a profound influence on our thinking habits, and, therefore, on our thinking abilities.

- Edsger Dijkstra (Dijkstra, 1982)

Introduction

A paradigm is a way of doing and thinking. In terms of programming, that translates to the way you write code and how you reason about code. There are almost as many paradigms as there are programmers. However, not all of those paradigms are familiar to you, perhaps because they are not taught in your curriculum.

The paradigm most programmers are familiar with is procedural programming. Python comes to mind, where it has consistently shown to be a popular choice at the time of this writing (*PYPL PopularitY of Programming Language Index*, n.d.). We can also think of all C-like, curly brace languages such as Javascript, PHP, etc. as fully supporting this paradigm. In those languages, programs are sequences of statements that mutate some state. Those statements can be grouped into procedures, thus the name. The following is an operation written in a procedural pseudocode that computes the sum of integers from a list of integers. We will use this operation and others like it to demonstrate different functional programming concepts and compare paradigms.

```
sumOfInts(ints: []integer): integer
{
    integer result := 0;
    for i in range(0, length(ints))
    {
        result = result + ints[i];
    }
    return result;
}
```

The first statement is an assignment where the `result` variable is initialized with a starting state, in this case the integer 0. The next statement iterates each index of the list and adds each integer into the result. After we have iterated through all elements in the list, we return the result to the procedure's caller.

We can observe a few properties latent to this paradigm. First, mutations happen regularly, in an unmanaged manner. Second, the programmer must explicitly specify the flow of control. Each statement in the procedure can be considered to be a command, which is why another term for this paradigm is imperative programming.

This guide, however, concerns itself with a paradigm called functional programming. In this paradigm, programs are composed of functions. Those functions are also made of functions, all the way down to language primitives. Instead of mutating state with sequences of statements, a program is executed by evaluating expressions (Harrison, 1997). Languages that fully support this paradigm such as OCaml, Haskell, etc. double down on this execution model by treating everything as an expression.

The following is the same operation written in a functional pseudocode.

```
sumOfInts(ints) : []integer -> integer =
    match ints
        [] then 0
        [x] then x
        [x, ..xs] then x + sumOfInts(xs)
```

Firstly, we are introduced to pattern matching. Pattern matching is a control flow construct that checks if the structure of an expression matches a particular pattern. Secondly, the function calls itself. This construct is called recursion. Each of these constructs will be explained in the following sections.

Let us compare the functional approach with the following approximation in the procedural language.

```
sumOfInts(ints: []integer): integer
{
   if (length(ints) == 0) {
      return 0;
   } else if (length(ints) == 1) {
      return ints[0];
   } else {
      return ints[0] + sumOfInts(ints[1..]);
   }
}
```

In the approximation, we must explicitly check the length of the list and explicitly return to the procedure's caller with the return statement. In the functional language, all of that is implicitly taken care of by the language, allowing for a more declarative style of programming that focuses on what the solution is instead of how the solution is done.

The concepts in this introduction will be further explained in the rest of the guide. The goal is not to discredit other paradigms. Each paradigm has their flaws and strengths, suited for different tasks. It is up to you, the reader, if the benefits explained in this guide suits what you're trying to do. To quote (Harrison, 1997), horses for courses.

If you have any questions, do not hesitate to send an email to bramadityaw@duck.com. I am happy to help another programmer learn.

Discussion

Purity and Side Effects

A function is pure if it computes its return value without side effects. It is called pure because it behaves like functions in mathematics, where the only noticeable effect is simply the transformation of their input to their output. A side effect is an effect that is other than what is stated in the function's signature. We will use an earlier example, the `sumOfInts` procedure.

```
sumOfInts(ints: []integer): integer
{
    integer result := 0;
    for i in range(0, length(ints))
    {
        result = result + ints[i];
    }
    return result;
}
```

Inside the **`for`** loop is a side effect called mutation, and you most certainly have used it before. Mutation is when a program changes the state of some variable. When a variable is able to be mutated, we call that variable to be mutable. As a noun, we refer to this side effect as mutability.
Side effects like mutation can be problematic. Consider the signature of this procedure.

```
add(x: integer, y: integer): integer
```

An add procedure

Given that signature, what is the return value of `add(3, 5)`? Because of the possibility of this procedure using some global state, you cannot be sure. There is no guarantee that the procedure does not call a random number generator, or something like the following.

```
integer num_calls = 1;
add(x: integer, y: integer): integer
{
   integer result = (x + y) * num_calls;
   num_calls = num_calls + 1;
   return result;
}
```

Problematic use of mutation

One way to remove problems caused by this side effect is to practice immutability. It is the inverse of mutability. Immutability is simply not changing the state of a variable. When you practice this technique, the flow of data from function to function becomes more transparent and trackable only by looking at the signature of the function.
A programmer does not need to care about how a function is implemented if it performs no side effects. Its signature is enough to inform a consumer of an interface on its behavior. This property of a pure function is called referential transparency. Functional programmers rely on

this property heavily as it guarantees that given the same input, a function will return the same output.
In a functional language, everything is immutable by default. When you define a variable, you cannot change it. If you assign `x = 1`, x will continue to be 1 until you edit the source code.
This does not mean that functional languages do not support mutation. Languages like SML and OCaml have special syntax for handling mutable data (*Mutability and Imperative Control Flow · OCaml Documentation*, n.d.). Unison has the `Ref` data type that wraps the underlying mutable data inside an immutable data structure and manipulates them using regular functions (*Unison Share - Ref Type*, n.d.).
Another common side effect is input output. This side effect is done when a program interacts with the outside world, such as file access, user input, database queries, etc. The reason this is a side effect is that the input is non-deterministic. A file can change contents or even be deleted by other programs. A user may input different things to a prompt, or click different sequences of buttons. Another example is in Java, where the URL class performs a DNS lookup when you want to compare two URL objects (*URL (Java Platform SE 8)*, n.d.). The lookup will block execution until both URLs are resolved. This has some issues, such as execution being dependent on device network speed.
As you may already think, a pure program is a useless program. To make a program useful, it must be able to interact with the outside world and perform effects.
A way to do that is to isolate them to a dedicated part of the program. What we would end up with is a mostly pure codebase with its effects bundled to a controlled section of the program. Programs that do this transform their side effects into managed effects. This approach is the way purely functional languages such as Haskell and Elm take.
Another way to do it is through an algebraic effect system. This mechanism enables effects to be part of function signatures and reduces the information burden of API consumers. This is a newer approach and is the way languages such as Unison or Koka manage effects.

Pattern Matching

Pattern matching is syntax for binding an expression to one or more variables. The simplest form of pattern matching is the variable pattern, shown below.

```
let num = 8 * 2
```

Here is a simple term declaration. The pattern `num` matches the result of computing `8 * 2`. Put another way, the variable is bound to whatever is in the right hand side of the declaration. num can then be used as a term for other parts of the program to represent `8 * 2`.
But variables aren't the only valid patterns. We can also use literals.

```
let 7 = 8 * 2
```

7 is a valid pattern, because it belongs to the same type, but it will not match. 7 will continue to be 7 throughout the language. This form of pattern matching, usually called a let binding, is most useful when destructuring types such as tuples and records.

```
let (a, b) = pair
```

```
let {id, name, age} = user
```

Pattern matching can also be used in match-cases. They are a generalization of if-then-else that work for data types other than boolean.

With this construct, we can even define **if** as a function that takes in three arguments. The first argument is a predicate, returns the second argument if it is true, and returns the third argument if it is false.

```
let iff(pred, a, b) : boolean, a, a -> a =
  match pred to
    True then a
    False then b
```

The type **a** is a type variable, explained further in the Polymorphism section. For the purpose of this section, it means that the variables `a` and `b` must be of the same type, whatever it is.
The `pred` variable, which is called the scrutinee, is compared to two patterns: the `True` case and the `False` case. The function will evaluate to the right hand side of the case where the pattern matches.
This control flow construct is built into many languages, functional or not. They are most useful in languages with algebraic data types, such as Rust or Elm. Proposals are being considered for C++ and Javascript as of this time of writing. The reason this construct is so popularly included in a lot of language specifications is because it allows for a more declarative way of writing code.

Recursion

Functional languages generally don't offer looping constructs. When a function needs to evaluate its body more than once, it could call itself. The reason is that for most use cases, recursion is enough to express loops. Here is the factorial function, a classic example to demonstrate recursion.

```
let factorial(n) : integer -> integer =
    if n == 0 || n == 1 then 1
    else n * factorial(n-1)
```

We first start with the base case, to prevent the function from recursing indefinitely. If the base case is not satisfied, it will then recurse.

First Class Functions

Languages that support functional programming can treat functions as values. That means functions can be accepted as arguments to other functions, and functions can also return a function. This allowed for greater modularity, and the reason will be explained with examples.
Before we go any further, we have to talk about the arithmetic operator `+.` `+` is a binary operator that adds two numbers which results in another number. From that, we can generalize the notion of a binary operator as a function with two arguments, with the left hand side being the first argument and the right hand side being the second. This holds true for other arithmetic operators, such as `-`, `*` and `/`, and also for other binary operators like `&&` and `||` for booleans, `>=`, `==`, and `>=` for comparison, and `&` and `|` for bitwise operations.
Let us recall the `sumOfInts` function from the beginning of this guidebook.

```
let sumOfInts(ints) : [integer] -> integer =
    match ints to
        [] then 0
        [x] then x
        [x, ..xs] then x + sumOfInts(xs)
```

Say we want to instead calculate the product of integers. We may define a function like so.

```
let productOfInts(ints) : [integer] -> integer =
    match ints to
        [] then 1
        [x] then x
        [x, ..xs] then x * productOfInts(xs)
```

Notice how the functions body is similar in structure. When we encounter an empty list, we return a base value. When the list has only one element, we return that element. When the list has more than one element, we do an operation on the first element and the recursive result of the current function applied to the rest of the list. This has a name in the functional programming tradition, fold (Harrison, 1997).

```
let fold(list, base, op) :
    [integer], integer, (integer, integer -> integer) -> integer =
    match list to
        [] then base
        [x] then x
        [x, ..rest] then fold(rest, op(base, x), op)
```

`fold` generalizes the two previous functions and makes them available for other integer operations. We could then define both `sumOfInts` and `productOfInts` in terms of `fold`.

```
sumOfInts(ints) = fold(ints, 0, +)
productOfInts(ints) = fold(ints, 1, *)
```

We can also return functions. As an example, we will make a logging function that returns a different implementation depending on the argument passed

```
let logger(env) =
    match env to
        "dev" then print
        "prod" then writeLog
```

`print` and `writeLog` are functions containing the implementation of the logger. `print` writes to standard output, while `writeLog` writes to a predefined file path. Side note, notice how this function is pure, but the result is not.

Polymorphism

What if we want to fold other than integers? Recall the definition of fold below.

```
let fold(list, base, op) :
    [integer], integer, (integer, integer -> integer) -> integer =
    match list to
        [] then base
        [x] then x
        [x, ..rest] then fold(rest, op(base, x), op)
```

Say we want to fold over a list of predicates, and we would like to know if they are all true or at least one predicate is true. We can define two functions like so.

```
allTrue(preds) = foldBool(preds, True, &&)
oneTrue(preds) = foldBool(preds, False, ||)
```

`foldBool` is defined like so.

```
let foldBool(list, base, op) :
    [boolean], boolean, (boolean, boolean -> boolean) -> boolean =
    match list to
        [] then base
        [x] then x
        [x, ..rest] then fold(rest, op(base, x), op)
```

Notice how the `foldBool`'s body is identical to `fold`. The only thing different here is the types.

We introduce a mechanism called polymorphism. Polymorphism allows a function to deal with more than one data type. The form of polymorphism we discuss here is parametric polymorphism. A function that is parametrically polymorphic uses generic types that are stand-ins for concrete types.

Let us redefine `fold` to be polymorphic.

```
let fold(list, base, op) :
    [a], b, (a, b -> b) -> b =
    match list to
        [] then base
        [x] then x
        [x, ..rest] then fold(rest, op(base, x), op)
```

The types `a` and `b` are called type variables. If a normal variable is a stand-in for a value, a type variable is a stand-in for a type. `a` and `b` could stand-in for the same or different types, because they are universally quantified. The function fold can now operate for other types other than boolean and integer, and the result of fold need not be the same type as in the list.

We can redefine `allTrue` and `oneTrue` with the new polymorphic `fold`. Notice how nothing changes apart from the function name.

```
allTrue(preds) = fold(preds, True, &&)
oneTrue(preds) = fold(preds, False, ||)
```

Type Inference

Type inference is a mechanism where the data type of an expression or variable is able to be deduced only by its abstract syntax. From a programmer's perspective, this simplifies writing code as you don't need to check the return types and write in the declaration yourself. This is the underlying mechanism for parametric polymorphism, as it facilitates replacing type variables into the concrete type.

You will see this mechanism in programming languages descended from or inspired by ML, such as OCaml or Rust. It could also be found in C++ (the **`auto`** keyword), Go (the `:=` operator), Kotlin (the **`val`** keyword), and many others. For further reading, you may research the Hindley-Milner type inference algorithms.

Currying

A previous section mentions that functions can return functions. This allows for a technique called currying. Currying is when a function takes multiple arguments by taking a single argument and applies each argument to its result one at a time.
For example, let us take the polymorphic fold earlier and redefine it into curried form.

```
let fold(list) :
    [a] -> b -> (a, b -> b) -> b =
    fn(base)
        fn(op)
            match list to
                [] then base
                [x] then x
                [x, ..rest] then fold(rest, op(base, x), op)
```

In this redefinition, fold only takes one argument. It then returns an anonymous function that takes the base value. That anonymous function will then take a function and return the result of fold. We define a function like this to take advantage of partial application.
To fully take advantage of partial application, we have to reorder the arguments to conform to a convention known to functional programmers as data-last.
Let us declare the types first.

```
fold : (a, b -> b) -> b -> [a] -> b
```

Data last is called such because the list of as, the data, is the last argument. Second last is the base value, and the first argument is the folding function.
Here is the data-last definition.

```
let fold(op) :
    (a, b -> b) -> b -> [a] -> b =
    fn(base)
        fn(list)
            match list to
                [] then base
                [x] then x
                [x, ..rest] then fold(rest, op(base, x), op)
```

There are a few reasons for adopting this convention. First, it allows a more concise style of programming called point free style. In this style, functions don't need to specify their arguments at their definition.
We can alternatively define the boolean folding functions in this style as follows.

```
allTrue = fold(&&)(True)
oneTrue = fold(||)(False)
```

Second, as mentioned, we can use this for partial application. fold can be supplied with a function first, and the function that is returned can be defined as a more specific fold, with the base value and list supplied later.
In ML-descended languages such as Haskell, OCaml etc. functions are curried by default, and by consequence has less cumbersome syntax for this technique.

The following example is written in Haskell.

```
fold :: (a -> b -> b) -> b -> [a] -> b
fold op base list =
    case list of
        [] -> base
        [x] -> x
        (x:rest) -> fold rest (op base x) op
```

As we can see, each argument is positioned to the right of the function name, separated by spaces.
This technique is named after Haskell Curry, which is also where Haskell got its name. It was originally conceived to circumvent a limitation of lambda calculus that only allows functions to accept one argument (Harrison, 1997).

Conclusion

Functional programming is a paradigm that emphasizes the use of pure functions, immutability, and declarative code to create robust and maintainable software. Central to this paradigm is the concept of purity and side effects, where pure functions are those that produce the same output for the same input without modifying any external state or causing observable side effects. This characteristic ensures predictability and simplifies debugging and testing. Side effects, such as modifying global variables or performing I/O operations, are typically minimized or isolated in functional programming to maintain referential transparency. Pattern matching further enhances the expressiveness of functional languages by allowing developers to deconstruct data structures and handle different cases concisely, often replacing verbose conditional logic.
Another cornerstone of functional programming is the use of first-class functions, which treat functions as values that can be passed as arguments, returned from other functions, or assigned to variables. This enables higher-order functions, such as map, filter, and reduce, which abstract common patterns of computation and promote code reuse. Polymorphism, whether parametric or ad-hoc, allows functions to operate on multiple types, enhancing flexibility and generality. For instance, parametric polymorphism enables the creation of generic data structures and algorithms, while type classes or interfaces support ad-hoc polymorphism by defining shared behavior across types. Type inference further reduces boilerplate code by allowing the compiler to deduce types automatically, improving readability without sacrificing type safety. Finally, currying transforms functions with multiple arguments into a sequence of single-argument functions, enabling partial application and fostering composability. Together, these concepts form the foundation of functional programming, enabling developers to write concise, modular, and maintainable code that aligns with mathematical principles and modern software engineering practices.

References

Dijkstra, E. W. (1982). *Selected Writings on Computing*. Springer New York. https://www.cs.utexas.edu/~EWD/ewd04xx/EWD498.PDF

Harrison, J. (1997). Introduction to Functional Programming. *Lecture Notes, Cambridge*. https://www.cl.cam.ac.uk/teaching/Lectures/funprog-jrh-1996/all.pdf

Mutability and Imperative Control Flow · OCaml Documentation. (n.d.). OCaml. Retrieved February 26, 2025, from https://ocaml.org/docs/mutability-imperative-control-flow

PYPL PopularitY of Programming Language index. (n.d.). Retrieved February 24, 2025, from https://pypl.github.io/PYPL.html

Unison Share—Ref Type. (n.d.). Retrieved February 26, 2025, from https://share.unison-lang.org/@unison/website/code/main/latest/namespaces/lib/base/mutable/;/types/Ref

URL (Java Platform SE 8). (n.d.). Retrieved March 2, 2025, from https://docs.oracle.com/javase/8/docs/api/java/net/URL.html#equals-java.lang.Object-