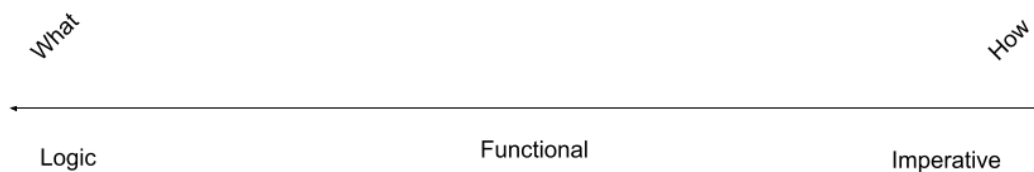


The Daily PL - 7/13/2023

Introduction to Functional Programming

As we said at the beginning of the semester when we were learning about programming paradigms, FP is very different than imperative programming. In imperative programming, developers tell the computer how to do the operation. While functional programming is *not* logic programming (where developers just tell the computer *what* to compute and leave the *how* entirely to the language implementation), the writer of a program in a functional PL is much more concerned with specifying what to compute than how to compute it.



Four Characteristics of Functional Programming

There are four characteristics that epitomize FP:

1. There is no state
2. Functions are central
 1. Functions can be parameters to other functions
 2. Functions can be return values from other others
 3. Program execution *is* function evaluation
3. Control flow is performed by recursion and conditional expressions
4. Lists are a fundamental data type

In a functional programming language, there are no variables, per se. And because there are no variables, there is no state. That does *not* mean there are no names. Names are still important. It simply means that names refer to expressions themselves and not their values. The distinction will become more obvious as we continue to learn more about writing programs in functional languages.

Because there is no state, a functional programming language is not *history sensitive*. A language that is *history sensitive* means that results of operations in that language can be affected by operations that have come before it. For example, in an imperative programming language, a function may be history sensitive if it relies on the value of a global variable to calculate its return value. Why does that count as history sensitive? Because the value in the global variable could be affected by prior operations.


A language that is not history sensitive has *referential transparency*. We learned the definition of referential transparency before, but now it might make a little more sense. In a language that has referential transparency, the same function called with the same arguments generates the same result no matter what operations have preceded it.

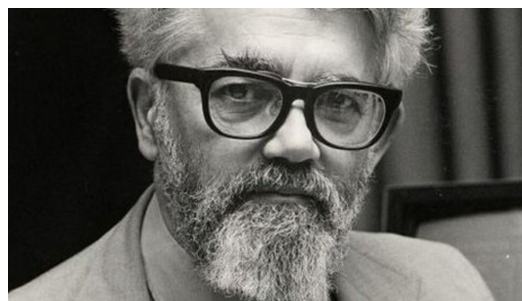
In a functional programming language there are no loops (unless they are added as syntactic sugar) -- recursion is *the* way to accomplish repetition. Selective execution (as opposed to sequential execution) is accomplished using the *conditional expression*. A conditional expression is, well, an *expression* that evaluates to one of two values depending on the value of a condition. That a conditional statement can have a value (thus making it a conditional expression) is relatively surprising for people who only have experience in imperative programming languages. Nevertheless, the conditional expressions is a very, very sharp sword in the sheath of the functional programmer.

A functional program is a series of functions and the execution of a functional program is simply an evaluation of those functions. That sounds abstract at this point, but will become more clear when we see some real functional programs.

Lists are a fundamental data type in functional programming languages. Powerful syntactic tools for manipulating lists are built in to most functional PLs. Understanding how to wield these tools effectively is vital for writing code in functional PLs.

The Historical Setting of the Development of Functional PLs

The first functional programming language was developed in the mid-1950s by **John McCarthy**.  ([https://en.wikipedia.org/wiki/John_McCarthy_\(computer_scientist\)](https://en.wikipedia.org/wiki/John_McCarthy_(computer_scientist))). At the time, computing was most associated with mathematical calculations. McCarthy was instead focused on artificial intelligence which involved symbolic computing. Computer scientists thought that it was possible to represent cognitive processes as lists of symbols. A language that made it possible to process those lists would allow developers to build systems that work like our brains, or so they thought.



McCarthy started with the goal of writing a system of meta notation that programmers could attach to Fortran. These meta notations would be reduced to actual Fortran programs. As they did their work, they found their way to a program representation built entirely of lists (and lists of lists, and lists of lists of lists, etc). Their thinking resulted in the development of Lisp, a *list* processing

language. In Lisp, data are lists and programs are lists. They showed that list processing, the basis of the semantics of Lisp, is capable of universal computing. In other words, Lisp, and other list processing languages, is/are Turing complete.

The inability to execute a Lisp program efficiently on a physical computer based on the von Neumann model has given Lisp (and other functional programming languages) a reputation as slow and wasteful. (*N.B.*: This is *not true* today!) Until the late 1980s hardware vendors thought that it would be worthwhile to build physical machines with non-von Neumann architectures that made executing Lisp programs faster. Here is an image of a so-called Lisp Machine.



LISP

We will not study Lisp in this course. However, there are a few aspects of Lisp that you should know because they pervade the general field of computer science.

First, you should know CAR, CDR and CONS -- pronounced *car*, *could-er*, and *cahns*, respectively. CAR is a function that takes a list as a parameter and returns the first element of the list. CDR is a function that takes a list as a parameter and returns the tail, everything but the head, of the list. CONS takes two parameters -- a single element and a list -- and returns a new list with the first argument appended to the front of the second argument.

For instance,

```
(car (1 2 3))
```

is `1`.

```
(cdr (1 2 3))
```

is `(2 3)`.

Second, you should know that, in Lisp, all data are lists *and* programs are lists.

```
(a b c)
```

is a list in Lisp. In Lisp, `(a b c)` could be interpreted as a list of *atoms* `a`, `b` and `c` or an invocation of function *a* with parameters *b* and *c*.

Lambda Calculus

Lambda Calculus is the theoretical basis of functional programming languages in the same way that the Turing Machine is the theoretical basis of the imperative programming languages. The Lambda Calculus is nothing like "calculus" -- the word calculus is used here in its strict sense: **a method or system of calculation** [https://en.wikipedia.org/wiki/Calculus_\(disambiguation\)](https://en.wikipedia.org/wiki/Calculus_(disambiguation)). It is better to think of Lambda Calculus as a programming language rather than a branch of mathematics.

Lambda Calculus is a model of computation defined entirely by function application. The Lambda Calculus is as powerful as a Turing Machine which means that anything computable can be computed in the Lambda Calculus. For a language as simple as the Lambda Calculus, that's remarkable!

The entirety of the Lambda Calculus is made up of three entities:

1. Expression: a *name*, a *function* or an *application*
2. Function: $\lambda <name> . <expression>$
3. Application: $<expression> <expression>$

Notice how the elements of the Lambda Calculus are defined in terms of themselves. In most cases it is possible to restrict *names* in the Lambda Calculus to be any single letter of the alphabet -- `a` is a name, `z` is a name, etc. Strictly speaking, functions in the Lambda Calculus are anonymous -- in other words they have no name. The *name* after the λ in a function in the Lambda Calculus can be thought of as the parameter of the function. Here's an example of a function in the Lambda Calculus:

```
 $\lambda x . x$ 
```

Lambda Calculiticians (yes, I just made up that term) refer to this as the *identity* function. This

function simply returns the value of its argument! But didn't I say that functions in the Lambda Calculus don't have names? Yes, I did. *Within* the language there is no way to name a function. That does not mean that we cannot assign semantic values to those functions. In fact, associating meaning with functions of a certain format is *exactly* how high-level computing is done with the Lambda Calculus.

The Daily PL - 7/18/2023

Lambda Calculus -- Where Were We?

Remember that we said Lambda Calculus is the theoretical basis of functional programming languages in the same way that the Turing Machine is the theoretical basis of the imperative programming languages. Again, don't freak out when you hear the phrase "calculus". As we said in class, it is better to think of the Lambda Calculus as a programming language rather than a branch of mathematics.

Lambda Calculus is a model of computation defined entirely by function application. The Lambda Calculus is as powerful as a Turing Machine which means that anything computable can be computed in the Lambda Calculus. For a language as simple as the Lambda Calculus, that's remarkable!

Remember that the entirety of the Lambda Calculus is made up of a small number of entities:

1. Expression: a *name*, a *function* or an *application*
2. Function: $\lambda\langle name \rangle \cdot \langle expression \rangle$
3. Application: $\langle expression \rangle \langle expression \rangle$

We made the point in class that, without loss of generality, we will assume that all names are single letters from the alphabet. In other words, if you see two consecutive letters, e.g., *ab*, those are two separate names.

Bound and Free Names and the Tao of Function Application

Because the entirety of the Lambda Calculus is function application, it is important that we get it exactly right. Let's recall the simplest example of function application:

$$(\lambda a. a) x = [x/a]a = x$$

The $[x/a]a$ means "replace all instances of *a* with *x* in whatever comes after the λ ". This is so easy. What about this, though?

$$(\lambda a. \lambda b. ba) b$$

The first thing to realize is that the *b* in the expression that is the body of the nested lambda function is *completely* separate from the *b* to which the lambda function is being applied. Why is that? Because the *b* in the nested lambda function is the "parameter" to that function. So, what are we to do?

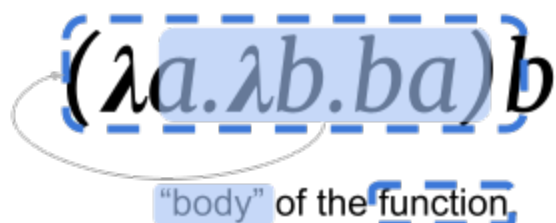
First, let's step back and consider some definitions: *free* and *bound* names. Loosely speaking, a

name is *bound* as soon as it is used as a parameter to a lambda function.



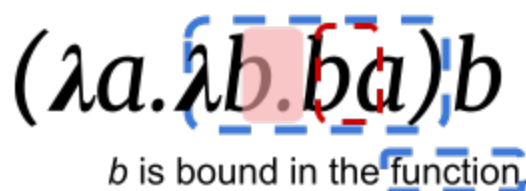
$(\lambda a. \lambda b. ba) b$

body of the function



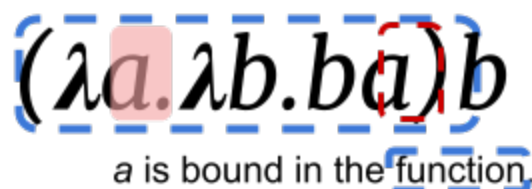
$(\lambda a. \lambda b. ba) b$

body of the function



$(\lambda a. \lambda b. ba) b$

b is bound in the function



$(\lambda a. \lambda b. ba) b$

a is bound in the function

Names are *bound* in nested expressions *but may be rebound*! For example,

$\lambda x. x \lambda x. x$

The "body" of the outer function is $x \lambda x. x$ and the leftmost x is the x from the outer function's parameter. In other words,

$(\lambda x. x \lambda x. x) (a) = a \lambda x. x$

The substitution of a for x continues no further because x is *rebound* at the start of the nested lambda function. You will be relieved to know that,

$\lambda x. x \lambda x. x = \lambda x. x \lambda a. a$

In fact, renaming like that has a special name: *alpha conversion*!

Free names are those that are not bound.

Wow, that got pretty complicated pretty quickly! This is one case where some formalism actually


improves the clarity of things, I think. Here is the formal definition of what it means for a name to be bound:

- $name$ is bound in $\lambda name_1. expression$ if $name = name_1$ or $name$ is bound in $expression$.
- $name$ is bound in $E_1 E_2$ if $name$ is bound in either E_1 or E_2 .

Here is the formal definition of what it means for a name to be free:

- $name$ is free in $name$
- $name$ is free in $\lambda name_1. expression$ when $name \neq name_1$ and $name$ is free in $expression$
- $name$ is free in $E_1 E_2$ if $name$ is free in either E_1 or E_2

Note that a name can be free and bound at the same time.

All this **[hullabaloo](https://en.wiktionary.org/wiki/hullabaloo#English)**  (<https://en.wiktionary.org/wiki/hullabaloo#English>) means that we need to be *slightly* more sophisticated in our function application (and, therefore, our "replacement" algorithm for reduction). We have to check two boxes before assuming that can treat function application as a simple textual search/replace:

When applying $\lambda x. E_1$ to E_2 , we only replace the *free* instances of x in E_1 with E_2 and if E_2 contains a free name that is bound in E_1 , we have to alpha convert the conflicting bound name in E_1 to avoid the collision. There is a good example of this in Section 1.2 of the **[Tutorial Introduction to the Lambda Calculus](https://uc.instructure.com/courses/1610326/files/167464105?wrap=1)** (<https://uc.instructure.com/courses/1610326/files/167464105?wrap=1>) that I will recreate here:

$$(\lambda x. (\lambda y. (x \lambda x. xy))) y$$

First, note y (our E_2 in this case) contains y (a free name) that is bound in $(\lambda y. (x \lambda x. xy))$ (our E_1). In other words, before doing a straight substitution, we have to alpha convert the bound y in E_1 to something that doesn't conflict. Let's choose t :

$$(\lambda x. (\lambda t. (x \lambda x. xt))) y$$

Now we can do our substitution! But, be careful: x appears free in $(\lambda y. (x \lambda x. xy))$ (again, our E_1) only one time -- its leftmost appearance! So, the substitution would yield:

$$(\lambda t. (y \lambda x. xt))$$

Voila!

Currying Functions

Currying is the process of turning a function that takes multiple parameters into a sequence of functions that

1. take a single parameter, and

2. return another function (that may, itself, take a single parameter and return a function ... and so on).

Currying is only possible in languages that support *high-order functions*: functions that a) take functions as parameters, b) return functions or c) both. Python is such a language. Let's look at how you would write a function that calculates the sum of three numbers in Python:

```
def sum3(a, b, c):  
    return a + b + c
```

That makes perfect sense!

Let's see if we can *Curry* that function. Because a Curried function can only take one parameter and we are Currying a function with three parameters, it stands to reason that we are going to have to generate three different functions. Let's start with the first:

```
def sum1(a):  
    # Something?
```

What *something* are we going to do? Well, we are going to declare another function inside `sum1`, call it `sum2`, that takes a parameter and then uses that as the return value of `sum1`! It looks something like this:

```
def sum1(a):  
    def sum2(b):  
        pass  
    return sum2
```

That means, if we call `sum1` with a single parameter, the result is *another function*, one that takes a single parameter! So, we've knocked off two of the three parameters, now we need one more. So, let's write something like this:

```
def sum1(a):  
    def sum2(b):  
        def sum3(c):  
            pass  
        return sum3  
    return sum2
```

This means that if we call `sum1` with a single parameter and call the result of *that* with a single parameter, the result is *another function*, one that also takes a single parameter! What do we want that innermost function to do? That's right: the summation! So, here's our final code:

```
def sum1(a):  
    def sum2(b):  
        def sum3(c):  
            return a + b + c  
        return sum3  
    return sum2
```

We've successfully Curried a three-parameter summation function! There's just one issue left to address? How can we possibly use `a` and `b` in the innermost function? Will, I thought you told us that Python was statically scoped! In order for this to work correctly, wouldn't Python have to have something magical and dynamic-scope-like? Well, yes! And, it does. It has *closures*.

When you return `sum2` from the body of `sum1`, Python *closes around* the variables that are needed by any code in the implementation of the returned function. Because `a` is needed in the implementation of `sum2` (the function returned by `sum1`), Python creates a *closure* around that function which includes the value of `a` at the time `sum2` was returned. It is important to remember that every time `sum2` is defined pursuant to an invocation of `sum1`, a *new* version of `sum2` is returned with a *new* closure and the values in that closure are (by default) copies of the values in the variables at the time the closure was generated. This closure-creation process repeats when we return `sum3` pursuant to an invocation of `sum2` (which itself was generated as a result of an invocation of `sum1`)! Whew.

Because we Curried the `sum3` function as `sum1`, we have to call them slightly differently:

```
sum3(1, 2, 3)
sum1(1)(2)(3)
```

As we learn more about functional programming in Haskell, you will see this pattern more and more and it will become second nature.

The "good" news, if you can call it that, is that functions in the Lambda Calculus always exist in their Curried form. Prove it to yourself by looking back at the way we formally defined the Lambda Calculus.

But, because it is laborious to write all those λ s over and over, we will introduce a shorthand for functions in the Lambda Calculus that take more than one parameter:

$\lambda p_1 p_2 \dots p_n. expression$

is a function with n parameters named p_1 through p_n (which are each one letter). Simply put,

$\lambda x. \lambda y. xy = \lambda xy. xy$

for example.

No One Puts Locals In a Corner

One of the really cool things about functional programming languages is their first-class support for functions. In functional programming languages, the programmer can pass functions as parameters to other functions and return functions from functions. We've worked with the former

already. Now let's look at the latter -- functions that "generate" other functions. JavaScript has the capability to return functions from functions so we'll use that language to explore:

```
function urlGenerator(prefix) {  
  function return_function(url) {  
    return prefix + "://" + url;  
  }  
  return return_function;  
}
```

The `urlGenerator` function takes a single parameter -- `prefix`. The caller of `urlGenerator` passes the protocol prefix as the argument (probably either "http" or "https"). The return value of `urlGenerator` is *itself a function* that takes a single parameter, a `url`, and returns `url` prepended with the `prefix` specified by the call to `urlGenerator`. An example might help:

```
const httpsUrlGenerator = urlGenerator("https");  
const httpUrlGenerator = urlGenerator("http");  
  
console.log(httpsUrlGenerator("google.com"));  
console.log(httpUrlGenerator("google.com"));
```

generates

```
"https://google.com"  
"http://google.com"
```

In other words, the definition of `httpsUrlGenerator` is (conceptually)

```
function httpsUrlGenerator(url) {  
  return "https" + "://" + url;  
}
```

and the definition of `httpUrlGenerator` is (conceptually)

```
function httpUrlGenerator(url) {  
  return "http" + "://" + url;  
}
```

But that's only a *conceptual* definition! The real definition continues to contain `prefix`:

```
return prefix + "://" + url;
```

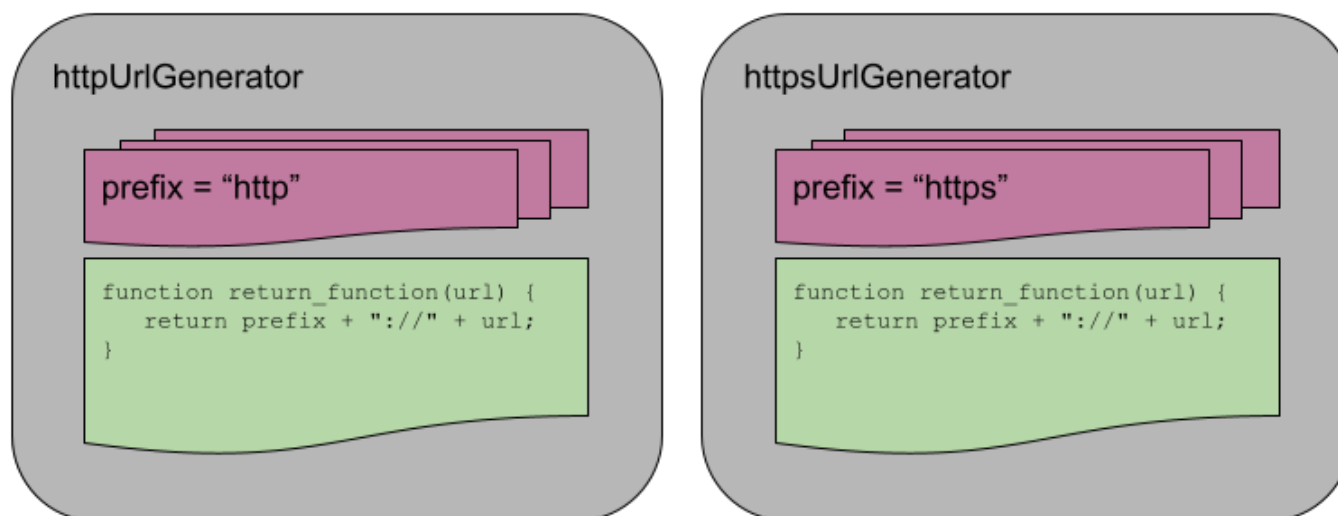
But, `prefix` is locally scoped to the `urlGenerator` function. So, how can `httpUrlGenerator` and `httpsUrlGenerator` continue to use its value after leaving the scope of `urlGenerator`?

The Walls Are Closing In

JavaScript, and other languages like it, have the concept of closures. A *closure* is a context that travels with a function returned by another function. Inside the closure are values for the *free*

variables (remember that definition?) of the function generated by the higher-order function that creates it. In `urlGenerator`, the returned function (`return_function`) uses the free variable `prefix`. Therefore, the closure associated with `return_function` contains a variable named `prefix` whose value initially begins as the value of `prefix` *at the time the closure is created*!

In the example above, there are two different copies of `return_function` generated -- one when `urlGenerator` is called with the argument "http" and one when `urlGenerator` is called with the parameter "https". Visually, the situation is



Doing Something with Lambda Calculus

Remember how we have stressed that you cannot name functions *inside* the Lambda Calculus but how I have stressed that does not mean we cannot assign meaning to function from *outside* the Lambda Calculus? Well, here's where it starts to pay off! We are going learn how to do Boolean operations using the Lambda Calculus. Let's assume that anytime we see a λ function that takes two parameters and reduces to the first, we call that T . When we see a λ function that takes two parameters and reduces to the second, we call that F :

$$T \equiv \lambda xy. x$$

$$F \equiv \lambda xy. y$$

To reiterate, it is the *form* that matters. If we see

$$\lambda ab. a$$

that is T too! In what follows, I will type T and F to save myself from writing all those λ s, but remember: T and F are just functions and we are comparing them based on their structure (recall our definition of *alpha reduction* from above)!!

Okay, let's do something with Boolean operations. We can define the *and* operation as

$$\wedge = \lambda xy. xyF$$

Let's give it a whirl. First, let's get on the same page: True and False is False.

$$\wedge T F = (\lambda xy. xyF) T F = TFF = (\lambda xy. x) FF = F$$

Awesome! Let's try another: True and True is True.

$$\wedge T T = (\lambda xy. xyF) T T = TTF = (\lambda xy. x) TF = T$$

We can define the *or* operation as

$$\vee = \lambda xy. xTy$$

Try your hand at working through a few examples and make sure you get the expected results!

Function Invocation in Functional Programming Languages

In imperative programming languages, it may matter to the correctness of a program the order in which parameters to a function are evaluated. (Note: For the purposes of this discussion we will assume that all operators $[+, -, /]$, etc] are implemented as functions that take the operands as arguments in order to simplify the discussion. In other words, when describing the order of evaluation of a function's arguments we are also talking about the order of evaluation of the operands to a binary (or unary) operator.) While the choice of the order in which we evaluate the operands is the language designer's prerogative, the choice has consequences. Why? Because of side effects! For example:

```
#include <stdio.h>

int operation(int parameter) {
    static int divisor = 1;
    return parameter / (divisor++);
}

int main() {
    int result = operation(5) + operation(2);
    printf("result: %d\n", result);
    return 0;
}
```

prints

```
result: 6
```

whereas

```
#include <stdio.h>

int operation(int parameter) {
    static int divisor = 1;
    return parameter / (divisor++);
}
```

```

}

int main() {
    int result = operation(2) + operation(5);
    printf("result: %d\n", result);
    return 0;
}

```

prints

```
result: 4
```

In the difference between the two programs we see vividly the role that the `static` variable plays in the state of the program and its ultimate output.

Because of the referential transparency in pure functional programming languages, the designer of such a language does not need to worry about the consequences of the decision about the order of evaluation of arguments to functions. However, that does not mean that the language designer of a pure functional programming language does not have choices to make in this area.

A very important choice the designer has to make is the *time* when function arguments are evaluated. There are two options available:

1. All function arguments are evaluated *before* the function is invoked.
2. Function arguments are evaluated *only when their results are needed by the function*.

Let's look at an example: Assume that there are two functions: `dbl`, a function that doubles its input, and `average`, a function that averages its three parameters:

```

dbl x = (+) x x
average a b c = (/) ((+) a ((+) b c)) 3

```

Both functions are written using prefix notation (i.e., $\langle operator \rangle \langle operand_1 \rangle \dots \langle operand_n \rangle$). We will call these functions like this:

```
dbl (average 3 4 5)
```

If the language designer chooses to evaluate function arguments *only when their results are needed*, the execution of this function call proceeds as follows:

```

dbl (average 3 4 5)
+ (average 3 4 5) (average 3 4 5)
+ ((/) ((+) 3 ((+) 4 5)) 3) (average 3 4 5)
+ (4) (average 3 4 5)
+ (4) ((/) ((+) 3 ((+) 4 5)) 3)
+ (4) (4)
8

```

The outermost function is always *reduced* (expanded) before the inner functions. Note: Primitive functions (`+`, and `/` in this example) cannot be expanded further so we move inward in

evaluation if we encounter such a function for reduction.

If, however, the language designer chooses to evaluate function arguments before the function is evaluated, the execution of the function call proceeds as follows:

```
dbl (average 3 4 5)
dbl ((/) ((+) 3 ((+) 4 5)) 3)
dbl 4
+ 4 4
8
```

No matter the designer's choice, the outcome of the evaluation is the same. However, there is something strikingly different about the two. Notice that in the first derivation, the calculation of the average of the three numbers happens twice. In the second derivation, it happens only once! That efficiency is not a fluke! Generally speaking, the method of function invocation where arguments are evaluated *before* the function runs faster.

These two techniques have technical names:


1. *applicative order*: "all the arguments to ... procedures are evaluated when the procedure is applied."
2. *normal order*: "delay evaluation of procedure arguments until the actual argument values are needed."

These definitions come from

Abelson, H., Sussman, G. J., with Julie Sussman (1996). Structure and Interpretation of Computer Programs. Cambridge: MIT Press/McGraw-Hill. ISBN: 0-262-01153-0 
(<http://uclid.uc.edu/record=b2528617~S39>)

It is obvious, then, that any serious language designer would choose applicative order for their language. There's no redeeming value for the inefficiency of normal order.

The Implications of Applicative Order

Scheme is a **Lisp dialect**  ([https://en.wikipedia.org/wiki/Lisp_\(programming_language\)#Major_dialects](https://en.wikipedia.org/wiki/Lisp_(programming_language)#Major_dialects)). I told you that we weren't going to work much with Lisp, but I lied. Sort of. Scheme is an applicative-order language with the same list-is-everything syntax as all other Lisps. In Scheme, you would define an *if* function named **myif** like this:

```
(define (myif c t f) (cond (c t) (else f)))
```

c is a boolean and **myif** returns **t** when **c** is true and **f** when **c** is false. No surprises.

We can define a name **a** and set its value to 5:

```
(define a 5)
```

Now, let's call `myif`:

```
(myif (= a 0) 1 (/ 1 a))
```

If `a` is equal to 0, then the call returns 1. Perfect. If `a` is not zero, the call returns the reciprocal of `a`. Given the value of `a`, the result is `1/7`.

Let's define the name `b` and set its value to 0:

```
(define b 0)
```

Now, let's call `myif`:

```
(myif (= b 0) 1 (/ 1 b))
```

If `b` is equal to 0, then the call returns 1. If `b` is not zero, the call returns the reciprocal of `b`. Given the value of `b`, the result is 1:

```
/: division by zero
context...:
"/home/hawkinsw/code/uc/cs3003/scheme/applicative/applicative.rkt": [running body]
temp37_0
for-loop
run-module-instance!125
perform-require!78
```

That looks *exactly* like 1. What happened?

Remember we said that the language is applicative order. No matter what the value of `b`, both of the arguments are going to be evaluated *before* `myif` starts. Therefore, Scheme attempts to evaluate `1 / b` which is `1 / 0` which is `division by zero`.

Thanks to situations like this, the Scheme programming language is forced into defining special semantics for certain functions, like the built-in *if* expression. As a result, function invocation is not *orthogonal* in Scheme -- the general rules of function evaluation in Scheme must make an exception for applying functions like the built-in *if* expression. Remember that the orthogonality decreases as exceptions in a language's specification increase.

Sidebar: Solving the problem in Scheme

Feel free to skip this section if you are not interested in details of the Scheme programming language. That said, the concepts in this section are applicable to other languages.

In Scheme, you can specify that the evaluation of an expression be `delay`ed until it is `force`d.


```
(define d (delay (/ 1 7)))
```

defines `d` to be the *eventual* result of the evaluation of the division of 1 by 7. If we ask Scheme to print out `d`, we see

```
#<promise:d>
```

To bring the future tense into the present tense, we `force` a `delay`ed evaluation:

```
(force d)
```

If we ask Scheme to print the result of that expression, we see:

```
1/7
```

Exactly what we expect! With this newfound knowledge, we can rewrite the `myif` function:

```
(define (myif c t f) (cond (c (force t)) (else (force f))))
```

Now `myif` can accept `t`s and `f`s that are `delay`ed and we can use `myif` safely:

```
(define b 0)
(myif (= b 0) (delay 1) (delay (/ 1 b)))
```

and we see the reasonable result:

```
1
```

Kotlin, a modern language, has a concept similar to `delay` called **lazy** [↗\(https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/lazy.html\)](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/lazy.html). Ocaml, an object-oriented functional programming language, **contains the same concept** [↗\(https://ocaml.org/manual/coreexamples.html#s:lazy-expr\)](https://ocaml.org/manual/coreexamples.html#s:lazy-expr). Swift has some sense of **laziness** [↗\(https://docs.swift.org/swift-book/LanguageGuide/Properties.html\)](https://docs.swift.org/swift-book/LanguageGuide/Properties.html), too!

Well, We Are Back to Normal Order

I guess that we are stuck with the inefficiency inherent in the normal order function application. Going back to the `dbl / average` example, we will just have to live with invoking `average` twice.

Or will we?

Real-world functional programming languages that are normal order use an interesting optimization to avoid these recalculations! When an expression is passed around and it is unevaluated, Haskell and languages like it represent it as a **thunk** [↗\(https://wiki.haskell.org/Thunk\)](https://wiki.haskell.org/Thunk). The thunk data structure is generated in such a way that it can calculate the value of its

associated expression some time in the future when the value is needed. Additionally, the thunk then caches (or *memoizes*) the value so that future evaluations of the associated expression do not need to be repeated.

As a result, in the `dbl/average` example,

1. a thunk is created for `(average 3 4 5)`,
2. that thunk is passed to `dbl`, where it is duplicated during the reduction of `dbl`,
3. `(average 3 4 5)` is (eventually) calculated when the value of `(average 3 4 5)` is needed,
4. 4 is stored (cached, memoized) in the thunk, and
5. the cached/memoized value is retrieved from the thunk instead of doing a fresh evaluation the next time that the value of `(average 3 4 5)` is needed.

A thunk, then, is the mechanism that allows the programmer to have the efficiency of applicative order function invocation with the semantics of the normal order function invocation!

The Daily PL - 7/20/2023

The Tail That Wags the Dog

There are no loops in functional programming languages. We've learned that, instead, functional programming languages are characterised by the fact that they use recursion for control flow. As we discussed earlier in the class (much earlier, in fact), when running code on a von Neumann machine, iterative algorithms typically perform faster than recursive algorithms because of the way that the former leverages the properties of the hardware (e.g., **spatial and temporal locality of code** \Rightarrow https://en.wikipedia.org/wiki/Locality_of_reference). In addition, recursive algorithms typically use much more memory than iterative algorithms.

Why? To answer that question, we will need to recall what we learned about stack frames. For every function invocation, an activation record (aka stack frame) is allocated on the run-time stack (aka call stack). The function's parameters, local variables, and return value are all stored in its activation record. The activation record for a function invocation remains on the stack until it has completed execution. In other words, if a function f invokes a function g , then function f 's activation record remains on the stack until (at least) g has completed execution.

Consider some algorithm A . Let's assume that an iterative implementation of A requires n executions of a loop and l local integer variables. If that implementation is contained in a function, an invocation of that function would consume approximately $l * \text{sizeof}(\text{integer variable}) + \text{activation record overhead}$ space on the runtime stack. Let's further assume that a function implementing the recursive version of A also uses l local integer variables and also requires n executions of itself. Each invocation of the implementing function, then, needs $l * \text{sizeof}(\text{integer variable}) + \text{activation record overhead}$ space on the runtime stack (because that's how much space is required for the stack frame). Multiply that by n , the number of recursive calls, and you can see that, at it's peak, executing the recursive version of algorithm A requires $n * (l * \text{sizeof}(\text{integer variable}) + \text{activation record overhead})$ space on the run-time stack. In other words, the recursive version requires n times more memory!

That escalated quickly \Rightarrow <https://www.youtube.com/watch?v=rFeVfwDvTyM>



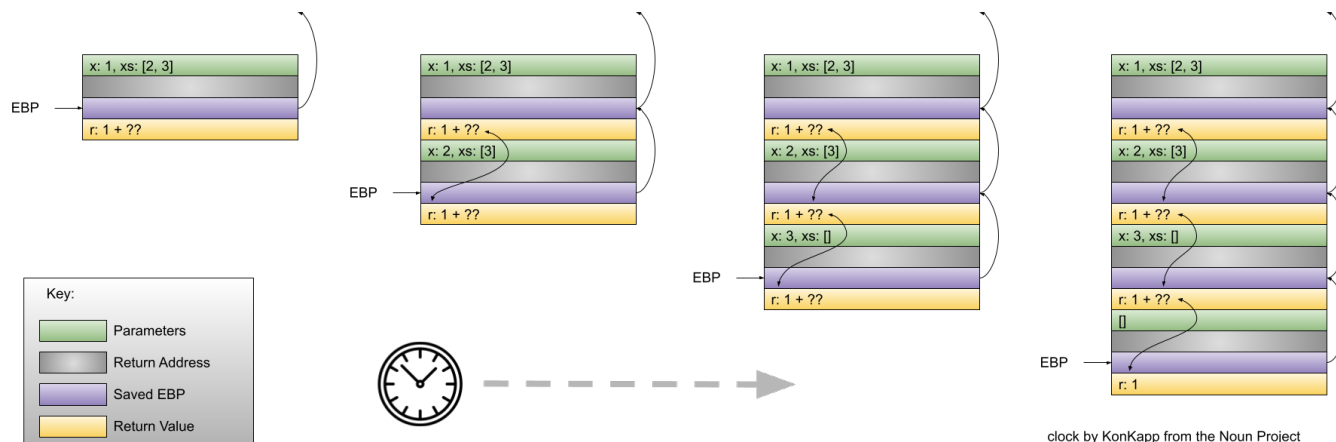
<https://www.youtube.com/watch?v=rFeVfwDvTyM>

That's all very abstract. Let's look at the implications with a real-world example, which we will write

in Haskell syntax:

```
myLen [] = 0
myLen (x:xs) = 1 + (myLen xs)
```

`myLen` is a function that recursively calculates the length of a list. Let's see what the stack would look like when we call `myLen [1,2,3]`:



When the recursive implementation of `myLen` reaches the base case, there are four activation records on the run-time stack.

Allocating space on the stack takes time. Therefore, the more activation records placed on the stack, the more time the program will take to execute. But, if we are willing to live with a slower program, then there's nothing else to worry about.

Right?

Wrong. Modern hardware is fast. Modern computers have lots of memory. Unfortunately, they don't have an infinite amount of memory. A program only has a finite amount of stack space. Given a long enough list, `myLen` could cause so many activation records to be placed on the stack that the amount of stack space is exhausted and the program crashes. In other words, it's not just that a recursive algorithm might execute slower, a recursive algorithm might fail to calculate the correct result entirely!

Tail Recursion - Hope

The activation records of functions that recursively call themselves remain on the stack because, presumably, they need the result of the recursive invocation to complete their work. For example, in our `myLen` function, an invocation of `myLen` cannot completely calculate the length of the list given as a parameter until the recursive call completes.

What if there was some way to rewrite a recursive function in a way that it did not need to wait on a future recursive invocation to completely calculate its result? If that could happen, then the stack

frame of the current invocation of the function could be *replaced* by the stack frame of the recursive invocation. Why? Because the information contained in the current invocation of the function has no bearing on its overall result -- the only information needed to completely calculate the result of the function is the result of the future recursive invocation! The implementation of a recursive function that matches this specification is known as a *tail-recursive function*. The book says "A function is tail recursive if its recursive call is the last operation in the function."

With a tail-recursive function, we get the expressiveness of a recursive definition of the algorithm along with the efficiency of an iterative solution! Ron Popeil, that's a deal!

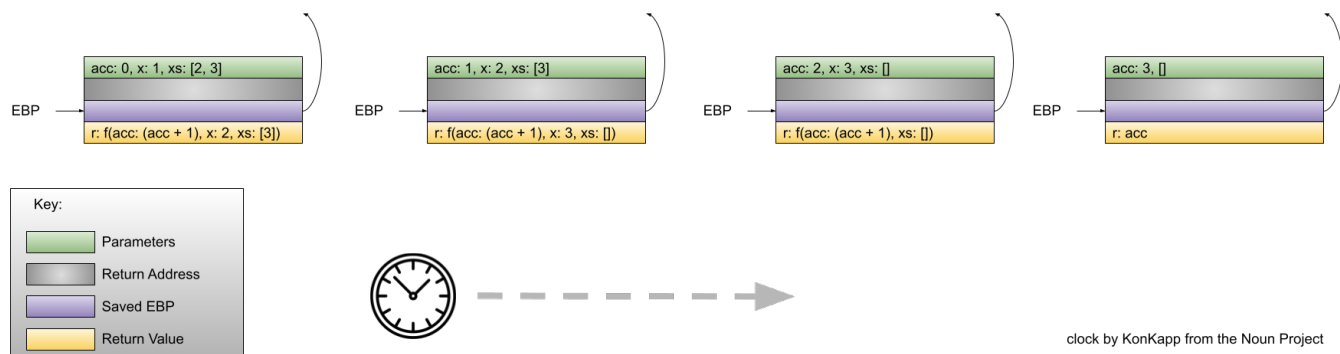
Rewriting

The rub is that we need to figure out a way to rewrite those non-tail recursive functions into tail-recursive versions. I am not aware of any general purpose algorithms for such a conversion. However, there is one technique that is widely applicable: accumulators. It is sometimes possible to add a parameter to a non-tail recursive function and use that parameter to define a tail-recursive version. Seeing an accumulator in action is the easiest way to *define* the technique. Let's rewrite `myLen` in a tail-recursive manner using an accumulator:

```
myLen list = myLenImpl 0 list
myLenImpl acc [] = acc
myLenImpl acc (x:xs) = myLenImpl (1 + acc) xs
```

First, notice how we are turning `myLen` into a function that simply invokes a *worker* (a.k.a. *helper*) function whose job is to do the actual calculation! The first parameter to `myLenImpl` is used to hold a running tally of the length of the list so far. The first invocation of `myLenImpl` from the implementation of `myLen`, then, passes 0 as the argument to the accumulator because the running tally of the length so far is, well, 0. The implementation of `myLenImpl` adds 1 to that accumulator variable for every list item that is stripped from the front of the list. The result is that the result of an invocation of `myLenImpl` does not rely on the completion of a recursive execution. Therefore, `myLenImpl` qualifies as a tail-recursive function! Woah.

Let's look at the difference in the contents of the run-time stack when we use the tail-recursive version of `myLen` to calculate the length of the list `[1,2,3]`:



A constant amount of stack space is being used -- amazing!

Your Total Is ...

Let's go from a function to sum numbers in a list to the concept of a *fold*. We'll start by writing the simple, recursive definition of a sum function in Haskell (using pattern matching):

```
simpleSum [] = 0
simpleSum (first:rest) = first + (simpleSum rest)
```

When invoked with the list `[1,2,3,4]`, the result is `10`:

```
*Summit> simpleSum [1,2,3,4]
10
```

Exactly what we expected. Let's think about our job security: The boss tells us that they want a function that "products" all the elements in the list. Okay, that's easy:

```
simpleProduct [] = 1
simpleProduct (first:rest) = first * (simpleProduct rest)
```

When invoked with the list `[1,2,3,4]`, the result is `24`:

```
*Summit> simpleProduct [1,2,3,4]
24
```

Notice that there are only minor differences between the two functions: the value returned in the base case (0 or 1) and the operation being performed on the first element of the list passed as an argument and the result of the recursive invocation.

I hear some of your shouting at me already: This isn't tail recursive; you told us that tail recursive functions are important. Fine! Let's rewrite the two functions so that they are tail recursive. We will do so using an accumulator and a helper function:

```
trSimpleSum list = trSimpleSumImpl 0 list
trSimpleSumImpl runningTotal [] = runningTotal
trSimpleSumImpl runningTotal (x:xs) = trSimpleSumImpl (runningTotal + x) xs
```

When invoked with the list `[1,2,3,4]`, the result is `10`:

```
*Summit> trSimpleSum [1,2,3,4]
10
```

And, we'll do the same for the function that calculates the product of all the elements in the list:

```
trSimpleProduct list = trSimpleProductImpl 1 list
trSimpleProductImpl runningTotal [] = runningTotal
trSimpleProductImpl runningTotal (x:xs) = trSimpleProductImpl (runningTotal * x) xs
```

When invoked with the list `[1,2,3,4]`, the result is `24`:

```
*Summit> trSimpleProduct [1,2,3,4]
24
```

One of These Things is Just Like The Other

Notice the similarities between `trSimpleSumImpl` and `trSimpleProductImpl`. Besides the names, the only difference, really, is the operation that is performed on the `runningTotal` and the head element of the list. Because we're using a functional programming language, what if we wanted to let the user specify that operation in terms of a function parameter? Such a function would need to accept two arguments (the up-to-date running total and the head element) and return a new running total. For summing, we might write a `sumOperation` function:

```
sumOperation runningTotal headElement = runningTotal + headElement
```

Next, instead of defining `trSimpleSumImpl` and `trSimpleProductImpl` with fixed definitions of their operation, let's define a `trSimpleOpImpl` that could use `sumOperation`:

```
trSimpleOpImpl runningTotal operation [] = runningTotal
trSimpleOpImpl runningTotal operation (x:xs) = trSimpleOpImpl (operation runningTotal x) operation xs
```

Fancy! Now, let's use `trSimpleOpImpl` and `sumOperation` to recreate `trSimpleSum` from above:

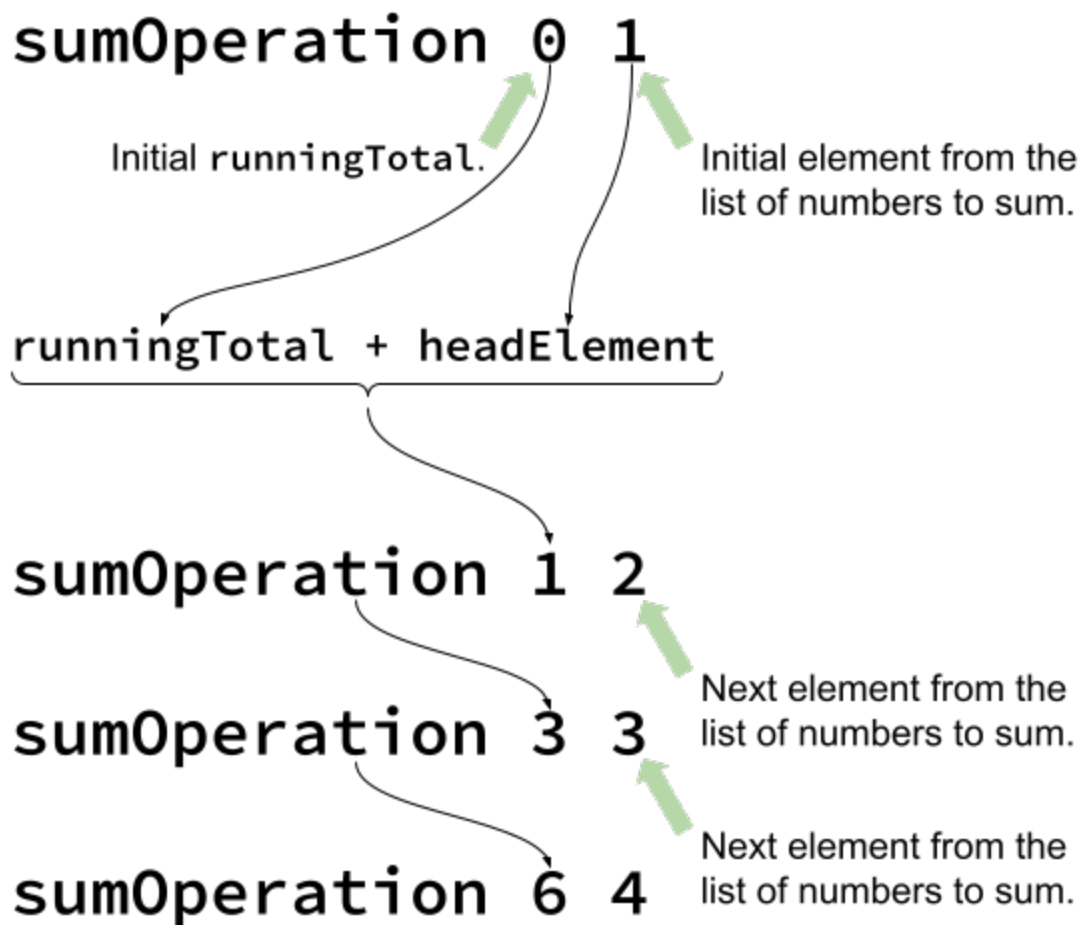
```
trSimpleSum list = trSimpleOpImpl 0 sumOperation list
```

Let's check to make sure that we get the same results: When invoked with the list `[1,2,3,4]`, the result is `10`:

```
*Summit> trSimpleSum [1,2,3,4]
10
```

To confirm our understanding of what's going on here, let's visualize the invocations of `sumOperation` necessary to complete the calculation of `trSimpleSum`:

```
sumOperation 0 1
sumOperation 1 2
sumOperation 3 3
sumOperation 6 4
```



Let's do a similar thing for `trSimpleProduct`:

```
productOperation runningTotal headElement = runningTotal * headElement
trSimpleProduct list = trSimpleOpImpl 0 productOperation list
```

Let's check to make sure that we get the same results: When invoked with the list `[1,2,3,4]`, the result is `24`:

```
*Summit> trSimpleProduct [1,2,3,4]
24
```

Think About the Types:

We've stumbled on a pretty useful pattern! Let's look at its components:

1. A "driver" function (called `trSimpleOpImpl`) that takes three parameters: an initial value (of a particular type, T), an operation function (see below) and a list of inputs, each of which is of type T .
2. An operation function that takes two parameters -- a running total, of some type R ; an element

- to "accumulate" on to the running total (of type T) -- and returns a new running total of type R .
3. A list of inputs, each of which is of type T .

Here are the types in Haskell:

operation function: `R -> T -> R`

list of inputs: `[T]`

driver function: `T -> (R -> T -> R) -> R`

Let's play around and see what we can write using this pattern. How about a concatenation of a list of strings in to a single string?

```
concatenateOperation concatenatedString newString = concatenatedString ++ newString
concatenateStrings list = trSimpleOpImpl "" concatenateOperation list
```

(the `++` just concatenates two strings together). When we run this on `["Hello", ",", "World"]` the result is `"Hello, World"`:

```
*Summit> concatenateStrings ["Hello", ",", "World"]
"Hello,World"
```

So far our T s and R s have been the same -- integers and strings. But, the signatures indicate that they could be different types! Let's take advantage of that! Let's use `trSimpleOpImpl` to write a function that returns `True` if every element in the list is equal to the number 1 and `False` otherwise. Let's call the operation function `continuesToBeAllOnes` and define it like this:

```
continuesToBeAllOnes equalToOneSoFar maybeOne = equalToOneSoFar && (maybeOne == 1)
```

This function will return `True` if the list (to this point) has contained all ones (`equalToOneSoFar`) and the current element (`maybeOne`) is equal to one. In this case, the R is a boolean and the T is an integer. Let's implement a function named `isListAllOnes` using `continuesToBeAllOnes` and `trSimpleOpImpl`:

```
isListAllOnes list = trSimpleOpImpl True continuesToBeAllOnes list
```

Does it work? When invoked with the list `[1,1,1,1]`, the result is `True`:

```
*Summit> isListAllOnes [1,1,1,1]
True
```

When invoked with the list `[1,2,1,1]`, the result is `False`:

```
*Summit> isListAllOnes [1,2,1,1]
False
```

Naming those "operation" functions every time is getting annoying, don't you think? I bet that we could be lazier!! Let's rewrite `isListAllOnes` without specifically defining the `continuesToBeAllOnes` function:

```
isListAllOnes list = trSimpleOpImpl True (\equalToOneSoFar maybeOne -> equalToOneSoFar && (maybeOne == 1)) list
```

Now we are really getting functional!

I am greedy. I want to write a function that returns `True` if *any* element of the list is a one:

```
isAnyElementOne list = trSimpleOpImpl False (\anyOneSoFar maybeOne -> anyOneSoFar || (maybeOne == 1)) list
```

This is just way too much fun!

Fold the Laundry

This type of function is so much fun and, more importantly, so useful that it is included in the standard implementation of Haskell! It's called fold! And, in true Haskell fashion, there are two different versions to maximize flexibility *and* confusion: the fold-left and fold-right operation. The signatures for the functions are the same in both cases:

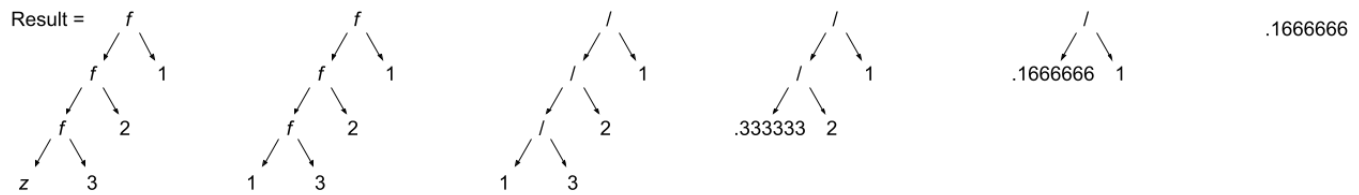
`fold[l,r] :: operation function -> initial value -> list -> result`

In all fairness, these two versions are necessary. Why? Because certain operations are not associative! It doesn't matter the order in which you add or multiply a series of numbers -- the result of $(5 * (4 * (3 * 2)))$ is the same as $((5 * 4) * 3) * 2$. The problem is, that's not the case of an operation like division!

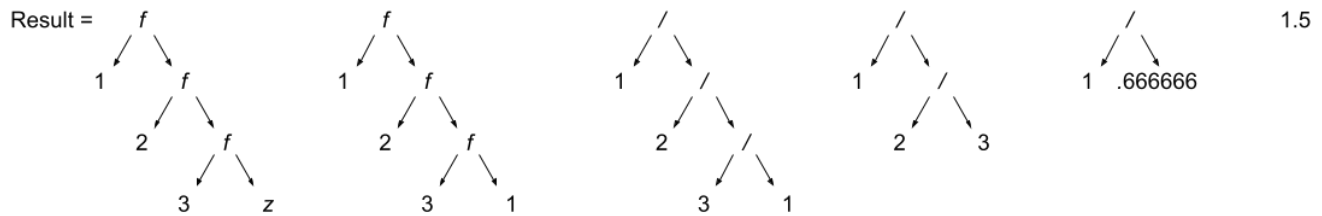
A fold-left operation (`foldl`) works by starting the operation (essentially) from the first element of the list and the fold-right operation (`foldr`) works by starting the operation (essentially) from the last element of the list. Furthermore, the choice of `foldl` vs `foldr` affects the order of the parameters to the operation function: in a `foldl`, the running value (which is known as the *accumulator* in mainstream documentation for fold functions) is the left parameter; in a `foldr`, the accumulator is the right parameter.

This will make more sense visually, I swear:

```
*Summit> foldl (\x y -> x / y ) 1 [3,2,1]
0.16666666666666666
```



```
*Summit> foldr (\x y -> x / y) 1 [3,2,1]
1.5
```



Let's use our newfound fold power, to recreate our work from above:

```
isAnyElementOne list = foldl1 (\anyOneSoFar maybeOne -> anyOneSoFar || (maybeOne == 1)) False list
isListAllOnes list = foldl1 (\equalToOneSoFar maybeOne -> equalToOneSoFar && (maybeOne == 1)) True list
concatenateStrings list = foldl1 (\concatenatedString newString -> concatenatedString ++ newString) ""
list
```