# 1 The Lambda Calculus

## 1.1 Fixpoint Combinator

We want a combinator `FIX` such that `FIX STEP` calls `STEP` with itself as the first argument; that is,

```
FIX STEP
=*> STEP (FIX STEP)
```

### 1.1.1 The Y Combinator

Recall that we had

```
(\x -> x x) (\x -> x x)
=b> (\x -> x x) (\x -> x x)
```

Note that this is *self-replicating* code. Now, in order to do recursion, we need something like this, but a bit more involved.

This beings us to the *Y Combinator*. Discovered by Haskell Curry, we can define `FIX` like so:

```
let FIX = \stp -> (\x -> stp (x x)) (\x -> stp (x x))
```

Here, `stp` is the *step*. To see how this works, consider the following reductions:

```
eval fix_step:
    FIX STEP
    =d> (\stp -> (\x -> stp (x x)) (\x -> stp (x x))) STEP
    =b> (\x -> STEP (x x)) (\x -> STEP (x x))
    =b> STEP (\x -> STEP (x x)) (\x -> STEP (x x))              -- The fix step
```

With this beta reduction, it duplicated itself but also prepended itself. This gives us the ability to do another beta reduction, thus repeating everything again.

# 2 Haskell

A typed, lazy, *purely functional* programming language. Essentially, Haskell is just $\lambda$-calculus with:

- Better syntax.
- Types.
- Built-in features like:
  - Booleans, numbers, and characters.
  - Records (tuples).
  - Lists.
  - Recursion.
  - And much more.

Haskell programs tend to be concise and correct, which is why we're primarily interested in learning it.

## 2.1 Similarities: Haskell vs. Lambda Calculus

We will begin by looking at some similarities.

### 2.1.1 Programs

A program is an **expression**, not a sequence of statements. It **evaluates** to a value (it does *not* perform actions).

- In Lambda Calculus, a program might look like:

```
(\x -> x) apple          -- apple
```

- In Haskell, a program might look like:

```
(\x -> x) "apple"        -- "apple"
```

  Note that we used the *string* `apple` because, like many other programming languages, using an undefined variable would result in errors.

### 2.1.2 Functions

Like Lambda Calculus, functions are *first-class* values. They can be

- passed as arguments to other functions.
- returned as results from other functions.
- partially applied (arguments passed one at a time).

But, unlike Lambda Calculus, not everything is a function. Consider the following program:

```
(\f x -> f (f x)) (\z -> z + 1) 0
```

First, we note that we have the built-in `0` and `1`! Second, notice that we're incrementing in terms of the built-in integers, not in terms of Churchill numerials. The above program evalutes to:

```
(\f x -> f (f x)) (\z -> z + 1) 0
=> (\x -> (\z -> z + 1) ((\z -> z + 1) x)) 0
=> (\z -> z + 1) ((\z -> z + 1) 0)
=> (\z -> z + 1) 1
=> 2
```

### 2.1.3 Top-Level Bindings

Like in *Elsa*, we can name terms to use them later. For example, we have:

```
let T       = \x y -> x
let F       = \x y -> y
let PAIR    = \x y -> \b -> ITE b x y
let FST     = \p -> p T
let SND     = \p -> p F

eval fst:
    FST (PAIR apple orange)
    =~> apple
```

In Haskell, we have:

```
haskellIsAwesome = True

pair = \x y -> \b -> if b then x else y
fst = \p -> p haskellIsAwesome
snd = \p -> p False

-- In GHCi (the Haskell interpreter/compiler)
> fst (pair "apple" "orange")               -- "apple"
```

Note that the **names** are called **top-level variables**. Their **definitions** are called **top-level bindings**.

## 2.2 Differences: Haskell vs. Lambda Calculus

We will begin by looking at some differences.

### 2.2.1 Better Syntax: Equations and Patterns

We can define function bindings using **equations**. For example,

```
pair x y b = if b then x else y             -- Same as: pair = \x y b -> ...
fst p      = p True                         -- Same as: fst = \p -> ...
snd p      = p False                        -- Same as: snd = \p -> ...
```

The parameters are put on the left of the equal sign, thus omitting the lambda part. One way to think about this is that if you see anything on the left-hand side of the equal sign, you can replace it with everything on the right-hand side.

A single function binding can have multiple equations with different **patterns** of parameters. For example (**A**),

```
pair x y True  = x      -- If the 3rd argument evaluates to True,
                        -- then use this equation.

pair x y False = y      -- If the 3rd argument evaluates to False,
                        -- then use this equation.
```

At runtime, the first equation whose pattern matches the actual arguments is chosen. For now, we can define a **pattern** to be:

- A *variable* (matches any value)
- A *value* (matches only that value)

The above equations from (A) can be written like so (**B**):

```
pair x y True  = x      -- If the 3rd argument evaluates to True,
                        -- then use this equation.

pair x y b     = y      -- Otherwise, use this equation
```

To see how this works, let's consider the following expression (where `haskellIsAwesome` is `True`):

```
> pair "apple" "banana" (not haskellIsAwesome)
```

Informally, the evaluation of the above expression works like so:

- Since variables match anything, `"apple"` and `"banana"` match the first two variables in the above definitions (`pair x y`, where `x` is `"apple"` and `y` is `"banana"`).
- Note that `not haskellIsAwesome` is `False`. Since this doesn't match the first definition (as the first definition expects `True` as the last argument), then this will match the second definition (since its last argument, `b`, takes in anything that isn't `True`).

The reason why (B) is equivalent to (A) is because `b` can only be `False` (since the first definition expects `True` for the third parameter). The reason why this is the case is because Haskell is, indeed, a *typed* language (which we'll see later).

Now, we also have that (B) is the same as (**C**):

```
pair x y True  = x
pair x y _     = y      -- Wilcard pattern. '_' is like a variable,
                        -- but cannot be used on the right.
```

In other words, `_` just says that you don't actually need said value.

(Quiz.) Which of the following definitions of `pair` is **not the same** as the original?

```
pair = \x y -> \b -> if b then x else y
```

(a)
```
pair x y = \b -> if b then x else y.
```

(b)
```
pair x _ True = x
pair _ y _    = y
```

(c)
```
pair x _ True = x
```

(d)
```
pair x y b     = x
pair x y False = y
```

(e) C and D

The answer is **E**.

- C is incorrect because the `False` case is never covered. In other words, the pattern matching is non-exhaustive.
- D is incorrect because patterns are evaluted *in order*. Remember, "*the first equation whose pattern matches the actual arguments is chosen*". So, everything will be matched against the first case, not the second.

### 2.2.2 Equation with Guards

Guards are another way of giving *multiple definitions* for a function. We can use guards as opposed to using different patterns when your definition depends on something that cannot be expressed as a pattern.

Consider the following code:

```
cmpSquare x y   | x > y*y   = "bigger :)"
                | x == y*y  = "same :|"
                | x < y*y   = "smaller :("
```

This is the same thing (and is more preferred) as:

```
cmpSquare x y   | x > y*y   = "bigger :)"
                | x == y*y  = "same :|"
                | otherwise = "smaller :("
```

This case (the one with the `otherwise`[1] case) is preferred to the one above (the one without the `otherwise` case), as we have a "catch-all" case (i.e. a case that will be executed when the other cases do not suffice). Roughly speaking, this example looks like the cases in math; that is:

$$\text{cmpSquare}(x, y) = \begin{cases} \texttt{"bigger :)"} & x > y \cdot y \\ \texttt{"same :|"} & x = y \cdot y \\ \texttt{"smaller :("} & \text{otherwise} \end{cases}.$$

### 2.2.3 Recursion

Unlike in Lambda Calculus, recurison is built-in; there is no need for a fixpoint combinator. For example, the following program is valid Haskell:

```
sum = \n -> if n == 0
                then 0
                else n + sum(n - 1)
```

Which is the same thing as writing:

```
sum n = if n == 0
            then 0
            else n + sum(n - 1)
```

A more idiomatic way would be using equations and pattern matching[2], like so:

```
sum 0 = 0
sum n = n + sum(n - 1)
```

A slightly less idiomatic way would be using guards:

```
sum n | n == 0      = 0
      | otherwise   = n + sum(n - 1)
```

As a warning, Haskell is indentation-sensitive; that is, the following is *invalid* (your *then* and *else* should at least line up with the `if`):

```
sum n = if n == 0
    then 0
    else n + sum(n - 1)
```

[1] `otherwise` is a variable in the standard library that literally is just `True`.
[2] If you can use patterns, then use patterns.

### 2.2.4 Scope of Variables

There are two scope types.

- **Global:** Top-level variables have **global** scope, so you can write:

```
message = if haskellIsAwesome
            then "I love CSE 130"
            else "I'm dropping CSE 130"

-- Notice how 'haskellIsAwesome' is defined below where it's being used.
-- This is because 'haskellIsAwesome' has a global scope.
haskellIsAwesome = True
```

  Here, global scope means that it is available in the *whole module*.

- **Local:** Consider the following code:

```
sum 0 = 0
sum n = let n' = n - 1 in n + sum n'
```

  Here, the `let` keyword defines a new local variable. So, `let n'` defines a new variable `n'`, and `= n - 1` is the definition of `n'`. Now, this is known as a `let`-expression, and its scope is whatever is after the `in` keyword (in reality, see below). Here, anything after the `in` is the *body of the `let`-expression* and can use the local variable.

  Note that the following code segment is invalid:

```
sum 0 = 0
sum n = (let n' = n - 1 in n + sum n') + n'
```

  This is because the last `n'` doesn't have an associated definition. However, note that the following is *valid* because this is needed in case of a recursive definition.

```
sum 0 = 0
sum n = let n' = n' - 1 in n + sum n'
```

  So, actually, the **scope of a local variable** is its definition and its body.