Language Engineering (TB1)

Taken by Joseph MacManus
on a course given by Dr Nicolas Wu (2018)

May 12, 2019

Contents

1 DSLs and Catamorphisms

1.1 Domain-Specific Languages

A programming language consists of three main components:

- Syntax - the shape of grammer / words / vocab.
- Semantics - the meaning; a function from syntax to some domain.
- Pragmatics - The purpose of a language.

A domain-specific language (DSL) is a language that has been crafted with a specific purpose in mind. These are not necessarily Turing-complete. Some DSLs come equipped with all the features of general purpose languages:

- Parser
- Syntax highlighting
- IDE
- Compiler
- Documentation

and so on. An embedded DSL (EDSL) is defined within another host language. The advantage is that there is less work to perfom, but this is at the cost of restricted flexibility. An EDSL can be either a *deep* or *shallow* embedding. A deep embedding is where the syntax is given by concrete datatypes, and the semantics given by evaluation. A shallow embedding has syntax borrowed directly from it's host language, and semantics is directly given.

For example, consider:

$$\text{"}3+5\text{"} \text{ in a string.}$$

$$[\![3+5]\!] :: \text{Int}$$

Where $[\![]\!]$ are denotational brackets. We use them to ascribe a semantics.

$$[\![3+5]\!] = [\![3]\!] + [\![5]\!] = 3+5 = 8$$

We can model this using a deep embedding in Haskell with the following code:

```
data Expr = Var Int
          | Add Expr Expr

eval :: Expr → Int
eval (Var n) = n
eval (Add x y) = (eval x) + (eval y)
```

So instead of $[\![3+5]\!]$, we can now write `eval (Add (Var 3) (Var 5))`.

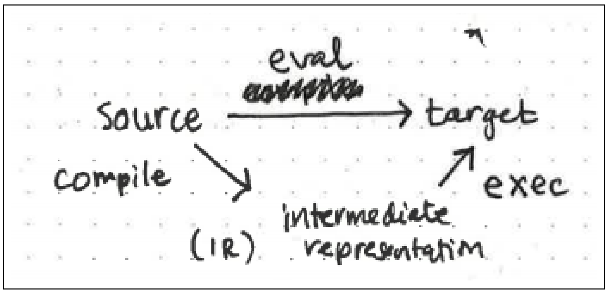
The shallow embedding is given directly by functions: (We will redefine `Expr` here)

```
type Expr = Int

var :: Int → Expr
var n = n

add :: Expr → Expr → Expr
add x y = x + y
```

Our example is now written as:

```
add (var 3) (var 5)
```

Remark. What is a compiler, then? A compiler is code that transforms a language to a target language through some intermediate representation.

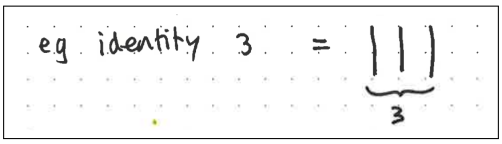


Typical examples of this diagram include the C compiler gcc, which takes a C source file and compiles this to assembly. That assembly is then executed in the terminal. Javascript tends to ignore the compile stage since it is an interpreted language. The web browser performs eval, which turns JS into rendered output. Haskell has two modes, when using GHC, it compiles .hs files into assembly, which can then be executed in the terminal. However, when using GHCi, it takes source code and interprets it directly by evaluating in the terminal.

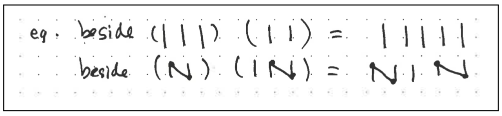
1.2 Case Study: Circuit Language

We will study a particular example of a DSL, and the different ways to embed it in Haskell. The circuit language is a DSL for describing circuits. It consists of several operations. Rather than define each operation formally, we shall give the type of each operation and an example of what it does.

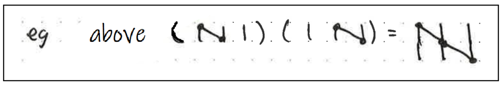
- `identity :: Int → Circuit`



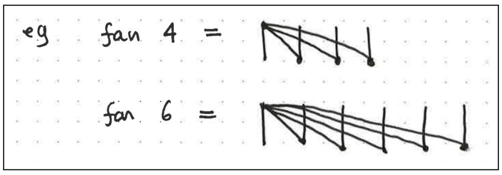
- beside :: Circuit → Circuit → Circuit



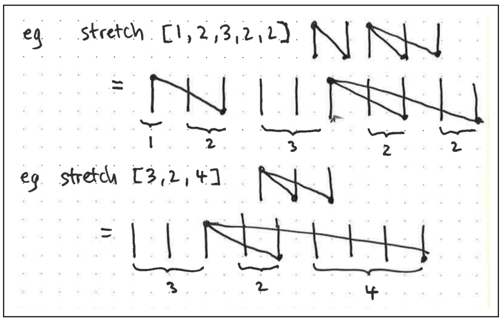
- above :: Circuit → Circuit → Circuit



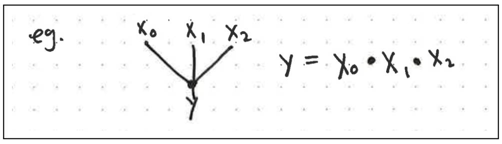
- fan :: Int → Circuit



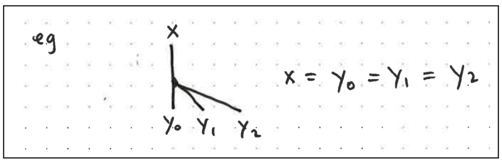
- stretch :: [Int] → Circuit → Circuit



This language is used to describe how circuits work, information starts at the top of each line and travels downwards. Information is combined by joining lines, and applying the associative operation of our choosing:



And information is duplicated when lines seperate:



There are many different ways of interpeting this circuit language. For example, we may want to simply find the width or height of a circuit, or perhaps we will evaluate the output of a circuit given an input and the node operation. We start with a deep embedding. This is achieved in two steps:

1. Create a data structure for the abstract syntax.
2. Define a semantics with an evaluation function.

The first step isn't too hard, we simply get:

```
data Circuit = Identity Int
             | Beside Circuit Circuit
             | Above Circuit Circuit
             | Fan Int
             | Stretch [Int] Circuit
```

Let's interpet the circuit language by stipulating that the semantics of a term is the width of the circuit generated. We will define a semantics with a function `width`.

```
type Width = Int

width :: Circuit → Width
width (Identity n) = n
width (Beside c1 c2) = (width c1) + (width c2)
width (Above c1 c2) = width c1
width (Fan n) = n
width (Stretch ws c) = sum ws
```

We can have multiple semantics easily by supplying more evaluation functions. For instance, the height of the circuit is:

```
type Height = Int

height :: Circuit → Height
height (Identity n) = 1
...
height (Above c1 c2) = height c1 + height c2
```

Sometimes the semantics are interwined in a dependent way. For instance, calculating if a circuit is well connected requires us to calculate the width even though all we are interested in is one bool.

```
type Connected = Bool

connected :: Circuit → Connected
connected (Identity n) = True
connected (Beside c1 c2) = connected c1 && connected c2
connected (Above c1 c2) = connected c1 && connected c2
                          && width c2 == width c2
connected (Fan n) = True
connected (Stretch ws c) = connected c && width c == length ws
                          && (not ∘ (elem 0)) ws
```

In a shallow embedding we simply have to give a semantics in terms of the operations directly.

```
type Width = Int

type Circuit = Width

identity :: Int → Circuit
identity n = n

beside :: Circuit → Circuit → Circuit
beside c1 c2 = c1 + c2

above :: Circuit → Circuit → Circuit
above c1 c2 = c1

fan :: Int → Circuit
fan n = n

stretch :: [Int] → Circuit → Circuit
stretch ws c = sum ws
```

Shallow is problemaic because it is hard to add a different semantics. In order to do so we must redefine Circuit, but this might break any code that depends on the old definition. Additionally, a dependent semantics requires all of the interpretations to be considered at once. This is not compositioned. However, in a shallow semantics it is easy to extend the language with new operations, since this involves adding new functions: nothing breaks. With a deep embedding a new constructor must be added, so all semtantics must be extended accordingly.

Is it possible to extend the syntax and semtnaics of a language in a modular fashion? This is known as *The Expression Problem.* For instance, consider the data type `Expr` from before:

```
data Expr =val Int | Add Expr Expr
```

Here we want to extend the syntax by adding a new operation for multiplication, but we do not want to modify any existing code. I.e. we cannot simply add a new `Mul` constructor. Similarly, consider the semantics:

```
eval :: Expr → Int
```

Again, we want to extend the semantics without modifying the code. (Though in this case adding semantics is easy as we simply write another function of type `Expr → b`). To solve the expression problem, we will study a generalisation of folds called a *catamorphism.* We do this because folds are a way of reducing data structures in a composable way, and syntax trees are just data structures.

1.3 Catamorphisms

Consider the fold for a list:

```
data [a] = []
         | a : [a]

foldr :: b → (a → b → b) → [a] → b
foldr k f [] = k
foldr k f (x:xs) = f x (foldr k f xs)
```

How do we generalise this? Our first step is to deconstruct the type of lists to expose its generic structure. The defintion of lists is the same as the following, when we remove the syntactic sugar.

```
data List a = Empty
            | Cons a (List a)
```

We remove recursion from this data type, and mark it with a parameter `k`, for *k*ontinuation.

```
data List a k = Empty
              | Cons a k
```

Next, we make the recursive paramter something we can change programatically by giving a Functor instance to `List` a

```
instance Functor (List a) where
    fmap :: (x → y) → List a x → List a y
    fmap f Empty = Empty
    fmap (Cons a x) = Cons a (f x)
```

We now need to derive a type that gives us the fixed point of data. This is defined as follows:

```
data Fix f = In (f (Fix f))
```

This datatype allows us to generalise all recursive data types (except mutually recursive ones). For example, instead of `List a`, we can write `Fix (List a)`. To demonstrate this, we show that `List a` and `Fix (List a)` are isomorphic.

```
toList :: Fix (List a) → List a
fromList :: List a → Fix (List a)
```

We say that `List a` and `Fix (List a)` are isomorphic when composing the above functions in any order yields the identity. Let's define these functions.

```
fromList :: List a → Fix (List a)
fromList Empty = In Empty
```

Some examples of values of type `Fix (List a)` are:

```
In Empty :: Fix (List a)
```

(Note that the type of `In` is `In :: f (Fix f) → Fix f` so the example above is where `f` is `List a`.)

```
In (Cons 5 (In Empty))
```

and for two elements we have

```
In (Cons 6 (In (Cons 7 (In Empty))))
```

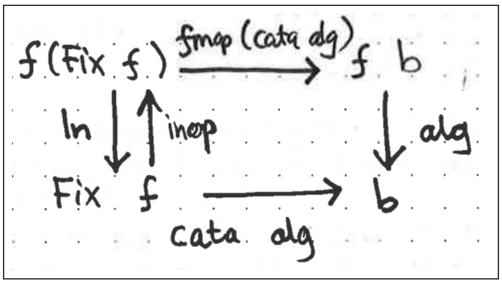
So now we have enough to finish a definition of `fromList`.

```
fromList :: List a → Fix (List a)
fromList Empty = In Empty
fromList (Cons x xs) = In (Cons x (fromList xs))
```

We are now ready to generise fold to be a catamorphism. Consider functions of the form `List a b → b`:

```
h :: List a b → b
h Empty = k
h (Cons a y) = f a y
    where
        k :: b
        f :: a → b → b
```

Functions of that type correspond to replacing the constructors of `List a` with functions `k` and `f` just like in `foldr`. A catamorphism arises from this diagram:



The function `inop` is the opposite of `In`. We define it by the following:

```
inop :: Fix f → f (Fix f)
inop (In x) = x
```

So finally we can write the function cata by chasing the arrows of this square:

```
cata :: Functor f ⇒ (f b → b) → Fix f → b
cata alg x = (alg ∘ fmap (cata alg) ∘ inop) x
```

An alternative and equivalent definition is:

```
cata alg (In x) = alg (fmap (cata alg) x)
```

To use this, we only need to supply the `alg`. We will define the function `toList` using a `cata`:

```
toList :: Fix (List a) → List a
toList = cata alg
    where
        alg :: List a (List a) → List a
        alg Empty = Empty
        alg Cons x xs = Cons x xs
```

or, equivalently

```
toList' :: Fix (List a) → [a]
toList' = cata alg
    where
        alg :: List a [a] → [a]
        alg Empty = []
        alg Cons x xs = x : xs
```

We can also define a function that returns the length of a `Fix (List a)`.

```
length :: Fix (List a) → Int
length :: cata alg
    where
        alg :: List a Int → Int
        alg Empty = 0
        alg (Cons x y) = y + 1
```

Here is an example of evaluation.

```
    length (In (Cons 7 (In (Cons 9 (In Empty)))))
  = {length}
    cata alg (In (Cons 7 (In (Cons 9 (In Empty)))))
  = {cata}
    alg (fmap (cata alg) (Cons 7 (In ...)))
  = {fmap}
    alg (Cons 7 (cata alg (In (Cons 9 (In Empty)))))
  = {alg}
    1 + cata alg (In (Cons 9 (In Empty)))
  = {cata}
    1 + alg (fmap (cata alg) (Cons 9 (In Empty)))
  = {fmap}
    1 + alg (Cons 9 (cata alg (In Empty)))
  = {alg}
    1 + 1 + cata alg (In Empty)
  = {cata}
    1 + 1 + alg (fmap (cata alg) Empty)
  = {fmap}
    1 + 1 + alg Empty
  = {alg}
    1 + 1 + 0
  = {(+)}
    2
```

Now another exmaple, of summing a list.

```
sum :: Fix (List Int) → Int
sum = cata alg
    where
        alg :: List Int Int → Int
        alg Empty = 0
        alg (Cons x y) = x + y
```

1.4 Case Study: Peano Numbers

A Peano number is either zero, or a successor of another Peano number.

```
data Peano = z
           | S Peano
```

So the number 3 is written $S(S(S(0)))$. Our goal is to create a modular DSL which models Peano numbers, a fashion similar to above. Step 1: make a signature functor for `Peano`.

```
data Peano k = z
             | S k
```

Step 2: Define a functor instance for `Peano`.

```
instance Functor Peano where
    fmap :: (a → b) → Peano a → Peano b
    fmap f z = z
    fmap f (S x) = S (f x)
```

Step 3: Write functions using `cata`.

```
toInt :: Fix Peano → Int
toInt = cata alg where
    alg :: Peano Int → Int
    alg z = 0
    alg (S x) = x + 1
```

Now we can define a doubling function.

```
double :: Fix Peano → Fix Peano
double = cata alg  where
    alg :: Peano (Fix Peano) → Fix Peano
    alg z = In z
    alg (S x) = In (S (In (S x)))
```

1.5 Composing Languages

Previously, we looked at the following data type as the language for addition.

```
data Expr = Val Int
          | Add Expr Expr
```

We then learnt to extract the signature functor for this by locating recursive calls:

```
data Expr k = Val Int
            | Add k k
```

The `Fix Expr` datatype is essentially `Expr`. Suppose we want to add multiplication to this language. We need a way to extend `Expr` with more constructors. This is achieved by the coproduct of functors. The coproduct functor is defined as:

```
data (f :+: g) a = L (f a)
                 | R (g a)
```

This takes two functors and makes the functor `(f:+:g)`. It introduces these constructors:

```
L :: f a → (f:+:g) a
R :: g a → (f:+:g) a
```

The functor instance is defined as follows

```
instance (Functor f, Functor g) ⇒ Functor (f:+:g) where
    fmap :: (a → b) → (f:+:g) a → (f:+:g) b
    fmap f (L x) = L (fmap f x)
    fmap f (R y) = R (fmap f y)
```

Now we are ready to define the signature functor for multiplication:

```
data Mul k = Mul k k
```

This is the datatype constructor:

```
Mul :: k → k → Mul k
```

We define its functor instance:

```
instance Functor Mul where
    fmap f (Mul x y) = Mul (f x) (f y)
```

Finally, we can put the `Expr` and `Mul` languages together, to have a language with both addition and multiplication.

```
Fix (Expr :+: Mul)
```

This is essentially the same as describing the following datatype, but in a compositional way:

```
data Expr' = Val' Int
           | Add' Expr' Expr'
           | Mul' Expr' Expr'
```

For practical purposes, we do not work with `Expr'` but with `Fix (Expr :+: Mul)`. We need to write algebras[1] of the form:

```
Expr :+: Mul b → b
```

to reduce a `Fix (Expr :+: Mul)` type to `b`. To do this in a composotional way, we define a way of combining `Expr` algebras and `Mul` algebras. We call this the junction of algebras:

```
(∇) :: (f a → a) → (g a → a) → ((f:+:g) a → a)
(falg ∇ galg) (L x) = falg x
(falg ∇ galg) (R y) = galg y
```

So now, we can give a semantics to the language `Fix (Expr:+:Mul)` by defining algebras.

[1]An *algebra* is any function of type `f a → a`, where `f` is a functor.

```
add :: Expr Int → Int
add (Val x) = x
add (Add x y) = x + y

mul :: Mul Int → Int
mul (Mul x y) = x * y
```

To evaluate, we write:

```
eval :: Fix (Expr :+: Mul) → Int
eval = cata (add ∇ mul)
```

And thus, we have solved the expression problem. In fact, we can decompose the `Expr` into constituent parts:

```
data Val k = Val Int

data Add k = Add k k
```

After defining functor instances, we can define functor instances for:

```
Fix (Val :+: Add :+: Mul)
```

2 Grammars and Parsers

2.1 BNF: Baccus-Naur Form

BNF is a language used to express the shape of grammars. It was invented in around 1958 for the expression of the Algol programming language. A BNF statement is made up of:

- ε represents empty strings.
- `<n>` represents a non-terminal.
- `"x"` represents a terminal.
- `p|q` represents a choice between `p` and `q`.

An example of BNF is the following:

```
<digit> ::= "0" | "1" | "2" | ... | "8" | "9"
    // ::= signifies defintion.
```

We can aproximate numbers by this:

```
<num> ::= <digit>
       |  <digit><num>
```

The core language of BNF is usually extended with some constructs:

```
[ e ]  // optional e
( e )  // grouping e
  e*   // 0 or more repetitions of e
  e+   // 1 or more repetitions of e
```

For a more complex example, consider expressions:

```
<expr> ::= <num>
        |  <num> "+" <expr>
```

This corresponds to the following type:

```
data Expr = Val Num
          | Add Num Expr
```

In principle, we do the same to convert `<num>` into a Num datatype. However, we will approximate this by the type Int.

```
type Num = Int
```

Grammars can sometimes be ambiguous, a single string can be accepted by the grammar in multiple ways:

```
<expr> ::= <num>
        |  <expr> "+" <expr>
```

The problem here is also that `<expr>` is left-recursive: there is a branch which has an `<expr>` before any terminal. This causes problems in recursive descent parsers, which we will study later. The solution is to refactor the grammar.

2.2 Paull's Modified Algorithm

We can remove recursion as follows. Suppose we have the following grammar:

$$\texttt{A ::= A } \alpha_1 \texttt{ | } \cdots \texttt{ | A } \alpha_n \texttt{ | } \beta_1 \texttt{ | } \cdots \texttt{ | } \beta_m$$

To apply the algorithm, we rewrite the term above to be the following:

$$\begin{array}{l} \texttt{A ::= } \beta_1 \texttt{ A' | } \cdots \texttt{ | } \beta_m \texttt{ A'} \\ \texttt{A' ::= } \alpha_1 \texttt{ A' | } \cdots \texttt{ | } \alpha_n \texttt{ A' | } \varepsilon \end{array}$$

In practive here is how we convert the following:

```
<expr> ::= <num>
        |  <expr> "+" <expr>

// becomes

<expr> ::= <num> <expr'>
<expr'> ::= "+" <expr> <expr'> | ε
```

2.3 Parsers

A parser is a function that takes in a list of characters, and returns an item that was parsed, along with the unconsumed string. We can define it by:

```
newtype Parser a = Parser (String → [(a, String)])
```

We can use a parser by defining a function, parse.

```
parse :: Parser a → String → [(a, String)]
parse (Parser px) = px

(px :: String → [(a, String)])
```

Some very simple examples of parsers include the `fail` parser which always fails, the `item` parser which knows how to process a single character, and the `look` parser, which allows one to look into the input stream without consuming anything.

```
fail :: Parser a
fail = Parser ( λ ts → [] )

item :: Parser Char
item :: Parser ( λ ts → case ts of
                  [] → []
                  (t:ts') → [(t,ts')])

look :: Parser String
look = Parser (λ ts → [(ts,ts)])

-- e.g. parse (fail) "Hello" = []
--      parse (item) "Hello" = [('H', "ello")]
--      parse look "Hello" = [("Hello","Hello")]
```

Often we want to transform our parsers from producing values of one type to another. For instance, we might transform a parser for a single `Char` into producing the corresponding `Int`. This is achieved by giving a functor instance for parsers. We can use this to define a `Parser` for `Int`s from our item parser (see the homework).

```
instance Functor Parser where
  -- fmap :: (a → b) → Parser a → Parser b
  fmap f (Parser px) =
          Parser ( λts → [(f x, ts') | (x, ts') ← px ts])

(⟨$⟩) :: (a→b) → Parser a → Parser b
f ⟨$⟩ px = fmap f px
```

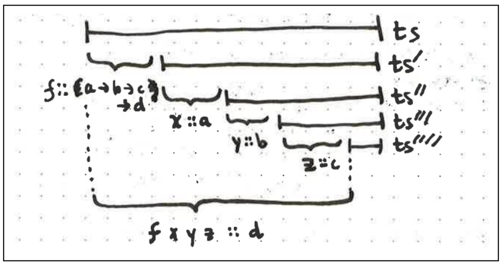
The following variation is often useful:

```
(⟨$) :: a → Parser b → Parser a
x ⟨$ py = fmap (const x) py
```

We can use this to build a function called `skip` that parses input but outputs nothing useful.

```
skip :: Parser a → Parser ()
skip px = () <$ px
```

Now we want to apply a function to the different outputs of parse:



To make something like this we use a combination of ⟨∗⟩ and ⟨$⟩ like this:

```
pf <*> px <*> py <*> pz
```

This uses the `(<*>)` operation, which we will define shortly. The applicative introduces pure and `(<*>)`.

```
class Functor f ⇒ Applicative f where
    pure :: a → f a
    (<*>) :: f (a → b) → f a → f b
```

These have the following definitions:

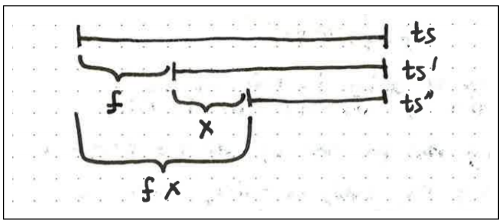
```
instance Application Parser where
    -- pure :: a Parser a
    pure x = Parser (λts → [(x,ts)])
```

The `pure x` parser will not consume any input but always generates the value `x`.

```
parse (pure 5) "hello" = [(5, "hello")]
```

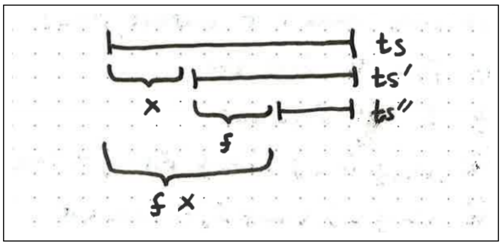
Now we define `(<*>)`, pronounced "ap", for apply.

```
-- (<*>) :: Parser (a → b) → Parser a → Parser b
Parser pf <*> Parser px = Parser ( λts →
                                [(f x, ts'')
                                | (f, ts')  ← pf ts
                                , (x, ts'') ← px ts' ])
```



The operation we defined first parses a function, then a value, and finally applies the function to the value. Thos can be done the other way around too:

```
(<**>) :: Parser a → Parser (a → b) → Parser b
Parser px <**> Parser pf = Parser ( λts →
                              [ (f x, ts'')
                              | (x, ts')  ← px ts
                              , (f, ts'') ← pf ts' ])
```



Other derived operators are (<*) and (*>), their types are:

```
(<*) :: Parser a → Parser b → Parser a
(*>) :: Parser a → Parser b → Parser b
```

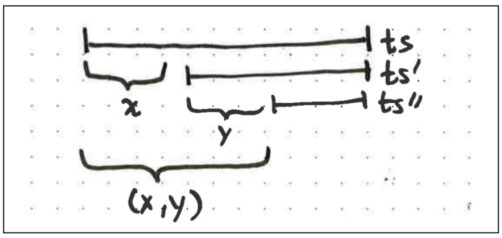
2.4 Monoidal

The Monoidal class is equivalent to the Applicative class:

```
class Monoidal f where
    unit :: f ()
    mult :: f a → f b → f (a,b)
```

For parsers, the monoidal instance is defined as follows:

```
instance Monoidal Parser where
    -- unit :: Parser ()
    unit = Parser ( λts → [((), ts)])

    -- mult :: Parser a → Parser b → Parser (a,b)
    mult px py = Parser ( λts →
                  [ ((x,y), ts'')
                  | (x, ts')  ← px ts
                  , (y, ts'') ← py ts'])
```



It is useful to make mult a binary operation, so we introduce one:

```
((~)) :: Monoidal f ⇒ f a → f b → f (a,b)
px ⟨~⟩ py = mult px py
```

We the derive these useful combinators:

```
((~) :: Monoidal f ⇒ f a → f b → f a
px ⟨~ py = fst ⟨$⟩ (px ⟨~⟩ py)

(~)) :: Monoidal f ⇒ f a → f b → f b
px ~⟩ py = snd ⟨$⟩ (px ⟨~⟩ py)

-- where fst (x,y) = x, snd (x,y) = y.
```

Note that we have this equivalence:

```
(⟨~) = (⟨*)
(~⟩) = (*⟩)
```

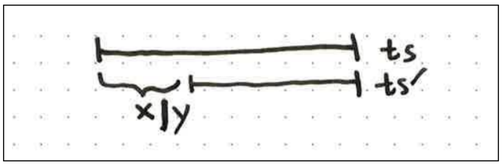
2.5 Alternatives

Now we produce parsers that can deal with choice in a grammar.

```
class Alternative f where
    empty :: f a
    (⟨|⟩) :: f a → f a → f a
```

Here is the instance for parsers:

```
instance Alternative Parser where
    -- empty :: Parser a
    empty = fail -- from before
    -- (⟨|⟩) :: Parser a → Parser a → Parser a
    Parser px ⟨|⟩ Parser py = Parser ( λts →
                                px ts ++ py ts )
```




```
-- parse px ts ++ parse py ts = parse (px ⟨|⟩ py) ts
```

Sometimes we want to extend ⟨|⟩ to many input parsers.

```
choice :: [Parser a] → Parser a
choice pxs = foldr (⟨|⟩) empty pxs
```

We can define a combinator that appends the result of a parse onto others:

```
(⟨:⟩) :: Parser a → Parser [a] → Parser [a]
px ⟨:⟩ pxs = (:) ⟨$⟩ px ⟨*⟩ pxs
```

To understand this, first recall that most parser combinators are left associative.

```
    (:) ⟨$⟩ px ⟨*⟩ pxs
=
    ((:) ⟨$⟩ px) ⟨*⟩ pxs

    -- check the types to see this is correct, recall:
    -- (⟨$⟩) :: (a → b) → f a → f b
    -- (⟨*⟩) :: f (a → b) → f a → f b
```

Now we're ready to define combinators that correspond to + and $*$ from BNF.

- $e+$ is written some e
- $e*$ is written many e

```
some :: Parser a → Parser [a]
some px = px ⟨:⟩ many px
```

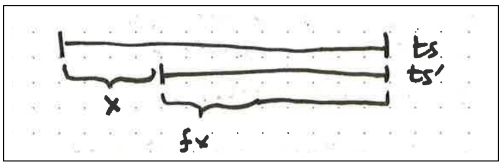
this parses one `px` and appends it to the result of many `px`

```
many :: Parser a → Parser [a]
many px = some px ⟨|⟩ empty
```

2.6 Monadic Parsing

Sometimes we want the control flow a parser to depend on what was parsed. Suppose we have `px :: Parser a`, we can define a function `f :: a → Parser b`. The function `f` inspects the value `x` which came from `px` and produces a new parser accordingly. The result should be a parser of time `Parser b`.

```
instance Monad Parser where
  -- return :: a → Parser a
  return :: pure -- from Applicative

  -- (»=) :: Parser a → (a → Parser b) → Parser b
  Parser px »= f = Parser ( λts →
                  concat [parse  (f x) ts' | (x, ts') ← px ts])
```



To use this combinator we combine a parser with a function. The satify parser takes in a function that is a predicate on `Chars` and returns the parsed value if it satisfies the predicate.

```
satisfy :: (Char → Bool) → Parser Char
satisfy p = item >>= λt → if p t
                            then pure t
                            else empty
```

This is perhaps the most useful combinator. Rather than the monadic definition, we can write one directly:

```
satisfy :: (Char → Bool) → Parser Char
satisfy p = Parser (λts → case ts of
                [] → []
                (t:ts') → [(t,ts') | p t])
```

We can now parse a single character as follows:

```
char :: Char → Parser Char
char c = satisfy (c ==)

-- or char c = satisy (λc' → c == c')
```

Example:

```
parse (char 'x') "xyz" = [('x',"yz")]
parse (char 'a') "xyz" = []
```

2.7 Chain for left-recursion

The problem with ambiguous grammars that are left-recursive can be resolved with Paull's algorithm.

```
<expr> ::= <number> | <expr> "+" <expr>
```

However, without applying Paull's algorithm, we have a nice data type:

```
data Expr = Num Int | Add Expr Expr
```

We can decide to use `chainl1` to parse into this data structure from the original grammar, assuming that + is left associative. (`chainr` exists if we want it to be right associative). Essentially, we have this combinator:

```
chainl1 :: Parser a → Parser (a→a→a) → Parser a
```

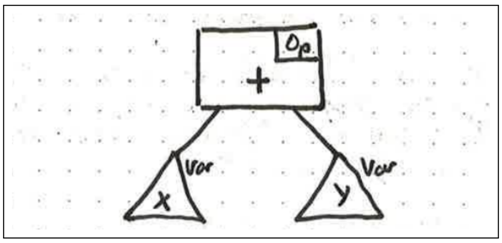
This allows us to write a parser of the form:

```
expr :: Parser Expr
expr = chainl1 number add

add :: Parser (Expr → Expr → Expr)
add = Add <$ tok "+"
```

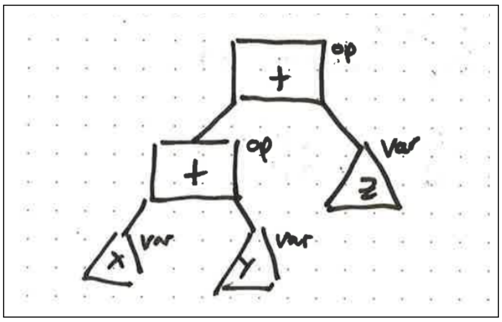
3 Abstraction and Semantics

3.1 The Free Monad

Suppose we are interested in giving a semantics to a language for addition. The syntax for this language could look like `x+y`. This corresponds to a syntax tree:



Or for a more complex example, consider `(x+y)+z`:



We want to give the shape of + nodes by using a signature functor:

```
data Add k = Add k k
```

In Haskell we can also write:

```
data Add k = k :+ k
```

The provision of variables is left to the Free Monad. The free monad `Free f a` provides syntax trees whose nodes are shaped by `f`, and whose variables come from the type `a`.

```
data Free f a = Var a
              | Op (f (Free f a))
```

It is worth comparing this to the definition of `Fix`:

```
data Fix f = In (f (Fix f))
```

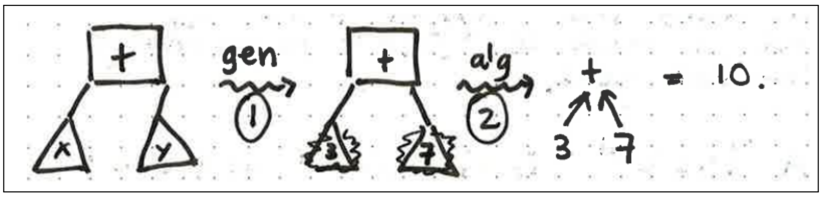
The previous trees can be expressed with the following values of type `Free Add String`.

```
Op (Add (Var "x") (Var "y"))

Op (Add (Op (Add (Var "x") (Var "y")))(Var "z"))
```

To interpret these free trees, we work in two stages:

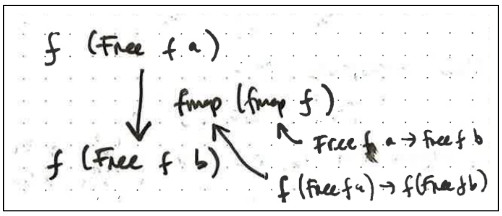
1. We change the variables into values. *(the Generator)*
2. We evaluate the operations. *(The Algerba)*

In pictures, we do this to interpret a tree[2]:



(Stage 1) The first stage involves replacing variables with their corresponding numbers. This is achieved by defining `Free f` to be a functor. This is only possible is `f` is a functor too.

```
instance Functor f ⇒ Functor (Free f) where
 -- fmap :: (a→b) → Free f a → Free f b
    fmap h (Var x) = Var (h x)
    fmap h (Op op) = Op (fmap (fmap h) op)

    -- note that op :: f (Free f a)
```



(Stage 2) The second stage extracts semantics by applying an algebra. This is a recursive function defined as follows (We could use a `cata`, but that is out of the scope of this lecture series).

```
extract :: Functor f ⇒ (f b → b) → Free f b → b
extract alg (Var x) = x
extract alg (Op op) = alg (fmap (extract alg) op)

-- x :: b, op :: f (Free f b)
```

Finally, we can combine these two stages to define an evaluation function:

```
eval :: Functor f ⇒ (f b → b) (a → b) → Free f a → b
eval alg gen = extract alg ∘ fmap gen

-- fmap gen is stage 1, extract alg is stage 2
```

[2] The triangles on the second tree are an error.

In pictures, we can represent an operation with a box, and a variable with a triangle, and `alg` will replace boxes, `gen` will replace triangles. First we define an algebra for a functor. Consider the `Add` functor from before.

```
add :: Add Int → Int
add (x :+ y) = x + y
```

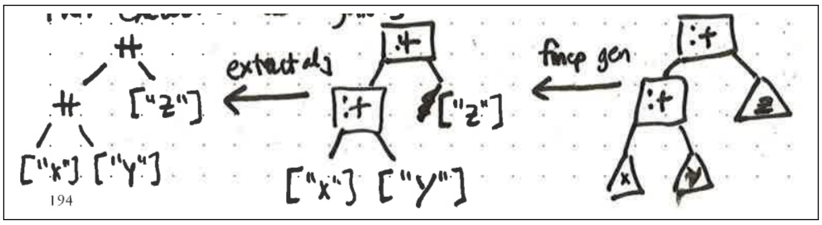
We also need a generator from the type of our variables. Variables are often given as strings:

```
type Var = String
```

The Generator for addition is a function from `Var` to `Int`.

```
env :: Var → Int
env "x" = 3
env "y" = 5
env  _   = 0
```

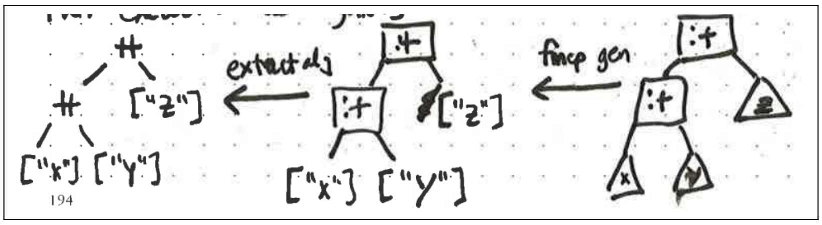
This is the environment for evaluation. Suppose we want to evaluate this tree:



This is an example of `eval add env`. A second example is to collect all variables in an expression as a list. To do this we provide a function `Vars`, which is defined used `eval`.

```
vars :: Free Add Var → [Var]
vars = eval alg gen
    where
        gen :: Var → [Var]
        gen x = [x]

        alg :: Add [Var] → [Var]
        alg (xs :+ ys) = xs ++ ys
```

This executes as follows:



Suppose we want to add an operation to our language, that performs division.

```
data Div k = Div k k
```

If we want to provide a semantics that collects all the variables, we must provide an algebra.

```
divVars :: Div [Var] → [Var]
divVars (Div xs ys) = xs ++ ys
```

If we want a language with both addition and division, we need to take the coproduct of `Add` and `Div`. This means expressions of the form `Add :+: Div`. For example, we can work with `Div` alone:

```
evalDiv :: Free Div Var → [Var]
evalDiv = eval alg gen where
    gen :: Var → [Var]
    gen x = [x]

    alg :: Div [Var] → [Var]
    alg (Div xs ys) = xs ++ ys
```

Dealing with both `Add` and `Div` at once requires this:

```
vars :: Free (Add :+: Div) Var → [Var]
vars = eval alg gen where
    gen x = [x]

    alg :: (Add :+: Div) [Var] → [Var]
    alg (L (Add xs ys)) = xs ++ ys
    alg (R (Div xs ys)) = xs ++ ys
```

When we try to evaluate this language, naively, we encounter a problem.

```
expr :: Free (Add :+: Div) Var → Double
expr = eval alg gen where
    gen :: Var → Double
    gen = env -- this function magically knows how to assign
                 values to variables

    alg :: Add :+: Div Double → Double
    alg (L (Add x y)) = x + y
    alg (R (Div x y)) = x / y
```

The sad truth is that this function is broken. To see why, consider `alg (R (Div x 0))`. This will fail! To fix this problem, we must be upfront about the fact that an error can happen. The basic way to do this is to interpret into a `Maybe` datatype.

```
expr :: Free (Add :+: Div) Var → Maybe Double
expr = eval alg gen where
    gen = env

    alg (L (Add x y)) = mAdd  -- {note x :: Maybe Double}
    alg (R (Div x y)) = mDiv
```

We must now define `mAdd` and `mDiv`, with division we are sensitive to zero:

```
mAdd :: Maybe Double → Maybe Double → Maybe Double
```

```
mAdd (Just x) (Just y) = Just (x + y)
mAdd mx my             = Nothing

mDiv :: Maybe Double → Maybe Double → Maybe Double
mDiv (Just x) (Just 0) = Nothing
mDiv (Just x) (Just y) = Just (x / y)
mDiv mx my             = Nothing
```

3.2 Failure

We need to create syntax for failure:

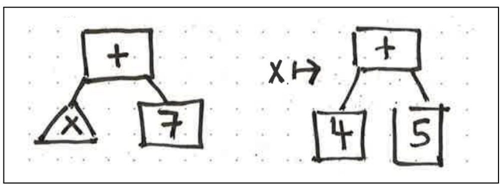
```
data Fail k = Fail

instance Functor Fail where
    fmap f Fail = Fail
```

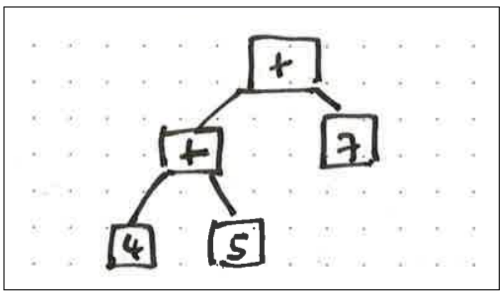
The functor instance shows us that computations can not follow a fail. If we deal with division alone, we have this:

```
evalFail :: Free Div Double → Free Fail Double
evalFail = eval alg gen
    where
        gen :: Double → Free Fail Double
        gen x = Var x

        alg :: Div (Free Fail Double) → Free Fail Double
        alg (Div (Var x) (Var 0)) = Op Fail
        alg (Div (Var x) (Var y)) = Var (x / y)
        alg (Div    tl      tr )  = Op Fail
```

3.3 Substitution

Substitution in a language is a very useful feature. For example, consider `x+7`. We can evaluate this into a new syntax tree when we have a notion of substitution, where we might bind `x` to another expression rather than just a constant, like `x → x+5`, then we expect the above to become `(4+5)+7`. We can depict this by the following trees.





We will define substitution using code. Usually an expression `e` with a variable `x` is substituted with `e'` with the following syntax:

```
e [x → e']
```

where `e` corresonds to `x+7`, and `e'` is `4+5`. Sometimes we write:

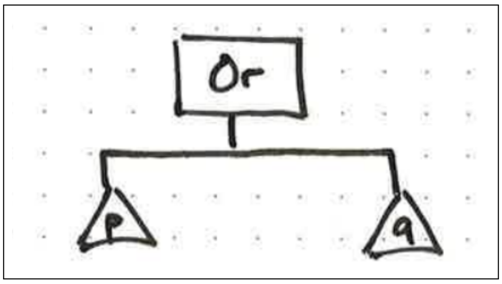
```
e [x \ 4+5]
-- or
e [4+5 / x]
```

For our purposes, a syntax tree is given by a datatype `Free f a`, where `f` is the shape of the syantx, and `a` is the type of the variables, substitution is defined by `(>>=)` as follows:

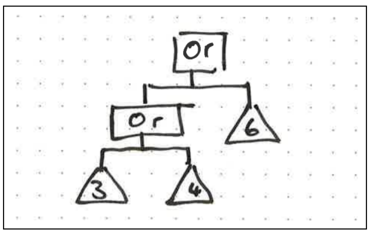
```
(>>=) :: Free f a → (a → Free f b) → Free f b
Var x >>= f  =  f x
Op op >>= f  =  Op (fmap (>>= f) op)
```

3.4 Non-Determinism

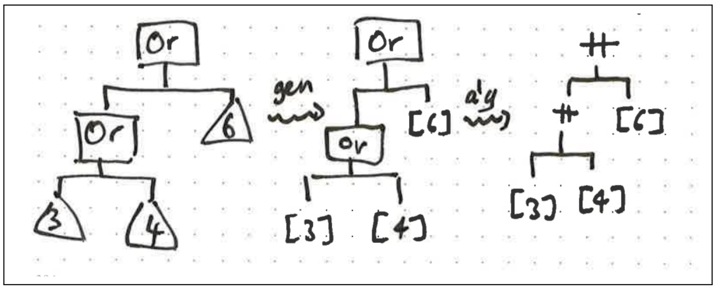
A non-deterministic computation is one that provides the choice between two different computations. For example, `p □ q` is the program that gives answers from p or q.



Here we use `Or` to represent □.



One interprets this tree as follows:



In terms of code, we first need to express syntax, we must also create a functor instance. With this in place, we can define an evaluation function:

```
data Or k = Or k k

instance Functor Or where
    fmap f (Or x y) = Or (f x) (f y)

list :: Free Or a → [a]
list = eval alg gen where
    gen :: a → [a]
    gen :: x =  [x]

    alg :: Or [a] → [a]
    alg (Or xs ys) = xs ++ ys
```

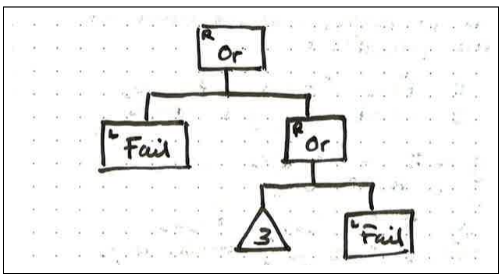
Another interpretation of these trees is to simply return the first result. We can define the semantics using `once`.

```
once :: Free Or a → Maybe a
once = eval alg gen where
    gen :: a → Maybe a
    gen x = Just x

    alg :: Or (Maybe a) → Maybe a
    alg (Or Nothing y)  = y
    alg (Or (Just x) y) = Just x
```

This works, but we want a way of signalling that there was no solution. For this, we will make use of `Fail`. Nondeterminism is the syntax provided by the following synonym.

```
type Nondet a = (Fail :+: Or) a
```

Trees of type `Free Nondet a` have this shape:



As before, we give semantics to `Nondet` languages by providing a generator and an algebra.

```
list :: Free Nondet a → [a]
list = eval alg gen where
    gen :: a → [a]
    gen x = [a]

    alg :: Nondet [a] → [a]
    alg (L Fail)     = []
    alg (R (Or x y)) = x ++ y
```

The semantics for `once` is similar.

3.5 Alternation

An alternative way to do `Or` is to model a pair of `k` values as a function from `Bool`. For this, we define the following:

```
data Alt k = Alt (Bool → k)

instance Functor Alt where
    fmap :: (a → b) → Alt a → Alt b
    fmap f (Alt k) = Alt (f ∘ k)
```

The idea is that we parse `True` when we want the first child, and `False` for the second child. Now we can give different semantics for nondeterminism.

```
type Nondet’ a = (Fail :+: Alt) a

list :: Free Nondet’ a → [a]
list = eval alg gen where
    gen :: a → [a]
    gen x = [x]

    alg :: Nondet’ [a] → [a]
    alg (L Fail)    = []
    alg (R (Alt k)) = k True ++ k False

    -- k :: Bool → [a]
```

This demonstrates that the parameter to a syntax functor sometimes has the form of a function, i.e. `Bool → k`.

3.6 State

A stateful computation can be modelled by having two operations, `Get` and `Put`.

```
data State s k = Put s k
               | Get (s → k)
```

The intuition is that `Put s k` will put the value `s` into the state before continuing with the computation `k`. The `Get f` operation will only continue when `f :: s → k` is given a variable of type `s`. The semantic domain for `State` is a function of type:

```
s → (a,s)
```

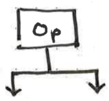
This is a carrier for stateful computations.

```
evalState :: Free (State s) a → (s → (a,s))
evalState = eval alg gen where
    gen :: a → (s → (a,s))
    gen x s = (a,s)
    -- or equivalently gen x = λs → (x,s)

    alg :: State s (s → (a,s)) → (s → (a,s))
    alg (Put s' k) = λs → k s'
    alg (Get k)    = λs → k s s
    -- in k s s, the first s is the state that generates programs,
       and the second is the state parsed on to future programs.
```

This function carries on computations generated by `k` when supplied with the new state `s`.

3.7 Diagrams of operations

We have already seen many diagrams, here are some conventions:



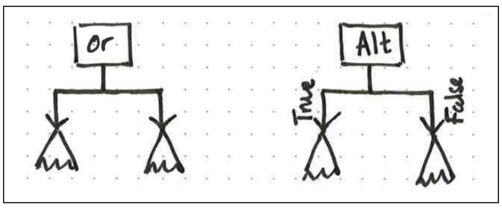
- Operations are written in boxes, and the arrows emerging below represent different *k* values.

- Trianglular leaves represent variables.

- We will often represent an arbitrary subtree with this.

To represent nondeterminism and alternation, we have these diagrams:

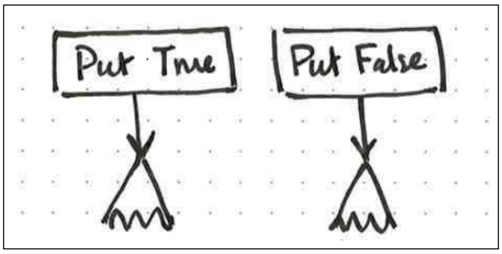
```
data Or  k = Or k k
data Alt k = Alt (Bool → k)
```



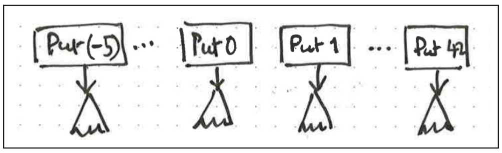
We can imagine that `State s = Put s :+: Get s`:

```
data Put s k = Put s k
data get s k = Get (s → k)
```

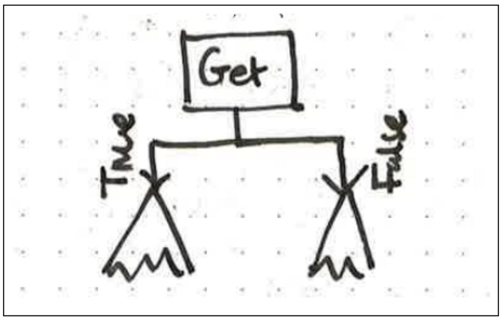
To draw the operation `Put`, we have the following:



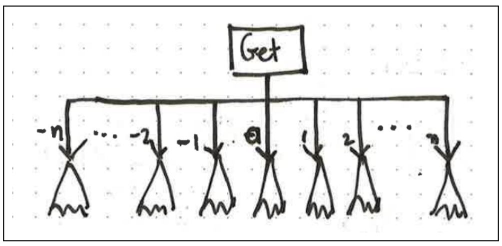
These examples are of `Free (Put Bool) a` syntax trees: Since `s = Bool` we can only construct these two `Put` nodes. We cab parameterise `s` different, so if we had `s = Int`, we would have a huge (infinite) number of nodes:



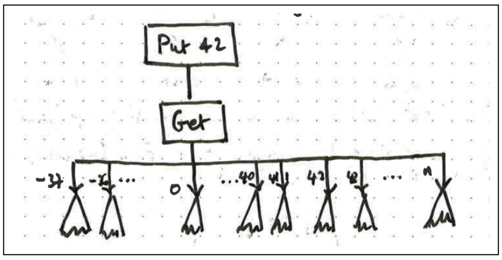
The `Get` nodes are a generalisation of `Alt`. If we consider trees of the form `Free (Get Bool) a` then this is as follows:



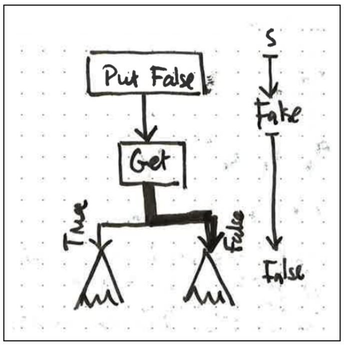
However, with `s = Int`, we have `Free (Get Int) a`:



Note that `Alt` and `Get Bool` are not syntactically very similar; they share the same structure but their differences are exhibited when we provide different algebras. Another thing we can do is compose trees together:



When we gave the algebra for `State s` when the carrier was `s → (a,s)`, we were storing the state `s` in output, and reacting as `s` as input. Consider `s = Bool`:



Note that here we have used `False` to choose which child to use, and we also passed `False` on to the next stage.

```
alg (Get k) = λs → k s s

-- The k in 'Get k' is represented by the children in the diagram.
-- The first s in 'k s s' is selecting the child.
-- The second s is passing the s onwards.
```

These syntax trees all correspond to values of type `Free (State s) a`. They can be cumbersome to work with because we must wrap everything in `Op`.

```
Op (Put False (Op (Get (λs → ...))))
```

We can avoid this by introducing smart constructors for `Put` and `Get`:

```
put :: s → Free (State s) ()
put s = op (Put s (Var ()))

get :: Free (State s) s
get = Op (Get (λs → Var s))
```

And with these tools, we can simply substitute

```
put False ≫ get
```

and this produces the same tree as above.