LANGUAGE ENGINEERING

- A Programming language consists of three main components. These are:
  - **Syntax** – the shape of grammar | 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.
  ↳ Not necessarily Turing-complete

- Some DSLs come equiped with all the features of general-purpose languages:
  - Parser
  - Compiler
  - Syntax-highlighting
  - Documentation
  - IDE

- An embedded domain-specific language (EDSL) is defined within another (host) language.
  ↳ Advantage is that there's less work to perform. This is at the cost of restricted flexibility

- EDSL can be either deep or shallow embedding

Deep Embedding - Syntax in concrete datatypes, Semantics given by evaluation

Shallow Embedding - Syntax is borrowed from host Semantics directly given

Example, consider :

"3 + 5" in a String

Denotational brackets → ⟦ 3+5 ⟧ :: Int

- we use denotational brackets to ascribe a semantics:

⟦ 3 + 5 ⟧ = ⟦3⟧ + ⟦5⟧ = 3 + 5 = 8

DEEP IMBEDDING

- we can model this using a deep embedding in Haskell with the following code

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

- concrete syntax for 3+5 is then give by
- Add (var 3) (var 5) :: Expr
- A parser is a function that takes a string and produces such a concrete representation

- The semantics is given by an evaluation function:

```
> eval :: Expr -> Int
> eval (var n) = n
> eval (Add x y) = eval x + eval y
```

- Notice that instead of [[3+5]], we can now write : eval (Add (var 3) (var 5))

SHALLOW EMBEDDING

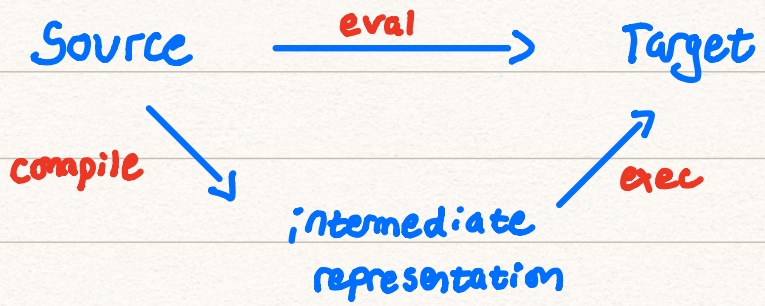
- The shallow embedding is given directly by functions: (we will redefine Expr here)

```
> 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)

COMPILERS

- A compiler is code that transforms a source 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. This assembly is executed in the terminal

- Javascript tends to ignore the compile stage since it is an interpreted language. The web browser performs eval, which turns it into rendered output

- Haskell has two modes. When using ghc it compiles its files into assembly to be executed in terminal. However, when using ghci it takes source and interprets it directly by evaluating in terminal.

CASE STUDY : CIRCUIT LANGUAGE

[Golding domain-specific languages by Gibbons]

- The circuit language is a DSL for describing circuits. It consists of several operations

* identity :: Int -> Circuit

e.g. identity 3 = | | |

* beside :: Circuit -> circuit -> Circuit

e.g. beside (| | |) (| |) = | | | | |

* above :: Circuit -> Circuit -> Circuit

e.g. above (⋀ |) (| ⋀) = |⋀

↳ This assumes that the width of the two circuits is equal

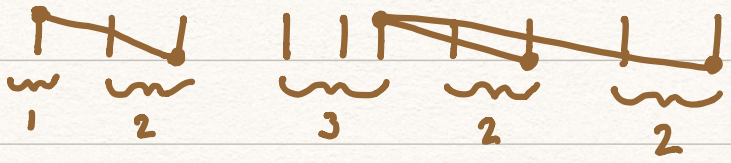
* fan :: Int -> circuit

e.g. fan 4 = 

* Stretch :: [Int] -> Circuit -> Circuit

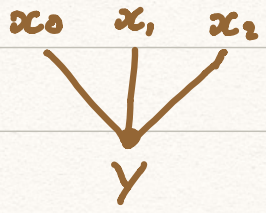
e.g. stretch [1, 2, 3, 2, 2]

=



- Information is combined by joining lines, and applying the operation (∘) "blob"

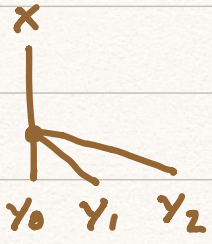
e.g.



$Y = X_0 \bullet X_1 \bullet X_2$

- Information is duplicated when lines separate

e.g.



$x = Y_0 = Y_1 = Y_2$

DEEP EMBEDDING

- There are many different ways of interpreting this circuit language.

e.g. find width of circuit
find height of circuit

- we start with a deep embedding

* First create a data structure for abstract syntax
* Define a semantics with an evaluation function

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

LANGUAGE DESIGN (Circuit Language continued)

We can interpret the circuit language by stipulating that the semantics of a term is the width of the circuit generated

```
> 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 c2
> width (Fan n) = n
> width (Strecth ws c) = Sum ws
```

Choice of semantics here

We can give 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 intertwined 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

```
> type Connected = Boolean
> connected :: Circuit --> Connected
> connected (Identity n) = True
> connected (Beside c1 c2) = Connected c1 ∧ connected c2
> connected (Above c1 c2) = connected c1 ∧ connected c2
                          ∧ width c1 == width c2
> connected (Fan n) = True
> connected (stretch ws c) = connected c
                           ∧ width c == length ws
                           ∧ ((not . elem 0) ws)
```

SHALLOW EMBEDDING

- In a shallow embedding we simply have to give a semantics in terms of the operation 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 problematic because:

- It's hard to add a different semantics, in order to do so we have to redefine Circuit, but this might break any code that depends on the old definition
- Aditionally, a dependent semantics requires all of the interpretations to be considered at once. This is not compositional

- However in shallow semantics it is easy to extend a language with new operations since

this envolves adding new functions : nothing breaks

↳ in Deep, a new constructor must be added

so all semantics must be adjusted accordingly

THE EXPRESSION PROBLEM

- The expression problem concerns itself with finding a solution the following:
  * Is it possible to extend the syntax and semantics of a language in a modular fashion

- 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 do not want to add a new multiply constructor.

Similarly consider:

```
eval :: Expr -> Int
```

- Again we want to extend the semantics without modifying the code.
  ↳ In this case, adding semantics is easy because we simply write another function of type Expr -> b)

- To solve the expression problem we will study the 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.

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)
```

- So the first step is to deconstruct the type of lists to expose the generic structure

- The definition of lists is the same as the following when we remove syntactic sugar:

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

- we remove recursion from this datatype, and mark it with a parameter `k' for Kontinuation

```
> data ListF a k = EmptyF
               | ConsF a k
```

Shows us where List a was recursive

- We now make the recursive parameter something you can change programmatically by giving a functor instance to a ListF a .

```
> instance Functor (ListF a) where
>   fmap :: (x -> y) -> ListF a x -> ListF a y
```

- we now need to define a function or a type that gives us the fixed point of data

- This is defined as follows:

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

- This data type allows us to generalise all recursive data types (except mutually recursive)

- For example, instead of List a, we can write : Fix (ListF a)

- To demonstrate this, we show that List a and Fix (List f a) are isomorphic:

```
> toList   :: Fix (ListF a) -> List a
> FromList :: List a -> Fix (ListF a)
```

- We say that `List a' and `Fix (List F a)' are isomorphic when :

(toList • fromList) = id and (fromList • toList) = id

- In other notation we write : List a $\cong$ Fix (ListF a)

- Let's define these functions:

```
> FromList :: List a -> Fix (ListF a)
> FromList Empty = In Empty F
```

F (Fix F)

= ListF a (Fix (ListF a))

- Some examples of values of type Fix (ListF a) are:

```
In  EmptyF  :: Fix (ListF a)
    ListF a (Fix (ListF a))
```

This works because EmptyF :: ListF a b for any a or b

- Note that the type of In is:

```
In :: f (Fix f) -> Fix f
```

- So the example above is where f = ListF a

```
In (ConsF 5 (In EmptyF))
```

Fix (ListF a)

Fix (ListF a)

- For two elements we have:

```
In ((ConsF 6 (In (ConsF 7 (In EmptyF)))))
```

- Now we have enough to finish a definition of fromList:

```
> fromList :: List a -> Fix (ListF a)
> fromList Empty = In EmptyF
> fromList (cons x xs) = In (consF x (fromList xs))
```

(x : a, xs : List a)

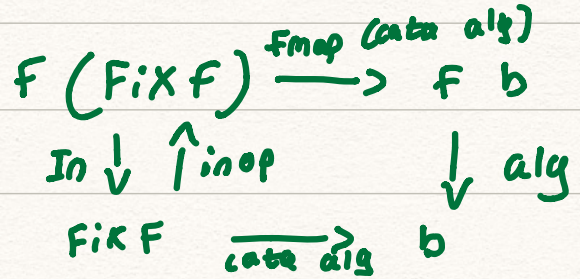
- we are now ready to generalise fold to be a catamorphism:

- consider functions of type ListF a b -> b

```
h :: ListF a b -> b
h EmptyF = k
h (consF 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
```

f (Fix f)

- 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 :: Functor f => (f b -> b) -> Fix f -> b
> 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 cata:

```
> toList :: Fix (ListF a) -> List a
> toList = cata alg
```

```
>     where
>         alg :: ListF a (List a) -> List a
>         alg EmptyF = Empty
>         alg (ConsF x xs) = Cons x xs
```

Here is another function:

```
> toList' :: Fix (ListF a) -> [a]
> toList' = cata alg
>     where
>         alg :: ListF a [a] -> [a]
>         alg EmptyF = []
>         alg (ConsF x xs) = x : xs
```

CATAMORPHISM (continued)

- We can also define a function that returns the length of a Fix (ListF a)

```
> length :: Fix (ListF a) -> Int
> length = cata alg where
>     alg :: ListF a Int -> Int
>     alg EmptyF = 0
>     alg (ConsF x y) = 1 + y
```

(y : Int)

- Example evalvation: (ignoring F's on cons & Empty)

```
  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
```

- Here's another example 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
```

PEANO NUMBERS

- A Peano number is either zero, or the successor of another peano number

```
> data Peano = Z
>            | S Peano
```

- So the number 3 is written S (S (S Z))

* Step 1 -> make Signature functor for Peano

```
> data Peano K = Z
>              | S K
```

we identified the recursive call in Peano

* Step 2 -> Define a functor instance for Peano

```
> instance Functor where
>     Fmap :: (a -> b) -> Peano a -> Peano b
>     Fmap f Z     = Z
>     Fmap f (S x) = Σ F x
```

* Step 3 -> Profit, write function using cata

```
> toInt :: Fix Peano -> Int
> toInt = cata alg where
```

```
> alg :: Peano Int -> Int
> alg Z = 0
> alg S x = 1 + x
```

- Now we can define a doubling function

```
> double :: Fix Peano -> Fix Peano
> double = cata alg where
    alg :: Peano Int -> Peano Int
    alg Z = Z
    alg S x = In (S (In S (x)))
```

x : Fix Peano

(In S (x))) : doubled the number of S

COMPOSING LANGUAGES

- Previously, we looked at the following datatype as the language for addition:

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

- we 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:

```
> Fix Expr = Expr
```

- Suppose we want to add multiplication to this language; we need a way to extend Expr with more constructors. This is acheived 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 f, and g, and makes the functor (f :+: g). It introdues 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 ready for signature functor for multiplication:

```
> data Mul k = Mul' k k
```

- This introduces datatype constructor:

```
> Mul' :: k -> k -> Mul k
```

- Define its functor instance:

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

- Finally we can combine 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'   } from Expr
             | Mul' Expr' Expr'   } from Mul
```

- For practical purposes we don't work with Expr', but with Fix (Expr :+: Mul)

- we need to write algebra of the form:
  `Expr :+: Mul  b -> b` to reduce a 'Fix (Expr :+: Mul)' tree to a 'b'

- To do this in a compositional way, we define a way of combining Expr algebras and Mul algebras: we call this the function of algebras:

```
> (▽) :: (F a -> a) -> (g a -> a) -> ((F :+: g) a -> a)
> (Falg ▽ galg) (L x) = falg x
> (Falg ▽ galg) (R y) = galg y
```

- Can now give semantics language Fix (Expr :+: Mul) by defining algebras:

```
> 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)
```

- Expression problem solved
- Able to now decompose Expr into constituent parts:

```
> data Val k = Val Int

> data Add k = Add k k
```

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

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

- Given a functor f, an algebra is any function of type `f a -> a`

PARSERS

"A parser of things, is a function of things, to lists of pairs of things and strings"

– Fritz Ruehr

BACKUS - NAUR FORM (BNF)

- Language used to express the shape of grammars, invented in 1958 for expression of Algol programming language

- A BNF statement is made up of:

ε	=	Empty strings
<n>	=	Non-terminal
"x"	=	Terminal
p | q	=	Choice between p, q

- An example of BNF is the following:

<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

(::= : is defined to be)

- We can approximate 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 principal 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 ambigvous - 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-decent pasers which we'll cover later
  ↳ The solution is to refactor the grammar

PAUL'S MODIFID ALGORITHM

- we can remove recursion as follows:

- Suppose we have the following grammar

$$A ::= A\,\alpha_1 \mid \ldots \mid A\,\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$$

- where a is a non-terminal and α_i, β_j are BNF expressions

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

$$A ::= \beta_1 A' \mid \ldots \mid \beta_m A'$$
$$A' ::= \alpha_1 A' \mid \ldots \mid \alpha_n A' \mid \varepsilon$$

- In practice here is how to convert the following:

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

<expr> ::= <num> <expr'>

<expr'> ::= "+" <expr> <expr'> | ε

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 p) = p
```

String -> [(a, String)]

- Now we define combinators that allow us to build parsers:

```
> Fail :: Parser a
> Fail = Parser ( λ ts -> [])
```

- This parser always fails to parse:

```
parse (fail) "Hello" = []
```

```
> item :: Parser a
> item = Parser (λ ts -> case ts of
>                 [] -> []
>                 (t:ts') -> [(t, ts')])
```

[(char, string)]

- Here we have:

parse (item) "Hello" = [('H', "ello")]

- Sometimes it is usegul to look at the input stream without consuming anything:

```
> look :: Parser a
> look = Parser (λ ts -> [(ts, ts)])
```

parse (look) "Hello" = [("Hello", "Hello")]

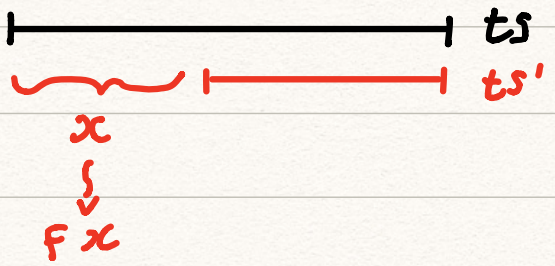
- Often we want to transform our parsers from producing values of one type to another. For instance, we might transform a parser for a single char into producing the corresponding int

- This is acheived by giving a Functor instance for parsers:

```
> instance Functor Parser where
>     -- fmap : (a -> b) -> Parser a -> Parser b
```

```
> fmap f (parser px) =
                  string -> [(a, string)]
>     Parser (λ ts -> [ (f x, ts')
                      | (x, ts') <- px ts ]
```

- Here is a diagram of what is happening



- We can use this to define a parser for Ints from our 'item' parser

- Now we introduce some combinators, `<$>`, `<$` :

```
> (<$>) :: (a -> b) -> Parser a -> Parser b
> f <$> px = fmap f px
```

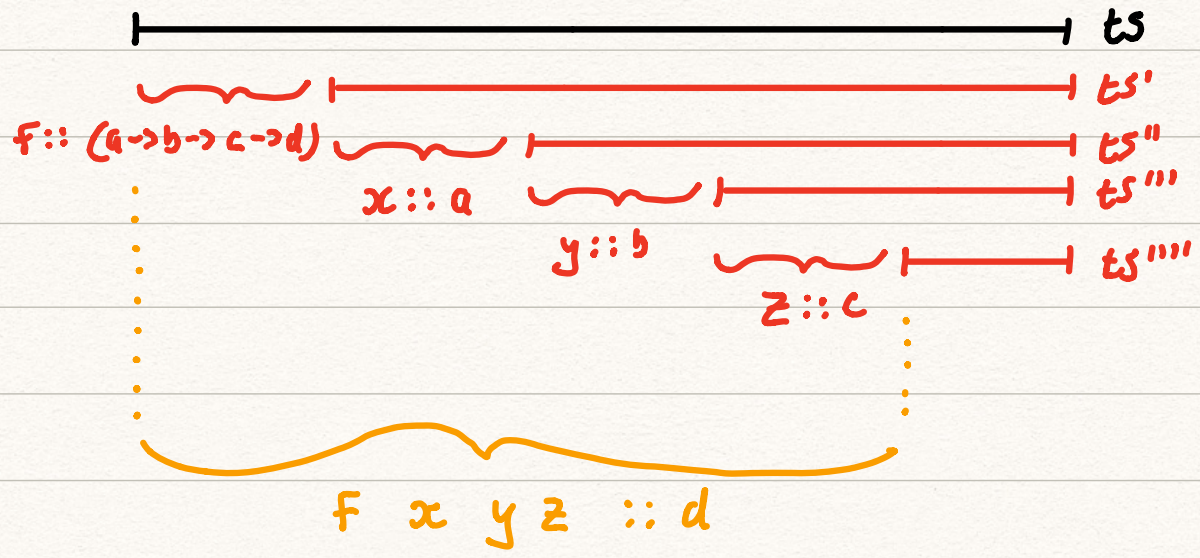
- The following variation is often useful :

```
> (<$) :: a -> Parser b -> Parser a
> x <$ py = fmap (const x) py
```

- We can use this to build a function called 'skip' that parses inputs, but outputs nothing useful

```
> skip :: Parser a -> Parser ()
> skip px = () <$ px
```

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



- To make something like this, we use a combination of <*> or <$> like this:

```
pf <*> px <*> py <*> pz :: Parser d
```

- This uses the (<*>) operation, which we will define shortly

- The applicative class introduces pure and (<*>)

```
> class Functor f => Applicative f where
>     pure :: a -> f a
>     (<*>) :: f (a -> b) -> fa -> f b
```

- These have the following definitions:

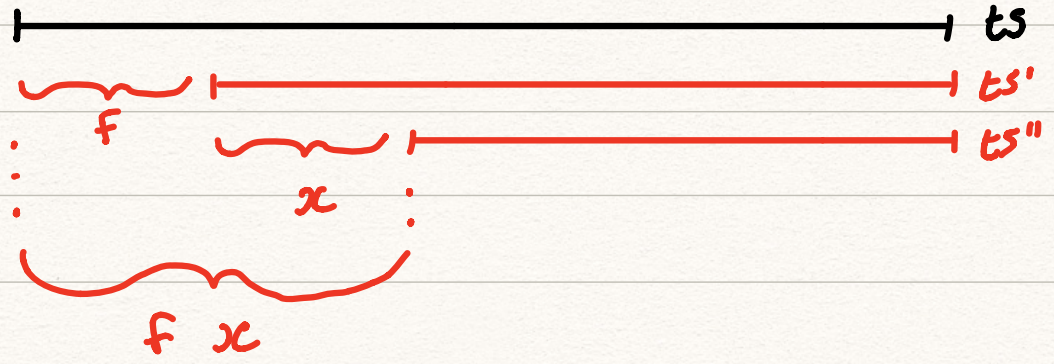
```
> instance Applicative Parser where
>   -- pure :: a -> Parser a
>      pure x = Parser (λts -> [(x, ts)])
```

- The `pure x' Parser will not consume any input but always generates the valve `x'

```
parse (Pure 5) "Hello" = [(5, "Hello")]
```

- Now we define `<*>' (app)

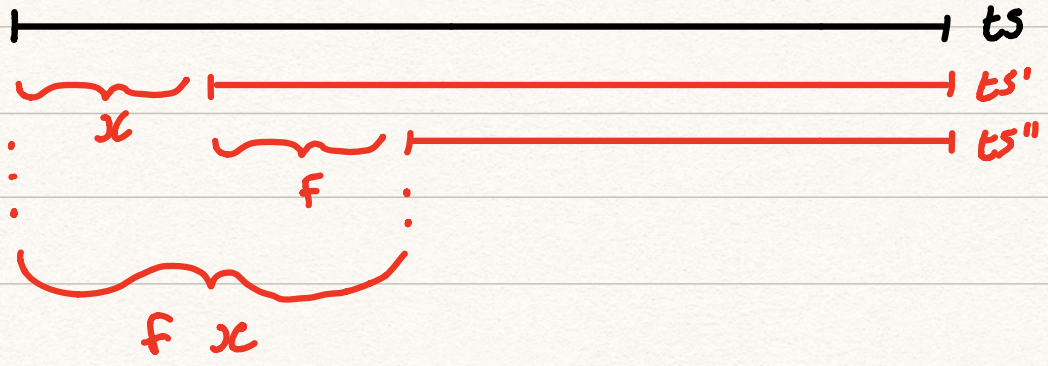
```
> -- (<*>) :: Parser (a -> b) -> Parser a -> Parser b
> Parser pf <*> Parser px = Parser (λts ->
>                             [(f x, ts'')
>                              | (f, ts') <- pf ts
>                              , (x, ts'') <- px ts'])
```



- The operation we just defined first parses a function, then a valve, and then applies the function to the valve

- This can be done the other way round

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



- Other derived operators are (<*) and (*>)

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

- The monoidal type class is equivalent to Applicative:

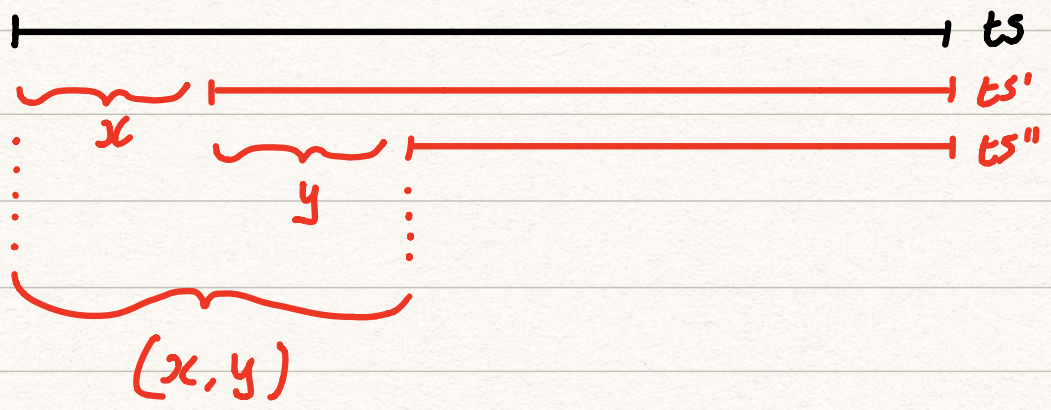
#MONOIDAL

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

```
const :: a -> b -> a
const x y = x
```

- For Parsers the monoidal instance is as follows:

```
> Instance Monoidal Parser where
> -- unit :: Parser ()
>    unit = Parser (λ ts -> [((), ts)])
>
> -- mult :: Parser a -> Parser b -> Parser (a,b)
>    mult Parser px Parser py = (λ ts ->
>                                [ (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 derive these useful combinators

```
> (<~) :: Monoidal F => Fa -> Fb -> Fa
>    px <~ py = Fst <$> (px <~> py)
```

```
> (~>) :: Monoidal f => fa -> fb -> fb
> px ~> py = snd <$> (px <~> py)
```

(fst (x, y) = x , snd (x, y) = y)

- Note that we have this equivalence:

```
(<~) = (<*)
(~>) = (*>)
```

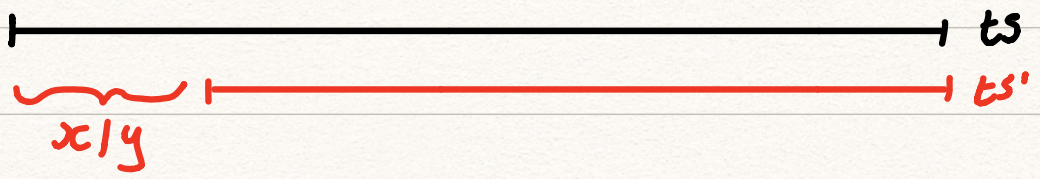
#ALTERNATIVES

- Now we can produce parsers that can deal with the choice in a grammars

```
> 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 (\ts ->
                                  px ts ++ py ts)
```




```
parse px ts ++ parse py ts = parse (px <|> py) ts
```

- Sometimes we want to extend <|> for many input parsers

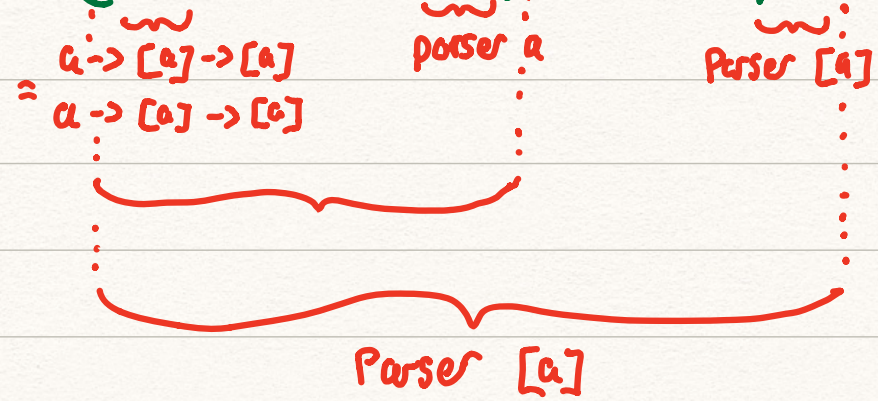
```
> choice :: [Parser a] -> Parser a
> choice pxs = foldr (<|>) empty pxs
```

- 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
= a -> [a] -> [a]
```



Note:

```
(<$>) :: (a->b) ->
         Fa -> Fb

(<*>) :: F(a->b) ->
         Fa -> Fb
```

- Now we're ready to define combinators that correspond to + and * from BNF:
  - ↳ e+ is written as some e
  - ↳ e* is written as 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 <|> pure []
```

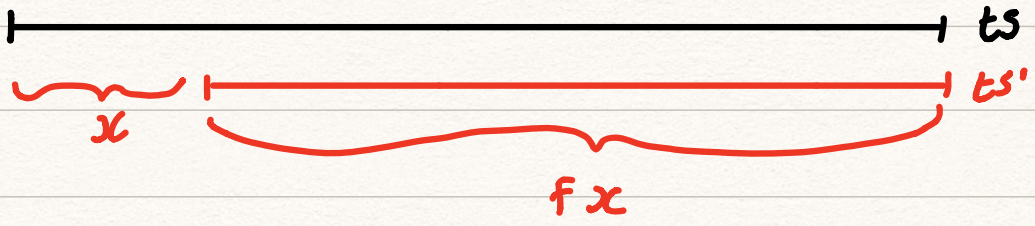
#MONADIC PARSING

- Sometimes we want the control flow of 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 type `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 (λ ts ->
>                   concat [parse (f x) ts'
>                   | (x, ts') <- px ts ])
```



- To use this combinator we combine a parser with a function

- The satisfy 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 >>= λ t -> 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 (λ ts -> case ts of
                        [] -> []
                        (t:ts') -> [(t, ts') | p t])
```

```
(t:ts') -> [(t,ts') | p t])
```

this is equivalent to: if p t then [(t,ts')]
else []

- we can now parse a single character as follows:

```
> char :: Char -> Parser Char
> char c = satisfy (c ==)
```

or in other words:

```
> char c = satisfy (λ c' -> c == c')
```

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

CHAIN FOR LEFT-RECURSION

- The problem for ambiguous grammars that are left recursive can be resolved w/ 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 datastructure 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
```

↳ definition is complex, don't need to know

- This allows us to write a parser of the form

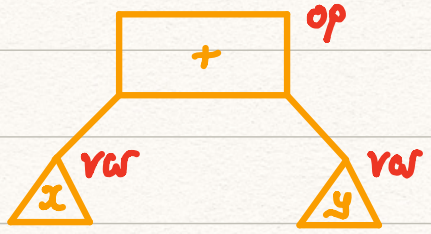
```
> expr :: Parser Expr
> expr = chainl1 number add

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

THE FREE MONAD (aka abstract syntax)

- Suppose we are interested in giving a semantics to a language for addition. The syntax for this language could look like the following:

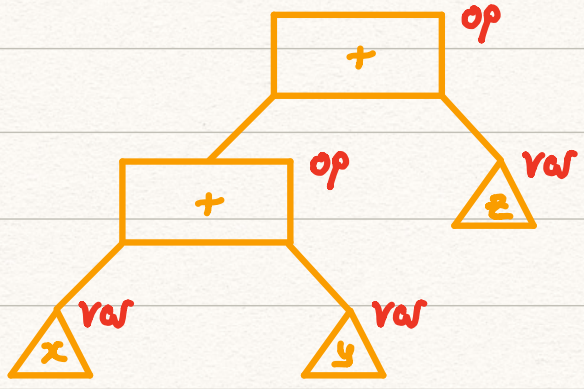
$$x + y$$

- This corresponds to a syntax tree:



- For a more complex example:

$$(x + y) + z$$



- 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))
```

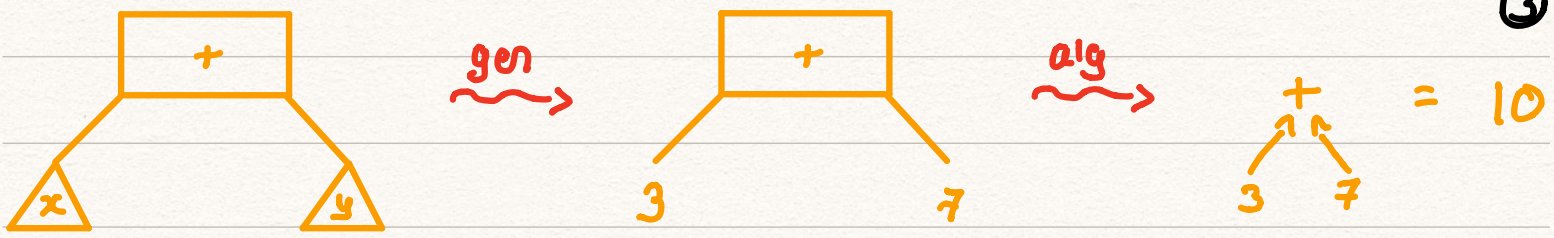
- The trees (on previous page) can be expressed with the following values of type `Free Add String`

```
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) Change variables into a value - generator
2) evaluate the operations - algebra

- In pictures we do the following:



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 if `f` is a functor too

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

f(Free f a)

f (Free f a)
↓ fmap (fmap f)
f (Free f b)

②

- The second stage extracts semantics by applying an algebra

- This is a recursive function defined as follows overleaf

↳ 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)
```

F b->b F (Free f b)

(3)

- Finally, we can combine these two stages to become an evaluation function

```
> eval :: Functor f => (f b -> b) -> (a->b) -> Free f a -> b
> eval alg gen = extract alg . fmap gen
```

- In pictures we can represent an operation with a box, and a variable with a triangle, and alg will replace boxes and 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 variables. Variables are often given as strings:

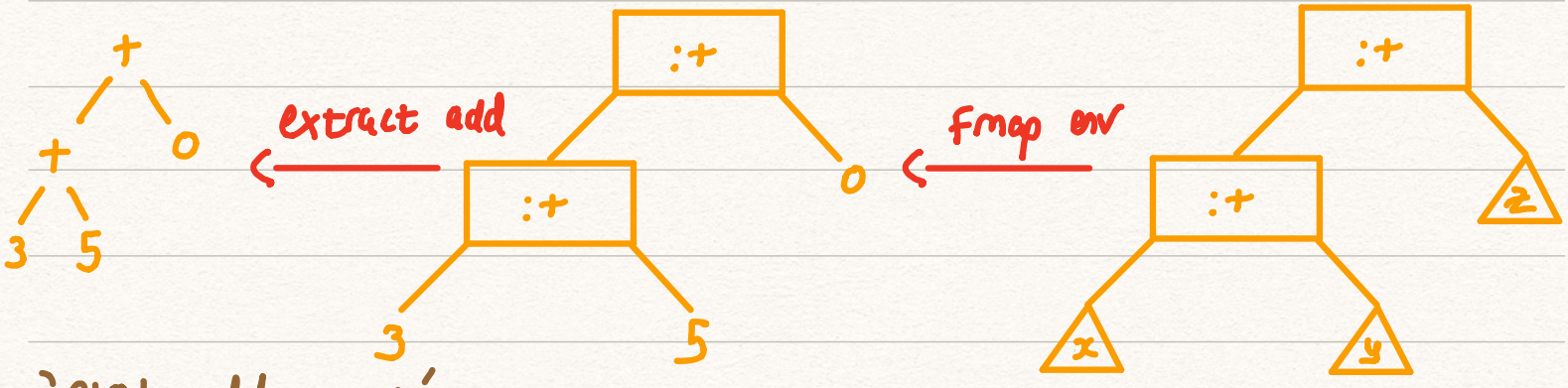
```
> type Var = String
```

- The generator for addition is a function Var -> Int

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

environment for evaluation

- Suppose we want to evaluate this tree:



`eval add env'

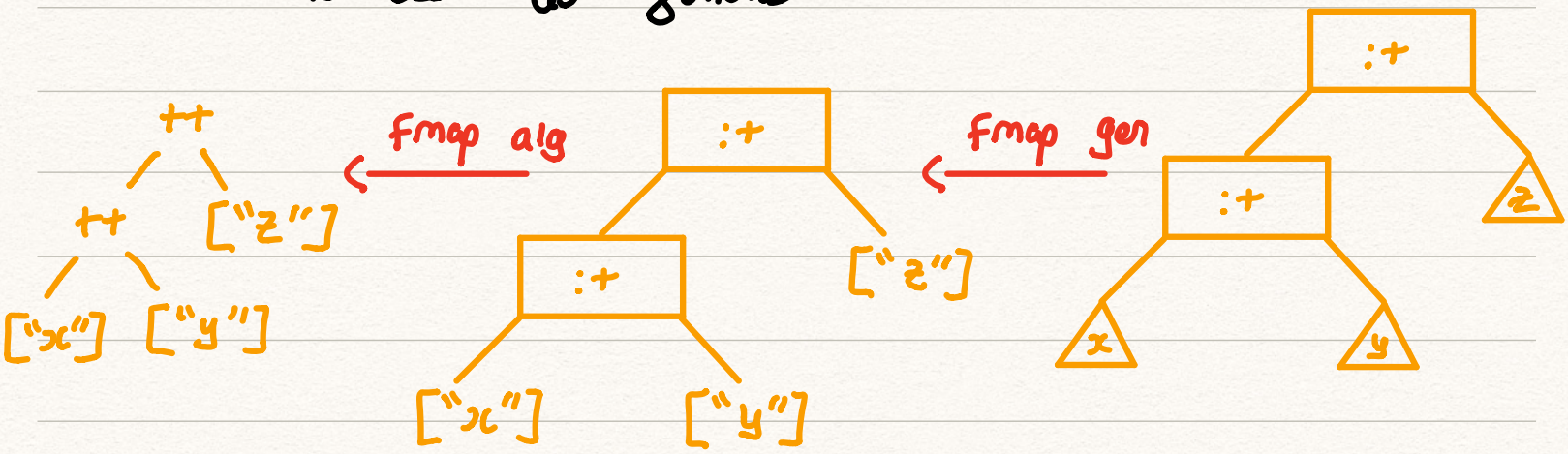
- A second example is to collect all the variables in an expression as a list. For instance in x + y + z, we see ["x", "y", "z"]

- To do this we provide a function `vars', which is defined using '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 Add and Div 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 wher
>   gen :: Var -> Double
>   gen = env <-- this function 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!

↳ `alg (R (Div x 0)) -> Fail !`

- To fix this problem we must be upfront about the fact 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
>        alg (R (Div x y)) = mDiv
```

- We must now define mAdd and mDiv

```
> 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
```

FAILURE

- we need to create syntax for failure:

```
> Data Fail k = Fail
```

- The functor instance shows us that computations cannot follow a fail:

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

- If we deal with division alone, we have this:

```
> evalFail :: Free Div Double -> Free Fail Double
> evalFail = alg gen where
>   gen :: Double -> Free Fail Double
>   gen x = Var x
>
>   alg :: Div (Free Fail Double) -> Free Fail Double
>   alg (Div x y) = ...
```

Free Fail Double ← we can pattern match a tree

- Algebra should ∴ be:

```
> 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
```

SUBSTITION

- Substition in a langvage is a very vsegul feature. For example, consider this:

$x + 7$

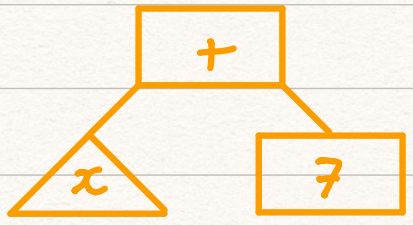
- We can evalvate this into a new syntak tree when we have a notion of substitution, where we might bind 'x' to another expression rather than just a constant:

e.g. $x \mapsto 4 + 5$

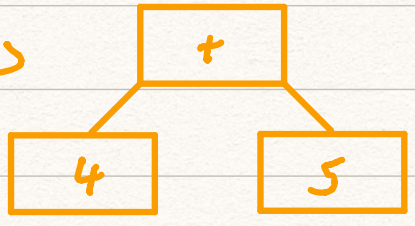
- Then we expect the above to become:

$(4 + 5) + 7$

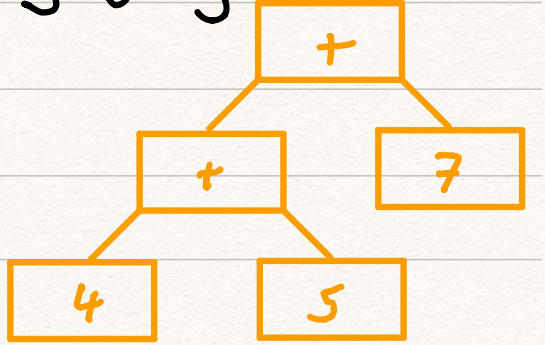
- We can depict this by the following trees:



$x \mapsto$



- Substitution is the act of grafting the tree on the right into the left



- We will define substitution using code.

- Usually an expression e with a variable x is substituted using e' with the following syntax:

e [x ↦ e']

this corresponds to x+7 — this is 4+5

- Sometimes we also write e [x \ 4+5]
  or e [4+5 / x]

- For our example, a syntax tree is given by a datatype `Free f a`, where `f` is the shape of the syntax, and `a` substition is defined by (>>=) as follows:

Syntax tree — substition function

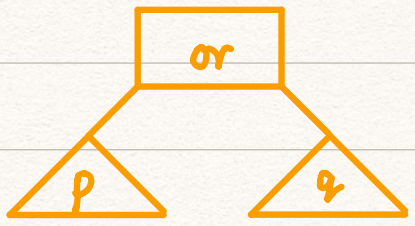
```
> (>>=) :: Free f a -> (a -> Free f b) -> Free f b
> Var x >>= f = f x
> Op op >>= f = Op (fmap (>>= f) op)
```

op : f (Free f a) — f : a -> Free f a — fmap (>>= f) : Free f a -> Free f b — op : f (Free f a) — Op (...) : f (Free f b)

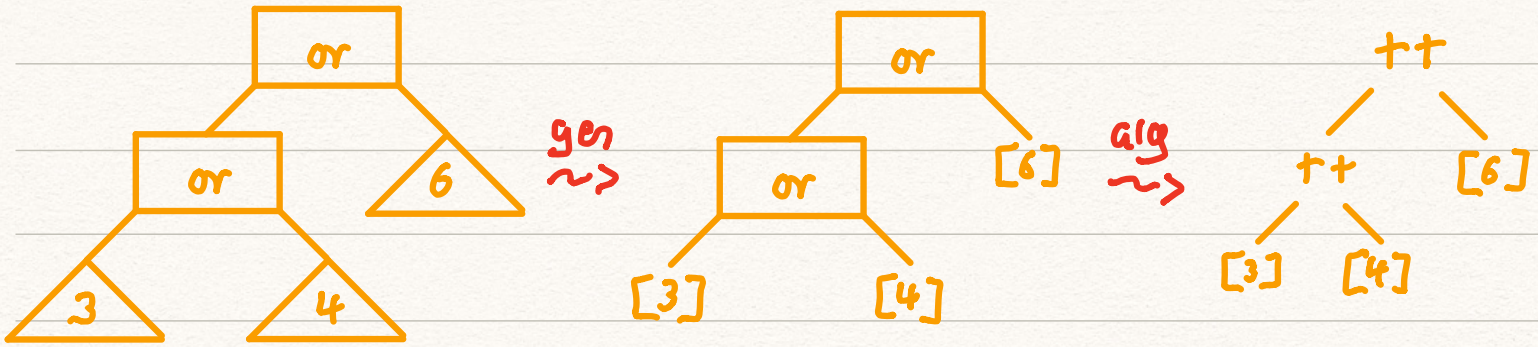
NON-DETERMINISM

- A non-deterministic computation is one that provides the choice between two different computations

- For example, `p □ q` is the program that answers `p or q`



- Here we use `or` to represent `□`



- In terms of code we first need to express syntax:

```
> data Or k = Or k k
```

- we must ensure that this is a Functor:

```
> instance Functor Or where
>     fmap f (Or x y) = Or (fx) (Fy)
```

- with this in place we can define an evaluation function:

```
> 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 enterpretation of these trees is to simply return the first result:

- we can define using 'once':

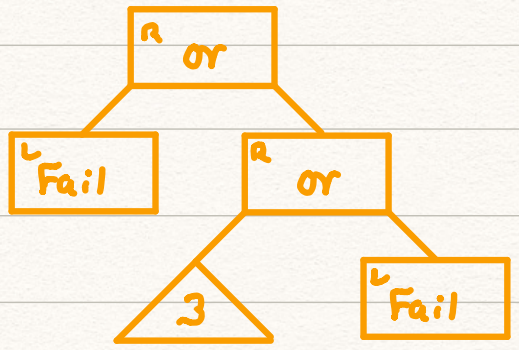
```
> Once :: Free Or a -> Maybe a
> Once = eval alg gen where
>     gen :: a -> Maybe a
>     gen x = Just x
>
>     alg :: Or (Maybe a) -> Maybe a
>     alg Or Nothing y = y
>     alg Or Just x y = Just x
```

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

- The semantics for once is similar

#ALTENATION

- An alternative way to do 'Or' is to model a pair of R values as a function from `Bool`. To do this we define the following:

```
> data Alt R = Alt (Bool R)
```

- The idea is that we pass `True' when we want the first child and false for the second child. We must show that this is a functor:

```
> instance Functor Alt where
>   fmap f (Alt k) = Alt (f . k)
```

f : a->b, k : Bool->a, (f . k) : Bool->b

- Now we can give different semantics for nondeterminism:

```
> type Nondet' a = (Fail :+: Alt) a
```

- For `list' we do this:

```
> 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

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 valve `s' into the state before continuing with the computation in `k'

- The `Get k' 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 the 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 = (x, s)
> -- another way is  gen x = λs -> (x, s)
```

- alg is as follows:

```
> alg :: State s (s -> (a,s)) -> (s -> (a,s))
> alg (Put s' k) = λs -> k s'
```

`k` : s -> (a,s)

λs -> k s' : s -> (a,s)

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

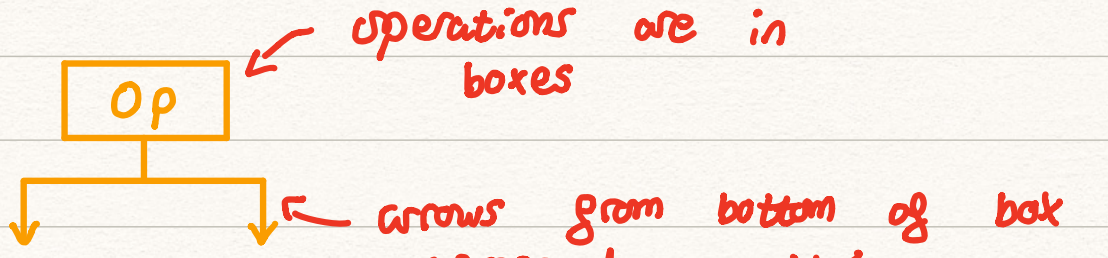
```
> alg (Get k) = λs -> k s s
```

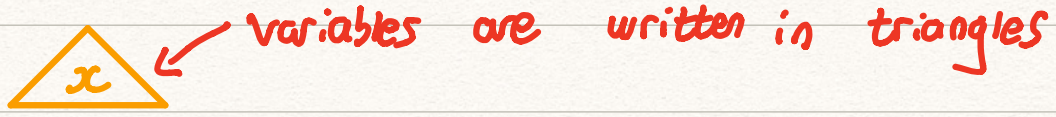
`k` : s -> s -> (a,s)

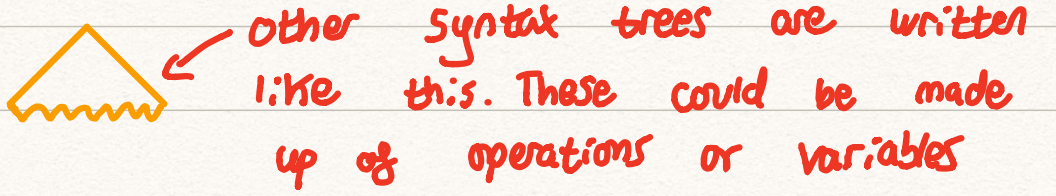
- In the term `k s s`
  - first `s`: the state that generates program
  - second `s`: state passed on to future programs

DIAGRAMS OF OPERATIONS

- We have already seen some diagrams. Here are some conventions

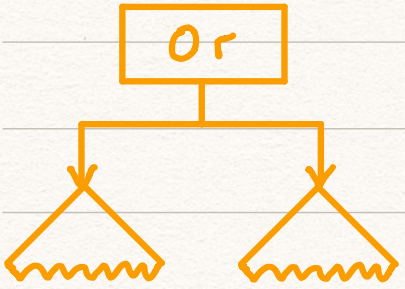


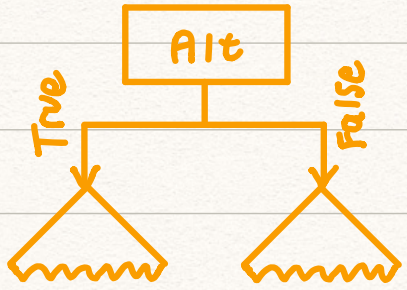




- To represent Nondeterminism and alternation we have these diagrams:

```
> data Or k = Or k k
> data Alt k = Alt (Bool -> k)
```





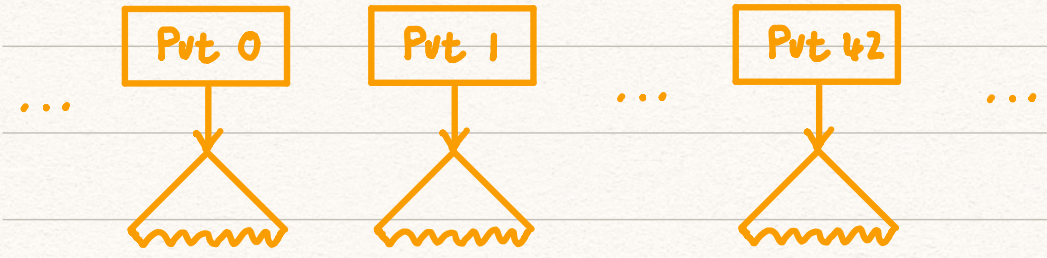
- We can imagine that `'State S = Put S :+: Get S'`

```
> Data Put s k = Put s k
> Data Get s k = Get (s -> k)
```

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

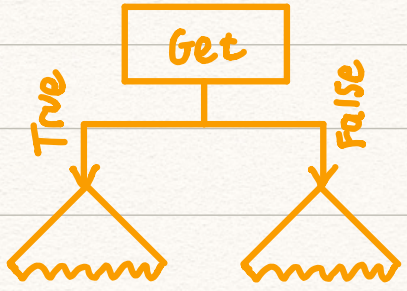
- These examples are of `Free (Put Bool) a' syntax trees: since `S = Bool', we can only construct these two Put nodes

- we can parametise S differently, so if we had `S = Int', then we have a huge number of nodes

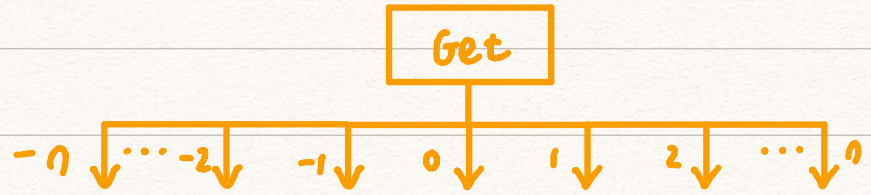


- The 'Get' nodes are a generalisation of 'Alt'

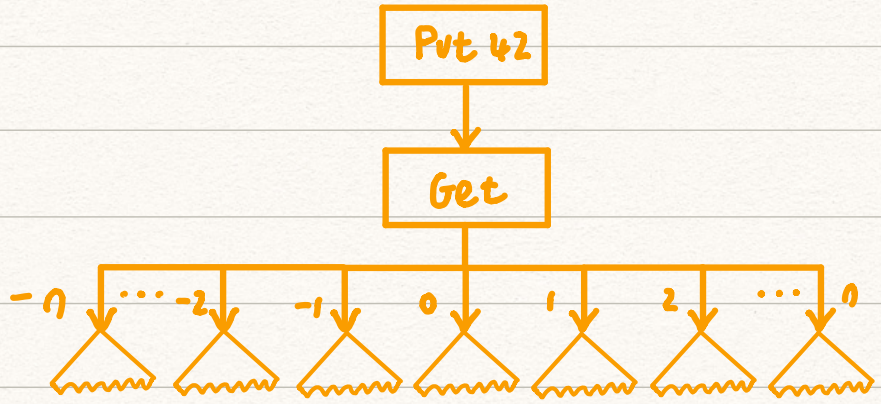
- If we consider trees of the form `Free (Get Bool) a' then this follows:



- With `S = Int', we have `Free (Get Int) a':

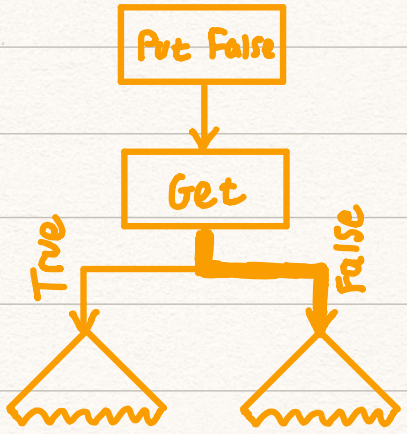


- Note that `Alt' and `Get Bool' are not syntactically very similar: they share the same structure. Their differences are exhibited when we provide different algebras



- 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 to `S' as input

consider S = Bool :



Note that here we used 'False' to choose which child to use, and we also passed False to the next stage

alg (Get k) = λs -> k s s

selecting child

represented by children

passing s onwards

- The 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 an `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 ()))
>
```

`Var ()` : `Free (State s) ()`

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

`λs -> Var s` : `s -> Free (State s)`

- With these tools, we can simply substitute:

```
put False >>= get
```

- This produces the tree as above