Minimal Haskell syntax

Module organisation

```
{-# LANGUAGE Blah #-}
-- name must be Foo/Bar.hs
module Foo.Bar where
-- import everything in scope
import Buzz
-- must prefix functions with Buzz.
import qualified Buzz
-- must prefix functions with B.
import qualified Buzz as B
-- only import foo in scope
import Buzz(foo)
```

Types

```
foo :: a -> b -> c
```

- function `foo` takes two arguments, of type `a` and `b`.
- its return type is `c`
- you can't write this function ;)

Concrete and variable types

```
foo :: a -> Item -> Foo
```

- starts with uppercase : concrete type
- starts with lowercase : type variable

Defining types

```
data TypeName = Constructor1
              | Constructor2
```

Type and constructor names start with an uppercase letter. Constructors are functions.

Product types

```
data Triplet = Triplet Int Char Bool
```

The constructor holds all three data types.

Sum types

```
data Beverage = Water | Beer
```

The type is inhabited by *distinct* values. A bit like a tagged union in C.

Mixing it all

```
data ETA
  = StartDelayed Time
  | ArrivalIn Time
  | Arrived
```

Deriving

```
data Foo = ...
  deriving (Show, Eq)
```

Just do it for now :)

Built-ins

- `Bool`, is not an int!
- `Int`, like in C, architecture dependant
- `Integer`, GMP-backed bigints
- `Char`, unicode character
- `[a]`, linked list of as
- `String`, `[Char]`
- `Maybe x`, optional value
- `Either l r`, usually a result type
- `Text` (from `Data.Text`)
- `Map` (from `Data.Map.Strict`)
- `()`, *unit*, a type with a single inhabitant

Functions

```
f :: a -> b -> (a,b)
f pa pb = (pa, pb)
```

Equivalent to the following Python:

```
fun f(pa,pb):
  return (pa,pb)
```

where and let

```
f a b c d =
  let s1 = a + b
      s2 = c + d
  in  s1 + s2

f a b c d = s1 + s2
  where s1 = a + b
        s2 = c + d
```

Working with ADTs

```
data Foo a = A a | B Int
```

Building

```
a :: a -> Foo
a = A
b :: Int -> Foo
b = B
```

Case

```
ga :: Foo a -> Maybe a
ga x = case x of
  A v -> Just v
  B _ -> Nothing
```

Fonction definition

```
ga :: Foo a -> Maybe a
ga (A v) = Just v
ga _ = Nothing
```

Misc

if/then/else

Le mot clef 'if' est une expression, les deux branches (`then` et `else`) doivent être présentes et être du même type.

```
let k = if x == y then 3 else 6
```

ghci

- `:l FILENAME`, charger un fichier
- `:r`, recharger un fichier (et oublie toutes les variables!)
- `:i NOM`, donne des informations (instances, déclaration, etc.)
- `:t EXPR`, donne le type d'une expression