1 Basic Concepts

1.1 Local Binder in Haskell

```
let v = e1 in e2
```

Scope of `v` is in both `e1` and `e2`

2 Data & Constructs

2.1 Ordinal Types in Haskell

```
data DaysObj = Mon | Tue | Wed | Thu | Fri | Sat | Sun
↪  deriving (Show, Enum, Eq)
data Data = I Int | F Float | S String deriving Show
```

Enumeration is a special case of ADT

```
weekend x = case x of
  Sat -> True
  Sun -> True
  _   -> False
```

Pattern matching

2.2 Type Synonym

```
type Student = (String, String, Int)
type Pair a b = (a, b)
type String = [Char]
```

Type synonyms cannot be recursive.

2.3 Record

```
data Student = Student { name :: String; matrix ::
↪  String; year :: Int }
```

Automatically derive name, matrix, year as access methods

```
name :: Student -> String
matrix :: Student -> String
year :: Student -> Int
```

Record pattern matching

```
f :: Student -> Int
f Student { year = y } = y
```

2.4 Product vs Sum Types

- Product types include tuples and records (similar to conjunction), e.g. tuple
- Sum types include ordinals and General ADT (similar to disjunction), e.g. `Either`

2.5 Fibonacci

```
fib2 n =
  let
    aux n = if n <= 0 then (1, 0) else let (a, b) = aux
    ↪  (n - 1) in (a + b, a)
  in fst (aux n)

fib = 1 : 1 : [a+b | (a,b) <- zip fib (tail fib)]
```

3 Higher Order Function

3.1 Strict Evaluation

```
data RealFloat a => Complex a = !a :+ !a
```

In Haskell, if need be, can mark with `!`

3.2 Composition

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

4 Other Haskell Features

4.1 List Comprehension

```
[f x | x <- xs] = map (\x -> f x) xs
[f x | x <- xs, x>5] = map (\x -> f x) (filter (\x -> x
↪  > 5) xs)
[(x,y) | x <- xs, y <- ys] = concatMap (\x -> map (\y ->
↪  (x, y)) ys) xs
```

5 Monad

```
foo1 = getLine >>= readFile >>= putStrLn

foo2 = do
  filename <- getLine
  contents <- readFile filename
  putStrLn contents
```

`foo1` and `foo2` are equivalent

5.1 Laws of Monad

```
(return a) >>= k = k a
m >>= return = m
(m >>= (\a -> (k a) >>= (\b -> h b)) = (m >>= (\a -> k
↪  a) >>= (\b -> h b)
```

6 References

Note: `Integer` is arbitrary precision, `Int` is at least 30 bits.

6.1 Enum

```
succ :: Enum a => a -> a
pred :: Enum a => a -> a
toEnum :: Enum a => Int -> a
fromEnum :: Enum a => a -> Int
enumFrom :: Enum a => a -> [a]
enumFromThen :: Enum a => a -> a -> [a]
enumFromTo :: Enum a => a -> a -> [a]
enumFromThenTo :: Enum a => a -> a -> a -> [a]
```

6.2 Show

```
showsPrec :: Show a => Int -> a -> ShowS
show :: Show a => a -> String
showList :: Show a => [a] -> ShowS
```

6.3 Eq

```
(==) :: Eq a => a -> a -> Bool
(/=) :: Eq a => a -> a -> Bool
```

6.4 Ord

```
data Ordering = LT | EQ | GT
compare :: Ord a => a -> a -> Ordering
(<) :: Ord a => a -> a -> Bool
(<=) :: Ord a => a -> a -> Bool
(>) :: Ord a => a -> a -> Bool
(>=) :: Ord a => a -> a -> Bool
max :: Ord a => a -> a -> a
min :: Ord a => a -> a -> a
```

6.5 Num

```
(+) :: Num a => a -> a -> a
(-) :: Num a => a -> a -> a
(*) :: Num a => a -> a -> a
negate :: Num a => a -> a
abs :: Num a => a -> a
signum :: Num a => a -> a
fromInteger :: Num a => Integer -> a
```

6.6 Integral

```
class (Real a, Enum a) => Integral a where
  quot :: a -> a -> a
  rem :: a -> a -> a
  div :: a -> a -> a
  mod :: a -> a -> a
  quotRem :: a -> a -> (a, a)
  divMod :: a -> a -> (a, a)
  toInteger :: a -> Integer
```

6.7 Fractional

```
class Num a => Fractional a where
  (/) :: a -> a -> a
  recip :: a -> a
  fromRational :: Rational -> a
```

6.8 Floating

```
class Fractional a => Floating a where
  pi :: a
  exp :: a -> a
  log :: a -> a
  sqrt :: a -> a
  (**) :: a -> a -> a
  logBase :: a -> a -> a
  sin :: a -> a
  cos :: a -> a
  tan :: a -> a
  asin :: a -> a
  acos :: a -> a
  atan :: a -> a
  sinh :: a -> a
  cosh :: a -> a
  tanh :: a -> a
  asinh :: a -> a
  acosh :: a -> a
  atanh :: a -> a
  GHC.Float.log1p :: a -> a
  GHC.Float.expm1 :: a -> a
  GHC.Float.log1pexp :: a -> a
  GHC.Float.log1mexp :: a -> a
```

6.9 Foldable

```
all :: Foldable t => (a -> Bool) -> t a -> Bool
any :: Foldable t => (a -> Bool) -> t a -> Bool
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
length :: Foldable t => t a -> Int
sum :: (Foldable t, Num a) => t a -> a
product :: (Foldable t, Num a) => t a -> a
null :: Foldable t => t a -> Bool
concatMap :: Foldable t => (a -> [b]) -> t a -> [b]
```

Note:

```
foldr (-) 0 [1,2,3,4] = (1 - (2 - (3 - (4 - 0)))) = -2
foldl (-) 0 [1,2,3,4] = ((((0 - 1) - 2) - 3) - 4) = -10
```

`foldl` is tail-recursive

6.10 Functor

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

6.11 Applicative

```
pure :: Applicative f => a -> f a
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
(*>) :: Applicative f => f a -> f b -> f b
(<*) :: Applicative f => f a -> f b -> f a
GHC.Base.liftA2 :: Applicative f => (a -> b -> c) -> f a
↪  -> f b -> f c
```

6.12 Monad

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
(>>) :: Monad m => m a -> m b -> m b
return :: Monad m => a -> m a
fail :: Monad m => String -> m a
```

6.13 IO

```
getChar :: IO Char
putChar :: Char -> IO ()
getLine :: IO String
putStrLn :: String -> IO ()
putStr :: String -> IO ()
```

6.14 GHC.List

```
head :: [a] -> a
tail :: [a] -> [a]
last :: [a] -> a
reverse :: [a] -> [a]
filter :: (a -> Bool) -> [a] -> [a]
map :: (a -> b) -> [a] -> [b]
takeWhile :: (a -> Bool) -> [a] -> [a]
dropWhile :: (a -> Bool) -> [a] -> [a]
take :: Int -> [a] -> [a]
drop :: Int -> [a] -> [a]
zip :: [a] -> [b] -> [(a, b)]
(:) :: a -> [a] -> [a]
(++) :: [a] -> [a] -> [a]
```

6.15 Array

```
class Ord a => Ix a

range :: Ix a => (a, a) -> [a]
index :: Ix a => (a, a) -> a -> Int
GHC.Arr.unsafeIndex :: Ix a => (a, a) -> a -> Int
inRange :: Ix a => (a, a) -> a -> Bool
rangeSize :: Ix a => (a, a) -> Int
GHC.Arr.unsafeRangeSize :: Ix a => (a, a) -> Int

array :: Ix i => (i, i) -> [(i, e)] -> Array i e
```

6.16 Semigroup

```
class Semigroup a where
  (<>) :: a -> a -> a
  GHC.Base.sconcat :: GHC.Base.NonEmpty a -> a
  GHC.Base.stimes :: Integral b => b -> a -> a
```

6.17 Monoid

```
class Semigroup a => Monoid a where
  mempty :: a
  mappend :: a -> a -> a
  mconcat :: [a] -> a
```

6.18 Misc

```haskell
error :: [Char] -> a
data Maybe a = Nothing | Just a
```