1 Debug.Trace

```
trace          :: String -> a -> a
traceId        :: String -> String
trace          :: String -> a -> a
traceId        :: String -> String
traceShow      :: Show a => a -> b -> b
traceShowId    :: Show a => a -> a
traceStack     :: String -> a -> a
traceIO        :: String -> IO ()
traceM         :: Applicative f => String -> f ()
traceEventIO   :: String -> IO ()
traceEvent     :: String -> a -> a
traceMarker    :: String -> a -> a
traceMarkerIO  :: String -> IO ()
```

2 System.IO

```
stdin  :: Handle
stdout :: Handle
stderr :: Handle

withFile :: FilePath -> IOMode ->
            (Handle -> IO r) -> IO r
openFile :: FilePath -> IOMode -> IO Handle
hClose   :: Handle -> IO ()
hFlush   :: Handle -> IO ()
```

3 Control.Exception

`data SomeException` : root of the exception type hierarchy

```
class (Typeable e, Show e) => Exception e where
  toException      :: e -> SomeException
  fromException    :: SomeException -> Maybe e
  displayException :: e -> String

throw   :: e -> a
throwIO :: e -> IO a

catch   :: IO a -> (e -> IO a) -> IO a
catches :: IO a -> [Handler a] -> IO a
handle  :: (e -> IO a) -> IO a -> IO a
try     :: IO a -> IO (Either e a)
bracket :: IO a -> (a -> IO b) ->
           (a -> IO c) -> IO c
```

`e :: Exception`

4 Text

```
Data.Text
Data.Text.Lazy
Data.ByteString
Data.ByteString.Lazy
```

Common

```
pack      :: String -> Text
unpack    :: Text -> String
singleton :: Char -> Text
empty     :: Text

cons      :: Char -> Text -> Text
snoc      :: Text -> Char -> Text
append    :: Text -> Text -> Text
head      :: Text -> Char
last      :: Text -> Char
tail      :: Text -> Text
init      :: Text -> Text
null      :: Text -> Bool
length    :: Text -> Int

map       :: (Char -> Char) -> Text -> Text
foldl     :: (a -> Char -> a) -> a -> Text -> a
foldr     :: (Char -> a -> a) -> a -> Text -> a
replace   :: Text -> Text -> Text -> Text

toLower   :: Text -> Text
toUpper   :: Text -> Text
toTitle   :: Text -> Text

concat    :: [Text] -> Text
any       :: (Char -> Bool) -> Text -> Bool
all       :: (Char -> Bool) -> Text -> Bool

replicate :: Int -> Text -> Text
take      :: Int -> Text -> Text
takeEnd   :: Int -> Text -> Text
drop      :: Int -> Text -> Text
dropEnd   :: Int -> Text -> Text
strip     :: Text -> Text

splitAt   :: Int -> Text -> (Text, Text)
splitOn   :: Text -> Text -> [Text]
split     :: (Char -> Bool) -> Text -> [Text]

isPrefixOf :: Text -> Text -> Bool
isSuffixOf :: Text -> Text -> Bool
isInfixOf  :: Text -> Text -> Bool

filter    :: (Char -> Bool) -> Text -> Text
find      :: (Char -> Bool) -> Text -> Maybe Char
index     :: Text -> Int -> Char
findIndex :: (Char -> Bool) -> Text -> Maybe Int
```

Lazy

```
toStrict   :: Text -> Text
fromStrict :: Text -> Text
```

Data.Text[.Lazy].IO, Data.ByteString.[Lazy]

```
readFile     :: FilePath -> IO Text
writeFile    :: FilePath -> Text -> IO ()
appendFile   :: FilePath -> Text -> IO ()

hGetContents :: Handle -> IO Text
hGetLine     :: Handle -> IO Text
hPutStr      :: Handle -> Text -> IO ()
hPutStrLn    :: Handle -> Text -> IO ()

getContents  :: IO Text
getLine      :: IO Text

putStr       :: Text -> IO ()
putStrLn     :: Text -> IO ()
```

5 Function

```
id    :: a -> a
const :: a -> b -> a
(.)   :: (b -> c) -> (a -> b) -> a -> c
flip  :: (a -> b -> c) -> b -> a -> c
($)   :: (a -> b) -> a -> b
(&)   :: a -> (a -> b) -> b
fix   :: (a -> a) -> a
on    :: (b -> b -> c) -> (a -> b) -> a -> a -> c
```

6 Tuple

```
fst     :: (a, b) -> a
snd     :: (a, b) -> b
curry   :: ((a, b) -> c) -> a -> b -> c
uncurry :: (a -> b -> c) -> (a, b) -> c
swap    :: (a, b) -> (b, a)
```

7 Monoid

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

(<>) :: Semigroup a => a -> a -> a
```

8 Foldable

```
class Foldable t where
  foldMap :: Monoid m => (a -> m) -> t a -> m
  foldr   :: (a -> b -> b) -> b -> t a -> b

traverse_ :: (a -> f b) -> t a -> f ()
for_      ::  t a -> (a -> f b) -> f ()
sequenceA_:: t (f a) -> f ()

concat    :: t [a] -> [a]
and       :: t Bool -> Bool
or        :: t Bool -> Bool
any       :: (a -> Bool) -> t a -> Bool
all       :: (a -> Bool) -> t a -> Bool
```

t :: Foldable, f :: Applicative

9 Functor

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

(<$)   :: a -> f b -> f a
($>)   :: f a -> b -> f b
(<$>)  :: (a -> b) -> f a -> f b
(<&>)  :: f a -> (a -> b) -> f b
void   :: f a -> f ()
```

f :: Functor

10 Applicative

```
class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
  (*>)  :: f a -> f b -> f b
  (<*)  :: f a -> f b -> f a
  liftA2:: (a -> b -> c) -> f a -> f b -> f c

(<**>)  :: f a -> f (a -> b) -> f b

liftA   :: (a -> b) -> f a -> f b
liftA3  :: (a -> b -> c -> d) -> f a -> f b -> f c -> f d

void    :: f a -> f ()
forever :: f a -> f b
when    :: Bool -> f () -> f ()
unless  :: Bool -> f () -> f ()
```

f :: Applicative

11 Alternative

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
  some  :: f a -> f [a]
  many  :: f a -> f [a]

optional :: f a -> f (Maybe a)
guard    :: Bool -> f ()
```

f :: Alternative

12 Traversable

```
class (Functor t, Foldable t) => Traversable t where
  traverse  :: (a -> f b) -> t a -> f (t b)
  sequenceA :: t (f a) -> f (t a)
  mapM      :: (a -> m b) -> t a -> m (t b)
  sequence  :: t (m a) -> m (t a)

for :: t a -> (a -> f b) -> f (t b)
```

f :: Applicative, t :: Traversable, m :: Monad

13 Monad

```
class Applicative m => Monad m where
  (>>=)  :: forall a b. m a -> (a -> m b) -> m b
  (>>)   :: forall a b. m a -> m b -> m b
  return :: a -> m a

mapM_      :: (a -> m b) -> t a -> m ()
forM       :: t a -> (a -> m b) -> m (t b)
forM_      :: t a -> (a -> m b) -> m ()
sequence_  :: t (m a) -> m ()

(=<<)      :: (a -> m b) -> m a -> m b
(>=>)      :: (a -> m b) -> (b -> m c) -> a -> m c
(<=<)      :: (b -> m c) -> (a -> m b) -> a -> m c

join       :: m (m a) -> m a
filterM    :: (a -> m Bool) -> [a] -> m [a]
filterM    :: (a -> m Bool) -> [a] -> m [a]
foldM      :: (b -> a -> m b) -> b -> t a -> m b
foldM_     :: (b -> a -> m b) -> b -> t a -> m ()
replicateM :: Int -> m a -> m [a]
replicateM_:: Int -> m a -> m ()

liftM      :: (a1 -> r) -> m a1 -> m r
liftM2     :: (a1 -> a2 -> r) -> m a1 -> m a2 -> m r
ap         :: m (a -> b) -> m a -> m b
(<$!>)     :: (a -> b) -> m a -> m b
```

f :: Applicative, t :: Traversable, m :: Monad

14 MonadPlus

```
class (Alternative m, Monad m) => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a

msum    :: (Foldable t, MonadPlus m) => t (m a) -> m a
mfilter :: MonadPlus m => (a -> Bool) -> m a -> m a
```

f :: Applicative, t :: Foldable, m :: Monad

15 MonadIO

Control.Monad.IO.Class

```
class Monad m => MonadIO m where
  liftIO :: IO a -> m a
```

16 Category

```
class Category cat where
  id  :: cat a a
  (.) :: cat b c -> cat a b -> cat a c

(<<<) :: Category cat => cat b c -> cat a b -> cat a c
(>>>) :: Category cat => cat a b -> cat b c -> cat a c
```

17 Arrow

```
class Category a => Arrow a where
  arr     :: (b -> c) -> a b c
  first   :: a b c -> a (b, d) (c, d)
  second  :: a b c -> a (d, b) (d, c)
  (***)   :: a b c -> a b' c' -> a (b, b') (c, c')
  (&&&)   :: a b c -> a b c' -> a b (c, c')

newtype Kleisli m a b = Kleisli { runKleisli :: a -> m b }

returnA :: Arrow a => a b b

(^>>) :: Arrow a => (b -> c) -> a c d -> a b d
(>>^) :: Arrow a => a b c -> (c -> d) -> a b d
(>>>) :: Category cat => cat a b -> cat b c -> cat a c
(<<<) :: Category cat => cat b c -> cat a b -> cat a c
(<<^) :: Arrow a => a c d -> (b -> c) -> a b d
(^<<) :: Arrow a => (c -> d) -> a b c -> a b d
```

18 Maybe

```
data Maybe a = Nothing | Just a

maybe     :: b -> (a -> b) -> Maybe a -> b
isJust    :: Maybe a -> Bool
isNothing :: Maybe a -> Bool
fromJust  :: Maybe a -> a (throws error)
fromMaybe :: a -> Maybe a -> a
catMaybes :: [Maybe a] -> [a]
mapMaybe  :: (a -> Maybe b) -> [a] -> [b]
```

19 Either

```
data Either a b = Left a | Right b

either    :: (a -> c) -> (b -> c) -> Either a b -> c
lefts     :: [Either a b] -> [a]
rights    :: [Either a b] -> [b]
isLeft    :: Either a b -> Bool
isRight   :: Either a b -> Bool
fromLeft  :: a -> Either a b -> a
fromRight :: b -> Either a b -> b
```