Import

```
import Data.Set
import qualified Data.Map.Strict as M
```

Imports only the mentioned functions.

```
import Data.List (genericTake)
```

If Statement

```
doubleIfOdd :: Int -> Int
doubleIfOdd x = if mod x 2 == 1
                  then 2*x
                  else x
```

If statements are also expressions.

```
doubleIfOdd' :: Int -> Int
doubleIfOdd' x = (if mod x 2 == 1 then x else 0) + x
```

Guards

The first fulfilled condition determines the evaluated branch.

```
magicModify :: (Integral a) => a -> a
magicModify n
    | mod n 3 == 1  = 4*n
    | mod n 2 == 0  = 2*n
    | n >= 204      = n
    | otherwise     = 3*n
```

Pattern Matching

```
explain :: Maybe Int -> String
explain Nothing = "failed"
explain (Just n) = "success with " ++ (show n)
```

```
length' :: [a] -> Int
length' []     = 0
length' (x:xs) = 1 + length' xs
```

Where Binding

```
f :: (RealFloat a) => a -> a
f x = 2*(sin t + tan t)
    where t = 5*x - 4
```

```haskell
crazilyChange :: (RealFloat a) => a -> a
crazilyChange x
    | u <= a = 5*x
    | u <= b = 4*x+u
    | otherwise = x
    where u = 5*x + tan x + (a+b)/5
          a = 20
          b = 40
```

Let Binding

Unlike where bindings, let bindings are expressions.

```haskell
f' :: (RealFloat a) => a -> a
f' x = 2*(let t = 5*x - 4 in sin t + tan t)
```

```haskell
f'' :: (RealFloat a) => a -> a
f'' x = let t = 5*x - 4
            u = sin t + tan t
        in  2*u
```

Case Expression

Case expressions are for pattern matching as an expression.

```haskell
head' lst = case lst of []    -> error "List is empty."
                        (x:_) -> x
```

List Ranges

```haskell
oddsToTen = [1, 3..10]
```

List ranges can be infinite.

```haskell
odds = [1, 3..]
```

List Comprehensions

```haskell
coolNums = [i*j | i <- [1..5], j <- oddsToTen, i+j > 5]
```

Creating Types

```haskell
data YesNo = Yes | No
data BinaryTree a = Leaf a | Node a (BinaryTree a) (BinaryTree a)
```

Types with exactly one field and one constructor can be made with newtype. This creates a type with the same functionality that is distinct to the type-checker.

```
newtype OtherTree a = OtherTree (BinaryTree a)
newtype Pair a b = P (a, b)
```

Type Synonyms

Type synonyms are indistinguishable to the type-checker

```
type Pulse = Double
type Wave = [Pulse]
```