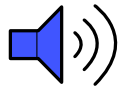
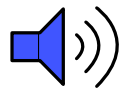
Find file with better filter



- With more file information, more complex filter may be used.

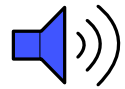
```
1  betterFind :: (FileInfo -> Bool)  -- filter predicate
2             -> FilePath            -- directory path
3             -> IO [FilePath]       -- resulting paths
4
5  betterFind p path = do
6     names <- getRecursiveContents path
7
8     -- filterM :: (a -> m Bool) -> [a] -> m [a]
9     filterM check names
10      where check name = p <$> getInfo name
11
12 -- find pdf files in the directories that match "790"
13 main = betterFind (
14           \x -> let path = filePath x
15                     ext = takeExtension x
16                 in (ext == ".pdf") && (path `match` "790")
17         )
18         "."
19             >>=
20                 (mapM_ putStrLn)  -- print the paths
```

Find file with better filter



- Define string match function

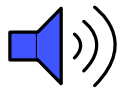
```haskell
match :: String -> String -> Bool

match s p                    -- True if s contains p

  | length s < l = False     -- False if s is shorter than p

  | otherwise = (take l s) == p || match (drop 1 s) p
                             -- True if a substring of s == p
  where l = length p
```

Find file with better filter 

- Filters can become complex quickly

```
1  -- find pdf files in the directories that match 790
2  --   or files larger than 2^28 bytes
3  --   or executable files
4  main = betterFind (
5     \x -> let path = filePath x
6               ext = takeExtension x
7               size = case fileSize x of Just y -> y
8                                         Nothing -> -1
9               exec = case filePermissions x of
10                           Just p -> executable p
11                           Nothing -> False
12          in  (ext == ".pdf") && (path `match` "790")
13              || size > 2^28
14              || exec
15    )
16    "."
17        >>=
18            (mapM_ putStrLn)  -- print the paths
```

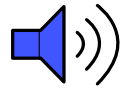
Simplify filter construction



```
1     \x -> let path = filePath x
2               ext = takeExtension x
3               size = case fileSize x of Just y -> y
4                                         Nothing -> -1
5               exec = case filePermissions x of
6                           Just p -> executable p
7                           Nothing -> False
8           in  (ext == ".pdf") && (path `match` "790")
9               || size > 2^28
10              || exec
```

- Move the file info parameter ‘x’ inside the filter.

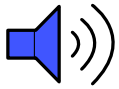
```
1  let path = filePath
2      ext  = takeExtension
3      size = \x -> case fileSize x of Just y -> y
4                                      Nothing -> -1
5      exec = \x -> case filePermissions x of
6                           Just p -> executable p
7                           Nothing -> False
8  in  \x -> (ext x == ".pdf") && (path x `match` "790")
9             || size x > 2^28
10            || exec x
```

Simplify filter construction 

```haskell
let path = filePath
    ext  = takeExtension
    size = \x -> case fileSize x of Just y -> y
                                    Nothing -> -1
    exec = \x -> case filePermissions x of
                      Just p -> executable p
                      Nothing -> False
in  \x -> (ext x == ".pdf") && (path x `match` "790")
          || size x > 2^28
          || exec x
```

- Moving ‘x’ further inside requires new operators:

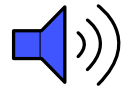
```haskell
-- FileP, ==?, >?, ~?, -- &&?, ||?

let pathP = FileP filePath
    extP  = takeExtension <$> pathP
    sizeP = FileP $ \x -> case fileSize x of
                          Just y -> y
                          Nothing -> -1
    execP = FileP $ \x -> case filePermissions x of
                          Just p -> executable p
                          Nothing -> False
in  (extP ==? ".pdf") &&? (pathP ~? "790")
    ||? (sizeP >? (2^28))
    ||? execP
```

Simplify filter construction 

- `FileP` lifts a `FileInfo` function to a `FilePredicate`. `==?`, `>?` compare a `FilePredicate` with a constant. `&&?` and `||?` are logical operators on two `FilePredicates`.

```
1  -- FileP        :: (FileInfo -> a) -> FilePredicate a
2  -- ==?, >?, ~?  :: FilePredicate a -> a -> FilePredicate a
3  -- &&?, ||?  ::
4  --   FilePredicate a -> FilePredicate a -> FilePredicate a
5
6  let pathP = FileP filePath
7      extP  = takeExtension <$> pathP
8      sizeP = FileP $ \x -> case fileSize x of
9                               Just y -> y
10                              Nothing -> -1
11     execP = FileP $ \x -> case filePermissions x of
12                              Just p -> executable p
13                              Nothing -> False
14 in  (extP ==? ".pdf") &&? (pathP ~? "790")
15     ||? (sizeP >? (2^28))
16     ||? execP
```

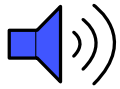
Simplify filter construction



- Improve further with ‘maybe’ function.

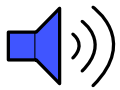
```
maybe :: b -> (a -> b) -> Maybe a -> b

let pathP = FileP filePath

    extP  = takeExtension <$> pathP

    sizeP = maybe (-1) id <$> FileP fileSize

    execP = maybe False executable <$> FileP filePermissions

in  (extP ==? ".pdf") &&? (pathP ~? "790")
    ||? (sizeP >? (2^28))
    ||? execP

-- Reusable (function) constants:
--   pathP, extP, sizeP, execP,
-- Resuable operators:
--   --? ~? ||?, >?
```

Predicate type class



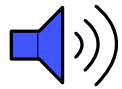
- Any Applicative can be made a Predicate with a complementary suite of operators.

```
1  class Applicative m => Predicate m where
2     (>?) :: (Ord a) => m a -> a -> m Bool  -- greater-than op
3     p >? a = (> a) <$> p
4
5     (<?) :: (Ord a) => m a -> a -> m Bool  -- less-than op
6     p <? a = (< a) <$> p
7
8     (==?) :: (Ord a) => m a -> a -> m Bool -- equal op
9     p ==? a = (== a) <$> p
10
11    (!=?) :: (Ord a) => m a -> a -> m Bool -- not equal op
12    p !=? a = (/= a) <$> p
```

Predicate type class 

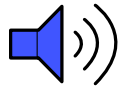
- Any Applicative can be made a Predicate with a complementary suite of operators.

```
1  class Applicative m => Predicate m where
2     (~?) :: m String -> String -> m Bool    -- contains string
3     p ~? s = (`match` s) <$> p
4
5     (&&?) :: m Bool -> m Bool -> m Bool -- logical and
6     p1 &&? p2 = (&&) <$> p1 <*> p2
7
8     (||?) :: m Bool -> m Bool -> m Bool -- logical or
9     p1 ||? p2 = (||) <$> p1 <*> p2
10
11    notP :: m Bool -> m Bool              -- logical negation
12    notP = fmap not
```

File Predicate type 

- Define a file predicate type that is an instance of Predicate.

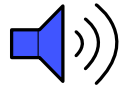
```haskell
data FileInfo = FileInfo {
                      filePath :: FilePath,
                      filePermissions :: Maybe Permissions,
                      fileSize :: Maybe Integer,
                      fileLastModified :: Maybe UTCTime
                    }

-- FilePredicate simply wraps
--   a `FileInfo -> a' function
newtype FilePredicate a = FileP {
                runFileP :: FileInfo -> a
              }
              deriving (Functor,Applicative)
              -- auto-deriving Functor and Applicative

-- do not need to implement anything
--    since Predicate is an Applicative
instance Predicate FilePredicate
```

File Predicate type 

- Making a FilePredicate value just needs FileP constructor.

```haskell
data FileInfo = FileInfo {
                      filePath :: FilePath,
                      filePermissions :: Maybe Permissions,
                      fileSize :: Maybe Integer,
                      fileLastModified :: Maybe UTCTime
                    }

newtype FilePredicate a = FileP {
                runFileP :: FileInfo -> a
              }
              deriving (Functor,Applicative)

pathP :: FilePredicate String
pathP = FileP filePath
  -- filePath :: FileInfo -> FilePath
  --     FileP :: (FileInfo -> a) -> FilePredicate a

extP :: FilePredicate String
extP = takeExtension <$> pathP -- <$> is just `fmap`

sizeP :: FilePredicate Integer
sizeP = maybe (-1) id <$> FileP fileSize
```

File Predicate type



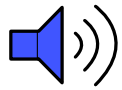
- Making a FilePredicate value just needs FileP constructor.

```haskell
data FileInfo = FileInfo {
                    filePath :: FilePath,
                    filePermissions :: Maybe Permissions,
                    fileSize :: Maybe Integer,
                    fileLastModified :: Maybe UTCTime
                }

newtype FilePredicate a = FileP {
              runFileP :: FileInfo -> a
            }
            deriving (Functor,Applicative)

getPermissionP :: (Permissions -> Bool) -> FilePredicate Bool
getPermissionP f = maybe False f <$> FileP filePermissions

readP :: FilePredicate Bool
readP = getPermissionP readable

writeP :: FilePredicate Bool
writeP = getPermissionP writable

execP :: FilePredicate Bool
execP = getPermissionP executable
```

Even better Find function

- Using FilePredicate, find function is more flexible.

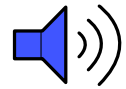
```haskell
find :: FilePredicate Bool -> FilePath -> IO [FilePath]

find predicate path = do
     names <- getRecursiveContents path    -- all path names
     filterM check names                   -- filter paths

    where check name = runFileP predicate -- run predicate
                     <$> getInfo name     -- get file info

main = do
   let downloads = "C:\\Users\\tzhao\\Downloads"
   let filterP = (extP ==? ".pdf") &&? (pathP ~? "790")
             ||? (sizeP >? (2^28))
             ||? execP

   find filterP downloads >>= mapM_ putStrLn
```

Set operator precedence 

- Operators by default has precedence level 9 and are left associative. We can lower that with `infixl` or `infixr`.

```
1  class Applicative m => Predicate m where
2     (>?) :: (Ord a) => m a -> a -> m Bool
3     p >? a = (> a) <$> p
4     infixl 4 >?
5     -- comparison operators has precedence level 4
6     -- and they are left associative
7     infixl 4 <?
8     infixl 4 ==?
9     infixl 4 !=?
10    infixl 4 ~?
11    infixl 3 &&? -- logical operators are level 3
12    infixl 3 ||? -- and they are left associative
13
14 main = do
15    let downloads = "C:\\Users\\tzhao\\Downloads"
16    let filterP = extP ==? ".pdf" &&? pathP ~? "790"
17                  ||? sizeP >? 2^28
18                  ||? execP
19
20    find filterP downloads >>= mapM_ putStrLn
```

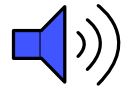
Control recursion in find function



- Add getRecursiveContents to ‘find’ with recursion predicate.

```haskell
find' :: FilePredicate Bool   -- recursion predicate
      -> FilePredicate Bool   -- filter predicate
      -> FilePath             -- start directory path
      -> IO [FilePath]        -- resulting paths

find' recurseP filterP path = do
  names <- listDirectory path

  paths <- forM names $ \name -> do
    let path' = path </> name
    info <- getInfo path'
    let isDirectory = runFileP searchP info

    if isDirectory && runFileP recurseP info
      then find' recurseP filterP path'
      else return $ if runFileP filterP info
                    then [path']
                    else []

  return $ concat paths
```

Control recursion in find function



- Add getRecursiveContents to 'find' with recursion predicate.

```
1  find' :: FilePredicate Bool   -- recursion predicate
2        -> FilePredicate Bool   -- filter predicate
3        -> FilePath             -- start directory path
4        -> IO [FilePath]        -- resulting paths
5
6
7  main = do
8     let downloads = "C:\\Users\\tzhao\\Downloads"
9
10    let recurseP = pathP ~? "790"
11
12    let filterP = (extP ==? ".pdf" &&? pathP ~? "790")
13                  ||? sizeP >? 2^28
14                  ||? execP
15
16    find' recurseP filterP downloads >>= mapM_ putStrLn
```