CS 410/510: Advanced Programming

Abstract Datatypes + Functions as Data

Mark P Jones
Portland State University

Back to Builders

Building Builders:

```
data Builder a = Builder { build::Int -> (a, [NFATrans], Int) }

newState :: Builder NFAState
newState  = Builder (\n -> (n, [], n+1))

addTrans :: NFATrans -> Builder ()
addTrans t = Builder (\n -> ((), [t], n))

returnB  :: a -> Builder a
returnB x = Builder (\n -> (x, [], n))

bindB    :: Builder a -> (a -> Builder b) -> Builder b
bindB b f = Builder (\n -> let (x, ts1, n1) = build b n
                               (y, ts2, n2) = build (f x) n1
                           in  (y, ts1++ts2, n2))

instance Monad Builder where
  return = returnB
  (>>=)  = bindB
```

These are the only operations that we will use to build Builders …

Example:

Example:

```
nfab' (C c) f  = do s <- newState
                    addTrans (Transition (Char c) s f)
                    return s
```

is syntactic sugar for:

```
nfab' (C c) f  = newState >>= \s ->
                 addTrans (Transition (Char c) s f) >>= \_ ->
                 return s
```

which, in turn, is an abbreviation for:

```
nfab' (C c) f  = newState `bindB` \s ->
                 addTrans (Transition (Char c) s f) `bindB` \_ ->
                 returnB s
```


Under the Hood:

Let's break this down:

```
nfab' (C c) f  = newState `bindB` \s ->
                 addTrans (Transition (Char c) s f) `bindB` \_ ->
                 returnB s
```

becomes:

```
nfab' (C c) f  = newState `bindB` g
 where
  g s = addTrans (Transition (Char c) s f) `bindB` h
  h _ = returnB s
```


Under the Hood:

Let's break this down:

```
nfab' (C c) f  = newState `bindB` \s ->
                 addTrans (Transition (Char c) s f) `bindB` \_ ->
                 returnB s
```

becomes:

```
nfab' (C c) f  = newState `bindB` g
 where
  g s = addTrans (t s) `bindB` h
  t s = Transition (Char c) s f
  h _ = returnB s
```

Under the Hood:

Let's break this down:

```
nfab' (C c) f  = newState `bindB` \s ->
                 addTrans (Transition (Char c) s f) `bindB` \_ ->
                 returnB s
```

becomes:

```
nfab' (C c) f  = newState `bindB` g
 where
  g s = Builder (\n -> let (x, ts1, n1) = build (addTrans (t s)) n
                           (y, ts2, n2) = build (h x) n1
                       in  (y, ts1++ts2, n2))
  t s = Transition (Char c) s f
  h _ = returnB s
```


Under the Hood:

Let's break this down:

```
nfab' (C c) f  = newState `bindB` \s ->
                 addTrans (Transition (Char c) s f) `bindB` \_ ->
                 returnB s
```

becomes:

```
nfab' (C c) f  = newState `bindB` g
 where
  g s = Builder (\n -> let (x, ts1, n1) = build (addTrans (t s)) n
                           (y, ts2, n2) = build (h x) n1
                       in  (y, ts1++ts2, n2))
  t s = Transition (Char c) s f
  h _ = Builder (\n -> (s, [], n))
```


Under the Hood:

Let's break this down:

```
nfab' (C c) f  = newState `bindB` \s ->
                 addTrans (Transition (Char c) s f) `bindB` \_ ->
                 returnB s
```

becomes:

```
nfab' (C c) f  = newState `bindB` g
 where
  g s = Builder (\n -> let (x, ts1, n1) = build (addTrans (t s)) n
                           (y, ts2, n2) = (s, [], n1)
                       in  (y, ts1++ts2, n2))
  t s = Transition (Char c) s f
  h _ = Builder (\n -> (s, [], n))
```


Under the Hood:

Let's break this down:

```
nfab' (C c) f  = newState `bindB` \s ->
                 addTrans (Transition (Char c) s f) `bindB` \_ ->
                 returnB s
```

becomes:

```
nfab' (C c) f  = newState `bindB` g
 where
  g s = Builder (\n -> let (x, ts1, n1) = build (addTrans (t s)) n

                       in  (s, ts1, n1))
  t s = Transition (Char c) s f
  h _ = Builder (\n -> (s, [], n))
```


Under the Hood:

Let's break this down:

```
nfab' (C c) f  = newState `bindB` \s ->
                 addTrans (Transition (Char c) s f) `bindB` \_ ->
                 returnB s
```

becomes:

```
nfab' (C c) f  = newState `bindB` g
 where
  g s = Builder (\n -> let (x, ts1, n1) = build (addTrans (t s)) n
                       in  (s, ts1, n1))
  t s = Transition (Char c) s f
  h _ = Builder (\n -> (s, [], n))
```


Under the Hood:

Let's break this down:

```
nfab' (C c) f  = newState `bindB` \s ->
                 addTrans (Transition (Char c) s f) `bindB` \_ ->
                 returnB s
```

becomes:

```
nfab' (C c) f  = newState `bindB` g
 where
  g s = Builder (\n -> let (x, ts1, n1) = build (addTrans (t s)) n
                       in  (s, ts1, n1))
  t s = Transition (Char c) s f
```

Under the Hood:

Let's break this down:

```
nfab' (C c) f  = newState `bindB` \s ->
                 addTrans (Transition (Char c) s f) `bindB` \_ ->
                 returnB s
```

becomes:

```
nfab' (C c) f  = newState `bindB` g
 where
  g s = Builder (\n -> let (x, ts1, n1) = ((), [t s], n)
                       in  (s, ts1, n1))
  t s = Transition (Char c) s f
```


Under the Hood:

Let's break this down:

```
nfab' (C c) f  = newState `bindB` \s ->
                 addTrans (Transition (Char c) s f) `bindB` \_ ->
                 returnB s
```

becomes:

```
nfab' (C c) f  = newState `bindB` g
 where
  g s = Builder (\n -> let (x, ts1, n1) = ((), [t s], n)
                       in  (s, [t s], n))
  t s = Transition (Char c) s f
```


Under the Hood:

Let's break this down:

```
nfab' (C c) f  = newState `bindB` \s ->
                 addTrans (Transition (Char c) s f) `bindB` \_ ->
                 returnB s
```

becomes:

```
nfab' (C c) f  = newState `bindB` g
 where
  g s = Builder (\n -> (s, [t s], n))
  t s = Transition (Char c) s f
```


Under the Hood:

Let's break this down:

```
nfab' (C c) f  = newState `bindB` \s ->
                 addTrans (Transition (Char c) s f) `bindB` \_ ->
                 returnB s
```

becomes:

```
nfab' (C c) f  = Builder (\n -> let (x,ts1,n1) = build newState n
                                    (y,ts2,n2) = build (g x) n1
                                in (y, ts1++ts2, n2))
 where
  g s = Builder (\n -> (s, [t s], n))
  t s = Transition (Char c) s f
```


Under the Hood:

Let's break this down:

```
nfab' (C c) f  = newState `bindB` \s ->
                 addTrans (Transition (Char c) s f) `bindB` \_ ->
                 returnB s
```

becomes:

```
nfab' (C c) f  = Builder (\n -> let (x,ts1,n1) = (n, [], n+1)
                                    (y,ts2,n2) = build (g x) n1
                                in (y, ts1++ts2, n2))
 where
  g s = Builder (\n -> (s, [t s], n))
  t s = Transition (Char c) s f
```


Under the Hood:

Let's break this down:

```
nfab' (C c) f  = newState `bindB` \s ->
                 addTrans (Transition (Char c) s f) `bindB` \_ ->
                 returnB s
```

becomes:

```
nfab' (C c) f  = Builder (\n -> let (x,ts1,n1) = (n, [], n+1)
                                    (y,ts2,n2) = (x, [t x], n1)
                                in (y, ts1++ts2, n2))
 where
  g s = Builder (\n -> (s, [t s], n))
  t s = Transition (Char c) s f
```

Under the Hood:

Let's break this down:

```
nfab' (C c) f  = newState `bindB` \s ->
                 addTrans (Transition (Char c) s f) `bindB` \_ ->
                 returnB s
```

becomes:

```
nfab' (C c) f  = Builder (\n -> let (x,ts1,n1) = (n, [], n+1)
                                    (y,ts2,n2) = (x, [t x], n1)
                                in (y, ts1++ts2, n2))
 where
  t s = Transition (Char c) s f
```


Under the Hood:

Let's break this down:

```
nfab' (C c) f  = newState `bindB` \s ->
                 addTrans (Transition (Char c) s f) `bindB` \_ ->
                 returnB s
```

becomes:

```
nfab' (C c) f  = Builder (\n -> let (x,ts1,n1) = (n, [], n+1)
                                    (y,ts2,n2) = (n, [t n], n+1)
                                in (n, []++ [t n], n+1))
 where
  t s = Transition (Char c) s f
```


Under the Hood:

Let's break this down:

```
nfab' (C c) f  = newState `bindB` \s ->
                 addTrans (Transition (Char c) s f) `bindB` \_ ->
                 returnB s
```

becomes:

```
nfab' (C c) f  = Builder (\n -> (n, []++ [t n], n+1))
 where
  t s = Transition (Char c) s f
```


Under the Hood:

Let's break this down:

```
nfab' (C c) f  = newState `bindB` \s ->
                 addTrans (Transition (Char c) s f) `bindB` \_ ->
                 returnB s
```

becomes:

```
nfab' (C c) f  = Builder (\n -> (n, [t n], n+1))
 where
  t s = Transition (Char c) s f
```


Under the Hood:

Let's break this down:

```
nfab' (C c) f  = newState `bindB` \s ->
                 addTrans (Transition (Char c) s f) `bindB` \_ ->
                 returnB s
```

becomes:

```
nfab' (C c) f  = Builder (\n->(n, [Transition (Char c) n f], n+1))
 where
  t s = Transition (Char c) s f
```


Under the Hood:

Let's break this down:

```
nfab' (C c) f  = newState `bindB` \s ->
                 addTrans (Transition (Char c) s f) `bindB` \_ ->
                 returnB s
```

becomes:

```
nfab' (C c) f  = Builder (\n->(n, [Transition (Char c) n f], n+1))
```

Under the Hood:

Let's break this down:

```
nfab' (C c) f  = newState `bindB` \s ->
                 addTrans (Transition (Char c) s f) `bindB` \_ ->
                 returnB s
```

becomes:

```
nfab' (C c) f  = Builder (\n->(n, [Transition (Char c) n f], n+1))
```

For example:

```
build (nfab' (C 'a') 0) 1  = (1, [Transition (Char 'a') 1 0], 2))
```


Back to Building Builders:

```
data Builder a = Builder { build::Int -> (a, [NFATrans], Int) }

newState :: Builder NFAState
newState  = Builder (\n -> (n, [], n+1))

addTrans :: NFATrans -> Builder ()
addTrans t = Builder (\n -> ((), [t], n))

returnB  :: a -> Builder a
returnB x = Builder (\n -> (x, [], n))

bindB    :: Builder a -> (a -> Builder b) -> Builder b
bindB b f = Builder (\n -> let (x, ts1, n1) = build b n
                               (y, ts2, n2) = build (f x) n1
                           in  (y, ts1++ts2, n2))

instance Monad Builder where
  return = returnB
  (>>=)  = bindB
```

These are the only operations that we will use to build Builders …

Bad Builders:

We don't want programmers to start creating arbitrary builders, because they might accidentally (or intentionally) break the invariants that we have for our Builder structures:

```
bad = Builder (\n -> (n, [epsilon n (n-1)], n-2))
```


Back to Building Builders:


```
data Builder a = Builder { build::Int -> (a, [NFATrans], Int) }

newState :: Builder NFAState
newState  = Builder (\n -> (n, [], n+1))

addTrans :: NFATrans -> Builder ()
addTrans t = Builder (\n -> ((), [t], n))

returnB  :: a -> Builder a
returnB x = Builder (\n -> (x, [], n))

bindB    :: Builder a -> (a -> Builder b) -> Builder b
bindB b f = Builder (\n -> let (x, ts1, n1) = build b n
                               (y, ts2, n2) = build (f x) n1
                           in  (y, ts1++ts2, n2))

instance Monad Builder where
  return = returnB
  (>>=)  = bindB
```


These are the only operations that we **can** use to build Builders …

Using a Haskell Module:

```
module Builder(Builder, build, newState, addTrans) where

data Builder a

build    :: Builder a -> Int -> (a, [NFATrans], Int) }

newState :: Builder NFAState

addTrans :: NFATrans -> Builder ()

instance Monad Builder where
  return = returnB
  (>>=)  = bindB
```

Inside the module: the full implementation of the Builder type is visible

Outside the module: only the names and types of the Builder type and operations are visible

Why we used data …

- Did you wonder why I'd used:
  data Builder a = Builder (Int -> (a, [NFATrans], Int))
  instead of just defining:
  type Builder a = Int -> (a, [NFATrans], Int)
  ?
- We could make the original code work just as well if we eliminated every use of the build function and the Builder constructor function
- But using a datatype makes it possible to distinguish Builder values from other functions that happen to have the same type … and makes it possible to conceal that implementation in a module

Monads:

- Monads are abstract types that represent computations
- Every monad has at least at return and a bind (>>=) operation
- If M is a monad, then a value of type M T represents:
  - A computation that returns values of type T
  - That uses the special features of monad M

The IO Monad

The IO Type:

- The type IO t represents interactive programs that produce values of type t
- The main function in every Haskell program is expected to have type IO ()
- If you write an expression of type IO t at the Hugs prompt, it will be evaluated as a program and the result discarded
- If you write an expression of some other type at the Hugs prompt, it will be turned in to an IO program using:

```
print :: (Show a) => a -> IO ()
print = putStrLn . show
```


I/O Primitives:

- putChar c is a program that prints the single character c on the console:

```
putChar :: Char -> IO ()
```

- (>>) is an infix operator that glues two IO programs together, returning the result of the second

```
(>>) :: IO a -> IO b -> IO b
```

- For example: putChar 'h' >> putChar 'i'

putStr and putStrLn:

- Now, for example, we can define:

```
putStr        :: String -> IO ()
putStr        = foldr1 (>>) . map putChar

putStrLn      :: String -> IO ()
putStrLn s    = putStr s >> putChar "\n"
```

- Alternatively

```
putStr        = mapM_ putChar
```

using the primitives

```
mapM          :: (a -> IO b) -> [a] -> IO [b]
mapM_         :: (a -> IO b) -> [a] -> IO ()
```


"do-notation":

- Syntactic sugar for writing (monadic) IO programs:

  do p_1
  p_2
  …
  p_n

  is equivalent to:
  p_1 >> p_2 >> … >> p_n

  and can also be written:
  do p_1; p_2; …; p_n or **do** { p_1; p_2; …; p_n }

return:

- We can make a program that returns x without doing any I/O using return x:

  return :: a -> IO a

- Note that return is not quite like the return we know from imperative languages:

  (**do** return 1; return 2) = return 2

Using Return Values:

- How can we use returned values?
- Another important primitive:

  (>>=) :: IO a -> (a -> IO b) -> IO b

- For example, putChar 'a' is equivalent to:

  return 'a' >>= putChar :: IO ()

- In fact, return and (>>=) satisfy laws:

  return e >>= f = f e

  p >>= return = p

Relating >>= and >>:

- (>>) can be defined as a special form of (>>=) that ignores the result of the first program:

  p >> q = p >>= (\ _ -> q)

- Special laws:

  (p>>q) >> r = p >> (q >> r)

  (p >>= f) >>= g

  = p >>= (\x -> f x >>= g)

Defining mapM and mapM_:

```
mapM_            :: (a -> IO b) -> [a] -> IO ()
mapM_ f []       = return ()
mapM_ f (x:xs)   = f x              >>
                   mapM_ f xs

mapM             :: (a->IO b) -> [a]->IO [b]
mapM f []        = return []
mapM f (x:xs)    = f x          >>= \y ->
                   mapM f xs    >>= \ys->
                   return (y:ys)
```


Extending "do-notation":

We can bind the results produced by IO programs to variables using an extended form of do-notation. For example:

do x_1 <- p_1

...

x_n <- p_n

q

all "generators" should have the same indentation

last item must be an expression

is equivalent to:

p_1 >>= \x_1 ->

...

p_n >>= \x_n ->

q

variables introduced in a generator are in scope for the rest of the expression

The "v <-" portion of a generator is optional and defaults to "_ <-" if

Defining mapM and mapM_:

```
mapM_            :: (a -> IO b) -> [a] -> IO ()
mapM_ f []       = return ()
mapM_ f (x:xs)   = do f x
                      mapM_ f xs

mapM             :: (a->IO b) -> [a]->IO [b]
mapM f []        = return []
mapM f (x:xs)    = do y  <- f x
                      ys <- mapM f xs
                      return (y:ys)
```

getChar:

- A simple primitive for reading a single character:

```
getChar :: IO Char
```

- A simple example:

```
echo :: IO a
echo = do c <- getChar
          putChar c
          echo
```


Reading a Complete Line:

```
getLine  :: IO String
getLine  = do c <- getChar
              if c=='\n'
                then return ""
                else do cs <- getLine
                        return (c:cs)
```


Alternative:

```
getLine   :: IO String
getLine   = loop []

loop      :: String -> IO String
loop cs   =  do c <- getChar
                case c of
                  '\n' -> return (reverse cs)
                  '\b' -> case cs of
                            []     -> loop cs
                            (c:cs) -> loop cs
                  c    -> loop (c:cs)
```


Simple File I/O:

- Read contents of a text file:

```
readFile :: FilePath -> IO String
```

- Write a text file:

```
writeFile :: FilePath -> String -> IO ()
```

- Example: Number lines

```
numLines inp out
  = do s <- readFile inp
       writeFile out (unlines (f (lines s)))
f = zipWith (\n s -> show n ++ s) [1..]
```


Handle-based I/O:

```
import IO
stdin, stderr, stdout :: Handle

openFile   :: FilePath -> IOMode -> IO Handle

hGetChar   :: Handle -> IO Char

hPutChar   :: Handle -> Char -> IO ()

hClose     :: Handle -> IO ()
```


References:

```
import Data.IORef

data IORef a = …

newIORef  :: a -> IO (IORef a)

readIORef :: IORef a -> IO a

writeIORef :: IORef a -> a -> IO ()
```

IO as an Abstract Type:

- IO is a primitive type constructor in Haskell with a large but limited set of operations:

```
return   :: a -> IO a
(>>=)    :: IO a -> (a -> IO b) -> IO b

putChar      :: Char -> IO ()
getChar      :: IO Char
...
```


There is No Escape from IO!

- There are plenty of ways to construct expressions of type IO t
- Once a program is "tainted" with IO, there is no way to "shake it off"
- There is no primitive of type IO t -> t that runs a program and returns its result
- Only two ways to run an IO program:
  - Setting it as your main function in GHC
  - Typing it at the prompt in Hugs/GHCi

Functions as Data

Functions as Data:

- Obviously, functions are an important tool that we use to manipulate data in functional programs
- But functions are first-class values in their own right, so they can also be used as data …

Sets as Functions:

```
type Set a        = a -> Bool
isElem            :: a -> Set a -> Bool
x `isElem` s      = s x
univ              :: Set a
univ              = \x -> True
empty             :: Set a
empty             = \x -> False
singleton         :: Eq a => a -> Set a
singleton v       = \x -> (x==v)
```


… continued:

```
(\/)      :: Set a -> Set a -> Set a
s \/ t    = \x -> s x || t x

(/\)      :: Set a -> Set a -> Set a
s /\ t    = \x -> s x && t x
```

- Stylistic detail: I write op x y = \z -> … to emphasize that op is a binary operator that returns a function as its result.
- Equivalent to: op x y z = …

Other Operations?

- Can I enumerate the elements of a Set?
  toList :: Set a -> [a]
- Can I compare sets for equality?
  setEq :: Set a -> Set a -> Bool
- Can I test for subsets?
  subset :: Set a -> Set a -> Bool

The Data Alternative:

```
data Set a = Empty
           | Univ
           | Singleton a
           | Union (Set a) (Set a)
           | Intersect (Set a) (Set a)
```

Now we can implement empty, univ, singleton, (\/) and (/\) directly in terms of these constructors: For example:

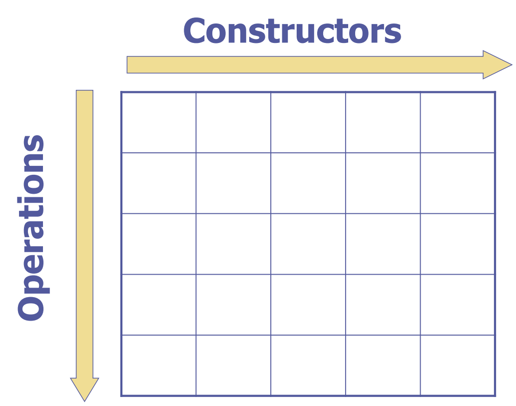
empty = Empty

Testing for Membership:

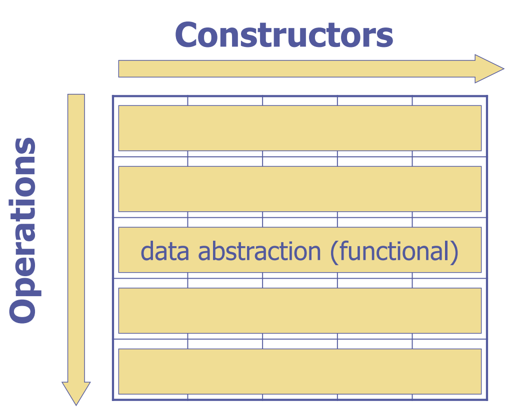
```
isElem :: Eq a => a -> Set a -> Bool
x `isElem` Empty          = False
x `isElem` Univ           = True
x `isElem` Singleton y    = (x==y)
x `isElem` Union l r      = x `isElem` l
                          || x `isElem` r
x `isElem` Intersect l r  = x `isElem` l
                          && x `isElem` r
```

Same code, different distribution …

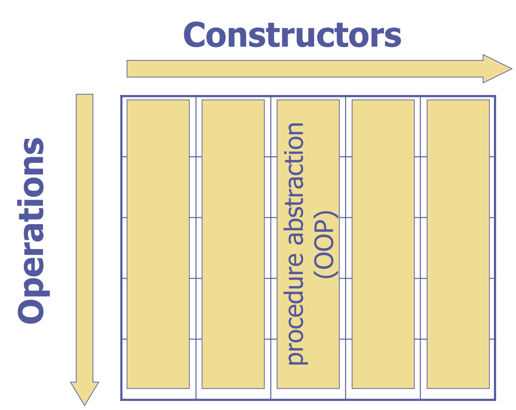
Rows and Columns:



Rows and Columns:



Rows and Columns:



... continued:

Representing sets using functions:
- "Easy" to add new constructors
- "Hard" to add new operations

Representing sets using trees:
- "Easy" to add new operations
- "Hard" to add new constructors

- Can we make it "easy" in both dimensions?
- A classic challenge for extensible software

Parser Combinators

Parsers:

```
data Parser a
        = Parser { applyP :: String -> [(a, String)]}

applyP      :: Parser a -> String -> [(a, String)]

noparse     :: Parser a
noparse     = Parser (\s -> [])

ok          :: a -> Parser a
ok x        = Parser (\s -> [(x, s)])
```


Parsers as a Monad:

```
instance Monad Parser where
  return x = ok x
  p >>= f = Parser (\s ->
                [ (y,s2) | (x,s1) <- applyP p s,
                           (y,s2) <- applyP (f x) s1 ])

(***)   :: Parser a -> (a -> b) -> Parser b
p *** f = do x <- p
             return (f x)
```


... continued:

```
item  :: Parser Char
item  = Parser (\s -> case s of
                        []     -> []
                        (t:ts) -> [(t,ts)])

sat    :: (Char -> Bool) -> Parser Char
sat p  = Parser (filter (p . fst) . applyP item)

is     :: Char -> Parser Char
is c   = sat (c==)
```


Examples:

```
digit   :: Parser Int
digit   = sat isDigit >>= \d -> ord d – ord '0'

alpha, lower, upper :: Parser Char
alpha   = sat isAlpha
lower   = sat isLower
upper   = sat isUpper

string        :: String -> Parser String
string ""     = return ""
string (c:cs) = do char c; string cs; return (c:cs)
```

Alternatives:

```
infixr 4 |||

(|||)    :: Parser a -> Parser a -> Parser a
p ||| q  = \s -> p s ++ q s

ex2      :: Parser Char
ex2      = alpha ||| ok '0'
```


Sequences:

```
infixr 6 >>>

(>>>)    :: Parser a -> Parser b -> Parser (a,b)
p >>> q = do x <- p; y <- q; return (x,y)

ex3 :: Parser (Char, Char)
ex3 = sat isDigit >>> sat (not . isDigit)
```


Repetition:

```
many      :: Parser a -> Parser [a]
many p    = many1 p ||| return []

many1     :: Parser a -> Parser [a]
many1 p   = do x <- p
               xs <- many p
               return (x:xs)
```


"Lexical Analysis":

```
number :: Parser Int
number = many1 digit
           *** foldl1 (\a x -> 10*a+x)
```


Context-Free Parsing:

Consider the following grammar:

```
expr   = term "+" expr
       | term "-" expr
       | term
term   = atom "*" term
       | atom "/" term
       | atom
atom   = "-" atom
       | "(" expr ")"
       | number
```


Context-Free Parsing:

A little refactoring:

```
expr   = term ("+" expr | "-" expr | ε)
term   = atom ("*" term | atom "/" | ε)
atom   = "-" atom
         | "(" expr ")"
         | number
```

Context-Free Parsing:

Translation into Haskell:

```
expr, term, atom :: Parser Int

expr    = term >>= \x ->
            (string "+" >> expr >>>= \y -> ok (x+y)) |||
            (string "-" >> expr >>>= \y -> ok (x-y))   |||
            ok x
```


… continued:

```
term
   = atom >>= \x ->
        (string "*" >> term >>= \y -> ok (x*y))     |||
        (string "/" >> term >>= \y -> ok (x`div`y)) |||
        ok x

atom
   = (string"-" >> atom) *** negate
       |||
     (string "(" >> expr >>= \n -> string ")" >> ok n)
       |||
     number
```


Examples:

```
Main> expr "1+2*3"
[(7,""),(3,"*3"),(1,"+2*3")]

Main> expr "(1+2)*3"
[(9,""),(3,"*3")]

Main> expr "---------1+2*----3"
[(5,""),(1,"*----3"),
  (-1,"+2*----3")]
```


Introducing a Helper:

```
Parse       :: Parser a -> String -> [a]
parse p s   = [ x | (x,"") <- applyP p s ]
```

```
Main> parse expr "1+2*3"
[7]
Main> parse expr "(1+2)*3"
[9]
Main> parse expr "---------1+2*----3"
[5]
Main>
```


Declarative Programming:

- Although it may not be immediately apparent, the structure of our program directly mimics the structure of the problem (i.e., the grammar)
- In principal, we get to express our parser at a high-level, and we don't have to worry about the details of how it is implemented
- In practice, we do (left recursion, exponential behavior, space leaks, etc..)

Constructing Abstract Syntax:

- Suppose that we define a datatype to represent arithmetic expressions:

```
data Expr = Add Expr Expr
          | Sub Expr Expr
          | Mul Expr Expr
          | Div Expr Expr
          | Neg Expr
          | Num Int
            deriving Show
```

- How can I construct an Expr from an input string?

… continued:

```
absyn :: Parser Char Expr
absyn  = expr
 where
  expr    = term >>= \x ->
              (string "+" >> expr >>= \y -> ok (Add x y)) |||
              (string "-" >> expr >>= \y -> ok (Sub x y)) |||
              ok x

  term    = atom >>= \x ->
              (string "*" >> term >>= \y -> ok (Mul x y)) |||
              (string "/" >> term >>= \y -> ok (Div x y)) |||
              ok x

  atom    =  (string "-" >> atom *** Neg)
            |||
             (string "(" >> expr >>= \n -> string ")" >> ok n)
            |||
             (number *** Num)
```


Examples:

Main> parse absyn "1+2*3"
[Add (Num 1) (Mul (Num 2) (Num 3))]

Main> parse absyn "------1"
[Neg (Neg (Neg (Neg (Neg (Neg (Num 1))))))]

Main> parse expr "------1"
[1]

Main>

Context-Sensitive Parsing:

We can easily go beyond context-free parsing in this framework:

```
brack   :: Parser String
brack   = do c <- char
             xs <- many (sat (c/=))
             sat (c==)
             return xs
```


Summary:

- Powerful ideas!
- Abstract types
- Monads as abstract types for computations
- Using functions as data
- Parser combinators