

Summarizing Patterns

Where patterns appear:

- **case** expressions (LHS of `->`)


```
case e of
  [] -> 0
  [_] -> 1
  _ -> 2
```
- function-definition parameters (LHS of `=`)


```
take 0 _ = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs
```
- list-comprehension generators (LHS of `<-`)


```
[ x | (x,True) <- exp ]
```
- left sides of definitions (LHS of `=`)


```
(d,r) = quotAndRem n 12
```

 (this is an example of an *irrefutable* pattern— if it compiles, it can't fail to match at run time)

What patterns are used for:

- to select an alternative based on a value's structural properties (in **case** expressions, multi-equation function definitions)
- to filter list elements (in list-comprehension generators)
- to bind names to (parts of) values

Kinds of patterns

- a data constructor applied to its full complement of arguments (which are themselves patterns)


```
(r,""):[]
```
- a literal (these are technically 0-ary constructors)


```
f 0 = ...
g '$' = ...
```
- a name —matches any value, which it binds to the name


```
f n = ... n ...
```

```
case n of
  [n] -> ... n ...
```
- `_` (underscore) is the *wild-card* pattern — it matches any value (and since it's not a name, it creates no name bindings)
- an “as-pattern” —useful for naming a value that matches a pattern


```
dropWhile p ys@(x:xs)
  | p x = dropWhile p xs
  | otherwise = ys
```

This example is equivalent to

```
dropWhile p ys
  | p x = dropWhile p xs
  | otherwise = ys
where
  x:xs = ys
```

For a full account, see *Gentle Introduction*, Chapter 4.

Prime Numbers: the Sieve of Eratosthenes

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
 35 36 37...
3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35
 37 ...
5 7 11 13 17 19 23 25 29 31 35 37 ...
```

```
> primes :: [Int]
> primes = sieve [2..]
> sieve (p:ns) = p : sieve [ n | n <- ns, n `mod` p > 0 ]
```

```
Main> take 10 (drop 100 primes)
[547,557,563,569,571,577,587,593,599,601]
```

The higher-order functions `map` and `filter` (previously defined using list comprehensions):

```
> map :: (a->b) -> [a] -> [b]
> map f [] = []
> map f (x:xs) = f x : map f xs

> filter :: (a->Bool) -> [a] -> [a]
> filter p [] = []
> filter p (x:xs)
>   | p x = x : filter p xs
>   | otherwise = filter p xs
```

List-folding functions

Functions that *combine* list elements are called *folding* functions. Some examples:

```
> sum, product :: Num a => [a] -> a
> sum [] = 0
> sum (n:ns) = n + (sum ns)

> product [] = 1
> product (n:ns) = n * (product ns)

> and, or :: [Bool] -> Bool
> and [] = True
> and (b:bs) = b && (and bs)

> or [] = False
> or (b:bs) = b || (or bs)
```

These functions share a common pattern, differing from one another only in respect to certain constants—

sum	product	and	or
(+)	(*)	(&&)	()
0	1	True	False

Applying the standard technique for generalizing, we replace constants with parameters.

The result is the higher-order function

```
> foldr :: (a->b->b) -> b -> [a] -> b
> foldr f k [] = k
> foldr f k (x:xs) = f x (foldr f k xs)
```

Using `foldr` we can redefine the functions we generalized, by supplying the constants as arguments:

```
> sum xs = foldr (+) 0 xs
> product xs = foldr (*) 1 xs
> and xs = foldr (&&) True xs
> or xs = foldr (||) False xs
```

These definitions can be expressed even more concisely:

```
> sum = foldr (+) 0
> product = foldr (*) 1
> and = foldr (&&) True
> or = foldr (||) False
```

because

$$(\forall x: f\ x = g\ x) \equiv f = g$$

i.e., if two functions return equal results for every argument, we say that the functions are equal.

More definitions using `foldr`:

```
> length = foldr (\_ n -> 1+n) 0
> xs ++ ys = foldr (:) ys xs
> reverse = foldr (\x xs -> xs ++ [x]) []
> map f = foldr (\x xs -> f x : xs) []
> filter p = foldr (\x xs -> if p x then x:xs else xs) []
```

Calculating using `foldr`:

```
sum [3,4,2]
= sum (3:(4:(2: [])))
~> foldr (+) 0 (3:(4:(2: [])))
~> (+) 3 (foldr (+) 0 (4:(2: [])))
= 3 + (foldr (+) 0 (4:(2: [])))
~> 3 + ((+) 4 (foldr (+) 0 (2: [])))
= 3 + (4 + (foldr (+) 0 (2: [])))
~> 3 + (4 + ((+) 2 (foldr (+) 0 [])))
= 3 + (4 + (2 + (foldr (+) 0 [])))
~> 3 + (4 + (2 + 0))
```

Comparing `foldr`'s list argument with its result suggests another way to think about `foldr`:

```
3 : ( 4 : ( 2 : [ ] ) )
3 + ( 4 + ( 2 + 0 ) )
```

we observe that evaluating

```
foldr ⊕ k xs ...
```

- replaces each `(:)` in `xs` with `⊕`
- replaces the `[]` of `xs` with `k`.

This observation can help one identify uses of `foldr`.

A related function which folds to the *left*:

```
> foldl :: (a->b->a) -> a -> [b] -> a
> foldl f k [] = k
> foldl f k (x:xs) = foldl f (f k x) xs
```

Using `foldl` to convert digit-Strings to Ints:

```
> ds2Int :: String -> Int
> ds2Int = foldl f 0
>   where
>     f n d = fromEnum d - fromEnum '0' + 10*n
```

A calculation:

```
ds2Int ['3','4','6']
~> foldl f 0 ('3': '4': '6': [])
~> foldl f (f 0 '3') ('4': '6': [])
~> foldl f (f (f 0 '3') '4') ('6': [])
~> foldl f (f (f (f 0 '3') '4') '6') []
~> (f (f (f 0 '3') '4') '6')
~> (f (f (3 + 0) '4') '6')
~> (f (f 3 '4') '6')
~> (f (34) '6')
~> 6 + 340
~> 346
```

Using infinite lists

A list of all of the powers of 2:

```
> powersOfTwo = [ 2^n | n <- [0..] ]
powersOfTwo ~> [1,2,4,8,16,32,...]
```

Generalization yields a function that generates all of the powers of any number:

```
> powersOf :: Int -> [Integer]
> powersOf x = [ x^n | n <- [0..] ]
```

This function calculates each list element independently from the others. Assuming that (\wedge) is performed by repeated multiplication, that's extra work.

A faster version computes each element from the preceding one:

```
> powersOf n = pwrs
>   where
>     pwrs = 1 : [ n * p | p <- pwrs ]
```

This is a commonly occurring pattern, so we generalize it to make a higher-order function:

```
> iterate :: (a->a) -> a -> [a]
> iterate f x = x : iterate f (f x)
```

Example:

```
iterate (\x -> x+1) 1
~> 1:2:3: ... = [1,2,3,...]
```

Then we can use the general-purpose function to redefine the functions it generalizes:

```
> powersOf x = iterate (\n -> n*x) 1
> powersOfTwo = powersOf 2
```