Dasar Haskell

Bacaan tambahan:

- *Learn You a Haskell for Great Good*, bab 2
- *Real World Haskell*, bab 1 dan 2

Apa itu Haskell?

Haskell adalah bahasa pemrograman yang *lazy* dan fungsional yang diciptakan pada akhir tahun 80-an oleh komite akademis. Pada saat itu, ada banyak bahasa pemrograman fungsional berseliweran dan setiap orang punya favoritnya sendiri-sendiri sehingga mempersulit pertukaran ide. Sekelompok orang akhirnya berkumpul bersama dan mendesain bahasa baru dengan mengambil beberapa ide terbaik dari bahasa yang sudah ada (dan menambah beberapa ide baru milik mereka sendiri). Lahirlah Haskell.

Jadi, seperti apa Haskell? Haskell itu:

Fungsional

Tidak ada pengertian tepat dan baku untuk istilah "fungsional". Tapi ketika kita mengatakan bahwa Haskell adalah bahasa pemrograman fungsional, kita biasanya mengingat dua hal ini:

- Fungsi-nya *first-class*, yakni fungsi adalah nilai yang bisa digunakan layaknya nilai-nilai yang lain.
- Program Haskell lebih bermakna *mengevaluasi ekspresi* ketimbang *mengeksekusi instruksi*.

Perpaduan keduanya menghasilkan cara berpikir tentang pemrograman yang sepenuhnya berbeda. Kebanyakan waktu kita di semester ini akan dihabiskan mengeksplorasi cara berpikir ini.

Pure

Ekspresi di Haskell selalu *referentially transparent*, yakni:

- Tanpa mutasi! Semuanya (variable, struktur data…) immutable
- Ekspresi tidak memiliki "efek samping" (seperti memperbarui variabel global atau mencetak ke layar).
- Memanggil fungsi yang sama dengan argumen yang sama selalu menghasilkan output yang sama setiap waktu.

Hal ini mungkin terdengar gila. Bagaimana mungkin bisa mengerjakan sesuatu tanpa mutasi dan efek samping? Tentunya ini memerlukan perubahan cara berpikir (jika kalian terbiasa dengan paradigma pemrograman berbasis objek). Tapi setelah kalian bisa berubah, akan ada beberapa keuntungan menakjubkan:

- *Equational reasoning* dan *refactoring*. Di Haskell kita bisa "mengganti equals dengan equals", seperti yang kita pelajari di aljabar.
- *Parallelism*. Mengevaluasi ekspresi secara paralel amatlah mudah ketika mereka dijamin tidak mempengaruhi yang lain.
- Lebih sedikit sakit kepala. Sederhananya, "efek tanpa batas" dan "aksi di kejauhan" membuat program sulit di-debug, di-maintain, dan dianalisa.

Lazy

Di Haskell, ekspresi tidak akan dievaluasi sampai hasilnya benar-benar dibutuhkan. Hal ini adalah keputusan sederhana dengan konsekuensi yang merambat kemana-mana, yang akan kita eksplorasi sepanjang semester ini. Beberapa konsekuensinya antara lain:

- Mendefinisikan *control structure* baru lewat pendefinisian fungsi menjadi mudah.
- Memungkinkan definisi dan pengerjaan dengan struktur data tak hingga.
- Mengakibatkan model pemrograman yang lebih komposisional (lihat *wholemeal programming* di bawah).
- Salah satu akibat negatif utamanya adalah analisa terhadap penggunaan ruang dan waktu menjadi lebih rumit.

Statically typed

Setiap ekspresi di Haskell memiliki tipe, dan tipe-tipe tersebut semuanya diperiksa pada waktu kompilasi. Program dengan kesalahan tipe tidak akan dikompilasi, apalagi dijalankan.

Tema

Selama kuliah ini, kita akan fokus pada tiga tema utama.

Tipe

Static type system bisa terlihat mengganggu. Faktanya, di bahasa seperti C++ dan Java, mereka memang mengganggu. Tapi bukan *static type system*-nya yang mengganggu, melainkan *type system* di C++ dan Java yang kurang ekspresif! Semester ini kita akan melihat lebih dekat pada *type system* di Haskell yang:

- *Membantu mengklarifikasi pemikiran dan ekspresi struktur program*

Langkah pertama dalam menulis program Haskell biasanya adalah dengan menulis semua tipenya. Karena *type system* Haskell sangat ekspresif, langkah desain non-trivial ini akan sangat membantu dalam mengklarifikasi pemikiran seseorang tentang programnya.

- *Menjadi salah satu bentuk dokumentasi*

Dengan type system yang ekspresif, hanya dengan melihat tipe pada suatu fungsi mampu memberitahu kalian tentang apa yang mungkin dikerjakan fungsi tersebut dan bagaimana ia bisa digunakan, bahkan sebelum kalian membaca dokumentasinya satu kata pun.

- Mengubah *run-time errors* menjadi *compile-time errors*

Jauh lebih baik jika kita bisa memperbaiki kesalahan di depan daripada harus menguji sebanyak mungkin dan berharap yang terbaik. "Jika program ini berhasil di-compile, maka program tersebut pasti benar" sering dianggap candaan (karena masih mungkin untuk memiliki kesalahan di logika meskipun programnya *type-correct*), tetapi hal tersebut sering terjadi di Haskell ketimbang bahasa lain.

Abstraksi

"Don't Repeat Yourself" adalah mantra yang sering didengar di dunia pemrograman. Juga dikenal sebagai "Prinsip Abstraksi", idenya adalah tidak ada yang perlu diduplikasi: setiap ide, algoritma, dan potongan data harus muncul tepat satu kali di kode kalian. Mengambil potongan kode yang mirip dan memfaktorkan kesamaannya sering disebut sebagai proses abstraksi.

Haskell sangatlah bagus dalam abstraksi: fitur seperti *parametric polymorphism*, fungsi *higher-order*, dan *type class* semuanya membantu melawan pengulangan yang tak perlu. Perjalanan kita dalam semester ini sebagian besar akan merupakan perjalanan dari yang spesifik menuju ke yang abstrak

Wholemeal programming

Satu lagi tema yang akan kita eksplorasi ialah *wholemeal programming*. Berikut adalah sebuah kuotasi dari Ralf Hinze:

> "Bahasa pemrograman fungsional unggul di *wholemeal programming*, istilah yang diciptakan oleh Geraint Jones. *Wholemeal programming* berarti berpikir besar dan menyeluruh. Bekerja dengan seluruh list secara utuh ketimbang barisan elemen-elemennya; mengembangkan ruang solusi ketimbang solusi individual; membayangkan sebuah *graph* ketimbang *path* tunggal. Pendekatan *wholemeal* seringkali menawarkan perspektif baru terhadap masalah yang diberikan. Hal ini juga dengan sempurna dilengkapi dengan ide dari pemrograman proyektif: pertama selesaikan masalah yang lebih umum, lalu ekstrak bagian dan potongan yang menarik dengan mentransformasikan masalah umum tadi ke yang masalah yang lebih spesifik."

Sebagai contoh, perhatikan *pseudocode* berikut ini di bahasa C/Java-ish:

```
int acc = 0;
for ( int i = 0; i < lst.length; i++ ) {
  acc = acc + 3 * lst[i];
}
```

Kode ini menderita apa yang dikatakan Richard Bird dengan istilah "indexities", yakni kita harus khawatir terhadap detail *low-level* dari iterasi array dengan tetap mencatat indeks saat ini. Kode tersebut juga menggabungkan apa yang baiknya dipisahkan sebagai dua operasi berbeda: mengalikan setiap item dengan 3, dan menjumlahkan semua hasilnya.

Di Haskell, kita cukup menuliskan

```
sum (map (3*) lst)
```

Semester ini kita akan mengeksplorasi pergeseran cara berpikir dengan cara pemrograman seperti ini, dan mememeriksa bagaimana dan mengapa Haskell membuatnya menjadi mungkin.

Literate Haskell

File ini adalah "dokumen *literate* Haskell". Baris yang diawali dengan > dan spasi (lihat dibawah) merupakan kode. Lainnya (seperti paragraf ini) adalah komentar. Tugas pemrograman kalian tidak harus berupa *literate* haskell, meskipun diperbolehkan jika kalian mau. Dokumen *literate* Haskell berekstensi .lhs, sedangkan kode sumber *non-literate* Haskell berekstensi .hs.

Deklarasi dan variabel

Berikut ini adalah kode Haskell:

```
x :: Int
x = 3

-- Perhatikan bahwa komentar (non-literate) normal diawali dengan dua tanda strip
{- atau diapit dalam pasangan
   kurung kurawal/strip. -}
```

Kode diatas mendeklarasikan variabel `x` dengan tipe `Int` (:: diucapkan "memiliki tipe") dan mendeklarasikan nilai `x` menjadi `3`. Perhatikan bahwa nilai ini akan menjadi nilai `x` selamanya (paling tidak dalam program kita saja). Nilai dari `x` tidak akan bisa diganti kemudian.

Coba *uncomment* baris dibawah ini; kalian akan mendapati kesalahan yang berbunyi `Multiple declarations of `x'`.

```
-- x = 4
```

Di Haskell, variabel bukanlah kotak mutable yang bisa diubah-ubah; mereka hanyalah nama untuk suatu nilai.

Dengan kata lain, `=` tidak menyatakan "assignment" seperti di bahasa lain. Alih-alih, `=` menyatakan definisi seperti di matematika. `x = 4` tidak seharus-

nya dibaca "`x` memperoleh `4`" atau "*assign* `4` ke `x`", tetapi harus dibaca "`x` *didefinisikan sebagai* `4`".

Menurut kalian apa arti dari kode berikut?

```
y :: Int
y = y + 1
```

Basic Types

```
-- Machine-sized integers
i :: Int
i = -78
```

`Int` dijamin oleh standar bahasa Haskell untuk mengakomodasi nilai paling tidak sebesar \(\pm 2^{29}\), tapi ukuran pastinya bergantung pada arsitektur kalian. Sebagai contoh, di mesin 64-bit saya kisarannya sampai 2^63. Kalian bisa mencari tahu kisarannya dengan mengevaluasi kode dibawah ini:

```
intTerbesar, intTerkecil :: Int
intTerbesar  = maxBound
intTerkecil = minBound
```

(Perhatikan bahwa Haskell idiomatik mengunakan camelCase untuk nama *identifier*. Jika kalian tidak menyukainya, terimalah saja.)

Di sisi lain, tipe Integer hanya dibatasi oleh kapasitas memori di mesin kalian.

```
-- Arbitrary-precision integers
n :: Integer
n = 123456789098765432198734098233498734987234987453 4

sangatBesar :: Integer
sangatBesar = 2^(2^(2^(2^2)))

banyaknyaDigit :: Int
banyaknyaDigit = length (show sangatBesar)
```

Untuk angka *floating-point*, ada `Double`:

```
-- Double-precision floating point
d1, d2 :: Double
d1 = 4.5387
d2 = 6.2831e-4
```

Ada juga tipe angka *single-precision floating point*, `Float`.

Akhirnya, kita juga punya *boolean*, karakter, dan string:

```
-- Booleans
b1, b2 :: Bool
```

```haskell
b1 = True
b2 = False

-- Karakter unicode
c1, c2, c3 :: Char
c1 = 'x'
c2 = 'Ø'
c3 = ' '

-- String adalah list dari karakter dengan sintaks khusus
s :: String
s = "Hai, Haskell!"
```

GHCi

GHCi adalah sebuah REPL (*Read-Eval-Print-Loop*) Haskell interaktif yang satu paket dengan GHC. Di *prompt* GHCi, kalian bisa mengevaluasi ekspresi, memuat berkas Haskell dengan `:load` (`:l`) (dan memuat ulang mereka dengan `:reload` (`:r`)), menanyakan tipe dari suatu ekspresi dengan `:type` (`:t`), dan banyak hal lainnya (coba `:?` untuk melihat perintah-perintahnya).

Aritmatika

Coba evaluasi ekspresi-ekspresi ini di GHCi:

```haskell
ex01 = 3 + 2
ex02 = 19 - 27
ex03 = 2.35 * 8.6
ex04 = 8.7 / 3.1
ex05 = mod 19 3
ex06 = 19 `mod` 3
ex07 = 7 ^ 222
ex08 = (-3) * (-7)
```

Perhatikan bagaimana ‘backticks‘ membuat fungsi menjadi operator *infix*. Perhatikan juga angka negatif seringkali harus diapit tanda kurung, untuk mencegah tanda negasi di-*parse* sebagai operasi pengurangan. (Ya, ini terlihat jelek. Mohon maaf.)

Sedangkan berikut ini akan menghasilkan *error*:

```haskell
-- badArith1 = i + n
```

Penjumlahan hanya berlaku untuk penjumlahan nilai bertipe numerik sama, dan Haskell tidak melakukan perubahan secara implisit. Kalian harus secara eksplisit mengubahnya dengan:

- `fromIntegral`: mengubah dari tipe integral apapun (`Int` atau `Integer`) ke tipe numerik lainnya.
- `round`, `floor`, `ceiling`: mengubah angka *floating-point* ke `Int` atau `Integer`.

Sekarang coba ini:

```
-- badArith2 = i / i
```

Ini juga menghasilkan *error* karena `/` melakukan pembagian hanya untuk angka *floating-point*. Untuk pembagian integer kita menggunakan `div`.

```
ex09 = i `div` i
ex10 = 12 `div` 5
```

Jika kalian terbiasa dengan bahasa lain yang melakukan perubahan tipe numerik secara implisit, ini terkesan sangat mengganggu pada awalnya. Akan tetapi, saya jamin kalian akan terbiasa, dan bahkan pada akhirnya akan menghargainya. Perubahan numerik secara implisit membuat pemikiran kita tentang kode numerik menjadi tidak rapi.

Logika *boolean*

Seperti yang kalian duga, nilai *Boolean* bisa digabung satu sama lain dengan `(&&)` (*logical and*), `(||)` (*logical or*), dan `not`. Sebagai contoh,

```
ex11 = True && False
ex12 = not (False || True)
```

Nilai juga bisa dibandingkan kesamaannya satu sama lain dengan `(==)` dan `(/=)`, atau dibandingkan urutannya dengan menggunakan `(<)`, `(>)`, `(<=)`, dan `(>=)`.

```
ex13 = ('a' == 'a')
ex14 = (16 /= 3)
ex15 = (5 > 3) && ('p' <= 'q')
ex16 = "Haskell" > "C++"
```

Haskell juga memiliki ekspresi `if`: `if b then t else f` adalah sebuah ekspresi yang mengevaluasi `t` jika ekspresi *boolean* `b` bernilai `True`, dan `f` jika `b` bernilai `False`. Perhatikan bahwa ekspresi `if` berbeda dengan *statement* `if`. Di dalam *statement* `if` bagian `else` adalah opsional, jika tidak ada maka berarti "jika tes bernilai `False`, jangan lakukan apapun". Sedangkan pada ekspresi `if`, bagian `else` wajib ada karena ekspresi `if` harus menghasilkan sebuah nilai.

Haskell yang idiomatik tidak banyak menggunakan ekspresi `if`, kebanyakan menggunakan *pattern-matching* atau *guard* (lihat bagian berikut).

Mendefinisikan fungsi

Kita bisa menulis fungsi untuk integer secara per kasus.

```
-- Jumlahkan integer dari 1 sampai n.
sumtorial :: Integer -> Integer
sumtorial 0 = 0
sumtorial n = n + sumtorial (n-1)
```

Perhatikan sintaks untuk tipe fungsi: `sumtorial :: Integer -> Integer` yang berarti `sumtorial` adalah sebuah fungsi yang menerima sebuah `Integer` sebagai input dan menghasilkan `Integer` sebagai output.

Tiap klausa dicek berurutan dari atas ke bawah dan yang pertama kali cocok akan digunakan. Sebagai contoh, evaluasi `sumtorial 0` akan menghasilkan `0`, karena cocok dengan klausa pertama. `sumtorial 3` tidak cocok dengan klausa pertama (`3` tidak sama dengan `0`) sehingga klausa kedua dicoba. Sebuah variabel seperti `n` cocok dengan apapun sehingga klausa kedua cocok dan `sumtorial 3` dievaluasi menjadi `3 + sumtorial (3 -1)` (yang juga bisa dievaluasi lebih lanjut).

Pilihan juga bisa dibuat dengan ekspresi Boolean menggunakan *guards*. Contoh:

```
hailstone :: Integer -> Integer
hailstone n
  | n `mod` 2 == 0 = n `div` 2
  | otherwise      = 3*n + 1
```

Tiap *guard* yang berupa ekspresi Boolean bisa diasosiasikan dengan klausa di definisi fungsi. Jika pola klausa cocok, *guards* akan dievaluasi berurutan dari atas ke bawah dan yang pertama dievaluasi `True` akan dipilih. Jika tidak ada yang `True`, pencocokan akan dilanjutkan ke klausa berikutnya.

Sebagai contoh, berikut adalah evaluasi `hailstone 3`. `3` dicocokkan dengan `n` dan cocok (karena variabel cocok dengan apapun). Lalu, `n `mod` 2` dievaluasi, hasilnya `False` karena `n = 3` tidak menghasilkan sisa `0` ketika dibagi `2`. `otherwise` hanyalah sinonim untuk `True`, sehingga *guard* kedua dipilih dan hasil dari `hailstone 3` ialah `3*3 + 1 = 10`.

Contoh yang lebih rumit:

```
foo :: Integer -> Integer
foo 0 = 16
foo 1
  | "Haskell" > "C++" = 3
  | otherwise         = 4
foo n
  | n < 0             = 0
  | n `mod` 17 == 2   = -43
  | otherwise         = n + 3
```

Apa hasil dari `foo (-3)`? `foo 0`? `foo 1`? `foo 36`? `foo 38`?

Misalkan kita ingin membawa test genapnya bilangan keluar dari definisi `hailstone`, berikut adalah contohnya:

```
isEven :: Integer -> Bool
isEven n
  | n `mod` 2 == 0 = True
  | otherwise      = False
```

Seperti ini juga bisa, tapi lebih rumit. Terlihat jelas kan?

Pairs

Kita bisa membuat hal berpasangan seperti berikut:

```
p :: (Int, Char)
p = (3, 'x')
```

Perhatikan bahwa notasi `(x,y)` digunakan untuk **tipe** dari *pair* dan **nilai** dari *pair*.

Elemen dari sebuah *pair* bisa diekstrak dengan mencocokkan pola (*pattern matching*):

```
sumPair :: (Int,Int) -> Int
sumPair (x,y) = x + y
```

Haskell juga memiliki *triple*, *quadruple*, tapi sebaiknya jangan kalian gunakan. Akan kita lihat minggu depan, ada cara lebih baik untuk menyatukan tiga atau lebih informasi.

Menggunakan fungsi dengan beberapa argumen

Untuk aplikasi fungsi ke beberapa argumen, cukup letakkan argumen-argumen tersebut setelah fungsi, dipisahkan dengan spasi seperti ini:

```
f :: Int -> Int -> Int -> Int
f x y z = x + y + z
ex17 = f 3 17 8
```

Contoh di atas menerapkan fungsi `f` ke tiga argumen: `3`, `17`, dan dan `8`. Perhatikan juga sintaks tipe untuk fungsi dengan beberapa argumen seperti: `Arg1Type -> Arg2Type -> ... -> ResultType`. Ini mungkin terlihat aneh (memang sudah seharusnya). Mengapa semuanya tanda panah? Bukannya lebih wajar kalau tipe untuk `f` berupa `Int Int Int -> Int`? Sebenarnya, sintaks ini memang disengaja dan memiliki alasan yang mendalam dan indah, yang akan kita pelajari beberapa minggu lagi. Untuk sementara ini, kalian percaya saja dulu.

Perhatikan bahwa **aplikasi fungsi memiliki prioritas (*precedence*) lebih tinggi ketimbang operator *infix***. Jadi penulisan seperti ini

`f 3 n+1 7`

adalah salah jika kalian ingin memberi `n+1` sebagai argumen kedua ke `f` karena akan di*parse* sebagai

`(f 3 n) + (1 7)`.

Penulisan yang benar adalah:

`f 3 (n+1) 7`.

List

List adalah satu tipe data dasar di Haskell.

```
nums, range, range2 :: [Integer]
nums   = [1,2,3,19]
range  = [1..100]
range2 = [2,4..100]
```

Haskell (seperti Python) juga memiliki *list comprehensions*. Kalian bisa mempelajarinya di LYAH.

String hanyalah list karakter. Dengan kata lain, `String` hanyalah singkatan dari `[Char]`, dan sintak literal string (teks di dalam tanda kutip ganda) hanyalah singkatan untuk literal list `Char`.

```
-- hello1 dan hello2 adalah sama.

hello1 :: [Char]
hello1 = ['h', 'e', 'l', 'l', 'o']

hello2 :: String
hello2 = "hello"

helloSame = hello1 == hello2
```

Ini berarti semua fungsi di librari standar untuk memproses list juga bisa digunakan untuk memproses `String`.

Membangun list

List yang paling sederhana ialah list kosong:

```
emptyList = []
```

List lainnya dibangun dari list kosong dengan menggunakan operator *cons* , `(:)`. *Cons* menerima argumen sebuah elemen dan sebuah list, dan mengembalikan list baru dengan elemen tersebut ditambahkan ke depan list.

```
ex17 = 1 : []
ex18 = 3 : (1 : [])
ex19 = 2 : 3 : 4 : []

ex20 = [2,3,4] == 2 : 3 : 4 : []
```

Kita bisa melihat bahwa notasi `[2,3,4]` hanyalah singkatan untuk `2 : 3 : 4 : []`. Perhatikan juga bahwa ini adalah *singly linked lists*, BUKAN arrays.

```
-- Buat barisan dari iterasi hailstone dari bilangan awal.
hailstoneSeq :: Integer -> [Integer]
hailstoneSeq 1 = [1]
hailstoneSeq n = n : hailstoneSeq (hailstone n)
```

Kita stop barisan *hailstone* ketika mencapai 1. Barisan *hailstone* untuk `n` terdiri dari `n` itu sendiri, diikuti dengan barisan dari `hailstone n` yang merupakan bilangan yang didapat dari menerapkan fungsi `hailstone` ke `n`.

Fungsi pada list

Kita bisa menulis fungsi pada list menggunakan pencocokan pola (*pattern matching*).

```
-- Hitung panjang sebuah list Integer.
intListLength :: [Integer] -> Integer
intListLength []     = 0
intListLength (x:xs) = 1 + intListLength xs
```

Klausa pertama menyatakan bahwa panjang dari sebuah list kosong adalah 0. Klausa kedua menyatakan jika input list berbentuk seperti `(x:xs)`, yaitu elemen pertama `x` disambung (*cons*) ke sisa list `xs`, maka panjang dari list tersebut ialah lebih dari satu panjangnya `xs`.

Karena kita tidak menggunakan `x` sama sekali, kita bisa menggantinya dengan *underscore*: `intListLength (_:xs) = 1 + intListLength xs`.

Kita juga bisa menggunakan pola bertumpuk (*nested patterns*):

```
sumEveryTwo :: [Integer] -> [Integer]
sumEveryTwo []         = []     -- Biarkan list kosong
sumEveryTwo (x:[])     = [x]    -- Biarkan list dengan elemen tunggal
sumEveryTwo (x:(y:zs)) = (x + y) : sumEveryTwo zs
```

Perhatikan bagaimana klausa terakhir mencocokkan list yang dimulai dengan `x` lalu diikuti dengan `y` dan diikuti dengan list `zs`. Kita sebenarnya tidak memer-

lukan tanda kurung tambahan, jadi bisa juga ditulis menjadi `sumEveryTwo (x:y:zs) = ...`.

Kombinasi fungsi

Menggabungkan fungsi-fungsi sederhana untuk membangun fungsi yang kompleks merupakan cara memprogram Haskell yang baik.

```
-- Jumlah hailstone yang dibutuhkan untuk mencapai 1
-- dari bilangan awal.
hailstoneLen :: Integer -> Integer
hailstoneLen n = intListLength (hailstoneSeq n) - 1
```

Ini mungkin terlihat tidak efisien bagi kalian. Fungsi tersebut membangun seluruh barisan *hailstone* lalu menghitung panjangnya. Tentunya boros memori, bukan? Ternyata tidak! Karena Haskell dievaluasi secara *lazy*, tiap elemen hanya akan dibangun ketika dibutuhkan. Jadi, pembuatan barisan dan penghitungan panjang dilakukan secara berselingan. Seluruh komputasi hanya memakai memori O(1), tak peduli sepanjang apapun barisannya (Sebenarnya ini sedikit dusta tapi penjelasannya dan cara mengoreksinya harus menunggu beberapa minggu).

Kita akan belajar lebih jauh mengenai evaluasi *lazy* di Haskell beberapa minggu lagi. Untuk saat ini cukup ketahui: jangan takut untuk menulis fungsi kecil yang mengubah seluruh struktur data, dan menggabungkan fungsi-fungsi tersebut untuk membangun fungsi yang lebih kompleks. Mungkin akan terasa ganjil pada awalnya, tapi beginilah cara menulis program Haskell yang idiomatis dan efisien. Lagipula, setelah terbiasa, kalian akan merasa nyaman dengannya.

Sepatah kata tentang pesan kesalahan (*error message*)

Sebenarnya, lima kata:

Jangan takut dengan pesan kesalahan

Pesan kesalahan dari GHC bisa panjang dan terlihat menakutkan. Biasanya pesan tersebut panjang bukan karena tidak jelas, tapi karena mengandung banyak informasi. Sebagai contoh:

```
Prelude> 'x' ++ "foo"

<interactive>:1:1:
    Couldn't match expected type `[a0]' with actual type `Char'
    In the first argument of `(++)', namely 'x'
    In the expression: 'x' ++ "foo"
    In an equation for `it': it = 'x' ++ "foo"
```

Pesan pertama: "Couldn't match expected type `[a0]` with actual type `Char`", yang berarti tidak bisa mencocokkan tipe yang diharapkan `[a0]` dengan tipe yang ada `Char`. Ini berarti *sesuatu* diharapkan bertipe list, tapi malah bertipe `Char`. *Sesuatu* apa? Baris berikutnya berkata, argumen pertama dari `(++)` yang salah, bernama `x`. Baris berikutnya lagi membuat semakin jelas. Masalahnya adalah: `x` bertipe `Char` seperti yang dikatakan oleh baris pertama. Mengapa diharapkan bertipe list? Karena itu digunakan sebagai argumen pertama `(++)`, yang menerima list sebagai argumen pertama.

Ketika menerima pesan kesalahan yang panjang, janganlah takut. Ambil nafas panjang, dan baca dengan seksama. Mungkin kalian tidak akan mengerti seluruhnya, tapi kalian akan belajar banyak dan mungkin bisa mendapatkan informasi bagaimana cara mengatasinya.

Algebraic data types

Bacaan tambahan:

- *Real World Haskell*, bab 2 dan 3

Tipe Enumerasi

Seperti bahasa pemrograman lain, Haskell membolehkan programer membuat tipe enumerasi (*enumeration types*) sendiri. Contoh sederhana:

```
data Thing = Shoe
           | Ship
           | SealingWax
           | Cabbage
           | King
  deriving Show
```

Kita mendeklarasikan tipe baru bernama `Thing` dengan lima konstruktor data (*data constructors*): `Shoe`, `Ship`, dan seterusnya, yang merupakan nilai dari tipe `Thing`. `deriving Show` ialah mantra yang memberitahu GHC untuk membuat kode konversi dari `Thing` ke `String` secara otomatis. Hal tersebut digunakan ketika `ghci` mencetak nilai dari ekspresi bertipe `Thing`.

```
shoe :: Thing
shoe = Shoe

listO'Things :: [Thing]
listO'Things = [Shoe, SealingWax, King, Cabbage, King]
```

Kita bisa menulis fungsi terhadap `Thing`s dengan pencocokkan pola (*pattern-matching*).

```
isSmall :: Thing -> Bool
isSmall Shoe       = True
isSmall Ship       = False
isSmall SealingWax = True
isSmall Cabbage    = True
isSmall King       = False
```

Mengingat klausa fungsi dicoba dari atas ke bawah, kita bisa menyingkat definisi dari `isSmall` seperti berikut:

```
isSmall2 :: Thing -> Bool
isSmall2 Ship = False
isSmall2 King = False
isSmall2 _    = True
```

Lebih jauh tentang enumerasi

`Thing` bertipe enumerasi (*enumeration type*), mirip dengan yang ada di bahasa lain seperti Java atau C++. Sebenarnya, enumerasi hanyalah kasus spesifik dari sesuatu yang lebih umum di Haskell: tipe data *algebraic* (*algebraic data types*). Sebagai contoh tipe data yang bukan sekedar enumerasi, perhatikan definisi dari `FailableDouble` berikut ini:

```
data FailableDouble = Failure
                    | OK Double
  deriving Show
```

Di sini, tipe `FailableDouble` memiliki dua konstruktor data. Yang pertama, `Failure`, tidak memerlukan argumen. Jadi `Failure` itu sendiri ialah nilai yang bertipe `FailableDouble`. Yang kedua, `OK`, menerima satu argumen bertipe `Double`. `OK` yang berdiri sendiri bukanlah bertipe `FailableDouble`, kita harus memberinya sebuah `Double`. Sebagai contoh, `OK 3.4` ialah nilai bertipe `FailableDouble`.

```
exD1 = Failure
exD2 = OK 3.4
```

Coba tebak: `OK` sendiri bertipe apa?

```
safeDiv :: Double -> Double -> FailableDouble
safeDiv _ 0 = Failure
safeDiv x y = OK (x / y)
```

Pencocokkan pola lagi! Perhatikan pada kasus `OK`, kita bisa memberi nama kepada `Double`-nya.

```
failureToZero :: FailableDouble -> Double
failureToZero Failure = 0
failureToZero (OK d)  = d
```

Konstruktor data bisa memiliki lebih dari satu argumen.

```
-- Simpan nama, umur, dan Thing favorit dari seseorang.
data Person = Person String Int Thing
  deriving Show

brent :: Person
brent = Person "Brent" 31 SealingWax

stan :: Person
stan  = Person "Stan" 94 Cabbage

getAge :: Person -> Int
getAge (Person _ a _) = a
```

Perhatikan bahwa konstruktor tipe (*type constructor*) dan konstruktor data sama-sama bernama `Person`, tetapi mereka berada di *namespace* yang berbeda dan merupakan dua hal yang berbeda pula. Konstruktor tipe dan data yang bernama sama ini cukup umum dan akan cukup membingungkan jika belum terbiasa.

Tipe data *algebraic* secara umum

Pada umumnya, sebuah tipe data *algebraic* memiliki satu atau lebih konstruktor data, dan tiap konstruktor data bisa memiliki nol atau lebih argumen.

```
data AlgDataType = Constr1 Type11 Type12
                 | Constr2 Type21
                 | Constr3 Type31 Type32 Type33
                 | Constr4
```

Ini menyatakan bahwa nilai dari tipe `AlgDataType` bisa dibangun dengan empat cara: menggunakan `Constr1`, `Constr2`, `Constr3`, atau `Constr4`. Tergantung dari konstruktor yang digunakan, nilai `AlgDataType` bisa mengandung nilai-nilai berbeda. Misalnya, jika dibangun dengan menggunakan `Constr1` maka nilai tersebut mengandung dua nilai, satu bertipe `Type11` dan satu lagi bertipe `Type12`.

Perlu diingat: nama konstruktor tipe dan konstruktor data harus selalu dimulai dengan huruf besar, sedangkan variabel (termasuk nama fungsi) harus selalu dimulai dengan huruf kecil. Hal ini untuk memudahkan parser Haskell mengetahui mana nama yang merepresentasikan variabel, dan mana yang merepresentasikan konstruktor.

Pencocokkan pola

Kita sudah melihat pencocokkan pola (*pattern-matching*) di beberapa kasus. Kali ini kita akan melihat bagaimana pencocokkan pola bekerja secara umum. Pada dasarnya, pencocokkan pola ialah memisahkan nilai berdasarkan konstruktor yang membangunnya. Informasi tersebut bisa digunakan sebagai penentu apa yang harus dilakukan. Ini adalah satu-satunya cara di Haskell.

Sebagai contoh, untuk menentukan apa yang harus dilakukan dengan nilai bertipe `AlgDataType` (tipe yang kita buat sebelumnya), kita bisa menulis seperti

```
foo (Constr1 a b)   = ...
foo (Constr2 a)     = ...
foo (Constr3 a b c) = ...
foo Constr4         = ...
```

Perhatikan kita juga memberikan nama ke nilai-nilai di dalam tiap konstruktor. Perhatikan juga tanda kurung diperlukan untuk pola (*patterns*) yang terdiri dari lebih dari konstruktor tunggal.

Itulah ide utama dari pola, tapi ada beberapa hal lain yang perlu diperhatikan.

1. Sebuah *underscore* `_` bisa digunakan sebagai "wildcard pattern" yang cocok dengan apapun.

2. Sebuah pola berbentuk `x@pat` bisa digunakan untuk mencocokkan nilai dengan pola `pat`, tapi *juga* memberikan nama `x` ke seluruh nilai yang dicocokkan. Sebagai contoh:

   ```
   baz :: Person -> String
   baz p@(Person n _ _) = "The name field of (" ++ show p ++ ") is " ++ n
   ```

   ```
   *Main> baz brent
   "The name field of (Person \"Brent\" 31 SealingWax) is Brent"
   ```

3. Pola bisa bertumpuk (*nested*). Sebagai contoh:

   ```
   checkFav :: Person -> String
   checkFav (Person n _ SealingWax) = n ++ ", you're my kind of person!"
   checkFav (Person n _ _)          = n ++ ", your favorite thing is lame."
   ```

   ```
   *Main> checkFav brent
   "Brent, you're my kind of person!"
   *Main> checkFav stan
   "Stan, your favorite thing is lame."
   ```

   Perhatikan bagaimana kita menumpuk pola `SealingWax` di dalam pola `Person`.

Grammar berikut mendefinisikan apa yang bisa digunakan sebagai pola:

```
pat ::= _
```

```
|  var
|  var @ ( pat )
|  ( Constructor pat1 pat2 ... patn )
```

Baris pertama menyatakan bahwa *underscore* adalah sebuah pola (*pattern*). Baris kedua menyatakan sebuah variabel juga merupakan sebuah pola, yang cocok dengan apapun dan memberikan nama variabel tersebut ke nilai yang dicocokkan. Baris ketiga adalah pola `@`. Baris terakhir menyatakan bahwa nama konstruktor yang diikuti oleh barisan pola juga merupakan sebuah pola. Pola ini cocok dengan nilai yang dibangun dengan konstruktor tersebut, *dan* semua dari `pat1` sampai `patn` cocok dengan nilai-nilai di dalam konstruktor secara rekursif.

(Sebenarnya masih banyak yang terkandung di pola *grammar*, tapi kita tidak perlu sejauh itu.)

Nilai seperti `2` atau `'c'` bisa dibayangkan sebagai konstruktor tanpa argumen. Dengan kata lain, bagaikan tipe `Int` dan `Char` didefinisikan seperti

```
data Int  = 0 | 1 | -1 | 2 | -2 | ...
data Char = 'a' | 'b' | 'c' | ...
```

yang berarti kita bisa mencocokkan pola dengan nilai literal. (Tentu saja sebenarnya `Int` dan `Char` tidak didefinisikan seperti demikian.)

Ekspresi *case*

Konstruksi dasar untuk pencocokkan pola di Haskell ialah ekspresi `case`. Pada umumnya, sebuah ekspresi `case` berbentuk

```
case exp of
  pat1 -> exp1
  pat2 -> exp2
  ...
```

Ketika dievaluasi, ekspresi `exp` dicocokkan dengan tiap pola `pat1`, `pat2`, dan seterusnya secara bergantian. Pola yang pertama cocok akan dipilih, dan seluruh ekspresi `case` akan terevaluasi menjadi ekspresi yang bersesuaian dengan pola yang cocok tersebut. Sebagai contoh,

```
exCase = case "Hello" of
           []      -> 3
           ('H':s) -> length s
           _       -> 7
```

terevaluasi menjadi `4` (pola kedua yang terpilih, pola ketiga juga cocok tapi tidak terjangkau).

Sintaks untuk mendefinisikan fungsi yang telah kita lihat sebelumnya hanyalah singkatan untuk mendefinisikan ekspresi `case`. Sebagai contoh, definisi

`failureToZero` yang telah kita temui sebelumnya bisa ditulis ulang menjadi

```
failureToZero' :: FailableDouble -> Double
failureToZero' x = case x of
                     Failure -> 0
                     OK d    -> d
```

Tipe data rekursif

Tipe data bisa *rekursif*, yaitu didefinisikan dalam bentuk dirinya sendiri. Kita sudah melihat sebuah tipe rekursif, list. List bisa kosong, atau berisi satu elemen diikuti list sisanya. Kita bisa mendefinisikan list kita sendiri seperti:

```
data IntList = Empty | Cons Int IntList
```

Built-in list di Haskell cukup serupa. List-list tersebut menggunakan sintaks bawaan, `[]` dan `:`. Mereka juga berfungsi pada tipe elemen apapun , bukan hanya `Int` (akan dibahas lebih jauh di bab berikutnya).

Kita sering menggunakan fungsi rekursif untuk memproses tipe data rekursif:

```
intListProd :: IntList -> Int
intListProd Empty      = 1
intListProd (Cons x l) = x * intListProd l
```

Contoh lainnya, kita bisa mendefinisikan sebuah tipe pohon biner (*binary tree*) dengan sebuah nilai `Int` tersimpan di tiap *node* internal, dan `Char` di tiap *leaf*:

```
data Tree = Leaf Char
          | Node Tree Int Tree
  deriving Show
```

(Jangan bertanya untuk apa pohon tersebut, ini hanyalah contoh. Ok?)

Contohnya:

```
tree :: Tree
tree = Node (Leaf 'x') 1 (Node (Leaf 'y') 2 (Leaf 'z'))
```

Pola rekursi, polimorfisme, dan *Prelude*

Setelah menyelesaikan Tugas 2, kalian pasti banyak menghabiskan waktu menulis fungsi rekursif. Kalian mungkin mengira programer Haskell banyak menghabiskan waktu untuk menulis fungsi rekursif. Sebaliknya, programer Haskell yang berpengalaman sangatlah jarang menulis fungsi rekursif!

Bagaimana mungkin? Kuncinya adalah menyadari meskipun fungsi rekursif bisa melakukan apapun, pada prakteknya ada beberapa pola umum yang seringkali muncul. Dengan mengabstrak keluar (*abstracting out*) pola-pola tersebut

ke fungsi-fungsi pustaka (*library*), programer bisa meninggalkan detil *low level* rekursif ke fungsi-fungsi tersebut, dan fokus ke masalah yang *higher level*. Itulah tujuan dari *wholemeal programming*.

Pola rekursi

Ingat definisi list yang berisi nilai `Int` sebelumnya:

```
data IntList = Empty | Cons Int IntList
  deriving Show
```

Apa yang kita bisa lakukan terhadap `IntList`? Berikut ini adalah beberapa kemungkinan:

- Lakukan operasi pada tiap elemen di list
- Buang sebagian elemen, dan simpan sisanya, berdasarkan sebuah tes
- "Simpulkan" seluruh elemen di list (seperti: cari jumlah total (*sum*), *product*, maximum, dan lain-lain).
- Kalian mungkin masih bisa memberikan contoh lain!

Map

Mari kita bahas yang pertama: lakukan operasi pada tiap elemen di list. Sebagai contoh, kita tambahkan satu ke tiap elemen di list.

```
addOneToAll :: IntList -> IntList
addOneToAll Empty       = Empty
addOneToAll (Cons x xs) = Cons (x+1) (addOneToAll xs)
```

Atau kita bisa memastikan semua elemen di list tidak negatif dengan mengambil nilai mutlak (*absolute*):

```
absAll :: IntList -> IntList
absAll Empty       = Empty
absAll (Cons x xs) = Cons (abs x) (absAll xs)
```

Atau kita bisa mengkuadratkan tiap elemen:

```
squareAll :: IntList -> IntList
squareAll Empty       = Empty
squareAll (Cons x xs) = Cons (x*x) (squareAll xs)
```

Sampai di sini, kalian seharusnya sudah menyadari. Ketiga fungsi tersebut terlihat sangat serupa. Pasti ada cara untuk mengabstraksi keluar kesamaannya sehingga kita tidak perlu mengulang-ulang.

Ternyata ada caranya. Bisakah kalian menemukannya? Bagian mana yang serupa di ketiga hal tersebut dan mana yang berbeda?

Yang berbeda tentunya adalah operasi yang ingin kita lakukan pada tiap elemen di list. Kita bisa menyebutnya sebagai fungsi bertipe `Int -> Int`. Di sini kita bisa melihat betapa berguna untuk bisa memberikan fungsi sebagai input ke fungsi lainnya.

```
mapIntList :: (Int -> Int) -> IntList -> IntList
mapIntList _ Empty       = Empty
mapIntList f (Cons x xs) = Cons (f x) (mapIntList f xs)
```

Sekarang kita bisa menggunakan `mapIntList` untuk membuat `addOneToAll`, `absAll`, dan `squareAll`:

```
exampleList = Cons (-1) (Cons 2 (Cons (-6) Empty))

addOne x = x + 1
square x = x * x

mapIntList addOne exampleList
mapIntList abs    exampleList
mapIntList square exampleList
```

Filter (saring)

Satu pola yang sering muncul ialah ketika kita ingin menyimpan sebagian elemen dari list dan membuang sisanya dengan melakukan sebuah tes. Sebagai contoh, kita ingin menyimpan hanya bilangan positif saja:

```
keepOnlyPositive :: IntList -> IntList
keepOnlyPositive Empty = Empty
keepOnlyPositive (Cons x xs)
  | x > 0     = Cons x (keepOnlyPositive xs)
  | otherwise = keepOnlyPositive xs
```

Atau hanya bilangan genap:

```
keepOnlyEven :: IntList -> IntList
keepOnlyEven Empty = Empty
keepOnlyEven (Cons x xs)
  | even x    = Cons x (keepOnlyEven xs)
  | otherwise = keepOnlyEven xs
```

Bagaimana bisa mengeneralisir pola ini? Mana yang serupa dan mana yang bisa diabstrak keluar?

Yang bisa diabstrak keluar adalah tes (atau *predicate*) yang digunakan untuk menentukan apakah nilai tersebut akan disimpan. Sebuah *predicate* adalah sebuah fungsi bertipe `Int -> Bool` yang mengembalikan `True` untuk tiap elemen yang akan disimpan, dan `False` untuk yang akan dibuang. Jadi, kita bisa menulis `filterIntList` seperti berikut:

```
filterIntList :: (Int -> Bool) -> IntList -> IntList
filterIntList _ Empty = Empty
```

```
filterIntList p (Cons x xs)
  | p x       = Cons x (filterIntList p xs)
  | otherwise = filterIntList p xs
```

–>

Fold (lipat)

Pola terakhir yang tadi kita singgung adalah "menyimpulkan" atau "merangkum" semua elemen di list. Ini biasa disebut operasi *fold* atau *reduce*. Kita akan kembali ke sini minggu depan. Sementara itu, kalian mungkin bisa berpikir bagaimana untuk mengabstrak keluar pola ini.

Polimorfisme (*Polymorphism*)

Kita telah menulis beberapa fungsi umum untuk *mapping* dan *filtering* (menyaring) list yang berisi `Int`. Tapi kita belum selesai menggeneralisir! Bagaimana jika kita ingin menyaring list berisi `Integer`? Atau `Bool`? Atau list yang berisi list yang berisi *tree* yang berisi tumpukan `String`? Kita jadi harus membuat tipe data baru dan fungsi baru untuk tiap kasus. Lebih buruk lagi, kodenya sama persis! Yang berbeda hanyalah notasi tipenya. Bisakah Haskell membantu kita di sini?

Tentu bisa! Haskell mendukung polimorfisme untuk tipe data dan fungsi. Kata polimorfis berasal dari kata Yunani ( ) yang berarti "memiliki banyak bentuk". Sesuatu yang polimorfis bisa bekerja untuk beberapa tipe.

Tipe data polimorfis

Pertama, kita lihat bagaimana mendeklarasikan tipe data polimorfis.

```
data List t = E | C t (List t)
```

(Kita tidak bisa menggunakan `Empty` dan `Cons` karena kita telah memakainya sebagai konstruktor untuk `IntList`. Sebagai gantinya kita menggunakan `E` dan `C`.)

Sebelumnya kita memiliki `data IntList = ...`, sekarang `data List t = ...`. `t` merupakan variabel tipe (*type variable*) yang bisa berarti tipe apapun. Variabel tipe harus dimulai dengan huruf kecil, sedangkan tipe dengan huruf besar. `data List t = ...` berarti tipe `List` terparameter (*parameterized*) berdasarkan tipe, serupa dengan sebuah fungsi yang terparameter berdasarkan input.

Jika ada tipe `t`, maka sebuah `(List t)` terdiri atas konstruktor `E`, atau konstruktor `C` bersama nilai bertipe `t` dan `(List t)` lainnya. Berikut beberapa contoh:

```
lst1 :: List Int
lst1 = C 3 (C 5 (C 2 E))
```

```haskell
lst2 :: List Char
lst2 = C 'x' (C 'y' (C 'z' E))

lst3 :: List Bool
lst3 = C True (C False E)
```

Fungsi polimorfis

Sekarang mari menggeneralisir `filterIntList` supaya bisa bekerja terhadap `List` yang baru kita buat. Kita bisa mengambil kode `filterIntList` yang sudah ada dan mengganti `Empty` dengan `E` dan `Cons` dengan `C`:

```haskell
filterList _ E = E
filterList p (C x xs)
  | p x       = C x (filterList p xs)
  | otherwise = filterList p xs
```

Sekarang, bertipe apakah `filterList`? Kita lihat tipe apakah yang `ghci` *infer* untuknya:

```
*Main> :t filterList
filterList :: (t -> Bool) -> List t -> List t
```

Kita bisa membacanya sebagai: "untuk apapun tipe `t`, `filterList` menerima fungsi dari `t` ke `Bool` dan list `t`, dan mengembalikan list `t`".

Bagaimana dengan menggeneralisir `mapIntList`? Apa tipe yang harus kita berikan ke `mapList` sehingga bisa mengaplikasikan sebuah fungsi ke tiap elemen di `List t`?

Mungkin kita berpikir untuk memberikan tipe

```haskell
mapList :: (t -> t) -> List t -> List t
```

Ini bisa, tapi ketika kita mengaplikasikan `mapList` kita akan selalu mendapatkan list yang elemennya bertipe sama dengan elemen di list yang kita berikan. Hal ini cukup kaku, karena mungkin saja kita ingin melakukan `mapList show` untuk mengubah list `Int` menjadi list `String`. Berikut adalah tipe yang paling umum untuk `mapList`, berikut implementasinya:

```haskell
mapList :: (a -> b) -> List a -> List b
mapList _ E        = E
mapList f (C x xs) = C (f x) (mapList f xs)
```

Satu hal penting yang perlu diingat mengenai fungsi polimorfis ialah **pemanggil fungsi yang menentukan tipe**. Ketika kalian menulis sebuah fungsi polimorfis, fungsi tersebut harus bisa bekerja ke semua tipe. Hal ini –ditambah dengan Haskell yang tidak bisa memutuskan langsung berdasarkan tipe– memiliki implikasi menarik yang akan kita pelajari lebih jauh nanti.

Prelude

`Prelude` adalah sebuah modul berisi definisi fungsi-fungsi standar yang terimpor secara implisit ke tiap program Haskell. [Melihat-lihat dokumentasinya] (https://downloads.haskell.org/~ghc/latest/docs/html/libraries/base-4.8.1.0/Prelude.html) sangatlah dianjurkan untuk mengenal *tools* yang tersedia di sana.

Tentu saja list polimorfis terdefinisi di `Prelude`, beserta [fungsi-fungsi polimorfis untuk list tersebut] (https://downloads.haskell.org/~ghc/latest/docs/html/libraries/base-4.8.1.0/Prelude.html#g:13) Sebagai contoh, `filter` dan `map` adalah padanan dari `filterList` dan `mapList` yang tadi kita bahas. Bahkan, masih banyak fungsi-fungsi untuk list di modul `Data.List`.

Tipe polimorfis lain yang cukup berguna untuk diketahui ialah `Maybe`, didefinisikan sebagai

```
data Maybe a = Nothing | Just a
```

Sebuah nilai bertipe `Maybe a` bisa mengandung nilai bertipe `a` (terbungkus di dalam konstruktor `Just`), atau berupa `Nothing` (mewakili kegagalan atau kesalahan). Modul `Data.Maybe` memiliki fungsi-fungsi yang bekerja terhadap nilai bertipe `Maybe`.

Fungsi total dan parsial

Perhatikan tipe polimorfis berikut:

```
[a] -> a
```

Fungsi seperti apa yang bertipe demikian? Tipe di atas menyatakan bahwa jika diberi list apapun yang bertipe `a`, fungsi tersebut harus mengembalikan sebuah nilai bertipe `a`. Sebagai contoh, fungsi `head` di *Prelude* bertipe seperti ini.

Apakah yang terjadi jika `head` diberikan list kosong sebagai input? Mari kita lihat [kode sumber] (https://downloads.haskell.org/~ghc/latest/docs/html/libraries/base-4.8.1.0/src/GHC-List.html#head) dari `head`…

Program akan *crash*! Tak ada lagi yang bisa dilakukan karena program harus bisa bekerja untuk *semua* tipe. Tak mungkin untuk mengetahui tipe dari elemen yang tidak ada.

`head` dikenal sebagai *fungsi parsial*: ada input yang bisa membuat `head` *crash*. Fungsi-fungsi yang rekursif tanpa henti pada beberapa input juga disebut parsial. Fungsi yang terdefinisi lengkap pada semua kemungkinan input dikenal dengan nama *fungsi total*.

Menghindari fungsi-fungsi parsial sebisa mungkin adalah praktek Haskell yang bagus. Bahkan, menghindari fungsi parsial bisa dibilang praktek yang bagus di

bahasa pemrograman apapun – meski sangat sulit di beberapa bahasa. Haskell membuat hal ini mudah dan masuk akal.

`head` adalah sebuah kesalahan! Seharusnya itu tidak berada di `Prelude`. Fungsi-fungsi parsial lainnya di `Prelude` yang sebaiknya jangan kalian gunakan antara lain `tail`, `init`, `last`, dan `(!!)`. Sejak sekarang penggunaan salah satu dari fungsi-fungsi tersebut di tugas kuliah akan mengurangi nilai!

Jadi mesti bagaimana?

Mengganti fungsi-fungsi parsial

Kebanyakan fungsi seperti `head`,tail‘, dan lain-lain bisa digantikan dengan pencocokan pola (*pattern-matching*). Perhatikan definisi berikut:

```
doStuff1 :: [Int] -> Int
doStuff1 []  = 0
doStuff1 [_] = 0
doStuff1 xs  = head xs + (head (tail xs))

doStuff2 :: [Int] -> Int
doStuff2 []        = 0
doStuff2 [_]       = 0
doStuff2 (x1:x2:_) = x1 + x2
```

Fungsi-fungsi di atas menghasilkan hasil yang sama, dan keduanya total. Akan tetapi, hanya fungsi kedua yang *jelas* terlihat total, dan lebih mudah dibaca.

Menulis fungsi parsial

Bagaimana jika kalian suatu saat tersadar sedang menulis fungsi parsial? Ada dua pilihan. Yang pertama, ubah tipe hasil fungsi ke tipe yang bisa mengindikasikan kegagalan. Ingat kembali definisi dari `Maybe`:

```
data Maybe a = Nothing | Just a
```

Sekarang, andaikan kita sedang membuat `head`. Kita bisa menulisnya dengan aman seperti ini:

```
safeHead :: [a] -> Maybe a
safeHead []    = Nothing
safeHead (x:_) = Just x
```

Sebenarnya, fungsi tersebut sudah terdefinisi di paket [`safe`] (http://hackage.haskell.org/package/safe).

Mengapa ini ide yang bagus?

1. `safeHead` tak akan pernah *crash*.
2. Tipe `safeHead` memperjelas bahwa fungsi tersebut bisa gagal pada beberapa input.
3. Sistem tipe (*type system*) memastikan pengguna `safeHead` harus selalu mengecek tipe hasilnya untuk melihat apakah nilai yang didapat atau `Nothing`.

Sebenarnya, `safeHead` masih "parsial"; tapi kita telah mencatat keparsialannya ke sistem tipe sehingga aman. Tujuannya ialah membuat tipe yang ada bisa menjelaskan sifat fungsi sebaik mungkin.

Oke, tapi bagaimana jika kita tahu kita akan memakai `head` di mana kita *terjamin* untuk memiliki list yang tidak kosong? Dalam situasi tersebut, akan sangat mengganggu untuk mendapatkan hasil berupa `Maybe a` karena kita harus bekerja lebih untuk mengatasi kasus yang kita "tahu" tak akan mungkin terjadi.

Jawabannya, jika ada kondisi yang *terjamin*, maka tipe harus merefleksikan jaminan tersebut! Lalu *compiler* bisa menjamin hal tersebut untuk kalian. Sebagai contoh:

```
data NonEmptyList a = NEL a [a]

nelToList :: NonEmptyList a -> [a]
nelToList (NEL x xs) = x:xs

listToNel :: [a] -> Maybe (NonEmptyList a)
listToNel []     = Nothing
listToNel (x:xs) = Just $ NEL x xs

headNEL :: NonEmptyList a -> a
headNEL (NEL a _) = a

tailNEL :: NonEmptyList a -> [a]
tailNEL (NEL _ as) = as
```

Kalian mungkin berpikir melakukan hal seperti ini hanyalah untuk orang bodoh yang bukan jenius seperti kalian. Tentu, *kalian* tak mungkin membuat kesalahan seperti memberikan list kosong ke fungsi yang mengharapkan list yang tidak kosong. Ya kan? Ada orang bodoh yang terlibat, tapi bukan yang kalian kira.

Higher-order programming dan *type inference*

Bacaan tambahan:

- "*Learn You a Haskell for Great Good*", "*Higher-Order Functions*" (Bab 5 di buku; Bab 6 online)

Fungsi anonim

Andai kita ingin menulis sebuah fungsi

```
lebihDari100 :: [Integer] -> [Integer]
```

yang menyaring `Integer` yang lebih besar dari 100. Sebagai contoh,

`lebihDari100 [1,9,349,6,907,98,105] = [349,907,105].`

Kita bisa membuatnya begini:

```
ld100 :: Integer -> Bool
ld100 x = x > 100

lebihDari100 :: [Integer] -> [Integer]
lebihDari100 xs = filter ld100 xs
```

Akan tetapi, cukup merepotkan untuk menamakan `ld100`, karena mungkin kita tak akan menggunakannya lagi. Untuk itu, kita bisa menggunakan fungsi anonim (*anonymous function*), yang juga dikenal sebagai abstraksi lambda (*lambda abstraction*):

```
lebihDari100_2 :: [Integer] -> [Integer]
lebihDari100_2 xs = filter (\x -> x > 100) xs
```

`\x -> x > 100` (*backslash* dianggap menyerupai lambda tanpa kaki yang pendek) adalah sebuah fungsi yang menerima sebuah argument `x` dan mengembalikan apakah `x` lebih besar dari 100.

Abstraksi lambda juga bisa memiliki beberapa argumen. Contohnya:

```
Prelude> (\x y z -> [x,2*y,3*z]) 5 6 3
[5,12,9]
```

Akan tetapi, untuk `lebihDari100`, ada cara lebih baik untuk menulisnya. Yaitu tanpa abstraksi lambda:

```
lebihDari100_3 :: [Integer] -> [Integer]
lebihDari100_3 xs = filter (>100) xs
```

`(>100)` adalah sebuah *operator section*. Jika `?` adalah sebuah operator, maka `(?y)` sama saja dengan fungsi `\x -> x ? y`, dan `(y?)` sama dengan `\x -> y ? x`. Dengan kata lain, *operator section* memungkinkan kita menerapkan sebagian (*partially apply*) operator ke salah satu di antara dua argumen. Yang kita dapat ialah suatu fungsi dengan satu argumen. Contohnya seperti ini:

```
Prelude> (>100) 102
True
Prelude> (100>) 102
False
Prelude> map (*6) [1..5]
[6,12,18,24,30]
```

Komposisi fungsi

Sebelum membaca lebih lanjut, bisakah kalian menulis suatu fungsi bertipe ini:

```
(b -> c) -> (a -> b) -> (a -> c)
```

?

Mari kita coba. Fungsi tersebut harus menerima dua argumen dan mengembalikan suatu fungsi. Masing-masing argumen juga berupa suatu fungsi.

```
foo f g = ...
```

`...` harus kita ganti dengan suatu fungsi bertipe `a -> c`. Kita bisa membuatnya dengan menggunakan abstraksi lambda:

```
foo f g = \x -> ...
```

`x` bertipe `a`, dan sekarang dalam `...` kita akan menulis ekspresi bertipe `c`. Kita punya fungsi `g` yang mengubah sebuah `a` menjadi sebuah `b`, dan fungsi `f` yang mengubah sebuah `b` menjadi sebuah `c`, jadi ini seharusnya benar:

```
foo :: (b -> c) -> (a -> b) -> (a -> c)
foo f g = \x -> f (g x)
```

(Kuis: mengapa tanda kurung diperlukan di `g x`?)

OK, jadi apa gunanya? Apakah `foo` berguna dalam suatu hal atau hanya sekedar latihan iseng dengan *types*?

Ternyata, `foo` disebut `(.)` yang berarti komposisi fungsi (*function composition*), yaitu, jika `f` dan `g` adalah fungsi, maka `f . g` adalah fungsi yang melakukan `g` lalu `f`.

Komposisi fungsi berguna untuk menulis kode yang ringkas dan elegan. Ini sangat cocok dengan gaya "*wholemeal*" di mana kita berpikir untuk menyusun transformasi *high level* yang berurutan untuk struktur data.

Sebagai contoh:

```
myTest :: [Integer] -> Bool
myTest xs = even (length (lebihDari100 xs))
```

Kita bisa menggantinya dengan:

```
myTest' :: [Integer] -> Bool
myTest' = even . length . lebihDari100
```

Versi ini lebih jelas: `myTest` hanyalah "pipa" yang terdiri dari tiga fungsi yang lebih kecil. Ini juga menjelaskan mengapa komposisi fungsi terlihat "terbalik": karena aplikasi fungsi juga terbalik! Karena kita membaca dari kiri ke kanan, lebih masuk akal untuk membayangkan nilai juga mengalir dari kiri ke kanan. Tapi dengan begitu kita seharusnya menulis \((x)f \) untuk menggambarkan pemberian nilai \(x\) sebagai input ke fungsi \(f\). Karena Alexis Claude Clairaut dan Euler, kita terjebak dengan notasi terbalik sejak 1734.

Mari lihat lebih dekat tipe dari `(.)`. Tanyakan tipenya ke `ghci`, kita akan mendapatkan:

```
Prelude> :t (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Tunggu. Apa yang terjadi? Ke mana tanda kurung di sekitar `(a -> c)`?

Currying dan aplikasi parsial

Ingat kalau tipe dari fungsi yang memiliki beberapa argumen terlihat aneh dengan ekstra anak panah? Seperti:

```
f :: Int -> Int -> Int
f x y = 2*x + y
```

Sebelumnya dikatakan ada alasan yang indah dan mendalam tentangnya, dan sekarang saatnya untuk diketahui: *semua fungsi di Haskell hanya menerima satu argumen*. Loh?! Bukankah fungsi `f` di atas menerima dua argumen? Sebenarnya tidak: fungsi `f` menerima satu argumen (sebuah `Int`) dan *mengembalikan sebuah fungsi* (bertipe `Int -> Int`) yang menerima satu argumen dan mengembalikan hasil akhir. Jadi kita menyatakan tipe dari `f`'s seperti ini:

```
f' :: Int -> (Int -> Int)
f' x y = 2*x + y
```

Perhatikan bahwa anak panah fungsi *asosiatif ke kanan*, dengan kata lain, `W -> X -> Y -> Z` sama dengan `W -> (X -> (Y -> Z))`. Kita bisa menambahkan atau mengurangi tanda kurung di sekitar anak panah *top level* paling kanan di notasi tipe.

Aplikasi fungsi, kebalikannya, bersifat asosiatif ke *kiri*. `f 3 2` sama dengan `(f 3) 2`. Masuk akal mengingat tentang `f` sebenarnya menerima satu argumen dan mengembalikan sebuah fungsi: kita terapkan `f` ke sebuah argumen `3`, yang mengembalikan fungsi bertipe `Int -> Int`, yaitu sebuah fungsi yang menerima sebuah `Int` dan menambahkan 6 ke argumen tersebut. Lalu kita terapkan fungsi tersebut ke argumen `2` dengan menuliskan `(f 3) 2`, yang menghasilkan sebuah `Int`. Karena aplikasi fungsi asosiatif ke kiri, kita bisa menulis `(f 3) 2` sebagai `f 3 2`. Dengan begini, kita mendapatkan notasi yang bagus untuk `f` sebagai fungsi yang memiliki beberapa argumen.

Abstraksi lambda untuk fungsi dengan beberapa argumen

```
\x y z -> ...
```

hanyalah versi lain (*syntax sugar*) dari

```
\x -> (\y -> (\z -> ...)).
```

Sebaliknya, definisi fungsi

```
f x y z = ...
```

hanyalah versi lain dari

`f = \x -> (\y -> (\z -> ...))`.

Sebagai contoh, kita bisa menulis ulang komposisi fungsi sebelumnya di atas dengan memindahkan `\x -> ...` dari sisi kanan `=` ke sisi kiri:

```
comp :: (b -> c) -> (a -> b) -> a -> c
comp f g x = f (g x)
```

Ide untuk merepresentasikan fungsi dengan beberapa arguman sebagai fungsi satu argumen disebut *currying*, mengambil nama matematikawan dan ahli logika Inggris Haskell Curry (orang yang juga mengispirasi penamaan Haskell). Curry (1900-1982) banyak menghabiskan hidupnya di Penn State, dan juga membantu mengerjakan ENIAC di UPenn. Ide representasi tersebut sebenarnya ditemukan oleh Moses Schönfinkel, jadi mungkin kita layak menyebutnya *schönfinkeling*. Curry sendiri mengatributkan ide tersebut ke Schönfinkel, akan tetapi orang lain sudah terlanjur menyebutnya *currying*.

Jika kita ingin benar-benar menyatakan sebuah fungsi dengan beberapa argumen, kita bisa memberikan satu argumen berupa *tuple*. Fungsi ini

```
f'' :: (Int,Int) -> Int
f'' (x,y) = 2*x + y
```

bisa dibayangkan menerima dua argumen, meskip sebenarnya hanya menerima satu argumen berupa sebuah "pair". Untuk mengubah antar dua representasi fungsi yang menerima dua argumen, *standard library* menyediakan fungsi `curry` dan `uncurry` yang didefinisikan sebagai berikut (dengan nama berbeda):

```
schönfinkel :: ((a,b) -> c) -> a -> b -> c
schönfinkel f x y = f (x,y)

unschönfinkel :: (a -> b -> c) -> (a,b) -> c
unschönfinkel f (x,y) = f x y
```

`uncurry` berguna jika kita ingin menerapkan sebuah fungsi ke sebuah *pair*. Sebagai contoh:

```
Prelude> uncurry (+) (2,3)
5
```

Aplikasi Parsial

Fakta bahwa fungsi di Haskell *curried* membuat aplikasi parsial (atau penerapan sebagian, *partial application*) menjadi mudah. Aplikasi parsial adalah kita mengambil sebuah fungsi yang menerima beberapa argumen dan menerapkannya ke *sebagian* dari argumen-argumen tersebut, dan mendapatkan sebuah fungsi yang menerima argumen-argumen sisanya. Tapi seperti yang kita baru saja saksikan, di Haskell *tidak ada* fungsi dengan beberapa argumen! Tiap fungsi bisa diterapkan sebagian ke argumen pertama (dan satu-satunya), menghasilkan sebuah fungsi yang menerima argumen-argumen sisanya.

Perhatikan bahwa tidak mudah di Haskell untuk melakukan aplikasi parsial ke argumen selain yang pertama. Pengecualian untuk operator *infix*, yang kita sudah lihat bisa di aplikasikan sebagian ke salah satu dari dua argumennya menggunakan sebuah *operator section*. Pada prakteknya ini bukanlah suatu masalah. Seni menentukan urutan argumen untuk membuat aplikasi parsial benar-benar berguna: argumen harus diurutkan dari yang *kurang bervariasi* sampai ke yang *paling bervariasi*. Artinya, argumen yang akan sering sama harus diletakkan lebih dulu (paling depan), dan argumen yang akan sering berbeda diletakkan belakangan.

Wholemeal programming

Mari kita rangkum semua yang sudah kita pelajari ke dalam sebuah contoh yang menunjukkan kekuatan pemrograman bergaya *wholemeal*. Fungsi `foobar` didefinisikan sebagai berikut:

```
foobar :: [Integer] -> Integer
foobar []     = 0
foobar (x:xs)
  | x > 3     = (7*x + 2) + foobar xs
  | otherwise = foobar xs
```

Fungsi tersebut terlihat cukup jelas, akan tetapi bukan gaya Haskell yang bagus. Masalahnya adalah:

- melakukan terlalu banyak hal sekaligus; dan
- bekerja di level yang terlalu rendah.

Daripada memikirkan apa yang mau kita lakukan ke tiap elemen, kita lebih baik memikirkan transformasi bertahap ke semua elemen dengan menggunakan pola rekursi yang kita ketahui. Berikut adalah implementasi `foobar` yang lebih idiomatis:

```
foobar' :: [Integer] -> Integer
foobar' = sum . map (\x -> 7*x + 2) . filter (>3)
```

Kali ini `foobar` didefinisikan sebagai "pipa" dari tiga fungsi: pertama, kita membuang semua elemen yang tidak lebih besar dari tiga; lalu kita terapkan operasi aritmetik ke tiap elemen sisanya; akhirnya, kita jumlahkan semuanya.

Perhatikan `map` dan `filter` diaplikasikan parsial. Sebagai contoh, tipe `filter` adalah

```
(a -> Bool) -> [a] -> [a]
```

Mengaplikasikannya ke `(>3)` (yang bertipe `Integer -> Bool`) menghasilkan fungsi bertipe `[Integer] -> [Integer]`, yang merupakan fungsi yang tepat untuk digabungkan dengan fungsi lain yang menerima `[Integer]`.

Gaya pemrograman seperti ini yang mendefinisikan fungsi tanpa referensi ke argumennya (atau bisa juga dibilang menyatakan *definisi* sebuah fungsi ketimbang apa yang fungsi tersebut *lakukan*) disebut sebagai "point-free". Gaya seperti ini

lebih enak dilihat. Beberapa orang bahkan sampai berkata bahwa gaya "point-free" harus sebisa mungkin dipakai, meski tentu jika terlalu jauh malah akan membingungkan. `lambdabot` di kanal IRC `#haskell` memiliki perintah `@pl` untuk mengubah fungsi menjadi bergaya "point-free". Contohnya:

```
@pl \f g x y -> f (x ++ g x) (g y)
join . ((flip . ((.) .)) .) . (. ap (++)) . (.)
```

Terlihat untuk kasus ini, fungsi malah menjadi sulit dibaca.

Folds

Satu lagi pola rekursi yang akan dibahas: *folds* (terjemahan: lipat). Berikut adalah beberapa fungsi yang memiliki pola yang serupa: semuanya "menggabungkan" elemen-elemen yang dimiliki menjadi satu jawaban.

```
sum' :: [Integer] -> Integer
sum' []     = 0
sum' (x:xs) = x + sum' xs

product' :: [Integer] -> Integer
product' [] = 1
product' (x:xs) = x * product' xs

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

Apa yang sama dari ketiganya dan apa yang berbeda? Kita akan memisahkan bagian yang berbeda dengan menggunakan fungsi "*higher-order*".

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

Perhatikan bahwa `fold` sebenarnya mengganti `[]` dengan `z`, dan `(:)` dengan `f`. Dengan kata lain:

```
fold f z [a,b,c] == a `f` (b `f` (c `f` z))
```

(Jika kalian membayangkan `fold` dengan perspekif tersebut, kalian mungkin bisa menemukan cara untuk menggeneralisir `fold` ke tipe data lain selain list)

Mari tulis ulang `sum'`, `product'`, dan `length'` dengan menggunakan `fold`:

```
sum''     = fold 0 (+)
product'' = fold 1 (*)
length''  = fold 0 (\_ s -> 1 + s)
```

(Selain `(\_ s -> 1 + s)` kita juga bisa menulisnya `(\_ -> (1+))` atau bahkan `(const (1+))`.)

`fold` sudah tersedia di `Prelude`, dengan nama [`foldr`] (https://downloads.haskell.org/~ghc/latest/docs/html/li
4.8.1.0/Prelude.html#v:foldr) Argumen-argumen untuk fungsi `foldr` sedikit berbeda urutannya dengan `fold` tadi, meski bekerja dengan cara yang sama. Berikut adalah beberapa fungsi dari `Prelude` yang didefinisikan dengan `foldr`:

- `length ::          [a] -> Int`
- `sum :: Num a => [a] -> a`
- `product :: Num a => [a] -> a`
- `and ::          [Bool] -> Bool`
- `or ::          [Bool] -> Bool`
- `any :: (a -> Bool) -> [a] -> Bool`
- `all :: (a -> Bool) -> [a] -> Bool`

Ada juga `foldl` yang "melipat (*fold*) dari kiri",

```
foldr f z [a,b,c] == a `f` (b `f` (c `f` z))
foldl f z [a,b,c] == ((z `f` a) `f` b) `f` c
```

Pada umumnya, kita sebaiknya menggunakan `foldl'` dari `Data.List` , yang sama seperti `foldl` tapi lebih efisien.

Polimorfisme lanjutan dan *type classes*

Polimorfisme (*polymorphism*) di Haskell dikenal sebagai polimorfisme parametrik (*parametric polymorphism*). Ini berarti fungsi polimorfis harus bekerja secara *seragam* terhadap tipe input apapun. Hal tersebut memiliki implikasi yang menarik bagi pemrogram maupun pengguna fungsi polimorfis tersebut.

Parametrisitas (*parametricity*)

Perhatikan tipe

```
a -> a -> a
```

Ingat bahwa `a` adalah sebuah variable tipe yang bisa berarti tipe apapun. Fungsi seperti apa yang bertipe seperti itu?

Mari kita coba begini

```
f :: a -> a -> a
f x y = x && y
```

Ternyata tidak bisa. *Syntax*-nya valid, tetapi tidak *type check* (tipenya tidak cocok). Pesan *error*-nya:

```
2012-02-09.lhs:37:16:
    Couldn't match type `a' with `Bool'
      `a' is a rigid type variable bound by
          the type signature for f :: a -> a -> a at 2012-02-09.lhs:37:3
    In the second argument of `(&&)', namely `y'
    In the expression: x && y
    In an equation for `f': f x y = x && y
```

Hal ini dikarenakan *pemanggil* fungsi polimorfis yang menentukan tipenya. Implementasi di atas mencoba memilih tipe yang spesifik (`Bool`), tapi tetap ada kemungkinan untuk bertipe `String`, `Int`, atau lainnya. Bahkan, bisa juga tipe baru buatan orang lain yang didefinisikan dengan `f`. Kita tidak mungkin mengetahui tipe apa yang akan kita terima.

Dengan kata lain, tipe seperti

```
a -> a -> a
```

bisa dibaca sebagai jaminan kalau fungsi bertipe demikian akan bekerja dengan benar, tidak peduli tipe apapun yang akan diberikan.

Impelementasi yang lain bisa seperti

```
f a1 a2 = case (typeOf a1) of
            Int  -> a1 + a2
            Bool -> a1 && a2
            _    -> a1
```

di mana `f` memberikan perlakuan yang berbeda sesuai dengan tipe. Sama seperti halnya di Java:

```
class AdHoc {

    public static Object f(Object a1, Object a2) {
        if (a1 instanceof Integer && a2 instanceof Integer) {
            return (Integer)a1 + (Integer)a2;
        } else if (a1 instanceof Boolean && a2 instanceof Boolean) {
            return (Boolean)a1 && (Boolean)a2;
        } else {
            return a1;
        }
    }

    public static void main (String[] args) {
        System.out.println(f(1,3));
        System.out.println(f(true, false));
        System.out.println(f("hello", "there"));
    }

}
```

```
[byorgey@LVN513-9:~/tmp]$ javac Adhoc.java && java AdHoc
4
false
hello
```

Hal ini tidak bisa dilakukan di Haskell. Haskell tidak memiliki operator seperti `instanceof` di Java, dan tidak mungkin untuk menanyakan tipe dari suatu nilai. Salah satu alasannya ialah tipe di Haskell dihapus oleh *compiler* setelah dicek: tidak ada informasi tipe di saat *runtime*! Akan tetapi, ada pula alasan lainnya.

Gaya polimorfisme seperti ini dikenal sebagai polimorfisme parametrik. Fungsi seperti `f :: a -> a -> a` dikatakan parametrik untuk tipe `a`. Di sini, parametrik hanyalah sebutan untuk "bekerja secara seragam untuk semua tipe yang diberikan". Di Java, gaya polimorfisme seperti ini disediakan oleh *generics* (yang terinspirasi dari Haskell: salah satu disainer Haskell, Philip Wadler, adalah salah satu pengembang kunci Java generics).

Jadi fungsi apa yang mungkin bertipe seperti ini? Ternyata hanya ada dua!

```
f1 :: a -> a -> a
f1 x y = x

f2 :: a -> a -> a
f2 x y = y
```

Jadi tipe `a -> a -> a` cukup banyak mengandung informasi.

Mari bermain *game* parametrisitas! Perhatikan tipe-tipe berikut. Untuk tiap tipe, tentukan perilaku yang mungkin dimiliki.

- `a -> a`
- `a -> b`
- `a -> b -> a`
- `[a] -> [a]`
- `(b -> c) -> (a -> b) -> (a -> c)`
- `(a -> a) -> a -> a`

Dua pandangan tentang parametrisitas

Sebagai penulis implementasi fungsi, ini terasa membatasi. Terlebih jika kita sudah terbiasa dengan bahasa yang memiliki hal seperti `instanceof` di Java. "Koq gitu? Kenapa tidak boleh melakukan X?"

Akan tetapi, ada pandangan berbeda. Sebagai *pengguna* dari fungsi polimorfis, parametrisitas bukan berarti *larangan*, tapi lebih sebagai *jaminan*. Pada umumnya, *tools* akan lebih mudah digunakan dan dibuktikan jika *tools*-nya memberikan jaminan tentang *sifatnya*. Parametrisitas adalah salah satu alasan

mengapa dengan hanya melihat tipe dari sebuah fungsi Haskell bisa memberitahu kalian banyak hal tentang fungsi tersebut.

OK, tapi terkadang kita perlu menentukan sesuatu berdasarkan tipe. Contohnya, penjumlahan. Kita telah melihat bahwa penjumlahan adalah polimorfis (berlaku untuk `Int`, `Integer`, dan `Double`) tapi tentu fungsi tersebut perlu mengetahui tipe yang sedang dijumlahkan untuk mengetahui apa yang harus dilakukan. Menjumlahkan dua `Integer` tentu berbeda dengan menjumlahkan dua `Double`. Jadi bagaimana? Sihir?

Ternyata tidak! Dan di Haskell, kita bisa menentukan apa yang harus dilakukan berdasarkan tipe, meski berbeda dari yang sebelumnya kita bayangkan. Mari mulai dengan melihat tipe dari `(+)`:

```
Prelude> :t (+)
(+) :: Num a => a -> a -> a
```

Hmm, apa yang `Num a =>` lakukan di sini? Sebenarnya, `(+)` bukanlah satu-satunya fungsi standar yang memiliki *double-arrow* di tipenya. Berikut contoh-contoh lainnya:

```
(==) :: Eq a   => a -> a -> Bool
(<)  :: Ord a  => a -> a -> Bool
show :: Show a => a -> String
```

Apakah yang terjadi?

Type class

`Num`, `Eq`, `Ord`, and `Show` adalah *type class*, dan kita sebut `(==)`, `(<)`, dan `(+)` "*type-class polymorphic*". Secara intuisi, *type class* bisa dianggap sebagai himpunan dari tipe yang memiliki beberapa operasi yang terdefinisikan untuk mereka. Fungsi *type class polymorphic* hanya menerima tipe yang merupakan anggota (*instances*) dari *type class* tersebut. Sebagai contoh, mari lihat detil dari *type class* `Eq`.

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

Kita bisa membacanya seperti ini: `Eq` dideklarasikan sebagai *type class* dengan satu argumen, `a`. Tiap tipe `a` yang ingin menjadi anggota (*instance*) dari `Eq` harus mendefinisikan dua fungsi, `(==)` dan `(/=)`, dengan tipe yang tercantum. Misalnya, untuk membuat `Int` menjadi anggota `Eq` kita harus mendefinisikan `(==) :: Int -> Int -> Bool` dan `(/=) :: Int -> Int -> Bool`. (Tidak perlu tentunya, karena *Prelude* sudah mendefinisikan `Int` sebagai anggota `Eq`.)

Mari lihat tipe dari `(==)` lagi:

```
(==) :: Eq a => a -> a -> Bool
```

`Eq a` sebelum `=>` adalah sebuah *type class constraint*. Kita bisa menyebutnya: untuk semua tipe `a`, *selama* `a` *adalah anggota* `Eq`, `(==)` bisa menerima dua nilai bertipe `a` dan mengembalikan sebuah `Bool`. Pemanggilan `(==)` kepada tipe yang bukan anggota `Eq` mengakibatkan *type error*.

Jika tipe polimorfis adalah suatu jaminan fungsi akan bekerja dengan apapun tipe input yang pemanggil berikan, fungsi polimorfis *type class* adalah jaminan *terbatas* bahwa fungsi akan bekerja dengan apapun tipe yang diberikan *selama* tipe tersebut anggota dari *type class* tersebut.

Hal yang perlu diingat, ketika `(==)` (atau *method type class* lainnya) digunakan, *compiler* menggunakan *type inference* (pada argumen-argumennya) untuk mencari tahu *implementasi* `(==)` *yang mana yang akan dipilih*. Mirip dengan *overloaded method* di bahasa seperti Java.

Untuk mengerti lebih jauh, mari buat tipe sendiri dan jadikan sebagai anggota `Eq`.

```
data Foo = F Int | G Char

instance Eq Foo where
  (F i1) == (F i2) = i1 == i2
  (G c1) == (G c2) = c1 == c2
  _ == _ = False

  foo1 /= foo2 = not (foo1 == foo2)
```

Cukup merepotkan untuk mendefinisikan `(==)` and `(/=)`. Sebenarnnya *type class* bisa memberikan implementasi *default* untuk *method* dalam bentuk *method* lainnya. Ini akan digunakan jika ada anggota yang tidak meng-*override* implementasi *default*. Kita bisa mendeklarasikan `Eq` seperti ini:

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  x /= y = not (x == y)
```

Sekarang yang mendeklarasikan anggota `Eq` cukup memberikan implementasi dari `(==)` saja, dan akan langsung mendapatkan `(/=)`. Tapi jika ada alasan tertentu untuk mengganti implementasi *default* dari `(/=)`, kita tetap bisa melakukannya.

Sebenarnya, *class* `Eq` dideklarasikan seperti ini:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x == y = not (x /= y)
  x /= y = not (x == y)
```

Ini berarti tiap kita membuat anggota dari `Eq`, kita hanya perlu mendefinisikan *salah satu dari* `(==)` atau `(/=)`, manapun yang lebih praktis; yang lainnya akan

terdefinisi secara otomatis dalam bentuk yang kita definisikan. (Hati-hati: jika salah satu tidak terdefinisi, kita akan mendapatkan *infinite recursion*!)

Ternyata, `Eq` (beserta beberapa *type class* standar lainnya) istimewa: GHC bisa membuat anggota `Eq` secara otomatis untuk kita.

```
data Foo' = F' Int | G' Char
  deriving (Eq, Ord, Show)
```

Di atas, kita memberitahu GHC untuk menjadikan tipe `Foo'` milik kita anggota dari *type class* `Eq`, `Ord`, dan `Show`.

Type class dan interface di Java

Type class serupa dengan *interface* di Java. Keduanya mendefinisikan himpunan tipe/*class* yang mengimplementasikan beberapa operasi yang spesifik. Akan tetapi, ada dua hal penting yang menunjukkan kalau *type class* itu lebih umum daripada *interface* di Java:

1. Ketika sebuah *class* didefinisikan di Java, *interface* yang dimiliki juga dideklarasikan. Sedangkan anggota (*instance* dari) *type class* dideklarasikan di tempat berbeda dari tipenya sendiri, bahkan bisa diletakkan di modul berbeda.

2. Tipe untuk *method* di *type class* bisa lebih umum dan fleksibel ketimbang yang bisa diberikan di *method interface* di Java. Terlebih jika mengingat *type class* bisa menerima beberapa argumen. Sebagai contoh:

   ```
   class Blerg a b where
     blerg :: a -> b -> Bool
   ```

   Penggunaan `blerg` berarti melakukan *multiple dispatch*: implementasi `blerg` yang dipilih *compiler* bergantung kepada *kedua* tipe `a` dan `b`. Di Java hal ini bukanlah hal yang mudah.

   type class di Haskell juga bisa menerima *method* biner (atau ternari, dst) dengan mudah, seperti

   ```
   class Num a where
     (+) :: a -> a -> a
     ...
   ```

   Di Java hal ini bukanlah sesuatu yang mudah: satu contoh, salah satu dari dua argumen haruslah "istimewa" yaitu yang menerima *method* `(+)`. Hal ini mengakibatkan funsi menjadi asimetris dan canggung. Lebih jauh lagi, karena *subtyping* di Java, *interface* yang menerima dua argumen *tidak* menjamin kalau dua argumen tersebut bertipe sama, yang mengakibatkan implementasi operator biner seperti `(+)` menjadi canggung (biasanya melibatkan semacam pengecekan saat *runtime*).

Type Class standar

Berikut adalah beberapa *type class* standar yang perlu kalian ketahui:

- Ord adalah untuk tipe-tipe yang tiap elemennya bisa diurutkan secara total (*totally ordered*). Dengan kata lain, tiap elemennya bisa dibandingkan satu sama lain untuk dilihat mana yang lebih kecil dari yang lain. Ini menyediakan operasi perbandingan seperti `(<)` dan `(<=)`, serta fungsi `compare`.
- Num adalah untuk tipe "numerik", yang mendukung hal-hal seperti penjumlahan, pengurangan, dan perkalian. Satu hal yang penting: literal *integer* adalah polimorfis *type class*:

  ```
  Prelude> :t 5
  5 :: Num a => a
  ```

  Ini berarti literal seperti `5` bisa digunakan sebagai `Int`, `Integer`, `Double`,atau tipe apapun yang merupakan anggota (*instance*) dari `Num` (seperti `Rational`, `Complex Double`, atau tipe kalian sendiri...)
- Show mendefinisikan *method* `show`, yang mengubah nilai menjadi `String`.
- Read adalah dual (catatan penerjemah: istilah dari Category Theory, yang bisa diartikan kurang lebih sebagai "kebalikan") dari `Show`.
- Integral mewakili semua tipe bilangan bulat seperti `Int` dan `Integer`.

Contoh type class

Sebagai contoh untuk membuat *type class* sendiri:

```
class Listable a where
  toList :: a -> [Int]
```

Kita bisa anggap `Listable` sebagai himpunan sesuatu yang bisa diubah menjadi sebuah list yang berisi `Int`. Perhatikan tipe dari `toList`:

```
toList :: Listable a => a -> [Int]
```

Mari kita buat anggota (*instance*) dari `Listable`. Pertama, sebuah `Int` bisa diubah menjadi sebuah `[Int]` hanya dengan menciptakan list singleton, begitu pula dengan `Bool`, dengan mengubah `True` menjadi `1` dan `False` menjadi `0`:

```
instance Listable Int where
  -- toList :: Int -> [Int]
  toList x = [x]

instance Listable Bool where
  toList True  = [1]
  toList False = [0]
```

Kita tidak perlu repot untuk mengubah sebuah list `Int` ke list `Int`:

```
instance Listable [Int] where
  toList = id
```

Terakhir, kita ubah sebuah *binary tree* menjadi list dengan *flattening*:

```haskell
data Tree a = Empty | Node a (Tree a) (Tree a)

instance Listable (Tree Int) where
  toList Empty        = []
  toList (Node x l r) = toList l ++ [x] ++ toList r
```

Jika kita membuat fungsi baru dengan menggunakan `toList`, fungsi tersebut akan mendapatkan `constraint Listable` juga. Sebagai contoh:

```haskell
-- to compute sumL, first convert to a list of Ints, then sum
sumL x = sum (toList x)
```

`ghci` akan memberitahukan kita bahwa tipe dari `sumL` adalah

```haskell
sumL :: Listable a => a -> Int
```

Masuk akal karena `sumL` hanya akan bekerja untuk tipe yang merupakan anggota dari `Listable`, karena menggunakan `toList`. Bagaimana dengan yang ini?

```haskell
foo x y = sum (toList x) == sum (toList y) || x < y
```

`ghci` memberitahukan bahwa tipe dari `foo` adalah

```haskell
foo :: (Listable a, Ord a) => a -> a -> Bool
```

`foo` bekerja untuk tipe yang merupakan anggota dari `Listable` dan `Ord` , karena menggunakan `toList` dan perbandingan argumen-argumennya.

Sebagai contoh terakhir yang lebih rumit:

```haskell
instance (Listable a, Listable b) => Listable (a,b) where
  toList (x,y) = toList x ++ toList y
```

Perhatikan bahwa kita bisa meletakkan *type class constraint* pada tipe anggota (*instance*) dan juga pada tipe fungsi. Contoh tersebut menunjukkan bahwa sebuah *pair* bertipe `(a,b)` adalah anggota dari `Listable` selama `a` dan `b` juga anggota dari `Listable`. Lalu kita bisa menggunakan `toList` untuk `a` dan `b` di dalam definisi `toList` untuk *pair*. Definisi ini tidaklah rekursif! Versi `toList` untuk *pair* memanggil versi `toList` yang *berbeda*, bukan yang didefinisikan di dirinya sendiri.

Evaluasi *lazy*

Bacaan tambahan:

- foldr foldl foldl' dari Haskell wiki

Di hari pertama kelas, saya mengatakan bahwa Haskell bersifat *lazy* (translasi: enggan, malas), dan berjanji akan menjelaskan lebih lanjut apa artinya di lain waktu. Sekaranglah saatnya!

Evaluasi *strict*

Sebelum membahas evaluasi *lazy* (literal: malas, enggan), kita akan melihat beberapa contoh dari kebalikannya, evalusi *strict* (literal: tegas, kaku)

Pada evaluasi *strict*, argumen fungsi akan terevaluasi *sebelum* diberikan ke fungsi. Sebagai contoh, misalkan kita telah mendefinisikan

```
f x y = x + 2
```

Di bahasa yang *strict*, `f 5 (29^35792)` pertama kali akan mengevaluasi `5` (sudah selesai) dan `29^35792` (yang harus dikerjakan terlebih dulu) sebelum memberikan hasil-hasil tersebut ke `f`.

Tentu dalam contoh tersebut hal ini terkesan bodoh, karena `f` mengabaikan argumen kedua. Sehingga seluruh usaha untuk menghitung `29^35792` terbuang percuma. Jadi kenapa kita memerlukannya?

Keuntungan dari evaluasi *strict* ialah kemudahan untuk menebak *kapan* dan *urutan* sesuatu akan terjadi. Biasanya bahasa yang *strict* menjelaskan urutan evaluasi argumen-argumen dari fungsi (dari kiri ke kanan misalnya).

Misalkan kita menulis ini di Java

```
f (release_monkeys(), increment_counter())
```

Kita tahu bahwa kera-kera akan dibebaskan, lalu *counter* akan bertambah, dan hasil-hasil tersebut akan diberikan ke `f`. Tidak peduli apakah `f` akan menggunakan hasil-hasil tersebut atau tidak.

Jika hal-hal berikut:

- Evaluasi atau tidaknya kedua argumen secara independen
- Urutan evaluasi kedua argumen tersebut

bergantung kepada apakah `f` akan menggunakan hasilnya, tentunya akan sangat membingungkan. Ketika "efek samping" seperti ini diperbolehkan, kita perlu evaluasi *strict*.

Efek samping dan *purity*

Jadi yang sebenarnya bermasalah adalah keberadaan *efek samping* (*side effect*). Yang dimaksud dengan efek samping ialah "apapun yang dapat menyebabkan evaluasi sebuah ekspresi berinteraksi dengan sesuatu di luar ekspresi itu sendiri". Akar permasalahannya adalah interaksi dengan dunia luar tersebut sensitif terhadap waktu. Sebagai contoh:

- Mengubah variabel global — ini bermasalah karena bisa mempengaruhi evaluasi ekspresi-ekspresi lain

- Mencetak ke layar — ini bermasalah karena bisa saja ini diperlukan dalam urutan tertentu terhadap ekspresi lain yang juga melakukan pencetakan ke layar
- Membaca berkas atau jaringan — ini bermasalah karena isi dari berkas bisa mempengaruhi hasil evaluasi ekspresi

Seperti yang kita lihat, evaluasi *lazy* membuat sulit untuk mengetahui kapan evaluasi terjadi, sehingga menyertakan efek samping ke dalam bahasa yang *lazy* menjadi sangat tidak intuitif. Secara sejarah, inilah alasan mengapa Haskell *pure* (literal: murni). Pada awalnnya, perancang Haskell berniat untuk membuat bahasa yang *lazy*, dan kemudian dengan cepat menyadari bahwa hal tersebut mustahil jika tidak melarang efek samping.

Tapi.. sebuah bahasa tanpa efek samping tidaklah begitu berguna. Kalian hanya akan bisa me-*load* program di *interpreter* dan mengevaluasi ekspresi-ekspresi. Kalian tidak akan bisa mendapatkan masukan dari pengguna, atau mencetak ke layar, atau membaca berkas. Tantangan yang dihadapi para perancang Haskell adalah menyediakan cara untuk mengizinkan efek-efek tersebut dengan terkontrol dan tidak mengganggu *purity* dari bahasa itu sendiri. Akhirnya mereka berhasil menyediakannya (bernama `IO` monad) yang akan kita bahas dalam beberapa minggu kemudian.

Evaluasi *lazy*

Setelah pembahasan evaluasi *strict*, sekarang mari kita lihat seperti apa evaluasi *lazy*. Di dalam evaluasi *lazy*, evaluasi argumen-argumen fungsi "ditunda selama mungkin". Argumen-argumen tersebut tidak akan dievaluasi sampai benar-benar diperlukan. Ketika ekspresi diberikan ke fungsi sebagai argumen, ekspresi tersebut disimpan sebagai ekspresi yang belum dievaluasi (disebut "*thunk*", entah mengapa).

Sebagai contoh, ketika mengevaluasi `f 5 (29^35792)`, argumen kedua akan disimpan sebagai *thunk* tanpa ada komputasi dan `f` akan langsung dipanggil. Karena `f` tak pernah menggunakan argumen kedua, *thunk* tersebut akan dibuang oleh *garbage collector*.

Evaluasi dengan pencocokkan pola

Jadi kapankah kita perlu untuk mengevaluasi ekspresi? Contoh di atas mengkhususkan pada penggunaan argumen pada sebuah fungsi, tetapi itu bukanlah suatu perbedaan yang paling utama. Lihatlah contoh berikut:

```
f1 :: Maybe a -> [Maybe a]
f1 m = [m,m]

f2 :: Maybe a -> [a]
```

```
f2 Nothing  = []
f2 (Just x) = [x]
```

`f1` dan `f2` menggunakan argumen-argumen tetapi ada perbedaan besar di antara keduanya. Meskipun `f1` menggunakan argumen `m`, dia tidak perlu tahu apapun tentang argumen tersebut. `m` bisa merupakan ekspresi yang sepenuhnya tidak terevaluasi, dan letakkan ekspresi tersebut ke dalam list. Dengan kata lain, hasil dari `f1 e` tidak bergantung pada bentuk dari `e`.

Di sisi lain, `f2` perlu tahu sesuatu tentang argumennya untuk bekerja lebih jauh. Apakah dikonstruksi dengan `Nothing` atau `Just`? Untuk mengevaluasi `f2 e` kita perlu mengevaluasi `e` terlebih dahulu, karena hasil dari `f2` bergantung pada bentuk dari `e`.

Satu hal lagi yang patut diingat ialah *thunks* hanya dievaluasi **secukupnya**. Sebagai contoh, kita ingin mengevaluasi `f2 (safeHead [3^500, 49])`. `f2` akan memaksa evaluasi `safeHead [3^500, 49]`, yang akan menghasilkan `Just (3^500)`. Perhatikan bahwa `3^500` tidak dievaluasi, karena `safeHead` tidak memerlukannya, demikian juga dengan `f2`. Apakah `3^500` akan dievaluasi nantinya bergantung pada penggunaan hasil dari `f2`.

Slogannnya ialah "*evaluasi dengan pencocokkan pola*". Hal-hal yang perlu diingat:

- Ekspresi hanya dievaluasi ketika polanya cocok
- …dan hanya seperlunya saja!

Mari kita lihat contoh yang lebih menarik: kita akan mengevaluasi `take 3 (repeat 7)`. Sebagai referensi, berikut adalah definisi dari `repeat` dan `take`:

```
repeat :: a -> [a]
repeat x = x : repeat x

take :: Int -> [a] -> [a]
take n _      | n <= 0 =  []
take _ []              =  []
take n (x:xs)          =  x : take (n-1) xs
```

Evaluasi secara bertahap akan berjalan seperti ini:

```
  take 3 (repeat 7)
      { 3 <= 0 bernilai False, jadi kita lanjut ke klausa kedua, yang
    perlu mencocokkan dengan argumen kedua. Kita perlu ekspansi
    repeat 7 satu kali. }
= take 3 (7 : repeat 7)
      { klausa kedia tidak cocok tapi klausa ketiga cocok. Perhatikan
        bahwa (3-1) belum terevaluasi! }
= 7 : take (3-1) (repeat 7)
      { Untuk memutuskan pada klausa pertama, kita perlu mengecek (3-1)
        <= 0 yang memerlukan evaluasi (3-1). }
```

```
= 7 : take 2 (repeat 7)
      { 2 <= 0 bernilai False, jadi kita perlu ekspansi repeat 7 lagi. }
= 7 : take 2 (7 : repeat 7)
      { Sisanya serupa. }
= 7 : 7 : take (2-1) (repeat 7)
= 7 : 7 : take 1 (repeat 7)
= 7 : 7 : take 1 (7 : repeat 7)
= 7 : 7 : 7 : take (1-1) (repeat 7)
= 7 : 7 : 7 : take 0 (repeat 7)
= 7 : 7 : 7 : []
```

Catatan: Meski evaluasi bisa diimplementasikan seperti di atas, kebanyakan *compiler* Haskell melakukan hal lebih kompleks. Contohnya, GHC menggunakan teknik *graph reduction*, di mana ekspresi yang dievaluasi direpresentasikan dalam bentuk *graph* sehingga sub-ekspresi bisa berbagi *pointer* ke sub-ekspresi yang sama. Hal ini menjamin tak ada duplikasi pengerjaan. Sebagai contoh, jika `f x = [x,x]`, evaluasi `f (1+1)` hanya akan melakukan sekali penjumlahan karena sub-ekspresi `1+1` akan hanya ada satu, terbagi untuk dua buah `x`.

Konsekuensi

Laziness memiliki konsekuensi yang menarik, mengganggu, dan tersembunyi. Mari kita lihat beberapa di antaranya.

Purity (Kemurnian)

Seperti yang telah kita lihat, pemilihan evaluasi *lazy memaksa* kita juga untuk memilih *purity* (dengan asumsi kita tak mau menyyulitkan programer lain).

Memahami penggunaan ruang

Laziness tidak selalu memberikan keuntungan. Salah satu kekurangannya ialah terkadang kita akan kesulitan untuk mengetahui penggunaan ruang (*space*) di program. Perhatikan contoh berikut (yang terlihat biasa saja):

```
-- Fungsi di standard library foldl, disediakan sebagai referensi
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl _ z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

Mari kita lihat bagaimana evaluasi berjalan ketika kita mengevaluasi `foldl (+) 0 [1,2,3]` (yang akan menjumlahkan semua bilangan di list):

```
  foldl (+) 0 [1,2,3]
= foldl (+) (0+1) [2,3]
= foldl (+) ((0+1)+2) [3]
= foldl (+) (((0+1)+2)+3) []
= (((0+1)+2)+3)
```

```
= ((1+2)+3)
= (3+3)
= 6
```

Karena nilai dari akumulator tidak diperlukan hingga selesai rekursi seluruh list, akumulator membangun dirinya tiap langkah menjadi satu ekspresi besar yang belum terevaluasi `(((0+1)+2)+3)`. Pada akhirnya, ekspresi tersebut tereduksi menjadi sebuah nilai. Setidaknya, ada dua masalah dengan hal ini. Pertama, tidak efisien. Sebenarnya tidak perlu untuk memindahkan semua bilangan dari list ke sesuatu yang menyerupai list (*thunk akumulator*) sebelum menjumlahkan semuanya. Masalah kedua lebih tersembunyi. Evaluasi `(((0+1)+2)+3)` membutuhkan `3` dan `2` untuk dimasukkan ke `stack` terlebih dahulu sebelum menghitung `0+1`. Hal ini tidak masalah untuk contoh sederhana, tapi bermasalah untuk list yang sangat besar karena mungkin saja tidak ada ruang yang cukup untuk `stack` sehingga menyebabkan `stack overflow`.

Solusinya adalah dengan menggunakan fungsi `foldl'`, bukan `foldl`. `foldl'` mengevaluasi argumen kedua (akumulator) sebelum melanjutkan, sehingga *thunk* tidak bertambah besar.

```
  foldl' (+) 0 [1,2,3]
= foldl' (+) (0+1) [2,3]
= foldl' (+) 1 [2,3]
= foldl' (+) (1+2) [3]
= foldl' (+) 3 [3]
= foldl' (+) (3+3) []
= foldl' (+) 6 []
= 6
```

Terlihat bahwa `foldl'` melakukan penjumlahan secara langsung. Terkadang *laziness* menimbulkan masalah. Jika demikian kita harus membuat program kita menjadi kurang *lazy*.

Jika tertarik mempelajari bagaimana `foldl'` melakukannya, kalian membaca tentang `seq` di Haskell wiki.

Short-circuiting operators

Di beberapa bahasa (Java, C++) operator *boolean* `&&` dan `||` (AND dan OR) bersifat *short-circuiting*. Contohnya, jika evaluasi argumen pertama dari `&&` bernilai `False`, maka seluruh ekspresi akan langsung bernilai `False` tanpa menyentuh argumen kedua. Akan tetapi, perilaku ini harus diatur dalam standar Java dan C++ sebagai kasus khusus. Biasanya, di bahasa yang *strict*, kedua argumen akan dievaluasi sebelum pemanggilan fungsi. Jadi sifat *short-circuiting* dari `&&` dan `||` merupakan pengecualian dari semantik bahasa yang *strict*.

Di Haskell kita bisa mendefinisikan operator yang *short-circuiting* tanpa perlakuan istimewa. Bahkan sebenarnya `(&&)` dan `(||)` hanyalah fungsi pustaka biasa. Berikut definisi dari `(&&)`:

```
(&&) :: Bool -> Bool -> Bool
True  && x = x
False && _ = False
```

Perhatikan bahwa definisi dari `(&&)` tidak mencocokkan pola pada argumen kedua. Terlebih lagi jika argumen pertama bernilai `False`, argumen kedua akan diabaikan. Karena `(&&)` sama sekali tidak mencocokkan pola pada argumen kedua, maka fungsi ini juga bersifat *short-circuiting* sama seperti operator `&&` di Java atau C++.

Perhatikan bahwa `(&&)` bisa juga didefinisikan seperti ini:

```
(&&!) :: Bool -> Bool -> Bool
True  &&! True  = True
True  &&! False = False
False &&! True  = False
False &&! False = False
```

Meskipun versi tersebut menerima dan menghasilkan nilai yang sama seperti sebelumnya, terdapat perbedaan perilaku. sebagai contoh:

```
False &&  (34^9784346 > 34987345)
False &&! (34^9784346 > 34987345)
```

Keduanya akan terevaluasi menjadi `False`, tapi yang kedua akan menghabiskan waktu lebih lama. Bagaimana dengan yang berikut ini?

```
False &&  (head [] == 'x')
False &&! (head [] == 'x')
```

Yang pertama akan bernilai `False` lagi, sedangkan yang kedua akan *crash*. Cobalah!

Semua ini menunjukkan bahwa ada beberapa isu menarik berkaitan dengan *lazyness* yang perlu diperhatikan ketika mendefinisikan sebuah fungsi.

Struktur kontrol buatan pengguna

Dengan mengembangkan ide operator *short-circuiting* lebih jauh, kita bisa membuat *struktur kontrol* kita sendiri di Haskell.

Sebagian besar bahasa pemrograman memiliki konstruk `if` bawaan. Ini sedikit mirip dengan operator *Boolean* yang *short-circuiting*. Berdasarkan nilai yang dites, `if` hanya akan mengeksekusi atau mengevaluasi salah satu dari dua cabang.

Di Haskell kita bisa mendefinisikan `if` sebagai sebuah fungsi pustaka!

```
if' :: Bool -> a -> a -> a
if' True  x _ = x
if' False _ y = y
```

Tentu Haskell memiliki ekspresi `if` bawaan tapi saya tak pernah mengerti mengapa. Mungkin para desainer Haskell beranggapan orang-orang akan mengharapkannya. "Apa? Bahasa ini tidak punya *if*!?" Lagipula, `if` jarang digunakan di Haskell. Kebanyakan kita lebih suka menggunakan pencocokkan pola atau *guards*.

Kita juga bisa mendefinisikan struktur kontrol lain. Kita akan lihat contohnya nanti ketika membahas monad.

Struktur data tak terhingga

Evaluasi *lazy* memungkinkan kita untuk memiliki struktur data tak terhingga. Kita telah melihat beberapa contohnya, seperti `repeat 7` yang merupakan list berisikan `7` sebanyak tak terhingga. Mendefinisikan struktur data tak terhingga sebenarnya hanyalah menciptakan `thunk`, yang bisa kita ibaratkan sebagai "benih" di mana seluruh struktur data tersebut bisa tumbuh sesuai kebutuhan.

Contoh praktis lainnya ialah data yang *terlalu besar*, seperti *tree* yang merepresentasikan *state* dari sebuah *game* (seperti catur atau go). Meski *tree* tersebut terbatas secara teori, tapi bisa menjadi terlalu besar untuk memori. Dengan Haskell, kita bisa mendefinisikan *tree* untuk semua kemungkinan langkah di *game*, lalu membuat algoritma terpisah untuk menjelajahi *tree* tersebut sesuai kemauan. Hanya bagian *tree* yang terjelajah yang akan disediakan.

Pemrograman *pipelining/wholemeal*

Seperti yang telah disebutkan sebelumnya, melakukan transformasi bertahap (*pipelined*) terhadap struktur data yang besar bisa terjadi secara efisien dalam penggunaan memori. Sekarang kita bisa tahu mengapa: karena *lazy*, tiap operasi terjadi secara bertahap. Hasil hanya ada jika dibutuhkan oleh proses selanjutnya dalam *pipeline*.

Pemrograman dinamis

Satu contoh menarik hasil dari evaluasi *lazy* ialah pemrograman dinamis (*dynamic programming*). Biasanya kita harus sangat berhati-hati dalam mengisi tabel pemrograman dinamis sesuai urutan, sehingga tiap kali kita menghitung nilai dari sebuah sel, dependensinya sudah terhitung. Jika urutannya salah, hasil yang didapat juga salah.

Dengan evaluasi *lazy* di Haskell, kita bisa menggunakan Haskell *runtime* untuk mengerjakan pengurutan evaluasi tersebut! Sebagai contoh, berikut kode Haskell untuk memecahkan 0-1 knapsack problem. Perhatikan bagaimana kita mendefinisikan *array* `m` dalam bentuk dirinya sendiri (rekursif), dan membiarkan evaluasi *lazy* mencari urutan yang benar untuk menghitung sel-selnya.

```
import Data.Array

knapsack01 :: [Double]  -- values
           -> [Integer]  -- nonnegative weights
```

```
          -> Integer     -- knapsack size
          -> Double      -- max possible value
knapsack01 vs ws maxW = m!(numItems-1, maxW)
  where numItems = length vs
        m = array ((-1,0), (numItems-1, maxW)) $
              [((-1,w), 0) | w <- [0 .. maxW]] ++
              [((i,0), 0) | i <- [0 .. numItems-1]] ++
              [((i,w), best)
                  | i <- [0 .. numItems-1]
                  , w <- [1 .. maxW]
                  , let best
                          | ws!!i > w  = m!(i-1, w)
                          | otherwise = max (m!(i-1, w))
                                            (m!(i-1, w - ws!!i) + vs!!i)
              ]

example = knapsack01 [3,4,5,8,10] [2,3,4,5,9] 20
```

Folds and monoids

Bacaan tambahan:

- *Learn You a Haskell*, *Only folds and horses*
- *Learn You a Haskell*, *Monoids*
- *Fold* dari wiki Haskell
- *Monoids and Finger Trees*, Heinrich Apfelmus
- *Haskell Monoids and their Uses*, Dan Piponi
- Dokumentasi Data.Monoid
- Dokumentasi Data.Foldable

Fold lagi

Kita telah melihat bagaimana mendefinisikan fungsi *fold* untuk list, tapi kita masi bisa membuatnya lebih umum sehingga bisa digunakan untuk tipe data lainnya.

Perhatikan tipe data *binary tree* berikut, yang menyimpan data di dalam *node* internal:

```
data Tree a = Empty
            | Node (Tree a) a (Tree a)
  deriving (Show, Eq)

leaf :: a -> Tree a
leaf x = Node Empty x Empty
```

Mari kita buat sebuah fungsi untuk menghitung besarnya *tree* tersebut (banyaknya *node*):

```
treeSize :: Tree a -> Integer
treeSize Empty        = 0
treeSize (Node l _ r) = 1 + treeSize l + treeSize r
```

Bagaimana dengan jumlah keseluruhan data dalam *tree* yang berisikan `Integer`?

```
treeSum :: Tree Integer -> Integer
treeSum Empty     = 0
treeSum (Node l x r)  = x + treeSum l + treeSum r
```

Atau kedalaman sebuah *tree*?

```
treeDepth :: Tree a -> Integer
treeDepth Empty        = 0
treeDepth (Node l _ r) = 1 + max (treeDepth l) (treeDepth r)
```

Atau meratakan semua elemen di *tree* menjadi sebuah list?

```
flatten :: Tree a -> [a]
flatten Empty        = []
flatten (Node l x r) = flatten l ++ [x] ++ flatten r
```

Dapatkah kalian melihat polanya? Tiap fungsi di atas:

1. menerima sebuah `Tree` sebagai input
2. cocokkan pola pada `Tree` tersebut
3. jika `Empty`, berikan hasil sederhana
4. jika berisi `Node`:
    1. panggil dirinya sendiri secara rekursif di kedua *subtree*
    2. gabungkan hasil dari rekursif tersebut dengan data `x` untuk mendapatkan hasil akhir

Sebagai programer yang baik, kita harus selalu berjuang mengabstraksikan pola yang berulang. Mari kita generalisasi. Kita perlu menjadikan bagian-bagian dari contoh diatas sebagai parameter, yang berbeda dari contoh satu ke lainnya:

1. Tipe hasil
2. Jawaban untuk kasus `Empty`
3. Cara untuk menggabungkan pemanggilan rekursifnya

Kita sebut tipe data yang terkandung di dalam *tree* sebagai `a`, dan tipe hasilnya sebagai `b`.

```
treeFold :: b -> (b -> a -> b -> b) -> Tree a -> b
treeFold e _ Empty        = e
treeFold e f (Node l x r) = f (treeFold e f l) x (treeFold e f r)
```

Sekarang kita bisa mendefinisikan `treeSize`, `treeSum` dan contoh lainnya dengan lebih sederhana. Mari kita lihat:

```haskell
treeSize' :: Tree a -> Integer
treeSize' = treeFold 0 (\l _ r -> 1 + l + r)

treeSum' :: Tree Integer -> Integer
treeSum' = treeFold 0 (\l x r -> l + x + r)

treeDepth' :: Tree a -> Integer
treeDepth' = treeFold 0 (\l _ r -> 1 + max l r)

flatten' :: Tree a -> [a]
flatten' = treeFold [] (\l x r -> l ++ [x] ++ r)
```

Kita juga bisa membuat fungsi *fold tree* baru dengan mudah:

```haskell
treeMax :: (Ord a, Bounded a) => Tree a -> a
treeMax = treeFold minBound (\l x r -> l `max` x `max` r)
```

Fold pada ekspresi

Di mana lagi kita telah melihat *fold*?

Ingat tipe `ExprT` dan fungsi `eval` dari tugas 5:

```haskell
data ExprT = Lit Integer
           | Add ExprT ExprT
           | Mul ExprT ExprT

eval :: ExprT -> Integer
eval (Lit i)     = i
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2
```

Hmm... terlihat familiar! Bagaimanakah bentuk *fold* untuk `ExprT`?

```haskell
exprTFold :: (Integer -> b) -> (b -> b -> b) -> (b -> b -> b) -> ExprT -> b
exprTFold f _ _ (Lit i)     = f i
exprTFold f g h (Add e1 e2) = g (exprTFold f g h e1) (exprTFold f g h e2)
exprTFold f g h (Mul e1 e2) = h (exprTFold f g h e1) (exprTFold f g h e2)

eval2 :: ExprT -> Integer
eval2 = exprTFold id (+) (*)
```

Sekarang kita bisa melakukan hal lain dengan mudah, seperti menghitung jumlah literal dalam sebuah ekspresi:

```haskell
numLiterals :: ExprT -> Int
numLiterals = exprTFold (const 1) (+) (+)
```

Fold secara umum

Intinya, kita bisa membuat sebuah *fold* untuk banyak tipe data (meski tidak semua). *Fold* dari tipe `T` akan menerima satu (*higher-order*) argumen untuk

tiap konstruktor dari `T`, dan menjabarkan bagaimana mengubah nilai di dalam konstruktor tersebut menjadi nilai yang bertipe sama dengan tipe hasil akhirnya (dengan asumsi tiap rekursif tehadap `T` sudah di-*fold* ke hasil akhir tersebut). Hal ini menyebabkan fungsi-fungsi yang kita akan buat untuk `T` bisa diekspresikan dalam bentuk *fold*.

Monoid

Berikut adalah satu lagi *type class* yang patut kalian ketahui, yang terdapat di modul `Data.Monoid`:

```
class Monoid m where
    mempty  :: m
    mappend :: m -> m -> m

    mconcat :: [m] -> m
    mconcat = foldr mappend mempty

(<>) :: Monoid m => m -> m -> m
(<>) = mappend
```

`(<>)` didefinisikan sebagai sinonim untuk `mappend` (sejak GHC 7.4.1) karena menuliskan `mappend` cukup merepotkan.

Tipe yang merupakan anggota dari `Monoid` memiliki elemen spesialyang disebut `mempty`, dan sebuah operasi biner `mappend` (disingkat menjadi `(<>)`) yang menerima dua nilai dari tipe tersebut dan mengembalikan satu. `mempty` merupakan identitas untuk `<>`, dan `<>` bersifat asosiatif. Dengan kata lain, untuk semua `x`, `y`, and `z`,

1. `mempty <> x == x`
2. `x <> mempty == x`
3. `(x <> y) <> z == x <> (y <> z)`

Asosiatif berarti kita bisa menulis seperti ini tanpa ambigu:

```
a <> b <> c <> d <> e
```

karena kita akan mendapatkan hal yang sama tidak peduli di mana kita meletakkan tanda kurung.

Ada pula `mconcat`, yang digunakan untuk menggabungkan nilai-nilaipada sebuah list.

Secara *default*, implementasi `mconcat` menggunakan `foldr`, tapi dia dimasukkan ke dalam kelas `Monoid` karena beberapa anggota `Monoid` mungkin saja memiliki implementasi yang lebih efisien.

Jika kalian perhatikan, `monoid` ada di mana-mana. Mari kita buat beberapa anggotanya sebagai latihan (semua ini sudah ada di pustaka bawaan).

List, dengan *concatenation*, merupakan sebuah monoid:

```
instance Monoid [a] where
  mempty  = []
  mappend = (++)
```

Bisa dilihat sebelumnya bahwa penjumlahan merupakan monoid pada bilangan bulat (atau bilangan rasional, atau bilangan riil..). Akan tetapi, begitu pula dengan perkalian! Jadi bagaimana? Kita tidak bisa membuat satu tipe menjadi dua anggota yang berbeda dari *type class* yang sama. Kita akan membuat dua buah `newtype`, satu untuk tiap anggota *type class*:

```
newtype Sum a = Sum a
  deriving (Eq, Ord, Num, Show)

getSum :: Sum a -> a
getSum (Sum a) = a

instance Num a => Monoid (Sum a) where
  mempty  = Sum 0
  mappend = (+)

newtype Product a = Product a
  deriving (Eq, Ord, Num, Show)

getProduct :: Product a -> a
getProduct (Product a) = a

instance Num a => Monoid (Product a) where
  mempty  = Product 1
  mappend = (*)
```

Perhatikan bahwa misalnya untuk menemukan *product* dari sebuah list `Integer` dengan menggunakan `mconcat`, kita harus mengubahnya dulu menjadi nilai-nilai bertipe `Product Integer`:

```
lst :: [Integer]
lst = [1,5,8,23,423,99]

prod :: Integer
prod = getProduct . mconcat . map Product $ lst
```

(Tentu contoh ini konyol karena kita bisa langsung menggunakan fungsi bawaan `product`, tapi pola seperti ini bisa berguna nantinya.)

Pair merupakan sebuah monoid selama komponen-komponennya juga monoid:

```
instance (Monoid a, Monoid b) => Monoid (a,b) where
  mempty = (mempty, mempty)
  (a,b) `mappend` (c,d) = (a `mappend` c, b `mappend` d)
```

Tantangan: bisakah kalian membuat anggota `Monoid` untuk `Bool`? Ada berapa anggota berbeda?

Tantangan: bagaimana kalian membuat tipe fungsi sebagai anggota `Monoid`?

IO

Bacaan tambahan:

- *LYAH Chapter 9: Input and Output*
- *RWH Chapter 7: I/O*

Masalah dengan *purity*

Ingat bahwa Haskell bersifat *lazy* dan juga *pure*. Ini mengakibatkan dua hal:

1. Fungsi tidak boleh memiliki efek eksternal. Sebagai contoh, sebuah fungsi tidak bisa mencetak ke layar. Fungsi hanya bisa menghitung hasil.
2. Fungsi tidak boleh bergantung pada hal di luar dirinya. Misalnya, tidak boleh membaca dari *keyboard*, sistem berkas, atau jaringan. Fungsi hanya boleh bergantung pada inputnya. Dengan kata lain, fungsi harus memberikan hasil yang sama untuk input yang sama setiap saat.

Tapi terkadang kita perlu melakukan hal-hal tersebut di atas! Jika di Haskell kita hanya bisa menulis fungsi untuk dievaluasi di ghci, maka tentu tak akan berguna banyak.

Di Haskell kita sebenarnya bisa melakukan semua itu, tapi terlihat sangat berbeda jika dibandingkan dengan bahasa pemrograman lain.

Tipe `IO`

Solusi dari masalah di atas ialah tipe khusus bernama `IO`. Nilai bertipe `IO a` adalah *deskripsi* dari komputasi yang memiliki efek. Dengan kata lain, jika dijalankan (mungkin) akan melakukan operasi I/O dan (pada akhirnya) menghasilkan nilai bertipe `a`. Ada tingkat indireksi di sini yang harus dimengerti. Nilai bertipe `IO a` *dengan sendirinya* adalah sesuatu yang *tidak aktif* tanpa efek samping. Itu hanyalah *deskripsi* dari sebuah komputasi dengan efek samping. Bisa dibayangkan tipe `IO a` merupakan sebuah program imperatif *first-class* di dalam Haskell.

Sebagai ilustrasi, misalkan kalian memiliki

```
c :: Cake
```

Apa yang kalian miliki? Tentu *cake* yang enak. Cukup sederhana.

Sebaliknya, jika kalian memiliki

```
r :: Recipe Cake
```

Apa yang kalian punya? *Cake*? Bukan. Kalian hanya memiliki *instruksi* (resep) untuk membuat *cake*, hanya selembar kertas dengan tulisan di atasnya.

Memiliki resep tak menghasilkan efek apapun. Hanya dengan memegang resep tidak akan menyebabkan *oven* menjadi panas, tepung berserak di lantai, dan lain sebagainya. Untuk menghasilkan *cake*, resep tersebut harus diikuti (yang akan menyebabkan tepung berserakan, bahan-bahan tercampur, oven menjadi panas, dsb).

Sama seperti di atas, nilai bertipe `IO a` hanyalah sebuah "resep" untuk mendapatkan nilai bertipe `a` (dan memiliki efek samping). Seperti nilai lainnya, dia bisa diberikan sebagai argumen, dikembalikan sebagai hasil dari fungsi, disimpan di struktur data, atau (yang akan segera kita lihat) digabungkan dengan nilai `IO` lain menjadi resep yang lebih kompleks.

Jadi bagaimana nilai bertipe `IO a` bisa dijalankan? Hanya satu cara: *compiler* Haskell mencari nilai spesial

```
main :: IO ()
```

yang akan diberikan ke sistem *runtime* dan dijalankan. Bayangkan sistem *runtime* di Haskell sebagai *master chef*, satu-satunya orang yang diizinkan untuk memasak.

Jika kalian mau resep kalian juga disertakan maka kalian harus membuatnya menjadi bagian dari resep besar (`main`) yang diberikan ke *master chef*. Tentunya `main` bisa menjadi kompleks, dan biasanya terdiri dari beberapa komputasi `IO` yang lebih kecil.

Untuk yang pertama kalinya, mari kita buat program Haskell yang *executable*! Kita bisa menggunakan fungsi

```
putStrLn :: String -> IO ()
```

yang jika diberikan sebuah `String`, akan mengembalikan sebuah komputasi `IO` yang akan (ketika dijalankan) mencetak `String` tersebut di layar. Kita cukup menulis ini ke sebuah file bernama `Hello.hs`:

```
main = putStrLn "Hello, Haskell!"
```

Mengetikkan `runhaskell Hello.hs` di *command-line prompt* menghasilkan pesan kita tercetak di layar! Kita juga bisa menggunakan `ghc --make Hello.hs` untuk menghasilkan berkas *executable* bernama `Hello` (atau `Hello.exe` di Windows).

Tidak ada `String` "di dalam" `IO String`

Banyak pemula di Haskell bertanya-tanya "Saya punya `IO String`, bagaimana cara mengubahnya menjadi `String`?", atau "Bagaimana cara mengeluarkan `String` dari `IO String`?". Dari penjelasan sebelumnya, jelas bahwa pertanyaan-pertanyaan tersebut tidak masuk akal. Tipe `IO String` merupakan deskripsi komputasi, sebuah resep, untuk menghasilkan `String`. Tidak ada `String` di dalam `IO String`, seperti halnya tidak ada *cake* di dalam resep *cake*. Untuk menghasilkan `String` (atau *cake* yang lezat), kita perlu menjalankan komputasinya (atau resep). Dan satu-satunya cara untuk melakukannya ialah dengan memberikannya (mungkin sebagai bagian dari `IO` yang lebih besar) ke sistem *runtime* Haskell melalui `main`.

Menggabungkan `IO`

Sudah jelas kalau kita perlu sebuah cara untuk menggabungkan komputasi-komputasi `IO` menjadi satu komputasi yang lebih besar.

Cara paling sederhana untuk menggabungkan dua buah komputasi `IO` ialah dengan menggunakan operator `(>>)` (dilafalkan "*and then*", terjemahan: "lalu" atau "kemudian") yang bertipe

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

`(>>)` menciptakan sebuah komputasi `IO` yang terdiri dari menjalankan dua komputasi input secara berurutan. Perhatikan bahwa hasil dari komputasi pertama diabaikan. Kita hanya peduli terhadap efeknya. Sebagai contoh:

```
main = putStrLn "Hello" >> putStrLn "world!"
```

Ini tidak masalah untuk kode berbentuk "do this; do this; do this" di mana hasilnya diabaikan. Akan tetapi ini saja belum cukup. Bagaimana kalau kita tidak ingin mengabaikan hasil dari komputasi pertama?

Yang pertama kali terpikirkan mungkin dengan memiliki tipe seperti `IO a -> IO b -> IO (a,b)` akan memecahkan masalah. Ini pun belum cukup. Alasannya, kita ingin komputasi kedua bergantung terhadap hasil komputasi yang pertama. Misalnya kita ingin membaca sebuah integer dari pengguna, lalu mencetak bilangan tersebut ditambah satu. Dalam kasus ini, komputasi kedua (mencetak ke layar) akan berbeda dan bergantung pada hasil dari komputasi pertama.

Solusinya, dengan operator `(>>=)` (dilafalkan "*bind*") yang bertipe

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

Ini mungkin akan sulit dimengerti pada awalnya. `(>>=)` menerima sebuah komputasi yang akan menghasilkan nilai bertipe `a`, dan sebuah *fungsi* yang akan melakukan komputasi kedua berdasarkan nilai bertipe `a` yang tadi dihasilkan.

Hasil dari `(>>=)` adalah (deskripsi dari) sebuah komputasi yang menjalankan komputasi pertama, gunakan hasilnya untuk menentukan komputasi selanjutnya, lalu jalankan komputasi tersebut.

Sebagai contoh, kita bisa menulis program yang menerima bilangan dari pengguna dan mencetak suksesornya (ditambah 1). Perhatikan penggunaan `readLn :: Read a => IO a` yang merupakan komputasi yang membaca input dari pengguna, dan mengubahnya jadi tipe apapun selama merupakan anggota dari `Read`.

```
main :: IO ()
main = putStrLn "Please enter a number: " >> (readLn >>= (\n -> putStrLn (show (n+1))))
```

Tentu ini terlihat jelek. Nantinya kita akan mempelajari cara menulisnya dengan lebih baik.

Sintaks *record*

Materi ini tidak dibahas di kuliah tapi disediakan ekstra untuk mengerjakan tugas 8.

Misalkan kita memiliki tipe data seperti

```
data D = C T1 T2 T3
```

Kita juga bisa mendeklarasi tipe data tersebut dengan sintaks *record* sebagai berikut:

```
data D = C { field1 :: T1, field2 :: T2, field3 :: T3 }
```

di mana kita tidak hanya mendeskripsikan tipe tapi juga *nama* untuk tiap *field* yang terdapat di konstruktor `C`. `D` versi baru ini bisa digunakan sama seperti versi lamanya. Kita bisa membuat konstruksi dan mencocokkan pola terhadap nilai bertipe `D` seperti `C v1 v2 v3`. Selain itu, kita juga mendapatkan keuntungan tambahan.

1. Tiap nama *field* secara otomatis merupakan fungsi proyeksi (*projection function*) yang mendapatkan nilai dari *field* tersebut di nilai bertipe `D`. Sebagai contoh, `field2` ialah sebuah fungsi bertipe

   ```
   field2 :: D -> T2
   ```

   Sebelumnya kita harus membuat implementasi `field2` sendiri dengan menuliskan

   ```
   field2 (C _ f _) = f
   ```

   Ini menghilangkan banyak kode tambahan (*boilerplate*) seandainya kita memiliki tipe data dengan banyak *field*!

2. Terdapat sintaks khusus untuk *membuat*, *mengubah*, and *mencocokkan pola* untuk nilai bertipe `D` (selain sintaks biasa).

Kita bisa membuat *membuat* sebuah nilai bertipe `D` menggunakan sintaks seperti

```
C { field3 = ..., field1 = ..., field2 = ... }
```

dengan `...` diisi dengan ekspresi bertipe tepat. Perhatikan bahwa kita bisa deskripsikan *field*-nya dengan urutan bebas.

Jika kita memiliki sebuah nilai `d :: D`. Kita bisa *mengubah* `d` menggunakan sintaks seperti

```
d { field3 = ... }
```

Tentu yang dimaksud "mengubah" di sini bukan *mutating* `d`, tapi membuat nilai baru bertipe `D` yang sama persis seperti `d` kecuali `field3` yang nilainya tergantikan dengan nilai baru.

Akhirnya, kita bisa *mencocokkan pola* pada nilai bertipe `D` seperti berikut:

```
foo (C { field1 = x }) = ... x ...
```

Ini hanya mencocokkan *field* `field1` dari nilai bertipe `D`, dengan menyebutnya sebagai `x`, mengabaikan *field* lainnya. (Tentu `x` di sini hanya contoh, kita bisa mencocokkan dengan pola apapun).

Functor

Bacaan tambahan:

- *Learn You a Haskell*, *The Functor typeclass*
- *Typeclassopedia*

Motivasi

Dalam beberapa minggu terakhir kita telah melihat beberapa fungsi yang dirancang untuk "memetakan" (*map*) sebuah fungsi ke tiap elemen di dalam sesuatu yang menyerupai *container*. Seperti:

- `map :: (a -> b) -> [a] -> [b]`
- `treeMap :: (a -> b) -> Tree a -> Tree b`
- Di tugas 5 banyak yang melakukan hal yang serupa ketika harus menerapkan `eval :: ExprT -> Int` ke sebuah `Maybe ExprT` untuk mendapatkan sebuah `Maybe Int`.

  ```
  maybeEval :: (ExprT -> Int) -> Maybe ExprT -> Maybe Int
  ```

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

Terdapat pengulangan pola di sini, dan sebagai programer Haskell yang baik kita harus tahu bagaimana cara menggeneralisirnya! Jadi yang mana yang serupa dari tiap contoh, dan mana yang berbeda?

Yang berbeda tentu *container* yang dipetakan terhadap fungsi:

```
thingMap :: (a -> b) -> f a -> f b
```

Akan tetapi apa sebenarnya *container* ini? Bisakah kita *assign* variabel tipe seperti `f` ke *container* tersebut?

Bahasan singkat tentang *kind*

Seperti halnya ekspresi yang memiliki tipe, tipe juga memiliki "tipe", yang disebut *kind* (translasi: jenis). Sebelum kalian bertanya: tidak, tak ada lagi tingkat di atas *kind* – setidaknya di Haskell. Di `ghci` kita bisa menanyakan *kind* dari tipe dengan menggunakan `:kind`. Sebagai contoh, mari kita cari tahu apa *kind* dari `Int`:

```
Prelude> :k Int
Int :: *
```

Kita lihat bahwa `Int` memiliki *kind* `*`. Sebenarnya tiap tipe dari sebuah nilai memiliki *kind* `*`.

```
Prelude> :k Bool
Bool :: *
Prelude> :k Char
Char :: *
Prelude> :k Maybe Int
Maybe Int :: *
```

Jika `Maybe Int` memiliki *kind* `*`, bagaimana dengan `Maybe`? Perhatikan bahwa tidak ada nilai untuk tipe `Maybe`. Terdapat nilai untuk tipe `Maybe Int`, dan tipe `Maybe Bool`, tapi tidak untuk tipe `Maybe`. Tapi `Maybe` merupakan sesuatu yang menyerupai tipe yang valid. Jadi apa dong? Apa *kind* yang dimilikinya? Mari tanyakan `ghci`:

```
Prelude> :k Maybe
Maybe :: * -> *
```

`ghci` barkata bahwa `Maybe` memiliki *kind* `* -> *`. `Maybe` bisa dikatakan sebagai sebuah fungsi terhadap tipe. Kita biasa menyebutnya *type constructor* (konstruktor tipe). `Maybe` menerima input tipe dengan *kind* `*`, dan menghasilkan sebuah tipe lain dengan *kind* `*`. Sebagai contoh, dia bisa menerima input `Int :: *` dan menghasilkan tipe baru `Maybe Int :: *`.

Adakah konstruktor tipe lain dengan *kind* `* -> *`? Tentunya. Contohnya `Tree`, atau konstruktor tipe list yang ditulis sebagai `[]`.

```
Prelude> :k []
[] :: * -> *
Prelude :k [] Int
[] Int :: *
Prelude> :k [Int]  -- special syntax for [] Int
[Int] :: *
Prelude> :k Tree
Tree :: * -> *
```

Bagaimana dengan konstruktor tipe dengan *kind* lainnya? Bagaimana dengan `JoinList` dari tugas 7?

```
data JoinList m a = Empty
                  | Single m a
                  | Append m (JoinList m a) (JoinList m a)
```

```
Prelude> :k JoinList
JoinList :: * -> * -> *
```

Masuk akal. `JoinList` menerima *dua* tipe sebagai parameter dan memberikan sebuah tipe baru. Tentunya, `JoinList` *curried*, jadi kita juga bisa menganggapnya menerima sebuah tipe dan menghasilkan sesuatu dengan *kind* `* -> *`. Berikut satu contoh lagi:

```
Prelude> :k (->)
(->) :: * -> * -> *
```

Konstruktor tipe fungsi menerima dua tipe sebagai argumen. Seperti operator kita bisa menggunakannya *infix*:

```
Prelude> :k Int -> Char
Int -> Char :: *
```

Bisa juga tidak:

```
Prelude> :k (->) Int Char
(->) Int Char :: *
```

OK, bagaimana dengan ini?

```
data Funny f a = Funny a (f a)
```

```
Prelude> :k Funny
Funny :: (* -> *) -> * -> *
```

`Funny` menerima dua argumen, yang pertama tipe dengan *kind* `* -> *`, dan yang kedua tipe dengan *kind* `*`, dan menghasilkan sebuah tipe. Bagaimana GHCi tahu apa *kind* dari `Funny`? Dia bisa melakukan *kind inference*, serupa dengan *type inference*. `Funny` adalah sebuah konstruktor tipe *higher-order*, serupa dengan `map` yang merupakan sebuah fungsi *higher-order*. Perhatikan bahwa tipe juga bisa diterapkan sebagian (*partially applied*) seperti fungsi:

```
Prelude> :k Funny Maybe
Funny Maybe :: * -> *
Prelude> :k Funny Maybe Int
Funny Maybe Int :: *
```

Functor

Inti dari pola pemetaan yang kita lihat adalah fungsi *higher-order* yang bertipe seperti

```
thingMap :: (a -> b) -> f a -> f b
```

di mana `f` ialah variabel tipe yang mewakili tipe dengan *kind* `* -> *`. Jadi, bisakah kita menulis fungsi bertipe demikian untuk semuanya?

```
thingMap :: (a -> b) -> f a -> f b
thingMap h fa = ???
```

Tidak juga. Tidak banyak yang bisa kita lakukan jika kita tidak tahu apa `f` tersebut. `thingMap` harus bekerja secara berbeda untuk tiap `f` apapun. Solusinya adalah dengan membuat *type class*, yang secara tradisi disebut `Functor`:

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

Catatan: `Functor` didefinisikan di *Prelude*. Perhatikan bahwa nama "*functor*" berasal dari *category theory*, dan *tidak sama* dengan functors di C++ (yang merupakan fungsi *first-class*).

Sekarang kita bisa implemen *type class* tersebut secara spesifik untuk tiap `f`. Perhatikan bahwa `Functor` melakukan abstraksi terhadap tipe dengan *kind* `* -> *`. Jadi mustahil untuk menulis

```
instance Functor Int where
  fmap = ...
```

Jika kita coba, kita akan mendapatkan *kind mismatch error*:

```
[1 of 1] Compiling Main             ( 09-functors.lhs, interpreted )

09-functors.lhs:145:19:
    Kind mis-match
    The first argument of `Functor' should have kind `* -> *',
    but `Int' has kind `*'
    In the instance declaration for `Functor Int'
```

Jika kita sudah memahami *kind*, pesan di atas cukup jelas.

Cukup beralasan untuk menjadikan `Maybe` sebagai anggota dari `Functor`. Mari kita lakukan. Dengan mengikuti tipe, ini menjadi sangat mudah:

```
instance Functor Maybe where
  fmap _ Nothing  = Nothing
  fmap h (Just a) = Just (h a)
```

Bagaimana dengan list?

```
instance Functor [] where
  fmap _ []     = []
  fmap f (x:xs) = f x : fmap f xs
  -- or just
  -- fmap = map
```

Gampang! Bagaimana dengan `IO`? Bisakah menjadikan `IO` sebagai anggota `Functor`?

Tentu. `fmap :: (a -> b) -> IO a -> IO b` menghasilkan sebuah *IO* yang akan menjalankan aksi `IO a`, lalu menerapkan fungsi tersebut untuk mengubah hasilnya sebelum dikembalikan. Kita bisa mengimplementasikannya tanpa masalah:

```
instance Functor IO where
  fmap f ioa = ioa >>= (\a -> return (f a))
```

atau

```
instance Functor IO where
  fmap f ioa = ioa >>= (return . f)
```

Sekarang mari kita coba sesuatu yang lebih memutar otak:

```
instance Functor ((->) e) where
```

Apa!? Mari kita ikuti tipenya: jika `f = (->) e` maka kita ingin agar

```
fmap :: (a -> b) -> (->) e a -> (->) e b
```

atau, jika `(->)` ditulis dalam bentuk infix:

```
fmap :: (a -> b) -> (e -> a) -> (e -> b)
```

Hmm, tipe *signature* tersebut terlihat familiar…

```
instance Functor ((->) e) where
  fmap = (.)
```

Gila! Apa artinya? Kita bisa mengibaratkan nilai bertipe `(e -> a)` adalah *container* berindeks `e` dengan nilai `a` untuk tiap nilai `e`. Pemetaan sebuah fungsi ke tiap nilai di dalam *container* tersebut sangatlah persis dengan komposisi fungsi. Untuk mengambil elemen dari *container* hasilnya, kita pertama menerapkan fungsi `(e -> a)` untuk mendapatkan `a` dari *container* asal, lalu menerapkan fungsi `(a -> b)` untuk mengubah elemen yang didapat.

Applicative Functor, Bagian I

Bacaan tambahan:

- *Applicative Functors* dari *Learn You a Haskell*
- *Typeclassopedia*

Motivasi

Perhatikan tipe `Employee` berikut:

```
type Name = String

data Employee = Employee { name    :: Name
                         , phone   :: String }
                deriving Show
```

Tentunya konstruktor `Employee` bertipe

```
Employee :: Name -> String -> Employee
```

Jika kita memiliki sebuah `Name` dan sebuah `String`, kita bisa terapkan (*apply*) konstruktor `Employee` untuk mendapatkan sebuah `Employee`.

Misalkan kita tidak memiliki sebuah `Name` dan `String`, melainkan `Maybe Name` dan `Maybe String`. Mungkin karena kita mendapatkannya dengan melakukan *parsing* berkas yang penuh *error*, atau dari *form* yang tidak sepenuhnya diisi, atau kasus-kasus lainnya. Mungkin kita tidak bisa membuat `Employee`, tapi paling tidak kita bisa membuat `Maybe Employee`.

Kita akan mengubah fungsi `(Name -> String -> Employee)` menjadi fungsi `(Maybe Name -> Maybe String -> Maybe Employee)`. Bisakah kita membuat sesuatu bertipe seperti ini?

```
(Name -> String -> Employee) ->
(Maybe Name -> Maybe String -> Maybe Employee)
```

Tentu saja bisa. Saya pun yakin kalian sudah bisa membuatnya sambil tidur sekarang. Kita bisa membayangkan bagaiman fungsi tersebut bekerja. Jika salah satu dari *name* atau string berupa `Nothing`, kita mendapatkan `Nothing`. Jika keduanya berupa `Just`, kita mendapatkan `Employee` yang dibuat dengan konstruktor `Employee` (terbungkus dengan `Just`). Mari kita lanjutkan…

Sekarang begini: bukannya kita memiliki sebuah `Name` dan `String`, namun kita punya `[Name]` dan `[String]`. Mungkin kita bisa mendapatkan `[Employee]` di sini? Sekarang kita mau

```
(Name -> String -> Employee) ->
([Name] -> [String] -> [Employee])
```

Kita bisa bayangkan dua cara untuk ini:

- Kita bisa memasangkan satu `Name` untuk satu `String` untuk membuat `Employee`
- Kita bisa memasangkan `Name` dan `String` dengan segala kombinasi kemungkinannya.

Atau bagaimana kalau begini: kita punya `(e -> Name)` dan `(e -> String)` untuk `e` apapun. Sebagai contoh, `e` mungkin sebuah struktur data yang besar dan kita punya fungsi untuk mengekstrak `Name` dan `String` darinya. Bisakah kita membuatnya menjadi `(e -> Employee)`, yang merupakan resep untuk mengekstrak `Employee` dari struktur tersebut?

```
(Name -> String -> Employee) ->
((e -> Name) -> (e -> String) -> (e -> Employee))
```

Tidak masalah, dan kali ini hanya ada satu cara untuk menulis fungsi tersebut.

Generalisir

Setelah melihat kegunaan pola seperti di atas, mari kita sedikit menggeneralisir. Tipe fungsi yang kita inginkan adalah seperti berikut:

```
(a -> b -> c) -> (f a -> f b -> f c)
```

Hmm, terlihat familiar… serupa dengan tipe dari `fmap`!

```
fmap :: (a -> b) -> (f a -> f b)
```

Satu-satunya perbedaan adalah sebuah argumen tambahan. Kita bisa menyebut fungsi baru ini sebagai `fmap2`, karena menerima sebuah fungsi dengan dua argumen. Mungkin kita bisa menuliskannya dalam bentuk `fmap`, sehingga kita hanya memerlukan *constraint* `Functor` pada `f`:

```
fmap2 :: Functor f => (a -> b -> c) -> (f a -> f b -> f c)
fmap2 h fa fb = undefined
```

Setelah mencoba, `Functor` tidak cukup membantu kita untuk membuat `fmap2`. Apa yang salah? Kita memiliki

```
h  :: a -> b -> c
fa :: f a
fb :: f b
```

Perhatikan bahwa kita bisa menuliskan tipe `h` sebagai `(a -> (b -> c))`. Jadi kita memiliki sebuah fungsi yang menerima `a`, dan sebuah nilai bertipe `f a`. Kita tinggal "mengangkat" fungsi tersebut melewati `f` dengan `fmap` yang akan menghasilkan:

```
h          :: a -> (b -> c)
fmap h     :: f a -> f (b -> c)
fmap h fa  :: f (b -> c)
```

Oke, sekarang kita memiliki sesuatu bertipe `f (b -> c)` dan `f b`... dan di sinilah kita *stuck*! `fmap` tidak bisa membantu lebih jauh. `fmap` memberikan cara untuk menerapkan fungsi ke nilai-nilai yang berada di dalam konteks `Functor`, tapi yang kita butuhkan sekarang adalah penerapan fungsi yang juga berada di dalam konteks `Functor` ke nilai-nilai yang berada di konteks `Functor`.

Applicative

Functor yang memiliki karakter seperti di atas (penerapan fungsi berdasarkan konteks, *contextual application*) disebut *applicative*. Kelas `Applicative` (didefinisikan di [`Control.Applicative`] (http://haskell.org/ghc/docs/latest/html/libraries/base/Control-Applicative.html)) berpola seperti berikut ini.

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

Operator `(<*>)` (biasa disebut "ap", versi singkat dari *apply*, terjemahan: terap) mewakili prinsip penerapan kontekstual (*contextual application*). Perhatikan bahwa kelas `Applicative` mewajibkan anggotanya untuk juga menjadi anggota `Functor`, sehingga kita selalu bisa menggunakan `fmap` terhadap anggota `Applicative`. `Applicative` juga memiliki *method* lain bernama `pure` yang memungkinkan kita untuk memasukkan nilai `a` ke sebuah *container*. Untuk saat ini, kita bisa bisa menyebut `pure` sebagai `fmap0`:

```
pure  :: a             -> f a
fmap  :: (a -> b)      -> f a -> f b
fmap2 :: (a -> b -> c) -> f a -> f b -> f c
```

Setelah kita memiliki `(<*>)`, kita bisa mengimplemen `fmap2`, yang disebut `liftA2` di pustaka standar:

```
liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c
liftA2 h fa fb = (h `fmap` fa) <*> fb
```

Bahkan, pola ini cukup umum sehingga `Control.Applicative` mendefinisikan `(<$>)` sebagai sinonim untuk `fmap`,

```
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<$>) = fmap
```

sehingga kita bisa menulis

```
liftA2 h fa fb = h <$> fa <*> fb
```

Bagaimana dengan `liftA3`?

```
liftA3 :: Applicative f => (a -> b -> c -> d) -> f a -> f b -> f c -> f d
liftA3 h fa fb fc = ((h <$> fa) <*> fb) <*> fc
```

(Perhatikan bahwa prioritas dan sifat asosiatif dari `(<$>)` dan `(<*>)` didefinisikan sedemikian rupa sehingga semua tanda kurung di atas menjadi tidak diperlukan.)

Ringkas! Tidak seperti perpindahan dari `fmap` ke `liftA2` (yang membutuhkan generalisasi dari `Functor` ke `Applicative`), dari `liftA2` ke `liftA3` (dan dari situ ke `liftA4`, … dan seterusnya) tidak memerlukan usaha tambahan. `Applicative` sudah cukup.

Sebenarnya, ketika kita memiliki semua argumen, kita tidak perlu untuk menyebutnya `liftA2`, `liftA3`, dan seterusnya. Cukup gunakan pola `f <$> x <*> y <*> z <*> ...` langsung. (`liftA2` dan lainnya berguna ketika saat aplikasi parsial.)

Bagaimana dengan `pure`? `pure` digunakan ketika kita ingin menerapkan sebuah fungsi ke beberapa argumen yang berada di dalam konteks *functor* `f`, tetapi salah satu argumennya tidak berada di dalam `f`. Argumen tersebut bisa disebut "*pure*" (murni). Kita bisa menggunakan `pure` untuk mengangkat mereka ke `f` sebelum melakukan penerapan. Sebagai contoh:

```
liftX :: Applicative f => (a -> b -> c -> d) -> f a -> b -> f c -> f d
liftX h fa b fc = h <$> fa <*> pure b <*> fc
```

Hukum-hukum *applicative*

Hanya ada satu hukum yang benar-benar menarik untuk `Applicative`:

```
f `fmap` x === pure f <*> x
```

Memetakan sebuah fungsi `f` pada *container* `x` harus memberikan hasil yang sama dengan memasukkan fungsi tersebut ke *container* lalu menerapkannya ke `x` dengan `(<*>)`.

Ada hukum-hukum lainnya, tetapi mereka tidak begitu instruktif. Kalian bisa membacanya sendiri jika mau.

Contoh *applicative*

Maybe

Mari tulis beberapa anggota dari `Applicative`, dimulai dengan `Maybe`. `pure` bekerja dengan memasukkan sebuah nilai ke bungkus `Just`. `(<*>)` adalah aplikasi/ penerapan fungsi dengan kemungkinan gagal, yang akan menghasilkan `Nothing` jika salah satu dari fungsi atau argumennya berupa `Nothing`.

```haskell
instance Applicative Maybe where
  pure              = Just
  Nothing <*> _     = Nothing
  _ <*> Nothing     = Nothing
  Just f <*> Just x = Just (f x)
```

Mari lihat contohnya:

```haskell
m_name1, m_name2 :: Maybe Name
m_name1 = Nothing
m_name2 = Just "Brent"

m_phone1, m_phone2 :: Maybe String
m_phone1 = Nothing
m_phone2 = Just "555-1234"

exA = Employee <$> m_name1 <*> m_phone1
exB = Employee <$> m_name1 <*> m_phone2
exC = Employee <$> m_name2 <*> m_phone1
exD = Employee <$> m_name2 <*> m_phone2
```

Applicative functor, Bagian II

Bacaan tambahan:

- *Applicative Functor* di *Learn You a Haskell*
- *The Typeclassopedia*

Mari kita perhatikan kembali *type class* `Functor` dan `Applicative`:

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

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

Tiap `Applicative` juga merupakan `Functor`, bisakah kita implemen `fmap` dalam bentuk `pure` dan `(<*>)`? Mari kita coba!

```haskell
fmap g x = pure g <*> x
```

Setidaknya, tipenya cocok! Akan tetapi, bukan mustahil untuk membuat anggota `Functor` and `Applicative` yang tidak cocok satu sama lain. Karena hal ini menimbulkan keraguan, kita membuat perjanjian bahwa anggota `Functor` dan `Applicative` untuk tipe apapun harus cocok satu sama lain.

Sekarang, mari kita lihat beberapa contoh lain dari anggota `Applicative`.

Contoh-contoh *Applicative* Lain

Lists

Bagaimana list menjadi anggota `Applicative`? Ada dua kemungkinan: satu yang memasangkan elemen dari list fungsi ke list argumen secara berurutan ("zip"), dan satu lagi yang menggabungkan fungsi dan argumennya dengan seluruh kemungkinan kombinasi.

Mari kita buat anggota yang melakukan seluruh kombinasi dahulu. (Dengan alasan yang akan diketahui minggu depan, ini merupakan anggota *default*). Dari sisi ini, list bisa dilihat sebagai hal yang "non-deterministik": nilai bertipe `[a]` bisa diibaratkan dengan sebuah nilai dengan beberapa kemungkinan. Jika demikian, maka `(<*>)` bisa disebut sebagai aplikasi fungsi non-deterministik, yaitu aplikasi sebuah fungi yang non-deterministik ke argumen yang non-deterministik.

```
instance Applicative [] where
  pure a        = [a]          -- a "deterministic" value
  [] <*> _      = []
  (f:fs) <*> as = (map f as) ++ (fs <*> as)
```

Ini contohnya:

```
names  = ["Joe", "Sara", "Mae"]
phones = ["555-5555", "123-456-7890", "555-4321"]

employees1 = Employee <$> names <*> phones
```

Mungkin contoh ini tidak masuk akal, tapi tak sulit untuk membayangkan situasi di mana kalian butuh menggabungkan hal dengan semua kemungkinan. Misalnya, aritmatika non-deterministik sebagai berikut:

```
(.+) = liftA2 (+)    -- penjumlahan diangkat ke konteks Applicative
(.*) = liftA2 (*)    -- juga perkalian

-- nondeterministic arithmetic
n = ([4,5] .* pure 2) .+ [6,1] -- (4 atau 5) dikali 2, plus (6 atau 1)

-- aritmatika yang mungkin gagal
m1 = (Just 3 .+ Just 5) .* Just 8
m2 = (Just 3 .+ Nothing) .* Just 8
```

Selanjutnya, kita akan menulis anggota yang melakukan pemasangan sesuai urutan. Pertama, kita harus menjawab pertanyaan penting: bagaimana mengatasi list dengan panjang yang berbeda? Hal yang paling masuk akal ialah memotong list yang lebih panjang sehingga sama panjangnya dengan yang lebih pendek, dan membuang elemen yang tidak diperlukan. Tentu saja ada cara lainnya. Mungkin kita bisa memanjangkan list yang lebih pendek dengan menyalin el-

emen terakhirnya (tapi bagaimana jika salah satu list kosong?). Atau bisa juga memanjangkan list dengan elemen "netral" (tapi tentu kita memerlukan anggota `Monoid`, atau sebuah argumen "*default*" tambahan).

Keputusan ini mendikte bagaimana kita mengimplemen `pure`, karena kita harus mematuhi hukum berikut

```
pure f <*> xs === f <$> xs
```

Perhatikan bahwa list di sisi kanan sama panjangnya dengan yang di sisi kiri. Satu-satunya cara kita bisa membuat sisi kiri sepanjang itu ialah dengan `pure` yang menciptakan list `f` tak berhingga, karena kita tidak tahu `xs` akan seberapa panjang.

Kita buat anggota dengan menggunakan pembungkus `newtype` untuk membedakannya dengan anggota yang lain. Fungsi `zipWith` yang terdapat di *Prelude* juga membantu kita.

```
newtype ZipList a = ZipList { getZipList :: [a] }
  deriving (Eq, Show, Functor)

instance Applicative ZipList where
  pure = ZipList . repeat
  ZipList fs <*> ZipList xs = ZipList (zipWith ($) fs xs)
```

Sebagai contoh:

```
employees2 = getZipList $ Employee <$> ZipList names <*> ZipList phones
```

Reader/environment

Contoh terakhir untuk `(->) e`. Ini disebut *applicative reader* (pembaca) atau *environment* (lingkungan), karena membuat kita bisa "membaca" dari "lingkungan" `e`. Implementasi anggota ini tidak begitu sulit kalau kita mengikuti tipe-tipenya:

```
instance Functor ((->) e) where
  fmap = (.)

instance Applicative ((->) e) where
  pure = const
  f <*> x = \e -> (f e) (x e)
```

Contoh `Employee` (pegawai):

```
data BigRecord = BR { getName         :: Name
                    , getSSN          :: String
                    , getSalary       :: Integer
                    , getPhone        :: String
                    , getLicensePlate :: String
                    , getNumSickDays  :: Int
                    }
```

```haskell
r = BR "Brent" "XXX-XX-XXX4" 600000000 "555-1234" "JGX-55T3" 2

getEmp :: BigRecord -> Employee
getEmp = Employee <$> getName <*> getPhone

exQ = getEmp r
```

Tingkat Abstraksi

`Functor` cukup berguna dan jelas. Pada awalnya, sepertinya `Applicative` tidak begitu membawa perubahan berarti dari apa yang telah dimiliki `Functor`. Akan tetapi, perubahan kecil ini memiliki efek yang besar. `Applicative` (dan yang akan kita lihat minggu depan, `Monad`) bisa disebut sebagai "model komputasi", sementara `Functor` tidak demikian.

Ketika bekerja dengan `Applicative` dan `Monad`, perlu selalu diingat bahwa ada beberapa tingkat abstraksi. Kasarnya, abstraksi ialah sesuatu yang menyembunyikan detil tingkat di bawahnya dan menyediakan antarmuka yang bisa digunakan (idealnya) tanpa memikirkan tingkat di bawahnya tersebut, meski terkadang ada detil dari tingkat bawah yang "bocor" di beberapa kasus. Ide tentang beberapa tingkat abstraksi ini banyak digunakan. Contohnya, program—OS—kernel—sirkuit terpadu (IC)—gerbang logika—*silicon*, atau HTTP—TCP—IP—Ethernet, atau bahasa pemrograman—*bytecode*—*assembly*—kode mesin.

Haskell memberi kita alat untuk membuat beberapa tingkat abstraksi *di dalam program Haskell sendiri*. Dengan kata lain, kita bisa meng-*extend* tingkat abstraksi dari bahasa pemrograman ke atas. Hal ini sangat kuat dan berguna tapi juga bisa membingungkan. Kita harus belajar berpikir di beberapa tingkat secara eksplisit, dan berpindah-pindah di antara tingkat-tingkat tersebut.

Dalam `Applicative` dan `Monad`, terdapat dua tingkatan untuk dipahami. Pertama ialah tingkat di mana implementasi anggota `Applicative` dan `Monad`. Kalian mendapatkan pengalaman di tingkat ini di tugas sebelumnya, ketika membuat `Parser` menjadi anggota `Applicative`.

Ketika `Parser` sudah menjadi angota `Applicative`, berikutnya kita "naik satu tingkat" dan membuat program dengan `Parser` melalui antarmuka `Applicative`, tanpa memikirkan bagaimana `Parser` dan implementasi anggota `Applicative`nya dibuat. Kalian mendapatkan pengalaman ini di tugas minggu lalu, dan akan mendapatkannya lagi minggu ini. Pemrograman di tingkat ini memiliki "rasa" yang berbeda dengan pengerjaan detil anggota. Mari kita lihat beberapa contoh.

Applicative API

Salah satu keuntungan memiliki antarmuka seragam seperti `Applicative` ialah kita bisa menulis perkakas umum dan struktur kontrol yang bisa bekerja dengan *semua* anggota `Applicative`. Sebagai contoh, mari kita tulis

```
pair :: Applicative f => f a -> f b -> f (a,b)
```

`pair` menerima dua nilai dan memasangkannya. Semua hal tersebut dilakukan dalam konteks `Applicative f`. Percobaan pertama, ambil fungsi untuk memasangkan, dan "angkat" melewati argumennya dengan menggunakan `(<$>)` dan `(<*>)`:

```
pair fa fb = (\x y -> (x,y)) <$> fa <*> fb
```

Ini sudah berfungsi, tapi masih bisa disederhanakan. Ingat di Haskell kita bisa menggunakan sintaks untuk melambangkan konstruktor *pair*, jadi kita bisa menuliskan

```
pair fa fb = (,) <$> fa <*> fb
```

Sebenarnya kita sudah melihat pola ini sebelumnya. Ini adalah pola `liftA2` yang membuat kita mulai belajar `Applicative`. Kita bisa menyederhanakannya lebih jauh dengan

```
pair fa fb = liftA2 (,) fa fb
```

Sekarang kita tidak perlu menuliskan argumen fungsi secara eksplisit, sehingga kita bisa memperoleh versi final dari fungsi ini:

```
pair = liftA2 (,)
```

Jadi, apa yang fungsi ini lakukan? Tergantung dari `f` yang diberikan. Mari kita lihat beberapa contoh kasus:

- `f = Maybe`: menghasilkan `Nothing` jika salah satu argumen berupa `Nothing`. Jika keduanya `Just`, hasilnya berupa `Just` dari pasangan tersebut.
- `f = []`: menghasilkan produk Cartesian dari dua buah list.
- `f = ZipList`: sama dengan fungsi `zip` standar.
- `f = IO`: menjalankan dua `IO` berurutan, mengembalikan pasangan hasilnya.
- `f = Parser`: menjalankan dua parser berurutan (parser-parser tersebut menerima input berurutan), mengembalikan hasil berupa pasangan. Jika satu gagal, semuanya gagal.

Bisakah kalian mengimplemen fungsi-fungsi berikut? Pertimbangkan apa yang tiap fungsi lakukan jika `f` diganti dengan tipe-tipe di atas.

```
(*>)       :: Applicative f => f a -> f b -> f b
mapA       :: Applicative f => (a -> f b) -> ([a] -> f [b])
```

```
sequenceA  :: Applicative f => [f a] -> f [a]
replicateA :: Applicative f => Int -> f a -> f [a]
```

Monad

Bacaan tambahan:

- *Typeclassopedia*
- LYAH Bab 12: *A Fistful of Monads*
- LYAH Bab 9: *Input and Output*
- RWH Bab 7: *I/O*
- RWH Bab 14: *Monads*
- RWH Bab 15: *Programming with monads*

Motivasi

Dalam beberapa minggu terakhir, kita telah melihat bagaimana `Applicative` memungkinkan kita melakukan komputasi di dalam sebuah "konteks khusus". Contohnya antara lain, mengatasi kemungkinan gagal dengan `Maybe`, hasil yang lebih dari satu dengan `[]`, melihat semacam "lingkungan" menggunakan `((->) e)`, atau membuat parser dengan pendekatan kombinator seperti di tugas.

Akan tetapi, sejauh ini kita hanya melihat komputasi dengan struktur tetap, seperti menerapkan konstruktor data ke sejumlah argumen yang telah diketahui. Bagaimana jika kita tidak tahu struktur komputasinya di awal? Bagaimana jika kita ingin memutuskan apa yang ingin dilakukan berdasarkan hasil-hasil sebelumnya?

Sebagai contoh, ingat tipe `Parser` dari tugas, dan anggap kita telah menjadikannya anggota `Functor` dan `Applicative`:

```
newtype Parser a = Parser { runParser :: String -> Maybe (a, String) }

instance Functor Parser where
  ...

instance Applicative Parser where
  ...
```

Ingat bahwa nilai bertipe `Parser a` berarti sebuah parser yang bisa menerima sebuah `String` sebagai input dan mungkin menghasilkan nilai bertipe `a` beserta sisa `String` yang belum di-*parse*. Sebagai contoh, parser integer yang diberikan input `String` sebagai berikut

```
"143xkkj"
```

akan menghasilkan

```
Just (143, "xkkj")
```

Seperti yang kalian lihat di tugas, kita sekarang bisa menulis seperti

```
data Foo = Bar Int Int Char

parseFoo :: Parser Foo
parseFoo = Bar <$> parseInt <*> parseInt <*> parseChar
```

dengan asumsi kita memiliki fungsi `parseInt :: Parser Int` dan `parseChar :: Parser Char`. Anggota `Applicative` akan mengatasi kemungkinan kegagalan (jika *parsing* salah satu komponen gagal, *parsing* seluruh `Foo` akan gagal). Selain itu, dia juga akan melanjutkan ke bagian `String` yang belum di-*parse* untuk dijadikan input ke komponen selanjutnya.

Nah, sekarang kita akan parsing berkas yang mengandung deret angka sebagai berikut:

```
4 78 19 3 44 3 1 7 5 2 3 2
```

Dengan catatan, angka pertama menandakan panjang "grup" angka berikutnya. Angka setelah grup tersebut adalah panjang grup berikutnya, dan seterusnya. Jadi contoh di atas bisa dipecah menjadi grup-grup sebagai berikut:

```
78 19 3 44   -- grup pertama
1 7 5        -- grup kedua
3 2          -- grup ketiga
```

Contoh ini memang terlihat aneh, tapi ada kasus nyata di mana format suatu berkas mengikuti prinsip yang sama. Kita membaca semacam "*header*" yang memberitahu panjang dari suatu blok, atau di mana kita bisa menemukan sesuatu di dalam berkas, dan lain sebagainya.

Kita ingin menulis parser bertipe

```
parseFile :: Parser [[Int]]
```

Sayangnya, ini tidak mungkin dengan `Applicative`. Masalahnya ialah `Applicative` tidak memberikan kita cara untuk memutuskan apa yang akan dilakukan berdasarkan hasil sebelumnya. Kita harus memutuskan di awal operasi parsing seperti apa yang kita jalankan sebelum kita bisa melihat hasilnya.

Akan tetapi, tipe `Parser` *bisa* mendukung hal seperti ini. Hal ini diabstraksi dengan *type class* `Monad`.

Monad

Type class `Monad` didefinisikan sebagai berikut:

```
class Monad m where
  return :: a -> m a

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

  (>>)  :: m a -> m b -> m b
  m1 >> m2 = m1 >>= \_ -> m2
```

Terlihat familiar! Kita telah melihat method-method ini di dalam konteks `IO`, tapi sebenarnya mereka tidak spesifik hanya untuk `IO` saja.

`return` juga terlihat familiar karena bertipe sama dengan `pure`. Sebenarnya, tiap `Monad` harusnya juga merupakan `Applicative`, dengan `pure = return`. Alasan kita mempunyai keduanya ialah `Applicative` diciptakan *setelah* `Monad` ada cukup lama.

`(>>)` adalah versi khusus dari `(>>=)` (dimasukkan ke dalam `Monad` jikalau ada anggota yang ingin menyediakan implementasi yang lebih efisien, meskipun biasanya implementasi *default* sudah cukup). Jadi untuk memahami `(>>)`, kita harus memahami `(>>=)` terlebih dahulu.

Sebenarnya ada *method* lagi bernama `fail`, tapi hal ini adalah sebuah kesalahan. Kalian sebaiknya jangan pernah memakainya, jadi saya tak perlu menjelaskan (kalian bisa membacanya di * Typeclassopedia* jika tertarik).

`(>>=)` (disebut "*bind*") adalah di mana semua aksi berada! Mari perhatikan tipenya dengan seksama:

```
(>>=) :: m a -> (a -> m b) -> m b
```

`(>>=)` menerima dua argumen. Yang pertama ialah nilai bertipe `m a`. Nilai tersebut biasa disebut "nilai monadik" (*monadic values*), atau "komputasi". Ada juga yang menyarankan untuk disebut *mobit*. Kalian *tidak* boleh menyebutnya "monad", karena itu salah (konstruktor tipe `m` lah yang merupakan monad.) Intinya, mobit bertipe `m a` melambangkan sebuah komputasi yang menghasilkan sebuah (atau beberapa, atau tidak ada) nilai bertipe `a`, dan juga memiliki semacam "efek":

- `c1 :: Maybe a` adalah komputasi yang mungkin gagal, atau menghasilkan nilai bertipe `a` jika sukses.
- `c2 :: [a]` adalah komputasi yang menghasilkan (beberapa) `a`.
- `c3 :: Parser a` adalah komputasi yang mengkonsumsi bagian dari sebuah `String` dan (mungkin) menghasilkan sebuah `a`.
- `c4 :: IO a` adalah komputasi yang mungkin memiliki efek *I/O* lalu menghasilkan sebuah `a`.

Dan lain sebagainya. Sekarang, bagaimana dengan argumen kedua dari `(>>=)`? Sebuah fungsi bertipe `(a -> m b)` yang akan *memilih* komputasi berikutnya berdasarkan hasil dari komputasi pertama. Inilah yang dimaksud dengan `Monad`

bisa merangkum beberapa komputasi yang bisa memilih apa yang akan dilakukan tergantung dari hasil komputasi sebelumnya.

Jadi apa yang `(>>=)` lakukan ialah menggabungkan dua mobit menjadi satu. Mobit hasilnya yang lebih besar ini akan menjalankan yang pertama lalu yang kedua, dan mengembalikan hasil dari yang kedua. Hal penting yang perlu diingat ialah kita bisa menentukan mobit kedua mana yang akan dijalankan berdasarkan hasil dari yang pertama.

Implementasi *default* dari `(>>)` tentunya menjadi jelas sekarang:

```
(>>)  :: m a -> m b -> m b
m1 >> m2 = m1 >>= \_ -> m2
```

`m1 >> m2` menjalankan `m1` lalu `m2`, mengabaikan hasil dari `m1`.

Contoh

Mari mulai dengan membuat anggota `Monad` untuk `Maybe`:

```
instance Monad Maybe where
  return  = Just
  Nothing >>= _ = Nothing
  Just x  >>= k = k x
```

`return`, tentunya, hanyalah `Just`. Jika argumen pertama dari `(>>=)` adalah `Nothing`, maka seluruh komputasi akan gagal. Sebaliknya, jika `Just x`, kita terapkan argumen kedua pada `x` untuk memutuskan apa yang akan dilakukan berikutnya.

Kebetulan, cukup umum memakai huruf `k` untuk argumen kedua dari `(>>=)` karena `k` merupakan singkatan dari "kontinuasi". *I wish I was joking.*

Beberapa contoh:

```
check :: Int -> Maybe Int
check n | n < 10    = Just n
        | otherwise = Nothing

halve :: Int -> Maybe Int
halve n | even n    = Just $ n `div` 2
        | otherwise = Nothing

exM1 = return 7 >>= check >>= halve
exM2 = return 12 >>= check >>= halve
exM3 = return 12 >>= halve >>= check
```

Bagaimana anggota `Monad` untuk konstruktor list `[]`?

```haskell
instance Monad [] where
  return x = [x]
  xs >>= k = concat (map k xs)
```

Contoh sederhana:

```haskell
addOneOrTwo :: Int -> [Int]
addOneOrTwo x = [x+1, x+2]

exL1 = [10,20,30] >>= addOneOrTwo
```

Monad combinator

Satu hal bagus dari `Monad` ialah dengan hanya menggunakan `return` dan `(>>=)` kita bisa membuat banyak kombinator umum untuk menulis program dengan monad. Mari kita lihat beberapa contohnya.

Pertama, `sequence` menerima list berisi nilai-nilai monadik dan menghasilkan satu nilai monadik yang merupakan gabungan dari semuanya. Artinya, masing-masing monad memiliki sifat yang berbeda. Sebagai contoh, untuk `Maybe` ini berarti seluruh komputasi sukses hanya jika semua komputasi yang membangunnya sukses. Untuk kasus `IO`, ini berarti menjalankan komputasi secara berurutan. Untuk kasus `Parser` ini berarti menjalankan semua parser terhadap bagian-bagian input secara berurutan (dan sukses hanya jika semuanya sukses).

```haskell
sequence :: Monad m => [m a] -> m [a]
sequence [] = return []
sequence (ma:mas) =
  ma >>= \a ->
  sequence mas >>= \as ->
  return (a:as)
```

Dengan menggunakan `sequence` kita juga bisa menulis kombinator lainnya seperti

```haskell
replicateM :: Monad m => Int -> m a -> m [a]
replicateM n m = sequence (replicate n m)
```

Dan akhirnya kita bisa menulis parser yang kita inginkan, hanya dengan

```haskell
parseFile :: Parser [[Int]]
parseFile = many parseLine

parseLine :: Parser [Int]
parseLine = parseInt >>= \i -> replicateM i parseInt
```

(`many` juga disebut sebagai `zeroOrMore` di dalam tugas).