Modul Pertemuan ke 3

CMC101 Topik Dalam Pemrograman

Sumber :

- haskell.web.id : haskell indonesia
- id.wikipedia.org/wiki/Haskell

1. PEMROGRAMAN FUNGSIONAL

Dalam ilmu komputer, pemrograman fungsional (bahasa Inggris: Functional programming, disingkat FP) adalah paradigma pemrograman yang dimana suatu program komputer dijalankan dengan mengevaluasi ekspresi yang terdapat pada program komputer itu sendiri.

Pemrograman fungsional biasanya menggunakan fungsi yang murni agar dapat menghindari data yang dapat berubah-ubah, efek samping dari dijalankannya fungsi, dan variabel yang dapat diakses dari lebih dari satu fungsi.

Pemrograman fungsional dikategorikan menjadi pemrograman fungsional murni dan pemrograman fungsional tidak murni. Sederhananya, pemrograman fungsional murni mengacu pada pemrograman yang mana suatu fungsinya hanya dapat mengelola masukannya lalu memberikan semacam keluaran berdasarkan masukan tersebut. Artinya, fungsi tersebut tidak diizinkan untuk mengambil nilai pengubah global, melakukan pencetakan pesan ke layar, dan hal-hal lain yang termasuk pada istilah efek samping (side effect) pada pemrograman fungsional.

1.1. Bahasa Pemrograman

Contoh bahasa pemrograman yang mendukung pemrograman fungsional murni antara lain:

- Agda
- Clean
- Coq
- Elm
- Haskell
- Idris
- Lisp
- Miranda
- Mercury
- PureScript

Adapun contoh bahasa pemrograman yang mendukung pemrograman fungsional tidak murni antara lain:

- C++

- Clojure
- Elixir
- Erlang
- JavaScript
- Julia
- PHP
- Python
- Rust
- Scala

1.2. Paradigma Pemrograman Fungsional

Berlainan sekali dengan paradigma prosedural, program fungsional harus diolah lebih dari program prosedural (oleh pemroses bahasanya), karena itu salah satu keberatan adalah kinerja dan efisiensinya. Karena itu, dalam bahasa pemrograman fungsional, program adalah fungsi hasil komposisi dari fungsi-fungsi lain, apakah fungsi itu dasar atau hasil komposisi dari fungsi dasar. Bahasa pemrograman fungsional memperoleh hasil dengan cara mengaplikasikan fungsi terhadap argumen atau parameternya,yang juga dapat berupa fungsi.

Bahasa pemrograman fungsional menonjol dalam kemampuan struktur datanya. Karena bahasa ini tidak dibatasi oleh variabel yang berasosiasi dengan lokasi memori, maka sebuah struktur data cukup ditangani sebagai sebuah nilai.

1.2.1. Jenis pradigma fungsional

Paradigma fungsional memiliki banyak macam bahasa pemrograman, antara lain: Haskell, Lisp. Haskell merupakan paradigma fungsional yang malas dan murni. Hal ini disebabkan karena dalam haskell tidak mengevaluasi ekspresi-ekspresi yang digunakannya yang sebenarnya memang tidak diperlukan untuk menentukan jawaban bagi suatu masalah. Selain itu bahasa ini tidak memperbolehkan adanya efek samping (Efek samping adalah sesuatuyang mempengaruhi bagian di program. Misalnya suatu fungsi yang mencetak sesuatu ke layar yang mempengaruhi nilai dari variabel global. Tentu saja, suatu bahasa pemrograman yang tanpa efek samping akan menjadi sangat tidak berguna; Haskell menggunakan sebuah system monads untuk mengisolasi semua komputasi kotor dari program dan menampilkannya dengan cara yang aman.

Lisp adalah bahasa ekspresi, karena baik program maupun data dinyatakan sebagai ekspresi. Selain itu Lisp juga lebih mengarah dalam pemanfaatan artificial intelligence .

Bahasa pemrograman Haskell merupakan bahasa pemrograman yang sangat sederhana dan mudah dipelajari. Hal ini tidak lain disebabkan karena Haskell merupakan bahasa pemrograman fungsional murni. Oleh karena itu Haskell dapat:

1. Meningkatkan produktifitas programmer (Ericsson banyak memanfaatkan hasil percobaan Haskell dalam software telephony)
2. Lebih singkat, lebih jelas dan kode-kodenya mudah dibaca
3. Errornya semakin sedikit dan reabilitynya lebih tinggi
4. Membuat jarak antara programmer dengan bahasa itu lebih pendek
5. Waktu untuk membuat program menjadi lebih singkat

Selain itu, dalam Haskell tidak ada varibel yang berubah, tidak ada efek samping dari penggunaan sebuah fungsi, tidak ada perulangan, dan tidak ada program order.

Lisp telah tersebar luas dan merupakan salah satu bahasa pilihan untuk aplikasi seperti artificial intelligence.Bahasa fungsional pada umumnya, dan LISP pada khususnya, memainkan perananpenting dalam definisi bahasa. Sebuah definisi bahasa harus ditulis ke dalam notasinotasi, disebut meta-bahasa(meta-language) atau bahasa yang didefinisikan (defininglanguage), dan bahasa yang didefinisikan cenderung menjadi fungsional. Dalam kenyataannya, implementasi LISP pertama dimulai, ketika LISP digunakan untuk mendefinisikan dirinya sendiri.

1.2.2. Keuntungan dari paradigma fungsional

- Singkat

  Karena program cenderung lebih ringkas dibandingkan program terstruktur.

- Mudah dimengerti

  Program fungsional seringkali lebih mudah untuk dimengerti, contohnya quicksort, tidak terlalu diperlukan pengetahuan mengenai quicksort.

- Tidak ada tumpukan memori

  Tidak ada kemungkinan memperlakukan integer sebagai pointer, atau dilanjutkan dengan pointer null.

- Manajemen memori yang terintegrasi

  Kebanyakan program rumit perlu mengalokasikan memori dinamis dari tumpukan (heap). Setiap bahasa fungsional memudahkan pemrogram dari beban manajemen penyimpanan tersebut. Penyimpanan dialokasikandan diinisialisaikan secara implisit, dan diselamatkan secara otomatis oleh kolektor sampah.

1.2.3. Operator

Dalam paradigma fungsional, operator yang berlaku masih sama dengan paradigma procedural. Berikut adalah jenis operator yang berlaku;

a. aritmatika (+,-,/,*)

b. logika (and.or,not,xor)

c. boolean

d. untai

e. himpunan

f. relasi

1.2.4. Tipe data

a. Tipe Polimorfis

Tipe yang dalam beberapa cara terukur di atas semua tipe. Ekspresi tipe polimorfis menguraikan keluarga dari tipe-tipe. Sebagai contoh, ( )[a] adalah keluarga dari tipe di mana untuk setiap tipe a berisi tipe list dari a. List dari integer (seperti [1,2,3]), list dari karakter ([‘a’,’b’,’c’]), sekalipun list dari list integer, dll, adalah anggota dari keluarga ini. (Sebagai catatan [2,’b’] bukan contohyang valid karena tidak ada tipe tunggal yang berisi 2 dan ‘b’).

b. Tipe User-Defined

Kita dapat menentukan tipe data sendiri dalam Haskell menggunakan deklarasi data. Tipe penting yang sudah dikenal oleh Haskell adalah nilai kebenaran:

```
data Bool = False | True
```

Tipe yang didefinisikan di sini adalah Bool yang mempunyai dua nilai, yaitu True dan False. Tipe Bool adalah sebuah contoh dari tipe konstruktor, dan True dan False adalah data konstruktor (atau konstruktor saja). Dengan cara yang sama kita dapat mendefinisikan tipe warna:

```
data Color = Red | Green | Blue | Indigo | Violet
```

Bool dan Color dalam contoh di atas merupakan tipe enumerasi. Berikut contoh dari tipe dengan hanya satu data konstruktor:

```
data Point a = Pt a a
```

Karena merupakan konstruktor tunggal, tipe seperti Point sering disebut tipe tuple, karena merupakan produk kartesian (dalam hal ini biner) dari tipe lain. Berlawanan dengan itu, tipe multi-konstruktor seperti Booldan Color disebut union dari tipe sum.

c. Tipe Rekursif

Tipe dapat juga rekursif, seperti tipe dalam pohon biner:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

Di sini didefinisikan tipe polimorfis pohon biner yang mana elemen-elemennya adalah node Leaf berisi nilai dari tipe a, atau node internal (“branch”) berisi dua sub-tree (rekursif). Saat membaca deklarasi data seperti ini, ingat bahwa Tree adalah tipe konstruktor, di mana Branch dan Leaf adalah data konstruktor. Di samping menciptakan koneksi antara konstruktor-konstruktor ini, deklarasi di atas mendefinisikan tipe berikut untuk Branch dan Leaf:

```
Branch :: Tree a -> Tree a -> Tree a

Leaf :: a -> Tree a
```

Dengan contoh di atas telah didefinisikan suatu tipe yang cukup kaya untuk mendefinisikan beberapa fungsi (rekursif) yang menggunakannya. Sebagai contoh akan didefinisikan sebuah fungsi fringe yang mengembalikan sebuah list dari semua elemen dalam daun dari sebuah pohon dari kiri ke kanan.

d. Tipe Sinonim

Untuk kenyamanan, Haskell menyediakan cara untuk mendefinisikan tipe sinonim, yaitu nama untuk tipe yang sering dipakai. Tipe sinonim dibuat menggunakan deklarasi type. Berikut beberapa contoh:

```
type String = [Char]

type Person = (Name,Address)

type Name = String

data Address = None | Addr String
```

Tipe sinonim tidak mendefinisikan tipe baru, tetapi memberi nama baru kepada tipe-tipe yang sudah ada. Sebagai contoh tipe Person -> Name setara dengan (String,Address) -> String. Nama yang baru seringkali lebih pendek dari tipe sinonimnya, tetapi itu bukan satu-satunya tujuan dari tipe sinonim: tipe sinonim meningkatkan kemudahan membaca sebuah program dengan menjadi lebih mnemonik. Bahkan tipe polimorfis sekalipun dapat diberi nama baru:

```
type AssocList a b = [(a,b)]
```

Ini merupakan “tipe asosiasi” yang mengasosiasikan nilai dari tipe a dengan nilai dari tipe b.

1.2.5. Notasi

a) Program adalah model solusi persoalan informatik, berisi kumpulan informasi penting yang mewakili persoalan.
b) b. Dalam konteks fungsional, program digambarkan dalam : himpunan nilai type, dengan nilainya adalah konstanta.
c) Fungsi di sini merupakan asosiasi antara 2 type yaitu domain dan range.
d) Domain range dapat berupa : type dasar dan type terkomposisi (bentukan).
e) Penulisan suatu program fungsional, dipakai bahasa ekspresi ada tiga macam bentuk komposisi ekspresi adalah ekspresi fungsional dasar, konditional dan rekursif.
f) Pemrograman fungsional didasari atas analisa top down. Analisa top down dalam pemrograman fungsional: Problema, Spesifikasi dan Dekomposisi .
g) Fungsi pada analisa topdown adalah strukturisasi teks. Sebuah fungsi mewakili sebuah tingkatan abstraksi.

Konstruksi Program Fungsional

a) Definisi Fungsi : Menentukan nama dan mendefinisikan domain dan range dari fungsi.

b) Spesifikasi fungsi : menentukan “arti” dari fungsi. Contoh : Fungsi bernama Pangkat3(x) artinya menghitung pangkat tiga x seperti pada umumnya.
c) Realisasi fungsi : mengasosiasikan pada nama fungsi, sebuah ekspresi fungsional dengan parameter formal yang cocok. Contoh : mengasosiasikan pada Pangkat Tiga: a*a*a atau a3 dengan a adalah nama parameter formal. Parameter formal fungsi adalah nama yang dipilih untuk mengasosiasikan domain dan range.
d) Aplikasi fungsi : memakai fungsi dalam ekspresi, yaitu dengan menggantikan semua nama parameter formal dengan nilai. Dengan aplikasi fungsi, akan dievaluasi ekspresi fungsional. Contoh : Pangkat Tiga (2) + Pangkat Tiga(3).
e) Argumen pada saat dilakukan aplikasi fungsi disebut parameter actual. Notasi untuk menuliskan program fungsional disebut dengan notasi fungsional, dimana terdiri dari empat bagian sesuai dengan tahapan pemrograman. Contoh: generic(template) teks program dalam notasi fungsional.

2. BAHASA PEMROGRAMAN HASKELL

2.1. Apa itu Haskell?

Haskell adalah bahasa pemrograman fungsional murni. Nama bahasa pemrograman Haskell diambil dari nama seseorang matematikawan Haskell Curry, yang terkenal akan karyanya di bidang combinatory logic. Haskell hanya mengenal expression dan equation.

Hello World

Berikut ini adalah contoh program sederhana yang akan mencetak kalimat "Hello, World!".

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

Penjumlahan Sederhana

Berikut ini adalah contoh program sederhana yang akan mencetak hasil penjumlahan antara 2 dan 10.

```
main = print (2 + 10)
```

Cetak Nama

Berikut ini adalah contoh program sederhana yang akan meminta nama pengguna dan mencetaknya kembali.

```
main = do
putStrLn "Masukan nama Anda:"
nama <- getLine
putStrLn ("Selamat datang, " ++ nama ++ "!")
```

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 dan setiap orang memiliki favoritnya masing-masing 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, maka lahirlah Haskell.

2.2. Fungsional

Tidak ada pengertian tepat dan baku untuk istilah "fungsional". Tapi ketika kita mengatakan bahwa Haskell adalah bahasa pemrograman fungsional, maka biasanya mengingatkan 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.

Bagaimana mungkin bisa mengerjakan sesuatu tanpa mutasi dan efek samping? Tentunya ini memerlukan perubahan cara berpikir, dengan perubahan cara berpikir 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.
- 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. 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.

2.3. Tema

Dalam materi ini, kita akan fokus pada tiga tema utama.

2.3.1. Tipe

Type system di Haskell berfungsi untuk :

- *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.

2.3.2. Abstraksi

“Don’t Repeat Yourself” adalah merupakan *term* 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. 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.

2.3.3. Wholemeal programming

Quote 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 menurut Richard Bird adalah "indexities", yakni kita harus fokus 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, cukup menuliskan

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

Pada modul ini kita akan mengeksplorasi pergeseran cara berpikir dengan cara pemrograman seperti ini, dan memeriksa bagaimana dan mengapa Haskell membuatnya menjadi mungkin.

Literate Haskell

Dokumen *literate* Haskell berekstensi .lhs, sedangkan kode sumber *non-literate* Haskell berekstensi .hs.

2.3.4. 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; maka akan didapati 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 seharusnya dibaca "x memperoleh 4" atau "*assign* 4 ke x", tetapi harus dibaca "x *didefinisikan sebagai* 4".

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 2^29, tapi ukuran pastinya bergantung pada arsitektur komputer masing-masing. Sebagai contoh, di mesin 64-bit saya kisarannya sampai 2^63. Kita 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*)

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 b1 = True b2 = False -- Karakter unicode c1, c2, c3 :: Char c1 = 'x' c2 = 'Ø' c3 = 'ダ' -- String adalah list dari karakter dengan sintaks khusus s :: String s = "Hai, Haskell!"

2.3.5. 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:

```
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*:

```
-- 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*.

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 memerlukan 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
```

Hal ini terlihat tidak efisien. 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.

2.3.6. Pesan kesalahan (*error message*)

Pesan kesalahan dari GHC bisa panjang. 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.