STUDI BAHASA PEMROGRAMAN HASKELL

SEBAGAI BAHASA PEMROGRAMAN FUNGSIONAL

TUGAS AKHIR

Diajukan sebagai Salah Satu Syarat

untuk Memperoleh Gelar Sarjana Jurusan Teknik Informatika

oleh:

Nama : Arif Hidayat

No. Mahasiswa : 01 523 324

JURUSAN TEKNIK INFORMATIKA

FAKULTAS TEKNOLOGI INDUSTRI

UNIVERSITAS ISLAM INDONESIA

YOGYAKARTA

2007

PERNYATAAN KEASLIAN TUGAS AKHIR

Saya yang bertanda tangan di bawah ini,

Nama : Arif Hidayat

No. Mahasiswa : 01 523 324

Menyatakan bahwa Tugas Akhir dengan judul :

"Studi Bahasa Pemrograman Haskell
Sebagai Bahasa Pemrograman Fungsional"

Yang diajukan untuk diuji pada tanggal 24 Juli 2007 adalah hasil karya saya. Dengan ini saya menyatakan bahwa seluruh komponen dan isi didalam Laporan Tugas Akhir ini adalah hasil karya saya sendiri. Apabila dikemudian hari terbukti bahwa ada beberapa bagian dari karya ini adalah bukan karya saya sendiri, maka saya siap menanggung resiko dan konsekuensi apapun.

Demikian pernyataan saya ini saya buat, semoga dapat dipergunakan sebagaimana mestinya.

Yogyakarta, 24 Juli 2007

(Arif Hidayat)

LEMBAR PENGESAHAN PEMBIMBING

STUDI BAHASA PEMROGRAMAN HASKELL SEBAGAI BAHASA PEMROGRAMAN FUNGSIONAL

TUGAS AKHIR

oleh :

Nama : **ARIF HIDAYAT**

No Mhs : **01 523 324**

Yogyakarta, 24 Juli 2007

Pembimbing

(Taufiq Hidayat, S.T.,M.CS)

LEMBAR PENGESAHAN PENGUJI

STUDI BAHASA PEMROGRAMAN HASKELL SEBAGAI BAHASA PEMROGRAMAN FUNGSIONAL

TUGAS AKHIR

oleh :

Nama : **ARIF HI**DAYAT
No. Mahasiswa : **01 523 324**

Telah Dipertahankan di Depan Sidang Penguji Sebagai Salah Satu Syarat
Untuk Memperoleh Gelar Sarjana Jurusan Teknik Informatika
Fakultas Teknologi Industri Universitas Islam Indonesia

Yogyakarta, 24 Juli 2007

Tim Penguji,

Taufiq Hidayat, S.T.,M.CS.
Ketua

Syarif Hidayat, S.Kom
Anggota I

Ami Fauziah, S.T., M.T.
Anggota II

Mengetahui,
Ketua Jurusan Teknik Informatika
Fakultas Teknologi Industri
Universitas Islam Indonesia

(Prayudi, S. Si., M. Kom)

HALAMAN PERSEMBAHAN

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ

Dengan perasaan bahagia dan rasa syukur atas limpahan karunia-Nya kupersembahkan Tugas Akhir ini untuk :

Kedua orang tuaku tercinta yang selalu memberikan segala kasih sayang, perhatian, doa restu, cinta yang abadi dengan penuh kesabaran, keikhlasan dan ketulusan serta pengorbanan yang tulus dan murni.

Kakak-kakak dan adikku tersayang yang selalu memberikanku doa, motivasi, dorongan dan semangat. Kasih sayang yang kalian berikan selalu membuatku tersenyum.

MOTTO

۞ قُلْ يَٰعِبَادِيَ ٱلَّذِينَ أَسْرَفُوا۟ عَلَىٰٓ أَنفُسِهِمْ لَا تَقْنَطُوا۟ مِن رَّحْمَةِ ٱللَّهِ ۚ إِنَّ ٱللَّهَ يَغْفِرُ ٱلذُّنُوبَ جَمِيعًا ۚ إِنَّهُۥ هُوَ ٱلْغَفُورُ ٱلرَّحِيمُ ﴿٥٣﴾

"Katakanlah: "Hai hamba-hamba-Ku yang malampaui batas terhadap diri mereka sendiri, janganlah kamu berputus asa dari rahmat Allah. Sesungguhnya Allah mengampuni dosa-dosa semuanya. Sesungguhnya Dia-lah yang Maha Pengampun lagi Maha Penyayang."

(QS: Az Zumar : 53)

إِنَّ مَعَ ٱلْعُسْرِ يُسْرًا ﴿٥﴾ فَإِذَا فَرَغْتَ فَٱنصَبْ ﴿٧﴾ وَإِلَىٰ رَبِّكَ فَٱرْغَب ﴿٨﴾

"Sesungguhnya sesudah kesulitan itu ada kemudahan. Maka apabila kamu Telah selesai (dari sesuatu urusan), kerjakanlah dengan sungguh-sungguh (urusan) yang lain Dan Hanya kepada Tuhanmulah hendaknya kamu berharap."

(QS: Al Insyirah : 6-8)

ٱلْحَقُّ مِن رَّبِّكَ ۖ فَلَا تَكُونَنَّ مِنَ ٱلْمُمْتَرِينَ ﴿١٤٧﴾

"Kebenaran itu adalah dari Tuhanmu, sebab itu jangan sekali-kali kamu termasuk orang-orang yang ragu."

(QS : Al_Baqarah :147)

مَنْ جَدَّ وَجَدَ

"Barang siapa bersungguh – sungguh, maka dia akan mendapatkannya"

مَنْ حُرِمَ الأُصُول حُرِمَ الوُصُل

"Barang siapa terhalang dari dasarnya, maka dia terhalang dari tujuannya"

KATA PENGANTAR

بِسْمِ اللهِ الرَّحْمَنِ الرَّحِيْمِ

السَّلَامُ عَلَيْكُمْ وَرَحْمَةُ اللهِ وَبَرَكَاتُهُ

Segala puji hanya bagi Allah, Rabb semesta alam, Pengatur langit dan bumi, Pemelihara seluruh makhluk, Pengutus seluruh rasul *shalawatullahi wa salaamuhu 'alaihim* kepada *mukallaf*. Kami memuji-Nya atas segala nikmat yang Dia limpahkan, kami memohon tambahan keutamaan dan karunia-Nya, memohon pertolongan-Nya, dan memohon Ampunan-Nya. Sesungguhnya, penyusunan laporan tugas akhir dengan judul "**Studi Bahasa Pemrograman Haskell Sebagai Bahasa Pemrograman Fungsional**" ini dapat diselesaikan hanya karena izin Allah semata. Semoga Allah senantiasa melimpahkan shalawat dan salam kepada Nabi Muhammad, kepada seluruh nabi dan rasul, seluruh keluarga mereka, dan segenap orang – orang shalih.

Laporan tugas akhir ini disusun untuk melengkapi salah satu syarat guna memperoleh gelar Sarjana Jurusan Teknik Informatika Universitas Islam Indonesia. Tugas akhir sendiri merupakan salah satu kegiatan kurikuler bersifat wajib, yang dimaksudkan untuk memberikan kesempatan bagi mahasiswa agar dapat melihat dan bersentuhan langsung dengan dunia di luar kampus guna mengaplikasikan ilmu pengetahuan yang didapat di bangku kuliah.

Pada kesempatan ini penyusun ingin mengucapkan banyak terima kasih kepada pihak-pihak yang mempunyai andil besar dalam pelaksanaan dan penyelesaian laporan tugas akhir ini, antara lain :

1. Kedua orang tuaku yang paling berjasa dalam seluruh hidupku, dan seluruh keluarga yang tercinta.
2. Bapak Fathul Wahid S.T., M.Sc sebagai Dekan Fakultas Teknologi Industri Universitas Islam Indonesia.
3. Bapak Yudi Prayudi, S.Si, M.Kom selaku Ketua Jurusan Informatika Fakultas Teknologi Industri Universitas Islam Indonesia.
4. Bapak Taufiq Hidayat, ST. MCS selaku Pembimbing Tugas Akhir. Terima kasih atas segala kesabarannya, bantuan, dukungan, pengetahuan dan semua kemudahan yang selalu diberikan.
5. Bapak, Ibu dosen teknik Informatika dan dosen-dosen Universitas Islam Indonesia. Terimakasih atas semua ilmu yang diberikan.
6. Ryan, Hendro, Niko, Endang Ahmad, Osama dan *Binner Community 01'*, yang memberikan semangat, dorongan, bantuan, maupun gangguannya sehingga penyusun dapat meyelesaikan tugas akhir dengan baik.
7. Semua pihak yang tidak bisa disebut satu persatu yang telah membantu sehingga laporan tugas akhir ini dapat terselesaikan

Di tengah keterbatasan penyusun dalam laporan tugas akhir ini, penyusun berharap kiranya laporan ini bermanfaat bagi pambaca. Semoga Allah SWT membimbing dan menyertai setiap langkah kita.

وَالسَّلَامُ عَلَيْكُمْ وَرَحْمَةُ اللهِ وَبَرَكَاتُهُ

Yogyakarta, 11 Juli 2007

Penyusun

SARI

Bahasa pemrograman semakin berkembang seiring dengan perkembangan zaman, baik pada pemrograman fungsional maupun pemrograman *imperative*. Setiap bahasa pemrograman memiliki kelebihan dibandingkan bahasa pemrograman yang lain. Terkadang sebuah bahasa pemrograman baru dibuat berdasarkan bahasa pemrograman yang telah ada yang digunakan untuk menyempurnakan kekurangan – kekurangannya.

Haskell merupakan salah satu pemrograman fungsional yang dapat dikatakan sebuah bahasa pemrograman baru. Bahasa pemrograman Haskell telah menjadi standard bahasa pemrograman fungsional dewasa ini. Haskell memiliki banyak fungsi dasar yang mendukung dalam sebuah pemrograman. Haskell mendukung konsep – konsep dasar pemrograman fungsional.

Hasil analisis menunjukkan bahwa dalam pemrograman Haskell mendukung tipe – tipe data abstrak, mudah dalam pendeklarasian tipe bentukan, dan memberikan kemudahan – kemudahan lain dalam sebuah pemrograman.

Kata kunci : Bahasa Pemrograman, Haskell.

TAKARIR

Abstract	Tidak nyata
Algebraic	Secara Aljabar
Binary Tree	Pohon biner
Constructor	Pembentuk
Convertion	Perubahan
Hardware	Perangkat Keras
Identifier	Pengenal
Imperative	Tidak boleh tidak
Input	Masukan
Left Associative	Sifat sebuah persamaan dimana pemberian tanda kurung tidak mempengaruhi hasil.
Numeric	Berdasarkan angka
Pharentesis	Tanda kurung
Primitive	Sederhana
Record	Merekam
Statement	Pernyataan
Storage device	Media Penyimpanan

DAFTAR ISI

HALAMAN JUDUL.....i
PERNYATAAN KEASLIAN TUGAS AKHIR.....ii
LEMBAR PENGESAHAN DOSEN PEMBIMBING.....iii
LEMBAR PENGESAHAN DOSEN PENGUJI.....iv
HALAMAN PERSEMBAHAN.....v
HALAMAN MOTTO.....vi
KATA PENGANTAR.....vii
SARI.....ix
TAKARIR.....x
DAFTAR ISI.....xi
DAFTAR GAMBAR.....xiii
BAB I PENDAHULUAN..... 1
1.1 Latar Belakang..... 1
1.2 Rumusan Masalah..... 2
1.3 Batasan Masalah..... 2
1.4 Tujuan Penelitian..... 2
1.5 Manfaat Penelitian..... 2
1.6 Metode Penelitian..... 3
1.6.1 Metode Pengumpulan Data..... 3
1.6.2 Metode Analisis..... 3
1.7 Sistematika Penulisan..... 3
BAB II LANDASAN TEORI..... 5
2.1 Tipe Data..... 5
2.1.1 Tipe Data *Primitive*..... 5
2.1.2 Tipe *Array*..... 6
2.1.3 Tipe *Record*..... 7
2.1.4 Tipe Data *Abstract*..... 7
2.2 Statement..... 8

2.3 Rekursif 10
2.4 Lambda Calculus 11
2.4.1 Deskripsi Informal 11
2.4.2 Definisi Formal 13
BAB III KONSEP PEMROGRAMAN FUNGSIONAL 16
3.1 Definisi Bahasa Pemrograman Fungsional 16
3.2 Tipe : Value dan Operasi 16
3.3 Deklarasi Fungsi 18
3.4 Pendekatan Evaluasi Expresi 19
3.5 Lexical Scope 22
3.6 Lambda Calculus 22
BAB IV KONSEP PEMROGRAMAN HASKELL 26
4.1 Tipe : Value dan Operasi 26
4.2 Deklarasi Fungsi 28
4.3 Pendekatan Evaluasi Ekspresi 29
4.4 Lexical Scope 30
4.5 Lambda Calculus 31
BAB V ANALISIS BAHASA PEMROGRAMAN HASKELL 34
5.1 Tipe Bentukan 34
5.2 Struktur Data 35
5.3 Binding 45
5.4 Input/Output 48
5.5 Modularitas 50
BAB VI PENUTUP 52
6.1 Kesimpulan 52
6.2 Saran 52
DAFTAR PUSTAKA 54

Gambar 5.24 Fungsi Untuk Mengetahui Daun Pada Binary Tree Dengan Haskell..42

Gambar 5.25 Fungsi Untuk Mengetahui Data Level Tertentu Pada Binary Tree....42

Gambar 5.26 Contoh Array Pada Haskell.....43

Gambar 5.27 Contoh Array Yang Tidak Terisi Penuh Pada Haskell.....43

Gambar 5.28 Fungsi Untuk Mengisi Elemen Array Pada Haskell.....44

Gambar 5.29 Contoh Pendeklarasian Array Pada C++.....45

Gambar 5.30 Contoh Pendeklarasian Array Yang Terisi Pada C++.....45

Gambar 5.31 Penggantian Elemen Array Pada C++.....45

Gambar 5.32 Source Code Fungsi Akar Persamaan Parabola Pada Haskell.....46

Gambar 5.33 Contoh Penggunaan Binding Pada Haskell.....46

Gambar 5.34 Fungsi Akar Persamaan Parabola dengan C++.....47

Gambar 5.35 Source Code Penggunaan Fungsi Input/Output Pada Haskell.....48

Gambar 5.36 Lanjutan Source Code Input/Output.....49

Gambar 5.37 Source Code Penggunaan Input/Output pada C++.....49

Gambar 5.38 Lanjutan Source Input/Output Pada C++.....50

Gambar 5.39 Source Code Modul Tree.hs.....50

Gambar 5.40 Source Code Modul Main.hs.....50

Gambar 5.41 Source Code Modul Tree.h.....51

Gambar 5.42 Source Code Modul Main.cpp.....51

DAFTAR GAMBAR

Gambar 5.1 Source Code Tipe Bentukan dengan Haskell.....35

Gambar 5.2 Source Code Tipe Bentukan dengan C++.....35

Gambar 5.3 Contoh List Pada Haskell.....36

Gambar 5.4 Struktur List.....36

Gambar 5.5 Source Code Untuk mereperesentasikan List Pada C++.....36

Gambar 5.6 Source Code fungsi pembentukan List Pada C++.....37

Gambar 5.7 Penambahan Elemen Pada Awal List Pada Haskell.....37

Gambar 5.8 Source Code Inisialisasi Node Pertama List dengan C++.....37

Gambar 5.9 Source Code Pengecekan Elemen Pertama List Dengan C++.....37

Gambar 5.10 Source Code Fungsi Penambahan Elemen Awal List Dengan C++...38

Gambar 5.11 Penambahan Elemen Akhir List Dengan Haskell.....38

Gambar 5.12 Source Code Fungsi Penambahan Akhir List Dengan C++.....38

Gambar 5.13 Fungsi Untuk Pengurutan Elemen List Pada Haskell.....39

Gambar 5.14 Fungsi Untuk Pengurutan Elemen List dari yang Terbesar.....39

Gambar 5.15 Fungsi Untuk Penyisipan List Pada Haskell.....39

Gambar 5.16 Penyisipan Elemen Pada List Pada Haskell.....39

Gambar 5.17 Representasi Binary Tree Pada Haskell.....40

Gambar 5.18 Struktur Binary Tree.....40

Gambar 5.19 Contoh Binary Tree Pada Haskell.....40

Gambar 5.20 Representasi Binary Tree Pada C++.....41

Gambar 5.21 Fungsi Menghitung Jumlah Daun Binary Tree Pada Haskell.....41

Gambar 5.22 Penggunaan Fungsi count_leaf.....41

Gambar 5.23 Source Code Fungsi Menghitung Node Pada Binary Tree.....41

BAB I

PENDAHULUAN

1.1 Latar Belakang

Dengan semakin rumitnya masalah yang harus dipecahkan maka bermunculan berbagai macam cara maupun metode untuk memecahkan masalah tersebut. Cara tersebut antara lain dapat berupa algoritma dan bahasa pemrograman baru.

Haskell merupakan salah satu bahasa pemrograman fungsional yang menjadi standard bahasa pemrograman fungsional dewasa ini. Bahasa Haskell mempunyai beberapa aplikasi yang mampu berjalan di beberapa platform sistem operasi. Dalam pengembangannya, bahasa ini telah digunakan dalam berbagai aplikasi.

Haskell dibuat oleh sebuah komite yang dibentuk pada tahun 1987 untuk mendesain sebuah bahasa pemrograman yang stabil untuk pengembangan aplikasi nyata, dan alat yang dapat membantu bahasa fungsional. Bahasa terakhir yang resmi dikeluarkan oleh organisasi standardisasi adalah Haskell 98.

1.2 Rumusan Masalah

Dari latar belakang masalah tersebut diatas didapatkan rumusan sebagai berikut:

"Menganalisis bahasa pemrograman Haskell sebagai bahasa pemrograman Fungsional."

1.3 Batasan Masalah

Tugas Akhir ini disesuaikan dengan rumusan masalah yang telah disebutkan. Agar pembahasan tidak meluas diberikan batasan – batasan masalah yang meliputi hal – hal sebagai berikut :

1. Analisis dilakukan dalam tinjauan bahasa fungsional.
2. Analisis juga dilakukan terhadap fasilitas tambahan.
3. kompiler yang digunakan adalah GHC versi 6.4.1.

1.4 Tujuan Penelitian

Sesuai dengan judul dan latar belakang masalah, maka sasaran dari studi ini adalah untuk mengetahui karakteristik bahasa pemrograman Haskell dan menyimpulkan masalah yang lebih tepat jika diselesaikan dengan Haskell.

1.5 Manfaat Penelitian

Manfaat yang diharapkan dari penelitian ini adalah untuk mengetahui kelebihan dan kekurangan bahasa Haskell dalam tinjuan sebagai bahasa pemrograman fungsional.

1.6 Metode Penelitian

Terdapat 2 tahapan metode yang digunakan dalam menyelesaikan penelitian ini, yaitu metode pengumpulan data, dan metode analisis.

1.6.1 Metode Pengumpulan Data

Metode pengumpulan data adalah metode yang digunakan untuk mencari dan mengumpulkan data atau teori yang berhubungan dengan penelitian yang dilakukan. Metode yang dilakukan dengan mempelajari teori-teori yang berhubungan dengan penelitian serta literatur-literatur lain yang dapat membantu untuk memecahkan masalah yang berkaitan dengan penelitian serta dengan melakukan konsultasi secara berkesinambungan dengan dosen pembimbing.

1.6.2 Metode Analisis

Tahapan ini merupakan tahapan akhir dari penelitian. Analisis dilakukan terhadap kelebihan dan kekurangan bahasa Haskell sebagai bahasa pemrograman bahasa fungsional.

1.7 Sistematika Penulisan

Sistematika penulisan dalam laporan Tugas Akhir adalah sebagai berikut:

BAB I Pendahuluan

Menjelaskan mengenai latar belakang, rumusan masalah, batasan, tujuan, manfaat, metodologi penelitian serta sistematika penulisan.

BAB II Landasan Teori

Menjelaskan tentang teori – teori bahasa pemrograman yang meliputi pengertian dan macam – macam tipe data, statement, serta lambda calculus.

BAB III Konsep Pemrograman Fungsional

Menjelaskan tentang dasar – dasar pemrograman fungsional yang meliputi pengertian bahasa pemrograman fungsional dan konsep dasar bahasa pemrograman fungsional.

BAB IV Konsep Bahasa Pemrograman Haskell

Menjelaskan tentang bahasa pemrograman Haskell secara umum yang meliputi pembahasan mengenai komponen – komponen yang ada dalam bahasa pemrograman Haskell.

BAB V Analisis Bahasa Pemrograman Haskell

Berisi tentang analisis terhadap bahasa pemrograman Haskell sebagai bahasa pemrograman fungsional.

BAB VI Penutup

Memuat kesimpulan dan saran.

BAB II

LANDASAN TEORI

2.1 Tipe Data

Tipe data adalah sebuah domain yang mendefinisikan nilai dari sekumpulan data dan operasi yang akan dikenakan pada data tersebut. Tipe data dalam pemrograman computer meliputi tipe *primitive* (seperti *integer*, *character*), *tuples*, *record*, tipe data *algebraic*, tipe data *abstract*, dan tipe fungsi. Sebuah tipe data mendeskripsikan representasi, interpretasi, dan struktur dari nilai yang akan di manipulasi oleh algoritma atau objek yang disimpan dalam *memory* komputer atau *storage device* yang lain.

2.1.1 Tipe Data *Primitive*

Tipe data yang tidak didefinisikan sebagai bentuk tipe data lain disebut dengan tipe data primitive. Hampir semua bahasa pemrograman menyediakan sekumpulan tipe data primitive. Beberapa dari tipe data primitive adalah refleksi dari hardware (sebagai contoh tipe integer).

Tipe data primitive dalam suatu bahasa pemrograman digunakan bersama satu atau lebih tipe *constructor*, untuk membentuk tipe terstruktur. Diantara yang termasuk tipe data *primitive* adalah sebagai berikut :

1. Tipe *Numeric*

   Tipe *numeric*, sesuai dengan namanya mendefinisikan angka – angka. Banyak bahasa pemrograman awal yang hanya mempunyai

tipe data *numeric* saja. Tipe data primitive terbagi menjadi beberapa tipe, dan diantaranya adalah tipe data *integer* dan *decimal*.

2. Tipe *Boolean*

   Tipe *Boolean* mungkin merupakan tipe yang paling sederhana dari seluruh tipe data yang dikenal dalam bahasa pemrograman. Tipe ini hanya memiliki dua nilai kemungkinan yaitu *true* dan *false*.

3. Tipe *Character*

   Tipe *character* adalah tipe yang mendefinisikan data sebagai huruf. Data *character* disimpan dalam computer dalam bentuk kode numeric. Biasanya digunakan pengkodean 8-bit code ASCII (*American Standard Code for Information Interchange*), yang menggunakan nilai 0 sampai 127 unuk mengkodekan 128 *character* yang berbeda.

4. Tipe *String*

   String merupakan sebuah tipe data yang nilai – nilainya terdiri dari rangkaian character. *String* biasa digunakan untuk memberikan label *input* dan *input/output* pada semua jenis data yang biasanya diselesaikan dalam bentuk *string*. Beberapa bahasa pemrograman mendefinisikan *string* sebagai tipe data bentukan.

2.1.2 Tipe *Array*

Tipe *array* adalah salah satu tipe bentukan tersruktur. *Array* dibutuhkan untuk menyimpan serangkaian elemen yang bertipe sama, berstruktur homogen, yang disebut tipe dasar. Letak masing masing elemen dalam array diidentifikasikan dengan sebuah indeks.

Elemen – elemen dari *array* dapat diakses secara acak dengan aturan tertentu, yaitu dengan mengetahui indeks dari elemen yang akan diakses. Berikut adalah contoh penggunaan tipe *array* dalam bahasa Pascal :

```
Type arrant = array [1..20] of integer;
```

2.1.3 Tipe *Record*

Sebuah *record* memungkinkan untuk pengumpulan bermacam – macam elemen data yang setiap elemen tersebut diidentifikasikan dengan nama. Sebagai contoh adalah informasi mengenai mahasiswa yang meliputi nama, nomor mahasiswa, nilai, alamat dan lain – lain. Tipe *record* digunakan untuk mengumpulkan data dengan beberapa tipe yang berbeda dan tidak diproses dengan cara yang sama.

Dengan kata lain, tipe *record* merupakan salah satu tipe bentukan terstruktur yang setiap datanya terdiri dari beberapa elemen yang disebut dengan *field*, dimana setiap field menggambarkan informasi tertentu. Berikut adalah contoh tipe record dalam bahasa Pascal :

```
Type Mahasiswa = Record
      Nama : String;
      NIM : String;
end record;
```

2.1.4 Tipe Data *Abstract*

Sebuah tipe data *abstract* adalah sekumpulan data khusus dan sekumpulan operasi yang dapat dijalankan pada data tersebut. Secara sintaktis, sebuah tipe data *abstract* adalah sebuah lampiran yang termasuk di dalamnya representasi data dari sebuah tipe data yang khusus dan subprogram – subprogram yang mendukung untuk tipe tersebut. Unit program yang menggunakan tipe data *abstract* dapat mendeklarasikan variabel dari tipe tersebut, meskipun representasi yang sebenarnya tersembunyi dari program tersebut.

Konsep dari sebuah tipe data *abstract* adalah bentuk dari tipe data yang sudah terbangun dalam sistem yang belum mendapatkan pengembangan.

Contoh dari tipe data *abstract* adalah *list*. *List* merupakan tipe data *abstract* yang mendefinisikan sekumpulan data dengan setruktur tertentu. *List* biasanya diimplementasikan dengan sebuah *array*.

2.2 Statement

Dalam ilmu computer, statement dapat dianggap sebagai sebuah elemen terkecil yang berdiri sendiri (dari kode program). Sebuah program akan dibentuk dari rangkaian - rangkaian *statement*. Dalam sebuah *statement* tersebut terdapat komponen – komponen internal (seperti ekspresi).

Beberapa bahasa pemrograman membedakan antara *statement* dan definisi, bahwa *statement* hanya berisi kode program yang dapat di eksekusi dan sebuah definisi adalah pendeklarasian sebuah *identifier*. Selain itu, *statement* juga dapat dibedakan menjadi *simple statement* dan *compound statement*.

Berikut adalah jenis – jenis statement beserta contohnya dalam bahasa pemrograman *imperative* :

1. definisi dan deklarasi

   definition : `type salary : integer`

   declaration : `var A : integer`

2. *simple statement*

   1. *assignment*:

      ```
      A := A + 1
      ```

   2. *call procedure*:

      ```
      CLEARSCREEN()
      ```

   3. *return*:

      ```
      return 5;
      ```

   4. *goto*: `goto 1`

   5. *assertion*: `assert(ptr != NULL);`

3. *compound statement*

   1. *statement block*:

      ```
      begin
      WRITE('Number? ');
      READLN(NUMBER);
      end
      ```

   2. *if-statement*:

      ```
      if A > 3 then
      WRITELN(A)
      else
      WRITELN("NOT YET")
      end
      ```

   3. *switch-statement*:

```
switch (c) { case 'a': alert(); break;
case 'q': quit(); break; }
```

4. *while-loop*:

```
while NOT EOF DO begin READLN end
```

5. *do-loop*:

```
do { computation(&i); } while (i < 10);
```

6. *for-loop*:

```
for A:=1 to 10 do WRITELN(A) end
```

2.3 Rekursif

Rekursif merupakan suatu proses dimana dalam proses tersebut terdapat pemanggilan proses itu sendiri atau didefinisikan dengan dirinya sendiri. Dalam matematika, definisi rekursif sebuah fungsi adalah definisi fungsi yang menggunakan fungsi tersebut.

Dalam pemrograman *imperative*, algoritma atau hubungan *rekursif* memungkinkan perulangan yang tidak akan berhenti atau tak-hingga. Karenanya, jika sebuah masalah dinyatakan dalam hubungan berulang, maka kondisi berhenti (*basis termination*) hubungan berulang tersebut harus jelas.

Untuk memecahkan masalah menggunakan hubungan berulangnya, maka harus ditentukan :

1. Basis (*termination*), yaitu kondisi dimana rekursif tersebut harus berhenti. Kondisi ini harus jelas sehingga tidak terjadi perulangan yang tak-hingga.

2. Definisi *rekursif*. Untuk membuktikan sebuah definisi rekursif benar adalah jika definisi tersebut benar untuk sebuah nilai, misal untuk (n-1), maka definisi tersebut juga harus benar untuk n.

2.4 Lambda Calculus

Dalam ilmu computer, lambda calculus adalah sebuah sistem formal yang didesain untuk menyelidiki definisi fungsi, aplikasi fungsi, dan rekursif. Lambda calculus dapat digunakan untuk mendefinisikan sebuah fungsi yang *computable* secara jelas.

Lambda calculus dapat disebut sebagai bahasa pemrograman universal terkecil. Lambda calculus terdiri dari aturan transformasi tunggal dan skema definisi fungsi tunggal. Lambda calculus bersifat universal dilihat dari kemampuan untuk mengevaluasi dan mengekspresikan setiap fungsi yang *computable* dengan bentuk formalnya.

2.4.1 Deskripsi Informal

Dalam Lambda calculus, setiap ekspresi merupakan sebuah fungsi dengan sebuah argument tunggal. Argument sebuah fungsi dapat dibuat dari perubahan sebuah fungsi dengan argument tunggal, dan nilai dari sebuah fungsi dapat diambil dari hasil computasi fungsi lain dengan argumen tunggal. Sebuah fungsi secara tidak langsung di definisikan oleh sebuah ekspresi lambda dengan mengekspresikan aksi fungsi tersebut pada argumennya. Sebagai contoh adalah fungsi *f* dengan

$$f(x) = x + 2$$

akan di ekspresikan dalam lambda calculus sebagai

$\lambda x. x + 2$.

Untuk nilai x = 3 maka fungsi tersebut menjadi

$(\lambda x. x + 2)\ 3.$

Fungsi aplikasi dalam lambda calculus bersifat *left Associative* :

g x y = (g x) y.

Misalnya terdapat fungsi *h* mengambil sebuah fungsi sebagai argument dan di aplikasikan dengan nilai 3 :

$\lambda\ h.\ \ h\ \ 3.$

Kemudian diaplikasikan pada fungsi sebelumnya :

$(x + 2) \rightarrow (\lambda\ h.\ h\ 3)\ (\lambda\ x.\ x+2).$

Sebuah fungsi dengan dua variabel diekspersikan dalam lambda calculus sebagai sebuah fungsi dengan satu argumen yang menghasilkan sebuah fungsi dengan sebuah argumen yang lain. Sebagai contoh, pada fungsi :

$f(x,y) = x - y \rightarrow \lambda\ x.\ \lambda\ y.\ x - y.$

Ekspresi berikut ini adalah eqivalen :

1. $(\lambda\ x\ y.\ x - y)\ 7\ 2$
2. $(\lambda\ y.\ 7 - y)\ 2$
3. $7 - 2$

Ekspresi – ekspresi tersebut adalah persamaan ekspersi lambda yang secara umum tidak dapat di selesaikan dengan sebuah algoritma.

2.4.2 Definisi Formal

Secara formal, konsep umum dari lambda calculus adalah ekspresi. Sebuah nama juga disebut sebagai variabel, adalah sebuah *identifier* bisa jadi

merupakan huruf – huruf {a, b,c,...z} . Seluruh kesatuan ekspresi lambda dapat didefinisikan sebagai berikut :

1. <expr> ::= <identifier>
2. <expr> ::= (λ <identifier>. <expr>)
3. <expr> ::= (<expr> <expr>)

Dua aturan yang pertama menunjukkan fungsi, dan yang ketiga mendeskripsikan aplikasi dari sebuah fungsi pada sebuah argumen. Biasanya *parenthesis* untuk ekspresi lambda (aturan 2) dan fungsi aplikasi (aturan 3) diabaikan jika tidak ada ambiguitas dalam asumsi bahwa (1) fungsi aplikasi adalah *left associative*, dan (2) sebuah lambda bergantung pada masukan ekspresi yang mengikutinya.

Ekspresi lambda seperti :

λ x. (x y)

tidak mendefinisikan fungsi karena proses pada variabel y adalah bebas, yaitu tidak bergantung pada λ dalam ekspresinya. Ikatan yang terjadi pada variabel (dengan induksi pada struktur lambda) didefinisikan dengan aturan sebagai berikut :

1. Misalkan diketahui ekspresi *V*, dimana *V* adalah variabel, maka variabel tersebut adalah proses bebas.
2. Pada ekspresi dari bentuk λ *V*. *E*, maka proses bebas adalah proses bebas pada fungsi *E*, kecuali dari *V* tersebut. Dalam hal ini proses dari *V* pada fungsi *E* disebut ikatan oleh λ sebelum *V*.
3. Pada ekspresi dengan bentuk (*E E'*), maka proses bebas adalah proses bebas yang terjadi pada fungsi *E* dan fungsi *E'*.

Semua ekspresi lambda merupakan hubungan persamaan (di notasikan ==) yang di definisikan sebagai dua ekspresi yang merupakan fungsi yang sama. Relasi persamaan tersebut di definisikan oleh:

1. Aturan α-conversion dimaksudkan untuk memberikan pernyataan bahwa nama dari variabel terikat tidak penting sebagai contoh bahwa $\lambda x.x$ dan $\lambda y.y$ adalah fungsi yang sama. Namun demikian, aturan tersebut tidak semudah yang terlihat. Ada banyak batasan ketika sebuah variabel terikat dapat diganti dengan yang lain.

   Aturan α-conversion menyatakan bahwa jika V dan W adalah variabel, E adalah ekspresi lambda, dan

   $$E[V := W]$$

   Berarti pernyataan E dengan setiap proses bebas dari V diganti dengan W, sehingga

   $$\lambda V.\ E \quad == \quad \lambda W.\ E[V := W]$$

2. Aturan β-reduction adalah menyatakan mengenai fungsi aplikasi. Aturan ini menyatakan bahwa pada pernyataan

   $$((\lambda V.\ E)\ E') \quad == \quad E[V := E']$$

   jika semua proses bebas pada E', maka proses tetap bebas pada $E[V := E']$. Relasi atau hubungan (==) di definisikan sebagai hubungan persamaan terkecil yang memenuhi dua aturan ini. Sebuah ekkspresi lambda yang tidak mengikuti aturan β-reduction, yaitu yang tidak mempunyai subekspresi dari bentuk $((\lambda V.\ E)\ E')$ disebut *normal form*.

3. Aturan yang ketiga adalah aturan η-conversion mengubah antara $\lambda x.\ f\ x$ dan f pada saat x terlihat tidak bebas pada f. Dalam hal ini η-conversion, menganggap bahwa dua fungsi dikatakan sama jika dan hanya jika keduanya memberikan hasil yang sama pada seluruh argument.

BAB III

KONSEP PEMROGRAMAN FUNGSIONAL

3.1 Definisi Bahasa Pemrograman Fungsional

Bahasa permrograman fungsional merupakan sebuah paradigma pemrograman yang yang menganggap perhitungan sebagai evaluasi fungsi matematika. Bahasa pemrograman fungsional dalam hal ini, membahas mengenai ekspresi. Dengan kata lain, bahasa pemrograman fungsional merupakan pemrograman yang berorientasi ekspresi, karena dalam pemrograman fungsional semua dijadikan sebagai ekspresi.

Fungsi dianggap sebagai objek dalam bahasa pemrograman fungsional. Sebuah pemrograman fungsional murni tersusun dengan mendefinisikan sebuah ekspresi yang mengambil inti dari sebuah program.

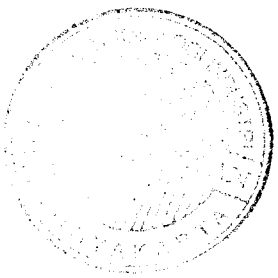
3.2 Tipe : Value dan Operasi

Sebuah tipe terdiri dari sekumpulan elemen yang disebut *value* atau nilai bersama dengan sekumpulan fungsi yang disebut dengan operasi. Nilai terstruktur seperti *list* dapat digunakan secara bebas dalam pemrograman fungsional. Nilai terstruktur sangat penting dalam pemrograman fungsional untuk menyesuaikan sebuah nilai pada sebuah aplikasi khusus.

Nilai dalam bahasa pemrograman fungsional terkadang mnggunakan nilai dasar yang mempunyai dukungan secara langsung dengan bahasa mesin. Integer

adalah nilai dasar yang mempunyai dukungan langsung dengan mesin. tetapi string merupakan nilai dasar yang tidak dibangun dalam bahasa mesin :

1. Tipe Dasar
    1. Sebuah tipe dikatakan tipe dasar apabila pada tipe tersebut tidak didefinisikan dari tipe yang lain.
    2. Nilai dasar tidak mempunyai struktur internal, sehingga operasi yang dapat dikenakan pada seluruh tipe dasar hanyalah operasi perbandingan persamaan.
2. Produk Tipe
    1. Produk A*B dari dua tipe A dan B terdiri dari pair yang ditulis sebagai (a,b), di mana a adalah nilai dari tipe A dan b adalah nilai dari tipe B.
    2. Sebuah pair dibentuk dari a dan b dengan menuliskan (a,b). Fungsi yang digunakan dalam pair disebut dengan fungsi proyeksi, yaitu untuk mengekstrak elemen pertama dan kedua dari pair tersebut.
3. List Element
    1. Sebuah list adalah sekumpulan elemen dengan panjang tertentu. Tipe sebuah list menentukan tipe seluruh elemen dalam list tersebut.sebagai contoh, list dengan tipe integer, maka setiap elemen list tersebut bertipe integer. Elemen list ditulis diantara tanda kolom [dan], serta setiap elemen dipisahkan dengan koma(,).



2. Operasi/manipulasi pada list berkaitan dengan pembentukan dan pemeriksaan list dan elemen – elemen list. Berikut adalah operasi pada list dalam bahasa ML:

Null(x)	bernilai benar jika list kosong dan bernilai salah jika sebaliknya.
hd(x)	mengembalikan elemen pertama dari list
tl(x)	mengembalikan seluruh elemen list kecuali elemen pertama.
a::x	membentuk sebuah list dengan elemen pertama a dan elemen berikutnya x.

3.3 Deklarasi Fungsi

Sebuah ekspresi dibentuk dengan mengaplikasikan sebuah fungsi atau operasi pada subekspresi. Fungsi yang telah dideklarasikan dapat digunakan sebagai sebuah operator dengan beberapa ekspresi.

Sebuah fungsi dalam bahasa pemrograman bersama dengan sebuah algoritma menghitung nilai dari fungsi tersebut pada setiap elemen dari domainnya. Sebuah deklarasi fungsi mempunyai tiga bagian :

1) Nama fungsi yang dideklarasikan.
2) Parameter – parameter fungsi.
3) Aturan penghitungan hasil dari parameter – parameter tersebut.

Sintak dasar pendeklarasian fungsi adalah :

```
fun [name] [formal-parameter] = [body];
```

Keyword **fun** menandai permulaan pendeklarasian fungsi. *[name]* adalah nama fungsi, *[formal-parameter]* adalah nama parameter, dan *[body]* adalah ekspresi yang dievaluasi. Penggunaan tanda kurung ( ) pada parameter formal tersebut adalah opsional. Sebagai contoh :

```
fun successor n = n + 1;
```

dapat ditulis dengan

```
fun successor (n) = n + 1;
```

Penggunaan fungsi dalam sebuah ekspresi disebut aplikasi dari fungsi. Notasi prefiks merupakan aturan untuk aplikasi fungsi yang dideklarasikan :

```
[name] [actual-parameter]
```

[name] menunjukkan nama fungsi, dan *[actual-parameter]* adalah opersi yang berhubungan dengan nama parameter dalam pendeklarasian fungsi.sebagai contoh

```
successor (2+3)
```

merupakan aplikasi fungsi *successor* pada parameter actual (2+3).

Sebuah fungsi *f* dikatakan rekursif apabila didalam fungsi tersebut terdapat aplikasi dari *f*. Dengan kata lain, fungsi *f* adalah rekursif apabila fungsi *f* tersebut dapat mengaktifasi dirinya sendiri, baik secara langsung maupun melalui fungsi yang lain. Sebagai contoh adalah pada bilangan *fibbonaci* yang direpresentasikan sebagai berikut:

```
fun fib(n) =
      If n = 0 orelse n = 1 then 1 else fib(n-1) + fib(n-2)
```

Fungsi tersebut rekursif karena didalam fungsi tersebut terdapat pemanggilan fungsi itu sendiri, yaitu pada aplikasi *fib(n-1)* dan *fib(n-2)*.

3.4 Pendekatan Evaluasi Expresi

Aturan untuk evaluasi ekspresi dalam hal ini didasarkan pada struktur masing – masing ekspresi. Sebagai contoh, sebuah pendekatan untuk evaluasi pada ekspresi :

$E_1 + E_2$

bahwa pada ekspresi tersebut mengevaluasi subekspresi E_1 dan E_2 serta menambahkan nilai dari keduanya. Evaluasi pada setiap subekspresi berbeda – beda sesuai dengan bentuk masing – masing ekspresi.

Aturan evaluasi ekspresi menekankan pada efisiensi. Evaluasi ekspresi menunjukkan apakah hasil dari suatu implementasi bahasa harus menghitung, dan bukan bagaimana hasil tersebut harus diproses.

1. Innermost Evaluation

   Dalam aturan innermost evaluation, sebuah aplikasi fungsi

   [name] [actual-parameter]

   Diproses sebagai berikut :

   1) Mengevaluasi ekspresi yang direpersentasikan oleh *[actual-parameter]*.
   2) Mengganti hasilnya pada parameter formal dalam fungsi.
   3) Mengevaluasi fungsi secara keseluruhan.
   4) Mengembalikan nilai akhir sebagai jawaban.

   Sebagai contoh pada fungsi berikut :

   ```
   fun successor n = n + 1;
   ```

   Innermost evaluation dari aplikasi

   ```
   successor (2+3)
   ```

adalah sebagai berikut :

1) Mengevaluasi argument (2+3).
2) Mengganti parameter formal dengan nilai 5 pada fungsi n+1.
3) Mengevaluasi hasil ekspresi 5+1.
4) Mengembalikan jawaban dengan 6.

Setiap evaluasi pada suatu fungsi disebut aktivasi fungsi. Apabila

```
successor (2+3)
```

Ditulis

```
successor (plus(2,3))
```

Maka, perhitungannya dapat dideskribsikan sebagai berikut :

1) Aktifkan *plus* untuk mengevaluasi *plus(2,3)*.
2) Mengembalikan nilai 5 sebagai hasil dari operasi *plus*.
3) Mengaktifkan successor 5.
4) Mengembalikan nilai 6 sebagai hasil operasi *successor 5*.

2. Selective Evaluation

Kemampuan untuk mengevaluasi dengan memilih beberapa bagian dan mengabaikan bagian yang lain direpresentasikan dengan bentuk berikut :

if [kondisi] then [ekspresi$_1$] else [ekspresi$_2$]

[kondisi] merupakan ekspresi yang mengevaluasi dengan dua nilai kemungkinan yaitu ***true*** dan ***false***. Jika *[kondisi]* mengevaluasi dengan nilai ***true***, maka nilai dari *[ekspresi$_1$]* menjadi nilai hasil dari fungsi tersebut. Sebaliknya jika *[kondisi]* mengevaluasi dengan nilai false, maka nilai dari *[ekspresi$_2$]* menjadi nilai dari fungsi tersebut.

3. Evaluation Of Recursive Functions

Pada aturan *evaluation of recursive functions*, parameter *actual* dievaluasi dan disubtitusikan ke dalam fungsi itu sendiri.

4. Outermost Evaluation From Left To Right

Pada aturan *outermost-evaluation*, sebuah fungsi aplikasi diproses sebagai berikut :

1) Menggantikan parameter actual dengan parameter formal dalam fungsi tersebut.
2) Mengevaluasi isi fungsi.
3) Mengembalikan nilai akhir sebagai hasil dari proses.

3.5 Lexical Scope

Lexical scope yang dimaksudkan dalam hal ini adalah bahwa pengubahan variabel tidak mempengaruhi nilai dari suatu ekspresi. Pengubahan parameter formal dari x menjadi n dalam deklarasi fungsi successor

```
Fun successor(x) = x + 1;
Fun successor(n) = n + 1;
```

Seharusnya tidak mempengaruhi program. Nilai dari ekspresi successor(5) adalah 6 dengan deklarasi yang lain.

3.6 Lambda Calculus

Dalam setiap bahasa pemrograman fungsional, Lambda Calculus merupakan bagian yang tak terpisahkan. Penggunaan Lambda Calculus dalam pemrograman fungsional sama seperti dalam Lambda Calculus murni, hanya saja pemakaian notasi λ diganti sesuai dengan bahasa pemrograman yang digunakan.

Dalam Standard ML, tidak dikenal ekspresi Lambda sebagaimana dalam Scheme. Seperti halnya dalam Lambda Calculus, fungsi aplikasi juga bersifat *left associative*, sehingga kita dapat menggunakan fungsi fn untuk merepresentasikan ekspresi Lambda. Dalam Standard ML, ekspresi :

```
λ x.t[x]
```

dapat ditulis dengan

```
fn x => t[x];
```

sehingga untuk ekspresi

```
λ x.x+1,
```

akan dituliskan sebagai

```
fn x => x +1;
```

Dalam Lambda Calculus, sebuah ekspresi lambda dapat diaplikasikan pada sebuah fungsi. Sebagai contoh, sebuah ekspresi lambda yang diaplikasikan pada sebuah fungsi penjumlahan sebagai berikut :

```
(λ x.x+1) (3+5);
```

Atau dalam Standard ML kita representasikan dengan,

```
(fn x => x+1) (3+5);
```

Dalam penggunaan fungsi fn, tidak dapat menerima hasil masukan dengan sebuah variable bebas yang tidak dikenal dalam system, atau yang tidak didefinisikan nilainya terlebih dahulu. Pada contoh berikut, misalnya :

```
(fn x => x + 1) (x +5);
```

akan memberikan pesan kesalahan, karena sebagaimana pada umumnya bahwa dalam sebuah fungsi tidak akan mengeksekusi dalam operasi penjumlahan selain tipe numeric. Untuk dapat mengeksekusi ekpresi tersebut, dapat dituliskan sebagai berikut :

```
val x = 5;
> val x = 5 : int
(fn x => x + 1) (x +5);
> val it = 11 : int
```

Lambda calculus juga mendukung konsep curry, yaitu mampu mengubah sebuah ekspresi lambda dengan multiple argument menjadi sebuah ekspresi dengan single argument. Misalnya contoh berikut :

```
val p = fn (x,y) => x + y;
>val p = fn : int * int -> int
```

Dapat diubah menjadi bentuk curry sebagai berikut :

```
val p = fn x => fn y => x + y;
>val p = fn : int -> int -> int
```

Contoh yang lain, misalnya pada fungsi fst dan snd berikut :

```
val fst = fn (x,y) => x
val snd = fn (x,y) => y;
```

Dapat diubah menjadi bentuk curry dengan mengubah seperti pada contoh sebelumnya, sehingga dalam setiap ekspresi lambda hanya mempunyai satu argument.

Pada fungsi fst tersebut, juga dapat diketahui bahwa lambda calculus bersifat polimorfisme. Yaitu bahwa fungsi tersebut dapat digunakan untuk beberapa tipe data. Contoh yang paling mudah yaitu pada fungsi identitas, dimana fungsi tersebut mengembalikan keluaran sesuai dengan data yang dimasukkan.

```
val I = fn x => x;
> val 'a I = fn : 'a -> 'a
```

Apabila fungsi tersebut diberikan masukan berupa huruf a, fungsi tersebut akan mengembalikan masukan tersebut sebagai hasil akhir. Demikian juga apabila

fungsi tersebut diberikan masukan dengan tipe pair, list ataupun tipe – tipe yang lain.

```
I "a";
> val it = "a" : string
I (1,2);
> val it = (1, 2) : int * int
I [1,2,3,4];
>val it = [1, 2, 3, 4] : int list
```

BAB IV

KONSEP PEMROGRAMAN HASKELL

4.1 Tipe : Value dan Operasi

Dalam setiap bahasa perograman, baik itu imperative, maupun fungsional selalu memiliki beberapa tipe. Sebagaimana dalam bahasa pemrograman fungsional yang lainnya, seluruh perhitungan dalam Haskell diselesaikan dengan evaluasi ekspresi untuk menghasilkan nilai sebagai jawaban. Hal ini, sesuai dengan konsep utama bahasa pemrograman fungsional, yaitu pemrograman yang berorientasi ekspresi.

Setiap nilai dalam Haskell, mempunyai tipe – tipe yang menghubungkan dengan operasi yang dapat dilakukan terhadap nilai tersebut. Nilai dalam Haskell, seperti halnya dalam pemrograman fungsional yang lainnya, dapat terdiri dari nilai dasar, maupun nilai terstruktur :

1. Tipe Dasar.
    1) Tipe – tipe dasar dalam Haskell antara lain adalah tipe numeric(integer,double dan lain – lain), char, Boolean, dan tipe – tipe yang tidak tebentuk dari tipe yang lain.
    2) Operasi – operasi yang dapat dilakukan terhadap tipe dasar yaitu operasi – operasi persamaan. Diantaranya yaitu :

        Id mengembalikan nilai dari nilai masukan.

        == berinilai benar apabila nilai yang pertama sama dengan nilai yang kedua.

2. Produk Tipe.

   1) Tipe – tipe produk dalam Haskell yaitu dapat berupa pair, triple, ataupun tuple. Sebuah tipe produk ditulis dengan tanda kurung, dan dipisahkan dengan koma pada setiap elemennya.

   2) Operasi – operasi yang dapat dikenakan dalam tipe ini adalah :

      fst(x,y) mengembalikan nilai pertama dari tuple.

      snd(x,y) mengembalikan nilai kedua dari tuple.

3. List Element.

   1) Sebuah list merupakan tipe terstruktur yang dibentuk dari beberapa elemen dengan tipe yang sama. Dalam Haskell, List dapat digunakan secara bebas sebagaimana penggunaan tipe dasar.

   2) Manipulasi yang dapat dilakukan pada list berkaitan dengan pembentukan dan pemeriksaan list dan elemen – elemen didalamnya. Berikut adalah operasi – operasi dasar pada list dalam Haskell:

      head mengembalikan elemen pertama dari list.

      tail membuang elemen pertama dari list.

      Length mengembalikan panjang dari list.

      ++ menggabungkan 2 buah list menjadi satu.

      : digunakan untuk pembentukan list dari elemen – elemennya.

4.2 Deklarasi Fungsi

Sebagai bahasa pemrograman fungsianal, fungsi dalam Haskell mempunyai peranan yang sangat penting. Dalam Haskell, seperti halnya dalam pemrograman fungsional yang lain, semuanya dianggap sebagai fungsi. Penulisan sebuah program dalam Haskell merupakan penulisan sebuah fungsi. Sebuah file dapat digunakan untuk mendeklarasikan beberapa fungsi.

Pendeklarasian sebuah fungsi, sebagaimana dalam pemrograman fungsional yang lain, terdapat tiga bagian :

1) Nama fungsi yang dideklarasikan,
2) Parameter – parameter fungsi,
3) Aturan penghitungan hasil fungsi tersebut.

Untuk pendeklarasian fungsi dalam Haskell, ditulis dalam sintaks berikut :

```
[nama]  [parameter]  = [body]
```

Keyword *[nama]* adalah nama fungsi, *[parameter]* adalah parameter – parameter yang digunakan, dan *[body]* menunjukkan ekspresi yang dievaluasi dalam fungsi tersebut. Sebagai contoh :

```
square x = x*x
```

Kadang – kadang sebelum deklarasi fungsi ditambahkan definisi tipe dari argument dan hasil evaluasi fungsi, sehingga sintaksnya menjadi seperti berikut :

```
[nama] :: [tipe parameter] -> [tipe hasil]
[nama]  [parameter]  = [body]
```

[nama] menunjukkan nama fungsi yang akan dideklarasikan, *[type parameter]* menunjukkan tipe dari argument atau nilai yang akan dievaluasi, dan *[type hasil]*

menunjukkan tipe hasil evaluasi fungsi. Sebagai contoh, fungsi diatas dapat ditulis menjadi :

```
square :: integer -> integer
square x = x*x
```

4.3 Pendekatan Evaluasi Ekspresi

Aturan evaluasi ekspresi dalam hal ini adalah hampir sama dalam setiap bahasa pemrograman, terutama pada setiap bahasa pemrograman fungsional. Evaluasi ekspresi dilakukan sesuai dengan struktur ekspresi. Proses evaluasi hanya dilakukan apabila proses tersebut diperlukan dan tidak dilakukan apabila hasil akhir suatu fungsi telah didapatkan.

1. Innermost Evaluation

   Innermost Evaluation dilakukan ketika sebuah fungsi diberikan nilai dengan fungsi lain. Evaluasi dilakukan terhadap fungsi yang kedua, kemudian setelah itu hasilnya diproses sebagai parameter dari fungsi pertama. Misalnya pada fungsi berikut:

   ```
   f :: int -> int
   f x = x + 3
   ```

   apabila diberikan nilai (2+3) menjadi

   ```
   f (2+3).
   ```

   Pada ekspresi tersebut, dilakukan Innermost Evaluation dengan urutan proses sebagai berikut :

   1) Mengevaluasi argument (2+3).
   2) Menggantikan parameter formal dengan nilai 5.
   3) Mengevaluasi hasil ekspresi 5+3.

4) Mengembalikan nilai 8 sebagai hasil akhir.

2. Selective Evaluation.

Evaluasi ini dilakukan terhadap ekspresi – ekspresi bersyarat. Pada evaluasi ini, suatu bagian tidak dievaluasi apabila kondisi yang sudah yang diinginkan tercapai. Bagian lain hanya akan dievaluasi apabila kondisi yang diinginkan belum tercapai. Misalnya pada fungsi berikut :

```
f x =
        case x of { 0 -> 1 ; 1 -> 5 ; 2 -> 2 ; _ -> 0 }
```

Apabila fungsi diberikan masukan dengan nilai nol (0), fungsi hanya akan melakukan evaluasi sampai pada nilai x yang pertama, dan mengabaikan yang lain.

3. Evaluation Of Recursive Functions

Pada Haskell, evaluasi ini dilakukan ketika sebuah fungsi memiliki parameter actual dai fungsi itu sendiri. Dengan kata lain, evaluasi ini dilakukan apabila fungsi tersebut harus mengevaluasi dirinya sendiri. Sebagai contoh :

```
factorial 1 = 1
factorial n = n * factorial (n-1)
```

fungsi factorial, apabila diberikan nilai masukan lebih dari 1, maka fungsi tersebut akan melakukan evaluasi terhadap fungsi itu sendiri sampai nilai parameter n bernilai 1.

4.4 Lexical Scope

Sebagaimana yang telah disebutkan sebelumnya, bahwa sebuah bahasa pemrograman fungsional bersifat *lexical scope*, yaitu bahwa pengubahan suatu variable tidak mempengaruhi nilai dari suatu ekspresi. Dalam Haskell pengubahan suatu variable dalam sebuah fungsi tidak mempengaruhi nilai dari hasil evaluasi ekspresi tersebut dengan pemberian nilai masukan yang sama. Apabila sebuah fungsi :

```
Square x = x*x
```

Diganti variable x tersebut menjadi n, sehingga menjadi:

```
Square n = n*n
```

maka proses pada fungsi tersebut tidak terpengaruh. Apabila diberikan sebuah nilai yang sama pada keduanya, maka keduanya akan didapatkan hasil akhir yang sama.

4.5 Lambda Calculus

Sebagai bahasa pemrograman fungsional, Haskell mendukung konsep Lambda Calculus. Dalam Haskell digunakan notasi (\) merepresentasikan ekspresi lambda. Untuk merepresentasikan ekspresi lambda calculus berikut :

```
λ x.t[x]
```

dalam Haskell, dapat ditulis dengan :

```
\x -> t[x]
```

Untuk merepresentasikan fungsi perkalian berikut :

```
λ x.x*x
```

dapat ditulis dengan

```
\x -> x*x
```

Secara teori, lambda calculus dapat diterapkan ke dalam sebuah fungsi, misalnya fungsi diatas diterapkan pada fungsi penjumlahan seperti berikut :

```
(λ x.x*x)(x+5)
```

Maka hasilnya sebagai berikut:

$(x^2+10x+25)$

Akan tetapi, dalam bahasa pemrograman, baik itu bahasa pemrograman imperative maupun pemrograman fungsional, penerapan tersebut tidak dapat dilakukan karena dalam sebuah bahasa pemrograman fungsi matematika hanya dapat diterapkan pada tipe numeric.

Dalam Haskell, pada fungsi tersebut x dianggap sebagai variable bebas. Untuk dapat mengeksekusinya diperlukan pendeklarasian nilai dari variable tersebut sebelumnya. Jadi, dengan kata lain sebuah variable dari suatu fungsi hanya dapat diproses dalam ekspresi lambda apabila telah dideklarasikan sebelumnya.

Untuk penggunaan multiple argument, Haskell mendukung konsep curry. Sama seperti pada Standard ML, dalam Haskell penggunaan multiple argument dapat diubah menjadi single argument. Contoh berikut :

```
\x y -> (y,x)
```

dapat diubah menjadi bentuk Curry seperti :

```
\x -> \y -> (y,x)
```

Untuk sifat polimorphisme, dapat dilihat pada penggunaan ekspresi lambda diatas. Pada contoh tersebut, fungsi akan mengembalikan hasil evaluasi sesuai dengan data dan tipe yang diberikan. Misalnya :

```
(\x -> \y -> (y,x)) 2 3
```

Akan menampilkan hasil (3,2). Begitu juga apabila diberikan nilai a dan b, akan mengembalikan hasil ("b", "a"). Contoh lain dari polimorphisme, yaitu pada fungsi identitas, dimana fungsi tersebut mengembalikan nilai hasil dari nilai masukan.

BAB V

ANALISIS BAHASA PEMROGRAMAN HASKELL

Setiap bahasa pemrograman selalu memiliki kelebihan dan kekurangan terhadap bahasa pemrograman yang lain. Untuk mengetahui kelebihan dan kekurangan bahasa pemrograman Haskell sebagai bahasa pemrograman fungsional perlu diadakan pengujian Haskell dalam menyelesaikan suatu permasalahan. Permasalahan yang dibahas dalam analisis ini meliputi :

1. Tipe Bentukan.
2. Struktur Data
3. Binding.
4. Input/Output.
5. Modularitas.

5.1 Tipe Bentukan

Pada pembahasan ini dilakukan pengujian kemampuan Haskell dalam membuat sebuah tipe bentukan. Tipe bentukan merupakan tipe yang dideklarasikan oleh pemrogram yang nilai dari tipe itu sesuai dengan keinginan pemrogram itu sendiri.

Untuk mengetahui kemampuan Haskell dalam membuat sebuah tipe data, dapat dilihat pada contoh fungsi first yang mempunyai masukan berupa triple dan mengembalikan nilai pertama dari triple tersebut :

```
data Triple = Tr Float Float Float
firsts :: Triple -> (Float)
firsts (Tr a b c) =  a
```

Gambar 5.1 Source Code Tipe Bentukan dengan Haskell

Pada contoh diatas, dapat dilihat bahwa fungsi firsts mempunyai tipe Triple. Tipe Triple adalah tipe data bentukan yang telah didefinisikan sebelumnya, dengan constructor yang dinamakan Tr, dan memiliki tiga parameter. Pada contoh tersebut, tipe data bentukan didefinisikan dengan kata kunci data sebagai pengenalnya.

Pada pemrograman *imperative*, khususnya dalam bahasa C++, kasus tersebut dapat diselesaikan dengan tipe data bentukan *struct*, seperti berikut :

```
struct Triple {
   char A[8];
   char B[8];
   char C[8];
};
      Triple Tr;
cin>>Tr.A>>Tr.B>>Tr.C;
cout<<Tr.A<<endl;
```

Gambar 5.2 Source Code Tipe Bentukan dengan C++

Untuk fungsi first, dapat digantikan dengan keluaran berupa nilai pertama. Program akan menerima tiga input, Tr.A, Tr.B dan Tr.C.

5.2 Struktur Data

Struktur data adalah cara pengaturan, pengorganisasian dan penyimpanan data dalam media penyimpanan computer sehingga data tersebut dapat digunakan secara efisien. Pada pembahasan ini mengulas mengenai struktur data yang terdapat pada Haskell yang meliputi tipe data :

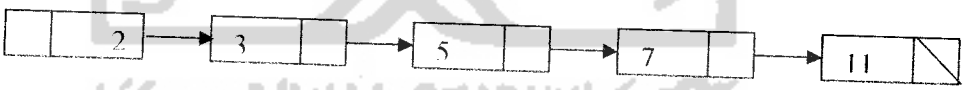
1. List

List merupakan tipe terstruktur yang merupakan tipe data asli dalam pemrograman fungsional. Dalam Haskell, sebuah List menyimpan data dalam sebuah memory dalam bentuk elemen – elemen yang berindex, dan tiap – tiap elemen terhubung dengan sebuah link. Sebagaimana pada pemrograman lain, list dalam Haskell terdapat kepala dan ekor. Kepala dari sebuah List adalah elemen pertama dari List tersebut. Ekor adalah elemen dari List yang bukan merupakan kepala, yaitu elemen kedua dan seterusnya. Berikut adalah contoh List yang bertipe numeric :

```
Prelude>[2,3,5,7,11]
[2,3,5,7,11]
```

Gambar 5.3 Contoh List Pada Haskell

Pada contoh List tersebut, nilai pertama (2) merupakan kepala dari List. Sedangkan nilai – nilai berikutnya merupakan ekor dari List. Setiap node dari List digunakan untuk menyimpan data, dan setiap node terhubung satu dengan yang lain. Struktur dari contoh List tersebut dapat digambarkan seperti berikut :



Gambar 5.4 Struktur List

Pada bahasa C++, list merupakan tipe data bentukan yang dapat direpresentasikan sebagai berikut :

```
typedef struct TNode
      {
  int data;
  TNode *next;
      };
```

Gambar 5.5 Source Code Untuk mereperesentasikan List Pada C++

Pada deklarasi tersebut, terdapat *field* data yang merupakan tipe terdefinisi yang menyimpan informasi sebuah elemen list, dan next yang merupakan address dari elemen berikutnya.

Untuk membentuk sebuah list, pada bahasa C++ dapat dilakukan dengan fungsi program berikut :

```
TNode *baru;
baru = new TNode;
baru->data = databaru;
baru->next = NULL;
```

Gambar 5.6 Source Code fungsi pembentukan List Pada C++

Untuk memanipulasi data – data yang ada pada sebuah list, Haskell telah menyediakan beberapa fungsi operasi. Misalnya ingin dilakukan penambahan elemen pada awal List, dapat dilakukan seperti contoh berikut :

```
Prelude>1:[2,3,5,7,11]
[1,2,3,5,7,11]
```

Gambar 5.7 Penambahan Elemen Pada Awal List Pada Haskell.

Sedangkan pada bahasa C++, untuk memanipulasi list digunakan bantuan pointer penunjuk node pertama dan inisialisasi seperti berikut :

```
TNode *head;
void init ()
{
  head = NULL;
}
```

Gambar 5.8 Source Code Inisialisasi Node Pertama List dengan C++

Untuk mengetahui kosong tidaknya suatu list, digunakan fungsi berikut :

```
int isEmpty(){
if(head == NULL) return 1;
else return 0;
}
```

Gambar 5.9 Source Code Pengecekan Elemen Pertama List Dengan C++

Untuk penambahan elemen awal list, dilakukan dengan fungsi berikut :

```
void insertDepan(int databaru){
TNode *baru;
baru = new TNode;
baru->data = databaru;
baru->next = NULL;
if(isEmpty()==1){
head=baru;
head->next = NULL;
}
else {
baru->next = head;
head = baru;
}
cout<<"Data masuk\n";
}
```

Gambar 5.10 Source Code Fungsi Penambahan Elemen Awal List Dengan C++

Untuk menambah elemen pada akhir List, pada Haskell dapat dilakukan sebagai berikut :

```
Prelude>reverse (9:( reverse [2,3,7,11] ))
[2,3,7,11,9]
```

Gambar 5.11 Penambahan Elemen Akhir List Dengan Haskell

Fungsi reverse, sebenarnya digunakan untuk membalikkan sebuah List. Sedangkan pada Bahasa C++, penambahan belakang List dilakukan dengan fungsi seperti berikut :

```
void insertBelakang (int databaru){
TNode *baru,*bantu;
baru = new TNode;
baru->data = databaru;
baru->next = NULL;
if(isEmpty()==1){
head=baru;
head->next = NULL;
}
else {
bantu=head;
while(bantu->next!=NULL){
bantu=bantu->next;
}
bantu->next = baru;
}
cout<<"Data masuk\n";
}
```

Gambar 5.12 Source Code Fungsi Penambahan Akhir List Dengan C++

Untuk mengurutkan sebuah List, Haskell menyediakan fungsi *List.sortBy*. Misalnya, sebuah List ingin diurutkan datanya dari yang terkecil, dapat dilakukan seperti berikut :

```
Prelude>List.sortBy compare [1,2,8,5,3,7]
[1,2,3,5,7,8]
```

Gambar 5.13 Fungsi Untuk Pengurutan Elemen List Pada Haskell

Apabila diinginkan pengurutan yang sebaliknya, dapat ditambahkan fungsi *flip*, yang merupakan fungsi untuk membalikan *order*. Berikut adalah contoh pengurutan List dari nilai terbesar ke nilai terkecil :

```
Prelude>List.sortBy (flip compare) [1,2,8,5,3,7]
[8,7,5,3,2,1]
```

Gambar 5.14 Fungsi Untuk Pengurutan Elemen List dari yang Terbesar.

Apabila diinginkan sebuah penyisipan sebuah elemen pada sebuah List, dapat dideklarasikan fungsi berikut :

```
F p e [] = [p]
F p e (x:xs) = (take (e-1) (x:xs)) ++ [p]
               ++ (drop (e-1) (x:xs))
```

Gambar 5.15 Fungsi Untuk Penyisipan List Pada Haskell.

Dengan p adalah nilai yang akan disisipkan, e adalah index yang diinginkan untuk penyisipan. Jadi, apabila diinginkan penyisipan data dengan nilai 11 apada index ke-3 dari List [1,2,3,4,5], dapat dilakukan seperti berikut :

```
main>F 11 3 [1,2,3,4,5]
[1,2,11,3,4,5]
```

Gambar 5.16 Penyisipan Elemen Pada List Pada Haskell.

Fungsi tersebut, sebenarnya dibentuk berdasarkan fungsi *concatenation*, yaitu fungsi untuk menggabungkan sebuah List dengan List yang lain.

2. Binary Tree

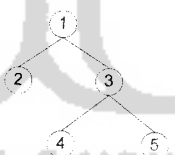
Sebuah Binary Tree adalah sebuah struktur yang mempunyai satu node akar, dan setiap node dapat berupa daun ataupun cabang. Apabila node tersebut berupa daun, maka node tersebut memiliki sebuah nilai, dan apabila node tersebut sebuah cabang, maka node tersebut memiliki dua node, yang dapat berupa daun, ataupun berupa cabang dan satu node sebagai pusat/root.

Tipe data Binary Tree, dalam Haskell dapat direpresentasikan sebagai berikut :

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

Gambar 5.17 Representasi Binary Tree Pada Haskell

Misalnya ada sebuah data biner yang memiliki node pusat dengan nilai 1, dan mempunyai cabang kiri dan kanan, yang masing – masing mempunyai nilai 2 dan 3, kemudian pada cabang kanan mempunyai cabang baru dengan nilai kiri 4 dan kanan 5, seperti gambar berikut :



Gambar 5.18 Struktur Binary Tree

Data tersebut, sesuai dengan tipe data yang telah didefinisikan dapat dituliskan sebagai berikut :

```
Branch (Branch Empty 2 Empty) 1 (Branch (Branch Empty 4
Empty) 3 (Branch Empty 5 Empty))
```

Gambar 5.19 Contoh Binary Tree Pada Haskell

Untuk representasi Binary Tree pada bahasa C++, dapat digunakan *struct* berikut :

```
typedef struct Tree{
int data;
Tree *left;
Tree *right;
}

Tree *pohon
```

Gambar 5.20 Representasi Binary Tree Pada C++

Untuk mengitung jumlah daun yang ada pada sebuah data dengan tipe binary Tree pada Haskell, dapat dibentuk fungsi berikut :

```
count_leaf  Empty                    = 0
count_leaf (Branch Empty a Empty) = 1
count_leaf (Branch left a right)  = count_leaf left
                                    + count_leaf right
```

Gambar 5.21 Fungsi Menghitung Jumlah Daun Binary Tree Pada Haskell

Misalnya contoh diatas dideklarasikan sebagai nilai x. Untuk menghitung banyaknya daun dari x dapat digunakan fungsi yang telah dibuat diatas sebagai berikut :

```
main>count_leaf x
3
```

Gambar 5.22 Penggunaan Fungsi count_leaf

Sedangkan fungsi untuk menghitung node yang ada pada binary tree pada bahasa C++, adalah seperti berikut :

```
int count(Tree *root)
{
if (root == NULL) return 0;
return count(root->left) + count(root->right) + 1;
}
```

Gambar 5.23 Source Code Fungsi Menghitung Node Pada Binary Tree

Untuk mengetahui data – data node yang merupakan daun, dapat dibuat fungsi seperti berikut :

```
leaves :: Tree a -> [a]
leaves Empty                    = []
leaves (Branch Empty a Empty)   = [a]
leaves (Branch lhs a rhs) = leaves lhs ++ leaves rhs
```

Gambar 5.24 Fungsi Untuk Mengetahui Daun Pada Binary Tree Dengan Haskell

Fungsi tersebut mengembalikan hasil dalam bentuk List. Sebagai contoh, data x yang telah didefinisikan diatas ingin ditampilkan nilai dari node yang merupakan daun, dapat dilakukan sebagai berikut :

```
main>leaves x
[2,4,5]
```

Untuk mengetahui data yang terletak pada level tertentu, dapat dibuat sebuah fungsi seperti berikut :

```
atlevel :: Tree a -> Int -> [a]
atlevel t level = loop t 1
  where
    loop Empty _            = []
    loop (Branch l a r) n
        | n == level = [a]
        | otherwise  = loop l (n+1) ++ loop r (n+1)
```

Gambar 5.25 Fungsi Untuk Mengetahui Data Level Tertentu Pada Binary Tree.

Fungsi tersebut menerima input sebuah tipe data Binary Tree, dan level yang data pada node – node level tersebut ingin diketahui. Misalnya seperti berikut :

```
main>atlevel x 2
[2,3]
```

x adalah sebuah data dengan tipe Binary Tree yang dideklarasikan pada contoh. Angka 2 menunjukkan level dari binary Tree x.

3. Array

Array adalah variasi dari List. Array merupakan struktur data dimana data dapat disimpan secara berjajar sebanyak N buah. Pada array, terdapat dua komponen utama, yaitu index array dan data yang disimpan dalam setiap index. Misalnya array berikut ini adalah array yang mempunyai nomor index 1 sampai 5 yang tiap – tiap indexnya berisi angka :

```
Prelude Array>listArray (1,5) [1,11,77,5,4]
Array (1,5) [(1,1),(2,11),(3,77),(4,5),(5,4)]
```

Gambar 5.26 Contoh Array Pada Haskell.

Pada contoh tersebut, array dibentuk dengan menetukan jumlah index, dan menentukan data yang diisikan pada masing – masing index. Struktur dari Array dari contoh tersebut dapat digambarkan seperti berikut:

```
Array : [1] [11] [77] [5] [4]
Index :  1    2    3    4   5
```

Pada contoh tersebut, array memiliki 5 buah memori yang setiap memori telah terisi dengan data – data yang berupa angka 1 pada memory pertama, 11 pada index ke-2, dan seterusnya.

Misalnya diinginkan sebuah array dengan index 1 sampai 9, dan diisi dengan 5 bilangan prima pertama, dituliskan seperti berikut :

```
Prelude Array>listArray (1,9) [2,3,5,7,11]
array (1,9) [(1,2),(2,3),(3,5),(4,7),(5,11),(6,***
Exception:(Array.!): undefined array element
```

Gambar 5.27 Contoh Array Yang Tidak Terisi Penuh Pada Haskell.

Pada contoh, muncul pesan error pada index ke-6, yang berarti index ke-6 dan seterusnya tidak terisi. Untuk menambahkan atau meng-update

element dari Array. dapat digunakan fungsi //. Misalnya Arrray diatas telah dideklarasikan sebagai x, kemudian ingin diisi index ke-6 dengan nilai 17, dapat dilakukan seperti berikut :

```
main>x // [(6,17)]
array (1,9) (1,2),(2,3),(3,5),(4,7),(5,11),(6,17),(7,
*** Exception:(Array.!): undefined array element
```

Gambar 5.28 Fungsi Untuk Mengisi Elemen Array Pada Haskell

Untuk melihat data pada index tertentu dapat digunakan fungsi (!). Misalnya :

```
main>x ! 3
5
```

Contoh tersebut, mengambil data pada index ke-3.

Untuk melihat index yang dimiliki sebuah Array.dapat digunakan fungsi *bounds*, atau fungsi *indices*. Untuk fungsi *bounds*, hasilnya akan ditampilkan dalam bentuk pair yang menunjukkan index pertama dan index terakhir. Untuk fungsi *indices*, setiap index akan ditampilkan dalam bentuk sebuah List. Contohnya seperti berikut :

```
main>bounds x
(1,9)
main>indices x
[1,2,3,4,5,6,7,8,9]
```

Pada bahasa C++, Array selalu memulai index pertamanya dengan nilai 0. Misalnya seperti contoh diatas, terdapat sebuah Array dengan 5 elemen index, dapat dituliskan sebagai berikut :

```
int larik[5];
```

Gambar 5.29 Contoh Pendeklarasian Array Pada C++

Sedangkan Untuk membuat sebuah Array yang telah terisi dengan elemen pada setiap *memory*-nya, dapat dideklarasikan sebagai berikut :

```
int larik[5] = { 1, 11, 77, 5, 4 };
```

Gambar 5.30 Contoh Pendeklarasian Array Yang Terisi Pada C++

Untuk manipulasi Array, misalnya mengubah elemen dapat dilakukan secara langsung dengan mendeklarasikan nilai yang akan diganti pada indek yang diinginkan. Contohnya untuk penggantian pada data ketiga misalnya, dapat dilakukan sebagai berikut :

```
larik[2] = 9;
```

Gambar 5.31 Penggantian Elemen Array Pada C++

5.3 Binding

Binding adalah sebuah pendeklarasian sebuah variable dengan nilai tertentu, atau pengikatan sebuah fungsi atau variable terhadap fungsi yang lain. Dengan kata lain, binding adalah pembuatan sebuah deklarasi local pada fungsi tertentu.

Pada pembahasan ini akan diujikan kemampuan Haskell dalam melakukan suatu pengikatan sebuah fungsi atau variable oleh sebuah fungsi. Kemampuan binding pada Haskell dapat dilihat pada penyelesaian contoh kasus penghitungan

akar – akar dari persamaan parabola. Persamaan parabola adalah persamaan kuadarat yang persamaannya sebagai berikut :

$$ax^2 + bx + c = 0$$

sedangkan persamaan untuk mencari akar persamaan tersebut adalah :

$$x = (-b \pm \sqrt{(b^2-4ac)})/2a$$

Pada contoh kasus, fungsi menerima input berupa nilai dari a,b dan c dari persamaan parabola tersebut. Kemudian fungsi akan memberikan hasil akhir berupa akar – akar dari persamaan tersebut. Dari kasus tersebut, dapat dibuat sebuah fungsi sebagai berikut :

```
roots a b c =
      let det = sqrt(b*b - 4*a*c)
          a2 = 2*a
      in ((-b + det) / a2, (-b - det) / a2)
```

Gambar 5.32 Source Code Fungsi Akar Persamaan Parabola Pada Haskell

Fungsi det dan fungsi a2 merupakan deklarasi local yang terikat pada fungsi roots. Misalkan ada sebuah persamaan :

$$x^2 + 4x + 4 = 0$$

Dapat dihitung akar – akarnya dengan :

```
main>roots 1 4 4
(-2.0,-2.0)
```

Apabila terdapat fungsi/ variable dengan nama det atau a2 yang dideklarasikan secara global, maka fungsi tersebut hanya berlaku diluar fungsi roots. Misalnya :

```
det = putStrLn "Hello"
a2 = 50
roots a b c =
      let det = sqrt(b*b - 4*a*c)
          a2 = 2*a
      in ((-b + det) / a2, (-b - det) / a2)
```

Gambar 5.33 Contoh Penggunaan Binding Pada Haskell

Untuk mengetahui bahwa kedua fungsi yang sama tersebut hanya dieksekusi pada masing – masing scope, maka source code tersebut dapat dijalankan. Misalnya :

```
main>det
"Hello"
main>roots 1 4 4
(-2.0,-2.0)
main>a2
50
```

Untuk kasus persamaan parabola tersebut, pada bahasa C++ tidak dapat diselesaikan dengan binding. Hal ini karena binding merupakan kemampuan pemrograman fungsional yang tidak ada pada pemrograman *imperative*. Kasus tersebut dapat diselesaikan seperti berikut :

```
#include <iostream.h>
#include <math.h>

int main() {

      double det;
      double dua;
      double root1;
      double root2;
      struct Triple {
                        double A;
                        double B;
                        double C;
      };

            Triple Tr;

 cin>>Tr.A>>Tr.B>>Tr.C;

      det = sqrt ((Tr.B*Tr.B)-(4*Tr.A*Tr.C));
      dua = 2*Tr.A;
      root1 = ((-Tr.B)+det)/dua;
      root2 = ((-Tr.B)-det)/dua;

 cout<<"("<<root1<<","<<root2<<")"<<endl;
 return 0;
}
```

Gambar 5.34 Source Code Fungsi Akar Persamaan Parabola dengan C++

5.4 Input/Output

Pembahasan ini akan menjelaskan kemampuan Haskell dalam mengatasi masalah input/output. Masalah input/output yang akan dibahas adalah kemampuan yang minimal harus dimiliki Haskell sebagai sebuah bahasa pemrogaman yang meliputi kemampuan untuk :

1. Menampilkan suatu nilai string ke layar.
2. Membaca input string dari *keyboard.*
3. Menuliskan data ke dalam bentuk file.
4. Membaca data dari sebuah file.

Untuk mengetahui kemampuan Haskell dalam keempat operasi tersebut dapat dilihat pada contoh program untuk membaca dan menulis file berikut :

```
module DoFile where

import IO

main = do
 putStrLn "Do you want to [read], [write] a file, or quit?"
 cmd <- getLine
 case cmd of
      "quit"  -> return ()
      "read"  -> do doRead
                    main
      "write" -> do doWrite
                    main
      _       -> do putStrLn ("I don't understand the
                                command " ++ cmd ++ ".")
                    main

doRead = do
  putStrLn "Enter a file name to read: "
  fn <- getLine
  bracket (openFile fn ReadMode) hClose
        (\h -> do txt <- hGetContents h
                  putStrLn txt)

doWrite = do
  putStrLn "Enter a file name to write:"
  fn <- getLine
  bracket (openFile fn WriteMode) hClose
        (\h -> do putStrLn "Enter text (...) :"
                  writeLoop h)
```

Gambar 5.35 Source Code Penggunaan Input/Output Pada Haskell

```
writeLoop h = do
  l <- getLine
  if l == "."
    then return ()
    else do hPutStrLn h l
            writeLoop h
```

Gambar 5.36 Lanjutan Source Code Input/Output.

Program tersebut, memberikan tiga pilihan command. Apabila dipilih command quit, program akan berhenti. *Command read*, akan mengeksekusi fungsi doRead, yang berfungsi untuk melakukan pembacaan file, dengan meminta input nama file. Command terakhir yaitu write, yang akan mengeksekusi fungsi doWrite. Fungsi doRead merupakan fungsi yang digunakan untuk membuat sebuah file, yang menerima input nama file yang akan dibuat dan isi dari file.

Pada program tersebut, terdapat perintah *putStrLn* yang berfungsi untuk menampilkan sebuah nilai string ke layar. Kemudian pada baris berikutnya terdapat variable *cmd* dengan tipe *getLine*. *getLine* adalah sebuah action yang digunakan untuk membaca input dari *keyboard*. Untuk membaca file, digunakan perintah doRead, sedangkan untuk menuliskan file digunakan perintah doWrite.

Input/Output pada dasarnya adalah fasilitas tambahan pada bahasa Haskell, karena Input/Output merupakan fungsi dasar yang diambil dari pemrograman *imperative*. Pada bahasa C++, untuk pembacaan file dapat dilakukan seperti program berikut :

```
#include <iostream.h>
#include <fstream.h>
#include <string.h>
void main ()
{
cout<<"Masukkan nama file : ";
char namaFile[50];
cin>>namaFile;
```

Gambar 5.37 Source Code Penggunaan Input/Output Pada C++

```
ifstream fileBaca (namaFile);
if (!fileBaca)
{
      cerr<<"File Tidak Ada\n";
      return;
}
while (fileBaca)
{
      char buffer[100];
      fileBaca.getline (buffer, sizeof (buffer));
      cout<<buffer<<endl;
}
fileBaca.close ();
}
```

Gambar 5.38 Lanjutan Source Code Input/Output Pada C++

5.5 Modularitas.

Modul merupakan subcomponent dari program. Hal ini digunakan untuk mempermudah dalam membuat program yang besar. Dalam pembahasan ini akan dibahas mengenai kemampuan Haskell dalam memecah suatu program menjadi beberapa file, yang mencakup tentang kemampuan eksport/import.

Berikut ini adalah sebuah program sederhana untuk mendeklarasikan sebuah fungsi menampilkan sebuah tipe data binary tree menjadi bentuk List. Program dibagi menjadi dua module, yaitu yang berisi tipe data, dan berisi fungsi. Fungsi menerima input bertipe binary tree, kemudian memberikan output berupa sebuah List. Module pertama disimpan dengan nama Tree.hs :

```
module Tree ( Tree(Leaf,Branch) ) where
data Tree a = Leaf a | Branch (Tree a) a (Tree a)
```

Gambar 5.39 Source Code Modul Tree.hs

module kedua disimpan dengan nama Main.hs :

```
module Main where
import Tree ( Tree(Leaf,Branch) )

count_leaf  Empty                    = 0
count_leaf (Branch Empty a Empty)    = 1
count_leaf (Branch left a right)     = count_leaf left
                                     + count_leaf right
```

Gambar 5.40 Source Code Modul Main.hs

Pada module Tree.hs, module secara explicit mengeksport entitas yang terdiri dari Tree, Leaf, dan Branch. Kemampuan import sebuah module pada Haskell dapat dilihat pada module Main.hs.

Penggunaan modularitas pada dasarnya sama dalam setiap bahasa pemrograman. Kemampuan suatu bahasa dalam mendukung konsep modularitas merupakan inti dari sebuah bahasa pemrograman untuk mendukung penggunaannya dalam membuat program besar.

Bahasa C++, sebagaimana dalam Haskell mendukung konsep modularitas. Hanya saja modul utama saja yang diberi ekstensi CPP. Pada modul pendukung di beri ekstensi H. Sebagai contoh modul Tree.hs tersebut dapat diubah dengan bahasa C++ seperti berikut :

```
#include <iostream.h>

        typedef struct Tree{
        int data;
        Tree *left;
        Tree *right;
        }

        Tree *pohon
```

Gambar 5.41 Source Code Tree.h

Untuk module Main.hs, dapat direpresentasikan sebagai berikut :

```
#include <iostream.h>
#include <Tree.h>
       int count(Tree *root)
       {
       if (root == NULL) return 0;
       return count(root->left) + count(root->right) + 1;
       }
int main ()
 {
      Count(akar);
 return 0;
}
```

Gambar 5.42 Source Code Main.cpp

BAB VI

PENUTUP

6.1 Kesimpulan

Setelah diadakan penelitian dan pengujian terhadap dasar – dasar pemrograman Haskell, dapat ditarik kesimpulan sebagai berikut :

1. Haskell merupakan sebuah bahasa pemrograman fungsional yang memudahkan dalam mendeklarasikan fungsi – fungsi dan tipe data atau memiliki karakter sintaks *definable operator*.
2. Haskell juga mempunyai beberapa kemampuan dari pemrograman imperative, seperti input/output.
3. Haskell sangat sesuai untuk penyelesaian masalah – masalah yang bersifat matematis.
4. Kemampuan Haskell dalam membuat sebuah tipe data bentukan terstruktur, sangat mendukung untuk penyelesaian masalah yang berkaitan dengan *state space*.

6.2 Saran

Beberapa saran untuk penelitian dan pengembangan software yang berkaitan dengan Haskell adalah sebagai berikut :

1. Dalam penelitian ini, hanya dilakukan analisis mengenai kemampuan dasar Haskell, dari segi kelayakannya sebagai sebuah bahasa pemrograman fungsional. Untuk penelitian berikutnya diharapkan

analisis dilakukan lebih spesifik dengan menggunakan sebuah studi kasus tertentu.

2. Analisis berikutnya diharapkan dengan menyertakan sebuah bahasa pemrograman pembanding, sehingga diketahui lebih jelas mengenai kelebihan dan kekurangan Haskell dalam mengatasi suatu permasalahan.

DAFTAR PUSTAKA

[DAU06] Daume III, H., "*Yet Another Haskell Tutorial Revision 2*", http://darcs.haskell.org/yaht/yaht.pdf, diakses tanggal 27 Januari 2006.

[GIL97] Gilmore, S., "Programming in Strandard ML '97 : A Tutorial Introduction", http://www.dcs.ed.ac.uk/home/stg/NOTES/notes.pdf , diakses tanggal 15 Oktober 2006.

[HAR05] Harper, R., "Programming in Standard ML", http://www.cs.cmu.edu/~rwh/smlbook/offline.pdf, diakses tanggal 23 Juni 2006

[HUD99] Hudak, P., Peterson, dan Fasel, J. H. "*A Gentle Introduction to Haskell 98*",http://www.haskell.org/tutorial/haskell-98-Tutorial.pdf diakases tanggal 29 Januari 2006.

[MED03] Medak, D., dan Navratil, G., "*Haskell-Tutorial*", ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf, diakses tanggal 25 Januari 2006.

[PAG97] Page, R., "Two Dozen Short Lesson in Haskell", http://www.cs.ou.edu/~rlpage/fpclassCurent/textbook/haskell.shtml , diakses tanggal 6 Juli 2006.

[POP01] Pope, B., *A Tour of the Haskell Prelude*, http://undergraduate.csse.uwa.edu.au/ units/ 230.301/ lectureNotes/ tourofprelude.html, diakses tanggal 16 April 2006.

[RAH04] Raharjo, B., C++ Mengungkap Rahasia Pemrograman dalam C++, Bandung : Informatika Bandung, 2004

[SEB06] Sebesta R. W., *Concepts of Programming Language : Seventh Editions*. Boston: Pearson, 2006.