Haskell Lecture 1 — Introduction

September 30, 2013

みなさんこんにちは。Kinokkory です。この講義を受講してくださって、ありがとうございます。この講義の目標は、Haskell を実践的に使えるようになることです。この講義では、簡単なことや単に覚えればいいことにはあまり触れず、難しい部分を噛み砕いていくことに重きをおきます。少し進度が速いかもしれませんが、みなさんなら付いて来られると信じています。講義で分からないことがあれば、気兼ねなく質問してください。

「Haskell は楽しい！」この気持ちさえあれば大丈夫です。

Haskell の学習

Haskell を勉強する環境は整ってきています。日本語の文献もいくつか出ています。英語の文献はずっとずっと数が多いです。うまく使いこなして勉強してください。

- お勧めの本

  すごい Haskell たのしく学ぼう！

  Web 版 http://learnyouahaskell.com/chapters

  関数プログラミング入門

  関数プログラミングの楽しみ

- お勧めの入門サイト

  Haskell 入門 5 ステップ

  http://www.haskell.org/haskellwiki/Haskell 入門_5 ステップ

  10 分で学ぶ Haskell

http://www.haskell.org/haskellwiki/10 分で学ぶ Haskell

本物のプログラマは Haskell を使う

http://itpro.nikkeibp.co.jp/article/COLUMN/20060915/248215/

- リファレンス

Hoogle http://www.haskell.org/hoogle/

Hackage http://hackage.haskell.org/packages/hackage.html

ghc 利用の手引き http://www.kotha.net/ghcguide_ja/latest/

- プログラミングコンテスト

CodeForces http://codeforces.com/

AtCoder http://atcoder.jp/

NPCA Judge https://judge.npca.jp/

- Haskeller が見るべきサイト

Haskell Wiki http://www.haskell.org/haskellwiki/Haskell

Wikibooks Haskell 日本語 https://ja.wikibooks.org/wiki/Haskell

Wikibooks Haskell 英語 https://en.wikibooks.org/wiki/Haskell

開発環境

Haskell Platform — 一般ユーザー向き。GHC に加えていくつかのソフトウェアが含まれている。

GHC — Haskell のプロ向き。

Cabal — Haskell Platform にはもともと入っている。便利なパッケージ管理システム。

諸概念

ここでは、Haskell におけるさまざまな概念を言葉だけで説明します。これらを読むことで大まかなイメージを身に付けてくれればうれしいです。イメージをつかめるまで何度も読みかえしてみてください。大まかなイメージが付いたら、Haskell のコードをたくさん書いてください。

- **Haskell** — いま学んでいるプログラミング言語

  純粋関数型言語。すごく頭のいい人たちによって作られた最高にクールな言語。

  数学的で、静的で、とにかく美しい言語。

  参照透過性、遅延評価、型システム。これらがキーワードである。

- **変数** — 変わるようで変わらない数

  参照透過性が守られている。つまり、文脈によらず、変数は同じ値を持つ。ある意味では、定数である。数学における変数に近い。数学において、値を記憶するために変数を使うなどということはないだろう。

  普通のプログラミング言語における変数とは全く違う概念である。

- **関数** — 数学的な関数

  参照透過性が守られている。つまり、関数は与えられる引数が同じであるならば常に同じ値を返す。数学における関数に近い。

  関数は変数の一種であると考えていい。もしくは普通の変数は引数のない関数であると考えてもいい。

  関数も立派な値の一種であるから、関数を書くためのリテラルが用意されている。これを**ラムダ式**という。関数は、一般的な旧世代のプログラミング言語のように、名前を持たなければならないものではない。関数は実に、値の一種なのである。

  関数を引数に取る、あるいは、関数を返り値にする関数を、**高階関数**という。高階関数だから何か特別なのかというと、別にそうでもない。普通に使われるものである。

  Haskell において、すべての関数は一引数である。複数の引数を持つ関数は、**カリー化**という

処理を経て入れ子の一引数関数にすることができる。こうしてできるのも高階関数である。

演算子も関数の一種である。一項演算子には、正の符号+や負の符号-がある。二項演算子には加減乗除+, -, *, /などがある。二引数関数はすべて二項演算子として解釈することができる。また、**セクション**という便利な記法も用意されている。構文上、Haskell ではかなり自由に演算子をあつかえる。

❄ **式** — プログラムのかけら

プログラムは、一つの式である。その式が **main** である。

式はいくつかの式を組み合わせて作ることができる。変数や関数も、ある意味では式の別名に過ぎない。式は、プログラムのかけらであるとみることができる。

プログラムの実行とは、式の**簡約**である。簡約とは、式を変換して値に変えることである。プログラムの実行における時間概念は、式の簡約という一段階に帰着される。その時間を分割することは、基本的にはできない。

❄ **参照透過性** — すべては透き通る

式の内容が同じであるならば、式を簡約し終えた結果は常に同じである、ということである。これを守っている関数型言語を**純粋関数型言語**という。関数型言語の理論的基盤である**λ計算**はもともと参照透過であったが、純粋関数型言語は少ない。Haskell は実用にたえる数少ない純粋関数型言語の一つである。

参照透過を守ることには多くの問題が付きまとう。入出力や乱数生成は素直に行うと参照透過性を壊してしまう。また、変数への再代入（**破壊的操作**）もできないので、状態の遷移があつかえない。こういったことを解決するために、Haskell ではモナドという機構を用いている。モナドを使うことによって、参照透過性を壊す操作を、参照透過性を保つ操作へと変えることができるのである。

❄ **モナド** — 心強い味方

モナドは、計算を表すための機構である。

よく、モナドを理解するためには圏論を理解しなければならない、といわれるが、圏論という概念を通さなくともモナドは十分理解できる。実践的にモナドを使っていくことが何よりも重要である。

モナドは参照透過性のためだけにあるものではなく、より広く使われるものである。たとえば、構文解析においてもモナドは普通に使われる。モナドをうまく使いこなせれば Haskell の上級者であるといっていいだろう。

❄ **値** — 見えるようで見えないもの

整数、浮動小数、文字などに加え、タプル、リスト、関数、IO モナドなどがある。値としてあつかえるものは第一級の対象と呼ばれる。型は第一級の対象ではない。

数学において「実数はどこに存在するか」などといちいち考えないように、値がどのようなビットで表現されているか、ということはいちいち考えない。とうぜん計算機上に何らかの形で値はあるが、その状況は C 言語などと比べればかなり込み入っている。値というものは抽象的に捉えたほうがいい。シンプルにいえば、値は式の究極の形である。

無限ループなどによって簡約が停止しない場合も、その式の値は**⊥**(ボトム)であると考えられる場合がある。⊥は具体的なデータではない。あくまで値として見るとすっきりするから、値として認めてしまうのだと考えてほしい。**持ち上げられた型**はその型の値として⊥を持つ。持ち上げられた型のみ⊥を持つ。多くの複雑な型は持ち上げられている。**持ち上げられていない型**は、⊥を持たず、普通の値のみを持つ。つまり、簡約が停止した状態しかありえない。

引数に⊥が渡されても返り値が⊥にならない、という関数が存在する。このような関数は**非正格な関数**という。引数に⊥が渡されれば返り値は⊥である、という関数は、**正格な関数**という。一般的なプログラミング言語は正格な関数しかあつかえないが、Haskell は非正格な関数も正格な関数もあつかえる。

❄ **遅延評価** — 怠けものは強い

Haskell では、参照透過性という素晴らしい性質のおかげで、計算はいつ行っても構わない。

だから計算は必要になった時に必要な分だけすればいいことになる。これを遅延評価という。

また、式をうまく共有することによって、同じ式はなるべく一度しか評価されないようにする。これを**グラフ簡約**と言う。

❄ **再帰** — 人間の思考の根幹にあるもの

Haskell では再帰的な定義を頻繁に使う。再帰的な定義は多くの場合で分かりやすいのである。

再帰は多くのプログラミング言語で使うことができるが、たいていスタックオーバーフローなどの問題があり、多用されない。しかし、Haskell では、遅延評価を行っているため、多くの場合で再帰が問題とならない。

変数も関数も再帰的に定義することができる。そして相互再帰的に定義することもできる。一般に Haskell では、C 言語のように定義を書く順番を気にする必要はない。

再帰的な定義は、ある関数の**不動点**として定義しているのと同じことだと考えることもできる。関数 f の不動点とは、f(x)=x となる値 x のことである。

再帰には様々なものがあるが、その中に「性質のよい再帰」が存在する。性質のよい再帰には、数学的に扱いやすいということなどの利点がある。性質のよい再帰は、「**折り畳み**」と「**展開**」に大きく二分される。再帰についてはいくつもの興味深い話題がある。

❄ **リスト** — あまりにも身近なもの

リストは再帰的なデータ型の代表格である。もう少し詳しくいえば、**片方向連結リスト**である。どのようなリストも、値とリストの対（**cons**）であるか、空（**nil**）であるかのどちらかである。

一般にリストは関数型プログラミングにおいて多用されるものである。関数型プログラミングとはリストの処理である、という定義もあながち間違っていない。たとえば、文字列が文字の配列として解釈される C 言語とは対照的に、Haskell において文字列は文字のリストである。

Haskell の醍醐味のひとつとして、**無限リスト**がある。

また、Haskell には**リスト内包表記**という便利な糖衣構文がある。

❄ **型** — 多くを語るもの

型は Haskell において非常に重要な存在である。型を見るだけで非常に多くの情報を見ることができる。型に注目して考えるだけで、驚くほどうまくいく。

型の意義は、値に対する制約を記述し、なおかつその制約が守られているかをコンパイル時に検査できるということにある。

型には、基本型（整数型、浮動小数型、文字型など）に加え、**関数型**、そして、代数的データ型がある。

Haskell の型付けは、**型安全な強い静的型付け**である。型は厳格にあつかわれる。型が暗黙的にキャストされることはないし、関数がオーバーロードされて複数の型を持つということもない。型検査はコンパイル時に済まされ、型エラーがあればコンパイルは失敗する。

しかし、型は多相化することができる。たとえば、関数の引数について任意の型を取るようにすることができる。型を表すのには**型変数**というものが用いられる。これは C++のテンプレートや Java のジェネリクスのようなものである。多相的な型のおかげでかなり柔軟にプログラムを記述できる。

任意の型を取るのではなく、型に対して制約を付けることもできる。この制約を**型シグネチャ**という。型シグネチャとしては、一般に**型クラス**が使われる。型クラスは型に対するある種の制約である。ある意味では、オブジェクト指向におけるインターフェースのようなものである。ある型が型クラスの条件を満たしていることの宣言を**インスタンス宣言**という。ある意味では、インターフェースの実装のようなものである。

Haskell のコンパイラは式があるとき、その型を推察し、勝手に型を付けてくれる。これを**型推論**という。Haskell の型推論は、本当に強力である。型推論のおかげで、Haskell プログラマはある意味では型をいちいち書かずにプログラミングすることができる。C 言語と比べても、かなり楽に型を扱うことができるのである。

❄ **代数的データ型** — 美しいデータ

代数的データ型は、簡潔であるが非常に豊かな表現力を持つ。ある意味では、C 言語の構造体

と共同体を組み合わせたようなものである。代数的データ型には、**リスト**や**タプル**などがある。代数的データ型の一部は内部構造がプログラム上隠蔽されている。そのようなものとしては、IO モナドや State モナドなどがある。プログラマは、もちろん、新しい代数的データ型を自由に作ることができる。代数的データ型から情報を取り出すには、C 言語の構造体や共同体のように、メンバに対し**フィールド**という名前を付けることもできるが、もっと便利な方法として、**パターンマッチ**がある。パターンマッチは、複雑な条件を見た目通りの記法で簡潔かつ明快に表せるものである。パターンマッチは Haskell の醍醐味の一つである。代数的データ型の値を作り出すために使う関数は、**データ構築子**と呼ばれる。ある意味では、オブジェクト指向のコンストラクタのようなものである。代数的データ型の定義は、ある意味では、いくつかのデータ構築子を定義することに等しい。

再帰的な代数的データ型。

- **型の類** —

- **モジュール** — シンプルな箱

  分けて隠すために。

文法まとめ

次は、文法の全体像を見ていきましょう。詳しくは、

http://www.kotha.net/ghcguide_ja/latest/an-external-representation-for-the-ghc-core-language-for-ghc-6.10.html

を見て調べてください。

演習問題

Haskell Lecture 2 — Monad

October 4, 2013

関手

モナド

Maybe

IO

リスト

State

乱数

モナドの拡張

演習問題

Haskell Lecture 3 — Recursion

October 18, 2013

再帰的データ型

catamorphism

anamorphism

再帰と計算量

再帰とメモリ

演習問題

Haskell Lecture 4 — Data Structures

October 21, 2013

FingerTree

差分リスト

Zippers

ねじれヒープ

IntTree

演習問題

Haskell Lecture 5 — Case Studies

October 25, 2013

高速化

文字列処理

パーサ

プリティプリンタ

QuickCheck

演習問題