Haskell Lecture 1

kinokkory@shiatsumat

Haskellはどんな言語？

Haskellの特徴 — PLUS

Pure	純粋関数型言語
Lazy	遅延評価
Useful	実用的
Simple	簡潔

HaskellはEsolangではない

- Haskellは十分実用性のある言語です。
- Haskellはわかりやすさを重視した言語です。
- 最初はとまどうこともあるとおもいますが、いつのまにか体に染みこんでくるでしょう。
- 最初の一歩は重いものです。思いきって踏みだしていきましょう。

Pure — 純粋関数型言語

- 純粋 — 参照透過性が守られているということ。
- 参照透過性 — すべての式は、いつ計算しても同じ値をもつということ。
- 処理の実行されるタイミングを考えなくてよくなる。とくに、並行処理が簡単になる。
- 仕様を書くような感覚でプログラミングできる。

純粋関数型言語 pure functional language
参照透過性 referential transparency

Lazy — 遅延評価

- 遅延評価、くわしくいうと必要渡しを原則としている。
- 必要渡し — 式を、必要になったときだけ計算する。一度計算した値は、なるべく再利用する。
- 計算量を減らすことができる。とくに、高速に再帰的な処理をすることができる。

遅延評価 lazy evaluation
必要渡し call by need

Useful — 実用的

- 高速である。
- バグがすくない。
- 書きやすく、読みやすい。
- テスト駆動開発、C言語との連携、デバッガ、パッケージ管理などのための使いやすいツールがそろっている。
- 急成長しているので、これからもっと使われていくはず。

Simple — 簡潔

- 短く、明快で、管理しやすい。
- インデントや記号をうまく利用している。
- 抽象的な記述ができる。
- すっきりした、合理的な文法である。
- 中毒性があるくらいに、書きやすい。

Haskellを動かそう！

Haskell Platform

- Haskell Platformを入れましょう。以上！

- GHC (コンパイラ), GHCi (インタープリタ), Cabal (パッケージマネージャ), Haddock (文書ジェネレータ), 便利なライブラリなどが入っています。

エディタ

- VimやEmacsなど好きなものを使えばいいとおもいます。インデントの設定だけ気をつけましょう (タブではなくスペースを推奨します) 。

- IDEをインストールすることもできますが、おすすめできるIDEはいまのところありません。

動かそう！

- ghc *.hs — コンパイラ
- ghci — インタープリタ
- cabal install * — ライブラリの取得

Haskellの文法を一気に学ぼう！

どうやって？

- 基本的な概念の意味を概観していきます。
- リファレンスとして、 BNF記法に近いスタイルの文法を書いてあります。文法は実際よりも多少簡略化してあります。いまのところは無視してもかまいません。
- 基本的にトップダウンの順番で説明します。

どうすればいいの？

- 基本的な文法を網羅することを重視しているので情報量が多いですが、重要度のぶんだけ星印★をつけたので、それを参考にして適当に読みとばしていってください。

- 入門書を併用することを強くおすすめします。

- とりあえずざっと理解したら、プログラムを書いてみましょう。

記号一覧

- [*pattern*] オプション
- {*pattern*} 0回以上の繰りかえし
- {*pattern*}$^{+}$ 1回以上の繰りかえし
- (*pattern*) グループ化
- $pattern_1 | pattern_2$ 選択
- $pattern_{<pattern'>}$ 差異
- ... 同様に繰りかえす
- [a..zA..Z] アルファベット一文字

さっそくはじめよう！

プログラム

- プログラムは、複数のモジュールからなる。
- プログラムはMainモジュールのmain関数から実行される。
- もうすこしキザな言いかたをすると、プログラムの値はMainモジュールのmainである。

モジュール

- モジュールは、モジュール宣言とインポート宣言とトップレベル宣言からなる。一つのファイルが一つのモジュールをなさなければならない。
- モジュールは階層化できる。
- Mainのモジュール宣言のみ省略できる。
- モジュール名とファイル名は基本的に一致している必要がある (Mainモジュールは例外)。また、モジュールの階層はフォルダの階層と一致している必要がある。

エクスポート宣言

- モジュール宣言にはエクスポート宣言を含めることができる。エクスポート宣言というのは、いわゆるpublicにするエンティティを指定することである。ほかのエンティティはprivateになる。エンティティとは、変数/関数や型や型クラスなどである。
- エクスポート宣言を付けない場合、すべてのエンティティがpublicとなる。

インポート宣言

- インポート宣言とは、あるモジュールでpublicになっているエンティティを使えるようにするためのものである。
- インポートするエンティティを指定することもできる。
- デフォルトではとくにプレフィクスを付けないで使えるが、プレフィクスを付けて使うようにすることもできる。

module →

module *modid* **where**
body
| **module** *modid* **(***export*$_1$**, …) where**
body
| *body*　　module Main (main) where を省略したことになる

modid → {*modid'***.**} *modid'*

modid' → [A..Z] {[a..zA..Z0..9']}　一文字目は大文字

export →

qvar

| *qtycon* 型構築子のみ

| *qtycon* **(..)** 型構築子・データ構築子・フィールドラベルのすべて

| *qtycon* **(***cname*$_1$**, ...)** データ構築子とフィールドラベルを指定

| *qtycls* 型クラスのみ

| *qtycls* **(..)** 型クラスとクラスメソッドのすべて

| *qtycls* **(***qvar*$_1$**, ...)** クラスメソッドを指定

| **module** *modid*

body → {*impdecl*}{*topdecl*}

impdecl →

import *modid* [*impspec*]
| **import** *modid* **as** *modid'* [*impspec*]
| **import qualified** *modid* [*impspec*]
| **import qualified** *modid* **as** *modid'* [*impspec*]

デフォルトではモジュール名がプレフィックス

プレフィックスを指定

qualifiedでプレフィックスを義務化

impspec →

(*import*$_1$**, ...)** インポートするものを指定
| **hiding (***import*$_1$**, ...)** インポートしないものを指定

import →

var
| *tycon* [**(..)** | **(***cname*$_1$**, ...)**]
| *tycls* [**(..)** | **(***var*$_1$**, ...)**]

topdecl →

    typedecl
| *datadecl*
| *newtypedecl*
| *classdecl*
| *instdecl*
| *defaultdecl*
| *decl*

decl →

    vardecl
| *sigdecl*
| *fixitydecl*

型

- Haskellの型は表現力豊かである。型を見るだけでかなりの情報を得られる。また、型エラーだけでかなりのバグを検出できる。
- 型安全 — 不正な型変換がない。
- 静的型付け — コンパイル時に式の型が決まる。

型安全 type-safe
静的型付け static typing

多相型

- パラメータ多相 — 型を型引数によって全称量化することができる。C++のテンプレート関数のようなものである。
- アドホック多相 — 型引数に型クラスという制約をつけて全称量化することができる。C++の関数オーバーロードのようなものである。アドホック多相は、Haskellの重要な特徴である。

多相型 polymorphic type
パラメータ多相 parametric polymorphism
アドホック多相 ad-hoc polymorphism

様々な型

- 数値型 Int, Integer, Float, Double, ...
- 文字型 Char (文字列型は[Char])
- リスト型 [a]
- ユニット型 ()
- タプル型 (a,b,c)
- 関数型 a->b

関数型

- 関数型は一引数関数を表す。
- 2引数関数は次の2つの表現方法がある。
  - f :: (a,b)->c
    - x :: a, y :: b のとき、f (x,y) :: c
  - f :: a->b->c
    - x :: a, y :: b のとき、f x y :: c
- Haskellでは2番目をよく用いる。1番目から2番目への変換をカリー化という。また、「f x」のようにすることを部分適用という。

a->b->c = a->(b->c)
f x y = (f x) y

カリー化 currying
部分適用 partial application

型推論

- コンパイラは強力な型推論器を持っている。コンパイラはどの式についても、自動で型を付けて、型エラーをチェックしてくれる。
- 推論された型は多相型になりうる。Haskellでは、どの式についても、すべての可能性を含む最も条件のきつい型（主要型）が存在する。型推論器が推論するのはまさにこの主要型である。

型推論 type inference
主要型 main type

type → *btype* [**->** *type*] 関数型

btype → {*atype*}$^{+}$ 型適用

atype →

qtycon	型構築子
| *tyvar*	型変数
| **()**	ユニット型
| **(***type*$_1$**, … ,***type*$_n$**)**	タプル型
| **[***type***]**	リスト型
| **(***type***)**	

simpletype → *tycon tyvar*$_1$ … *tyvar*$_n$

qtycon → [*modid* **.**] *tycon*

tycon → [A..Z] {[a..zA..Z0..9’]} 一文字目は大文字

tyvar → [a..z] {[a..zA..Z0..9’]} 一文字目は小文字

ここからが本番！

代数的データ型

- ユーザーが作ることのできるデータ型。
- 大まかにいうと、代数的データ型とは「直積の直和」である。C言語でいえば、「構造体の共用体」である。直積はタプルとして考えてよいが、直和がすこしややこしい。直和はC言語の共用体と少し違い、どの状態であるかが保持されている。とくに、代数的データ型のうち、空の型の直和であるものは列挙型であるとみなすこともできる。

代数的データ型 algebraic data type

代数的データ型 2

- 代数的データ型は型変数を付けくわえることができる。C++のテンプレートクラスのようなものである。
- 代数的データ型で宣言する型を型構築子といい、それぞれの直和にたいする名前をデータ構築子という。データ構築子はデータを生成する関数としてみなせると同時に、データの状態を表すものともみなせて、パターンマッチに使うことができる。

型構築子 type constructor
データ構築子 data constructor

再帰的な代数的データ型

- 代数的データ型は再帰的に定義することができる。
- その代表格がリストである。ほかにも二分木や多分木や抽象構文木なども再帰的な代数的データ型として書くことができる。基本的に、Haskellは木構造の処理をおこないやすい。

フィールド記法

- 基本的に代数的データ型の中のデータはパターンマッチで取り出すことができるが、メンバに対しフィールドラベルという名前を付けることができる。
- フィールドラベルによって、代数的データ型の中のデータを取り出すことができるだけでなく、代数的データ型を簡単に更新するのにも使うことができる。また、パターンマッチにも使うことができる。

型クラス

- 型クラスは、アドホック多相あるいは関数オーバーロードを実現するためのものである。C++のインターフェースのようなものであり、メンバを持つ。型クラスは型ではない。
- 基本的にメンバの型のみ宣言するが、メンバにデフォルトの値を持たせることもできる。
- 型クラスには、Read (文字列から変換可能)、Show (文字列へ変換可能)、Num (数値)、Ord (順序)、Functor (関手)、Monad (モナド) などがある。

型クラス type class

インスタンス宣言

- インスタンス宣言とは、ある型が型クラスの条件を満たしていることの宣言である。具体的には、メンバの実装を含む。メンバの実装は、その型について具体的に行われる。
- 一部の型クラス (Eq, Ord, Enum, Bounded, Show, Read など) については、代数的データ型の宣言の際にderiving修飾子のあとに書いておくだけで実装を書かなくてもインスタンス化してくれる。

文脈

- ある型変数がある型クラスのインスタンスでなければならないという条件を加えるためもの。関数/変数の型宣言に用いることができる。
- 型クラスの宣言でも、型変数にたいし文脈によって制限をくわえることができる。この文脈は、宣言する型クラスを別の型クラスの子クラスにするためのものとして見ることができる。

型シノニム

- 単なる型の別名。C言語のtypdefのようなもの。
- ただし、型変数を持たせることができる。

正格性フラグ

- 代数的データ型の宣言において、中のデータが正格評価されることを保証するものである。正格評価とは、遅延評価と対照的な概念であり、即座に評価するということである。
- 正格性フラグを使いこなすのは割と難しいが、メモリの削減のために使うべきことも多い。とくに、Intなどのサイズの小さい型については使うほうがいいことが多い。
- さらに{-#UNPACK#-}というプラグマを付ける場合もある。

newtype宣言

- 既存の型と似ている型を宣言するときにつかう。あくまで型シノニムとは違い、新しい型の宣言である。
- 機能としては代数的データ型の宣言に正格性フラグをくわえたようなものであり、いわば糖衣構文である。

datadecl →

data [*context* **=>**] *simpletype* **=** $constr_1$ **|** ...
[*deriving*]

constr →

con [**!**] $atype_1$ ... [**!**] $atype_k$
| (*btype* | **!***atype*) *conop* (*btype* | **!***atype*)
| *con* **{** var_{11}**,** ..., var_{1m} **::** ($type_1$ | **!**$atype_1$)**,** ...**}**

deriving →

deriving *qtycls*
| **deriving (***qtycls***,** ...**)**

con → *conid* | **(***consym***)**

conop → *consym* | **`***conid***`**

conid → [A..Z] {[a..zA..Z0..9']}

consym → (: {*symbol* | :})$_{<reservedop>}$

newtypedecl →
newtype [*context* **=>**] *simpletype* **=** *newconstr* [*deriving*]

newconstr →
con atype
| *con* **{** *var* **::** *type***}**

var → *varid* | (*varsym*)

varid → [a..z] {[a..zA..Z0..9’]}

varsym → (*symbol* {*symbol* | :})$_{<reservedop>}$

typedecl → **type** *simpletype* **=** *type*

classdecl → **class** [*scontext* **=>**] *tycls tyvar*
where {*cdecl*$_1$**; …}**

class → *qtycls tyvar* | *qtycls* **(***tyvar atype*$_1$ **…** *atype*$_n$**)**

context → *class* | **(***class*$_1$**, …,** *class*$_n$**)**

simpleclass → *qtycls tyvar*

scontext → *simpleclass* | **(***simpleclass*$_1$**,…,***simpleclass*$_n$**)**

cdecl → *vardecl'* | *sigdecl* | *fixitydecl*

qtycls → [*modid* **.**] *tycls*

tycls → [A..Z] {[a..zA..Z0..9']}

tyvar → [a..z] {[a..zA..Z0..9']}

instdecl → **instance** [*scontext* =>] *qtycls inst*
[**where** {*idecl*$_1$**;** …**}**]

inst →
qtycon
| (*qtycon tyvar*$_1$ … *tyvar*$_n$)
| (*tyvar*$_1$**,** …**,** *tyvar*$_n$)
| [*tyvar*]
| (*tyvar*$_1$ **->** *tyvar*$_2$)

idecl → *vardecl'*

式

- 参照透過性が保たれている。
- 単体の変数/関数も式の一種である。変数/関数は式の別名として見てもいい。
- if式・case式・ラムダ式などがある。while式のような繰り返しの機構はない。

リテラル

- 整数 123 :: (Num a) => a
- 浮動小数 1.23 :: (Fractional a) => a
- 文字 ‘c’ :: Char
- 文字列 “abc” :: String (=[Char])

ラムダ式

- いわば「関数のリテラル」である。
- 以上。

リスト

- aのリストの型は[a]
- [1,2,3] :: [Int]
- [1,2,3] = 1 : 2 : 3 : [] (: は右結合)

パターン

- データについて条件分岐と分解を同時にするためのもの。とくに代数的データ型を分解するのにつかえる。おそろしく便利である。
- パターンを照合することをパターンマッチという。
- なんだかんだいってすぐ慣れる。
- 変数/関数宣言やcase式などにおいて使える。

ガード

- パターンにさらに条件分岐をつけくわえるためのもの。
- なんだかんだいってすぐ慣れる。

$exp \rightarrow exp^{0}\ [\texttt{::}\ [context\ \texttt{=>}]\ type]$ 型シグネチャ

$exp^{i} \rightarrow$
$\quad exp^{i+1}\ [qop^{(n,i)}\ exp^{i+1}]$
$\quad \mid\ lexp^{i}$
$\quad \mid\ rexp^{i}$

$lexp^{i} \rightarrow (lexp^{i} \mid exp^{i+1})\ qop^{(l,i)}\ exp^{i+1}$

$rexp^{i} \rightarrow exp^{i+1}\ qop^{(r,i)}\ (rexp^{i} \mid exp^{i+1})$

中置演算子の処理

$qop \rightarrow qvarsym \mid \texttt{`}qvarid\texttt{`} \mid qconsym \mid \texttt{`}qconid\texttt{`}$

$qvarsym \rightarrow [modid\ \texttt{.}\ ]\ varsym$

$qvarid \rightarrow [modid\ \texttt{.}\ ]\ varid$

$qconsym \rightarrow [modid\ \texttt{.}\ ]\ consym$

$qconid \rightarrow [modid\ \texttt{.}\ ]\ conid$

exp^{10} →

	¥ $apat_1$ … $apat_n$ **->** exp	ラムダ式
|	**let {**$decl_1$**; …} in** exp	局所宣言
|	**if** exp **then** exp **else** exp	if式
|	**case** exp **of {**alt_1**; …}**	case式
|	$doexp$	do式
|	$\{aexp\}^+$	関数適用

alt →

	pat **->** exp [**where {**$decl_1$**; …}**]
|	pat {**|** exp^0 **->** exp}$^+$ [**where {**$decl_1$**; …}**]

$aexp \rightarrow$

$qvar$	
$qcon$	
$literal$	
(exp)	
$(exp_1, \ldots, exp_n)$	タプル
$[exp_1, \ldots, exp_n]$	リスト
$[exp_1\ [, exp_2]\ ..\ [exp_3]]$	等差数列リスト
$[exp \mid qual_1, \ldots, qual_n]$	リスト内包記法
$(exp^{i+1}\ qop^{(a,i)})$	セクション
$(lexp^{i}\ qop^{(l,i)})$	セクション
$(qop^{(a,i)}_{<->}\ exp^{i+1})$	セクション
$(qop^{(r,i)}_{<->}\ rexp^{i})$	セクション
$qcon\ \{qvar_1 = exp_1; \ldots\}$	データの構築
$aexp_{<qcon>}\ \{qvar_1 = exp_1; \ldots\}$	データの更新

$pat \rightarrow pat^{0}$

$pat^{i} \rightarrow$

$pat^{i+1}\ [qconop^{(n,i)}\ pat^{i+1}]$

$\mid\ lpat^{i}$

$\mid\ rpat^{i}$

$lpat^{i} \rightarrow (lpat^{i} \mid pat^{i+1})\ qconop^{(l,i)}\ pat^{i+1}$

$rpat^{i} \rightarrow pat^{i+1}\ qconop^{(r,i)}\ (rapt^{i} \mid pat^{i+1})$

中置演算子の処理

pat^{10} →		
	apat	
|	*gcon* $apat^1$ … $apat^n$	データ構築子
apat →		
	var	変数
|	*var* [@ *apat*]	アズパターン
|	*qcon* {$qvar_1$=pat_1, …}	ラベル付きデータ構築子
|	*literal*	リテラル
|	_	ワイルドカード
|	(pat_1, … , pat_n)	タプル
|	[pat_1, …, pat_n]	リスト
|	~*apat*	反駁不可パターン
|	(*pat*)	

変数/関数宣言

- 型シグネチャのあとに定義を複数の等式として書いていく。左辺、とくに引数についてパターンやガードによる条件分岐をおこなうことができる。わりと自由なパターンマッチができて、複数の変数/関数を一気に定義することもできる。
- 型シグネチャには文脈をくわえることができる。
- 変数宣言は0引数の関数の宣言のようなものだととらえることができる。
- そのほか、局所宣言としてletやwhileがある。

$vardecl \rightarrow (funlhs \mid pat^{\emptyset})\ rhs$

$vardecl' \rightarrow (funlhs \mid var)\ rhs$

$funlhs \rightarrow$

$var\ \{apat\}^{+}$

$\mid\ pat^{i+1}\ varop^{(a,i)}\ pat^{i+1}$

$\mid\ lpat^{i}\ varop^{(l,i)}\ pat^{i+1}$

$\mid\ pat^{i+1}\ varop^{(r,i)}\ rpat^{i}$

$\mid\ \mathbf{(}funlhs\mathbf{)}\ \{apat\}^{+}$

$rhs \rightarrow$

$= exp\ [\mathbf{where\ \{}decl_1\mathbf{;\ ...\}}]$

$\mid\ \{\mathbf{|}\ exp^{\emptyset}\ \mathbf{=}\ exp\}^{+}\ [\mathbf{where\ \{}decl_1\mathbf{;\ ...\}}]$

モナド

- モナドは関数型プログラミング上で手続き型プログラミングを再構築するために考えられた。
- 決して複雑怪奇なものではなく、慣れれば非常に便利なものである。
- モナドはこわくなんてない！

モナドの意味

-- m a は a の型の値を出力する計算である。
class Monad m where
    -- x >>= f は計算 x の結果を f に渡し、
    -- 新たな計算を得るということである。
    (>>=) :: m a -> (a -> m b) -> m b
    -- return c は値cを返す単純な計算である。
    return :: a -> m a

破壊的操作とモナド

- 入出力、乱数生成、（メモリの削減のための）変数への代入など、プログラミングにおいて参照透過性を壊すような操作（破壊的操作）が必要となることは多い。
- IOモナドやSTモナドをもちいることによって、これらの操作を参照透過性を壊さずにおこなうことができるようになる。（実は IO a -> a という型を持つある関数を使うと参照透過性は守られないが、この関数は原則としてつかわないことが推奨されている。）

do記法

- モナドを手続き型言語らしく書くための便利な記法である。
- do記法は糖衣構文であり、単純なルールによって>>=をもちいたモナドらしい書き方に変換される。

doexp → **do {***stmt*$_1$**; …;** *stmt*$_n$**;** *exp***}**

stmt →

exp

| *pat* **<-** *exp*

| **let {***decl*$_1$**; …}**

あとすこし！

コメント

- -- 一行コメント
- {- 複数行コメント
  {-入れ子にできる-}
  複数行コメント -}

インデント

- Haskellには{a;b;...}というような記法とは別に、インデントを用いる記法もある。単に記法の違いである。
- インデントは、オフサイドルールというルールにもとづいて解釈される。

中置演算子

- 記号は中置演算子としてもちいることができる。
- 普通の名前もバッククオート`...`で囲めば中置演算子としてつかえる。
- 記号についてはセクションという機能もある。たとえば、x::a, y::b のとき x%y::c だとすると、
  - (%) :: a -> b -> c
  - (x%) :: b -> c
  - (%y) :: a -> c
- 中置演算子の結合性 (左/右/無結合) と優先順位を宣言することができる。

文芸的スタイル

- ¥begin{code}と¥end{code}ではさまれた部分だけがソースコードとなり、残りはコメントとなる。
- そのままTeXのソースコードとして読み取ることもできる。
- 標準ライブラリで多用されているスタイル。
- 拡張子は.lhsとする。

fixitydecl →
(**infixl** | **infixr** | **infix**) [integer] op_1**, …,** op_n

op → *varop* | *conop*

おつかれさまです！

- ほんとうに、おつかれさまです。
- これで、文法はほぼ網羅しました。
- 次のステップへ進みましょう！

Haskellライフを始めよう！

ライブラリ

- Hoogleで簡単に調べられます。ソースコードを見ることもできます。
  - http://www.haskell.org/hoogle/

マニュアル

- ghcマニュアルの和訳があります。
  - http://www.kotha.net/ghcguide_ja/latest/

本

- すごいHaskellたのしく学ぼう！
- 関数プログラミング入門
- 関数プログラミングの楽しみ
- プログラミングHaskell
- ふつうのHaskellプログラミング

サイト

- 本物のプログラマはHaskellを使う
  - http://itpro.nikkeibp.co.jp/article/COLUMN/20060915/248215/
- Wikibooks Haskell
  - http://ja.wikibooks.org/wiki/Haskell
- HaskellWiki
  - http://www.haskell.org/haskellwiki/Haskell

プログラムを書いてみよう！

- とにかく書いてみましょう。

- オンラインジャッジもぜひ利用してください。
  - https://judge.npca.jp/problems
  - http://codeforces.com/problemset
  - http://www.spoj.com/problems/classical/

プログラムを読んでみよう！

- 実際のHaskellのプログラムをたくさん読んでみましょう。
- とくに、Hoogle上でソースコードを参照することができるので、読んでみましょう。Data.Listなどがおすすめです。

Haskell Lecture 2 へつづく