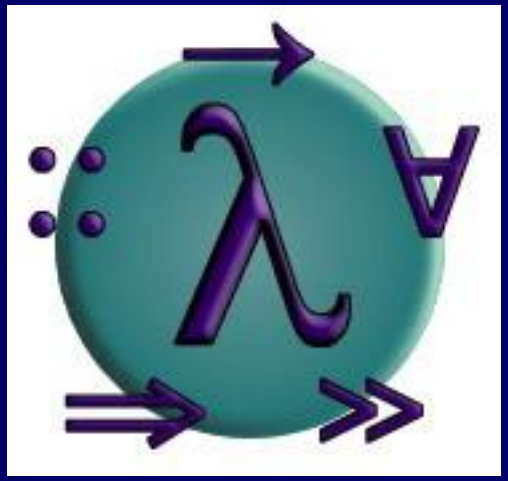
PROGRAMMING IN HASKELL
プログラミングHaskell

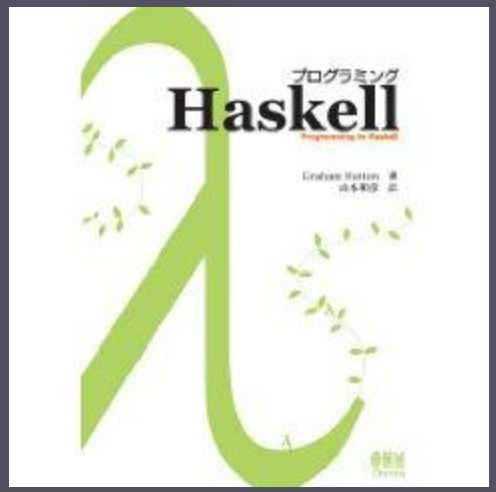


Chapter 1 – Introduction
導入

愛知県立大学 情報科学部 計算機言語論(山本晋一郎・大久保弘崇、2013年)講義資料
オリジナルは http://www.cs.nott.ac.uk/~gmh/book.html を参照のこと

はじめに

- 教科書
  プログラミングHaskell
  Graham Hutton (著), 山本 和彦 (翻訳)
  オーム社、全 232 ページ



- 成績: 試験とレポートを総合的に評価

参考 URL

- 愛知県立大学 情報科学部 計算機言語論
  (2013 年度の日本語版スライドなどを含む)
  http://www.ist.aichi-pu.ac.jp/lab/yamamoto/program_languages/
- 訳者によるサポートページ(正誤表など)
  http://www.mew.org/~kazu/doc/book/haskell.html
- 著者によるサポートページ
  (正誤表、英語版スライドなど)
  http://www.cs.nott.ac.uk/~gmh/book.html
- Haskellに関する包括的な情報源
  http://www.haskell.org/

それはどんな言語だ!?

- What's faster than C++, more concise than Perl, more regular than Python, more flexible than Ruby, more typeful than C#, more robust than Java, and has absolutely nothing in common with PHP? It's Haskell!
- C++よりも高速で、Perlよりも簡潔で、Pythonよりも例外が少なく、Rubyよりも柔軟で、C#よりも型付いていて、Javaよりも頑丈で、PHPとは似ても似つかない言語は? それは Haskell!

もっとも表現力に富んだ汎用プログラム言語は Clojure, CoffeeScript, Haskell
http://www.infoq.com/jp/news/2013/04/Language-Expressiveness

関数型プログラミング言語とは!?

- 全てが関数(全てが式と言い換えても良い)
  - 式は定数，変数，関数呼び出し(1 引数以上)から成る
    - 定数は一定の値を返す 0 引数の関数
    - 変数は 1 回だけ代入(変更できない 、何度参照しても同じ値)
  - 計算は式の値を求めること
- C 言語も(全てが)関数で構成されている
  - 本当?
    - if 文，for 文，ブロックは式ではない(値を持たない)
    - 関数を値にできない(ポインタ!?)
    - プログラムの実行中に新しい関数を作れない

ソフトウェア危機(The Software Crisis)

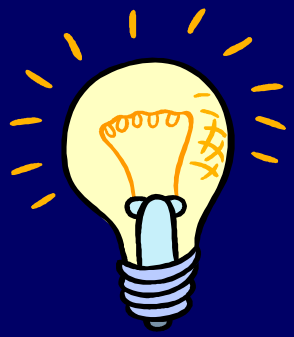
- プログラムの規模と複雑さにどのように対処すべきか?
- プログラム開発の期間とコストを削減する方法は?
- 開発したプログラムが正しく動作するという確信をどのようにしたら持てるのか?

プログラミング言語

次のような性質を備えた新しいプログラミング言語によりソフトウェア危機に対処する:

- 明確、簡潔、高い抽象度のプログラミングが可能
- ソフトウェア部品の再利用が支援されている
- 形式的検証(formal verification)が容易

- ラピッドプロトタイピング(rapid prototyping)が容易
- 問題解決の強力なツールとなり得る



関数型言語(functional languages)はこれらの目標を達成するためのエレガントな枠組み

関数型(プログラミング)言語とは?

さまざまな見解があり、正確に定義することは難しいが、一般論としては:

- 関数プログラミングとは、「関数を引数へ適用すること」を計算原理とするプログラミングの流儀(style)
- 関数型言語とは、関数プログラミングの流儀を支援(support)し、奨励(encourage)するプログラミング言語

例

1 から 10 を足し合わせる Java/C プログラム:

```
total = 0;
for (i = 1; i ≤ 10; i++) {
    total = total + i;
}
```

計算原理は変数への代入(variable assignment)

- 蓄えられている値が変化していく
- 人は変化を追跡することが苦手

例

1 から 10 を足し合わせる Haskell プログラム:

```
sum [1..10]
```

計算原理は関数を引数に適用すること
(function application)

[1..10] は整数のリスト [1, 2, 3,…, 10] を生成し、
sum はリストの要素の総和を求める

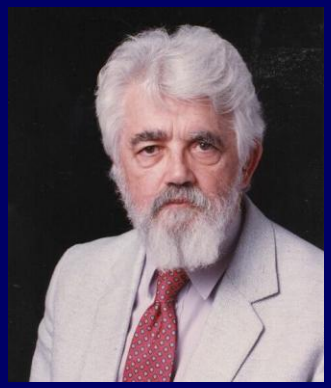
Historical Background

1930s:

Alonzo Church が単純だけど強力な関数の理論であるλ算法(lambda calculus)を開発

Historical Background

1950s:



John McCarthy が、最初の関数型言語である Lisp を開発。λ 算法に影響を受けているが、計算原理として変数代入を採用していた。

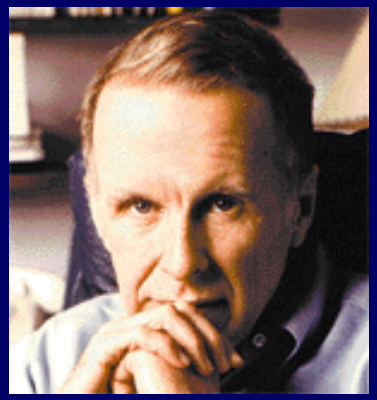
Historical Background

1960s:

Peter Landin が世界初の純粋な関数型言語である ISWIM を開発。λ 算法に強く基礎を置いており、変数代入を排除した。

Historical Background

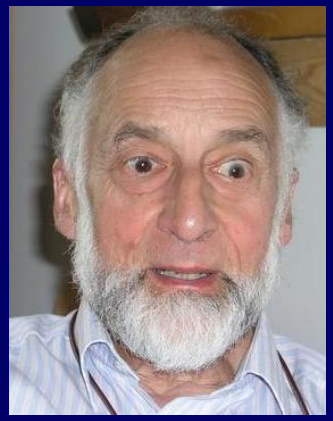
1970s:



John Backus が関数型言語 FP を開発。高階関数とプログラミングの論証に重点を置いた言語である。

Historical Background

1970s:



Robin Milner らが最初の現代的な関数型言語である ML を開発し、型推論と多相型を導入した

Historical Background

1970s - 1980s:

David Turner が遅延評価(lazy evaluation)をもつ関数型言語の開発を進めた(Mirandaとして結実)

lazyはほめ言葉!

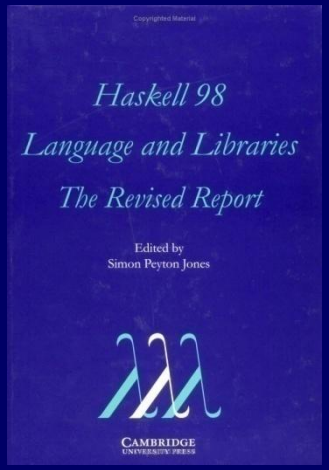
Historical Background

1987:

lazy な関数型言語が乱立したので、標準(Haskell)を設定する委員会が発足

Historical Background

2003:



委員会が Haskell 98 Report を公開し, 安定版の言語仕様を定めた(Haskell 2010 へ発展)

Haskell はこんな感じ(リスト)

要素は同一の型である
[1, 'a'] は許されない

```
[]          空のリスト(長さ0)
[1]         長さ1のリスト(要素は1)
[1,2]       長さ2のリスト(先頭要素は1、次の要素は2)
[1,2,3]     長さ3のリスト

x:xs        リスト xs の前に要素を 1 つ追加したリスト
1:[]        = [1]
1:2:[]      = 1:[2] = [1,2]
1:2:3:[]    = 1:2:[3] = 1:[2,3] = [1,2,3]
```

Haskell はこんな感じ(整数リスト)

```
[]                              // 長さ 0 の整数リスト
1:[]        = [1]               // 長さ 1 の整数リスト
1:2:[]      = 1:(2:[])          // 長さ 2 の整数リスト
            = 1:[2]
            = [1,2]
1:2:3:[]    = 1:(2:(3:[]))      // 長さ 3 の整数リスト
            = 1:(2:[3])
            = 1:[2,3]
            = [1,2,3]
```

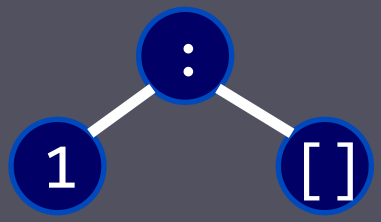
Haskell はこんな感じ(文字列リスト)

```
"a":[]     = ["a"]          // 長さ 1 の文字列リスト
"x":"y":[] = ["x","y"]      // 長さ 2 の文字列リスト

["Aichi","Gifu","Mie"]      // 長さ 3 の文字列リスト

"Shizuoka":["Aichi","Gifu","Mie"] =
["Shizuoka","Aichi","Gifu","Mie"]
                            // 長さ 4 の文字列リスト
```

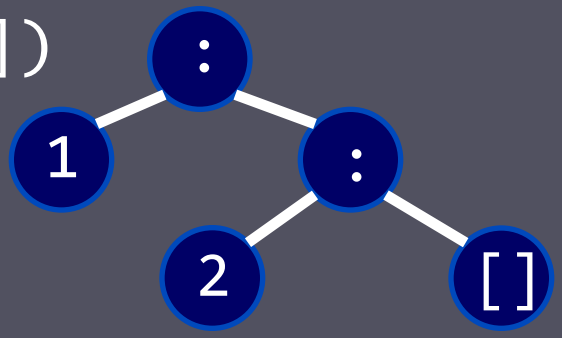
リスト(Cons 演算子)

演算子 `:' は、x:xs によって、リスト xs の前に要素を 1 つ追加したリストを返す

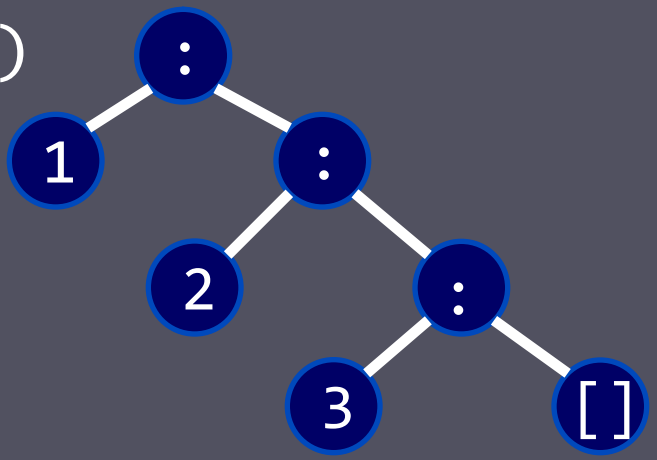
- `[1] = 1:[]`



- `[1,2] = 1:(2:[])`



- `[1,2,3] = 1:(2:(3:[]))`



Haskell はこんな感じ(リストの演算)

```
++              リストの連結
[] ++ [1,2,3] = [1,2,3]
[1,2,3] ++ [] = [1,2,3]
[1,2,3] ++ [4,5] = [1,2,3,4,5]
[1,2,3] ++ [] ++ [4,5] = [1,2,3,4,5]
head            リストの先頭要素
head [1,2,3] = 1
tail            リストの先頭要素を除いたリスト
tail [1,2,3] = [2,3]
```

Haskell はこんな感じ(リストの補足)

```
長さ 2 のリスト        [1,2]
                    = [1] ++ [2]
                    = [] ++ [1] ++ [2]
                    = [1] ++ [] ++ [2]
                    = [1] ++ [2] ++ []
==                  2 つのリストが等しいか否かを判定
> [1,2] == 1:[2]
True
> [1,2] == 1:[3]
False
```

Haskell はこんな感じ(数値リストの総和)

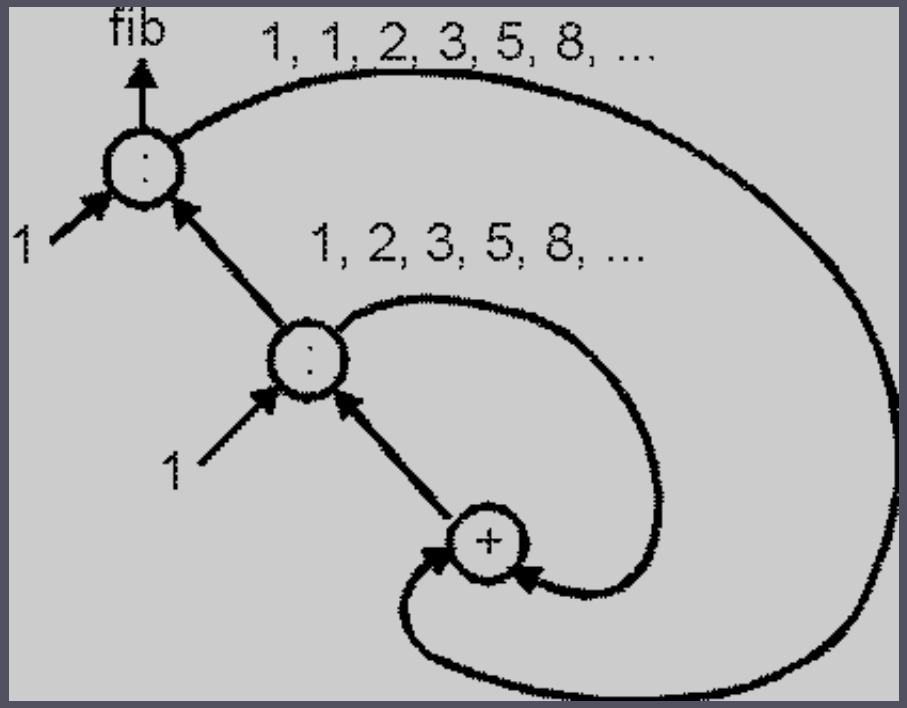
```
sum []     = 0
sum (x:xs) = x + sum xs


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

Haskell はこんな感じ(フィボナッチ数列)

```
fib  = 1:1:[a+b | (a,b) ← zip fib (tail fib)]
```

zip は 2 つのリストの要素を互いにペアにしたリストを返す

```
zip [1,2,3] ['a','b','c‘] = [(1,'a'),(2,'b'),(3,'c')]
```



Haskell はこんな感じ

```
q []     = []
q (x:xs) = q smaller ++ [x] ++ q larger
           where
             smaller = [a | a ← xs, a ≤ x]
             larger =  [b | b ← xs, b > x]
```

?

where は “ここで” あるいは “ただし” と読む

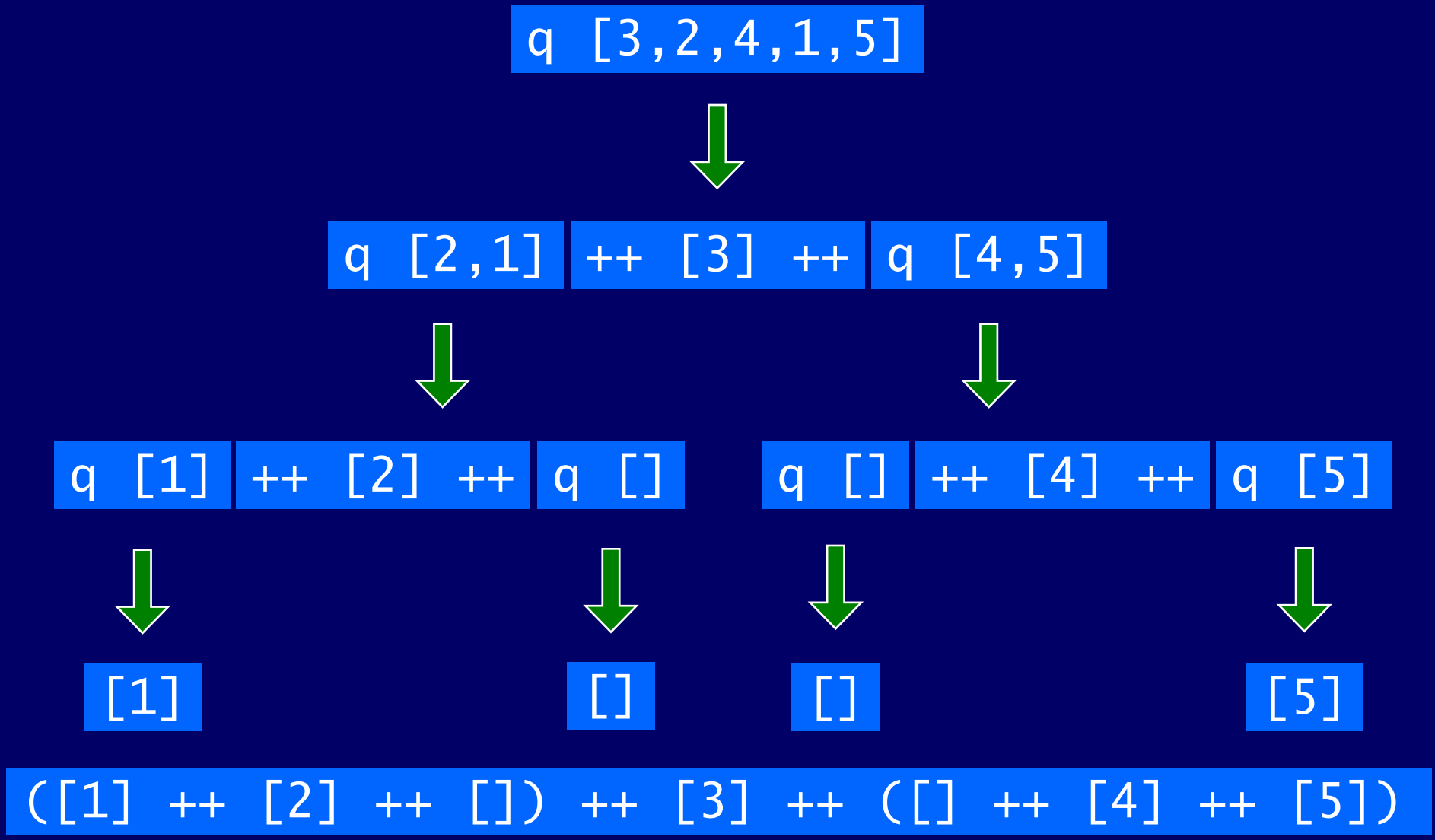
Haskell はこんな感じ(クイックソート)

他のプログラミング言語と比べて簡潔

```
q []     = []
q (x:xs) = (q smaller) ++ [x] ++ (q larger)
           where
             smaller = [a | a ← xs, a ≤ x]
             larger =  [b | b ← xs, b > x]
```

- 空リストはソート済み
- 非空リストのソートは以下の 3 つのリストの連結
  - (残りのリスト中の先頭要素以下の要素)をソートしたリスト
  - 先頭要素のみのリスト
  - (残りのリスト中の先頭要素より大きな要素)をソートしたリスト

例:



GHC (Haskell Platform)

- GHC は広く使われている Haskell 98 の実装
  - ghc はコンパイラ、ghciはインタプリタ
  - ghci の対話性は教育とプロトタイピングに適している
- Haskell Platform は，GHC にライブラリやツールなどを加えた環境一式を提供
  - Windows, Mac, Linux 版が利用可能
- 以下のサイトからダウンロードできる
  http://hackage.haskell.org/platform/

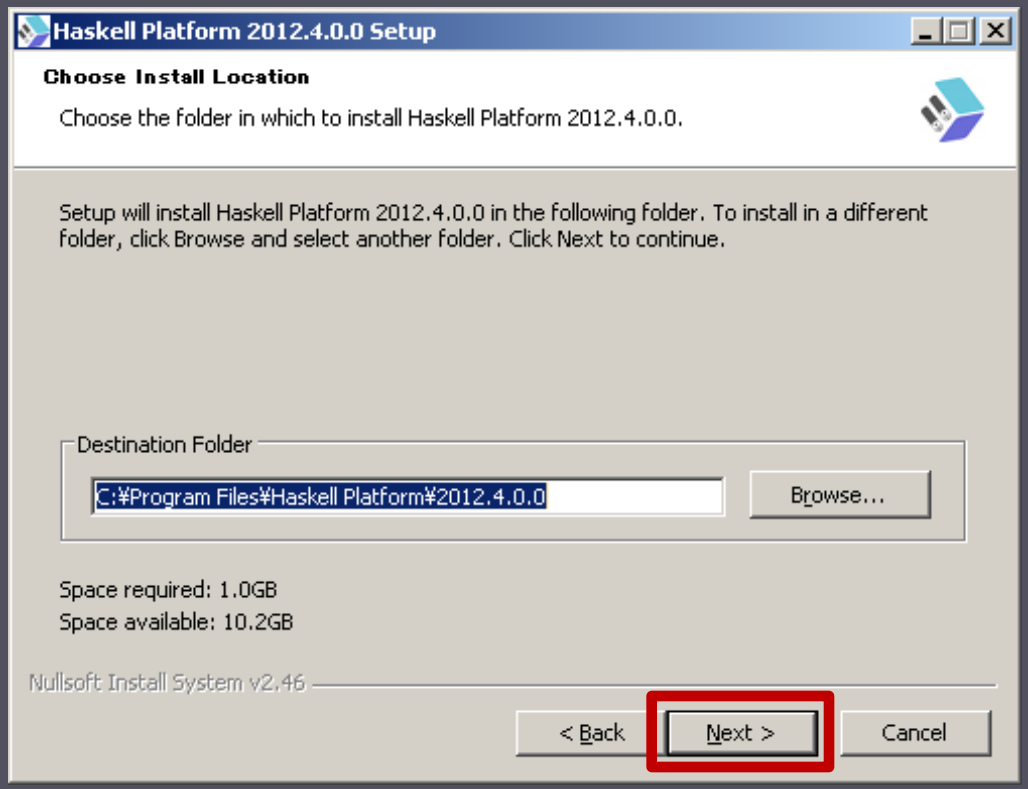
教科書では Hugs を使っているが、広く用いられている GHC (ghci)を用いれば良い

Haskell Platform のインストール (1/4)

- Windows へのインストール方法
  1. ブラウザから http://hackage.haskell.org/platform/windows.html へアクセス
  2. [Download Haskell for Windows] をクリックし，[HaskellPlatform-2012.4.0.0-setup.exe] を保存する
  3. 保存した [HaskellPlatform-2012.4.0.0-setup.exe] を実行する
  4. [Next] を押す

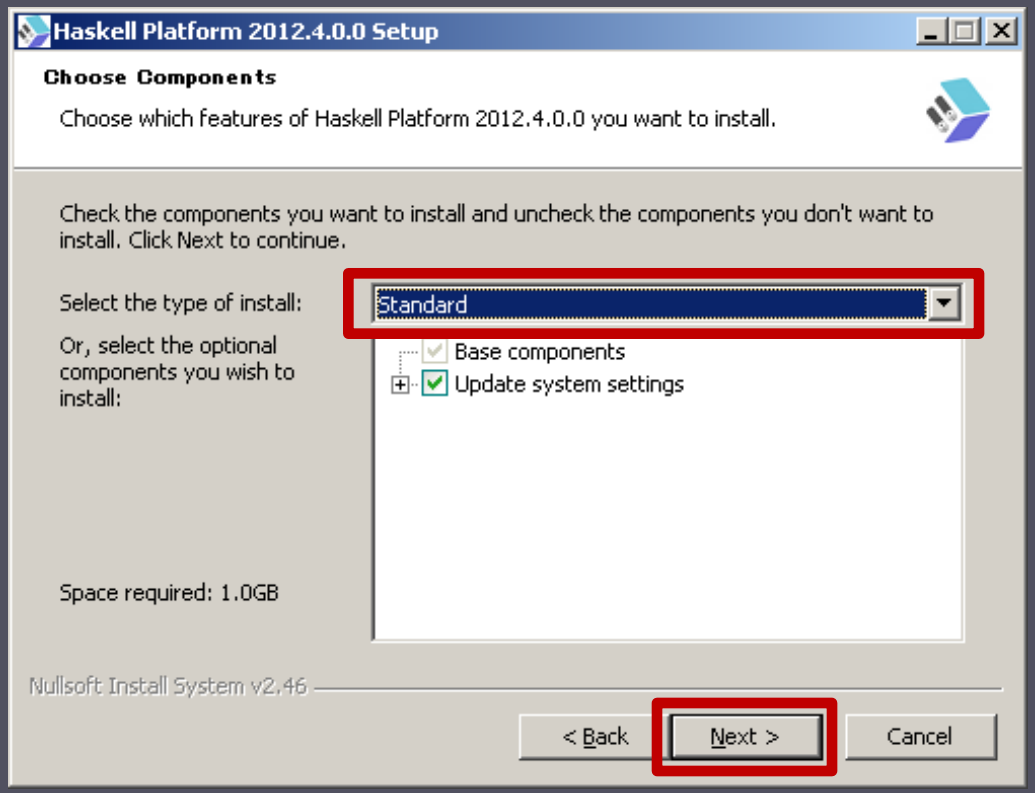
Haskell Platform のインストール (2/4)

- Windows へのインストール方法
  5. インストールの場所を選択し，[Next] をクリックする



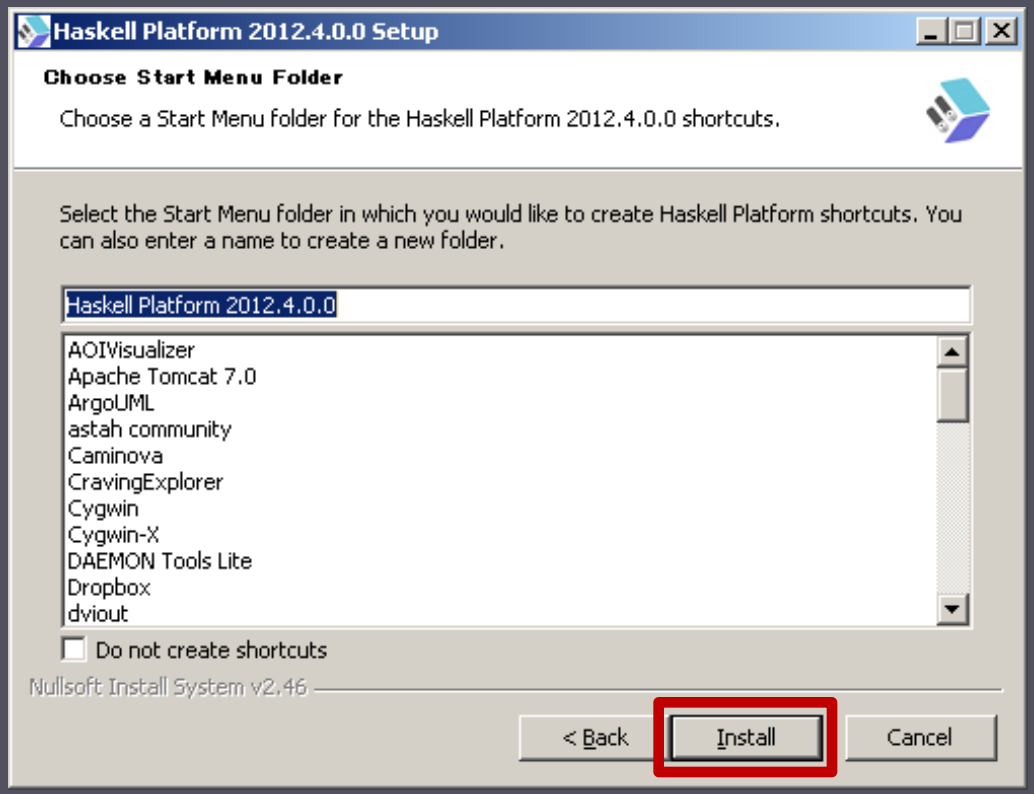
Haskell Platform のインストール (3/4)

- Windows へのインストール方法
  6. インストールのタイプは [Standard] を選択し，[Next] をクリックする



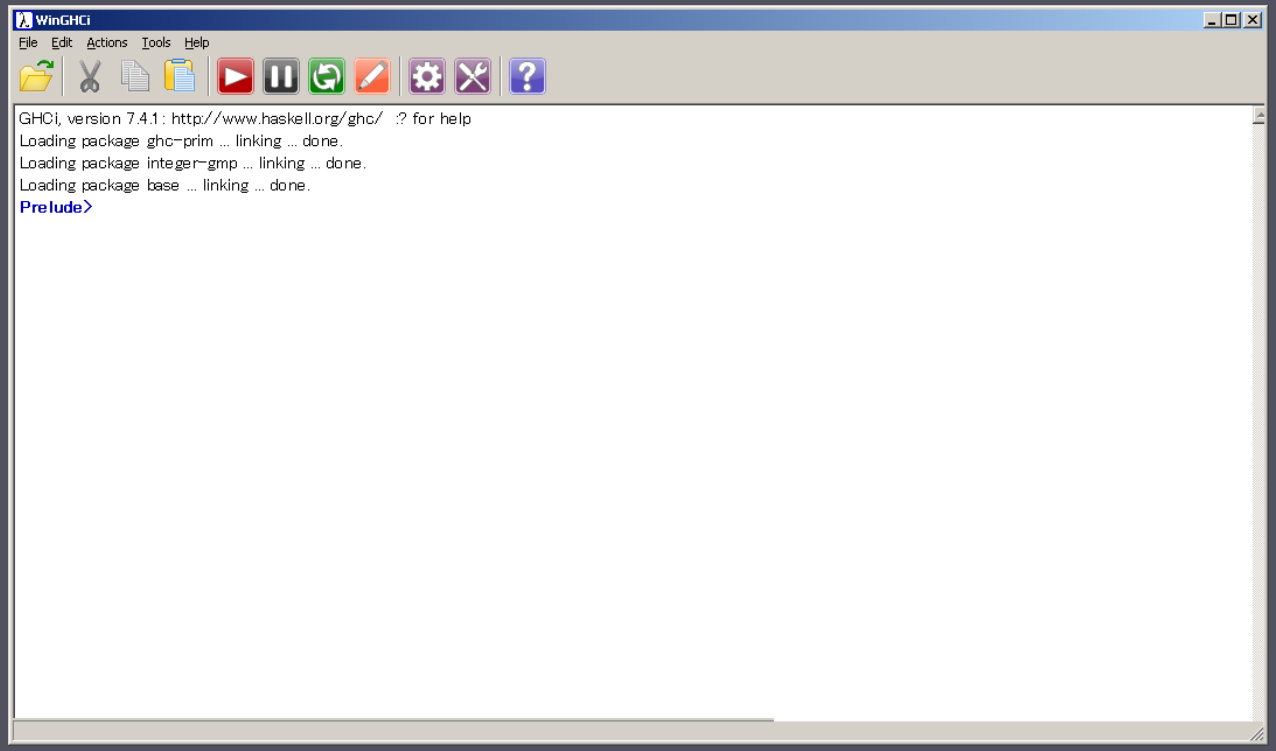
Haskell Platform のインストール (4/4)

- Windows へのインストール方法
  7. [Install] をクリックするとインストールが実行される



WinGHCi の起動方法

- 以下の手順で WinGHCi を起動する
  - [スタートボタン] → [すべてのプログラム] → [Haskell Platform 2012.4.0.0] → [WinGHCi]



Starting Hugs

- UNIX では、シェルプロンプトから hugs (ghci) を起動する
- Windows では、スタートメニュの Haskell Platformから、GHCi か WinGHCi を起動する

```
% hugs

__   __ __  __  ____   ___                ________________________________________________
||   || ||  || ||  || ||__                Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||               Copyright (c) 1994-2005
||---||         ___||                     World Wide Web: http://haskell.org/hugs
||   ||                                   Bugs: http://hackage.haskell.org/trac/hugs
||   || Version: September 2006 _____________________________________________________

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type :? for help
Hugs>
```

```
% ghci
GHCi, version 6.12.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude>
```

まとめ(1章)

- 関数(型)プログラミング言語
  - 「関数を引数へ適用すること」を計算原理とする
- 手続き型言語(C や Java 等)と何が違うの?
  - 「変数への代入」が計算原理
    - 蓄えられている値が変化していく、変化を追跡することは難しい
- 簡潔なプログラミングの例
  - 1 から 10 を足し合わせる

```
sum [1..10]
```

  - クイックソート

```
q []     = []
q (x:xs) = (q smaller) ++ [x] ++ (q larger)
           where smaller = [a | a ← xs, a ≤ x]
                 larger =  [b | b ← xs, b > x]
```

- 処理系
  - Windows, Mac, Linux 版の GHC がダウンロード可能