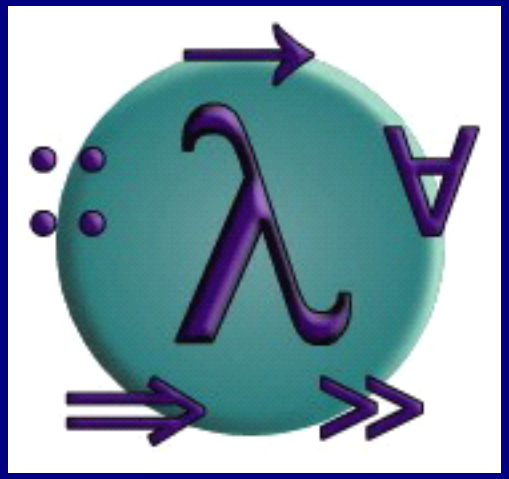
プログラミング Haskell



第1章　導入

目　次

λ

ギリシャ文字　λラムダ[小]　／　Λラムダ[大]
（IMEでは、ラムダで変換すると出てきます）
アスキーアートでは、人として扱われる
　例　λ…　人がしょんぼりとどこかへ去ってゆくさま

ギリシア文字							
Α	Β	Γ	Δ	Ε	Ζ	Η	Θ
α	β	γ	δ	ε	ζ	η	θ
Ι	Κ	Λ	Μ	Ν	Ξ	Ο	Π
ι	κ	λ	μ	ν	ξ	ο	π
Ρ	Σ	Τ	Υ	Φ	Χ	Ψ	Ω
ρ	σ,ς	τ	υ	φ	χ	ψ	ω

Haskellの成分分析結果

- Haskellの59%は真空で出来ています。
- Haskellの23%は鍛錬で出来ています。
- Haskellの8%は気の迷いで出来ています。
- Haskellの7%はミスリルで出来ています。
- Haskellの3%は成功の鍵で出来ています。

1.関　数

1つ以上の引数を取り、1つの結果を返す

例　**double x ＝ x ＋ x**

double 3
＝ 3 ＋ 3　　･･･　**double**を適用
＝ 6　　･･･　＋　を適用

1.関　数（入れ子）

double (double 2)

内側の**double**を適用

= **double** (**2** + **2**)

{ ＋を適用 }

= **double 4**

{ **double**を適用 }

= **4** + **4**

{ ＋を適用 }

= **8**

外側の**double**を適用

= **double 2** + **double 2**

{ １番目の**double**を適用 }

= (**2** + **2**) + **double 2**

{ １番目の＋を適用 }

= **4** + **double 2**

{ **double**を適用 }

= **4** + (**2** + **2**)

{ ２番目の＋を適用 }

= **4** + **4**

{ ＋を適用 }

= **8**

2.関数プログラミング

- Haskellで利用されている「関数」は、プログラミング言語でいうところの関数ではなく、数学的な意味合いでの関数に相当する。
- 関数型言語では、高レベルな What（目的は何か）を記述する
  手続型言語では、低レベルな How（目的をどのように達成するか）を記述する。

  ※手続き型言語の思考法に慣れたプログラマにとっては、移行が難しいと感じるケースが多い 。
- Haskellでは、再代入(破壊的代入)は許されない
  例　a = a + 1（再代入できることを副作用という）

  ※値の状態が変更されない制約があるため、堅牢性が高くバグも減らせる。
- プログラムの各部分の実行順序を任意に選ぶことができるため並列演算への応用が容易である。

2.関数プログラミング　例

1から5までの数を足し合わせる計算（Java）

```
total = 0;
for (i = 1; i <= 5; ++i)
   total = total+i;
```

Haskellでは、
値の再代入
はできない

※計算の基本は蓄えられている値を変えること

1から5までの数を足し合わせる計算（Haskell）

```
sum [1..5]
```

※計算の基本は関数を引数に適用すること

2.関数プログラミング　補足1

例　お茶を頼む場合

手続型言語　How

お湯を沸かして

↓

急須を用意して

↓

お茶の葉を入れて

↓

…

関数型言語　What

お茶をお願い

手順ではなく「何をしてほしいか」を記述する。

2.関数プログラミング　補足2

Haskellは純粋関数型言語です。

この場合の「純粋」とは、副作用がないことを意味します。
同じ引数を渡した関数はいつも同じ値を返し、状態の変更を
引き起こす代入も存在しません。
このことを「参照透過性」と呼びます。

Haskellは「遅延評価」が基本です。値の計算は必要になったとき
にはじめて行われます。

Haskellのこれらの性質は、参照透過性のおかげで暗黙の状態が
ないことから、保守性が高く、また遅延評価のおかげで不必要な
計算を避けることができ、無限列を簡単に取り扱うことができます。

2.関数プログラミング　補足3

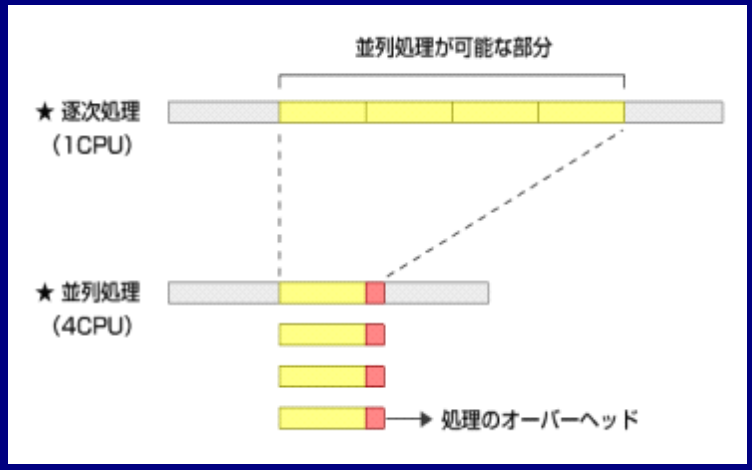
関数型言語比較一覧の表

言語	純粋関数型	遅延評価	静的型付け	その他の特徴
Haskell	純粋型	遅延評価	静的型付け	無限長のリストの処理などにおいて効力を発揮します。
Erlang	非純粋型	先行評価	動的型付け	並列処理に適したプログラミング言語。プロトコルの変換やスイッチを管理したりするテレコミュニケーションシステムで多用されています。
Scala	非純粋型	先行評価	静的型付け	Javaプラットフォーム（Java仮想マシン）上で動作し、既存のJavaのプログラムと容易に連携させることができます。
LISP	非純粋型	先行評価	動的型付け	全てのプログラミング言語の中でも2番目に古い高級言語であり、また、実装が比較的容易なため、非常に多くの方言が存在します。人工知能の分野で幅広く利用されています。
Scheme	非純粋型	先行評価	動的型付け	Lispの方言のひとつであり、コンパクトな設計が特徴です。
Ocaml	非純粋型	先行評価	静的型付け	強力な型推論を最大の特徴とします。関数型言語としてはかなり高速に動作します。
F#	非純粋型	先行評価	静的型付け	.NET Frameworkに対応した関数型言語。コア言語部分においてはC#、Ocaml、Haskellと互換性があります。

http://codezine.jp/article/detail/4254?p=2

2.関数プログラミング　補足4

これまでCPUはムーアの法則に従って高速化してきたが、現在、CPUは高速化よりも並列化が進んでいる。
今後、並列化プログラミングが重要になってくる。



Map関数

与えられたリストを別のリストに変換

Reduce関数(Haskellではfold?)

与えられたリストをある演算で計算

3.Haskellの特徴

- 簡潔なプログラム（2章と4章）
- 協力な型システム（3章と10章）
- リスト内包表記（5章）
- 再帰関数（6章）
- 高階関数（7章）
- モナド（8章と9章）
- 遅延評価（12章）
- プログラムの論証（13章）

4.歴史的背景

- 1930年代

Alonzo Churchは、単純だが強力な関数の理論であるラムダ計算を考え出した。

- 1950年代

John　McCarthyは、最初の関数型言語とみなされているLispを開発した。※変数の代入を採用。

- 1960年代

Petter　Landinは、ISWIMを開発した。λ計算に基づいた最初の純関数型言語である。

4.歴史的背景

- 1970年代

John Backusは、FPを開発した。FPは、高階関数とプログラミングの論証という特徴を持つ。

- 1970年代

Robin Milnerたちは、MLを開発した。MLは、最初のモダンな関数型言語であり、多相型と型推論を導入した。

- 1970年代～1980年代

David Turnerは、いくつかの遅延関数型言語を開発した。その頂点は商用のMirandaである。

4.歴史的背景

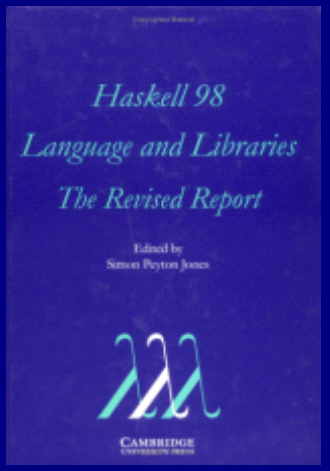
- 1987年

研究所で構成した国際委員会が、（論理学者のHaskell Curry に因んだ）Haskellを遅延関数型言語の標準となるべく開発を始めた。

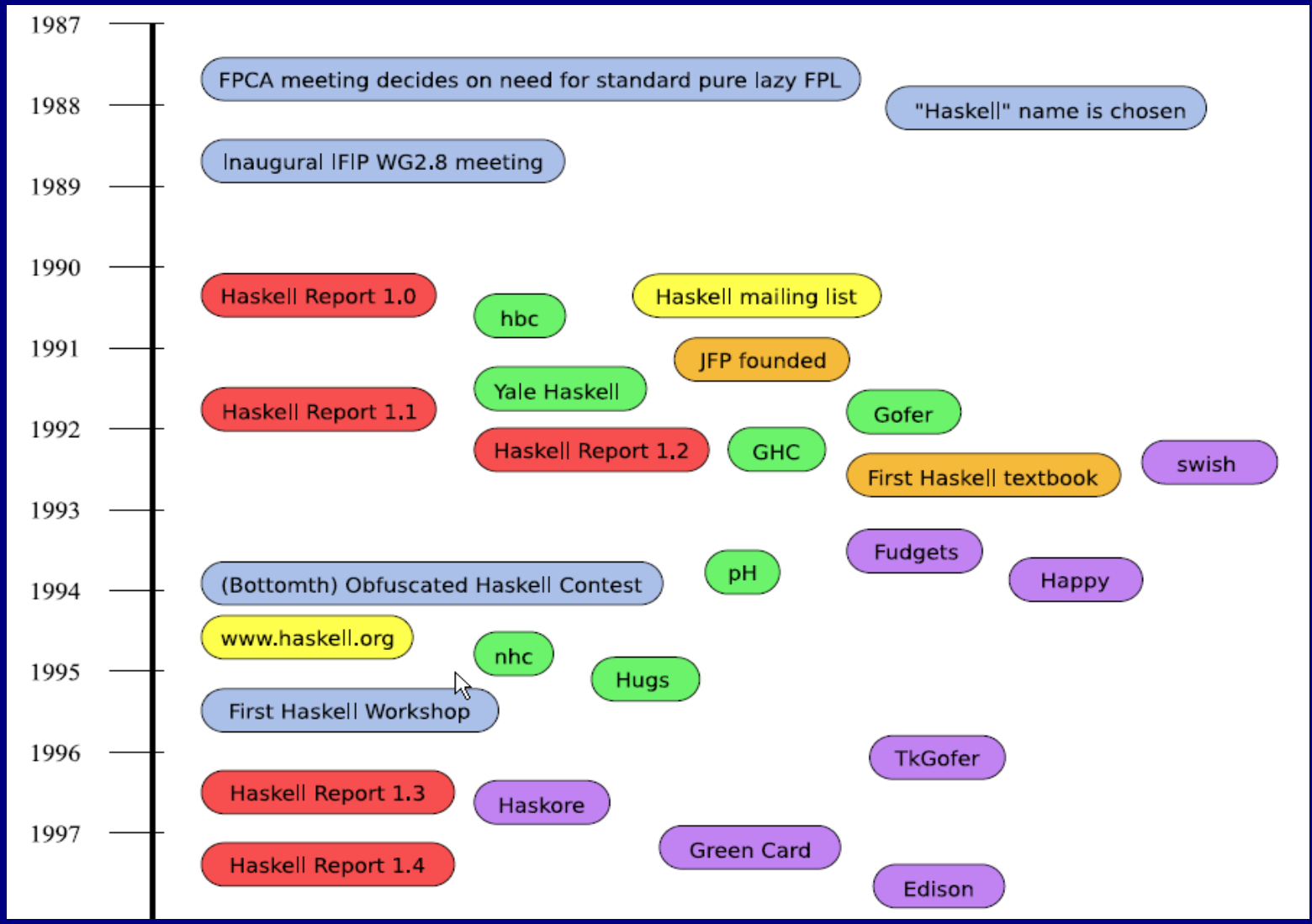
Haskell
A Purely Functional Language

- 2003年

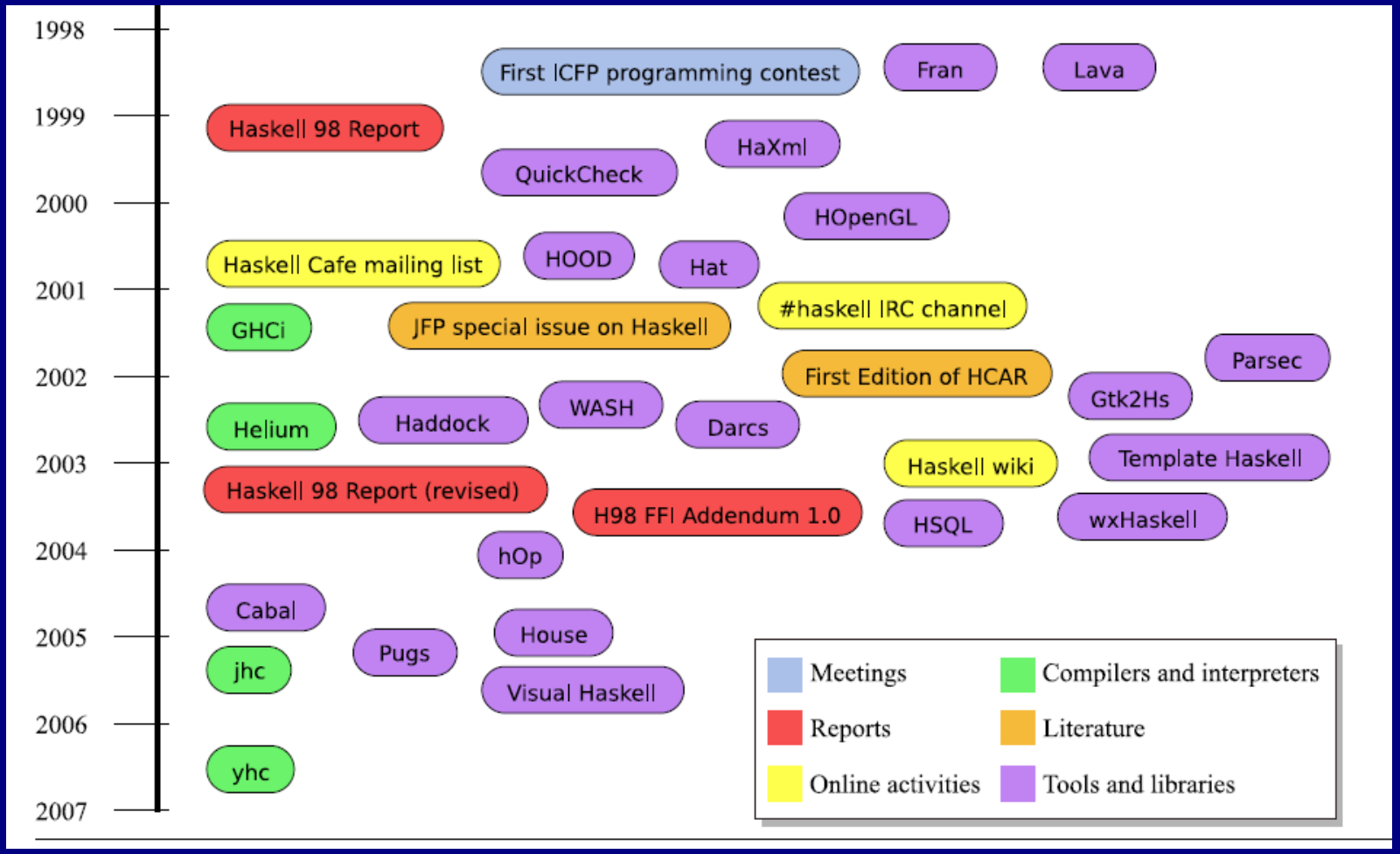
この委員会は、設計者の15年及ぶ作業の成果として、Haskell Reportを公開。



4.歴史的背景

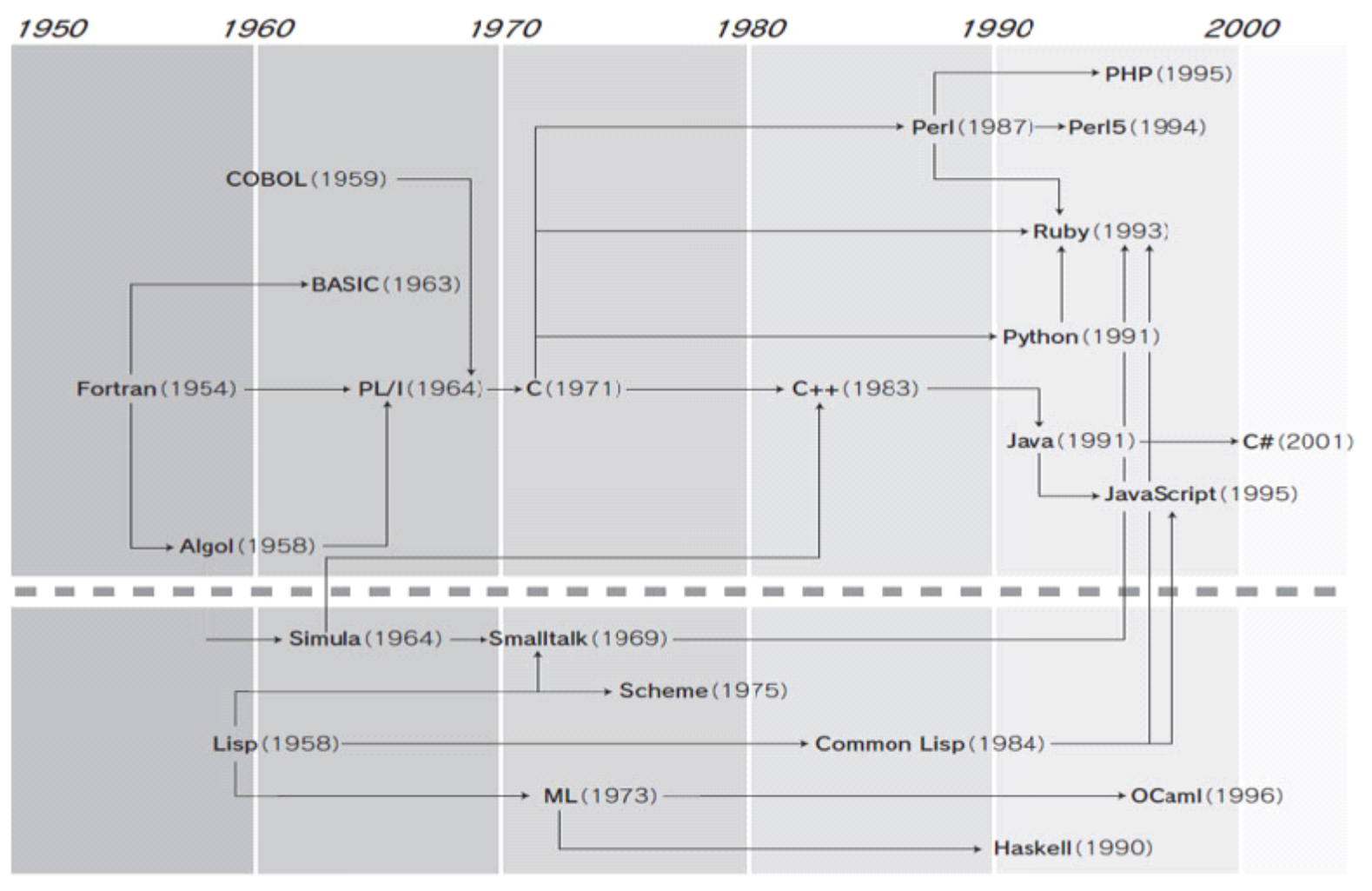

1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
FPCA meeting decides on need for standard pure lazy FPL
"Haskell" name is chosen
Inaugural IFIP WG2.8 meeting
Haskell Report 1.0
hbc
Haskell mailing list
JFP founded
Yale Haskell
Haskell Report 1.1
Gofer
Haskell Report 1.2
GHC
First Haskell textbook
swish
Fudgets
pH
Happy
(Bottomth) Obfuscated Haskell Contest
www.haskell.org
nhc
Hugs
First Haskell Workshop
TkGofer
Haskell Report 1.3
Haskore
Green Card
Haskell Report 1.4
Edison

4.歷史的背景



http://research.microsoft.com/en-us/um/people/simonpj/papers/history-of-haskell/history.pdf

4.歴史的背景



http://www.itmedia.co.jp/enterprise/articles/0703/26/news021.html

5.Haskellの妙味

例題1　　*sum[]　=　0*

sum(x:xs)　=　x　+　sum　xs

sum[1,2,3]　　リストが空になった時点で再帰が止まる

=　1　+　sum[2,3]　　・・・{　sum　を適用　}

=　1　+　(2　+　sum　[3])　　・・・{　sum　を適用　}

=　1　+　(2　+　(3　+　sum[])　　・・・{　sum　を適用　}

=　1　+　(2　+　(3　+　0)　　・・・{　+　を適用　}

=　6　　空リストの合計として0を返すのは適切

単位元・・・加法での0や乗法での1など

5.Haskellの妙味

例題2　　クイックソート

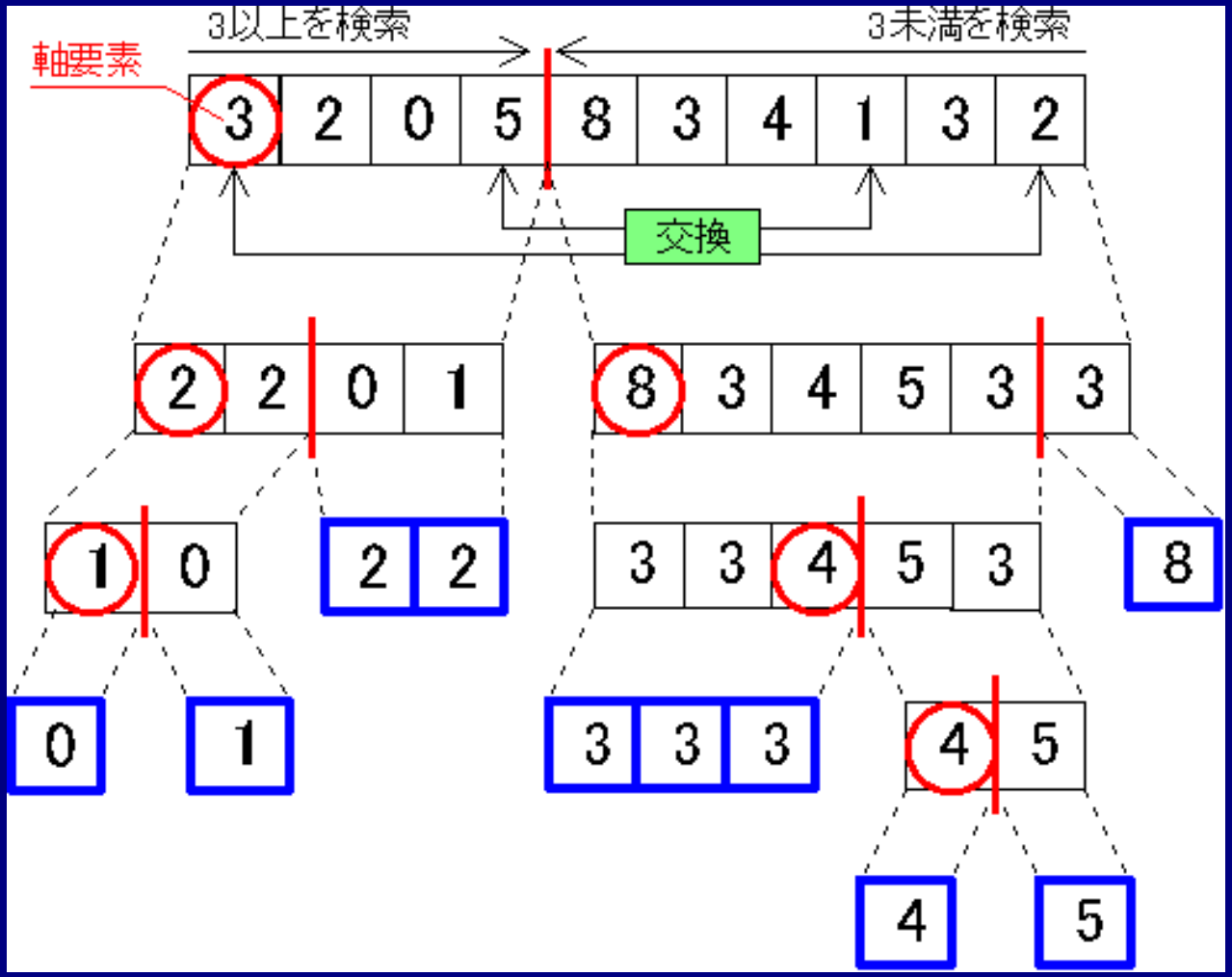
```
qsort[]  =  []
qsort(x:xs)  =  qsort  smaller  ++  [x]  ++  qsort
   larger
where
      smaller  =  [a  |  a  ←  xs,a  <=   x]
      larger     =  [b  |  b   ←  xs,b  <=   x]
```

アルゴリズム：

1.適当な数（ピボットという）を選択する
　（この場合はデータの総数の中央値が望ましい）
2.ピボットより小さい数を前方、大きい数を後方に移動させる（分割）
3.二分割された各々のデータを、それぞれソートする

5.Haskellの妙味

クイックソート



5.Haskellの妙味

```
qsort [3,5,1,4,2]
=   qsort [1,2] ++ [3] ++ qsort[5,4]
=   (qsort[] ++[1] ++ qsort[2]) ++ [3]  ++
      (qsort[4] ++ [5] ++ qsort[])
=   ([] ++ [1] ++ [2] ++ [3] ++ ([4] ++
  [5] ++ [])
=   [1,2] ++ [3] ++ [4,5]
=   [1,2,3,4,5]
```

発想の転換

発想の転換(1)

- 数え上げることが本質なら、単にリストを作る

```
for ($i=0; $i<10; $i++) { print "$i\n"; }
→ mapM_ print [0..9]
```

- メモリーの節約が本質なら、気にしない

```
$i = 0;
while (<>) { $i++ }
print "$i\n"
```

```
→ print . length . lines =<< getContents
```

発想の転換(2)

- 繰り返しが本質なら再帰で書く

```
for ($i = 1; $i <= $n; $i++) {
  $ret = $ret * $i;
}
→
```

  - 単純な再帰版

```
fact n = if n == 1
         then 1
         else n * fact (n-1)
```

  - パターンマッチ版

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

6.この章の参考文献

- Haskell Report

  http://www.haskell.org
- Programming in Haskell 解答やスライド等
  http://www.cs.nott.ac.uk/~gmh/book.html
- Haskell プログラミング

  http://www.mew.org/~kazu/material/2008-haskell.pdf
- 本物のプログラマはHaskellを使う
  http://itpro.nikkeibp.co.jp/article/COLUMN/20060915/248215/?ST=develop
- なぜ関数プログラミングは重要か

  http://www.sampou.org/haskell/article/whyfp.html

7.練習問題

1. *double(double 2)*の結果を算出する他の計算方法を考えよ。
2. *x* の値にかかわらず *sum[x] = x* であることを示せ。
3. 数値のリストに対し要素の積を計算する関数 *product*を定義せよ。そして、その定義を使って、*product [2,3,4] = 24* となることを示せ。
4. リストの逆順に整列するように関数 *qsort* の定義を変えるにはどうすればよいか？
5. *qsort* の定義で、≦ を＜ に置き換えるとどのような影響があるか？ ヒント:例として *[2,2,3,1,1]*を考えてみよ。