Adapted from Graham's lecture slides.

第一章：概论

函数，函数式程序设计，

历史回顾，Haskell的特点和例子

课程内容概览

北京大學
PEKING UNIVERSITY

函数

- In Haskell, a function is a mapping that takes one or more arguments and produces a single result.

```
double x = x + x
```

- Computation by function application

```
   double 3
=   { applying double }
   3 + 3
=   { applying + }
  6
```

- Computation by function application

```
   double (double 2)
=     { applying the inner double }
   double (2 + 2)
=     { applying + }
    double 4
=     { applying double }
    4 + 4
=     { applying + }
    8
```

- Computation by function application

```
   double (double 2)
=     { applying the outer double }
   double 2 + double 2
=     { applying the first double }
   (2 + 2) + double 2
=     { applying the first + }
   4 + double 2
=     { applying double }
   4 + (2 + 2)
=     { applying the second + }
   4 + 4
=     { applying + }
   8
```

函数式程序设计

- Functional programming is <u>style</u> of programming in which the basic method of computation is the application of functions to arguments;

- A functional language is one that <u>supports</u> and <u>encourages</u> the functional style.

例子

Summing the integers 1 to 10 in Java:

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

The computation method is <u>variable assignment</u>.

Summing the integers 1 to 10 in Haskell:

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

sum [1..10]
```

The computation method is <u>function application</u>.

历史回顾

1930s:

Alonzo Church develops the lambda calculus,
a simple but powerful theory of functions.

历史回顾

1950s:

John McCarthy develops Lisp, the first functional language, with some influences from the lambda calculus, but retaining variable assignments.

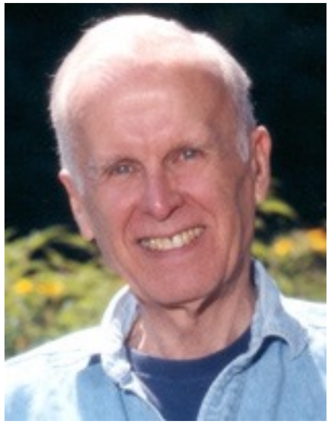
历史回顾

1960s:

Peter Landin develops ISWIM, the first *pure* functional language, based strongly on the lambda calculus, with no assignments.

历史回顾

1970s:



John Backus develops FP, a functional language that emphasizes *higher-order functions* and *reasoning about programs*.

历史回顾

1970s:

Robin Milner and others develop ML, the first modern functional language, which introduced *type inference* and *polymorphic types*.

历史回顾

1970s - 1980s:

David Turner develops a number of *lazy* functional languages, culminating in the Miranda system.

历史回顾

1987:

Haskell

An advanced, purely functional programming language

An international committee starts the development of Haskell, a standard lazy functional language.

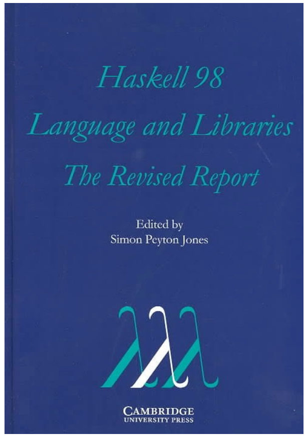
历史回顾

1990s:

Phil Wadler and others develop *type classes* and *monads*, two of the main innovations of Haskell.

历史回顾

2003:



The committee publishes the Haskell Report, defining a stable version of the language; an updated version was published in 2010.

历史回顾

2010-date:

Standard distribution, library support, new language features, development tools, use in industry, influence on other languages, etc.

Haskell的特点

- 简洁（声明式）：第2章，第4章
- 强有力的类型系统：第3章，第8章
- List comprehensions: 第5章
- 递归函数：第6章
- 高阶函数：第7章
- 表达副作用的函数：第10章，第12章
- Generic函数：第12章，第14章
- 惰性计算：第15章
- 程序推理：第16章，第17章

例1: 序列求和

```
sum []       = 0
sum (n:ns)   = n + sum ns
```

```
  sum [1,2,3]
=    { applying sum }”
  1 + sum [2,3]
=    { applying sum }
  1 + (2 + sum [3])
=    { applying sum }
  1 + (2 + (3 + sum []))
=    { applying sum }
  1 + (2 + (3 + 0))
=    { applying + }
  6
```

例2: 快速排序

```
qsort []     = []
qsort (x:xs) = qsort ys ++ [x] ++ qort zs
           where
              ys = [a | a ← xs, a ≤ x]
              zs = [b | b ← xs, b > x]
```

例3: 序列求"和"一般化: 高阶函数

```
sum []        = 0
sum (n:ns)    = n + sum ns
```

```
foldr plus zero []        = zero
foldr plus zero (n:ns)    = plus n (foldr plus zero ns)

sum = foldr (+) 0
```

练习：用foldr定义（1）计算数字列表的乘积；（2）计算列表的长度。

例4: 生成无限序列

```
cyclic = let x = 0 : y
             y = 1 : x
         in x

ns = 0 : foldr f 0 ns
     where  f n r = (1+n) : r
```

课堂内容一瞥：A Running Problem

如何构造出一个正确而且高效的解决最大子段和问题？

Maximum Segment Sum Problem

Given a list of numbers, find the maximum of sums of all *consecutive* sublists.

- $[-1, 3, 3, -4, -1, 4, 2, -1] \implies 7$
- $[-1, 3, 1, -4, -1, 4, 2, -1] \implies 6$
- $[-1, 3, 1, -4, -1, 1, 2, -1] \implies 4$

⇩

```
mss=0; s=0;
 for(i=0;i<n;i++){
      s += x[i];
      if(s<0) s=0;
      if(mss<s) mss= s;
 }
```

课堂内容一瞥：A Running Problem

- 课程第一部分（Haskell）：利用高阶函数等的组合描述要做什么

```
mss = maximum . map sum . segs
  where
    maximum = foldr max (-inf)
    sum = foldr (+) 0
    segs = concat . map tails . inits
    concat = ...
    tails = ...
    inits = ...
```

课堂内容一瞥：A Running Problem

- 课程第二部分（Agda)：构建序列上的演算理论

```
map : ∀ {A B : Set} → (A → B) → List A → List B
map f [] = []
map f (x ∷ xs) = f x ∷ map f xs
```

```
map-compose-p : ∀ {A B C : Set} (f : A → B) (g : B → C) (x : List A) →
                (map g ∘ map f) x ≡ map (g ∘ f) x
map-compose-p f g [] =
  begin
    (map g ∘ map f) []
  ≡⟨⟩
    map g (map f [])
  ≡⟨⟩
    map g []
  ≡⟨⟩
    []
  ≡⟨⟩
    map (g ∘ f) []
  ∎
```

```
map-compose-p f g (x ∷ xs) =
  begin
    (map g ∘ map f) (x ∷ xs)
  ≡⟨⟩
    map g (map f (x ∷ xs))
  ≡⟨⟩
    map g (f x ∷ map f xs)
  ≡⟨⟩
    g (f x) ∷ map g (map f xs)
  ≡⟨ cong (g (f x) ∷_) (map-compose-p f g xs) ⟩
    g (f x) ∷ map (g ∘ f) xs
  ≡⟨⟩
    (g ∘ f) x ∷ map (g ∘ f) xs
  ≡⟨⟩
    map (g ∘ f) (x ∷ xs)
  ∎
```

课堂内容一瞥：A Running Problem

- 课程第二部分（Agda)：构建序列上的演算理论

```
map-promotion :
    ∀{A B : Set} (f : A → B)
    → map f ∘ flatten ≡ flatten ∘ map (map f)

reduce-promotion :
    ∀{A : Set} (_⊕_ : A → A → A) (e : A)
           (p : IsMonoid _⊕_ e)
    → reduce _⊕_ e p ∘ flatten
      ≡ reduce _⊕_ e p ∘ map (reduce _⊕_ e p)
```

课堂内容一瞥：A Running Problem

- 课程第二部分（Agda)：程序推理/程序演算（+记号设计）

$$
\begin{array}{ll}
& mss \\
= & \quad \{ \text{ definition of } mss \ \} \\
& \uparrow / \cdot + / * \cdot segs \\
= & \quad \{ \text{ definition of } segs \ \} \\
& \uparrow / \cdot + / * \cdot +\!\!+ / \cdot tails * \cdot inits \\
= & \quad \{ \text{ map and reduce promotion } \} \\
& \uparrow / \cdot (\uparrow / \cdot + / * \cdot tails) * \cdot inits \\
= & \quad \{ \text{ Horner's rule with } a \odot b = (a + b) \uparrow 0 \ \} \\
& \uparrow / \cdot \odot \not\to_0 * \cdot inits \\
= & \quad \{ \text{ accumulation lemma } \} \\
& \boxed{\uparrow / \cdot \odot /\!\!\!/\!\!\!\to_0}
\end{array}
$$

作业：

【1-1】Define a function *product* that produces the product of a list of numbers, and show using your definition that *product* [2,3,4] = 24.

【1-2】Define list length function using foldr.

【1-3】How should the definition of the function qsort be modified so that it produces a reverse sorted version of a list?