CSCE 314
Programming Languages

Haskell 101

Dr. Hyunyoung Lee

Contents

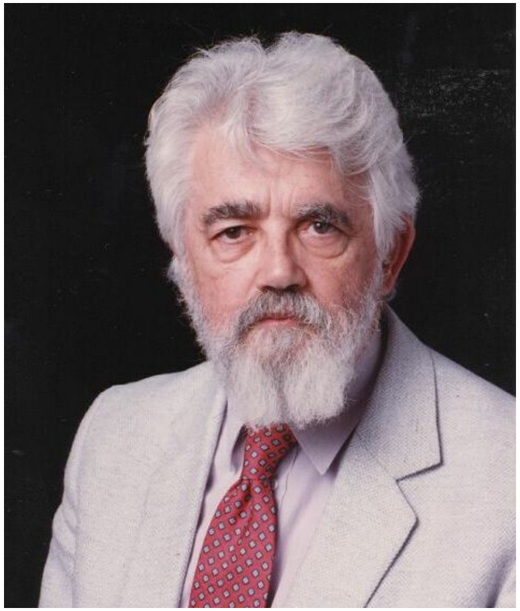
Historical Background (1/8)

1930s:

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

Historical Background (2/8)

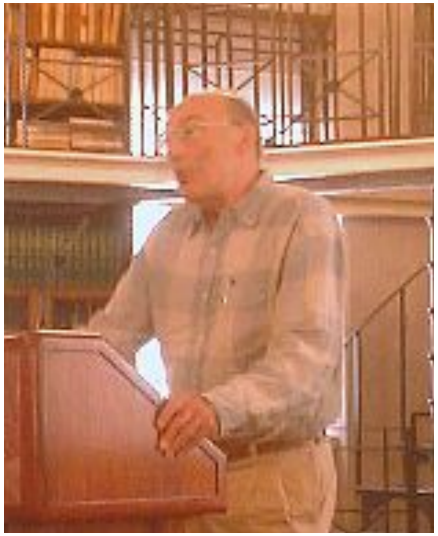
1950s:



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

Historical Background (3/8)

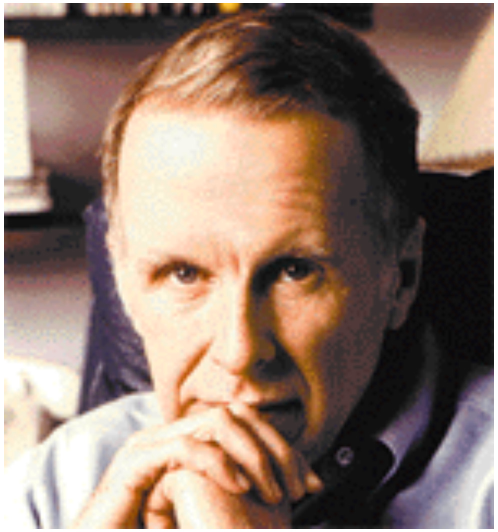
1960s:



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

Historical Background (4/8)

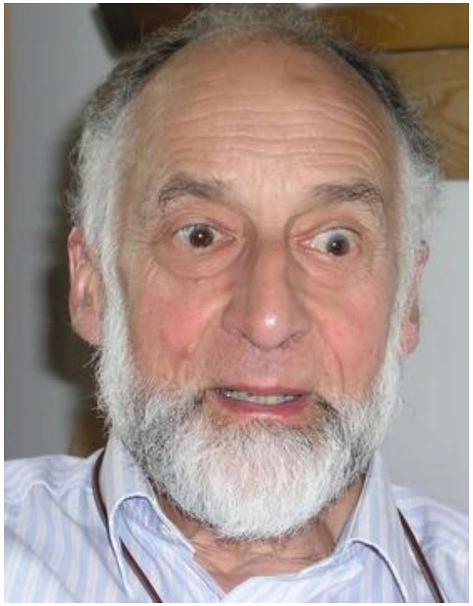
1970s:



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

Historical Background (5/8)

1970s:



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

Historical Background (6/8)

1970s - 1980s:

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

Historical Background (7/8)

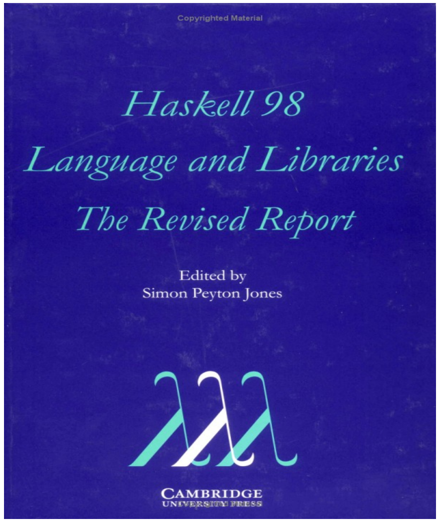
1987:

Haskell
A Purely Functional Language

An international committee of researchers initiates the development of Haskell, a standard lazy pure functional language

Historical Background (8/8)

2003:



The committee publishes the Haskell 98 report, defining a stable version of the language

Since then highly influential in language research and fairly widely used in commercial software. For example, Facebook's anti-spam programs, and Cardano, a cryptocurrency introduced in Sep. 2017, are written in Haskell.

Haskell is a
Lazy
Pure
Functional Language

"Haskell is a Lazy Pure Functional Language"

Lazy programming language only evaluates arguments when strictly necessary, thus,
(1) avoiding unnecessary computation and
(2) ensuring that programs terminate whenever possible. For example, given the definitions

```
omit x = 0
keep_going x = keep_going (x+1)
```

what is the result of the following expression?

```
omit (keep_going 1)
```

"Haskell is a Lazy Pure Functional Language"

Pure functional language, as with mathematical functions, prohibits side effects (or at least they are confined):

- Immutable data: Instead of altering existing values, altered copies are created and the original is preserved, thus, there's no destructive assignment:

```
a = 1; a = 2;    -- illegal
```

- Referential transparency: Expressions yield the same value each time they are invoked; helps reasoning. Such expression can be replaced with its value without changing the behavior of a program, for example,

```
y = f x  and  g = h y y
```

then, replacing the definition of g with `g = h (f x) (f x)`

will get the same result (value).

"Haskell is a Lazy Pure Functional Language"

Functional language supports the functional programming style where the basic method of computation is application of functions to arguments. For example, in C,

```
int s = 0;
for (int i=1; i <= 100; ++i)  s = s + i;
```

the computation method is variable assignment

In Haskell,

```
sum [1..100]
```

the computation method is function application

Features of Functional Languages

- Higher-order functions are functions that take other functions as their arguments. E.g.,

  > map reverse ["abc","def"]

  ["cba","fed"]

- Purity – prohibits side effects

(Expressions may result in some actions in addition to return values, such as changing state and I/O; these actions are called side effects)

- Recursion – the canonical way to iterate in functional languages

A Taste of Haskell

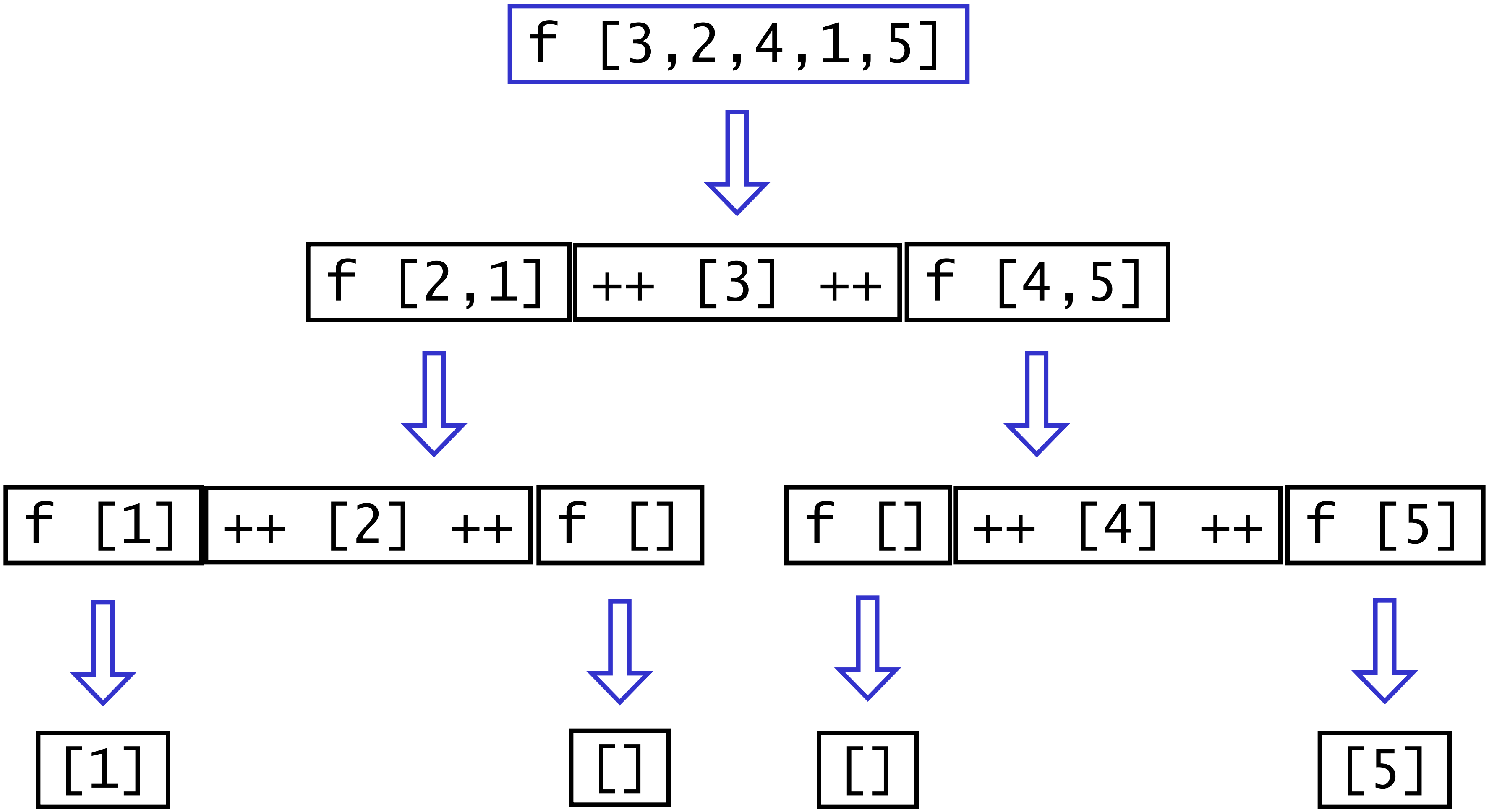
```
f []     = []
f (x:xs) = f ys ++ [x] ++ f zs
           where
             ys = [a | a <- xs, a <= x]
             zs = [b | b <- xs, b > x]
```

?

In C++

```
void f(int xs[], int first, int last)
{ int mid;
  if (first < last)
  { mid = partition(xs, first, last);
    f(xs, first, mid);
    f(xs, mid+1, last);
  }
  return;
}
int partition(int xs[], int first, int last)
{ int k = xs[first];
  int i = first-1;
  int j = last+1;
  int temp;
  do {
    do { j--; } while (k<xs[j]);
    do { i++; } while (k>xs[i]);
    if (i<j) { temp=xs[i]; xs[i]=xs[j]; xs[j]=temp; }
  } while (i<j);
  return j;
}
```

Recursive function execution:



Other Characteristics of Haskell

- Statically typed
- Type inference
- Rich type system
- Succinct, expressive syntax yields short programs
- Indentation matters
- Capitalization of names matters

Using GHC and GHCi

- From a shell window, the compiler is invoked as

  `> ghc myfile.hs`

  `> ghci` (or as `> ghc --interactive`)

- For multi-file programs, use `--make` option
- GHCi operates on an eval-print-loop:

```
> sqrt (3^2 + 4^2)
5.0
>
```

User types in a Haskell expression

The interpreter evaluates it and prints out the result

Waits for the next expression

- Efficient edit-compile-run cycle, e.g., using Emacs with haskell-mode (https://github.com/serras/emacs-haskell-tutorial/blob/master/tutorial.md) helps indenting, debugging, jumping to an error, etc.

Using GHCi

- Useful basic GHCi commands:

`:?`	Help! Show all commands
`:load test`	Open file `test.hs` or `test.lhs`
`:reload`	Reload the previously loaded file
`:main a1 a2`	Invoke `main` with command line args `a1 a2`
`:!`	Execute a shell command
`:edit` *name*	Edit script *name*
`:edit`	Edit current script
`:type` *expr*	Show type of *expr*
`:quit`	Quit GHCi

- Commands can be abbreviated. E.g., :r is :reload
- At startup, the definitions of the "Standard Prelude" are loaded

The Standard Prelude

Haskell comes with a large number of standard library functions. In addition to the familiar numeric functions such as + and *, the library also provides many useful functions on lists.

-- Select the first element of a list:

```
> head [1,2,3,4,5]
1
```

-- Remove the first element from a list:

```
> tail [1,2,3,4,5]
[2,3,4,5]
```

-- Select the nth element of a list:

```
> [1,2,3,4,5] !! 2
3
```

-- Select the first n elements of a list:

```
> take 3 [1,2,3,4,5]
[1,2,3]
```

-- Remove the first n elements from a list:

```
> drop 3 [1,2,3,4,5]
[4,5]
```

-- Append two lists:

```
> [1,2,3] ++ [4,5]
[1,2,3,4,5]
```

-- Reverse a list:

```
> reverse [1,2,3,4,5]
[5,4,3,2,1]
```

-- Calculate the length of a list:

```
> length [1,2,3,4,5]
5
```

-- Calculate the sum of a list of numbers:

```
> sum [1,2,3,4,5]
15
```

-- Calculate the product of a list of numbers:

```
> product [1,2,3,4,5]
120
```

Functions (1)

- Function and parameter names must start with a lower case letter, e.g., myFun1, arg_x, personName, etc.

  (By convention, list arguments usually have an s suffix on their name, e.g., xs, ns, nss, etc.)
- Functions are defined as equations:

```
square x = x * x            add x y = x + y
```

- Once defined, apply the function to arguments:

```
> square 7                  > add 2 3
49                          5
```

  In C, these calls would be `square(7);` and `add(2,3);`
- Parentheses are often needed in Haskell too

```
> add (square 2) (add 2 3)
9
```

Functions (2)

- Function application has the highest precedence

  `square 2 + 3` means `(square 2) + 3` not `square (2+3)`

- Function call associates to the left and is by pattern matching (first one to match is used)
- Function application operator $ has the lowest precedence and is used to rid of parentheses

  `sum ([1..5] ++ [6..10]) -> sum $ [1..5] ++ [6..10]`

- Combinations of most symbols are allowed as operator

`x @#$%^&*-+@#$% y = "What on earth?"` ☺

Another (more reasonable) example:

```
x +/- y = (x+y, x-y)
> 10 +/- 1
(11,9)
```

Function Application

In mathematics, function application is denoted using parentheses, and multiplication is often denoted using juxtaposition or space

```
f(a,b) + c d
```

Apply the function f to a and b, and add the result to the product of c and d

In Haskell, function application is denoted using space, and multiplication is denoted using *

```
f a b + c*d
```

As previously, but in Haskell syntax

Examples

Mathematics	Haskell
`f(x)`	`f x`
`f(x,y)`	`f x y`
`f(g(x))`	`f (g x)`
`f(x,g(y))`	`f x (g y)`
`f(x)g(y)`	`f x * g y`

Evaluating Functions (1)

Think of evaluating functions as substitution and reduction

```
add x y = x + y;  square x = x * x
```

```
add (square 2) (add 2 3)
-- apply square
add (2 * 2) (add 2 3)
-- apply *
add 4 (add 2 3)
-- apply inner add
add 4 (2 + 3)
-- apply +
add 4 5
-- apply add
4+5
-- apply +
9
```

Evaluating Functions (2)

- There are many possible orders to evaluate a function

```
head (1:(reverse [2,3,4,5]))
-- apply reverse
-- ... many steps omitted here
head ([1,5,4,3,2])
-- apply head
1
```

```
head (1:(reverse [2,3,4,5]))
-- apply head
1
```

- In a pure functional language, evaluation order does not affect the *value* of the computation
- It can, however, affect the *amount* of computation and whether the computation *terminates* or not (or fails with a run-time error)
- Haskell evaluates a function's argument lazily

  "Call-by-need" - only apply a function if its value is needed, and "memoize" what's already been evaluated

Haskell Scripts

A Haskell program consists of one or more scripts

A script is a text file comprising a sequence of definitions, where new functions are defined

By convention, Haskell scripts usually have a .hs suffix on their filename. This is not mandatory, but is useful for identification purposes.

Loading new script causes new definitions to be in scope:

```
Prelude> :l test.hs
[1 of 1] Compiling Main      ( test.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

My First Script

When developing a Haskell script, it is useful to keep two windows open, one running an editor for the script, and the other running GHCi:

Start an editor, type in the following two function definitions, and save the script as test.hs:

```
double x    = x + x

quadruple x = double (double x)
```

In another window start up GHCi with the new script:

```
% ghci test.hs
```

Now both the standard library and the file test.hs are loaded, and functions from both can be used:

```
> quadruple 10
40
> take (double 2) [1,2,3,4,5,6]
[1,2,3,4]
```

Leaving GHCi open, return to the editor, add the following definitions, and resave:

```
factorial n = product [1..n]
average ns  = sum ns `div` length ns
```

Note:

- div is enclosed in <u>back</u> quotes, not forward
- x `f` y is syntactic sugar for f x y
- Any function with two or more arg.s can be used as an infix operator (enclosed in back quotes)
- Any infix operator can be used as a function (enclosed in parentheses), e.g., (+) 10 20

GHCi does not automatically detect that the script has been changed, so a <u>reload</u> command must be executed before the new definitions can be used:

```
>:r
 ( test.hs, interpreted )
> factorial 10
3628800
> average [1,2,3,4,5]
3
```

The Layout Rule

- Layout of a script determines the structure of definitions
- Commonly use layouts instead of braces and semicolons (which are still allowed and can be mixed with layout)
- Each definition must begin in precisely the same column:

```
a = 10
b = 20
c = 30
```

✓

```
a = 10
  b = 20
c = 30
```

✗

```
  a = 10
b = 20
  c = 30
```

✗

implicit grouping by layout

```
a = b + c
      where
         b = 1
         c = 2
d = a * 2
```

means

```
a = b + c
      where
        {b = 1;
         c = 2}
d = a * 2
```

explicit grouping by braces and semicolons

Exercises

(1) Try out the codes in slides 15-24 using GHCi.

(2) Fix the syntax errors in the program below, and test your solution using GHCi.

```
N = a 'div' length xs
    where
       a = 10
      xs = [1,2,3,4,5]
```

```
n = a `div` length xs
    where
      a = 10
      xs = [1,2,3,4,5]
```

(3) Show how the library function <u>last</u> that selects the last element of a list can be defined using the functions introduced in this lecture.

```
last xs = head ( reverse xs )
```

(4) Can you think of another possible definition?

```
last xs = xs !! (length xs - 1)
```

(5) Similarly, show how the library function <u>init</u> that removes the last element from a list can be defined in two different ways.

```
init xs = take ( length xs - 1 ) xs

init xs = reverse ( tail ( reverse xs ))
```