本次直播是

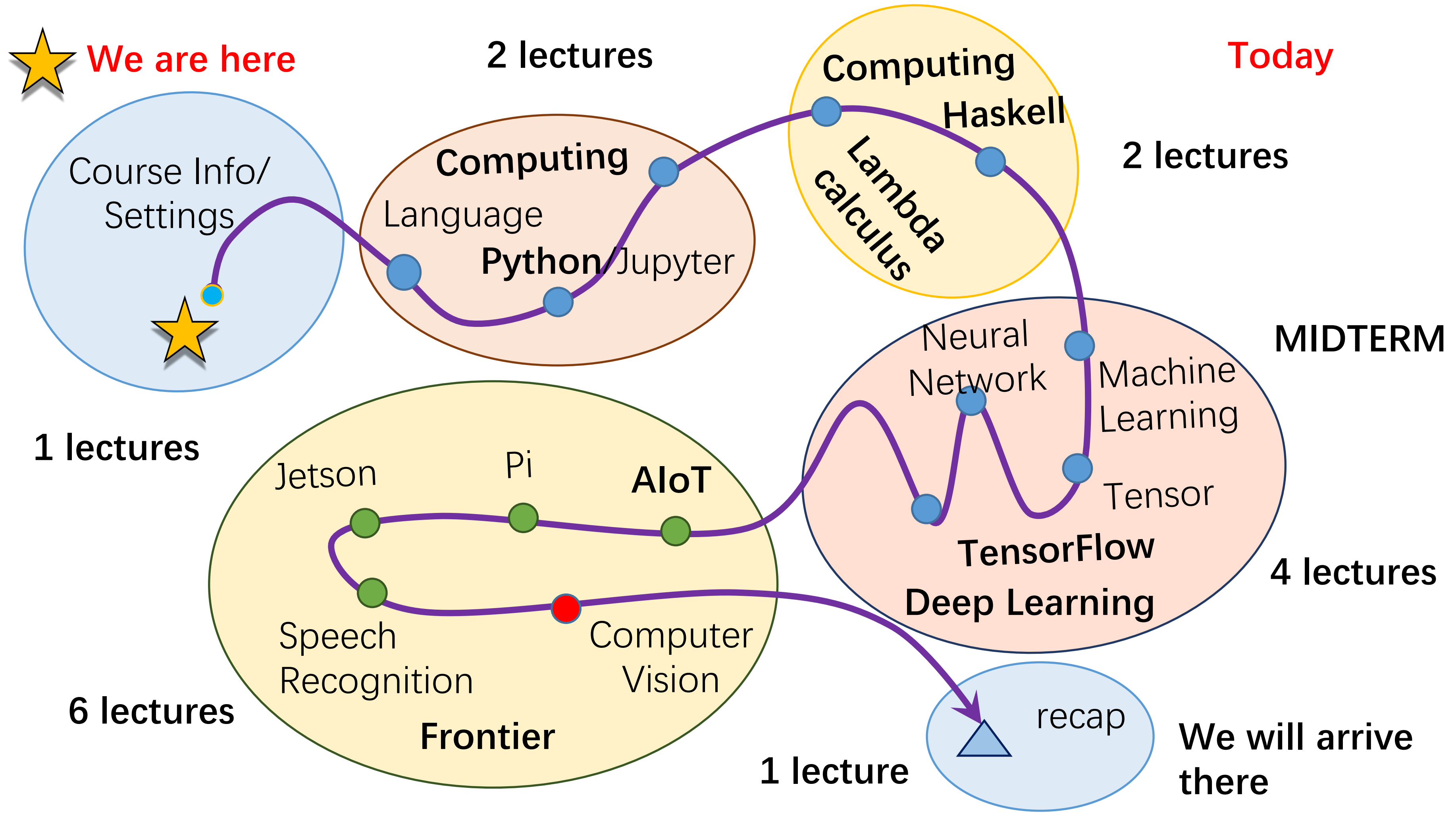
视频直播

请点击页面上方的

“直播中 点击观看”

打开直播窗口

课程路线图（Roadmap）



STIS-课程编号-01510202

计算基础

λ演算(lambda calculus)

智能系统实验室

清华大学基础工业训练中心

投票(匿名) 最多可选2项

设置

你觉得当前的视频直播的观看质量怎么样？

A PPT不清晰

B 声音不流畅

C 视频不流畅

D 都挺好的。

提交

计算

- 计算机（Computer），完成计算的机械（mechanics）；
- 计算（Compute），能用lambda演算表达的，即是可以被计算；（Church）（本节课）
- 可计算的函数可以转换为布尔电路(Boolean circuit)，从而可以用电子电路实现完成（Shannon）（下节课）

[x] Lewis H R, Papadimitriou C H., Elements of the Theory of Computation[M]. Prentice Hall PTR, 1997.

背景

- λ演算由阿隆索*邱奇(Alonzo Church)提出， 他是阿兰*图灵(Alan Turing)的导师

- 函数式编程语言实现了lambda演算。

- 如Haskell, Scala, Python, …

https://en.wikipedia.org/wiki/Alonzo_Church

λ演算-概要Brief

- λ x y . x + y
  - Curried function: λ x . (λ y . x + y)
- λ x . if (= x 0) then 1 else x^2
- α conversion to replace x with y
  - λ y . if (= y 0) then 1 else y^2
- β reduction is to apply a function in λ calculus
  - (λ x . x + 1) 3 is 3+1
- η （“eta”）conversion says that in any λ expression, replace the value f with the value g
  - x, f x = g x

λ演算的特点

- λ演算基本上是一个基于表达式的简单的编程语言
- λ演算是简单的
  - λ演算的表达式仅包含三个组件：定义，标识符引用和应用
  - 三个转换规则，alpha，beta，eta
- λ演算易于读写
  - 写λ演算，几乎就是编程
- λ演算是可扩展
  - 构造变体去探索各种不同结构化计算的性质或者语义

目录

- 1. λ-表达式（λ-Term）
- 2. 归约方式（Reduction）
- 3. 组合子(Combinator)
- 4. 应用案例- 逻辑/数码/递归/图灵完备性
- 5. 简单类型的λ演算
- 6. Haskell语言

1. λ-表达式（λ-term）-合法的三种表达式

λ-表达式有三种合法表达式：

a). 变量：所有的变量都是合法的λ-term，例如 **x**；

b). 抽象：形式上表述为**λx.M**，表示获取一个参数x并返回M的函数，

例如 λx. y;

c). 应用：形式上表述为 **M N**，表示将函数M应用于参数N，

例如 (λx.x) y就是将λx.x应用于y,得到结果y;

Plus. 三个以上term并列时，左结合优先。

详见lambda calculus, https://plato.stanford.edu/entries/lambda-calculus/

1. λ-表达式（λ-term）-2个 符号习惯

- Two notational conventions:
  1. Applications associate to the left (like in Haskell):
     “y z x”　　is　　“(y z) x”
  2. The body of a lambda extends as far as possible to the right:
     “\x.x \z.x z x”　　is　　“\x.(x \z.(x z x))”

1. λ-表达式（λ-term）-绑定变量与自由变量

λ-表达式中的变量可以分为两种类型：

a). 绑定变量：在抽象定义中的形参即为绑定变量；

b). 自由变量：除去绑定变量以外的变量。

e.g. λ x. x y中x为绑定变量，y为自由变量

我们用FV()函数来表示自由变量，BV()来表示绑定变量

1. λ-表达式（λ-term）-绑定变量与自由变量

	Free	Bound
(1)	$\mathbf{FV}(x) = \{x\}$	$\mathbf{BV}(x) = \{\varnothing\}$
(2)	$\mathbf{FV}(MN) = \mathbf{FV}(M) \cup \mathbf{FV}(N)$	$\mathbf{BV}(MN) = \mathbf{BV}(M) \cup \mathbf{BV}(N)$
(3)	$\mathbf{FV}(\lambda x[M]) = \mathbf{FV}(M) - \{x\}$	$\mathbf{BV}(\lambda x[M]) = \mathbf{BV}(M) \cup \{x\}$

1. λ-表达式（λ-term）-多参数与柯里化

柯里化（curring化）（又称局部嵌套）：输入第一个参数后，将函数变为单参数函数，再输入第二个参数。

E.g. $\lambda x\ y.xy = \lambda x.(\lambda y.xy)$，就这样我们把单参数拓展到了多参数，因此上述多参数的定义也是合法的。

2. 归约方式（reduction）-三大规则

a). α-转换： **λx.x= λy.y**，解决命名冲突问题；

b). β-归约： **λx.M N=M[x:=N]**，是归约的最重要规则；

要注意的是M[x:=N]是将M中的所有自由变量x用N替换，

而不是函数自变量的替换，M N才是将绑定变量替换，

因此(λx.M) [x:=N]= λx.M;

c). η-归约： λx.M x=M，是解决函数冗余的方式；

2. 归约方式（reduction）-示例1

三大规则

$\lambda x.(\lambda y.(x(y-5)))\ 2y = \lambda x.(\lambda z.(x(z-5)))\ 2y = \lambda z.(2y\ (z-5))$ √

$\lambda x.(\lambda y.(x(y-5)))\ 2y = \lambda y.(2y\ (y-5))$ ×

2. 归约方式（reduction）-示例2

(\x. x x) (\y. y)

--> x x [\y. y / x]

== (\y. y) (\y. y)

--> y [\y. y / y]

== \y. y

2. 归约方式（reduction）-示例3

(\x. x x) (\x. x x)

--> x x [\x. x x/x]

== (\x. x x) (\x. x x)

Non terminating computations in the lambda calculus

2. 归约方式（reduction） -示例4

$(\lambda x . x + 1)\ 3 = 4$

$\lambda y . (\lambda x . x + y))\ q = \lambda x .\ x + q$

$(\lambda x\, y.\ x\, y)\ (\lambda z . z * z)\ 3 = (\lambda z . z * z)\ 3 = 3*3 = 9.$

$\lambda z . (\lambda x . x + z))\ (x + 2) = \lambda z . (\lambda y . y + z))\ (x + 2)$ （α）

$= \lambda y . y + x + 2$ （β 归约）.

2. 归约方式（reduction）-次序

- a).应用次序（立即求值）：
  - 首先找到最里面的表达式，并按从右往左的次序进行β-归约；

- b).标准次序（惰性求值）：
  - 从最外面的表达式开始从外往内进行归约。

- 前者为立即求值（eager evaluation），后者为惰性求值（Lazy Evaluation）。

2. 归约方式（reduction）-求值次序示例1

- 立即求值或紧迫求值（eager evaluation或eager execution）.

要计算f(g(x+y), 2*x)，在调用f之前，必先计算g(x+y) 和 2*x

在计算g(x+y)时，必先计算 x+y。

- 惰性求值或（懒惰求值）（ Lazy Evaluation ）

而在惰性求值（懒惰求值）中，只有在需要的时候，才去计算。

要计算f，如果f从来不使用g(x+y)，那么永远不需要计算g

- 程序语言Haskell和Miranda采用懒惰求值

2. 归约方式（reduction）-求值次序示例2

应用次序(eager evaluation)

(λ x y z . + (* x x) y) (+ 3 2) (* 10 2) (/ 24 (* 2 3))

从右往左，β 归约：(* 2 3)=6. (/ 24 6)=4. (* 10 2) =20. (+ 3 2) =5.

(λ x y z . + (* x x) y) 5 20 4

(+ (* 5 5) 20)=45.

2. 归约方式（reduction）-求值次序示例3

标准次序/惰性求值（lazy evaluation)

(λ x y z . + (* x x) y) (+ 3 2) (* 10 2) (/ 24 (* 2 3))

从左往外，β 规约：(+ (* (+ 3 2) (+ 3 2)) (* 10 2).

(+ (* 5 5) 20)=45.

3. 组合子(Combinator)

FV(M)=∅时，称M为组合子（**没有自由变量**），函数抽象。

K $\lambda x. (\lambda y. x) = \lambda x\, y. x$ KM=M

I $\lambda x. x$ IM=M

Y $\lambda f. ((\lambda x. f(x\, x))(\lambda x. f(x\, x)))$ YX=X（YX）

Y组合子非常有用！

4. 应用案例1- 逻辑

A) Church 布尔值

逻辑选择，λ项作为逻辑常量

True λx[λy[x]]=λxy.x

False λx[λy[y]]=λxy.y

IfPthenAelseB:=PAB

如果P能规约到T=K，PAB=KAB=A

如果P能归约到F=KI，PAB=KIAB=B

布尔操作

BoolAND=λx y. x y False

BoolOr=λx y. x True y

BoolNot=λx y. x False True *

？兴趣小作业测试一下BoolAND True False？

Mark C. Chu-Carroll, Good Math: A Geek's Guide to the Beauty of Numbers, Logic, and Computation, Chapter 24-26.

4. 应用案例2- 数码

B) Church数码(Church numerals)

0= λ s z.z

1= λ s z.s z

2= λ s z.s(s z)

3= λ s z.s(s(s z))

… …

UnaryZero= λ x. “”
UnarySucc= λ x. append ” 1” x

7= λ s z.s(s(s(s(s(s(s z))))))

7计算为 1111111

add λ s z x y . x s (y s z)

add= λ x y.(λ s z.(x s(y s z)))

add_curry = λ x y. (λ s z . (x s (y s z)))

? 兴趣小作业：Adding 2 + 3 using a curried function！

4 应用案例3-递归

C) 递归的定义

- 递归：见递归

- Y组合子，是一个不动点组合子，能够复制自身；

- YX=X（YX）

4. 图灵完备性

- 图灵完备性主要有以下四个要求：
  - **存储**：变量和函数都可以存储无限的信息；
  - **算术**：church数码
  - **条件执行**：if/then/else的逻辑运算
  - **重复**：由组合子Y，YX=XYX，可以让一个函数首先调用自身，从而达成递归。

如果一个函数可以被任意可能的计算设备计算出来，那么它也一定能被写为λ-演算， 因此 λ-演算是图灵完备的。

Lambda演算是图灵完备的，它是一种通用的计算模型，可用于模拟任何图灵机。

5. 简单类型化的λ演算

- Typed λ-calculus
- x : σ ， x 具有类型 σ

λ x: σ x+3, x 为类型σ， +是一个σ→ σ 的函数

等价写法

λ x. x+3 : σ→ σ

类型化λ演算广泛应用于程序语言的设计。

静态类型语言，如Java，C, Haskell等

6. Haskell语言的Lambda函数

- Haskell 中lambda 函数也被称为匿名函数(anonymous function.)
- Haskell允许编写完全匿名的函数，这样就不必再费力地为函数想名字了。
- 在 Haskell 中，lambda函数以反斜杠符号 \ 为开始，后跟函数的参数（可以包含模式），而函数体定义在 -> 符号之后。
  - \ 符号读作 lambda。
- lambda函数是从 lambda 演算而来的。
- >\ x -> x * x

$$\lambda x \to x^2$$

square :: Integer -> Integer

square x = x * x

lambda calculus in Haskell

- the identity function:
  - \x.x
- applications associate to the left (like in Haskell):
  - "y z x" is "(y z) x"
- the body of a lambda extends as far as possible to the right:
  - "\x.x \z.x z x" is "\x.(x \z.(x z x))"

Lambda in Python

- lambda arguments : expression
- E.g.
    - sum = lambda x, y : x + y
    - print(sum(1,2))

https://github.com/saturn-lab/Smart-Things-TensorFlow-Practice/tree/master/PP1

参考资料

- David Walker, Programming Languages COS 441, Princeton University.
- Benjamin C. Pierce, Type and Programming languages, Chapter 5.
- Boaz Barak, Introduction to Theoretical Computer Science, Harvard CS 121, Chapter 7.
- Mark C. Chu-Carroll, Good Math: A Geek's Guide to the Beauty of Numbers, Logic, and Computation, Chapter 24-26.
- The Lambda Calculus, https://plato.stanford.edu/entries/lambda-calculus/.
- The Litter Schemer

谢谢指正！