CSE216
Programming Abstraction

State University of New York, Korea

Plan

- Lambda calculus History, Syntax and Semantics Overview
- Syntax Detail
- Semantics Detail
- Extending lambda calculus core

Some slides adapted from CMU. Thanks!
https://www.cs.cmu.edu/~venkatg/teaching/15252-sp20/notes/lambda-calculus-slides.pdf

Overview

Alonzo Church's model of computing, **Lambda Calculus.**

Alonzo Church (1903–1995)

What is a computation/ algorithm?

Hilbert's 10th problem (1900):

Given a multivariate polynomial w/ integer coeffs,
e.g. $4x^2y^3 - 2x^4z^5 + x^8$,

"*devise a process according to which it can be determined in a finite number of operations*"
whether it has an integer root.

Gödel (1934):

Discusses some ideas for definitions of what functions/languages are "computable" but isn't confident what's a good definition.

Church (1936):

Invents lambda calculus,
claims it should be the definition.

Meanwhile... a certain British grad student in Princeton, unaware of all these debates...

Turing

Lambda calculus Core

- TERM::= Var //Variables
  | lambda Var. TERM // Definition/Abstraction
  | TERM TERM // Application

  Var ::= x | y | z …

- λ x. λ y. xy

Temparily Introducing functions and constants

- We will start out with a version of the lambda calculus that has constants like 0, 1, 2… and functions such as + * =.
- We also include constants TRUE and FALSE, and logical functions AND, OR, NOT.

Syntax summary

<exp> ::= <constant>	Built-in constants & functions
| <variable>	Variable names, e.g. x, y, z…
| <exp> <exp>	Application
| λ <variable> . <exp>	Lambda Abstractions
| (<exp>)	Parens

Example: λf.λn. IF (= n 0) 1 (* n (f (- n 1)))

Example

A way to write expressions that denote functions.

Example: (λx . + x 1)

Example

A way to write expressions that denote functions.

Example: (λx . + x 1)

λ means here comes a function

Then comes the parameter x

Then comes the .

Then comes the body + x 1

The function is ended by the) (or end of string)

Example

A way to write expressions that denote functions.

Example: (λx . + x 1)

λ means here comes a function

Then comes the parameter x

Then comes the .

Then comes the body + x 1

The function is ended by the) (or end of string)

(λx . + x 1) is the increment function.

(λx . + x 1) 5 ➞ 6

(We'll explain this more thoroughly later.)

Lambda calculus Semantics

- **evaluating** the expression
    - Beta-reduction

“evaluating”

For example: (+ 4 5)

+ is a function. We write functions in prefix form.

Another example: (+ (* 5 6) (* 8 3))

“evaluating”?

For example: (+ 4 5)

+ is a function. We write functions in prefix form.

Another example: (+ (* 5 6) (* 8 3))

Evaluation proceeds by choosing a reducible expression and reducing it. (There may be more than one order.)

(+ (* 5 6) (* 8 3)) → (+ 30 (* 8 3)) → (+ 30 24) → 54

Function application is indicated by juxtaposition: f x

“The function f applied to the argument x”

Function application is indicated by juxtaposition: f x

"The function f applied to the argument x"

What if we want a function of more than one argument?

We could invent a notation f(x,y), but there's an alternative. To express the sum of 3 and 4 we write:

((+ 3) 4)

Function application is indicated by juxtaposition: f x

"The function f applied to the argument x"

What if we want a function of more than one argument?

We could invent a notation f(x,y), but there's an alternative. To express the sum of 3 and 4 we write:

((+ 3) 4)

The expression (+ 3) denotes the function that adds 3 to its argument.

So all functions take one argument. So when we wrote (+ 3 4) this is shorthand for ((+ 3) 4). (The arguments are associated to the left.)

So all functions take one argument. So when we wrote (+ 3 4) this is shorthand for ((+ 3) 4). (The arguments are associated to the left.)

This mechanism is known as currying (in honor of Haskell Curry).

So all functions take one argument. So when we wrote (+ 3 4) this is shorthand for ((+ 3) 4). (The arguments are associated to the left.)

This mechanism is known as currying (in honor of Haskell Curry).

Parentheses can be added for clarity. E g:

((f ((+ 4) 3)) (g x)) is identical to f (+ 4 3) (g x)

Exercise 1

Lambda calculus is the base of any and all functional languages. A major aspect of any and all functional languages is the use of recursion. Some problems in computer science are primarily, or can only be thought of in recursive terms, such as traversing a tree. In other cases, you can take an iterative problem and turn it into a recursive problem. Think of the idea of multiplication between positive numbers as repeatedly adding a number to itself n times, e.g.

```
int mult(x, y) {
        int temp = x;
        for (int i = 1; i<y; i++) {
            temp+=x;
        }
        return temp;
}
```

Now try to do this recursively. Feel free to add as many helper functions as you want, but you must only use addition and subtraction to get your goal. No multiplication must be used, and remember: no need to worry about negative numbers.

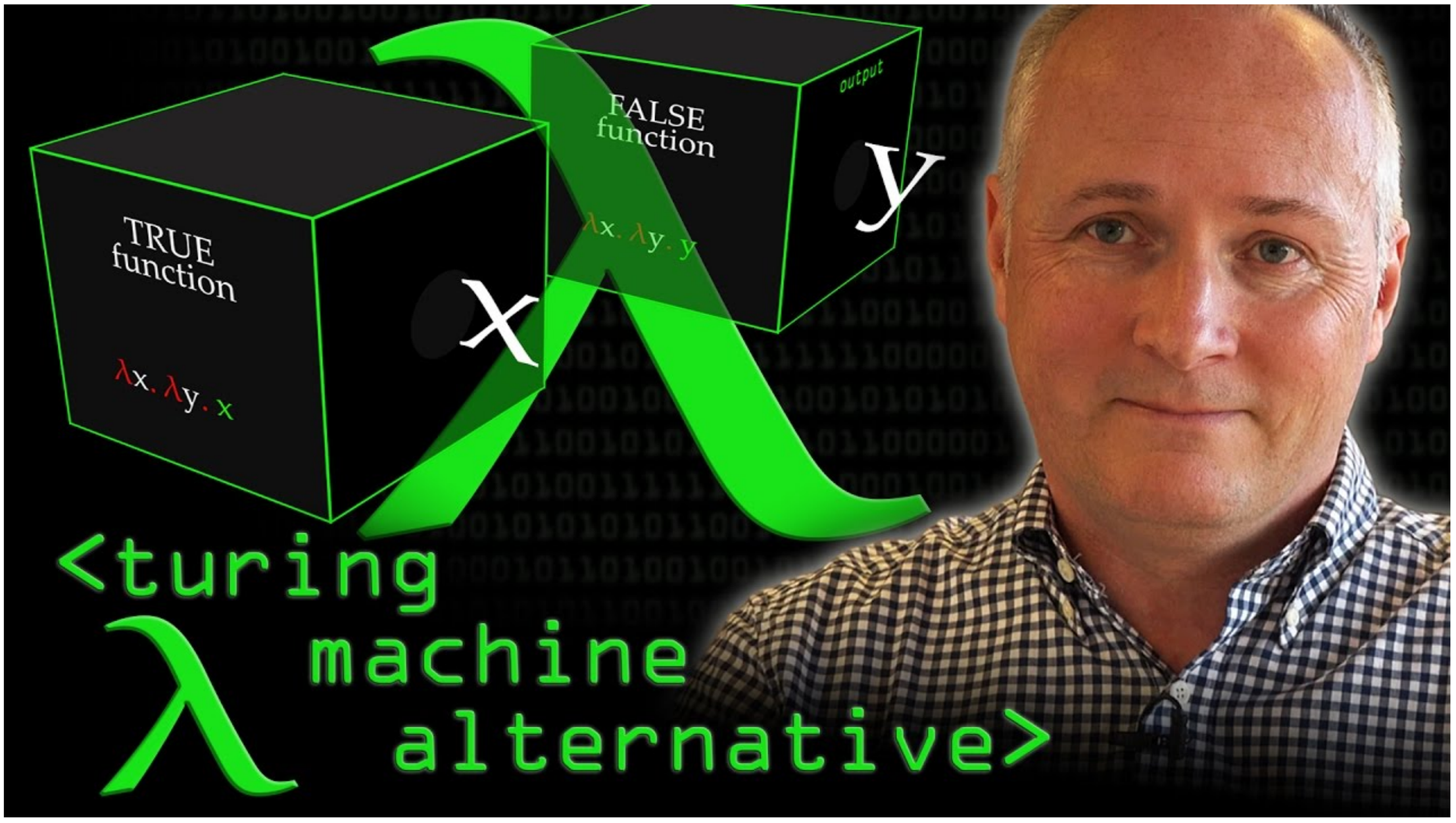
Exercise 2

- Evaluate the following lambda expressions until they are irreducible. (we have not learned this in detail)

- (λ x.λy.x) x y

- (λx.xx) (λx.xx)

Summary (first 7 min)



https://youtu.be/eis11j_iGMs

Details on Syntax

Lambda calculus syntax review

- TERM::= Var //Variables

  | lambda Var. TERM // Definition/Abstraction

  | TERM TERM // Application

Var ::= x | y | z …

Exercise: lambda terms?

- λ x. λ y. x y
- λ x. λ y. y
- λ x. λ y. λ z
- (λ x. x) (λ y. y)
- λ x. x λ y

Solution

- T
- T
- F
- T
- F

Implied parentheses

- Application is left associative
  - x y z same as (x y) z
- Definition is right associative
  - λ x. y z same as λ x. (y z)
- λ x. λ y. x y λ z. z y same as ?

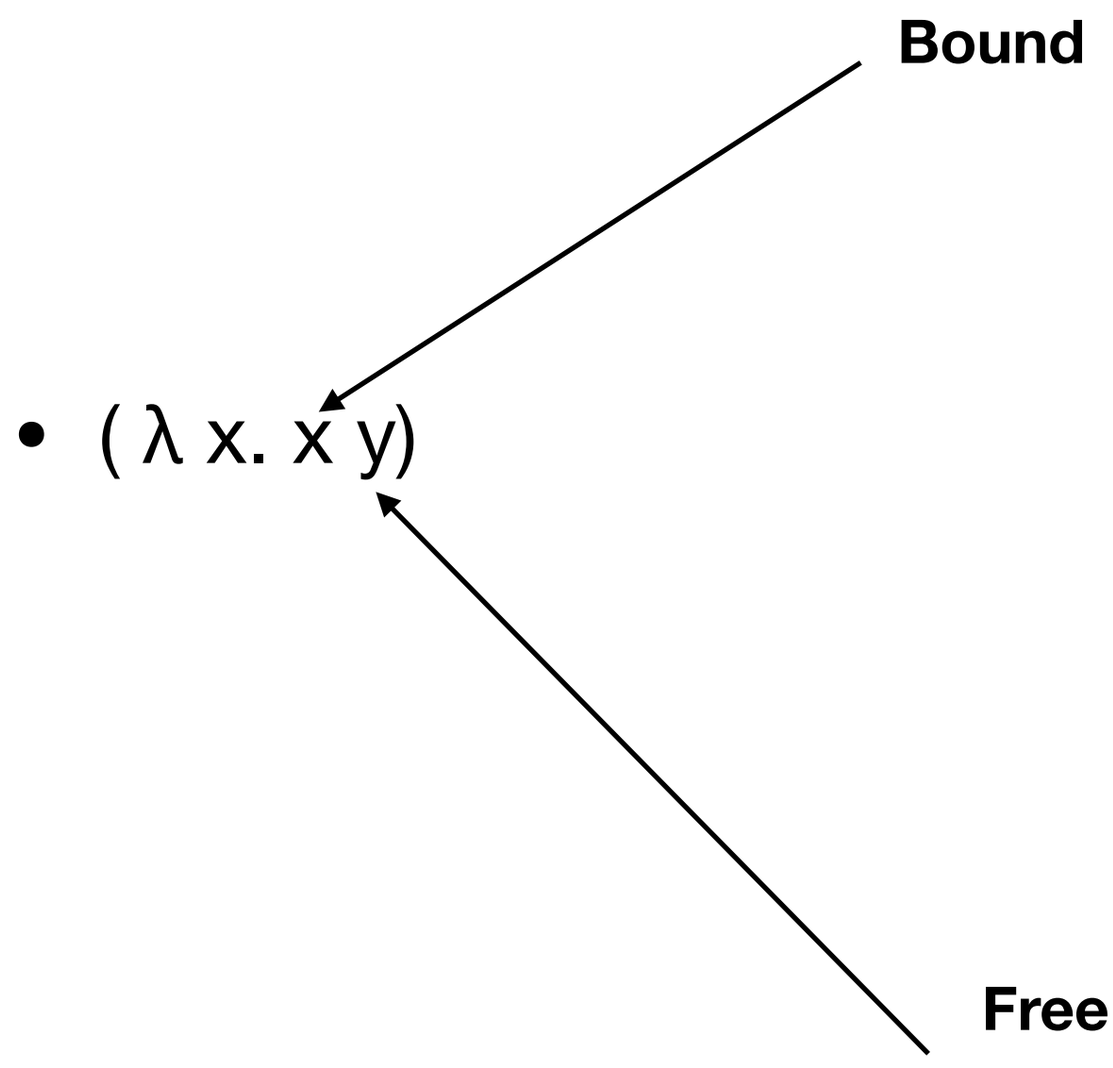
Exercise: Put parentheses

- λ x. λ y. x
- x y λ z. z
- λ x. λ y. x y z

Solution

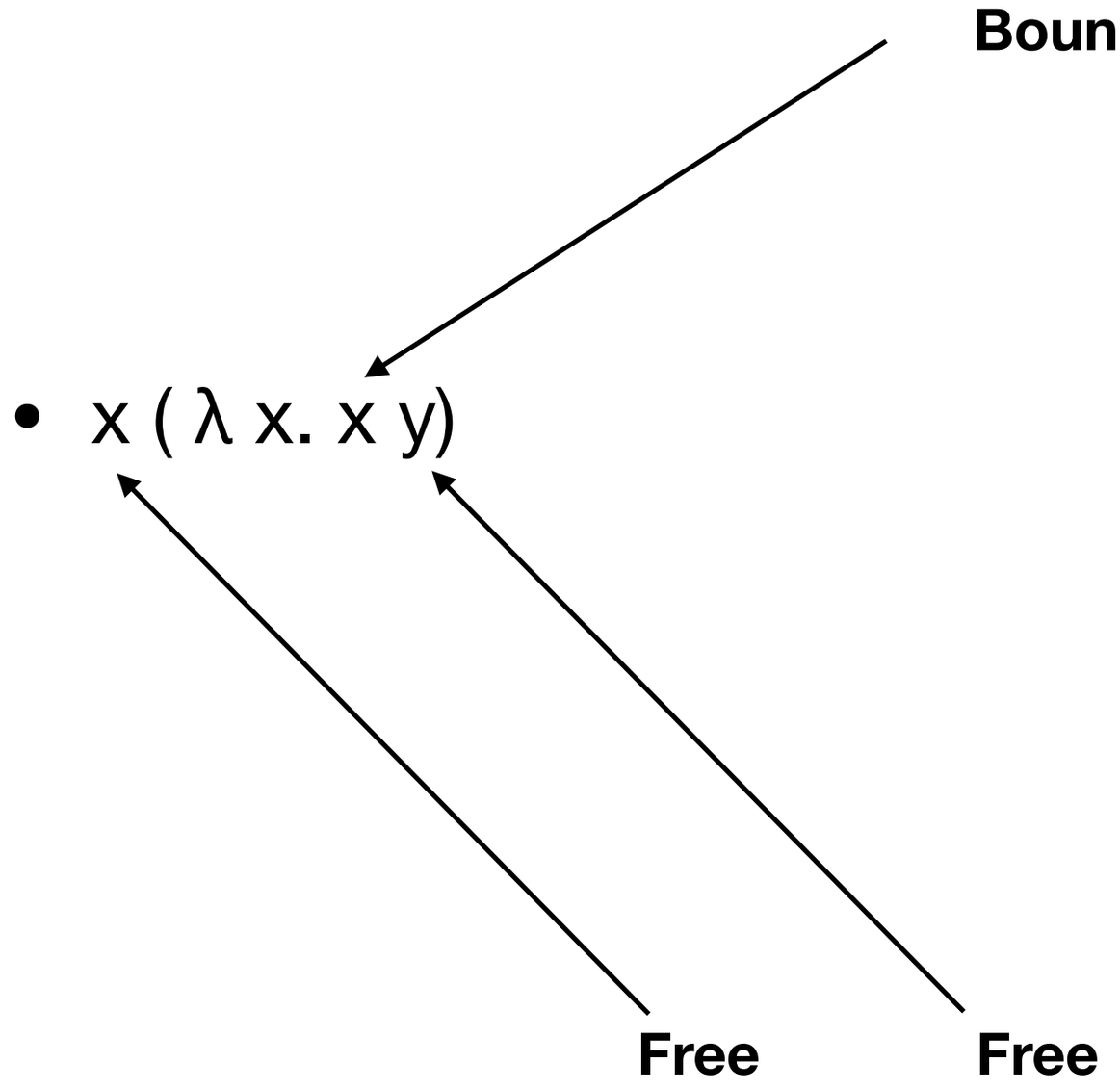
- λ x. (λ y. x)
- (x y) λ z. z
- λ x. (λ y. ((x y) z))

Free and bound variables



- (λ x. x y)

Free and bound variables



Exercise: Find free variables

- λ x. y
- λ x. λ y. x y z
- (λ x.λ y. xy) (λ x. y)
- λ x.λ y. x y λ x. y

Solution

- FV(λ x. y) = { y }
- FV(λ x. λ y. x y z) = { z }
- FV ((λ x.λ y. xy) (λ x. y))= {y} last occurrence of “y”
- FV (λ x.λ y. x y λ x. y) = {} emptyset

Details on Semantics

- Beta Reduction

β-Reduction (aka λ expression evaluation)

Consider this lambda expression:

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

β-Reduction (aka λ expression evaluation)

Consider this lambda expression:

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

This juxtaposition of $(\lambda x.\ +\ x\ 1)$ with 4 means to apply the lambda abstraction $(\lambda x.\ +\ x\ 1)$ to the argument 4. Here's how we do it:

The result of applying a lambda abstraction to an argument is an instance of the body of the lambda abstraction in which bound occurrences of the formal parameter in the body are replaced with copies of the argument.

β-Reduction (aka λ expression evaluation)

Consider this lambda expression:

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

This juxtaposition of $(\lambda x.\ +\ x\ 1)$ with 4 means to apply the lambda abstraction $(\lambda x.\ +\ x\ 1)$ to the argument 4. Here's how we do it:

The result of applying a lambda abstraction to an argument is an instance of the body of the lambda abstraction in which bound occurrences of the formal parameter in the body are replaced with copies of the argument.

I.e. we put the 4 in for the x in $+\ x\ 1$. The result is $+\ 4\ 1$.

$$(\lambda x.\ +\ x\ 1)\ 4 \rightarrow_{\beta} +\ 4\ 1$$

β-Reduction (some more examples)

$(\lambda x.\ +\ x\ x)\ 5 \rightarrow +\ 5\ 5 \rightarrow 10$

β-Reduction (some more examples)

$(\lambda x.\ +\ x\ x)\ 5 \rightarrow +\ 5\ 5 \rightarrow 10$

$(\lambda x.\ 3)\ 5 \rightarrow 3$

β-Reduction (some more examples)

$(\lambda x.\ +\ x\ x)\ 5 \rightarrow +\ 5\ 5 \rightarrow 10$

$(\lambda x.\ 3)\ 5 \rightarrow 3$

$(\lambda x.\ (\lambda y.\ -\ y\ x))\ 4\ 5 \rightarrow (\lambda y.\ -\ y\ 4)\ 5 \rightarrow -\ 5\ 4 \rightarrow 1$

(Note the currying – we peel off the arguments 4 then 5.)

β-Reduction (some more examples)

$(\lambda x.\ +\ x\ x)\ 5 \rightarrow +\ 5\ 5 \rightarrow 10$

$(\lambda x.\ 3)\ 5 \rightarrow 3$

$(\lambda x.\ (\lambda y.\ -\ y\ x))\ 4\ 5 \rightarrow (\lambda y.\ -\ y\ 4)\ 5 \rightarrow -\ 5\ 4 \rightarrow 1$

(Note the currying – we peel off the arguments 4 then 5.)

$(\lambda f.\ f\ 3)\ (\lambda x.\ +\ x\ 1) \rightarrow (\lambda x.\ +\ x\ 1)\ 3 \rightarrow +\ 3\ 1 \rightarrow 4$

β-Reduction (some more examples)

$(\lambda x.\ +\ x\ x)\ 5 \rightarrow\ +\ 5\ 5 \rightarrow\ 10$

$(\lambda x.\ 3)\ 5 \rightarrow\ \ \ \ 3$

$(\lambda x.\ (\lambda y.\ -\ y\ x))\ 4\ 5 \rightarrow\ (\lambda y.\ -\ y\ 4)\ 5 \rightarrow\ -\ 5\ 4 \rightarrow 1$

(Note the currying – we peel off the arguments 4 then 5.)

$(\lambda f.\ f\ 3)\ (\lambda x.\ +\ x\ 1) \rightarrow\ (\lambda x.\ +\ x\ 1)\ 3 \rightarrow\ +\ 3\ 1 \rightarrow 4$

$(\lambda x.\ (\lambda x.\ +\ (-\ x\ 1))\ \ x\ 3)\ 9 \rightarrow (\lambda x.\ +\ (-\ x\ 1))\ 9\ 3$

$\rightarrow\ +\ (\ -\ 9\ 1)\ 3$

$\rightarrow\ 11$

Exercise: beta reduction

- (λ x. λ y. x y) (λ s. s)
- (λ x. x) (λ y. y)
- (λ x. x x) (λ x. x x)

Solution

- (λ x. λ y. x y) (λ s. s) -> λ y. y

- (λ x. x) (λ y. y) -> λ y. y

- (λ x. x x) (λ x. x x) -> (λ x. x x) (λ x. x x) -> itself -> …
  (always the same)

β-Reduction (contd)

(λx. (λx. + (- x 1)) x 3) 9 → (λx. + (- x 1)) 9 3
→ + (- 9 1) 3
→ 11

In this last example we could have applied the reduction in a different order.

(λx. (λx. + (- x 1)) x 3) 9 → (λx. + (- x 1) 3) 9
→ + (- 9 1) 3
→ 11

β-Reduction (ordering)

So, to evaluate an expression we search for reductions, and make them. When the process stops (there are no more reductions to apply), the expression is said to be in ***normal form***. This is the *value* of the original expression.

Wait… there are different orders in which to do the reductions. Does the order matter? If so, we have a problem.

On the ordering of reductions

Church-Rosser Theorem : No expression can be converted into two distinct normal forms.

Details:
How beta reduction works

- Identify a redex (λ x.M)N
- Reduce the redex by substitution M [x:=N]
- Continue until no redex found

Redex

- Reducible expression
- A redex is formed when a lambda definition is immediately followed by an argument, indicating that the function can be applied to that argument

- (λ x.M)N

Exercise: Identify redex

- (λ x. λ y. x y) (λ s. s)
- (λ x. x) (λ y. y)
- (λ x. x x) (λ x. x x)

Substitution

- (λ x.M)N -> M [x := N]
- For all occurrence of x in M:
    - If x is bound by the formal parameter “x” in (λ x.M)
    - the replace x by N in M
- We will consider capture-avoiding situations later

Normal form

- No redex

Exercise: Normal form or not

- λ x.λ y. xy
- λ x. x λ y.x y
- (λ x.λ y. x)y
- (λ x. x λ y.x)y

Exercise: beta reduction

- (λ x. λ y. x y) (λ s. s)
- (λ x. x) (λ y. y)
- (λ x. x x) (λ x. x x)

One more thing about beta reduction: Capture-avoiding substitutions

- The specific names of bound variables in the lambda calculus are meaningless

- λ x. x same as λ y. y

- (λx.(λy.yx)) is equivalent to (λa.(λb.ba))

- Lambda terms that differ only by bound variable names are called alpha equivalent

Exercise: alpha equivalence?

- λ x. xy ? λ z. zy
- λ x. xy ? λ z. xz
- λ x. x λ y. y ? λ z. z λ p. p
- λ x. x λ y. y ? λ y. y λ y. y
- λ x. x λ y. x y ? λ y. y λ y. y y

Exercises: beta reduction

- (λ x.λ y. x)y

Exercises: beta reduction

- (λ x. x λ y.x)y

Exercises: beta reduction

- (λ x.λ y. x y) y

Exercises: beta reduction

- $(\lambda x.\lambda y.xy)(\lambda x.\lambda y.xy)$

Review exercises
alpha-renaming to avoid capture

Reduce each of the lambda terms below until it becomes a normal term

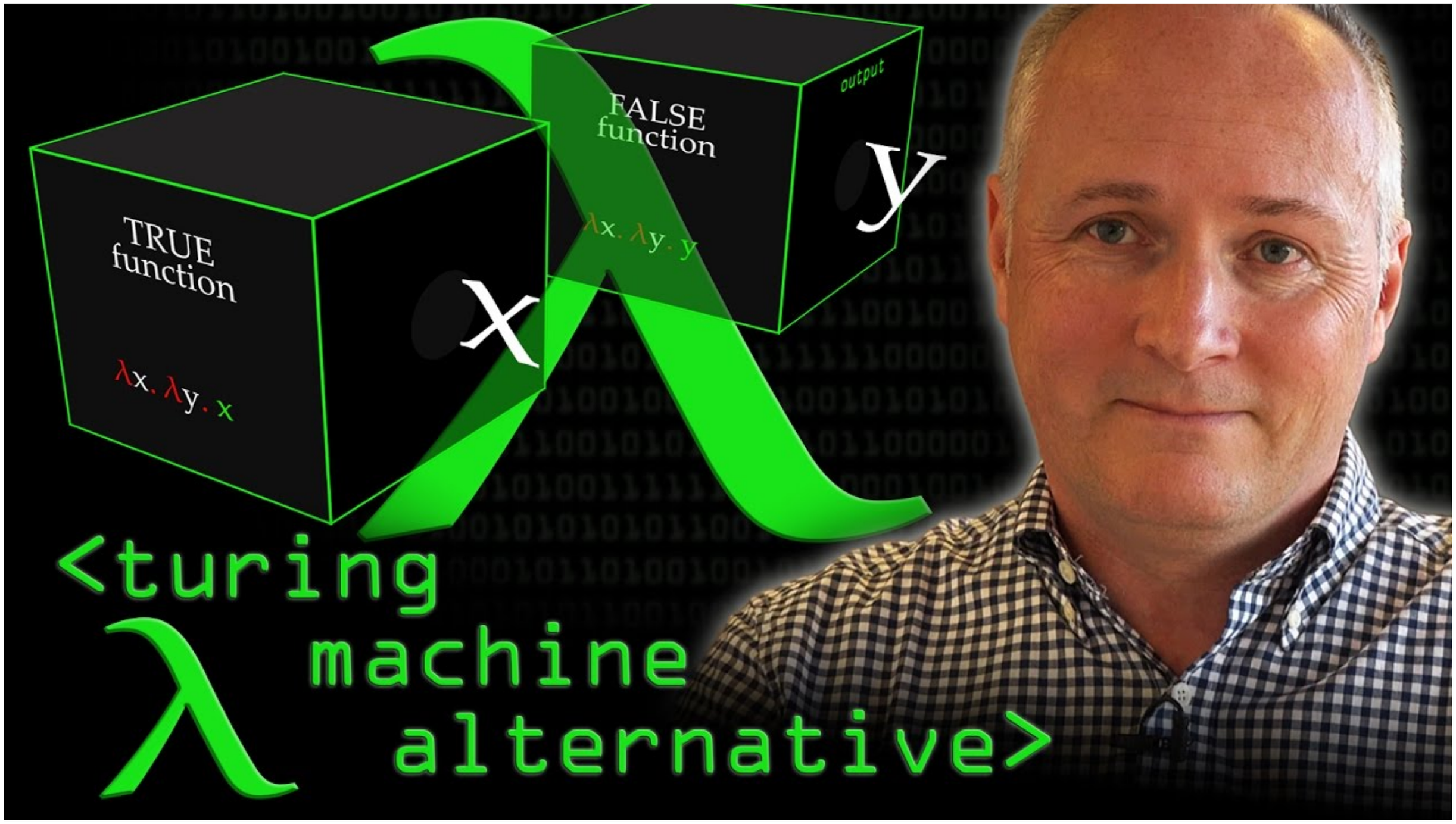
- (λ x .λ y. x y) y
- (λ x. λ y. λ f. f x y) (f x) (g y)

Solution

- (λ x .λ y. x y) y ->beta λ y. x y [x:=y] ->_alpha λ p. x p [x:=y] = λ p. y p

- (λ x. λ y. λ f. f x y) (f x) (g y) ->beta (λ y. λ f. f x y) (x := f x) (g y) ->alpha (λ y. λ p. p x y) [x := f x] (g y) = (λ y. λ p. p (f x) y) (g y) -> (λ p. p (f x) y) [y:= g y] = λ p. p (f x) (g y)

Extending lambda calculus core



6’-10’

- https://youtu.be/eis11j_iGMs

“extending lambda calculus core”?

- The core has only three constructs
- It does not have numbers, conditions, logic, loops
- These things can all be encoded by the core
- We will write **‖<languaage syntax>‖=<lambda-term>** for the encoding
- "*The integers were created by God, everything else is the work of man”*

TRUE, FALSE, and IF

- IF c e1 e2 returns e1 if c is TRUE, or e2 if c is FALSE
- So we encode TRUE by λ x. λ y. x
- We encode FALSE by λ x. λ y. y
- We encode IF by λ c. λ x. λ y. c x y

- Examples
  - if true then b else c = (λx.λy.x) b c → (λy.b) c → b
  - if false then b else c = (λx.λy.y) b c → (λy.y) c → c

Logic AND

- ||AND x y|| = ||IF x y FALSE|| = x y FALSE = x y x
- Thus ||AND|| = λ x. λ y. x y x
- Exercise: ||OR||=?
- Exercise: ||NOT||=?

Logic OR

- Exercise: ||OR||=?

Logic NOT

- Exercise: ||NOT||=?

Numbers

- Any counting system that makes sense would work
- We want n f x = f(f(f(f(… f(x)))))
- Since 0 f x = x, we have ||0|| = λ f. λ x. x
- Since 1 f x = f x, we have ||1|| = λ f. λ x. f x
- … (called Church numerals)

Exercise

- Write lambda calculus to encode SUM

- Think what would be SUM m n

Exercise

- PROD

An important thing in lambda calculus which we will go over quickly: Recursion

Implementing recursion

To give us access to a function itself, we pass it in as another parameter.

FACT = (λf.λn.if (= n 0) 1 (* n (f f (- n 1)))))

(FACT is just shorthand for that string of characters.)

Now if we write

FACT FACT 5

This will work, because the β-reduction substitutes FACT for f, resulting in a function call FACT FACT 4. Etc.

Exercise

- Beta-reduce FACT FACT 3

FACT = (λf.λn.if (= n 0) 1 (* n (f f (- n 1))))