Lambda Calculus

CSE130 - WI19

Agenda

- What is the lambda calculus
- Syntax in a nutshell
- Alpha and Beta reductions
- PA0 tips

λ

Agenda

- **What is the lambda calculus**
- Syntax in a nutshell
- Alpha and Beta reductions
- PA0 tips

λ

What is the Lambda Calculus

A simple programming language that is **turing complete**.

It supports functions *aaaaaand that’s it* :)

For the purposes of this class, you can ‘run it’ through the **Elsa Interpreter** by applying **alpha and beta reductions**.

What is the lambda calculus

When first introduced to it, it **may appear silly.** But notice that it is:

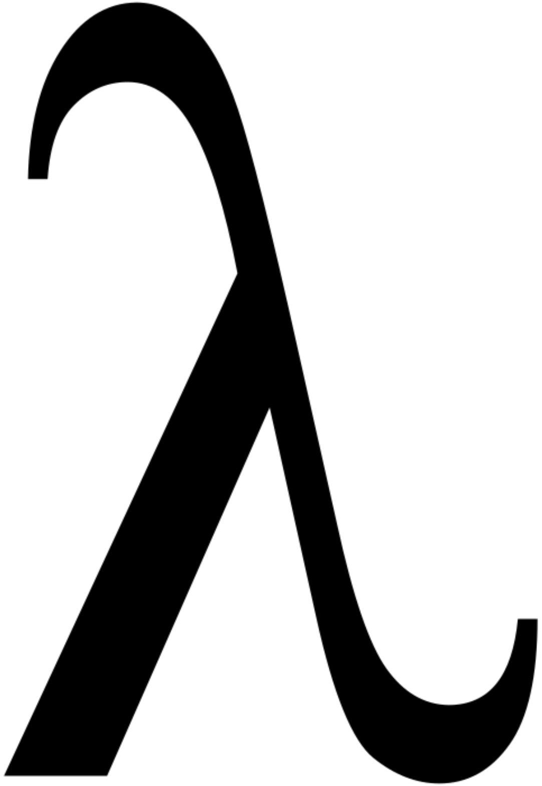
- Turing complete yet simple
- Introduces many prevalent concepts across FP languages
- Fundamental to much PL research
- **Probably going to be on the exam**

On a more serious note though

The lambda calculus is simple but powerful. By learning it, you may come to appreciate a different way of thinking about programming, which is the whole point of an intro PL class :)

Agenda

- What is the lambda calculus
- **Syntax in a nutshell**
- Alpha and Beta reductions
- PA0 advice



Syntax in a nutshell

Only two things you can do:

- Declare a function
- Call a function

$x \mapsto f(x)$

$(\lambda x \to f x)$

```
def fun(x):
    return f(x)
```

$$\left[\underbrace{\lambda y}_{\text{head}} \underset{\text{binder}\atop\text{parameter}}{\to} \underbrace{(\lambda x \to y x)}_{\text{body}}\right] \underset{\text{argument}}{z}$$

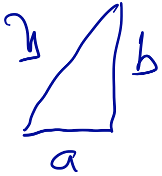
Syntax in a nutshell

Only two things you can do:

- **Declare a function**
- Call a function

$h(a,b) := \sqrt{a^2+b^2}$

$h'(a) := (b \mapsto \sqrt{a^2+b^2})$



$\lambda\ a\ b \to \mathrm{sqrt}\ (a*a + b*b)$

$\lambda\ a \to (\lambda\ b \to \underbrace{\mathrm{sqrt}\ (a*a+b*b)})$

$\mathrm{sqrt}(+\ (*\ a\ a)\ (*\ b\ b))$

Syntax in a nutshell

Declaring functions

Big Ideas:

The Lambda Calculus cannot *explicitly* create functions of two arguments or more.

But it can create *functions that return functions*.
This effectively recreates two (or more) argument functions

```
1  \a -> (\b -> b) -- Function that takes in parameter `a` and returns function
2                  -- that take in parameter `b`
3  \a -> \b -> b   -- Just syntactic sugar
4  \a b -> b       -- Just syntactic sugar (again).
```

(1) and (3) same by convention: | the function body on the RHS → goes as far right as possible

$\lambda\ x \to f\ x$ is understood $\lambda\ x \to (f\ x)$

not $(\lambda x \to f)\ x$

Syntax in a nutshell

Only two things you can do:

- Declare a function
- **Call a function**

Syntax in a nutshell

Call a function

Big Ideas:
It's *perfectly ok* to partially call a function. In other words, it's ok to give only *some* of the parameters to a function.

Why is this allowed?
The answer is in the previous two slides :)

Syntax in a nutshell

Call a function

Big Ideas:
It’s *perfectly ok* to partially call a function. In other words, it’s ok to give only *some* of the parameters to a function.

Why is this allowed?
Because there are *technically* only one-parameter functions. Thus, you still ‘return a value’ (another function) even if you only give some of the parameters

Syntax in a nutshell

$$f(x,y,z) := y$$

$$\lambda a \to (\lambda b \to (\lambda c \to b))\ p$$

$$\Downarrow$$

$$\lambda b \to (\lambda c \to b)$$

Call a function

Assume a variable `PARAM` *exists, then …*

`(\a b c -> b) PARAM`	**returns** `(\b c -> b)`
`(\b c -> b) ANOTHER_PARAM`	**returns** `(\c -> ANOTHER_PARAM)`
`(\c -> ANOTHER_PARAM) LAST_PARAM`	**returns** `ANOTHER_PARAM`

You can also do it a little quicker, ….

`(\a b c -> b) PARAM ANOTHER_PARAM`	**returns** `(\c -> ANOTHER_PARAM)`

Agenda

- What is the lambda calculus
- Syntax in a nutshell
- **Alpha and Beta reductions**
- PA0 Tips

λ

What are beta-steps?

$(\lambda x \to e)\ a$

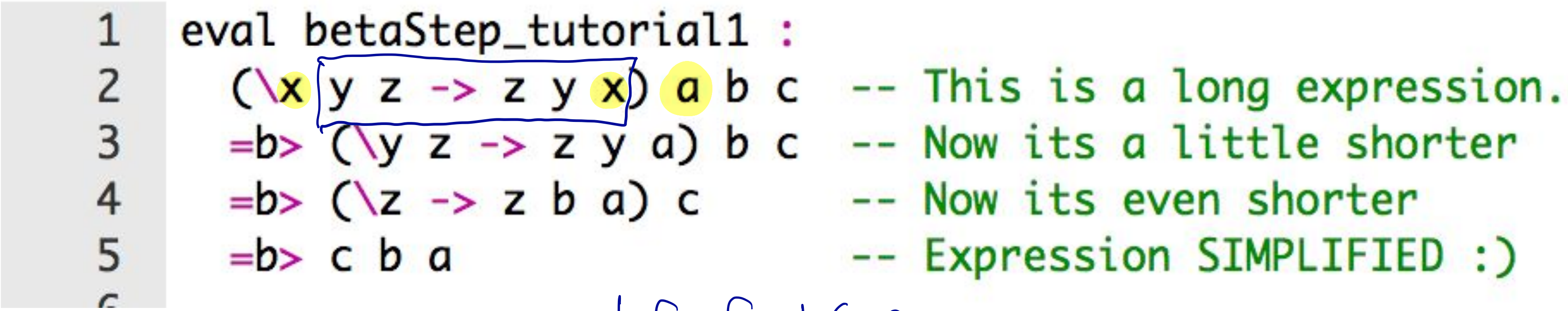
this beta reduces or beta-steps to

$e[a/x]$ or $e[x \mapsto a]$

"e with free occurences of x replaced by a"

The Beta-step:

It's goal is to simplify an expression by calling a function with an argument



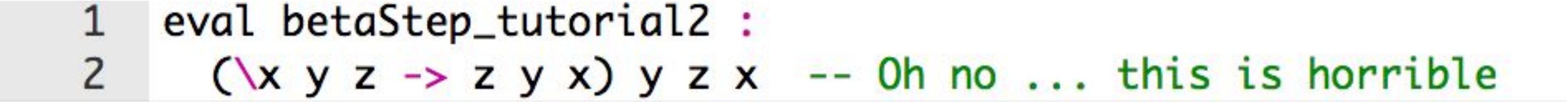
```
eval betaStep_tutorial1 :
  (\x y z -> z y x) a b c  -- This is a long expression.
  =b> (\y z -> z y a) b c  -- Now its a little shorter
  =b> (\z -> z b a) c      -- Now its even shorter
  =b> c b a                -- Expression SIMPLIFIED :)
```

```
def fun1(x):
    def fun2(x):
        return x
```

What are beta-steps?

The Beta-step:

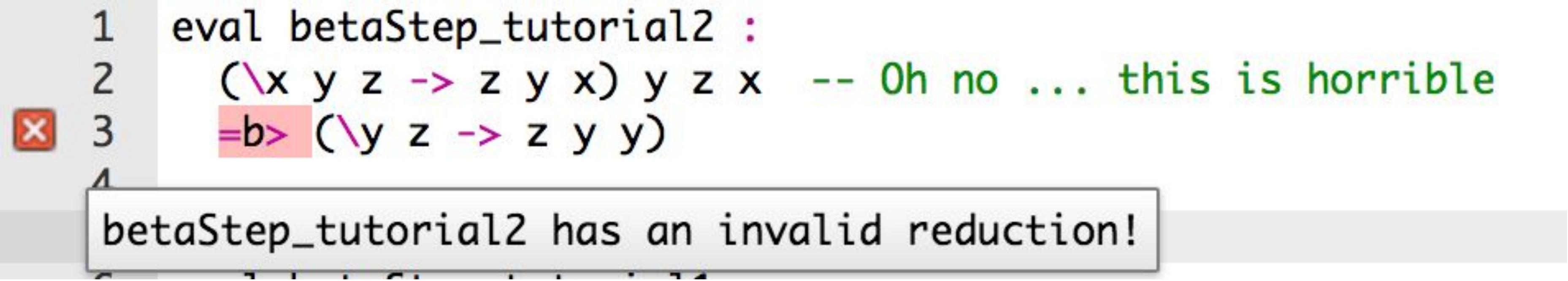
Sometimes variable names can make it hard to keep track of the *scope for the variables*!



```
eval betaStep_tutorial2 :
  (\x y z -> z y x) y z x  -- Oh no ... this is horrible
```

Can you do a beta-step here?

What are beta-steps?



```
eval betaStep_tutorial2 :
  (\x y z -> z y x) y z x  -- Oh no ... this is horrible
  =b> (\y z -> z y y)
```

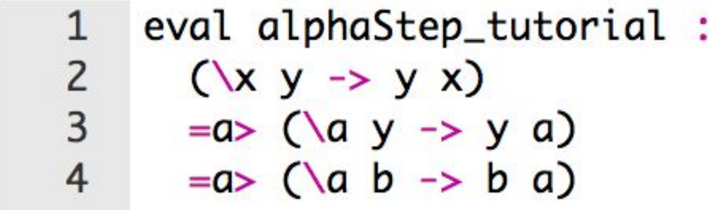
```
betaStep_tutorial2 has an invalid reduction!
```

We need to rename variables first

What are alpha-steps?

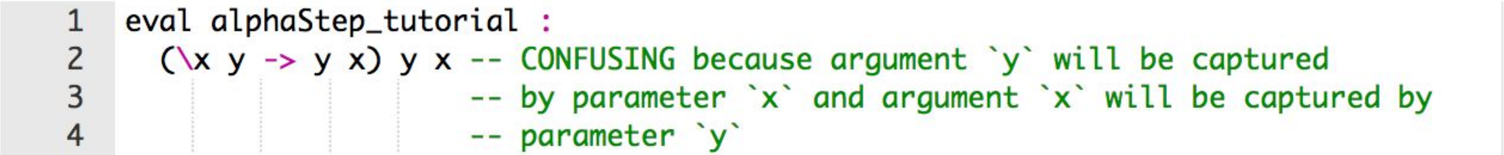
The Alpha-step:

It’s goal is to rename variables.



```
eval alphaStep_tutorial :
  (\x y -> y x)
  =a> (\a y -> y a)
  =a> (\a b -> b a)
```

You wanna use it to enable beta-steps

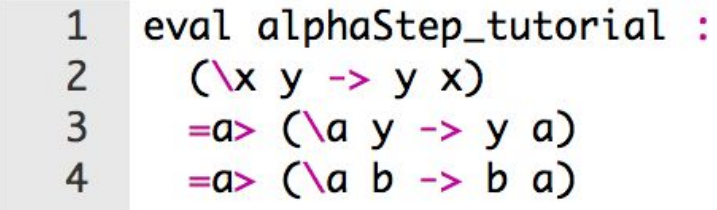


```
eval alphaStep_tutorial :
  (\x y -> y x) y x -- CONFUSING because argument `y` will be captured
                    -- by parameter `x` and argument `x` will be captured by
                    -- parameter `y`
```

What are alpha-steps?

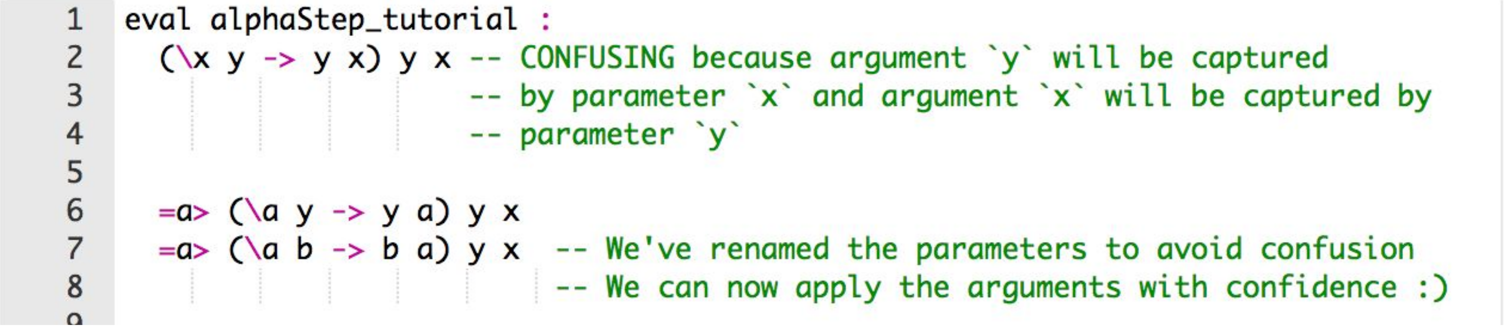
The Alpha-step:

It’s goal is to rename variables.



```
eval alphaStep_tutorial :
  (\x y -> y x)
  =a> (\a y -> y a)
  =a> (\a b -> b a)
```

You wanna use it to enable beta-steps

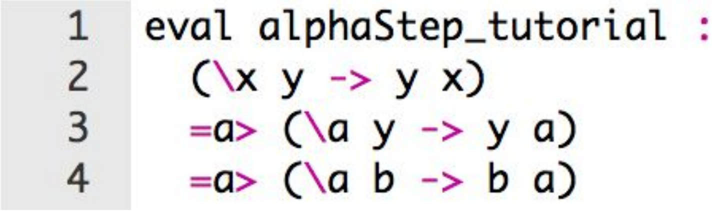


```
eval alphaStep_tutorial :
  (\x y -> y x) y x -- CONFUSING because argument `y` will be captured
                    -- by parameter `x` and argument `x` will be captured by
                    -- parameter `y`

  =a> (\a y -> y a) y x
  =a> (\a b -> b a) y x  -- We've renamed the parameters to avoid confusion
                         -- We can now apply the arguments with confidence :)
```

What are alpha-steps?

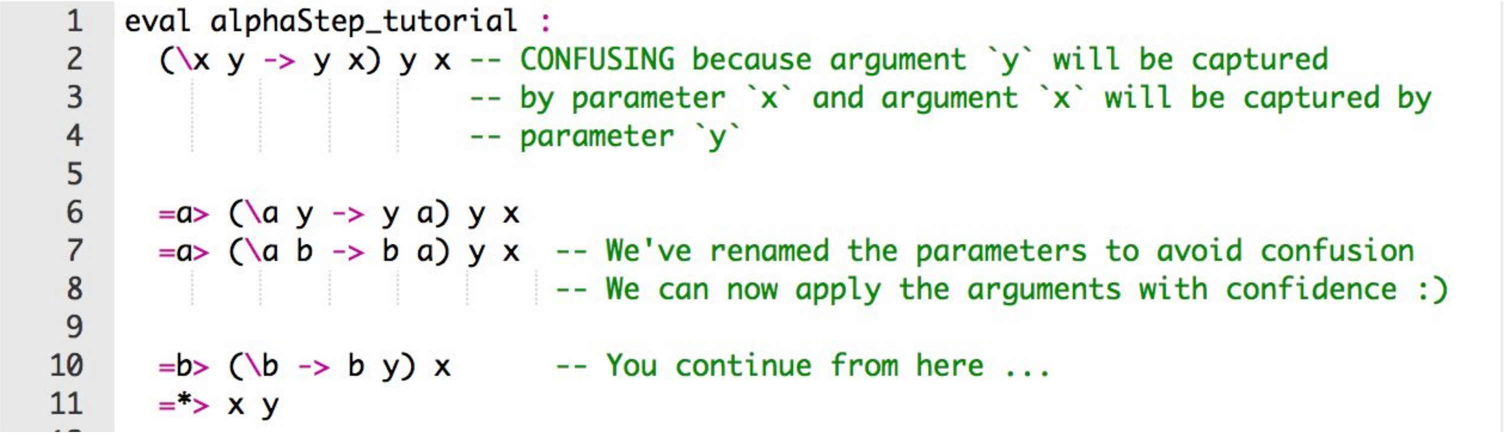
The Alpha-step:

It’s goal is to rename variables.



```
eval alphaStep_tutorial :
  (\x y -> y x)
  =a> (\a y -> y a)
  =a> (\a b -> b a)
```

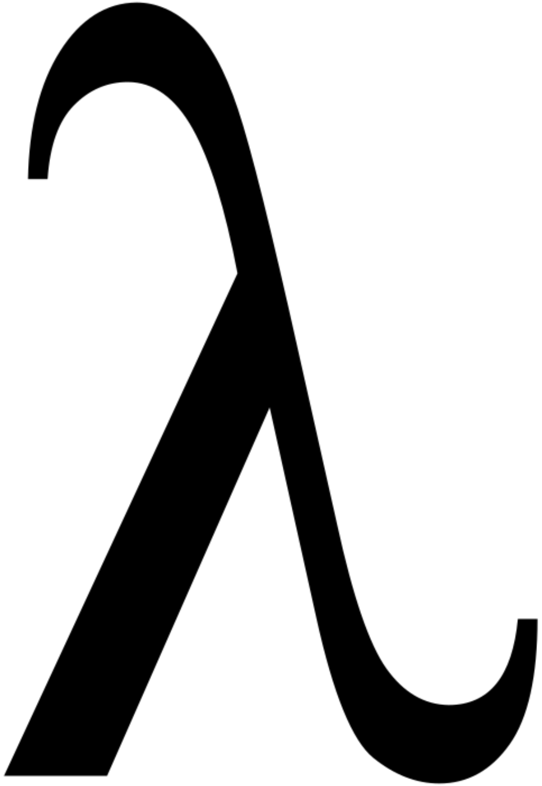
You wanna use it to enable beta-steps



```
eval alphaStep_tutorial :
  (\x y -> y x) y x -- CONFUSING because argument `y` will be captured
                    -- by parameter `x` and argument `x` will be captured by
                    -- parameter `y`

  =a> (\a y -> y a) y x
  =a> (\a b -> b a) y x  -- We've renamed the parameters to avoid confusion
                         -- We can now apply the arguments with confidence :)

  =b> (\b -> b y) x      -- You continue from here ...
  =*> x y
```

Agenda

- What is the lambda calculus
- Syntax in a nutshell
- Alpha and Beta reductions
- **PA0 Tips**



PA0 Overview

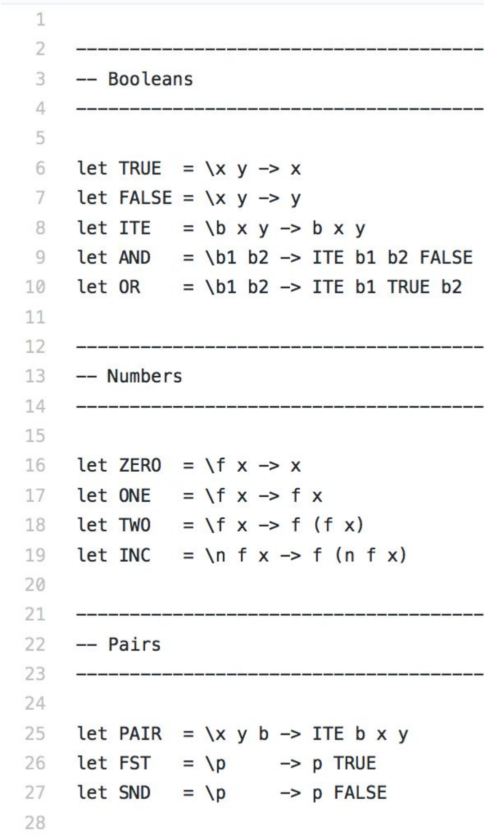
Goal: to simplify a lambda calculus expression through a sequence of alpha and beta reductions.

You'll need to understand:

- how to apply alpha and beta reductions
- The definitions provided to you at the beginning of each source code file

Be aware: that this assignment takes time so *start early*.

Homework Overview: Understanding the definitions



```
--------------------------------------
-- Booleans
--------------------------------------

let TRUE  = \x y -> x
let FALSE = \x y -> y
let ITE   = \b x y -> b x y
let AND   = \b1 b2 -> ITE b1 b2 FALSE
let OR    = \b1 b2 -> ITE b1 TRUE b2

--------------------------------------
-- Numbers
--------------------------------------

let ZERO  = \f x -> x
let ONE   = \f x -> f x
let TWO   = \f x -> f (f x)
let INC   = \n f x -> f (n f x)

--------------------------------------
-- Pairs
--------------------------------------

let PAIR  = \x y b -> ITE b x y
let FST   = \p     -> p TRUE
let SND   = \p     -> p FALSE
```

The Lambda Calculus is a super simple language

we have to implement booleans, numbers, tuples / pairs, and other convenient utilities ourselves.

Note that the definitions we provide only make sense in context

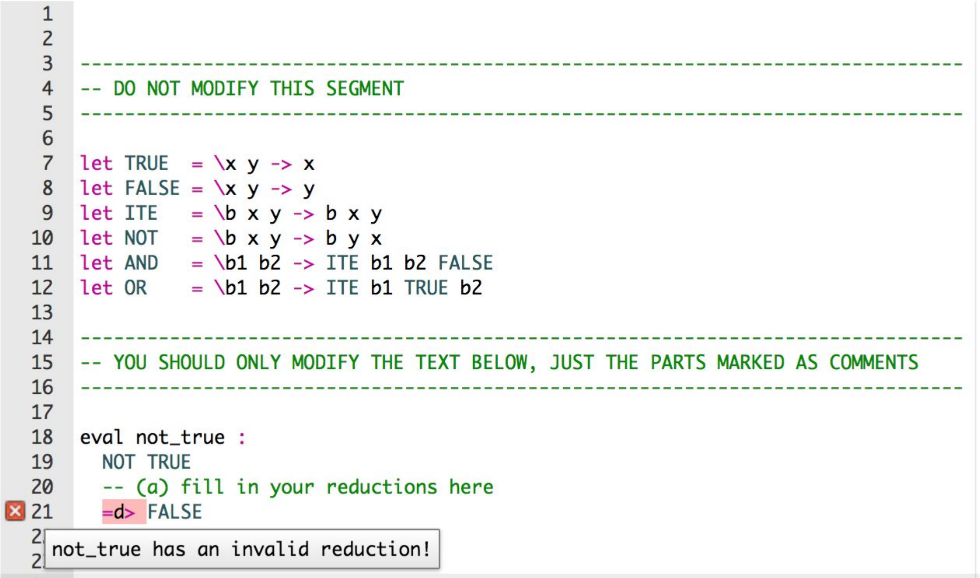
The definition of `TRUE` and `FALSE` will not make sense unless you read `ITE`. So read them all first and ponder on how they fit together!

In my experience, students often fail to realize that:

- `ITE` = If-Then-Else
- `INC` = Increment
- `FST` = Get the first element of a pair
- `SND` = Get the second element of a pair

Homework Overview: How to solve the problems

File: `01_bool.lc`



```
--------------------------------------------------------------------------------
-- DO NOT MODIFY THIS SEGMENT
--------------------------------------------------------------------------------

let TRUE  = \x y -> x
let FALSE = \x y -> y
let ITE   = \b x y -> b x y
let NOT   = \b x y -> b y x
let AND   = \b1 b2 -> ITE b1 b2 FALSE
let OR    = \b1 b2 -> ITE b1 TRUE b2

--------------------------------------------------------------------------------
-- YOU SHOULD ONLY MODIFY THE TEXT BELOW, JUST THE PARTS MARKED AS COMMENTS
--------------------------------------------------------------------------------

eval not_true :
  NOT TRUE
  -- (a) fill in your reductions here
  =d> FALSE
```

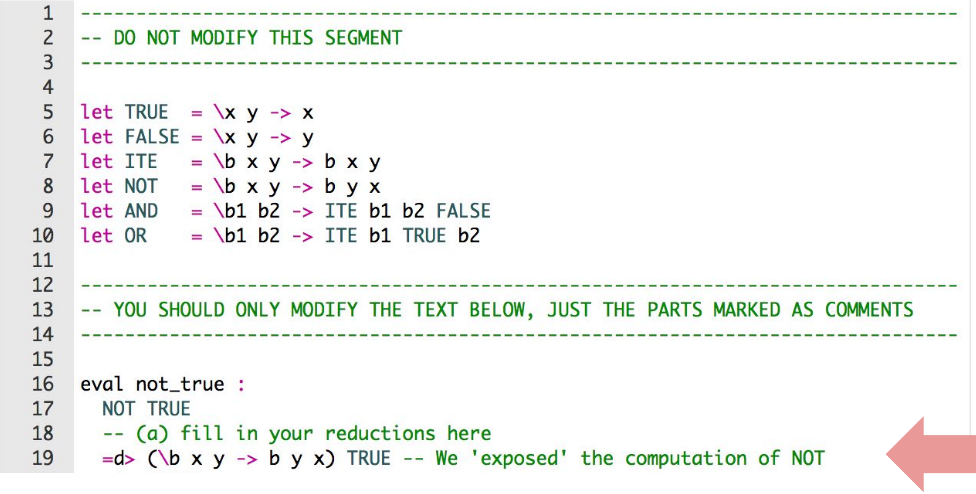
```
not_true has an invalid reduction!
```

How to solve the problems:

- Understand your start and end positions: you want to go from `NOT TRUE` to `FALSE`, it make sense
- Start with a d-step (`=d>`) such that you can expose the actual computation behind a term.

Homework Overview: How to solve the problems

File: `01_bool.lc`



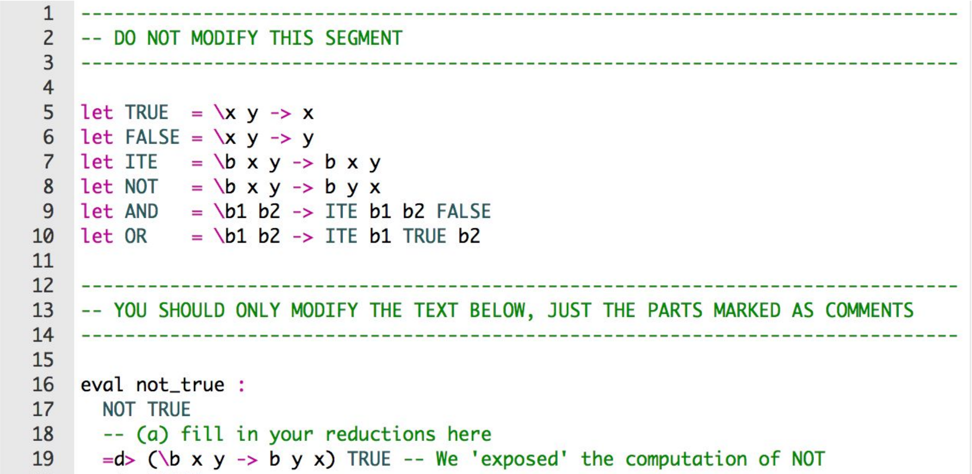
```
-------------------------------------------------------------------------------
-- DO NOT MODIFY THIS SEGMENT
-------------------------------------------------------------------------------

let TRUE  = \x y -> x
let FALSE = \x y -> y
let ITE   = \b x y -> b x y
let NOT   = \b x y -> b y x
let AND   = \b1 b2 -> ITE b1 b2 FALSE
let OR    = \b1 b2 -> ITE b1 TRUE b2

-------------------------------------------------------------------------------
-- YOU SHOULD ONLY MODIFY THE TEXT BELOW, JUST THE PARTS MARKED AS COMMENTS
-------------------------------------------------------------------------------

eval not_true :
  NOT TRUE
  -- (a) fill in your reductions here
  =d> (\b x y -> b y x) TRUE -- We 'exposed' the computation of NOT
```

How to solve the problems:

- Understand your start and end positions: you want to go from `NOT TRUE` to `FALSE`, it make sense
- Start with a d-step (`=d>`) such that you can expose the actual computation behind a term.
- Now simplify with alpha (`=a>`) and beta (`=b>`) reductions!

Homework Overview: How to solve the problems

File: `01_bool.lc`

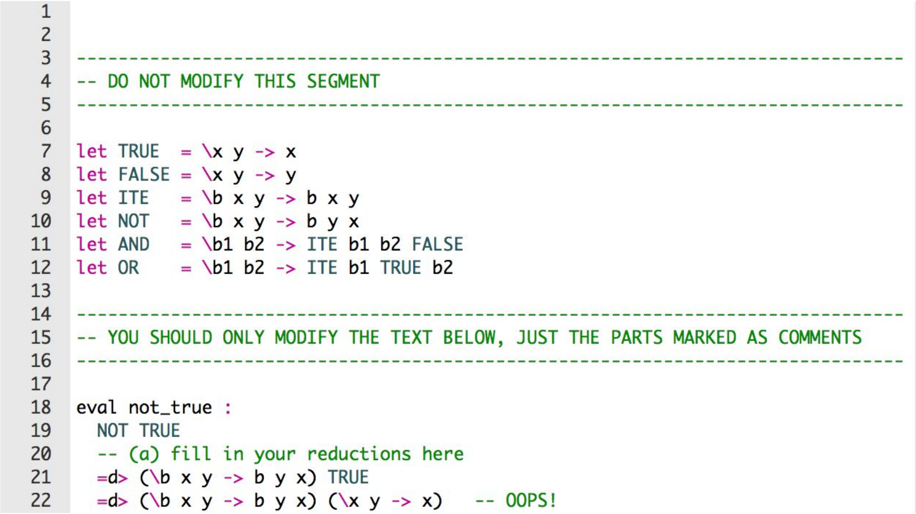


```
--------------------------------------------------------------------------------
-- DO NOT MODIFY THIS SEGMENT
--------------------------------------------------------------------------------

let TRUE  = \x y -> x
let FALSE = \x y -> y
let ITE   = \b x y -> b x y
let NOT   = \b x y -> b y x
let AND   = \b1 b2 -> ITE b1 b2 FALSE
let OR    = \b1 b2 -> ITE b1 TRUE b2

--------------------------------------------------------------------------------
-- YOU SHOULD ONLY MODIFY THE TEXT BELOW, JUST THE PARTS MARKED AS COMMENTS
--------------------------------------------------------------------------------

eval not_true :
  NOT TRUE
  -- (a) fill in your reductions here
  =d> (\b x y -> b y x) TRUE -- We 'exposed' the computation of NOT
```

Would it be a good idea to also expand the definition of `TRUE`?

Homework Overview: How to solve the problems

File: `01_bool.lc`



```


--------------------------------------------------------------------------------
-- DO NOT MODIFY THIS SEGMENT
--------------------------------------------------------------------------------

let TRUE  = \x y -> x
let FALSE = \x y -> y
let ITE   = \b x y -> b x y
let NOT   = \b x y -> b y x
let AND   = \b1 b2 -> ITE b1 b2 FALSE
let OR    = \b1 b2 -> ITE b1 TRUE b2

--------------------------------------------------------------------------------
-- YOU SHOULD ONLY MODIFY THE TEXT BELOW, JUST THE PARTS MARKED AS COMMENTS
--------------------------------------------------------------------------------

eval not_true :
  NOT TRUE
  -- (a) fill in your reductions here
  =d> (\b x y -> b y x) TRUE
  =d> (\b x y -> b y x) (\x y -> x)   -- OOPS!
```

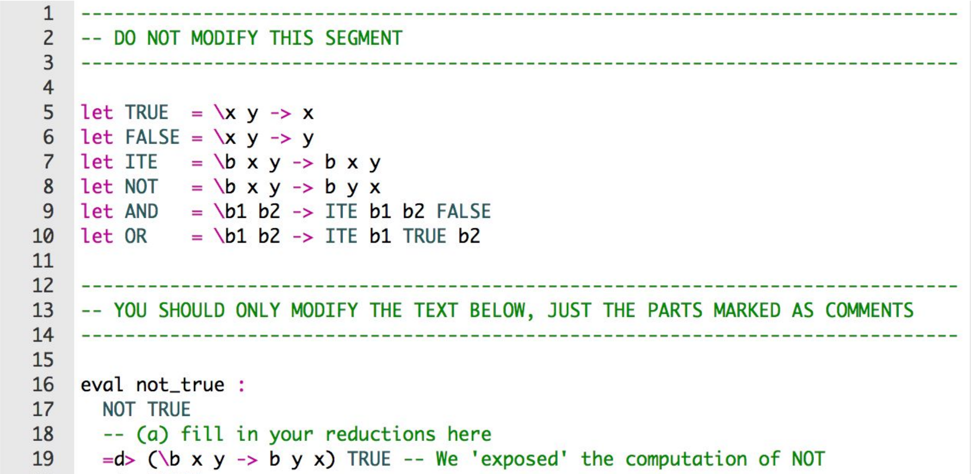
Would it be a good idea to also expand the definition of `TRUE`?

NO

While it’s perfectly legal, it only complicates things. Now you need to do all sorts of renaming before you can do a beta-step.

Homework Overview: How to solve the problems

File: `01_bool.lc`

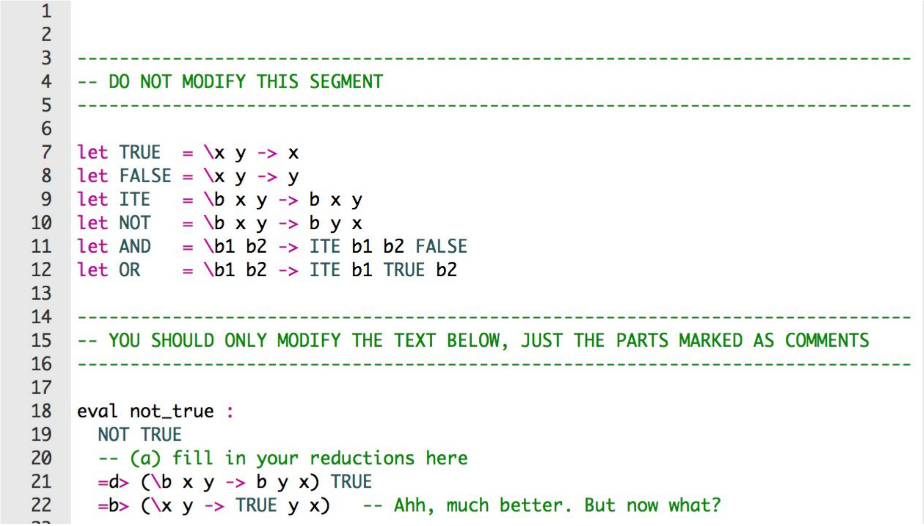


```
--------------------------------------------------------------------------------
-- DO NOT MODIFY THIS SEGMENT
--------------------------------------------------------------------------------

let TRUE  = \x y -> x
let FALSE = \x y -> y
let ITE   = \b x y -> b x y
let NOT   = \b x y -> b y x
let AND   = \b1 b2 -> ITE b1 b2 FALSE
let OR    = \b1 b2 -> ITE b1 TRUE b2

--------------------------------------------------------------------------------
-- YOU SHOULD ONLY MODIFY THE TEXT BELOW, JUST THE PARTS MARKED AS COMMENTS
--------------------------------------------------------------------------------

eval not_true :
  NOT TRUE
  -- (a) fill in your reductions here
  =d> (\b x y -> b y x) TRUE -- We 'exposed' the computation of NOT
```

Before expanding variables, make sure you have simplified your expression as much as possible!

Homework Overview: How to solve the problems

File: `01_bool.lc`



```


--------------------------------------------------------------------------------
-- DO NOT MODIFY THIS SEGMENT
--------------------------------------------------------------------------------

let TRUE  = \x y -> x
let FALSE = \x y -> y
let ITE   = \b x y -> b x y
let NOT   = \b x y -> b y x
let AND   = \b1 b2 -> ITE b1 b2 FALSE
let OR    = \b1 b2 -> ITE b1 TRUE b2

--------------------------------------------------------------------------------
-- YOU SHOULD ONLY MODIFY THE TEXT BELOW, JUST THE PARTS MARKED AS COMMENTS
--------------------------------------------------------------------------------

eval not_true :
  NOT TRUE
  -- (a) fill in your reductions here
  =d> (\b x y -> b y x) TRUE
  =b> (\x y -> TRUE y x)   -- Ahh, much better. But now what?
```

Before expanding variables, make sure you have simplified your expression as much as possible!

Homework Overview: How to solve the problems

The Overall Strategy

1. Start by using `=d>`to expand one of the terms to its definition
    a. If expanding the definition leads to conflicting variable names, then use `=a>` to rename variables
2. Use `=b>` steps to simply that expression
3. Check if done (does the expression you have match the expected goal?)
    a. If yes, congrats!
    b. Else, go back to step 1

Homework Overview: How to solve the problems

Before you go!

Remember that your final code should not have any =*> or =~> in your final solution.

However, they are helpful for checking that your partial solution is correct. I recommend using them while developing your answer but be responsible about it and double check that you do not submit an answer with them!