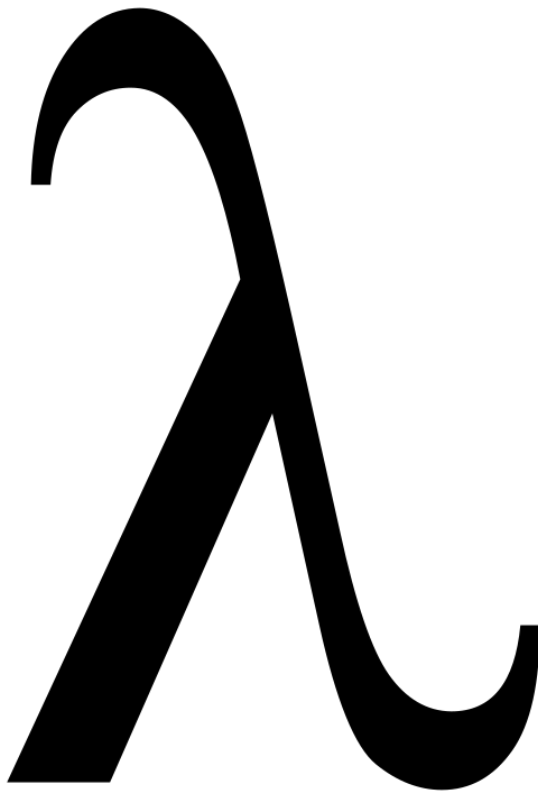


Lambda Calculus

CSE130 - WI19

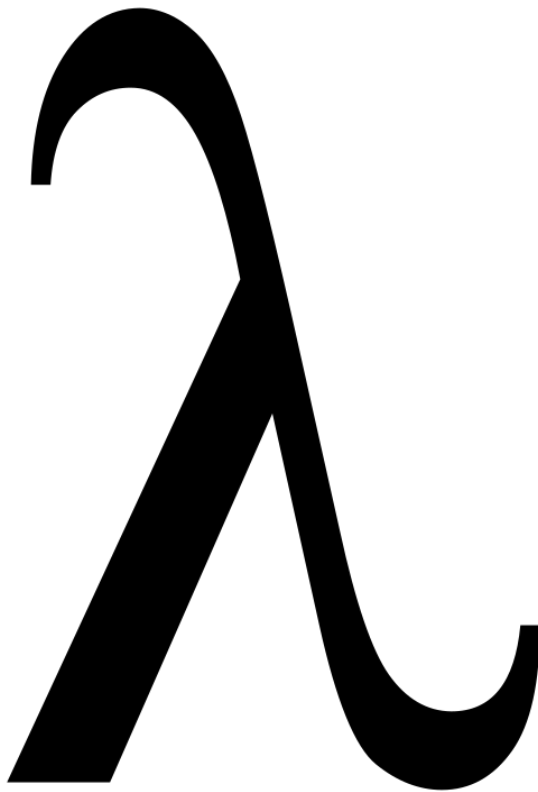
Agenda

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



Agenda

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



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:

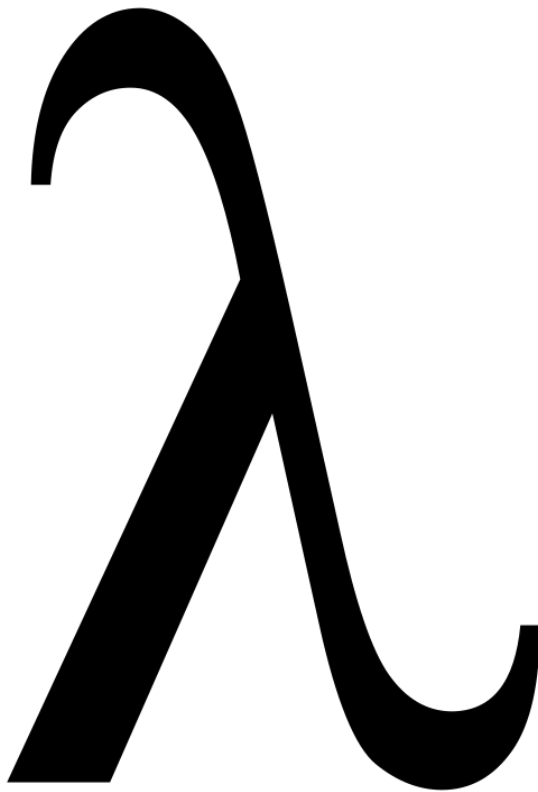
- Turing complete yet simple
- Introduces many prevalent concepts across FP languages
- Fundamental to much PL research
- **Probably going to be on the exam**

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

Syntax in a nutshell

Only two things you can do:

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

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).
```

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

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
```

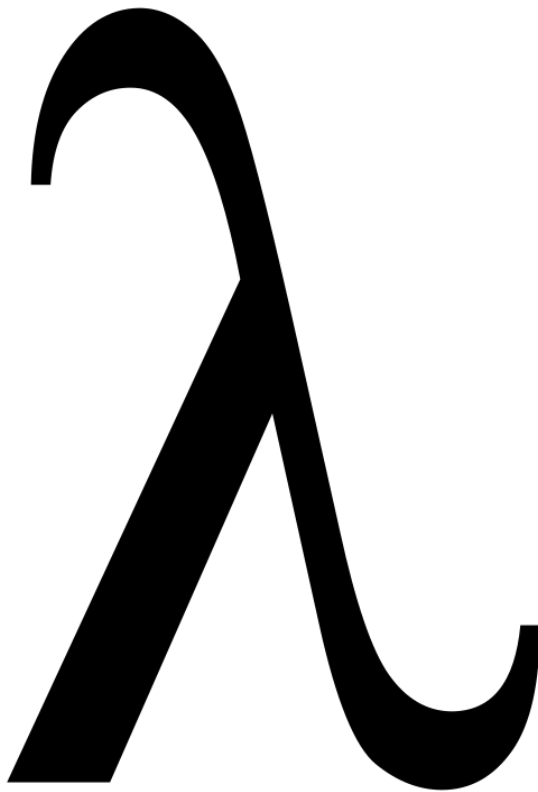
You can also do it a little quicker,

```
(\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



What are beta-steps?

The Beta-step:

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

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


What are beta-steps?

The Beta-step:

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

```
1 eval betaStep_tutorial2 :  
2  (\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?

```
1 eval betaStep_tutorial2 :  
2   (\x y z -> z y x) y z x -- Oh no ... this is horrible  
3   =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.

```
1 eval alphaStep_tutorial :
2   (\x y -> y x)
3   =a> (\a y -> y a)
4   =a> (\a b -> b a)
```

You wanna use it to enable beta-steps

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

What are alpha-steps?

The Alpha-step:

It's goal is to rename variables.

```
1 eval alphaStep_tutorial :
2   (\x y -> y x)
3   =a> (\a y -> y a)
4   =a> (\a b -> b a)
```

You wanna use it to enable beta-steps

```
1 eval alphaStep_tutorial :
2   (\x y -> y x) y x -- CONFUSING because argument `y` will be captured
3                       -- by parameter `x` and argument `x` will be captured by
4                       -- parameter `y`
5
6   =a> (\a y -> y a) y x
7   =a> (\a b -> b a) y x -- We've renamed the parameters to avoid confusion
8                       -- We can now apply the arguments with confidence :)
9
```

What are alpha-steps?

The Alpha-step:

It's goal is to rename variables.

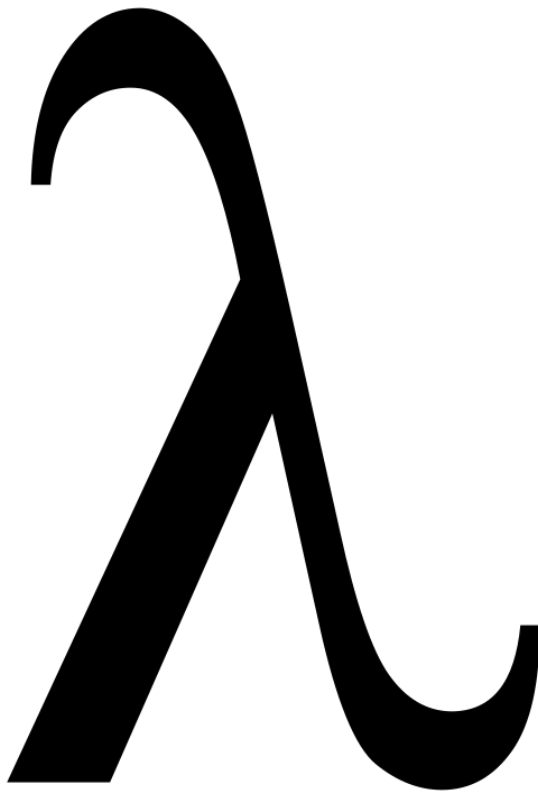
```
1 eval alphaStep_tutorial :
2   (\x y -> y x)
3   =a> (\a y -> y a)
4   =a> (\a b -> b a)
```

You wanna use it to enable beta-steps

```
1 eval alphaStep_tutorial :
2   (\x y -> y x) y x -- CONFUSING because argument `y` will be captured
3                       -- by parameter `x` and argument `x` will be captured by
4                       -- parameter `y`
5
6   =a> (\a y -> y a) y x
7   =a> (\a b -> b a) y x -- We've renamed the parameters to avoid confusion
8                       -- We can now apply the arguments with confidence :)
9
10  =b> (\b -> b y) x      -- You continue from here ...
11  =*> 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

```
1
2 -----
3 -- Booleans
4 -----
5
6 let TRUE  = \x y -> x
7 let FALSE = \x y -> y
8 let ITE   = \b x y -> b x y
9 let AND   = \b1 b2 -> ITE b1 b2 FALSE
10 let OR    = \b1 b2 -> ITE b1 TRUE b2
11
12 -----
13 -- Numbers
14 -----
15
16 let ZERO  = \f x -> x
17 let ONE   = \f x -> f x
18 let TWO   = \f x -> f (f x)
19 let INC   = \n f x -> f (n f x)
20
21 -----
22 -- Pairs
23 -----
24
25 let PAIR  = \x y b -> ITE b x y
26 let FST   = \p      -> p TRUE
27 let SND   = \p      -> p FALSE
28
```

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

```
1
2
3 -----
4 -- DO NOT MODIFY THIS SEGMENT
5 -----
6
7 let TRUE  = \x y -> x
8 let FALSE = \x y -> y
9 let ITE   = \b x y -> b x y
10 let NOT   = \b x y -> b y x
11 let AND   = \b1 b2 -> ITE b1 b2 FALSE
12 let OR    = \b1 b2 -> ITE b1 TRUE b2
13
14 -----
15 -- YOU SHOULD ONLY MODIFY THE TEXT BELOW, JUST THE PARTS MARKED AS COMMENTS
16 -----
17
18 eval not_true :
19   NOT TRUE
20   -- (a) fill in your reductions here
21   =d> FALSE
22
23 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

```
1 -----
2 -- DO NOT MODIFY THIS SEGMENT
3 -----
4
5 let TRUE  = \x y -> x
6 let FALSE = \x y -> y
7 let ITE   = \b x y -> b x y
8 let NOT   = \b x y -> b y x
9 let AND   = \b1 b2 -> ITE b1 b2 FALSE
10 let OR    = \b1 b2 -> ITE b1 TRUE b2
11
12 -----
13 -- YOU SHOULD ONLY MODIFY THE TEXT BELOW, JUST THE PARTS MARKED AS COMMENTS
14 -----
15
16 eval not_true :
17   NOT TRUE
18   -- (a) fill in your reductions here
19   =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

```
1 -----
2 -- DO NOT MODIFY THIS SEGMENT
3 -----
4
5 let TRUE  = \x y -> x
6 let FALSE = \x y -> y
7 let ITE   = \b x y -> b x y
8 let NOT   = \b x y -> b y x
9 let AND   = \b1 b2 -> ITE b1 b2 FALSE
10 let OR    = \b1 b2 -> ITE b1 TRUE b2
11
12 -----
13 -- YOU SHOULD ONLY MODIFY THE TEXT BELOW, JUST THE PARTS MARKED AS COMMENTS
14 -----
15
16 eval not_true :
17   NOT TRUE
18   -- (a) fill in your reductions here
19   =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

```
1
2
3
4  -- DO NOT MODIFY THIS SEGMENT
5  -----
6
7  let TRUE  = \x y -> x
8  let FALSE = \x y -> y
9  let ITE   = \b x y -> b x y
10 let NOT   = \b x y -> b y x
11 let AND   = \b1 b2 -> ITE b1 b2 FALSE
12 let OR    = \b1 b2 -> ITE b1 TRUE b2
13
14 -----
15  -- YOU SHOULD ONLY MODIFY THE TEXT BELOW, JUST THE PARTS MARKED AS COMMENTS
16  -----
17
18 eval not_true :
19   NOT TRUE
20   -- (a) fill in your reductions here
21   =d> (\b x y -> b y x) TRUE
22   =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

```
1 -----
2 -- DO NOT MODIFY THIS SEGMENT
3 -----
4
5 let TRUE  = \x y -> x
6 let FALSE = \x y -> y
7 let ITE   = \b x y -> b x y
8 let NOT   = \b x y -> b y x
9 let AND   = \b1 b2 -> ITE b1 b2 FALSE
10 let OR    = \b1 b2 -> ITE b1 TRUE b2
11
12 -----
13 -- YOU SHOULD ONLY MODIFY THE TEXT BELOW, JUST THE PARTS MARKED AS COMMENTS
14 -----
15
16 eval not_true :
17   NOT TRUE
18   -- (a) fill in your reductions here
19   =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

```
1
2
3 -----
4 -- DO NOT MODIFY THIS SEGMENT
5 -----
6
7 let TRUE  = \x y -> x
8 let FALSE = \x y -> y
9 let ITE   = \b x y -> b x y
10 let NOT   = \b x y -> b y x
11 let AND   = \b1 b2 -> ITE b1 b2 FALSE
12 let OR    = \b1 b2 -> ITE b1 TRUE b2
13
14 -----
15 -- YOU SHOULD ONLY MODIFY THE TEXT BELOW, JUST THE PARTS MARKED AS COMMENTS
16 -----
17
18 eval not_true :
19   NOT TRUE
20   -- (a) fill in your reductions here
21   =d> (\b x y -> b y x) TRUE
22   =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 simplify 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!