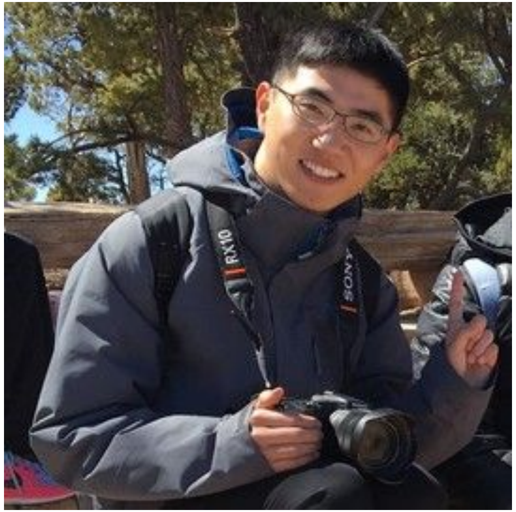
Intros + Lambda Calculus

CS 130 sp 21
4/2/21

Your TAs

Michael James



Zheng Guo

Agenda

Setup

What is the lambda calculus

Syntax

Beta reductions

PA0 tips

Agenda

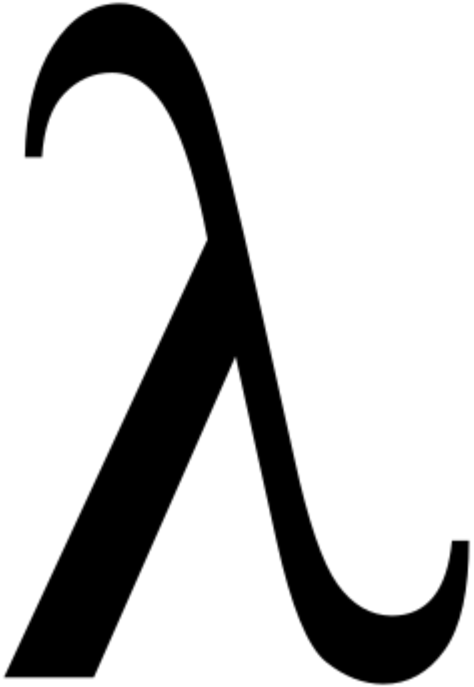
Setup

What is the lambda calculus

Syntax

Beta reductions

PA0 tips



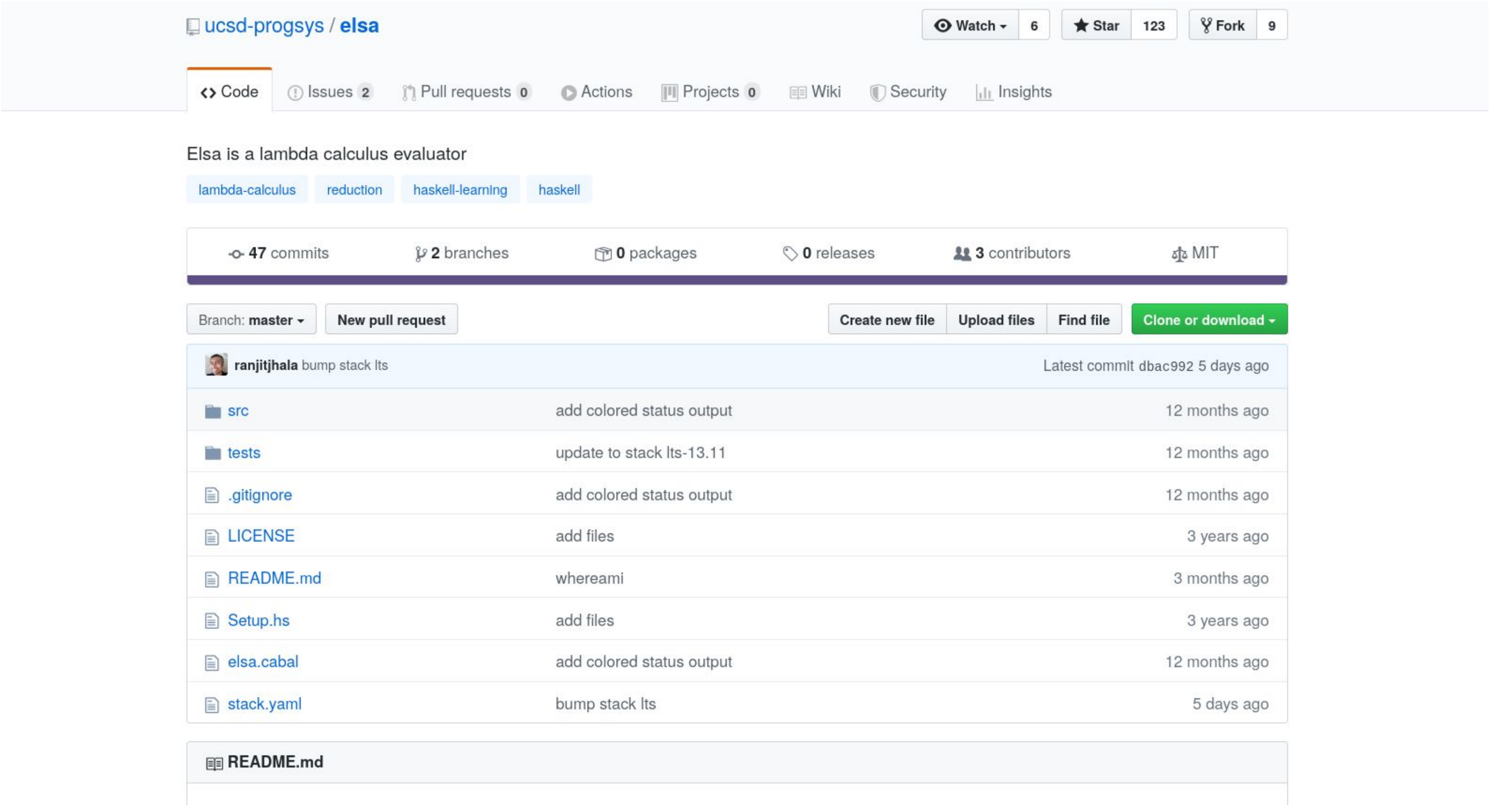
Setup

HW0 will have you *manually* evaluate lambda calculus terms

Elsa checks that **reductions** are valid

Elsa - what is it and how do I use it?

Elsa is implemented as a Haskell package



How do I run elsa and do the HW?

Options:

1. SSH into ieng6
2. Install stack locally
3. Use online demo

SSH into ieng6

Pros:

- Should have everything installed already
- Standardized and easy for us to help us with

Cons:

- Requires internet connection
- Watch out for quota!

Install stack locally

Pros:

- Everything can be done offline
- We will use Haskell throughout the class, you might want it locally

Cons:

- Installing Haskell's stack tool might be annoying
- Unix: should be easy
- Mac: should also be easy with brew
- Windows: Installer from Stack website
- WSL: ??
- `$ stack install elsa`

Online demo

Pros:

- Will “just work”

Cons:

- Very clumsy for doing the homework

Doing the homework

`make test` will check your work

Make sure to commit your work to GH!

Submit a .zip of your git repo to gradescope (use GH's "download zip" feature)

Confused? [The submission instructions are quite detailed]

Do not use =*> or =~> operators!

Agenda

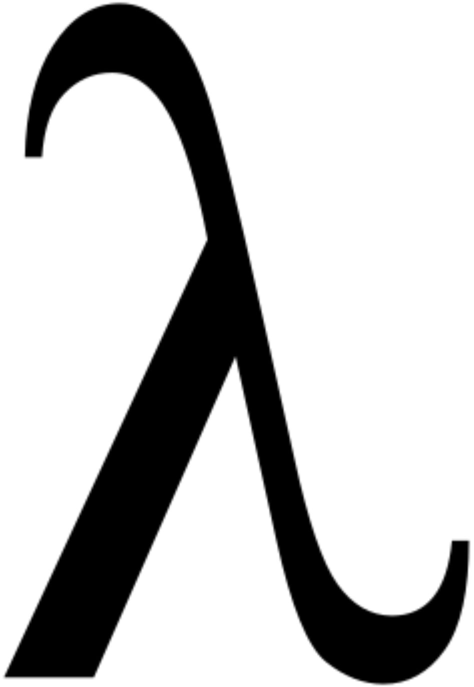
Setup

What is the lambda calculus

Syntax

Beta reductions

PA0 tips



What is the lambda calculus

Very simple programming language

Still Turing complete!

What is the lambda calculus

It might look silly but...

- Simple **formal model** of programming
- Provides a minimal framework for exploring and reasoning about various PL concepts
- Fundamental to lots of PL research (especially functional programming)
- **Definitely on the exam**

Agenda

Setup

What is the lambda calculus

Syntax

Beta reductions

PA0 tips

λ

Syntax

`x` : Variable

`(\x -> M)` : Function abstraction (M is a lambda term)

`(M N)` : Function application (M, N are lambda terms)

All we can do is declare functions and apply functions!

Functions are *first-class*: We can apply functions to other functions, and a function can return another function

Syntax

```
\a -> (\b -> b)  -- Function that takes a parameter “a” and
                 -- returns a function that takes a param “b”
```

```
\a -> \b -> b    -- Syntactic sugar for above
```

```
\a b -> b        -- More syntactic sugar
```

Syntax Quiz

Which is equivalent to:

```
(\foo -> (\bar -> (\baz -> baz bar foo)))
```

No reductions! Just syntactic sugar!

```
a.  (\foo bar baz -> baz bar foo)
b.  (\foo -> (\bar -> (\baz -> (baz (bar foo)))))
c.  (\foo -> \bar) -> (\baz -> baz bar foo)
d.  A & B
```

Syntax Quiz

Which is equivalent to:

```
(\foo -> (\bar -> (\baz -> baz bar foo)))
```

No reductions! Just syntactic sugar!

```
a.  (\foo bar baz -> baz bar foo)
b.  (\foo -> (\bar -> (\baz -> (baz (bar foo)))))
c.  (\foo -> \bar) -> (\baz -> baz bar foo)
d.  A & B
```

Syntax Quiz

Which is equivalent to:

```
(\foo -> (\bar -> (\baz -> baz bar foo)))
```

No reductions! Just syntactic sugar!

Not the same!

```
a.  (\foo bar baz -> baz bar foo)
b.  (\foo -> (\bar -> (\baz -> (baz (bar foo)))))
c.  (\foo -> \bar) -> (\baz -> bar bar foo)
d.  A & B
```

Syntactic Sugar

- `\x -> (\y -> (\z -> E))` we can rewrite: `\x -> \y -> \z -> E`
- `\x -> \y -> \z -> E` we can rewrite: `\x y z -> E`
- `(((E1 E2) E3) E4)` we can rewrite: `E1 E2 E3 E4`

Associativity

Application is **left** associative!

```
E1 E2 E3 =

((E1 E2) E3)
```

Abstraction is **right** associative!

```
\x -> \y -> \z -> E =

\x -> (\y -> (\z -> E)
```

Quiz

Fully parenthesize: **`\b1 b2 -> ITE b1 b2 FALSE`**

```
a.  ((((\b1 b2 -> ITE) b1) b2) FALSE)

b.  (\b1 b2 -> (ITE (b1 (b2 FALSE))))

c.  (\b1 b2 -> ((ITE b1) b2) FALSE)

d.  (\b1 b2 -> (ITE b1) (b2 FALSE))
```

Quiz

Fully parenthesize: `\b1 b2 -> ITE b1 b2 FALSE`

```
a.  ((((\b1 b2 -> ITE) b1) b2) FALSE)

b.  (\b1 b2 -> (ITE (b1 (b2 FALSE))))
```

c. `(\b1 b2 -> ((ITE b1) b2) FALSE)`

```
d.  (\b1 b2 -> (ITE b1) (b2 FALSE))
```

Agenda

Setup

What is the lambda calculus

Syntax

Beta reductions

PA0 tips

Alpha/Beta reductions

Beta step: Calling a function

Alpha step: Renaming a variable inside a function

Beta step

What do we do with `(\x -> x) y` ?

We can **substitute** y for x inside the body of the function: we just get `y`

More examples:

`(\a b c -> b) d` becomes `(\b c -> b)`

`(\b c -> b) e` becomes `(\c -> e)`

`(\a b c -> b) d e` becomes `(\c -> e)`

$\boxed{a\ (\backslash b \to b)\ c}$

~~-> b a c~~

No more reduction

$\underbrace{(a\ (\backslash b \to b))}\ c$

In Elsa

```

eval beta :
  (\f x -> f (f x)) g

```

In Elsa

```
eval beta :
  (\f x -> f (f x)) g
  =b> \x -> g (x x)
```

g

$(\lambda x \to g\,(g\,x))$

In Elsa

```
eval beta :
  (\f x -> f (f x)) g
  =b> \x ->  g (g x)
```

Quiz

What does: **`(\f x -> f (f x)) incr one`** beta-reduce to (in one step)?

a. `(\x -> incr (f x)) one`

b. `(\x -> one (one x)) incr`

c. `(\x -> incr (incr x)) one`

d. `(\f -> f (f one)) incr`

Quiz

What does `(\f x -> f (f x)) incr one` beta-reduce to (in one step)?

1st 2nd

a. `(\x -> incr (f x)) one`

b. `(\x -> one (one x)) incr`

c. `(\x -> incr (incr x)) one`

d. `(\f -> f (f one)) incr`

(\f x -> (f (f x))) incr one

=b> (\x -> incr (incr x)) one

=b> incr (incr one)

Quiz

What does: **`(\f x -> f (\f -> f x) x) incr one`** beta-reduce to?

a. `(\x -> incr (\f -> incr x) x) one`
b. `(\x -> incr (\incr -> incr x) x) one`
c. `(\f -> incr (incr x) x) one`
d. `(\x -> incr (\f -> f x) x) one`

Quiz

x is free

free fs

bound

What does: `(\f x -> f (\f -> f x) x) incr one` beta-reduce to?

a. `(\x -> incr (\f -> incr x) x) one`

b. `(\x -> incr (\incr -> incr x) x) one`

c. `(\f -> incr (incr x) x) one`

d. **`(\x -> incr (\f -> f x) x) one`** ← start

(incr (\f -> f one)) one

Agenda

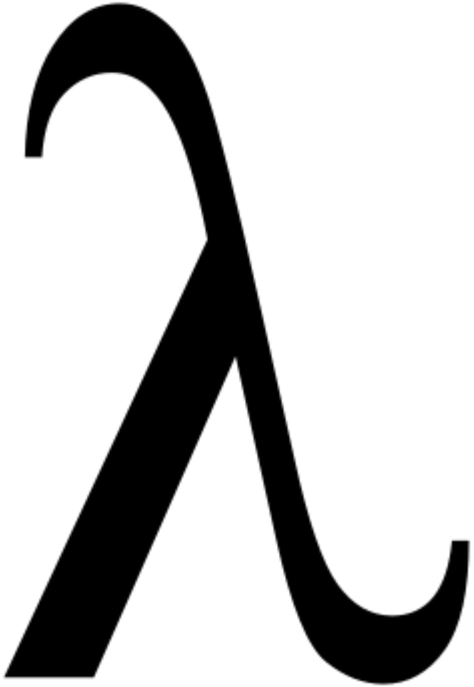
Setup

What is the lambda calculus

Syntax

Beta reductions

PA0 tips



PA0 Overview

Goal: Simplify lambda calculus expressions via alpha/beta steps

You will need to understand:

- How to apply alpha/beta steps
- The definitions in each source file

Be aware: the lambda calculus is weird! This might take time

PA0 Overview

Each problem will define higher-level concepts with lambda terms:

```
--------------------------------------------------------------------------------
-- 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
```

Most of these definitions will not make sense on their own!

TRUE and FALSE make no sense without the definition of ITE -- you need to read all the definitions and try to figure out how they work together

PA0 overview

(\b xy-> b xy) Not

apply Not

Expand (\b xy-> y x)

Elsa also offers a =d> operator

This allows you to replace symbols with their definition -- this is key! Use it early

```
--------------------------------------------------------------------------------
-- 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
```

let APPLE = \x -> x

However, you can also make the problems too complicated...

If we replace **all** definitions, we might end up with too much complexity!

Which of these is easier to work with? Why?

```
eval not_true :
  NOT TRUE
  =d> (\b x y -> b y x) TRUE
```

```
eval not_true :
  NOT TRUE
  =d> (\b x y -> b y x) (\x y -> x)
```

Q & A / Live Examples