CSCI 334:
Principles of Programming Languages

Lecture 8: Lambda, lambda, lambda!

Instructor: Dan Barowy

Williams

1

## Topics

Lambda calculus—how to survive it

2

## Your to-dos

1. Lab 4, **due Monday 3/4** (solo lab)
2. Review feedback if you haven't already…

3

## Announcements

- CS Colloquium this **Friday, Mar 1 @ 2:35pm in Wege Auditorium (TCL 123)**

Lightweight, Modular Verification for Systems Compilers
Prof. Alexa VanHattum (Wellesley)

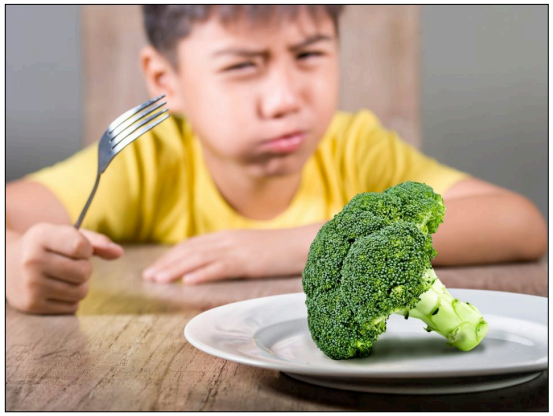
Language-level system guarantees, like runtime isolation for WebAssembly modules, are only as strong as the compiler that produces a native-machine-specific executable. Subtle wrong-code bugs in the compiler can introduce serious security flaws. In this talk, I'll describe Crocus, our system for lightweight, modular verification of instruction-lowering rules in Cranelift, an industry WebAssembly compiler. Crocus reproduces known bugs (including a 9.9/10 severity security bug) and identifies previously-unknown bugs and underspecified compiler invariants. More broadly, I'll discuss how integrating lightweight formal methods can free systems engineers from having to choose between prioritizing efficiency and correctness.

4

## Announcements

- **Midterm exam**, **Thursday, March 14**, on paper, in class.
- Resubmissions are due by the **last day of the final exam reading period**.

5



6

## Abstract Syntax Trees

7

An **abstract syntax tree** (**AST**) is a tree representation of a language such that **all operations are interior nodes** and **all data are leaf nodes**. As such, ASTs are frequently used to represent programs.

An AST can be obtained from a derivation by a set of **tree-transformation rules**. These rules are language-specific. **See handout**.

## Activity: Abstract Syntax Trees

8

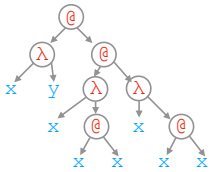
**Derive** this expression, and then **convert** it to an AST.

`(λx.y)((λx.xx)(λx.xx))`

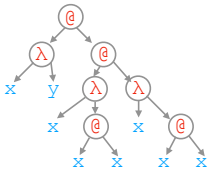
## Activity: Abstract Syntax Trees

9

Did you get this AST?



Which reduction do I perform?

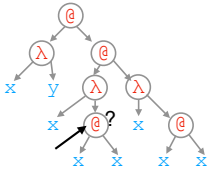


Redex: 

**redex** = application with abstraction as left child.

10

Which reduction do I perform?

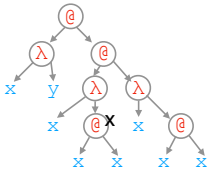



Redex: 

11

Is this a redex?

Which reduction do I perform?

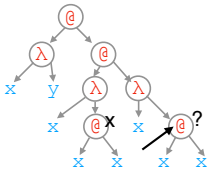



Redex: 

12

No.

Which reduction do I perform?



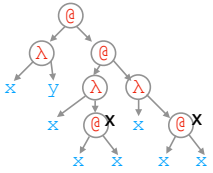

Redex:


13

How about this one?

---

Which reduction do I perform?



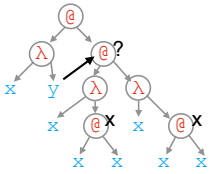

Redex:


14

No.

---

Which reduction do I perform?

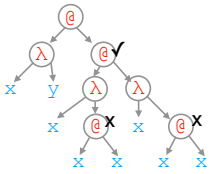



Redex:


15

This one?

Which reduction do I perform?



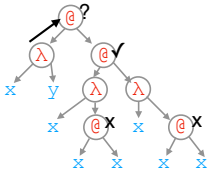
Redex:

16

Yes.

---

Which reduction do I perform?



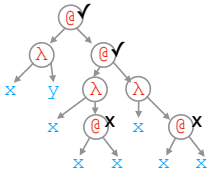
Redex:

17

This one?

---

Which reduction do I perform?

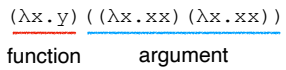


Redex:

18

Yes. So we have two redexes available. Remember if there's a redex, it means that you can beta reduce.

Reduction strategies

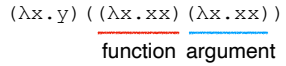



19

To see the same thing textually, here is the redex at the top of the last diagram.

---

Reduction strategies

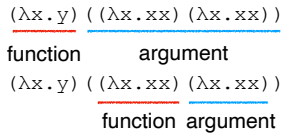



20

And here's the redex at the bottom of the last diagram.

---

Which reduction do I perform?




21

Which one should I pick?

## Two well-known reduction orders

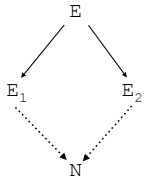
- Normal order
- Applicative order

22

There are two general strategies for picking. The normal order and the applicative order.

## Sometimes multiple reductions available

Order (mostly) does not matter




If $E \rightarrow E_1$ and $E \rightarrow E_2$
then $E_1 \rightarrow^* N$ and $E_2 \rightarrow^* N$
for some N

**confluence**

23

But it turns out that it doesn't matter all that much (it sort of matters… more later) which one you choose. The lambda calculus is confluent, so it will (usually) work out OK.

## Demonstration

**Normal order** ("outermost leftmost") reduction

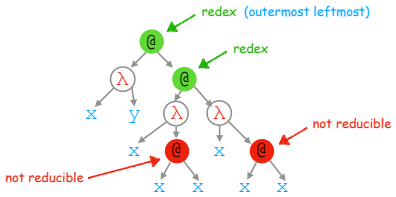
`(λx.y)((λx.xx)(λx.xx))`

What does "outermost leftmost" mean?


24

The normal order is outermost leftmost. What does this mean? Find the redex furthest up in the AST. If there's a tie for furthest up, choose the one on the left.

(λx.y) ((λx.xx) (λx.xx))




What does “outermost leftmost” mean?

25

Here’s the outermost leftmost redex.

---

Demonstration

**Applicative order** (“innermost leftmost”) reduction

(λx.y) ((λx.xx) (λx.xx))

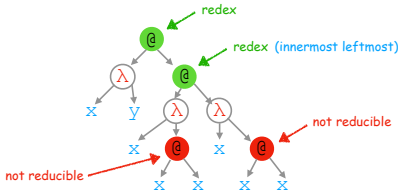
What does “innermost leftmost” mean?


26

The applicative order is innermost leftmost. What does this mean? Find the redex furthest down in the AST. If there’s a tie for furthest down, choose the one on the left.

---

(λx.y) ((λx.xx) (λx.xx))




What does “innermost leftmost” mean?

27

Here is the innermost leftmost redex.

## Meaning of "equivalence"

The only **equivalent** expressions in the lambda calculus are those that are **textually identical**.

`λa.aa ≠ λb.bb`

after alpha reducing a for b:

`λa.aa = λa.aa`

28

Determining whether two lambda programs do the same thing is easy: they have to have exactly the same text.

## One caveat about reduction orders

Although reduction order "does not matter" (because the LC is confluent), only the **normal order** reduction is **guaranteed to terminate** for expressions that **have a normal form**.

(see LC, part 2 from packet for more detail)

29

This is an annoying fact. Sometimes applicative order can get "stuck in a loop." You won't derive anything untrue, but the reduction may not make any progress.

## More practice finding redexes

`(λa.(λz.(+ x z))((λz.(+ x z)) a )) 2`

Normal order: `(λa.(λz.(+ x z))((λz.(+ x z)) a )) 2`

Applicative order: `(λa.(λz.(+ x z))(λz.(+ x z)) a )) 2`

Neither: `(λa.(λz.(+ x z))((λz.(+ x z)) a )) 2`

30

Here's another expression. Try covering up the answers and see if you can find the three redexes on your own.

Trouble matching parens? Try this.

```
(λa.(λz.(+ x z))((λz.(+ x z)) a )) 2
1   2   3     3 2 2 3   4     4 3   2 1
```

31

If you're having trouble doing these, try this trick. Number the parens so that you can tell where parts of the expression start and end.

---

More practice finding redexes

```
(λa.(λz.(+ x z))((λz.(+ x z)) a )) 2
```

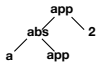

32

Here, I am going to generate an AST from the expression. I will highlight the parts in red and blue as I write them down so that you can see how I got them. Observe that I am not using a derivation tree to get the AST. But if you are confused, go back to the derivation tree approach, then derive the AST.

---

More practice finding redexes

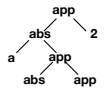
```
(λa.(λz.(+ x z))((λz.(+ x z)) a )) 2
```




33

## More practice finding redexes

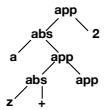
```
(λa.(λz.(+ x z))((λz.(+ x z)) a )) 2
```




34

## More practice finding redexes

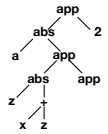
```
(λa.(λz.(+ x z))((λz.(+ x z)) a )) 2
```




35

## More practice finding redexes

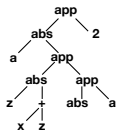
```
(λa.(λz.(+ x z))((λz.(+ x z)) a )) 2
```




36

## More practice finding redexes

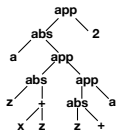
```
(λa.(λz.(+ x z))((λz.(+ x z)) a )) 2
```




37

## More practice finding redexes

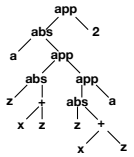
```
(λa.(λz.(+ x z))((λz.(+ x z)) a )) 2
```




38

## More practice finding redexes

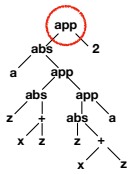
```
(λa.(λz.(+ x z))((λz.(+ x z)) a )) 2
```




39

## More practice finding redexes

```
(λa.(λz.(+ x z))((λz.(+ x z)) a )) 2
```

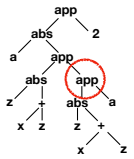



When I say *normal order*, I mean: “leftmost outermost” application

40

## More practice finding redexes

```
(λa.(λz.(+ x z))((λz.(+ x z)) a )) 2
```

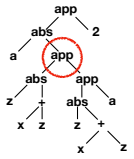



When I say *applicative order* I mean: “leftmost innermost” application

41

## More practice finding redexes

```
(λa.(λz.(+ x z))((λz.(+ x z)) a )) 2
```




Here’s a third one! Neither normal nor applicative.

42

## Solution to activity

| | |
|---|---|
| `(λa.(λz.(+ x z))((λz.(+ x z)) a)) 2` | given |
| `((λz.(+ x z))((λz.(+ x z)) 2))` | β reduce 2 for a |
| `(λz.(+ x z))((λz.(+ x z)) 2)` | eliminate parens |
| `(λb.(+ x b))((λz.(+ x z)) 2)` | α reduce b for z |
| `(λb.(+ x b))(((+ x 2)))` | β reduce 2 for z |
| `(λb.(+ x b))((+ x 2))` | eliminate parens |
| `(λb.(+ x b))(+ x 2)` | eliminate parens |
| `((+ x (+ x 2)))` | β reduce (+ x 2) for b |
| `(+ x (+ x 2))` | eliminate parens |
| | done |

Note: I chose reductions arbitrarily (it's really OK!)
Note: x is a free variable; we cannot rename it

43

Here is one possible reduction for this lambda expression. Others are also valid.

---

## Activity

Normal order reduction:

`(λf.λx.f(f x))(λz.(+ x z))2`

Normal order is "outermost leftmost" first.

44

Try doing the normal order reduction for this expression. It looks similar to the last one, but it is actually different.

---

## Activity

Applicative order reduction:

`(λf.λx.f(f x))(λz.(+ x z))2`

Applicative order is "innermost leftmost" first.

45

Try doing the applicative order reduction now.

I will post solutions to these on the course webpage.

## Recap & Next Class

### Today:

More lambda reductions

### Next class:

Higher-order functions