CSCI 334:
Principles of Programming Languages

Lecture 7: Evaluation by Rewriting

Instructor: Dan Barowy
Williams

## Topics

Lambda calculus—how to parse it

Lambda calculus—how to evaluate it

## Your to-dos

1. Lab 3, **due Sunday 10/1** (solo lab)
   Give yourself enough time to learn a small amount of $\LaTeX$
2. Read *Introduction to the Lambda Calculus, Part 2*, **for Thursday 10/5**.

## Announcements

- I added more **Office hours** on **Friday, every 10-11am** in the Ward Lab (TBL 301).
- 30 minute, 1-on-1 mentoring with TA Paul Kim
  Email Paul at pk6@williams.edu

## Announcements

- CS Colloquium this **Friday, Sept 29 @ 2:35pm in Wege Auditorium (TCL 123)**

Dzung Pham (UMass Amherst, Williams ’20)

Dzung is a second year PhD student in working in the area of security and privacy for AI/ML systems under the supervision of Prof. Amir Houmansadr. Dzung worked at Meta on algorithms to stop fraud, scams, and harassment.

Dzung is also an alum of Williams College and pursued a double major in CS and statistics. Dzung’s thesis was with Prof. Richard De Veaux in statistics, but he also worked on research projects with Profs. Bailey and Barowy (me!).

## Lambda calculus grammar

```
<expr>  ::= <var>
          |  <abs>
          |  <app>
<var>   ::= x
<abs>   ::= λ<var>.<expr>
<app>   ::= <expr><expr>
```

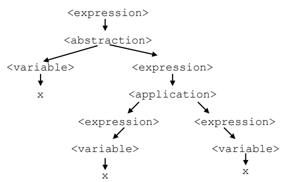
`<expr>` is the start symbol.

Here is the syntax of the lambda calculus, expressed in BNF.

## Parse tree

We can create a “derivation tree” by following the rules of a grammar as we interpret a sentence of a language.

λx.xx

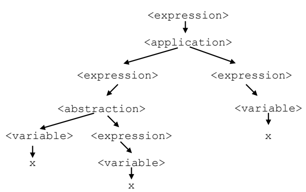



For now, though, let’s focus on derivation. What is the derivation for this lambda calculus expression?

## Ambiguity

You might have noticed that there is an alternative parse tree.

λx.xx




Note that BNF does not always capture every necessary detail. For example, here is another potential derivation for the same expression. However, this derivation is not correct because the lambda calculus includes additional rules to eliminate ambiguity. These rules are the most difficult rules for newcomers.

## Abiguity

In fact, the lambda calculus is never ambiguous because of its precedence and associativity rules—see the reading.

## Parentheses disambiguate grammar

```
<expr> = (<expr>)
```

Axiom of equivalence for parens

Let’s modify our grammar

One thing we can do is add parens to our grammar.

---

## Lambda calculus grammar

```
<expr>   ::= <var>
           |  <abs>
           |  <app>
           |  <parens>
<var>    ::= x
<abs>    ::= λ<var>.<expr>
<app>    ::= <expr><expr>
<parens> ::= (<expr>)
```

---

## While we’re at it…

```
<expr>   ::= <var>
           |  <abs>
           |  <app>
           |  <parens>
<var>    ::= α ∈ { a ... z }
<abs>    ::= λ<var>.<expr>
<app>    ::= <expr><expr>
<parens> ::= (<expr>)
```

Also, it is very helpful to have variables other than x.

## Also…

```
<expr>   ::= <value>
           |  <abs>
           |  <app>
           |  <parens>
<var>    ::= α ∈ { a ... z }
<abs>    ::= λ<var>.<expr>
<app>    ::= <expr><expr>
<parens> ::= (<expr>)
<value>  ::= v ∈ ℕ
           |  <var>
```

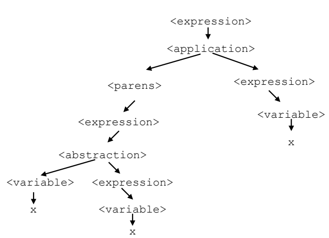
Finally, we will sometimes add arbitrary literal values to the lambda calculus. These are not strictly necessary, but they make working with the language a little easier.

## Class Lambda Grammar

```
<expr>   ::= <value>
           |  <abs>
           |  <app>
           |  <parens>
<var>    ::= α ∈ { a ... z }
<abs>    ::= λ<var>.<expr>
<app>    ::= <expr><expr>
<parens> ::= (<expr>)
<value>  ::= v ∈ ℕ
           |  <var>
```

## This expression is now unambiguous

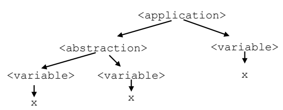
(λx.x)x




With parens, our original expression is unambiguous.

## However, this is the parse tree we really care about

(λx.x)x




Eventually, you will see that what we really care about is the abstract syntax tree.

---

## Free vs bound variables

(λx.x)x

bound

free

One very important aspect of the lambda calculus is whether a variable is "free" or "bound." This expression has two different x variables in it. Be on the lookout for this distinction.

---

## Lambda calculus: relevance

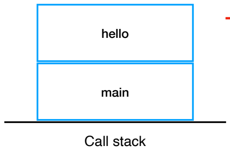
**Fundamental technique** for building programming languages that work **correctly** (and **intuitively**!).

But it can also be leveraged to do some **seemingly magical** things, like **type inference**:

```
Vector<Association<String,FrequencyList>> table =
   new Vector<Association<String,FrequencyList>>();

Vector<Association<String,FrequencyList>> table = new Vector<>();

let table = new Vector<>()

…
```

Why are we learning this? At its heart, the study of programming languages is about how a language "desugars" into a core mathematical idea. You do not *need* the lambda calculus to build a programming language. However, unless you understand the relationship between your language and the lambda calculus, certain kinds of insights about programs will be difficult or impossible to obtain.

## Evaluation: You know how Java does it




```
class Program {
  public static void hello() {
    println("Hello world!");
  }

  public static void main(…) {
    hello();
  }
}
```

Now, let's talk about how a program is evaluated. You might have some sense of how some languages are evaluated, like Java. C works essentially the same way as Java in this regard.

## Evaluation: Lambda calculus is like algebra

`(λx.x)x`

Evaluation consists of simplifying an expression using text substitution.

Only two simplification rules:

**α-reduction**

**β-reduction**

However, the lambda calculus is different. It is more like algebra.

## α-Reduction

`(λx.x)x`

This expression has two **different** `x` variables

Which should we rename?

Rule:

`⟦λx.<expr>⟧ =α ⟦λy.[y/x]<expr>⟧`

`[y/x]<expr>` means "substitute `y` for `x` in `<expr>`"

There are several "evaluation rules" in the lambda calculus. We call these rules "reductions." The first is alpha reduction, which is used to rename a variable in an expression.

## α-Reduction

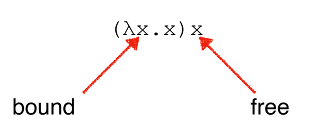
| | |
|---|---|
| `(λx.x)x` | given |
| `(λy.[y/x]x)x` | α-reduce y for x (binding) |
| `(λy.y)x` | α-reduce y with x (expr) |

For example, we can alpha reduce the expression (λx.x)x to (λy.y)x. This is OK because we’re just renaming a bound variable. Your intuition may already tell you that this is OK! For example, you probably already know that the following two Java programs are the same.

```
public static int id(int x) {
  return x;
}

public static int id(int y) {
  return y;
}
```

---

## Free vs bound variables




Note that there is a very important distinction between free and bound variables. The inner (leftmost) x is defined by the abstraction. The outer (rightmost) x is a TOTALLY DIFFERENT VARIABLE that happens to have the same name. We do not know how it is defined in this expression, so we must treat it with caution.

## Watch out!

| | |
|---|---|
| λx.xy | given |
| λy.[y/x]xy | α-reduce y for x |
| λy.yy | inner α-reduction |
| | **this is incorrect!** |

The lambda has “captured” the free y.
Substitution must be **capture-avoiding**.

Be careful not to “capture” a variable when performing an alpha reduction.

---

## β-Reduction

(λx.x)y

How we “call” or **apply** a function to an argument

Rule:

⟦(λx.<expr>)y⟧ $=_\beta$ ⟦[y/x]<expr>⟧

The second reduction rule is beta reduction, which has essentially the same meaning as a “function call.”  It passes an argument into a function definition, discards the lambda, and then rewrites the body of the function definition.

---

## Let’s reduce this

(λx.x)x

For example, let’s reduce this expression.  The result is ultimately x.

## Watch out!

| | |
|---|---|
| (λx.λx.x) x | given |
| ([x/x] λx.x) | β-reduce x for x |
| (λx.x) | β-reduce inner expr |
| | done |

The inner lambda term **redefines** x and therefore “blocks” substitution of x.

Not only do we want to avoid capturing variables, we must also make sure only to substitute as far as makes sense. Here, the inner lambda redefines x, so we must stop after substituting the first one. This is clearer if you do an alpha reduction first!

---

## How far do we go?

We keep going until there is **nothing left to simplify**.

| | |
|---|---|
| x | ← done |
| xx | ← done |
| λx.y | ← done |
| (λx.xy) z | ← not done |

That “most simplified” expression is called a **normal form**.

An expression that can be simplified is a called a **redex**.

How do we know when to stop evaluating? The answer is when no redexes remain.

---

## Try this one with a partner

(λx.λy.yx) xy

(don’t forget precedence/associativity rules)

## Recap & Next Class

**Today:**

Lambda calculus: how to parse

Lambda calculus: how to evaluate

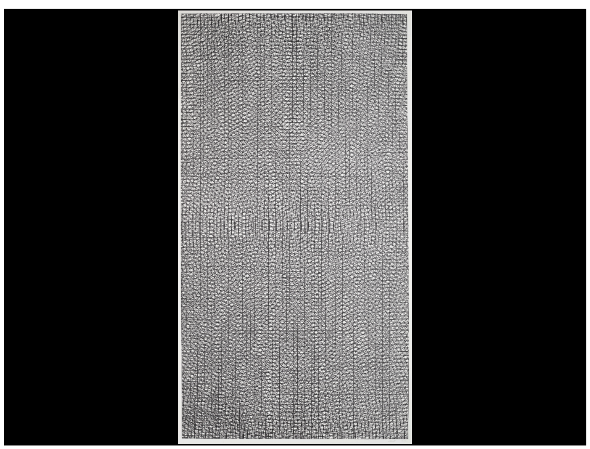
**Next class:**

Lambda calculus: how to survive

---

## Final projects

---



Final project idea. Start thinking about this. This could obviously be drawn by a computer. Could you make a language for a non-programmer artist to draw it? Could it be a “joy” for that artist to use?