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. Read *Syntax* and *Introduction to the Lambda Calculus, Part 1*, **for Lab 4**.
2. Lab 4, **due Wednesday 3/5** (solo lab)

Announcements

- CS Colloquium this **Friday, Feb 28 @ 2:35pm in Wege Auditorium (TCL 123)**

Senior Thesis Proposals, Part 2

λ Syntax Refresher

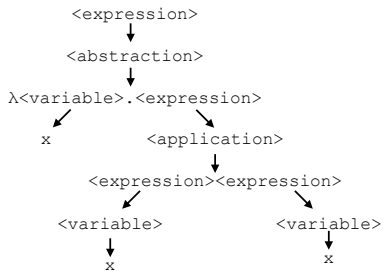
Lambda calculus grammar

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

Parsing Lambda Expressions

Let's try parsing this expression

λx.xx

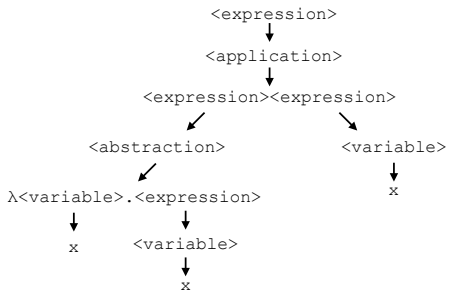


NB: this same slide from last class was correct, but made the **process** less clear.
Use this slide as a reference for your homework.

Ambiguity

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

λx.xx



We **do not accept** this parse due to other rules…

Disambiguation

The lambda calculus is **never ambiguous** because of its **precedence** and **associativity** rules.

element	precedence	associativity
application: `<expr><expr>`	high	left
abstraction: `λ<var>.<expr>`	low	right
variable: `x`	n/a	n/a

Higher precedence means “parses first.”

Example

Let’s derive this one.

`λx.xxλx.xx`

When in doubt, add parens

`<expr> = (<expr>)`

Axiom of equivalence for parens

Let’s modify our grammar

Lambda calculus grammar

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

Disambiguation

Parens have the **highest** precedence.

element	precedence	associativity
parens: `(<expr>)`	highest	n/a
application: `<expr><expr>`	high	left
abstraction: `λ<var>.<expr>`	low	right
variable: `x`	n/a	n/a

While we're at it…

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

Also…

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

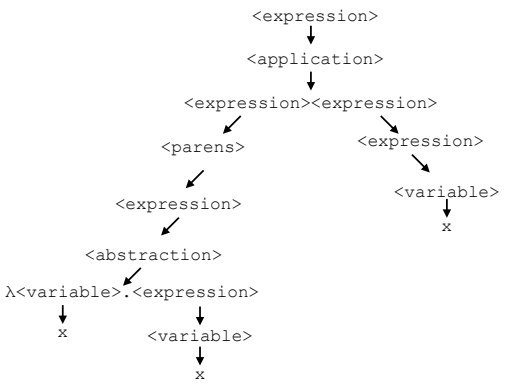
Memorize This†

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

†I rarely ask students to memorize things.

This expression is totally unambiguous

(λx.x) x

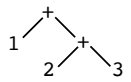


Abstract Syntax Tree

Ignores derivation details; only **essential structure**

```
<e> ::= <n> | <e>+<e> | <e>-<e>
<n> ::= <d> | <n><d>
<d> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

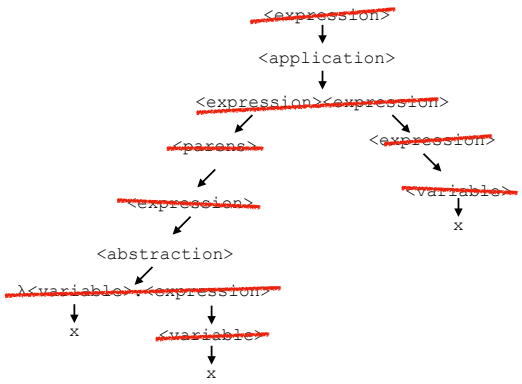
1+2+3



In an **AST**, internal nodes are **operations**, leaves are **data**.

Abstract syntax tree

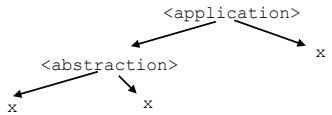
(λx.x) x



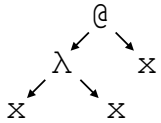
Start with a derivation and **remove** anything unrelated to **data** or **operations**.

Abstract syntax tree

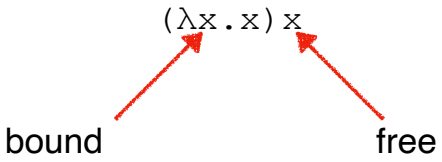
(λx.x) x



I use a lazy shorthand…



Free vs bound variables



Let's parse this together

```
xλy.a(xx)
```

Evaluating λ expressions

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**:

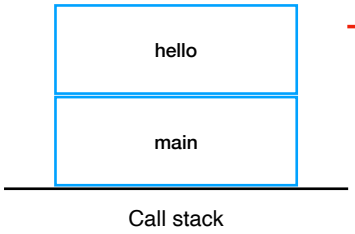
```
Vector<Association<String,FrequencyList>> table =
   new Vector<Association<String,FrequencyList>>();

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

let table = new Vector<>()

…
```

Evaluation:
You (might) know how Java does it




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

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

Evaluation:
Lambda calculus is like **algebra**

$(\lambda x.x)\,x$

Evaluation consists of simplifying an expression using **text substitution**.

Only two simplification rules:

α-reduction

β-reduction

α-Reduction

$(\lambda x.x)\,x$

This expression has two **different** x variables

Which should we rename?

Rule:

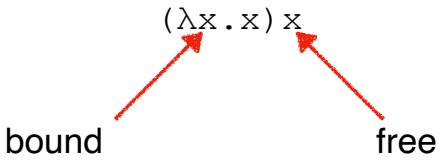
$[\![\lambda x.\langle expr\rangle]\!] =_\alpha [\![\lambda y.[y/x]\langle expr\rangle]\!]$

$[y/x]\langle expr\rangle$ means "substitute y for x in $\langle expr\rangle$"

α-Reduction

$(\lambda \mathbf{x}.\mathbf{x})\,x$	given
$(\lambda \mathbf{y}.[\mathbf{y}/\mathbf{x}]\mathbf{x})\,x$	α-reduce **y** for **x** (binding)
$(\lambda \mathbf{y}.\mathbf{y})\,x$	α-reduce **y** with **x** (expr)

Free vs bound variables



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**.

Recap & Next Class

Today:

Lambda calculus: how to parse

Lambda calculus: how to evaluate

Next class:

Lambda calculus: how to survive