CSCI 334:
Principles of Programming Languages

Lecture 6: PL Fundamentals II

Instructor: Dan Barowy

Williams

Announcements

Announcements

HW4 posted

Announcements

HW4 posted

Wednesday Office Hours

Recursive Data Structures

Recursive Data Structures

You learned these in CS136.

Recursive Data Structures

You learned these in CS136.

Let's talk about the list node in HW3.

Recursive Data Structures

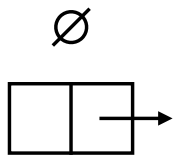
You learned these in CS136.

Let's talk about the list node in HW3.

Recursive Data Structures

You learned these in CS136.

Let's talk about the list node in HW3.



Activity

Activity

Write a function `swap` that swaps the values of `x` and `y`.

Activity

Write a function `swap` that swaps the values of `x` and `y`.

Start by drawing a picture of what you want.

Activity

Write a function `swap` that swaps the values of `x` and `y`.

Start by drawing a picture of what you want.

```
#include <stdio.h>

void swap(int *a, int *b) {
  ???
}

int main() {
  int x = 1;
  int y = 2;
  swap(&x, &y);
  printf("x = %d, y = %d\n", x, y);
  return 0;
}
```

Lambda calculus grammar

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

What is a variable?

```
<var>   ::= x
```

What is a variable?

```
<var>   ::= x
```

It's just a value.

What is an abstraction?

```
<abs>   ::= λ<var>.<expr>
```

What is an abstraction?

```
<abs>   ::= λ<var>.<expr>
```

It's a function definition

What is an abstraction?

```
<abs>   ::= λ<var>.<expr>
```

It's a function definition

```
def foo(x):
```

What is an abstraction?

```
<abs>   ::= λ<var>.<expr>
```

It's a function definition

```
def foo(x):
  <expr>
```

What is an application?

```
<app>    ::= <expr><expr>
```

What is an application?

```
<app>    ::= <expr><expr>
```

It's a "function call"

What is an application?

```
<app>    ::= <expr><expr>
```

It's a "function call"

```
foo(2)
```

What is an application?

```
<app>    ::= <expr><expr>
```

It's a "function call"

```
foo(2)
          <expr><expr>
```

What is an application?

```
<app>    ::= <expr><expr>
```

It's a "function call"

```
foo(2)
```

```
<expr><expr>
```

function

What is an application?

```
<app>    ::= <expr><expr>
```

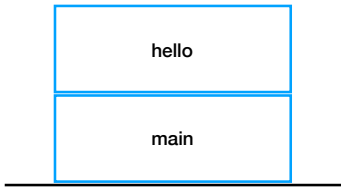
It's a "function call"

```
foo(2)
```

```
<expr><expr>
```

function argument

Evaluation: You know how C does it



Call stack

```
#include <stdio.h>

void hello() {
  printf("Hello world!\n");
}

int main() {
  hello();
  return 0;
}
```

Evaluation: Lambda calculus is like algebra

```
(λx.x)x
```

Evaluation: Lambda calculus is like algebra

`(λx.x)x`

Evaluation consists of simplifying an expression using text substitution.

Evaluation: Lambda calculus is like algebra

`(λx.x)x`

Evaluation consists of simplifying an expression using text substitution.

Only two simplification rules:

Evaluation: Lambda calculus is like algebra

`(λx.x)x`

Evaluation consists of simplifying an expression using text substitution.

Only two simplification rules:

α-reduction

Evaluation: Lambda calculus is like algebra

`(λx.x)x`

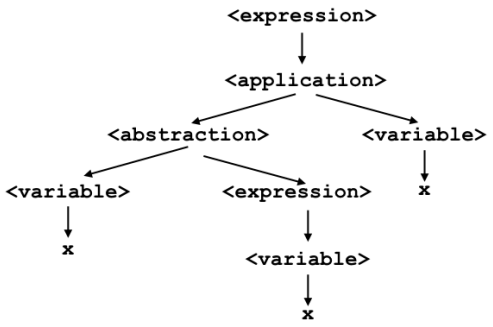
Evaluation consists of simplifying an expression using text substitution.

Only two simplification rules:

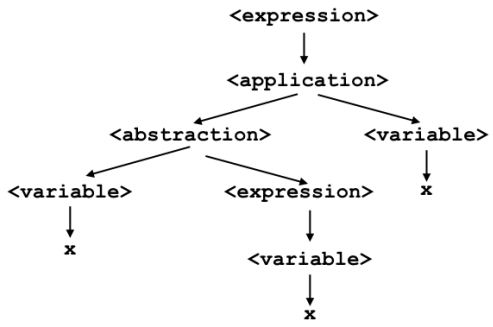
α-reduction

β-reduction

Ambiguity

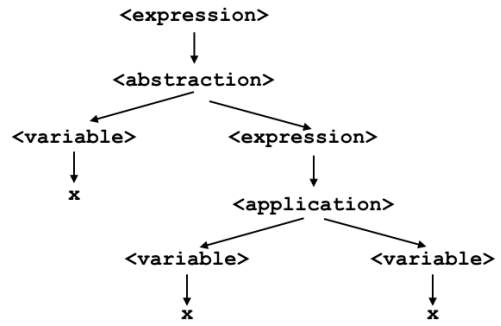


Ambiguity

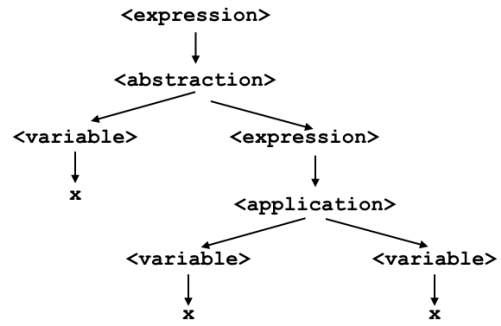


If we generate expr, what do we get?

Ambiguity



Ambiguity



This is not what we started with!

Parentheses disambiguate grammar

Parentheses disambiguate grammar

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

Parentheses disambiguate grammar

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

Axiom of equivalence for parens

Parentheses disambiguate grammar

```
<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>)
```

While we're at it…

While we're at it…

```
<expr>   ::= <var>
```

While we're at it…

```
<expr>   ::= <var>
           |  <abs>
           |  <app>
```

While we're at it…

```
<expr>   ::= <var>
           | <abs>
           | <app>
           | <parens>
```

While we're at it…

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

While we're at it…

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

While we're at it…

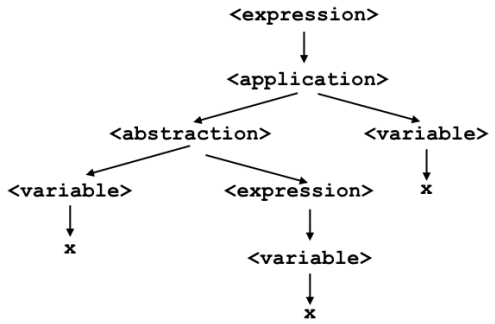
```
<expr>   ::= <var>
           | <abs>
           | <app>
           | <parens>
<var>    ::= α ∈ { a ... z }
<abs>    ::= λ<var>.<expr>
<app>    ::= <expr><expr>
<parens> ::= (<expr>)
```

Also…

```
<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

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



Free vs bound variables

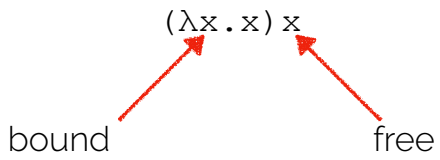
$(\lambda x.x)\,x$

Free vs bound variables

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

bound

Free vs bound variables



α-Reduction

(λx.x) x

α-Reduction

(λx.x) x

This expression has two *different* x variables

α-Reduction

(λx.x) x

This expression has two *different* x variables

Which should we rename?

α-Reduction

`(λx.x)x`

This expression has two *different* `x` variables

Which should we rename?

Rule:

α-Reduction

`(λx.x)x`

This expression has two *different* `x` variables

Which should we rename?

Rule:

`λx.<expr>` $=_\alpha$ `λy.[y/x]<expr>`

α-Reduction

`(λx.x)x`

This expression has two *different* `x` variables

Which should we rename?

Rule:

`λx.<expr>` $=_\alpha$ `λy.[y/x]<expr>`

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

α-Reduction

`(λx.x)x`

α-Reduction

(λx.x)x

(λy.[y/x]x)x

α-Reduction

(λx.x)x

(λy.[y/x]x)x

(λy.y)x

β-Reduction

(λx.x)y

β-Reduction

(λx.x)y

How we "call" or *apply* a function to an argument

β-Reduction

`(λx.x)y`

How we "call" or *apply* a function to an argument

Rule:

β-Reduction

`(λx.x)y`

How we "call" or *apply* a function to an argument

Rule:

$(\lambda x.\langle expr\rangle)y =_\beta [y/x]\langle expr\rangle$

Reduce this

`(λx.x)x`

How far do we go?

How far do we go?

We keep going until there is nothing left to do

How far do we go?

We keep going until there is nothing left to do

x

How far do we go?

We keep going until there is nothing left to do

x ← done

How far do we go?

We keep going until there is nothing left to do

x ← done

xx

How far do we go?

We keep going until there is nothing left to do

`x` ← done

`xx` ← done

How far do we go?

We keep going until there is nothing left to do

`x` ← done

`xx` ← done

`λx.y`

How far do we go?

We keep going until there is nothing left to do

`x` ← done

`xx` ← done

`λx.y` ← done

How far do we go?

We keep going until there is nothing left to do

`x` ← done

`xx` ← done

`λx.y` ← done

`(λx.xy)z`

How far do we go?

We keep going until there is nothing left to do

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

How far do we go?

We keep going until there is nothing left to do

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

That "most simplified" expression is called a *normal form*.

Sometimes multiple simplifications

Order (mostly) does not matter

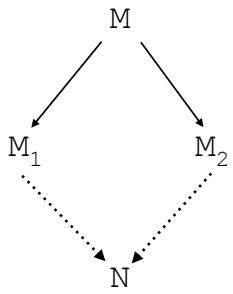
Sometimes multiple simplifications

Order (mostly) does not matter

If $M \rightarrow M_1$ and $M \rightarrow M_2$
then $M_1 \rightarrow^* N$ and $M_2 \rightarrow^* N$
for some N

Sometimes multiple simplifications

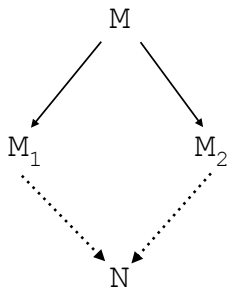
Order (mostly) does not matter



If $M \rightarrow M_1$ and $M \rightarrow M_2$
then $M_1 \rightarrow^* N$ and $M_2 \rightarrow^* N$
for some N

Sometimes multiple simplifications

Order (mostly) does not matter



If $M \rightarrow M_1$ and $M \rightarrow M_2$
then $M_1 \rightarrow^* N$ and $M_2 \rightarrow^* N$
for some N

"confluence"

Example

$(\lambda f.\lambda x.f(fx))(\lambda y.y)2$