CSCI 334:
Principles of Programming Languages

Lecture 6: PL Fundamentals II

Instructor: Dan Barowy

Williams

Announcements

HW4 posted

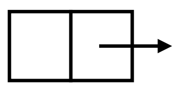
Wednesday Office Hours

Recursive Data Structures

You learned these in CS136.

Let's talk about the list node in HW3.

∅



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
```

It's just a value.

What is an abstraction?

```
<abs>    ::= λ<var>.<expr>
```

It's a function definition

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

What is an application?

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

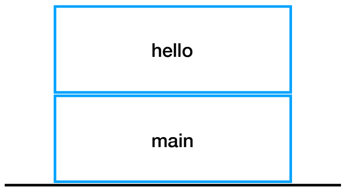
It's a “function call”

```
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

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

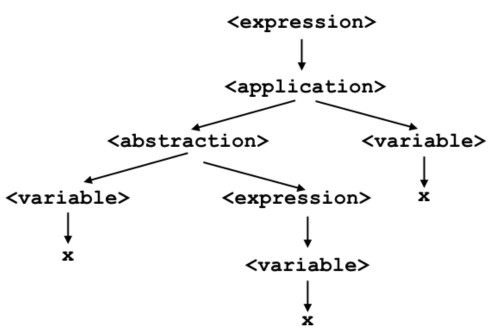
Evaluation consists of simplifying an expression using text substitution.

Only two simplification rules:

α-reduction

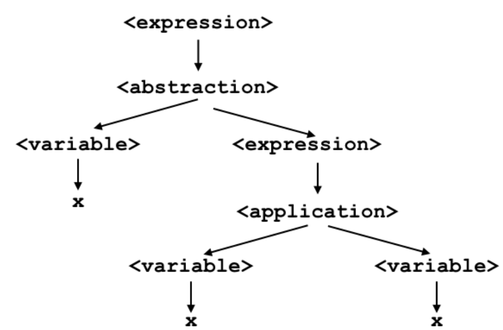
β-reduction

Ambiguity



If we generate expr, what do we get?

Ambiguity



This is not what we started with!

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…

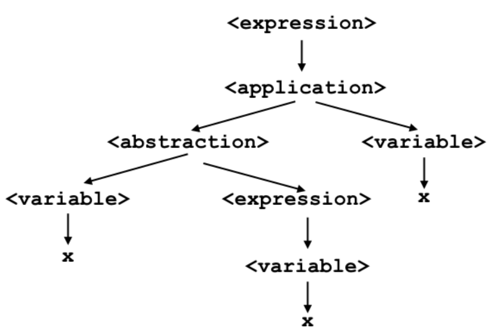
```
<expr>   ::= <var>
           |  <abs>
           |  <app>
           |  <parens>
<var>    ::= α ∈ { a ... z }
<abs>    ::= λ<var>.<expr>
<app>    ::= <expr><expr>
<parens> ::= (<expr>)
```

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

(λx.x)x



Free vs bound variables

(λx.x)x

bound

free

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

(λy.[y/x]x)x

(λy.y)x

β-Reduction

(λx.x)y

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

Rule:

(λx.<expr>)y $=_\beta$ [y/x]<expr>

Reduce this

(λx.x)x

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

$$\begin{tikzcd} & M \arrow[dl] \arrow[dr] & \\ M_1 \arrow[dr, dotted] & & M_2 \arrow[dl, dotted] \\ & N & \end{tikzcd}$$

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

“confluence”

Example

`(λf.λx.f(fx))(λy.y)2`