CS 320: Principles of Programming Languages

Katie Casamento, Portland State University

Fall 2018
Week 6: Untyped Lambda Calculus

Semantics

We can turn text into ASTs, but what can we do with ASTs?

- *Denotational semantics*: define a *function* in some metalanguage to produce some "meaning" object in the metalanguage from an AST
  - A Haskell function evaluates Prop ASTs to Haskell `Bool`s
  - A set-theoretic function evaluates Regex ASTs to sets
  - A Haskell function evaluates Regex ASTs to Haskell `String`-matching functions
- *Operational semantics*: interpret an AST as some kind of *state machine*
  - A term-rewriting machine normalizes a `Prop` AST
  - A DFA matches an input string against a regex
- *Axiomatic semantics*: describe a program's behavior in terms of *logical formulas* about program states
  - `A & B = B & A`
  - `r+ = r • r*`

Programming language semantics

In order to study programming language semantics, we need a programming language to give semantics to!

It should be:

- Minimal: few constructs, for implementation simplicity
- Turing-complete: able to encode any computable function
- Extensible: possible to add new features without modifying old features

Untyped lambda calculus

Untyped lambda calculus

- Introduced by Alonzo Church[1] in the 1930s as a formalization of *computable functions*
- Proven Turing-complete by Alan Turing[2] in 1936
- Basis for more complex calculi
  - Simply-typed lambda calculus: Alonzo Church[3], 1940
  - System F (polymorphic types): Jean-Yves Girard[4], 1972 and independently John C. Reynolds[5], 1974.
  - Intuitionistic type theory (dependent types): Per Martin-Löf[6], 1972
- Foundation of *dynamically-typed* functional programming languages

1. "A Set of Postulates for the Foundation of Logic"
2. "On Computable Numbers, with an Application to the Entscheidungsproblem"
3. "A Formulation of the Simple Theory of Types"
4. "Une Extension de l'Interpretation de Gödel à l'Analyse, et son Application à l'Élimination des Coupures dans l'Analyse et la Théorie des Types"
5. "Towards a theory of type structure"
6. "Intuitionistic Type Theory"

Concrete lambda calculus syntax

Where `x` is a member of some set of *names* (usually strings):

```
E → x
E → (λx. E)
E → (E E)
```

Or more concisely, in EBNF:

$e ::= x \mid (\lambda x.\ e) \mid (e_1\ e_2)$

Lambda calculus semantics

What does a lambda calculus expression mean?

- `x` is a variable reference
- `(λx. e)` is a function with argument `x` and return expression `e`
- $(e_1\ e_2)$ is an application of a function e_1 to an argument e_2

In pure lambda calculus, functions are the only kind of data!

- Lambda expressions in Python:
  "(λx. e)" = "lambda x: e"
  "$(e_1\ e_2)$" = "$e_1(e_2)$"
- Lambda expressions in Haskell:
  "(λx. e)" = "\x -> e"
  "$(e_1\ e_2)$" = "$e_1\ e_2$"

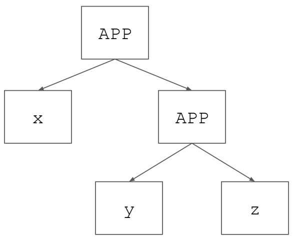
Some lambda calculus expressions

```
x
(x y)
((x y) z)
(x (y z))
(λx. y)
((λx. y) z)
(λx. (λy. x))
(λx. (x x)) (λx. (x x))
(λf. (λx. (f (x x))) (λx. (f (x x))))
```

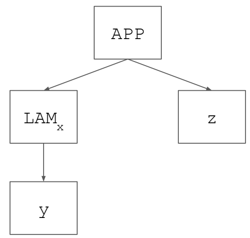
$e ::= x \mid (\lambda x.\ e) \mid (e_1\ e_2)$

Some lambda calculus ASTs

`(x (y z))`



`((λx. y) z)`



$e ::= x \mid (\lambda x.\ e) \mid (e_1\ e_2)$

Syntactic sugar

A couple common shorthand forms for writing expressions:

- Lambdas associate to the right
  - `(λx. λy. z) = (λx. (λy. z))`
- Lambdas can have multiple arguments
  - `(λx y z. e) = (λx. (λy. (λz. e)))`
- Application associates to the left
  - `(x y z w) = (((x y) z) w)`
- Outermost parentheses in an expression can be dropped
  - `x (y z) = (x (y z))`
  - `λx. (y z) = (λx. (y z))`
- Outermost parentheses in the body of a lambda can be dropped
  - `λx. y z w = (λx. ((y z) w))`
- **No other parentheses** can be dropped!

$e ::= x \mid (\lambda x.\ e) \mid (e_1\ e_2)$

Operational semantics

We can define evaluation of expressions as a *term-rewriting machine*.

$e_1 \Rightarrow e_2$ means e_1 can be rewritten to e_2 in one step
$e_1 \Rightarrow^* e_2$ means e_1 can be rewritten to e_2 in zero or more steps

- This is a *small-step semantics*: it describes the behavior of each step of the machine
- Rules are usually written in concrete syntax for readability, but we're really operating over ASTs
- In the context of lambda calculus, rewriting is usually called *reduction*
- A subexpression that can be reduced is called a *redex (reducible expression)*

$e ::= x \mid (\lambda x.\ e) \mid (e_1\ e_2)$

Operational semantics (nondeterministic)

$$\frac{e_1 \Rightarrow e'_1}{(e_1\ e_2) \Rightarrow (e'_1\ e_2)}$$

if e_1 reduces to e'_1 in one step

then $(e_1\ e_2)$ reduces to $(e'_1\ e_2)$ in one step

$$\frac{e_2 \Rightarrow e'_2}{(e_1\ e_2) \Rightarrow (e_1\ e'_2)}$$

if e_2 reduces to e'_2 in one step

then $(e_1\ e_2)$ reduces to $(e_1\ e'_2)$ in one step

Operational semantics (nondeterministic)

$$\frac{}{(\lambda x.\ e_1)\ e_2 \Rightarrow e_1[e_2/x]}$$

(under any condition)

(λx. e_1) e_2 reduces to $e_1[e_2/x]$ in one step

$e_1[e_2/x]$ = "e_1 with every occurrence of x replaced with e_2"

This action is called *substitution*, and the reduction rule is called *β-reduction*.

Operational semantics (nondeterministic)

$$\text{E-AppL}\quad \frac{e_1 \Rightarrow e'_1}{(e_1\ e_2) \Rightarrow (e'_1\ e_2)}$$

$$\text{E-AppR}\quad \frac{e_2 \Rightarrow e'_2}{(e_1\ e_2) \Rightarrow (e_1\ e'_2)}$$

$$\text{E-Beta}\quad \frac{}{(\lambda x.\ e_1)\ e_2 \Rightarrow e_1[e_2/x]}$$

- Evaluation is finished when there's no rule left to apply
- This semantics is *nondeterministic* because there may be more than one rule that applies to the same expression
- The action of substitution is sometimes written as a reduction rule
- These are **all of the reduction rules** in this semantics!

Some lambda calculus reductions

```
(λx. x) y                ⇒* y
(λx. x x) (λy. y)        ⇒* λy. y
(λx y. x) a b            ⇒* a
(λf g x. f (g x)) a b c ⇒* a (b c)
```


Some lambda calculus reductions

```
(λx. x) y ⇒* y

(λx. x) y ⇒ x[y/x]
⇒ y
```

(redexes highlighted with function in pink, function body with grey background and argument in blue)

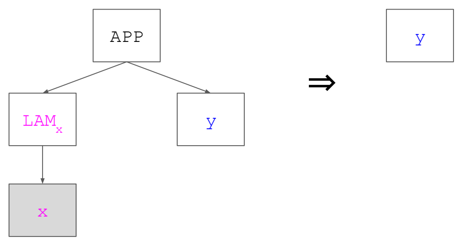
Substitution steps are often left implicit:

```
(λx. x) y ⇒ y
```

$e ::= x \mid (\lambda x.\ e) \mid (e_1\ e_2)$

Some lambda calculus reductions

```
(λx. x) y ⇒* y
```



$e ::= x \mid (\lambda x.\ e) \mid (e_1\ e_2)$

Some lambda calculus reductions

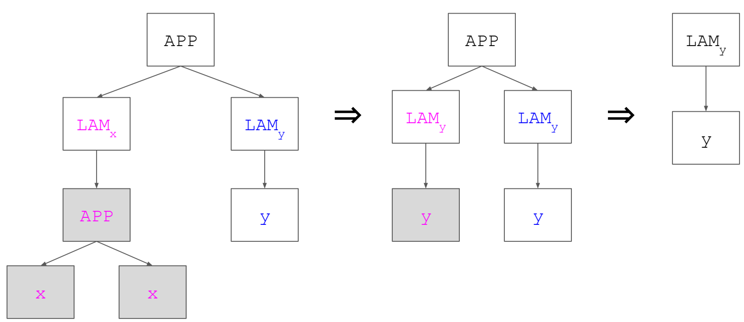
```
  (λx. x x) (λy. y) ⇒* λy. y

  ((λx. x x) (λy. y)) ⇒ (x x)[(λy. y)/x]
⇒ ((λy. y) (λy. y)) ⇒ y[(λy. y)/y]
⇒ λy. y
```

$e ::= x \mid (\lambda x.\ e) \mid (e_1\ e_2)$

Some lambda calculus reductions

```
(λx. x x) (λy. y) ⇒* λy. y
```



$e ::= x \mid (\lambda x.\ e) \mid (e_1\ e_2)$

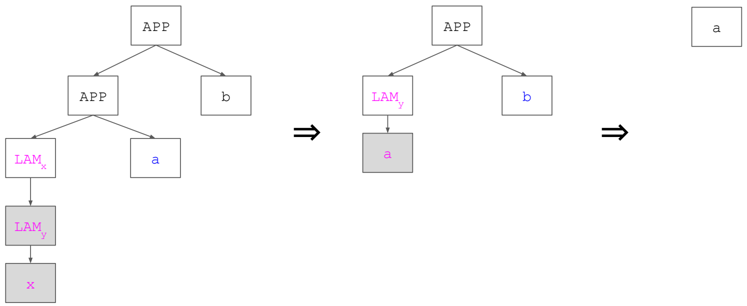
Some lambda calculus reductions

```
  (λx y. x) a b ⇒* a

  ((λx (λy. x)) a) b ⇒ ((λy. x)[a/x]) b
⇒ (λy. a) b ⇒ a[b/y]
⇒ a
```

Some lambda calculus reductions

`(λx y. x) a b ⇒* a`



Some lambda calculus reductions

`(λf g x. f (g x)) a b c ⇒* a (b c)`

```
  (((λf. (λg. (λx. f (g x)))) a) b) c ⇒ ((λg. (λx. f (g x)))[a/f]) b) c
⇒ ((λg. (λx. a (g x))) b) c ⇒ ((λx. a (g x))[b/g]) c
⇒ (λx. a (b x)) c ⇒ (a (b x))[c/x]
⇒ a (b c)
```


Proof trees

Formally, to show that one expression reduces to another, we combine reduction rules into a *proof tree*.

$$\text{E-AppL}\quad \frac{e_1 \Rightarrow e'_1}{(e_1\ e_2) \Rightarrow (e'_1\ e_2)}$$

$$\text{E-AppR}\quad \frac{e_2 \Rightarrow e'_2}{(e_1\ e_2) \Rightarrow (e_1\ e'_2)}$$

$$\text{E-Beta}\quad \frac{}{(\lambda x.\ e_1)\ e_2 \Rightarrow e_1[e_2/x]}$$

`(((λx. x) y) z) w ⇒ (y z) w`

$$\text{E-AppL}\quad \frac{\text{E-AppL}\quad \dfrac{\text{E-Beta}\quad \dfrac{}{(\lambda x.\ x)\ y \Rightarrow y}}{((\lambda x.\ x)\ y)\ z \Rightarrow y\ z}}{(((\lambda x.\ x)\ y)\ z)\ w \Rightarrow (y\ z)\ w}$$

Bound variables and free variables

- A variable is *bound* when it occurs in the body of a lambda expression whose argument has the same name as the variable
  - The expression that a variable is bound in is the variable's *scope*
- A variable is *free* when it's not bound
- An expression is *closed* when it has no free variables
  - A closed expression is sometimes called a *combinator*

- `a b`
- `(λx. x y)`
- `(λx. x) a`
- `(λx. x) x`

Variable capture

When substituting, we have to be careful to avoid *capturing* free variables - that is, turning a free variable into a bound variable.

For example, this should hold for any expressions e_1 and e_2:

$$(\lambda f\ x.\ f\ x)\ e_1\ e_2 \Rightarrow^* e_1\ e_2$$

But there's a problem:

$$(\lambda f.\ (\lambda x.\ f\ x))\ x\ y = ((\lambda x.\ f\ x)[x/f])\ y$$
$$\Rightarrow (\lambda x.\ x\ x)\ y = (x\ x)[y/x]$$
$$\Rightarrow y\ y$$

This is the wrong result - we should have gotten $(x\ y)$!

$e ::= x \mid (\lambda x.\ e) \mid (e_1\ e_2)$

Variable capture

What went wrong?

$$(\lambda f\ (\lambda x.\ f\ x))\ x\ y = ((\lambda x.\ f\ x)[x/f])\ y$$
$$\Rightarrow (\lambda x.\ x\ x)\ y = (x\ x)[y/x]$$
$$\Rightarrow y\ y$$

The free x turned into a bound x in the second step!

In C, this problem might look something like:

```
int x = 8;
int f(int x, int y) { return x + y; }
f(3, x) // should return 11, but might return 6 in an incorrect implementation
```

$e ::= x \mid (\lambda x.\ e) \mid (e_1\ e_2)$

Capture-avoiding substitution

A common solution is to restrict substitution:

$e_1[e_2/x]$ = "e_1 with every occurrence of x replaced with e_2
iff x is not free in e_2 and no free variables in e_2 are bound in e_1"

Along with a rule for *renaming* (*α-conversion*):

$$\frac{y \notin \mathrm{free}(e)}{(\lambda x.\ e) \Rightarrow (\lambda y.\ e[y/x])}$$

if y is not free in e

then $(\lambda x.\ e)$ reduces to $(\lambda y.\ e[y/x])$ in one step

$e ::= x \mid (\lambda x.\ e) \mid (e_1\ e_2)$

Capture-avoiding substitution

In the first step of this example, we can't immediately substitute x for f, because x is not free in $(\lambda x.\ f\ x)$.

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

So we have to rename the inner lambda first:

$$(\lambda f.\ (\lambda x.\ f\ x))\ x\ y \Rightarrow (\lambda f.\ (\lambda a.\ f\ a))\ x\ y \Rightarrow ((\lambda a.\ f\ a)[x/f])\ y$$
$$\Rightarrow (\lambda a.\ x\ a)\ y \Rightarrow (x\ a)[y/a]$$
$$\Rightarrow x\ y$$

Now we get the correct result!

$e ::= x \mid (\lambda x.\ e) \mid (e_1\ e_2)$

Termination

Are there any expressions that **never finish reducing**?

$e ::= x \mid (\lambda x.\ e) \mid (e_1\ e_2)$

Nontermination

Are there any expressions that **never finish reducing**?

The language is Turing-complete, so there should be!

This is the simplest one, sometimes called the *omega combinator*.

$$\Omega = (\lambda\ x.\ x\ x)\ (\lambda x.\ x\ x)$$

$$\begin{aligned} & (\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \Rightarrow (x\ x)[(\lambda x.\ x\ x)/x] \\ \Rightarrow\ & (\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \Rightarrow (x\ x)[(\lambda x.\ x\ x)/x] \\ \Rightarrow\ & \ldots \end{aligned}$$

When an expression never finishes reducing, we say it *diverges*.

$e ::= x \mid (\lambda x.\ e) \mid (e_1\ e_2)$

Values

A *value* (or *normal form*) is an expression that is "finished" being reduced to some sensible meaning.

$$v ::= x \mid (\lambda x.\ e) \mid x\ v_1 \ldots v_n$$

In pure untyped lambda calculus, the only kind of values are **free variables**, **functions**, and **applications of a free variable to some number of values**.

Ideally, an expression should always either reduce to a value or diverge.

This means a closed term (one without free variables) should **always reduce to a function**.

In pure untyped lambda calculus, these properties hold!

$e ::= x \mid (\lambda x.\ e) \mid (e_1\ e_2)$
$v ::= x \mid (\lambda x.\ e) \mid x\ v_1 \ldots v_n$

Reduction order

We've been using a nondeterministic semantics so far, where there may be a choice of more than one reduction step for an expression.

Does it matter which reduction we choose?

$e ::= x \mid (\lambda x.\ e) \mid (e_1\ e_2)$
$v ::= x \mid (\lambda x.\ e) \mid x\ v_1 \ldots v_n$

Reduction order

We've been using a nondeterministic semantics so far, where there may be a choice of more than one reduction step for an expression.

Does it matter which reduction we choose?

$(\lambda x\ y.\ y)\ \Omega\ z$

If we **reduce the arguments** first:

$(\lambda x\ y.\ y)\ \Omega\ z \Rightarrow (\lambda x\ y.\ y)\ \Omega\ z \Rightarrow (\lambda x\ y.\ y)\ \Omega\ z \Rightarrow \ldots$

Ω never finishes reducing, so the whole expression diverges.

$e ::= x \mid (\lambda x.\ e) \mid (e_1\ e_2)$
$v ::= x \mid (\lambda x.\ e) \mid x\ v_1 \ldots v_n$

Reduction order

We've been using a nondeterministic semantics so far, where there may be a choice of more than one reduction step for an expression.

Does it matter which reduction we choose?

$(\lambda x\ y.\ y)\ \Omega\ z$

If we **substitute** first:

$((\lambda x\ (\lambda y.\ y))\ \Omega)\ z \Rightarrow ((\lambda y.\ y)[\Omega/x])\ z$
$\Rightarrow (\lambda y.\ y)\ z \Rightarrow y[z/y]$
$\Rightarrow z$

Ω is thrown away in the first step, so it doesn't matter that it diverges!

$e ::= x \mid (\lambda x.\ e) \mid (e_1\ e_2)$
$v ::= x \mid (\lambda x.\ e) \mid x\ v_1 \ldots v_n$

Reduction order

These are examples of the two most common *deterministic* reduction orders:

- *Call-by-value* (or *strict*) reduction always reduces arguments as much as possible before substituting them into a function
  - Most common in high-level programming languages
  - Often easier to reason about efficiency
- *Call-by-name* (or *lazy*) reduction always does substitution first, only evaluating arguments when necessary to proceed
  - Default evaluation strategy in Haskell and some similar languages
  - Opt-in feature in many other high-level languages
  - Convenient for programming with infinite data (e.g. cyclic lists, infinite sets)
  - Special case: *short-circuiting* operators
    - In C, evaluating `false && f()` will not call `f()`

$e ::= x \mid (\lambda x.\ e) \mid (e_1\ e_2)$
$v ::= x \mid (\lambda x.\ e) \mid x\ v_1 \ldots v_n$

Operational semantics (strict)

Call-by-value (or *strict*) reduction always reduces arguments as much as possible before substituting them into a function.

These are **all of the reduction rules** for strict untyped lambda calculus:

$$\text{E-AppL}\quad \frac{e \Rightarrow e'}{(e\ v) \Rightarrow (e'\ v)} \qquad \text{E-AppR}\quad \frac{e_2 \Rightarrow e'_2}{(e_1\ e_2) \Rightarrow (e_1\ e'_2)}$$

$$\text{E-Alpha}\quad \frac{y \notin \mathrm{free}(e)}{(\lambda x.\ e) \Rightarrow (\lambda y.\ e[y/x])} \qquad \text{E-Beta}\quad \frac{}{(\lambda x.\ e)\ v \Rightarrow e[v/x]}$$

$e ::= x \mid (\lambda x.\ e) \mid (e_1\ e_2)$
$v ::= x \mid (\lambda x.\ e) \mid x\ v_1 \dots v_n$

Operational semantics (lazy)

Call-by-name (or *lazy*) reduction always does substitution first, only evaluating arguments when necessary to proceed.

These are **all of the reduction rules** for lazy untyped lambda calculus:

$$\text{E-AppL}\quad \frac{e_1 \Rightarrow e'_1}{(e_1\ e_2) \Rightarrow (e'_1\ e_2)}$$

$$\text{E-Alpha}\quad \frac{y \notin \mathrm{free}(e)}{(\lambda x.\ e) \Rightarrow (\lambda y.\ e[y/x])} \qquad \text{E-Beta}\quad \frac{}{(\lambda x.\ e_1)\ e_2 \Rightarrow e_1[e_2/x]}$$

Applied lambda calculus

We should be able to write **any computable function** in lambda calculus.

```
int three() { return 3; }
```

How do we write this function when all we have to work with are lambda expressions?

- *Church encoding*: represent a number N as a two-argument function that applies its first argument N times to the second argument
  - `1 = λf x. f x`
  - `3 = λf x. f (f (f x))`
  - `plus = λm n. λf x. m f (n f x)`
  - Any kind of data can be represented with Church encoding!
- Or, more practically, we can extend the language with new kinds of data
  - This is often called *applied* lambda calculus

Lambda calculus + numbers

Where `n` is any integer literal:

$$e ::= x \mid (\lambda x.\ e) \mid (e_1\ e_2) \mid n \mid (e_1 + e_2) \mid (e_1 * e_2)$$

$$\text{E-Plus}\quad \frac{n_1 + n_2 = n_3}{n_1 + n_2 \Rightarrow n_3}$$

if n_1 plus n_2 is n_3 (in standard arithmetic)

then $(n_1 + n_2)$ reduces to n_3 in one step

$$\text{E-Times}\quad \frac{n_1 * n_2 = n_3}{n_1 * n_2 \Rightarrow n_3}$$

if n_1 times n_2 is n_3 (in standard arithmetic)

then $(n_1 * n_2)$ reduces to n_3 in one step

$e ::= x \mid (\lambda x.\ e) \mid (e_1\ e_2) \mid n \mid (e_1 + e_2) \mid (e_1 * e_2)$

Lambda calculus + numbers

This semantics chooses (arbitrarily) to **reduce left operands first**:

$$\text{E-PlusL}\quad \frac{e_1 \Rightarrow e'_1}{(e_1 + e_2) \Rightarrow (e'_1 + e_2)} \qquad \text{E-PlusR}\quad \frac{e \Rightarrow e'}{(v + e) \Rightarrow (v + e')}$$

$$\text{E-TimesL}\quad \frac{e_1 \Rightarrow e'_1}{(e_1 * e_2) \Rightarrow (e'_1 * e_2)} \qquad \text{E-TimesR}\quad \frac{e \Rightarrow e'}{(v * e) \Rightarrow (v * e')}$$

Some numeric expressions

```
   λx. x + 4

   (λx. x + 4) 5
⇒  9

   (λf g x. f (g x)) (λa. a + 5) (λb. b * 2) 7
⇒  (λg x. g x + 5) (λb. b * 2) 7
⇒  (λx. (x * 2) + 5) 7
⇒  (7 * 2) + 5
⇒  14 + 5
⇒  19
```


Equivalences are not reductions!

This is a valid equivalence in arithmetic:

```
x * (y + z) = (x + y) * (x + z)
```

But this is **not** a valid reduction in lambda calculus with numbers:

```
x * (y + z) ⇒ (x + y) * (x + z)
```

There is no reduction rule in our operational semantics for it!

The **only valid reductions** are the ones specified by the reduction rules.

Lambda calculus + numbers + bools

Where `n` is any integer literal and `b` is any Boolean literal:

e ::= x | (λx. e) | (e_1 e_2) | n | (e_1 + e_2) | (e_1 * e_2)
    | b | if e_1 then e_2 else e_3

Lambda calculus + numbers + bools

We usually want to **avoid evaluating both branches of an if expression**.

For example, in C:

```
if (true)
   return 0;
else
   printf("this should not be printed");
```

$e ::= x \mid (\lambda x.\ e) \mid (e_1\ e_2) \mid n \mid (e_1 + e_2) \mid (e_1 * e_2)$
$\mid b \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$

Lambda calculus + numbers + bools

We usually want to **avoid evaluating both branches of an if expression**.

$$\text{E-IfTrue}\ \frac{}{\texttt{if true then}\ e_1\ \texttt{else}\ e_2 \Rightarrow e_1}$$

$$\text{E-IfFalse}\ \frac{}{\texttt{if false then}\ e_1\ \texttt{else}\ e_2 \Rightarrow e_2}$$

$$\text{E-If}\ \frac{e_1 \Rightarrow e'_1}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow \texttt{if}\ e'_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3}$$

$e ::= x \mid (\lambda x.\ e) \mid (e_1\ e_2) \mid n \mid (e_1 + e_2) \mid (e_1 * e_2)$
$\mid b \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$

Values

$v ::= x \mid x\ v_1 \dots v_n \mid (\lambda x.\ e) \mid n \mid b$

In this applied lambda calculus definition, the values are **free variables, applications of a free variable to some number of values**, **functions**, **numbers**, and **booleans**.

Ideally, an expression should always either reduce to a value or diverge.

This means a closed term (one without free variables) should **always reduce to a function, number, or boolean**.

Do these properties still hold in this applied lambda calculus?

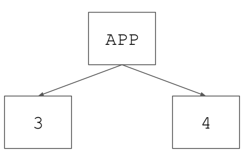
$e ::= x \mid (\lambda x.\ e) \mid (e_1\ e_2) \mid n \mid (e_1 + e_2) \mid (e_1 * e_2)$
$\mid b \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$

Stuck expressions

$v ::= x \mid x\ v_1 \dots v_n \mid (\lambda x.\ e) \mid n \mid b$

This is a syntactically valid expression:

`(3 4)`



There are no reduction rules that apply, but it doesn't mean anything sensible!

This expression is *stuck* - it's not a value, but it's also not reducible.

Dealing with stuck expressions

How can we avoid stuck expressions?

One solution: add an error expression to the set of values, and reduction rules to turn any stuck expression into an error.

$e ::= x \mid (\lambda x.\ e) \mid (e_1\ e_2) \mid n \mid (e_1 + e_2) \mid (e_1 * e_2) \mid \texttt{err}$
$\mid b \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$

$v ::= x \mid x\ v_1 \dots v_n \mid (\lambda x.\ e) \mid n \mid b \mid \texttt{err}$

$$\text{E-AppErr}\ \frac{}{n\ e \Rightarrow \texttt{err}} \qquad \text{E-PlusErrL}\ \frac{}{(\lambda x.\ e_1) + e_2 \Rightarrow \texttt{err}} \qquad \text{etc.}$$

Dealing with stuck expressions

Ideally, a closed expression should always reduce to a value without getting stuck.

How can we avoid stuck expressions?

Another solution: *types!*

- Categorize expressions by the kinds of things we can do with them
- Only allow *well-typed* expressions in the language
- Define a *typechecking* procedure that rules out expressions that aren't well-typed
- Next lecture!

Further reading (optional)

- Types and Programming Languages, Benjamin Pierce
  - Section 1: Untyped Systems
- Stanford Encyclopedia of Philosophy entry on The Lambda Calculus
  - https://plato.stanford.edu/entries/lambda-calculus/
- Alligator Eggs (cute and simple pictorial representation)
  - http://worrydream.com/AlligatorEggs/

Note that there are several different syntaxes for lambda calculus in common use - don't get confused!