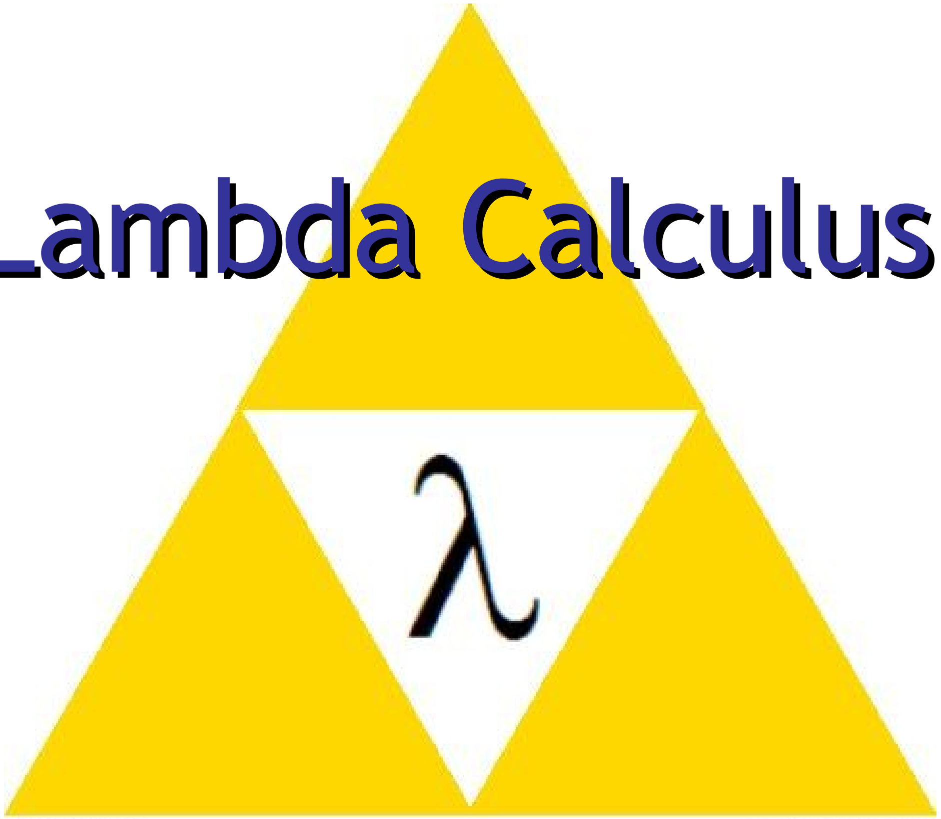
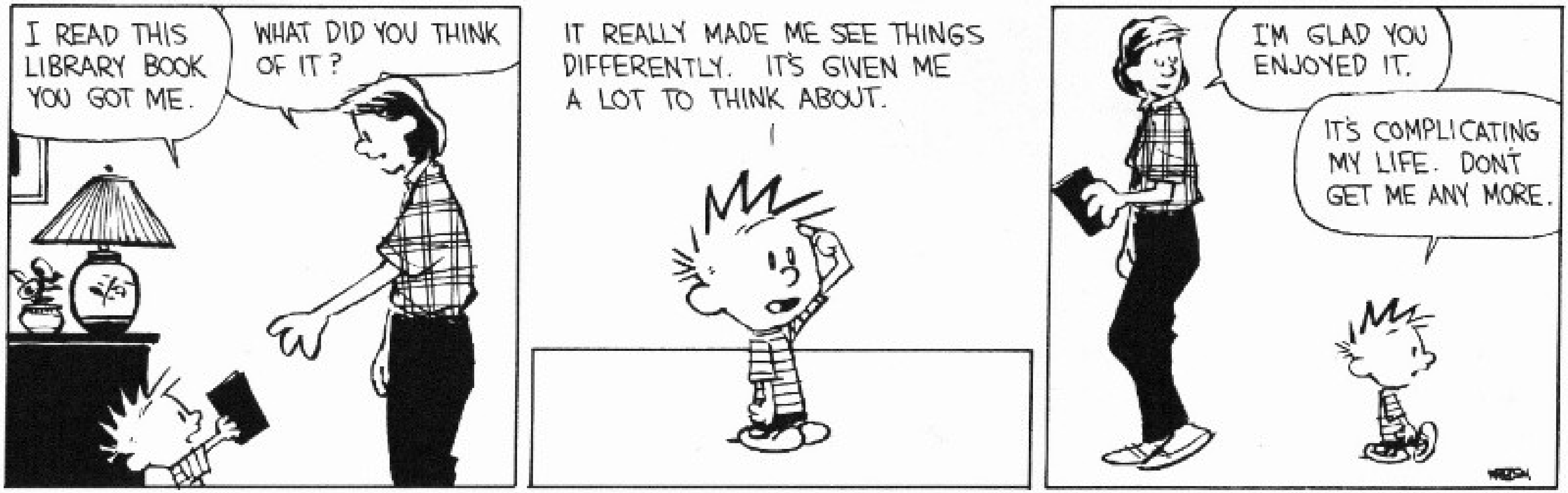
# Lambda Calculus




I READ THIS LIBRARY BOOK YOU GOT ME.
WHAT DID YOU THINK OF IT?
IT REALLY MADE ME SEE THINGS DIFFERENTLY. IT'S GIVEN ME A LOT TO THINK ABOUT.
I'M GLAD YOU ENJOYED IT.
IT'S COMPLICATING MY LIFE. DON'T GET ME ANY MORE.

## One-Slide Summary

- The **lambda calculus** is a simple, abstract, Turing-complete functional language that is often used to model and study other languages.
- The heart of the lambda calculus is the application of a function expression to its argument. This can be done via **substitution** and is called **beta-reduction**.
- There are multiple evaluation strategies.

# Plan

- Introduce lambda calculus
  - Syntax
  - Substitution
  - Operational Semantics (… with contexts!)
  - Evaluation strategies
  - Equality
- Later:
  - Relationship to programming languages
  - Study of types and type systems

# Lambda Background

- Developed in 1930's by Alonzo Church
- Subsequently studied by many people
  - Still studied today!
- Considered the "testbed" for procedural and functional languages
  - Simple
  - Powerful
  - Easy to extend with new features of interest
  - Lambda:PL :: Turning Machine:Complexity
  - Somewhat like a crowbar …

"Whatever the next 700 languages turn out to be, they will surely be variants of lambda calculus."

(Landin '66)

## Lambda Syntax

- The λ-calculus has 3 kinds of expressions (terms)

| | | |
|---|---|---|
| $e ::= x$ | | Variables |
| $\mid$ | $\lambda x.\ e$ | Functions (abstractions) |
| $\mid$ | $e_1\ e_2$ | Application |

- $\lambda x.\ e$ is a one-argument anonymous function with body $e$
- $e_1\ e_2$ is a function application
- Application associates to the left

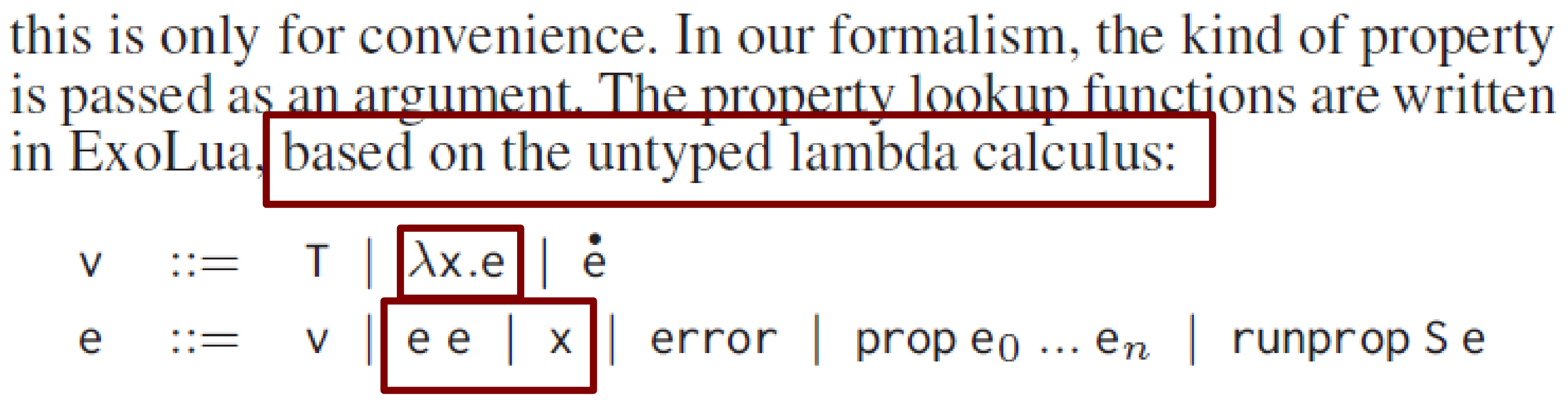
$$x\ y\ z \equiv (x\ y)\ z$$

- Abstraction extends far to the right

$$\lambda x.\ x\ \lambda y.\ x\ y\ z \equiv \lambda x.\ (x\ [\lambda y.\ \{(x\ y)\ z\}])$$

# Why Should I Care?

- A language with 3 expressions? Woof!
- DeVito et al. *First-class Runtime Generation of High-performance Types using Exotypes*. PLDI ’14
  - Just one example of a recent PL/security paper



this is only for convenience. In our formalism, the kind of property is passed as an argument. The property lookup functions are written in ExoLua, $\boxed{\text{based on the untyped lambda calculus:}}$

$$
\begin{array}{lcl}
\mathsf{v} & ::= & \mathsf{T} \mid \boxed{\lambda \mathsf{x}.\mathsf{e}} \mid \dot{\mathsf{e}} \\
\mathsf{e} & ::= & \mathsf{v} \mid \boxed{\mathsf{e}\ \mathsf{e} \mid \mathsf{x}} \mid \mathsf{error} \mid \mathsf{prop}\ \mathsf{e}_0 \ldots \mathsf{e}_n \mid \mathsf{runprop}\ \mathsf{S}\ \mathsf{e}
\end{array}
$$

# PLDI 2020

## FreezeML

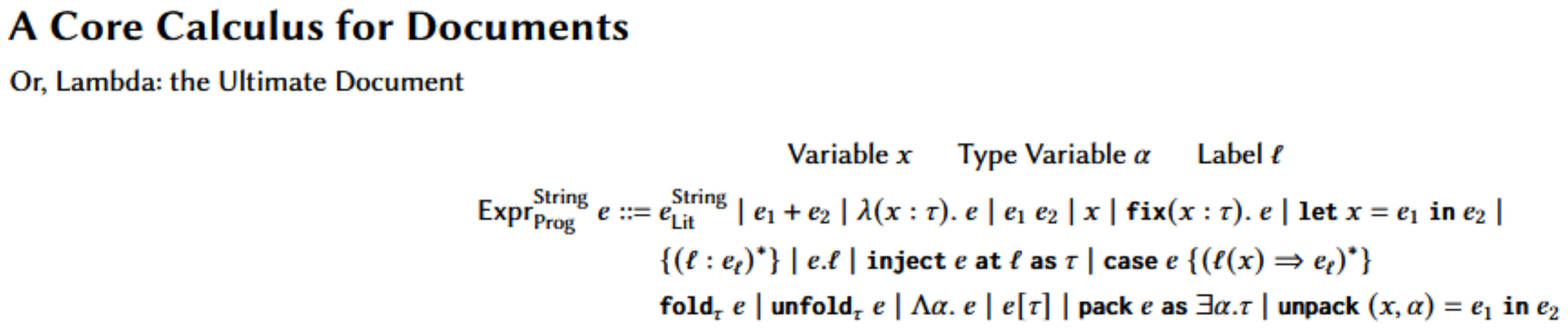
Complete and Easy Type Inference for First-Class Polymorphism

$$\begin{array}{lrl}
\text{Term Variables} & \mathsf{Var} \ni & x, y, z \\
\text{Terms} & \mathsf{Term} \ni M, N ::= & x \mid \lceil x \rceil \mid \lambda x.M \\
 & \mid & \lambda(x : A).M \mid M\,N \\
 & \mid & \mathbf{let}\ x = M\ \mathbf{in}\ N \\
 & \mid & \mathbf{let}\ (x : A) = M\ \mathbf{in}\ N
\end{array}$$

## From Folklore to Fact: Comparing Implementations of Stacks and Continuations

$$\begin{array}{lcl}
exp & ::= & \mathtt{let}\ (x_1, \ldots) = prim(y_1, \ldots) \\
 & \mid & \mathtt{fun}\ f(x_1, \ldots /k) = e_1\ \mathtt{in}\ e_2 \\
 & \mid & \mathtt{cont}\ k(x_1, \ldots) = e_1\ \mathtt{in}\ e_2 \\
 & \mid & \mathtt{if}\ x\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \\
 & \mid & \mathtt{apply}\ f(x_1, \ldots /k) \\
 & \mid & \mathtt{throw}\ k(x_1, \ldots)
\end{array}$$

# POPL 2024

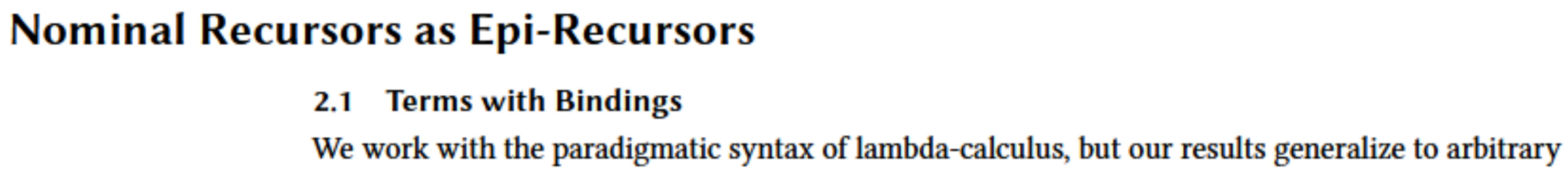


## A Core Calculus for Documents

Or, Lambda: the Ultimate Document

Variable $x$ Type Variable $\alpha$ Label $\ell$

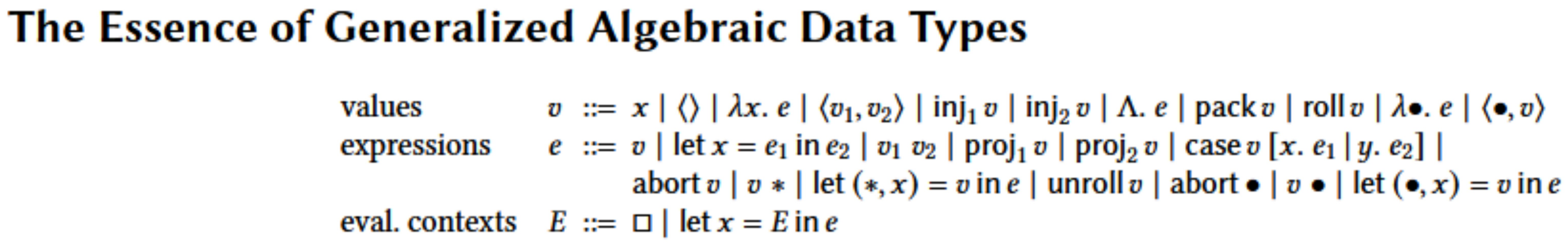
$$\begin{aligned}\mathrm{Expr}^{\mathrm{String}}_{\mathrm{Prog}}\ e ::= {} & e^{\mathrm{String}}_{\mathrm{Lit}} \mid e_1 + e_2 \mid \lambda(x:\tau).\ e \mid e_1\ e_2 \mid x \mid \mathbf{fix}(x:\tau).\ e \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid \\ & \{(\ell : e_\ell)^*\} \mid e.\ell \mid \mathbf{inject}\ e\ \mathbf{at}\ \ell\ \mathbf{as}\ \tau \mid \mathbf{case}\ e\ \{(\ell(x) \Rightarrow e_\ell)^*\} \\ & \mathbf{fold}_\tau\ e \mid \mathbf{unfold}_\tau\ e \mid \Lambda\alpha.\ e \mid e[\tau] \mid \mathbf{pack}\ e\ \mathbf{as}\ \exists\alpha.\tau \mid \mathbf{unpack}\ (x,\alpha) = e_1\ \mathbf{in}\ e_2\end{aligned}$$



## Nominal Recursors as Epi-Recursors

### 2.1 Terms with Bindings

We work with the paradigmatic syntax of lambda-calculus, but our results generalize to arbitrary



## The Essence of Generalized Algebraic Data Types

$$\begin{array}{llcl} \text{values} & v & ::= & x \mid \langle\rangle \mid \lambda x.\ e \mid \langle v_1, v_2\rangle \mid \mathsf{inj}_1\, v \mid \mathsf{inj}_2\, v \mid \Lambda.\ e \mid \mathsf{pack}\, v \mid \mathsf{roll}\, v \mid \lambda\bullet.\ e \mid \langle\bullet, v\rangle \\ \text{expressions} & e & ::= & v \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid v_1\ v_2 \mid \mathsf{proj}_1\, v \mid \mathsf{proj}_2\, v \mid \mathsf{case}\, v\ [x.\ e_1 \mid y.\ e_2] \mid \\ & & & \mathsf{abort}\, v \mid v\ * \mid \mathsf{let}\ (*, x) = v\ \mathsf{in}\ e \mid \mathsf{unroll}\, v \mid \mathsf{abort}\ \bullet \mid v\ \bullet \mid \mathsf{let}\ (\bullet, x) = v\ \mathsf{in}\ e \\ \text{eval. contexts} & E & ::= & \square \mid \mathsf{let}\ x = E\ \mathsf{in}\ e \end{array}$$

## Examples of Lambda Expressions

- The identity function:

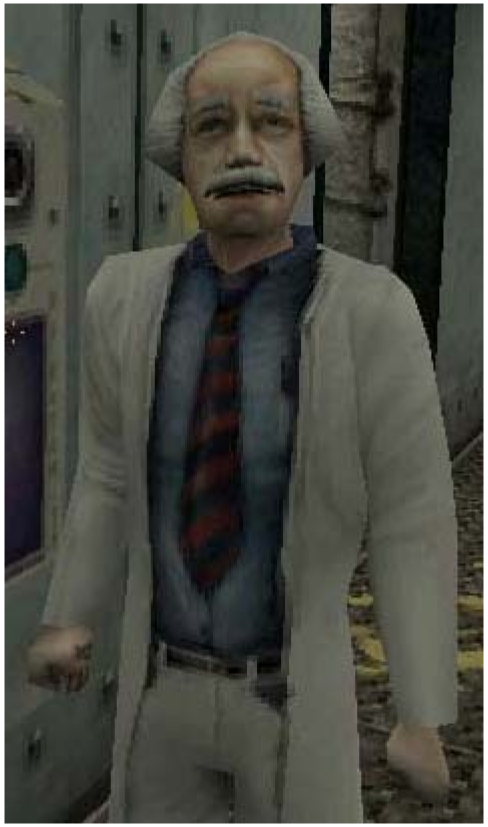
$$I =_{\text{def}} \lambda x.\ x$$

- A function that, given an argument y, discards it and yields the identity function:

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

- A function that, given an function f, invokes it on the identity function:

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



*"There goes our grant money."*

## Scope of Variables

- As in all languages with variables, it is important to discuss the notion of scope
  - The scope of an identifier is the portion of a program where the identifier is accessible
- An abstraction $\lambda x.\ E$ binds variable $x$ in $E$
  - $x$ is the newly introduced variable
  - $E$ is the scope of $x$ (unless x is shadowed)
  - We say $x$ is bound in $\lambda x.\ E$
  - Just like formal function arguments are bound in the function body

## Free and Bound Variables

- A variable is said to be free in E if it has occurrences that are not bound in E
- We can define the free variables of an expression E recursively as follows:
  - $\mathrm{Free}(x) = \{x\}$
  - $\mathrm{Free}(E_1\ E_2) = \mathrm{Free}(E_1) \cup \mathrm{Free}(E_2)$
  - $\mathrm{Free}(\lambda x.\ E) = \mathrm{Free}(E) - \{x\}$
- Example: $\mathrm{Free}(\lambda x.\ x\ (\lambda y.\ x\ y\ z)) = \{z\}$
- Free variables are (implicitly or explicitly) declared outside the expression

# Free Your Mind!

- Just as in any language with statically-nested scoping we have to worry about variable shadowing
  - An occurrence of a variable might refer to different things in different contexts
- Example in IMP with locals:

  let x = 5 in x + (let x = 9 in x) + x

- In λ-calculus:

  λx. x (λx. x) x

# Renaming Bound Variables

- λ-terms that can be obtained from one another by renaming bound variables are considered *identical*
- This is called α-equivalence
- Renaming bound vars is called α-renaming
- Ex: λx. x is identical to λy. y and to λz. z
- Intuition:
  - By changing the name of a formal argument and all of its occurrences in the function body, the behavior of the function *does not change*
  - In λ-calculus such functions are considered identical

## Make It Easy On Yourself

- Convention: we will always try to rename bound variables so that they are all unique
  - e.g., write $\lambda x.\ x\ (\lambda y.y)\ x$ instead of $\lambda x.\ x\ (\lambda x.x)\ x$
- This makes it easy to see the scope of bindings and also prevents confusion!

WHAT DOES IT MEAN WHEN SOMEONE SAYS TO "GIVE IT THE OL' COLLEGE TRY"?
IT MEANS YOU JOIN YOUR FRIENDS, GET SOME CHEAP BEER, ORDER A PIZZA, AND FORGET ABOUT TOMORROW.
THAT'S NOT WHAT IT MEANS!
WHERE DID YOU GO TO COLLEGE?
NEVER MIND.
WATTERSON

# Substitution

- The substitution of F for x in E (written [F/x]E)
  - Step 1. Rename bound variables in E and F so they are unique
  - Step 2. Perform the textual substitution of f for X in E
- Called capture-avoiding substitution
- Example: [y (λx. x) / x] λy. (λx. x) y x
  - After renaming: [y (λx. x) / x] λz. (λu. u) z x
  - After substitution: λz. (λu. u) z (y (λx. x))
- If we are not careful with scopes we might get:

  λy. (λx. x) y (y (λx. x)) ← wrong!

# The De Bruijn Notation

- An alternative syntax that avoids naming of bound variables (and the subsequent confusions)
- The De Bruijn index of a variable *occurrence* is the number of lambdas that separate the occurrence from its binding lambda in the abstract syntax tree
- The De Bruijn notation replaces names of occurrences with their De Bruijn indices
- Examples:

| | |
|---|---|
| – **λ x. x** | **λ. 0** |
| – **λ x. λ x. x** | **λ. λ. 0** |
| – **λ x. λ y. y** | **λ. λ. 0** |
| – **(λ x. x x) (λ z. z z)** | **(λ. 0 0) (λ. 0 0)** |
| – **λ x. (λ x. λ y. x) x** | **λ. (λ. λ. 1) 0** |

Identical terms have identical representations!

## Combinators

- A λ-term without free variables is closed or a combinator
- Some interesting combinators:

| | |
|---|---|
| I | = λ x. x |
| K | = λ x. λ y. x |
| S | = λ f. λ g. λ x. f x (g x) |
| D | = λ x. x x |
| Y | = λ f. (λ x. f (x x)) (λ x. f (x x)) |

- Theorem: any closed term is equivalent to one written with just S, K and I
  - Example: D $=_\beta$ S I I
  - (we'll discuss this form of equivalence later)

## Q:  Music  (241 / 842)

- Name the singer and his crossover 1982 album that holds (as of 2025) the record of being the best-selling album of all-original material in the US (34 times platinum, 37 weeks as Billboard #1). Much of that success was the result of the singer's use of the MTV music video.

## Q: Class Archives

- *This* is the third-largest city proper in the world (over 27 million in 2020), located on the Yangtze river delta. Its name means "upper sea" or "upon the sea".

# Q: Class Archives

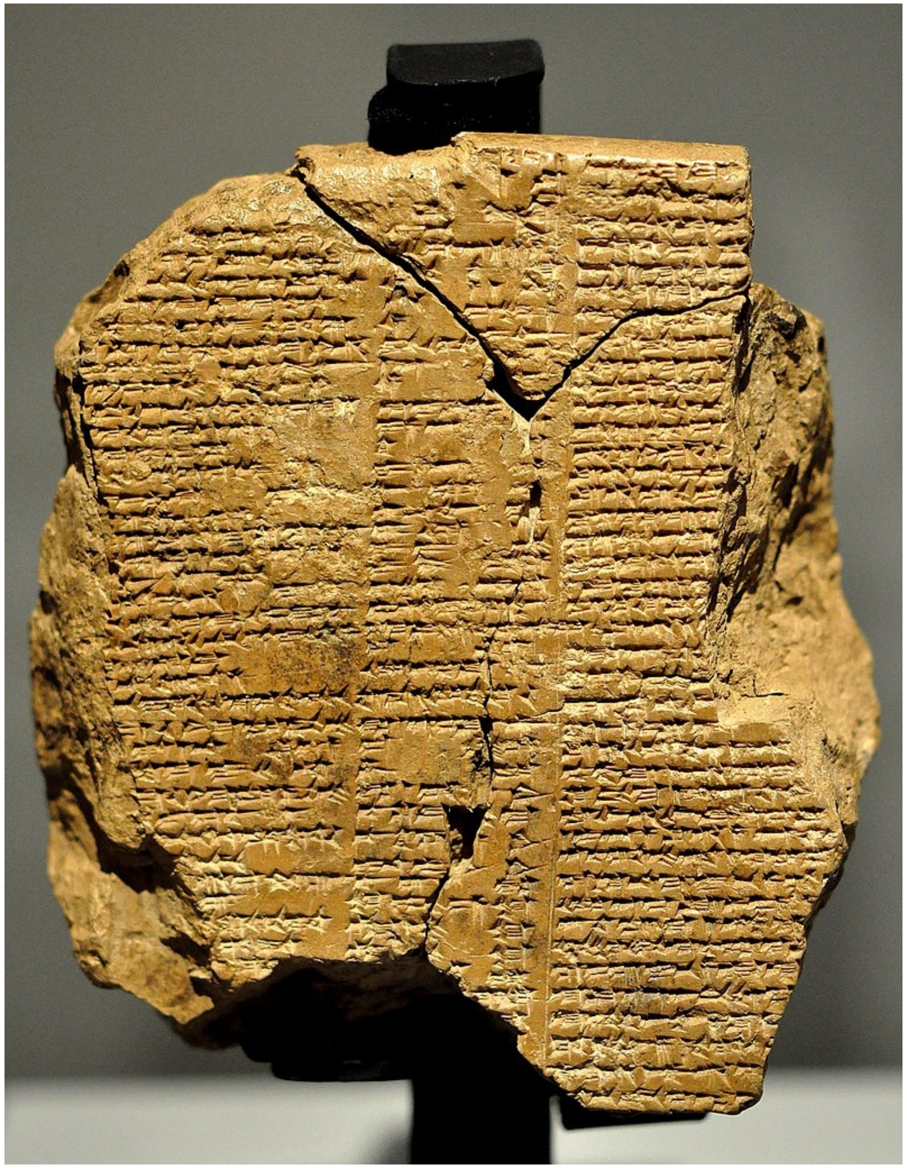
- This recreational activity is the world's most popular sport (over 250 million players in 200 countries). It is governed by the *Laws of the Game*. For fouls, players are cautioned with yellow cards and ejected with red cards.

## Q: Class Archive

- This *Prarie State* has a reputation as a predictor and model of the US as a whole. Barack Obama was a senator from this state before being elected president. It is also known as the *Land of Lincoln*.

## Q: Class Archives

- This king of Uruk reigned around 2500 BCE and is the eponymous protagonist of an *Epic*, the greatest surviving work of Mesopotamian literature. In it, he and his wild companion Enkidu kill the Bull of Heaven sent by Ishtar. Sadly, Enkidu dies.



## Q: Class Archives

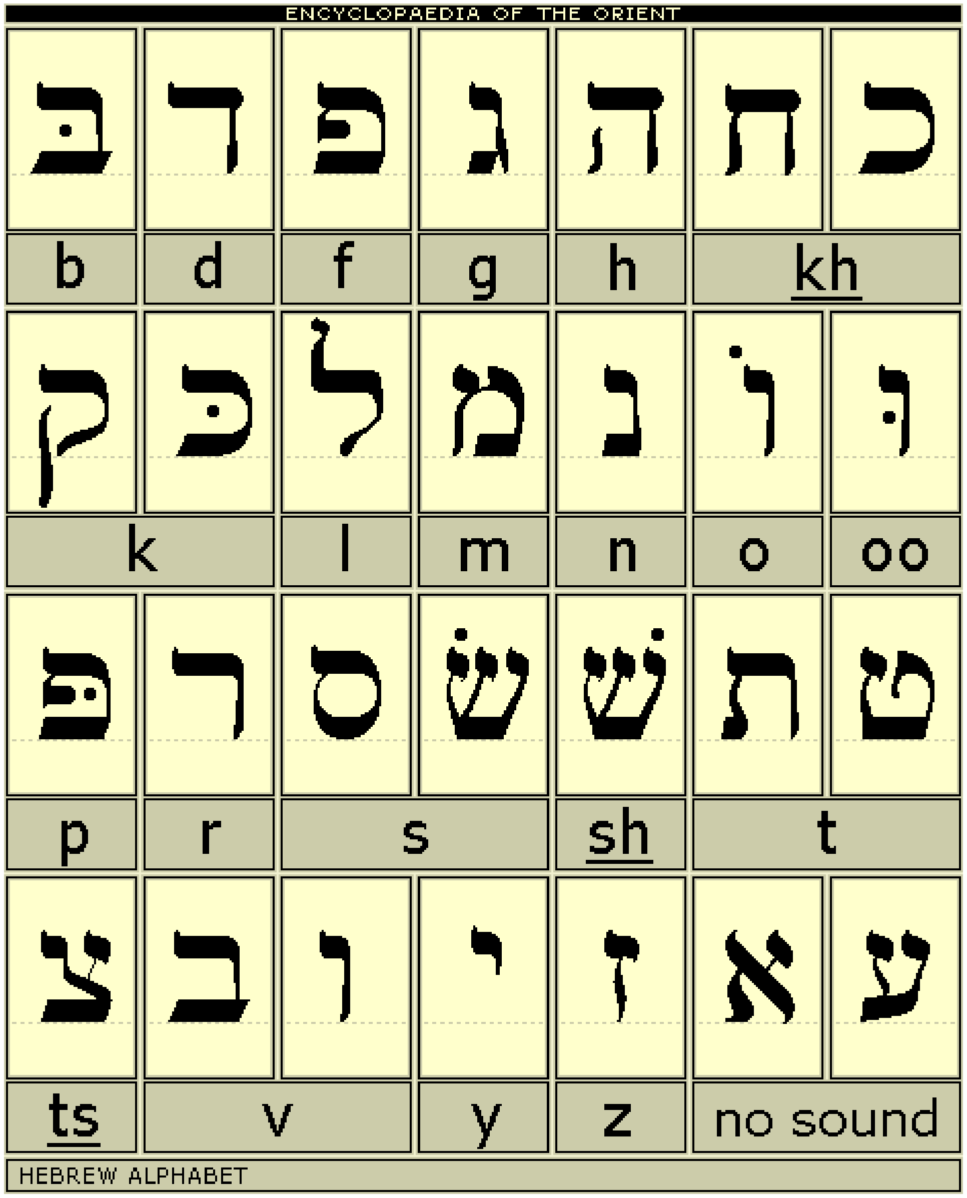
- This space-themed MMORPG has been uncharitably described as a "rousing game of combat spreadsheets". Characters train skills in real time, even when players are not logged into the game. Players can create in-game Ponzi schemes and illegal banks if they choose, and philosophies battle it out in a microcosm of society.

## Q: Class Archives

- In cryptography, *this* is an attack involving information gained from the *physical* reification of a cryptosystem, rather than via brute force or algorithmic flaws. Power consumption and EM leaks are classic examples.

# Q: General (452 / 842)

- Name any 3 of the 22 letters in the Hebrew alphabet.



ENCYCLOPAEDIA OF THE ORIENT

| | | | | | | |
|---|---|---|---|---|---|---|
| בּ | דּ | פ | ג | ה | ח | כ |
| b | d | f | g | h | kh | |
| ק | כּ | ל | מ | נ | וֹ | וּ |
| k | | l | m | n | o | oo |
| פּ | ר | ס | שׂ | שׁ | ת | ט |
| p | r | s | | sh | t | |
| צ | ב | ו | י | ז | א | ע |
| ts | v | | y | z | no sound | |

HEBREW ALPHABET

# Informal Semantics

- We consider only closed terms
- The evaluation of

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

  - Binds x to f
  - Evaluates e *with the new binding*
  - Yields the result of this evaluation
- Like a function call, or like "let x = f in e"
- Example:

$(\lambda\ f.\ f\ (f\ e))\ g$ *evaluates to* $g\ (g\ e)$

## Operational Semantics

- Many operational semantics for the $\lambda$-calculus
- All are based on the equation

  $$(\lambda\ x.\ e)\ f =_\beta [f/x]e$$

  usually read from left to right
- This is called the $\beta$-rule and the evaluation step a $\beta$-reduction
- The subterm $(\lambda\ x.\ e)\ f$ is a $\beta$-redex
- We write $e \rightarrow_\beta g$ to say that $e$ $\beta$-reduces to $g$ in one step
- We write $e \rightarrow_\beta^* g$ to say that $e$ $\beta$-reduces to $g$ in 0 or more steps
  - Remind you of the small-step opsem term rewriting?

# Examples of Evaluation

- The identity function:

  $(\lambda\ x.\ x)\ E \to [E\ /\ x]\ x = E$

- Another example with the identity:

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

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

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

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

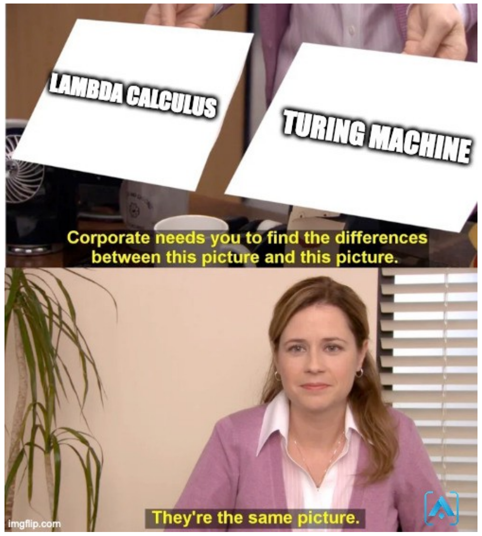
  $[\lambda\ y.\ y\ /\ x]\ x = \lambda\ y.\ y$

- A *non-terminating* evaluation:

  $(\lambda\ x.\ xx)\ (\lambda\ y.\ yy) \to$

  $[\lambda\ y.\ yy\ /\ x]\ xx = (\lambda\ y.\ yy)\ (\lambda\ y.\ yy) \to \ldots$

- Try T T, where T = $\lambda$x. x x x


LAMBDA CALCULUS
TURING MACHINE
Corporate needs you to find the differences between this picture and this picture.
They're the same picture.
imgflip.com

# Evaluation and the Static Scope

- The definition of substitution guarantees that evaluation respets static scoping:

$$(\lambda\ x.\ (\lambda\ y.\ y\ x))\ (y\ (\lambda\ x.\ x)) \to_\beta \lambda\ z.\ z\ (y\ (\lambda\ v.\ v))$$

(y remains free, i.e., defined externally)

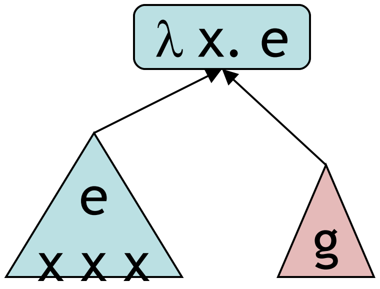
- If we forget to rename the bound y:

$$(\lambda\ x.\ (\lambda\ y.\ y\ x))\ (y\ (\lambda\ x.\ x)) \to_\beta^* \lambda\ y.\ y\ (y\ (\lambda\ v.\ v))$$

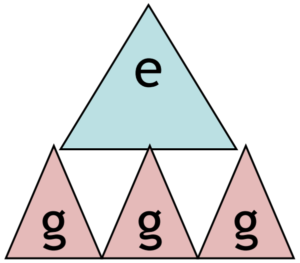
(y was free before but *is bound now*)

# Another View of Reduction

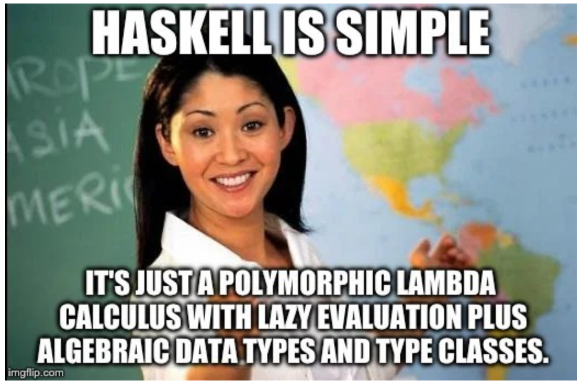
- The application



- Becomes:



(terms can grow substantially through β-reduction!)


HASKELL IS SIMPLE
IT'S JUST A POLYMORPHIC LAMBDA CALCULUS WITH LAZY EVALUATION PLUS ALGEBRAIC DATA TYPES AND TYPE CLASSES.
imgflip.com

# Normal Forms

- A term without redexes is in normal form
- A reduction sequence stops at a normal form

- If $e$ is in normal form and $e \to_{\beta}^{*} f$ then $e$ is identical to $f$

- $K = \lambda\ x.\ \lambda\ y.\ x$ is in normal form
- $K\ I$ is *not* in normal form

# Nondeterministic Evaluation

- We define a small-step reduction relation

$$\frac{}{(\lambda\ x.\ e)\ f \to [f/x]e}$$

$$\frac{e_1 \to e_2}{e_1\ f \to e_2\ f} \qquad \frac{f_1 \to f_2}{e\ f_1 \to e\ f_2}$$

$$\frac{e \to f}{\lambda\ x.\ e \to \lambda\ x.\ f}$$

- This is a non-deterministic semantics
- Note that we evaluate under $\lambda$ *(where?)*

# Lambda Calculus Contexts

- Define contexts with one hole
- $H ::= \bullet \mid \lambda\, x.\ H \mid H\ e \mid e\ H$
- Write $H[e]$ to denote the filling of the hole in $H$ with the expression $e$
- Example:

  $H = \lambda\, x.\ x\ \bullet \qquad H[\lambda\, y.\ y] = \lambda\, x.\ x\ (\lambda\, y.\ y)$
- Filling the hole allows variable capture!

  $H = \lambda\, x.\ x\ \bullet \qquad H[x] = \lambda\, x.\ x\ x$

## Contextual Opsem

$$\frac{}{(\lambda\ x.\ e)\ f \to [f/x]e} \qquad \frac{e \to f}{H[e] \to H[f]}$$

- Contexts allow concise formulations of congruence rules (application of local reduction rules on subterms)
- Reduction occurs at a $\beta$-redex that can be anywhere inside the expression
- The latter rule is called a congruence or structural rule
- The above rules to not specify which redex must be reduced first

# The Order of Evaluation

- In a $\lambda$-term there could be more than one instance of $(\lambda\ x.\ e)\ f$, as in:

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

  - Could reduce the inner or outer $\lambda$
  - Which one should we pick?

$$\begin{tikzcd}
& (\lambda\ y.\ (\lambda\ x.\ x)\ y)\ E \arrow[dl, "\text{inner}"'] \arrow[dr, "\text{outer}"] & \\
{(\lambda\ y.\ [y/x]\ x)\ E = (\lambda\ y.\ y)\ E} \arrow[dr] & & {[E/y]\ (\lambda\ x.\ x)\ y = (\lambda\ x.\ x)\ E} \arrow[dl] \\
& E &
\end{tikzcd}$$

# The Diamond Property

- A relation R has the diamond property if whenever $e \ R \ e_1$ and $e \ R \ e_2$ then there exists $e_3$ such that $e_1 \ R \ e_3$ and $e_2 \ R \ e_3$

$$\begin{tikzcd}
& E \arrow[dl, dash, "R"'] \arrow[dr, dash, "R"] & \\
e_1 \arrow[dr, dash, "R"'] & & e_2 \arrow[dl, dash, "R"] \\
& E_3 &
\end{tikzcd}$$

- $\rightarrow_\beta$ does *not* have the diamond property
- $\rightarrow_\beta{}^*$ has the diamond property
- Also called the confluence property

## A Diamond In The Rough

- Languages defined by non-deterministic sets of rules are common
  - Logic programming languages
  - Expert systems
  - Constraint satisfaction systems
    - And thus most pointer analyses …
  - Dataflow systems
  - Makefiles
- It is useful to know whether such systems have the diamond property

# (Beta) Equality

- Let $=_\beta$ be the reflexive, transitive and **symmetric** closure of $\rightarrow_\beta$

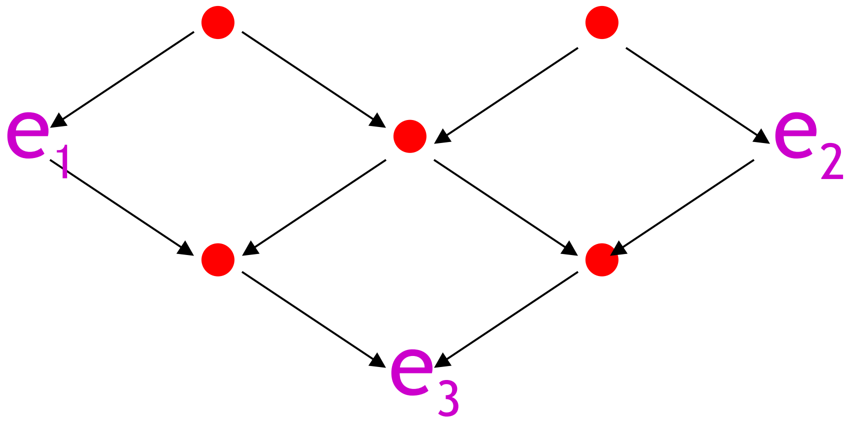
$$=_\beta \textbf{ is } (\rightarrow_\beta \cup \leftarrow_\beta)^*$$

- That is, $e =_\beta f$ if $e$ converts to $f$ via a sequence of forward and backward $\rightarrow_\beta$

$$\begin{tikzcd}
 & \bullet \arrow[dl] \arrow[dr] & & \bullet \arrow[dl] \arrow[dr] & \\
e & & \bullet & & f
\end{tikzcd}$$

# The Church-Rosser Theorem

- If $e_1 =_\beta e_2$ then there exists $e_3$ such that $e_1 \to_\beta^* e_3$ and $e_2 \to_\beta^* e_3$



- Proof (informal): apply the diamond property as many times as necessary

# Corollaries

- If $e_1 =_\beta e_2$ and $e_1$ and $e_2$ are normal forms then $e_1$ is identical to $e_2$
  - From C-R we have $\exists e_3.\ e_1 \rightarrow_\beta^* e_3$ and $e_2 \rightarrow_\beta^* e_3$
  - Since $e_1$ and $e_2$ are normal forms they are identical to $e_3$
- If $e \rightarrow_\beta^* e_1$ and $e \rightarrow_\beta^* e_2$ and $e_1$ and $e_2$ are normal forms then $e_1$ is identical to $e_2$
  - "All terms have a unique normal form."

# Evaluation Strategies

- Church-Rosser theorem says that independent of the reduction strategy we will find $\leq$1 normal form
- But some reduction strategies might find 0
- $(\lambda\ x.\ z)\ \underline{((\lambda\ y.\ y\ y)\ (\lambda\ y.\ y\ y))} \rightarrow$

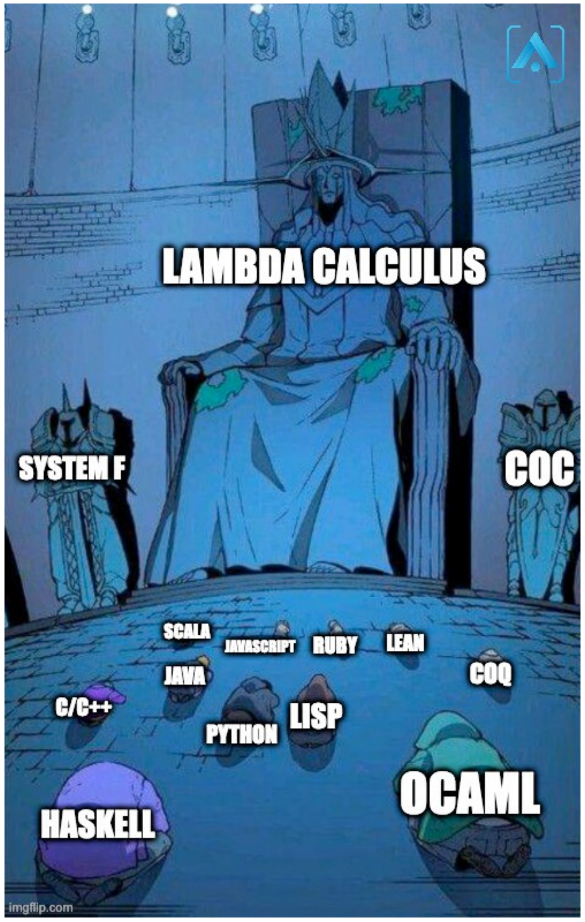
  $(\lambda\ x.\ z)\ \underline{((\lambda\ y.\ y\ y)\ (\lambda\ y.\ y\ y))} \rightarrow \ldots$
- $\underline{(\lambda\ x.\ z)\ ((\lambda\ y.\ y\ y)\ (\lambda\ y.\ y\ y))} \rightarrow z$
- There are three traditional strategies
  - normal order (never used, always works)
  - call-by-name (rarely used, cf. TeX or Haskell)
  - call-by-value (amazingly popular)

# Call By Value

- Normal Order
  - Evaluates the left-most redex not contained in another redex
  - If there is a normal form, this finds it
  - Not used in practice: requires partially evaluating function pointers and looking "inside" functions
- Call-By-Name ("lazy")
  - Don't reduce under $\lambda$, don't evaluate a function argument (until you need to)
  - Does not always evaluate to a normal form
- **Call-By-Value ("eager" or "strict")**
  - Don't reduce under $\lambda$, *do* evaluate a function's argument right away
  - Finds normal forms less often than the other two

## Lambda Will Continue

- This time: λ syntax, semantics, reductions, equality, …
- Next time: encodings, real programs, type systems, and all the fun stuff!



## Homework

- Homework 4 due soon
- Read ahead on Homework 6?

# Tricksy On The Board Answer

- Is this rule from lectures ago unsound?

$$\frac{\vdash \{A \wedge p\}\ c_{then}\ \{B_{then}\} \qquad \vdash \{A \wedge \neg p\}\ c_{else}\ \{B_{else}\}}{\vdash \{A\}\ \text{if } p \text{ then } c_{then} \text{ else } c_{else}\ \{B_{then} \vee B_{else}\}}$$

- Nope: it's our basic rule plus 2x consequence

$$\frac{\vdash \{A \wedge p\}\ c_1\ \{B\} \qquad \vdash \{A \wedge \neg p\}\ c_2\ \{B\}}{\vdash \{A\}\ \text{if } p \text{ then } c_1 \text{ else } c_2\ \{B\}}$$

$$\frac{\vdash A' \Rightarrow A \qquad \vdash \{A\}\ c\ \{B\} \qquad \vdash B \Rightarrow B'}{\vdash \{A'\}\ c\ \{B'\}}$$

- Note that $B_{then} \Rightarrow B_{then} \vee B_{else}$