Let's look at expanding IMP:

- What is IMP missing?
- All limited to IMP = loops + globals
- Syntax, semantics, equivalence
- Scope, functions, data structures
- (threads, I/O, exceptions, strings, ...)

So far:

Lecture 8: Lambda Calculus

CSE 505: Graduate Programming Languages
let loop k y = if k = 0 then y else k (y + loop (k - 1) y)

let cons k y = fun k' -> if k' = k then Empty else raise Empty

let empty = fun k -> raise Empty

Data : Closures store data, e.g. alias (association list)

let seven = add3 4

let x + y + z

let z = 2 in

let add 3 y =

let x = 1

let Scope : not all data available to all code

With higher order functions, we get both scope and data structures.

Data + Code
Division can also cause "stuckness"

What would C+ Scheme, Java, ML, Perl, etc?

Can have "stuck" expr or stmt, e.g., C.

Parts of values, not just pairs of intels.

Note:

\[ H : (e, e') \rightarrow (v, v') \]

\[ H : e \rightarrow v \]

\[ H, e_2 \rightarrow v_2 \]

\[ H, e \rightarrow v \]

H so c ... all old stuff plus

\[ (\text{no change to } s) \]

\[ v \leftarrow x \]

\[ H \leftarrow H' \]

\[ v = c \mid (v, v) \]

\[ e = c \mid (e', e) \mid e \mid e + c \mid e \mid e, e \]

Not so bad...

Data Structures for IMP
h

Sweet! Does this match our intuition?

\[ \text{If } e : (e) \rightarrow H \vdash x : \forall \]  
\[ \text{I: } e \vdash \text{fun } x \rightarrow \text{I: } e \vdash c \]

\[ \text{H: } \text{fun } x \rightarrow \text{S} \vdash \text{H: } \text{fun } x \rightarrow \text{S} \]

\[ \text{H: } \text{fun } x \rightarrow \text{S} \vdash \text{fun } x \rightarrow \text{S} \]

\[ \text{H: } \text{fun } x \rightarrow \text{S} \vdash \text{fun } x \rightarrow \text{S} \]

\[ \text{H: } \text{fun } x \rightarrow \text{S} \vdash \text{fun } x \rightarrow \text{S} \]

\[ \text{H: } \text{fun } x \rightarrow \text{S} \vdash \text{fun } x \rightarrow \text{S} \]

Is that one?...

is that one?...

mutual dependence...

watch!
H, e L(e2) \rightarrow \text{Hi} \land \text{Hi} \rightarrow \text{y} \land \text{y} \land \text{fresh} \{\text{y}, \text{y} \}\)

H, \text{e L(x) \rightarrow Hi \land Hi} \rightarrow \text{fun} x \rightarrow \text{Hi} \rightarrow \text{y} \land \text{y} \land \text{fresh} \{\text{y}, \text{y} \}\)

OK, maybe I'm just wrong semantically... what about

fun x \rightarrow \text{Hi} = \text{Hi} \land \text{Hi} \rightarrow \text{y} \land \text{y} \land \text{fresh} \{\text{y}, \text{y} \}\)

function, should only have local consequences - choice of local names should not escape - locals should be "local" - we care about escape, not variable bindings!

We want 1... if we want 1

\[ \text{ans} : x = x \]

(\text{fun} x \rightarrow y := x)(\text{x})

x = 1

\text{what about...}
Sorta weird:
- "fresh" not very IMP-ish, but ok (malloc)
- far from actual implementation
- inconvenient for reasoning about something basic as calling...

TOTALLY BROKEN!
- what if "f" calls another func that modifies global "x"?

\[
f := (\text{fun } x \rightarrow g := (\text{fun } z \rightarrow \text{ans} := x + z));
\]
\[
f (2);
\]
\[
x := 3;
\]
\[
g (4)
\]

WANT: \text{ans} = 6
- \(f(2)\) should make \(g\) a func which adds 2 to its arg and stores result in \(\text{ans}\)
Someone thought of this long ago...

- Integers (iii)
- Mutation, conditionals, loops (i), even
- Get rid of everything.
- Add IMP features back later
- Take a step back, figure out this core idea
- Assigns to globals
- Funs are not simply sugar for
  - Just a global heap of ints
  - Can't properly model local scope

Punch line

\[
\text{Current value of } x \text{ to its range: } \quad \text{Ans} = \sqrt{x}
\]
The lambda calculus. "Laundrin gco
to be, they will surely be variants of

whenever next 700 languages turn out

\[ \forall x \in \mathbb{R} \quad e \]

\[ e = x \cdot e \]

\[ e \]

The Lambda Calculus

(see coolest)
so it's name was irrelevant ... after subject bound for done 
subst was key idea we were missing!

Examples:
we'll see other ways to think about this too...

apply a fune by substuting the bound for the bound variable.

argument

\( \lambda x.x \) (\( \lambda x.x \)) (\( \lambda x.x \)) (\( \lambda x.x \))

variable

\( \lambda x.x \) (\( \lambda x.x \)) (\( \lambda x.x \)) (\( \lambda x.x \))

bound

\((\lambda x.x) (\lambda x.x) (\lambda x.x) (\lambda x.x)\)
Some value (i.e. \(x\)).

- Steps (terminates) when you get to small step, call by value (CBV), left-to-right.

\[
\begin{align*}
\text{true} & \leftrightarrow e, e_1, e_2, e_3 \\
\text{false} & \leftrightarrow \overline{e}, e_1, e_2, e_3 \\
\end{align*}
\]

- \(x \in [e] = e\)

Semantics

(don't think `set/Linux`!)

with `e` everywhere, `\(\text{true}\)` always occurs in `\(x\)'s` replacement.

```
\(\text{all} \in [e_2]/x\)
```

I know this notation is weird.

```
Substitution
```

```
``
\((\lambda a. \, \lambda b. \, a)(\lambda c. \, \lambda d. \, c)(\lambda e. \, \lambda f. \, f)\)

\((\lambda b. \, \lambda c. \, \lambda d. \, c)(\lambda e. \, \lambda f. \, f)\)

\((\lambda b. \, \lambda c. \, \lambda d. \, c)(\lambda e. \, \lambda f. \, f)\)

\((\lambda a. \, \lambda b. \, a)(\lambda d. \, \lambda e. \, f. \, f)\)

\((\lambda a. \, \lambda b. \, a)(\lambda d. \, \lambda e. \, f. \, f)\)

\(\lambda a. \, \lambda b. \, a\)

\(\lambda b. \, \lambda d. \, \lambda e. \, f. \, f\)
zatlock@nocatgw

2c.jpg.jpeg

Sun, 13 Oct 2013 13:29:04 -0700

ps541 / hp LaserJet 4200
A couple notes on concrete syntax

(it's equivalent to subject-rewriting, but more efficient than substitution... but)

though real implementations do something...

languages like ocaml, haskell, ada, log...

this is the heart, the asm, the core of


But we can get stuck too...
All computation literally nothing built something that can encode using substitution developed & formalized CBV & C-normal.

---

Great. So what have just done? (Import functions)

Don't even see the code, only see

A weird syntax ok but it's stored test at time

A5 in IMP, we assume AST under it all

A application associates to the left

A function bodies extend right until they hit a "", In general:
modern features (you're right)
- We don't really need all those fancy
  demonstrates model is realistic.
- Expand your mind. (Worfness)

Lambda Encodings

Could pull TMP

Dork, but this is

Does the test...

Church-Turing thesis;

Well, build up enough features to write

How would I even convince you it is?

How the heck is this thing Turing complete?

We left out conditionals, we left out loops.

OK, this is nuts. We left out numbers.
Note: "true" & "false" are 1 & 0. If "true" then "false" = "false".

Conditional: take 3 args: it's 1st and true you get 2nd any, otherwise 3rd.

"false" = x • x • x

"true" = x • x • x

Any implementation of Bool must provide 3 things.

Booleans
Some Boolean operators:

- **and** (\(A \land B\))
- **or** (\(A \lor B\))
- **not** (\(\neg A\))
- **true**
- **false**

Some equivalences:

- \(A \land B \equiv \neg (\neg A \lor \neg B)\)
- \(A \lor B \equiv \neg (\neg A \land \neg B)\)
- \(\neg A \equiv A\)
- \(A \equiv 

P.Ars (ollisiones)

Pairs meaning & things: Construction, Fact, and true.
Turing Tarrit:

Of course, just because we can do something, doesn't mean we should. Beware the

different things at the arith level: input, output, pattern to mean all sorts of

Example: and (first word pair, v2)'s v2
\( \text{null} = \lambda x. \text{snd} (\text{snd} (x)) \)

\( \text{head} = \lambda x. \text{fst} (\text{snd} (x)) \)

\( \text{is-empty} = \text{fst} \)

\( \text{cons} = \lambda h. t. \text{mkpair true (mkpair h t)} \)

\( \text{nil} = \text{mkpair false false} \)

Booils and pairs! build lists at a higher level. Just use turns out, were already encoded enough.

Lists
Note: trail nil does work stuff, but so does following null next in a list etc. (kinda how list work at asm level)