Where we are

- **Done:** Syntax, semantics, and equivalence
  - For a language with little more than loops and global variables

- **Now:** Didn’t IMP leave some things out?
  - In particular: scope, functions, and data structures
  - (Not to mention threads, I/O, exceptions, strings, ...)

Time for a new model...
Data + Code

Higher-order functions work well for scope and data structures

▶ Scope: not all memory available to all code

```plaintext
let x = 1
let add3 y =
    let z = 2 in
    x + y + z
let seven = add3 4
```

▶ Data: Function closures store data. Example: Association “list”

```plaintext
let empty = (fun k -> raise Empty)
let cons k v lst = (fun k' -> if k'=k then v else lst k')
let lookup k lst = lst k
```

(Later: Objects do both too)
Adding data structures

Extending IMP with data structures isn’t too hard:

\[
\begin{align*}
e & ::= \ c \mid x \mid e + e \mid e \ast e \mid (e, e) \mid e.1 \mid e.2 \\
v & ::= \ c \mid (v, v) \\
H & ::= \cdot \mid H, x \mapsto v
\end{align*}
\]

\[
H ; e \Downarrow v \quad \text{all old rules plus:}
\]

\[
\begin{align*}
H ; e_1 \Downarrow v_1 & \quad H ; e_2 \Downarrow v_2 & \quad H ; e \Downarrow (v_1, v_2) & \quad H ; e \Downarrow (v_1, v_2) \\
H ; (e_1, e_2) \Downarrow (v_1, v_2) & \quad H ; e.1 \Downarrow v_1 & \quad H ; e.2 \Downarrow v_2
\end{align*}
\]

Notice:

- We allow pairs of values, not just pairs of integers
- We now have stuck programs (e.g., \(c.1\))
  - Division also causes stuckness
What about functions

But adding functions (or objects) does not work well:

\[
\begin{align*}
e & ::= \ldots \mid \text{fun } x \rightarrow s \\
v & ::= \ldots \mid \text{fun } x \rightarrow s \\
s & ::= \ldots \mid e(e)
\end{align*}
\]

\[
H ; e \Downarrow v
\]

\[
H ; s \rightarrow H' ; s'
\]

Additions:

\[
\begin{align*}
H ; \text{fun } x \rightarrow s \Downarrow \text{fun } x \rightarrow s \\
H ; e_1 \Downarrow \text{fun } x \rightarrow s & \quad H ; e_2 \Downarrow v \\
H ; e_1(e_2) \rightarrow H ; x := v; s
\end{align*}
\]

Does this match “the semantics we want” for function calls?
What about functions

But adding functions (or objects) does not work well:

\[
\begin{align*}
e & ::= \ldots \mid \text{fun } x \rightarrow s \\
v & ::= \ldots \mid \text{fun } x \rightarrow s \\
s & ::= \ldots \mid e(e)
\end{align*}
\]

\[
\begin{align*}
H ; \text{fun } x \rightarrow s & \downarrow \text{fun } x \rightarrow s \\
H ; e_1 \downarrow \text{fun } x \rightarrow s & \quad H ; e_2 \downarrow v \\
H ; e_1(e_2) & \rightarrow H ; x := v; s
\end{align*}
\]

NO: Consider \( x := 1; (\text{fun } x \rightarrow y := x)(2); \text{ans} := x. \)

Scope matters; variable name doesn’t. That is:

- Local variables should “be local”
- Choice of local-variable names should have only local ramifications
Another try

\[
H \downarrow e_1 \Downarrow \text{fun } x \rightarrow s \quad H \downarrow e_2 \Downarrow v \quad y \text{ “fresh”}
\]
\[
\frac{H \downarrow e_1(e_2) \rightarrow H ; y := x \mathbin{;} x := v \mathbin{;} s ; x := y}{H ; e_1(e_2) \rightarrow H ; y := x \mathbin{;} x := v \mathbin{;} s ; x := y}
\]
Another try

\[
\frac{H; e_1 \downarrow \text{fun } x \to s}{H; e_2 \downarrow v \quad y \text{ “fresh”}} \\
H; e_1(e_2) \rightarrow H; y := x; x := v; s; x := y
\]

▶ “fresh” isn’t very IMP-like but okay (think malloc)
Another try

\[
\begin{align*}
H; e_1 \downarrow \text{fun } x \rightarrow s & \quad H; e_2 \downarrow v & \quad y \text{ “fresh”} \\
\hline
H; e_1(e_2) \rightarrow H & \quad y := x; x := v; s; x := y
\end{align*}
\]

- “fresh” isn’t very IMP-like but okay (think malloc)

- not a good match to how functions are implemented
Another try

\[
\begin{align*}
H ; e_1 \Downarrow \text{fun } x \rightarrow s & \quad H ; e_2 \Downarrow v & \quad y \text{ “fresh”} \\
\hline
H ; e_1(e_2) & \rightarrow H ; y := x; x := v; s; x := y
\end{align*}
\]

- “fresh” isn’t very IMP-like but okay (think malloc)
- not a good match to how functions are implemented
- yuck: the way we want to think about something as fundamental as a call?
Another try

\[
H ; e_1 \downarrow \text{fun} \ x \rightarrow s \quad H ; e_2 \downarrow v \quad y \text{ “fresh”}
\]

\[
H ; e_1(e_2) \rightarrow H ; y := x; x := v; s; x := y
\]

- “fresh” isn’t very IMP-like but okay (think malloc)
- not a good match to how functions are implemented
- yuck: the way we want to think about something as fundamental as a call?
- **NO**: wrong model for most functional and OO languages
  - (Even wrong for C if \(s\) calls another function that accesses the global variable \(x\))
The wrong model

\[ \frac{H ; e_1 \downarrow \text{fun } x \rightarrow s}{H ; \ e_1(e_2) \rightarrow H} \frac{H ; e_2 \downarrow v}{y \ 	ext{"fresh"}} \]

\[ H ; e_1(e_2) \rightarrow H ; y := x ; x := v ; s ; x := y \]

\[
\begin{align*}
\textbf{f}_1 & := (\text{fun } x \rightarrow \textbf{f}_2 := (\text{fun } z \rightarrow \text{ans} := x + z)) ; \\
\textbf{f}_1(2) ; \\
x & := 3 ; \\
\textbf{f}_2(4) 
\end{align*}
\]

"Should" set \textit{ans} to 6:

\[ \begin{itemize}
\item \textbf{f}_1(2) \text{ should assign to } \textbf{f}_2 \text{ a function that adds 2 to its argument and stores result in } \textit{ans} \\
\end{itemize} \]

"Actually" sets \textit{ans} to 7:

\[ \begin{itemize}
\item \textbf{f}_2(2) \text{ assigns to } \textbf{f}_2 \text{ a function that adds } \textit{the current value of } x \text{ to its argument} 
\end{itemize} \]
Punch line

Cannot properly model local scope via a global heap of integers.

- Functions are not syntactic sugar for assignments to globals

So let’s build a new model that focuses on this essential concept

- (can add back IMP features later)

Or just borrow a model from Alonzo Church

And drop mutation, conditionals, integers (!), and loops (!)
The Lambda Calculus

The Lambda Calculus:

\[ e ::= \lambda x. e \mid x \mid e \ e \mid v \mid e e \]

\[ v ::= \lambda x. e \]

You apply a function by *substituting* the argument for the *bound variable*

- (There is an equivalent *environment* definition not unlike heap-copying; see future homework)
Substitution is the key operation we were missing:

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

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

$$(\lambda x. x x)(\lambda x. x x) \rightarrow (\lambda x. x x)(\lambda x. x x)$$

After substitution, the bound variable is gone, so its “name” was irrelevant. (Good!)
A Programming Language

Given substitution \( e_1[e_2/x] = e_3 \), we can give a semantics:

\[
\begin{align*}
\frac{e \rightarrow e'}{e} & \\
\frac{e[v/x] = e'}{e[v/x]} & \\
(\lambda x. e) v & \rightarrow e' \\
\frac{e_1 \rightarrow e'_1}{e_1} & \\
\frac{e_2 \rightarrow e'_2}{e_2} & \\
\frac{e_1 e_2 \rightarrow e'_1 e_2}{e_1 e_2} & \\
\frac{v e_2 \rightarrow v e'_2}{v e_2} & 
\end{align*}
\]

A small-step, \textit{call-by-value (CBV)}, left-to-right semantics

- Terminates when the “whole program” is some \( \lambda x. e \)

But (also) gets stuck when there’s a \textit{free variable} “at top-level”

- Won’t “cheat” like we did with \( H(x) \) in IMP because scope is what we’re interested in

This is the “heart” of functional languages like Caml

- But “real” implementations don’t substitute; they do something \textit{equivalent}
Roadmap

- Motivation for a new model (done)
- CBV lambda calculus using substitution (done)
- Notes on concrete syntax
- Simple Lambda encodings (it’s Turing complete!)
- Other reduction strategies
- Defining substitution
Concrete-Syntax Notes

We (and Caml) resolve concrete-syntax ambiguities as follows:

1. $\lambda x. e_1 e_2$ is $\left( \lambda x. e_1 e_2 \right)$, not $\left( \lambda x. e_1 \right) e_2$
2. $e_1 e_2 e_3$ is $(e_1 e_2) e_3$, not $e_1 (e_2 e_3)$
   ▶ Convince yourself application is not associative

More generally:

1. Function bodies extend to an unmatched right parenthesis
   Example: $\left( \lambda x. y(\lambda z. z)w \right)q$
2. Application associates to the left
   Example: $e_1 e_2 e_3 e_4$ is $\left( \left( e_1 e_2 \right) e_3 \right) e_4$

▶ Like in IMP, assume we really have ASTs
   (with non-leaves labeled $\lambda$ or “application”)
▶ Rules may seem strange at first, but it’s the most convenient concrete syntax
   ▶ Based on 70 years experience
Lambda Encodings

Fairly crazy: we left out constants, conditionals, primitives, and data structures

In fact, we're *Turing complete* and can *encode* whatever we need (just like assembly language can)

Motivation for encodings:

- Fun and mind-expanding
- Shows we aren’t oversimplifying the model (“numbers are syntactic sugar”)
- Can show languages are *too expressive* (e.g., unlimited C++ template instantiation)

Encodings are also just “(re)definition via translation”
Encoding booleans

The “Boolean ADT”

- There are two booleans and one conditional expression.
- The conditional takes 3 arguments (e.g., via currying). If the first is one boolean it evaluates to the second. If it’s the other boolean it evaluates to the third.
Encoding booleans

The “Boolean ADT”

- There are two booleans and one conditional expression.
- The conditional takes 3 arguments (e.g., via currying). If the first is one boolean it evaluates to the second. If it’s the other boolean it evaluates to the third.

Any set of three expressions meeting this specification is a proper encoding of booleans

"true" = λx.λy.x
"false" = λx.λy.y
"if" = λb.λt.λf.btf

Example: "if" "true" v₁ v₂ → v₁
Encoding booleans

The “Boolean ADT”

- There are two booleans and one conditional expression.
- The conditional takes 3 arguments (e.g., via currying). If the first is one boolean it evaluates to the second. If it’s the other boolean it evaluates to the third.

Any set of three expressions meeting this specification is a proper encoding of booleans

Here is one of an infinite number of encodings:

- “true” \[\lambda x. \lambda y. x\]
- “false” \[\lambda x. \lambda y. y\]
- “if” \[\lambda b. \lambda t. \lambda f. b \, t \, f\]

Example: “if” “true” \(v_1 \, v_2 \rightarrow^* \, v_1\)
Evaluation Order Matters

Careful: With CBV we need to “thunk”. . .

“if” “true” (λx. x) ((λx. x x)(λx. x x))
\[\text{an infinite loop}\]

diverges, but

“if” “true” (λx. x) (λz. ((λx. x x)(λx. x x)) z))
\[\text{a value that when called diverges}\]

does not
Encoding Pairs

The “pair ADT”:

- There is 1 constructor (taking 2 arguments) and 2 selectors
- 1st selector returns the 1st arg passed to the constructor
- 2nd selector returns the 2nd arg passed to the constructor

Example:

\[
\text{snd} \left( \text{fst} \left( \text{mkpair} \left( \text{mkpair} v_1 v_2 \right) v_3 \right) \right) \rightarrow \ast v_2
\]
Encoding Pairs

The “pair ADT”:

- There is 1 constructor (taking 2 arguments) and 2 selectors
- 1st selector returns the 1st arg passed to the constructor
- 2nd selector returns the 2nd arg passed to the constructor

```
“mkpair”  λx. λy. λz. z x y
“fst”     λp. p(λx. λy. x)
“snd”     λp. p(λx. λy. y)
```

Example:

```
“snd” (“fst” (“mkpair” (“mkpair” v₁ v₂) v₃)) →* v₂
```
Reusing Lambdas

Is it weird that the encodings of Booleans and pairs both used $\lambda x. \lambda y. x$ and $\lambda x. \lambda y. y$ for different purposes?

Is it weird that the same bit-pattern in binary code can represent an int, a float, an instruction, or a pointer?

Von Neumann: Bits can represent (all) code and data

Church (?): Lambdas can represent (all) code and data

Beware the “Turing tarpit”
Encoding Lists

Rather than start from scratch, notice that booleans and pairs are enough to encode lists:

- Empty list is “mkpair” “false” “false”
- Non-empty list is $\lambda h. \lambda t. \text{“mkpair” “true” (“mkpair” } h t)$
- Is-empty is ...
- Head is ...
- Tail is ...

Note:

- Not too far from how lists are implemented
- Taking “tail” (“tail” “empty”) will produce some lambda
  - Just like, without page-protection hardware, null->tail->tail would produce some bit-pattern
Encoding Recursion

Some programs diverge, but can we write *useful* loops? Yes!

▶ Write a function that takes an \( f \) and calls it in place of recursion

▶ Example (in enriched language):

\[
\lambda f.\lambda x. \text{if } (x = 0) \text{ then } 1 \text{ else } (x \ast f(x - 1))
\]

▶ Then apply “fix” to it to get a recursive function:

\[
\text{“fix” } \lambda f.\lambda x. \text{if } (x = 0) \text{ then } 1 \text{ else } (x \ast f(x - 1))
\]

▶ “fix” \( \lambda f.e \) reduces to something roughly equivalent to \( e[ (\text{“fix” } \lambda f.e) / f ] \), which is “unrolling the recursion once” (and further unrollings will happen as necessary)

▶ The details, especially for CBV, are icky; the point is it’s possible and you define “fix” only once

▶ Not on exam:

\[
\text{“fix” } \lambda g. (\lambda x.g(\lambda y.xxy))(\lambda x.g(\lambda y.xxy))
\]
Encoding Recursion

Some programs diverge, but can we write *useful* loops? Yes!

- Write a function that takes an \( f \) and calls it in place of recursion
  - Example (in enriched language):
    \[
    \lambda f. \lambda x. \text{if } (x = 0) \text{ then } 1 \text{ else } (x \ast f(x - 1))
    \]

- "fix" \( \lambda f. \text{reduces to} \)
  - something roughly equivalent to \( e[\text{fix } \lambda f.e]/f \), which is "unrolling the recursion once" (and further unrollings will happen as necessary)
  - The details, especially for CBV, are icky; the point is it's possible and you define "fix" only once

- Not on exam:
  - "fix" \( \lambda g. (\lambda x.g(\lambda y.xxy))(\lambda x.g(\lambda y.xxy)) \)
Encoding Recursion

Some programs diverge, but can we write *useful* loops? Yes!

- Write a function that takes an \( f \) and calls it in place of recursion
  - Example (in enriched language):
    \[
    \lambda f. \lambda x. \text{if } (x = 0) \text{ then } 1 \text{ else } (x \times f(x - 1))
    \]
- Then apply “fix” to it to get a recursive function:
  - “fix” \( \lambda f. \lambda x. \text{if } (x = 0) \text{ then } 1 \text{ else } (x \times f(x - 1)) \)
Encoding Recursion

Some programs diverge, but can we write *useful* loops? Yes!

- Write a function that takes an \( f \) and calls it in place of recursion
  - Example (in enriched language):
    \[
    \lambda f. \lambda x. \text{if } (x = 0) \text{ then } 1 \text{ else } (x \ast f(x - 1))
    \]
- Then apply “fix” to it to get a recursive function:
  - “fix” \( \lambda f. \lambda x. \text{if } (x = 0) \text{ then } 1 \text{ else } (x \ast f(x - 1)) \)
  - “fix” \( \lambda f. e \) reduces to something roughly equivalent to \( e[(\text{“fix”} \lambda f. e)/f] \), which is “unrolling the recursion once” (and further unrollings will happen as necessary)
Encoding Recursion

Some programs diverge, but can we write *useful* loops? Yes!

- Write a function that takes an \( f \) and calls it in place of recursion
  - Example (in enriched language):
    \[
    \lambda f. \lambda x. \text{if } (x = 0) \text{ then } 1 \text{ else } (x * f(x - 1))
    \]

- Then apply “fix” to it to get a recursive function:
  - “fix” \( \lambda f. \lambda x. \text{if } (x = 0) \text{ then } 1 \text{ else } (x * f(x - 1)) \)
  - “fix” \( \lambda f. e \) reduces to *something roughly equivalent to* \( e[(\text{“fix” } \lambda f. e)/f] \), which is “unrolling the recursion once” (and further unrollings will happen as necessary)

- The details, especially for CBV, are icky; the point is it’s possible and you define “fix” only once

- Not on exam:
  - “fix” \( \lambda g. (\lambda x. g (\lambda y. x x y)))(\lambda x. g (\lambda y. x x y)) \)
Encoding Arithmetic Over Natural Numbers

How about arithmetic?

- Focus on non-negative numbers, addition, is-zero, etc.

How I would do it based on what we have so far:

- Lists of booleans for binary numbers
- Zero can be the empty list
- Use fix to implement adders, etc.
- Like in hardware except fixed-width avoids recursion
- Or just use list length for a unary encoding
- Addition is list append

But instead everybody always teaches Church numerals. Why?

- Tradition? Some sense of professional obligation?
- Better reason: You don’t need fix: Basic arithmetic is often encodable in languages where all programs terminate

In any case, we’ll show some basics “just for fun”
Encoding Arithmetic Over Natural Numbers

How about arithmetic?

- Focus on non-negative numbers, addition, is-zero, etc.

How I would do it based on what we have so far:

- Lists of booleans for binary numbers
  - Zero can be the empty list
  - Use fix to implement adders, etc.
  - Like in hardware except fixed-width avoids recursion
Encoding Arithmetic Over Natural Numbers

How about arithmetic?
▶ Focus on non-negative numbers, addition, is-zero, etc.

How I would do it based on what we have so far:
▶ Lists of booleans for binary numbers
  ▶ Zero can be the empty list
  ▶ Use fix to implement adders, etc.
  ▶ Like in hardware except fixed-width avoids recursion
▶ Or just use list length for a unary encoding
  ▶ Addition is list append
Encoding Arithmetic Over Natural Numbers

How about arithmetic?

- Focus on non-negative numbers, addition, is-zero, etc.

How I would do it based on what we have so far:

- Lists of booleans for binary numbers
  - Zero can be the empty list
  - Use fix to implement adders, etc.
  - Like in hardware except fixed-width avoids recursion
- Or just use list length for a unary encoding
  - Addition is list append

But instead everybody always teaches Church numerals. Why?

- Tradition? Some sense of professional obligation?
- Better reason: You don’t need fix: Basic arithmetic is often encodable in languages where all programs terminate
- In any case, we’ll show some basics “just for fun”
Church Numerals

“0” \( \lambda s. \lambda z. z \)
“1” \( \lambda s. \lambda z. s\ z \)
“2” \( \lambda s. \lambda z. s\ (s\ z) \)
“3” \( \lambda s. \lambda z. s\ (s\ (s\ z)) \)
...

- Numbers encoded with two-argument functions
- The “number \( i \)” composes the first argument \( i \) times, starting with the second argument
  - \( z \) stands for “zero” and \( s \) for “successor” (think unary)
- The trick is implementing arithmetic by cleverly passing the right arguments for \( s \) and \( z \)
Church Numerals

“0” \( \lambda s. \lambda z. z \)

“1” \( \lambda s. \lambda z. s \ z \)

“2” \( \lambda s. \lambda z. s \ (s \ z) \)

“3” \( \lambda s. \lambda z. s \ (s \ (s \ z)) \)

“successor” \( \lambda n. \lambda s. \lambda z. s \ (n \ s \ z) \)

successor: take “a number” and return “a number” that (when called) applies \( s \) one more time
Church Numerals

“0”  \( \lambda s. \lambda z. z \)
“1”  \( \lambda s. \lambda z. s z \)
“2”  \( \lambda s. \lambda z. s (s z) \)
“3”  \( \lambda s. \lambda z. s (s (s z)) \)

“successor”  \( \lambda n. \lambda s. \lambda z. s (n s z) \)
“plus”  \( \lambda n. \lambda m. \lambda s. \lambda z. n s (m s z) \)

plus: take two “numbers” and return a “number” that uses one number as the zero argument for the other
Church Numerals

“0” \[ \lambda s. \lambda z. z \]

“1” \[ \lambda s. \lambda z. s \, z \]

“2” \[ \lambda s. \lambda z. s \,(s \, z) \]

“3” \[ \lambda s. \lambda z. s \,(s \,(s \, z)) \]

“successor” \[ \lambda n. \lambda s. \lambda z. s \,(n \, s \, z) \]

“plus” \[ \lambda n. \lambda m. \lambda s. \lambda z. n \, s \,(m \, s \, z) \]

“times” \[ \lambda n. \lambda m. m \,(\text{“plus” } n) \text{ “zero”} \]

times: take two “numbers” \( m \) and \( n \) and pass to \( m \) a function that adds \( n \) to its argument (so this will happen \( m \) times) and “zero” (where to start the \( m \) iterations of addition)
Church Numerals

“0” \( \lambda s. \lambda z. z \)

“1” \( \lambda s. \lambda z. s \, z \)

“2” \( \lambda s. \lambda z. s \, (s \, z) \)

“3” \( \lambda s. \lambda z. s \, (s \, (s \, z)) \)

“successor” \( \lambda n. \lambda s. \lambda z. s \, (n \, s \, z) \)

“plus” \( \lambda n. \lambda m. \lambda s. \lambda z. n \, s \, (m \, s \, z) \)

“times” \( \lambda n. \lambda m. m \, (\text{“plus”} \, n) \, \text{“zero”} \)

“isZero” \( \lambda n. \, (\lambda x. \, \text{“false”}) \, \text{“true”} \)

isZero: an easy one, see how the two arguments will lead to the correct answer
Church Numerals

“0” \[ \lambda s. \lambda z. z \]
“1” \[ \lambda s. \lambda z. s \, z \]
“2” \[ \lambda s. \lambda z. s \,(s \, z) \]
“3” \[ \lambda s. \lambda z. s \,(s \,(s \, z)) \]

“successor” \[ \lambda n. \lambda s. \lambda z. s \,(n \, s \, z) \]
“plus” \[ \lambda n. \lambda m. \lambda s. \lambda z. n \, s \,(m \, s \, z) \]
“times” \[ \lambda n. \lambda m. m \,(\text{“plus” } n) \text{ “zero”} \]
“isZero” \[ \lambda n. \, (\lambda x. \text{ “false”}) \text{ “true”} \]

“predecessor” (with 0 sticky) the hard one; see Wikipedia
“minus” similar to times with pred instead of plus
“isEqual” subtract and test for zero
Roadmap

- Motivation for a new model (done)
- CBV lambda calculus using substitution (done)
- Notes on concrete syntax (done)
- Simple Lambda encodings (it’s Turing complete!) (done)
- Other reduction strategies
- Defining substitution

Then start type systems

- Later take a break from types to consider first-class continuations and related topics