Last time
- Introduced compact, powerful programming language: untyped lambda calculus (Church, 1930’s)
- All values are first-class, anonymous functions
- Syntax: $e ::= x \mid e_0 e_1 \mid \lambda x e_0$
- Missing:
  - multiple arguments
  - local variables
  - primitive values (booleans, integers, …)
  - data structures
  - recursive functions (this lecture)
  - assignment

Notation
- Lambda calculus is programming language and a mathematical notation for writing down functions
- When programming language: fully parenthesized
- Identity function program: $(\lambda x x)$
- Mathematical identity function: $f(x) = x$ operating on elements of set $T$ is
  $f = \lambda x \in T. x$

Lambda Calculus References
- Gunter (recommended text)
- Stoy, Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory
- Davie, An Introduction to Functional Programming Systems using Haskell
- Barendregt, The Lambda Calculus: Its Syntax and Semantics

Recursion
- How to express recursive functions?
- Consider factorial function:
  $$\text{fact}(x) = \begin{cases} 1 & \text{if } x = 0 \\ x \times \text{fact}(x-1) & \text{if } x > 0 \end{cases}$$
- Can’t write this recursive definition!
  $\text{FACT} \triangleq (\lambda x (\text{IF} (\text{ZERO?} x) 1 (* x (\text{FACT} (- x 1))))))$
- This is an equation, not a definition!
- Meaning:
  $\text{FACT}$ stands for a function that, if applied to an argument, gives the same result as does
  $(\lambda x (\text{IF} (\text{ZERO?} x) 1 (* x (\text{FACT} (- x 1))))))$
Defining a recursive function

- Idea: introduce a function just like \( \text{FACT} \) except that it has an extra argument that should be passed a function \( f \) such that \(((f \ f) \ x)\) computes factorial of \( x \)

\[
\text{FACT}' = ((\lambda \ x (\text{IF} (\text{ZERO?} \ x) 1 (* x ((f \ f) (- x 1)))))
\]

- Now define \( \text{FACT} \triangleq (\text{FACT}' \ \text{FACT}') \)
- \( \text{FACT} \) diverges but its application to a number does not!

Evaluation of FACT

\[
\text{FACT}' = ((\lambda \ (\lambda \ x (\text{IF} (\text{ZERO?} \ x) 1 (* x ((f \ f) (- x 1)))))
\]

- Now define \( \text{FACT} \triangleq (\text{FACT}' \ \text{FACT}') \)
- \( \text{FACT} \) diverges but its application to a number does not!

Generalizing

\[
\text{FACT}' = ((\lambda \ x (\text{IF} (\text{ZERO?} \ x) 1 (* x ((f \ f) (- x 1)))))
\]

- The recursion-removal transformation:
  - Add an extra argument variable \( f \) to the recursive function
  - Replace all internal references to the recursive function with an application of argument variable to itself
  - Replace all external references to the recursive function as application of it to itself
- Can this transformation itself be abstracted?

Fixed point operator

- Suppose we had an operator \( Y \) that found the fixed point of functions:
  \[
  ((Y \ f) \ x) = (f (Y \ f) x)
  \]
- \( (Y f) = f (Y f) \)
- Now write a recursive function as a function that takes itself as an argument:

\[
\text{FACTEQN} = \lambda f (\lambda x ((f f) (x (- x 1))))
\]

- Idea: \( \text{FACT} = (\text{FACTEQN} \ \text{FACTEQN}) \)

Is \( Y \) computable?

- Can we express the \( Y \) operator as a lambda expression? Maybe not!
- \# functions from \( Z \) to boolean: \( 2^{|Z|} \)
- \# functions: \( \geq 2^{|Z|} \)
- Set of all functions is uncountably infinite
- Set of computable functions is the same size as \( Z \): countably infinite
- Only an infinitesimal fraction of all functions are computable
- No reason to expect an arbitrary function to be computable! (e.g., halting function is not)

Definition of \( Y \)

- \( Y \) is a solution to this equation:
  \[
  Y = (\lambda f \ (f \ (Y \ f)))
  \]
- Now, apply our recursion-removal trick:
  \[
  Y' = (\lambda x ((f f) (x (- x 1))))
  \]
- Traditional form for \( Y \) (requires call-by-name):

\[
Y = (\lambda x ((f f) (x x)) (x x))
\]
**Problems with substitution**

- Rule for evaluating an application:
  \[(\lambda x e_0) e_1 \rightarrow e_1 \{ e_2 / x \} \]
- Can’t just stick \(e_2\) in for every occurrence of variable \(x\):
  \[(x (\lambda x x)) \{ (b a) / x \} = ((b a) (\lambda x (b a)))\]
- Can’t just stick \(e_2\) in for every occurrence of variable \(x\) outside any lambda over \(x\):
  \[y (\lambda x (x y)) \{ x / y \} = (x (\lambda x (x x)))\]

**Free variables**

- The function \(\text{FV} [e]\) gives the set of all free variables (unbound identifiers) in \(e\)
- Special brackets \(\llbracket \cdot \rrbracket\) are called *semantic brackets*; wrap syntactic arguments
  - \(\text{FV} [e]\) operates on abstract syntax tree for \(e\), not result of evaluating \(e\)
  - sometimes name of function is omitted: \(\llbracket e \rrbracket\)
- Inductive definition of \(\text{FV} [\cdot]\):
  \[
  \begin{align*}
  \text{FV}[x] &= \{ x \} \\
  \text{FV}[e_0 e_1] &= \text{FV}[e_0] \cup \text{FV}[e_1] \\
  \text{FV}[\lambda x e] &= \text{FV}[e] - \{ x \}
  \end{align*}
  \]

**Substitution into abstraction**

\[
\begin{align*}
(\lambda y e_0) \{ e_1 / x \} &\rightarrow (\lambda y e_0 \{ e_1 / x \}) \\
(\lambda y e_0) \{ e_1 / x \} &\rightarrow (\lambda y' e_0 \{ y' / y \} e_1 / x) \\
(\lambda y e_0) \{ e_1 / x \} &\rightarrow (\lambda y' e_0 \{ y' / y \} e_1 / x) \\
\end{align*}
\]

**Defining substitution inductively**

Let \(e' / x \rightarrow e''\) mean “\(e'\) can be the result of substituting \(e'\) for \(x\)"

\[
\begin{align*}
x \{ e / x \} &\rightarrow e \\
y \{ e / x \} &\rightarrow y \quad \text{(if } y \neq x) \\
(e_0 e_1) \{ e_2 / x \} &\rightarrow (e_0 \{ e_2 / x \} e_1 \{ e_2 / x \}) \\
(\lambda x e_0) \{ e_1 / x \} &\rightarrow (\lambda x e_0)
\end{align*}
\]

**Variable binding**

- Which variable is denoted by an identifier? \(\lambda x (\lambda x x)\)
- Lexical scope:
  \(\lambda p (\lambda q ((\lambda p (p q)) (\lambda r (p r))))\)

\[
\begin{align*}
\lambda p (\lambda q ((\lambda p (p q)) (\lambda r (p r)))) &\rightarrow (\lambda p (\lambda q) (\lambda r (p r))) \\
(\lambda p (\lambda q (\lambda r (p r)))) &\rightarrow (\lambda p (\lambda q) (\lambda r (p r)))
\end{align*}
\]

**Stay diagram**
Renaming

- Intuitively, meaning of lambda expression does not depend on name of argument variable: \((\lambda x x) = (\lambda y y) = (\lambda \, \bullet \, \bullet)\)
- \((\lambda x (\lambda y (y x))) = (\lambda p (\lambda q (q p))) = (\lambda y (\lambda x (x y))) = (\lambda \, \bullet \, (\lambda \, \bullet \, \bullet \, \bullet))\)
- \((\lambda y e_0) \{e_1 / x\} = (\lambda y' e_0 \{y/y\} \{e_1 / x\})\)
- \(\alpha\)-reduction: \((\lambda y e_0) \xrightarrow{\alpha} (\lambda y' e_0 \{y/y\})\)

\(\alpha\) reduction: \((\lambda y e_0) \rightarrow (\lambda y' e_0 \{y/y\})\)
where \(y' \not\in \text{FV}[e_0]\)
(does not change Stoy diagram)

Equivalence

- Two lambda expressions are \(\alpha\)-equivalent if they can be converted to each other using \(\alpha\)-reductions / have the same Stoy diagrams

\((\lambda p (\lambda q (q p))) \rightarrow (\lambda x (\lambda q (q x))) \rightarrow (\lambda x (\lambda y (y x)))\)
\((\lambda \, \bullet \, (\lambda \, \bullet \, \bullet \, \bullet)) \rightarrow (\lambda \, \bullet \, (\lambda \, \bullet \, \bullet \, \bullet)) \rightarrow (\lambda \, \bullet \, (\lambda \, \bullet \, \bullet \, \bullet))\)

- Lambda expressions form equivalence classes defined by their Stoy diagrams