Lambda Calculus

- So far, we’ve explored some simple but non-interesting languages
  - language of arithmetic expressions
  - IMP (arithmetic + while loops)
- We now turn our attention to a simple but interesting language
  - Turing complete (can express loops and recursion)
  - Higher-order (functional objects are values)
  - Interesting variable binding and scoping issues
  - Foundation for many real-world programming languages
    - Lisp, Scheme, ML, Haskell, Dylan, ....
Intuition

Suppose we want to describe a function that adds three to any input:

- `plus3 x = succ (succ (succ x))`
- Read “`plus3` is a function which, when applied to any number `x`, yields the successor of the successor of the successor of `x`”
- Note that the function which adds 3 to any number need not be named `plus3`; the name “`plus3`” is just a convenient shorthand for naming this function

\[
(plus3 \ x) \ (succ \ 0) \equiv
((\lambda \ x. \ (succ \ (succ \ (succ \ 0)))) \ (succ \ 0))
\]
Basics

- There are two new primitive syntactic forms:
  - \( \lambda x. t \)
    "The function which when given a value \( v \), yields \( t \) with \( v \) substituted for \( x \) in \( t \)."
  - \((t_1 t_2)\)
    "the function \( t_1 \) applied to argument \( t_2 \)"

Key point: functions are anonymous: they don’t need to be named (e.g., plus3). For convenience we’ll sometimes write:

\[
\text{plus3 } x \equiv \lambda x. (\text{succ} (\text{succ} (\text{succ} x)))
\]

but the naming is a metalanguage operation.
Abstractions

Consider the abstraction:

\[ g \equiv \lambda f. (f \ (f \ (\text{succ} \ 0))) \]

The argument \( f \) is used in a function position (in a call).

We call \( g \) a higher-order function because it takes another function as an input.

Now,

\[ (g \ \text{plus3}) = (\lambda f. (f \ (f \ (\text{succ} \ 0)))) \]

\[ (\lambda x . (\text{succ} \ (\text{succ} \ (\text{succ} \ x)))) \]

\[ = (((\lambda x. (\text{succ} \ (\text{succ} \ (\text{succ} \ x)))) \ (\text{succ} \ 0)))) \]

\[ = (((\lambda x. (\text{succ} \ (\text{succ} \ (\text{succ} \ x)))) \ (\text{succ} \ (\text{succ} \ (\text{succ} \ (\text{succ} \ 0)))))) \]

\[ = (\text{succ} \ (\text{succ} \ (\text{succ} \ (\text{succ} \ (\text{succ} \ (\text{succ} \ (\text{succ} \ (\text{succ} \ (\text{succ} \ (\text{succ} \ (\text{succ} \ 0))))))))))) \]
Abstractions

Consider

double ≡ λ f. λ y. (f (f y))

The term yielded by applying double is another function (λ y. (f (f y))

Thus, double is also a higher-order function because it returns a function when applied to an argument.
Example

\((\text{double plus3 0})\)

\[= ((\lambda f. \lambda y. (f (f y)))
   (\lambda x. (\text{succ (succ (succ x)))))) 0)\]
\[= ((\lambda y. ((\lambda x. (\text{succ (succ (succ x))))))
   ((\lambda x. (\text{succ (succ (succ x)))))) y))
   0)\]
\[= ((\lambda x. (\text{succ (succ (succ x)))))
   (\text{succ (succ (succ 0))))\)
\[= (\text{succ (succ (succ (succ (succ (succ 0))))))}\]
Key Issues

- How do we perform substitution:
  - how do we bind “free variables”, the variables that are non-local in the function
  - Think about the occurrences of f in
    \[ \lambda y. (f (f y)) \]

- How do we perform application:
  - There may be several different application subterms within a larger term.
  - How do we decide the order to perform applications?
Pure Lambda Calculus

- The only value is a function
  - Variables denote functions
  - Functions always take functions as arguments
  - Functions always return functions as results
- Minimalist
  - Can express essentially all modern programming constructs
  - Can apply syntactic reasoning techniques (e.g. operational semantics) to understand behavior.
Scope

- The $\lambda$ abstraction $\lambda x. t$ binds variable $x$.
- The scope of the binding is $t$.
- Occurrences of $x$ that are not within the scope of an abstraction binding $x$ are said to be free:
  - $\lambda x. \lambda y. (x y z)$
  - $\lambda x. ((\lambda y. z y) y)$
- Occurrences of $x$ that are within the scope of an abstraction binding $x$ are said to be bound by the abstraction.
Free Variables

- Intuitively, the free variables of an expression are "non-local" variables.
- Define $FV(M)$ formally thus:
  - $FV(x) = \{x\}$
  - $FV(M_1 \ M_2) = FV(M_1) \cup FV(M_2)$
  - $FV(\lambda \ x. M) = FV(M) - \{x\}$
- Free variables become bound after substitution.
- But, if proper care is not taken, this may lead to unexpected results:
  - $(\lambda x. \lambda y. y \ x) \ y = \lambda y. \ y \ y$
- We say that term $M$ is $\alpha$-congruent to $N$ if $N$ results from $M$ by a series of changes to bound variables:
  - $\lambda x. (x \ y)$ $\alpha$-congruent to $\lambda z. (z \ y)$ not $\alpha$-congruent to $\lambda y. (y \ y)$
  - $\lambda x. x \ (\lambda x. x)$ $\alpha$-congruent to $\lambda x'. x' \ (\lambda x. x)$ and $\alpha$-congruent to $\lambda x'. x' (\lambda x''. x'')$
Substitution

- $\lambda x. M \alpha$-congruent to $\lambda y. M[y/x]$ if $y$ is not free or bound in $M$.

- Define this more generally:
  - Let $x$ be a variable, and $M$ and $N$ expressions. Then $[M/x]N$ is the expression $N'$:
    - $N$ is a variable:
      - $N = x$ then $N' = M$
      - $N \neq x$ then $N' = N$
    - $N$ is an application ($Y Z$):
      - $N' = ([M/x]Y) ([M/x]Z)$
Substitution (cont)

- $N$ is an abstraction $\lambda y.Y$:
  - $y = x$ then $N' = N$
  - $y \neq x$ then:
    - $x$ does not occur free in $Y$ or if $y$ does not occur free in $M$:
      - $N' = \lambda y.[M/x]Y$
    - $x$ does occur free in $Y$ and $y$ does occur free in $M$:
      - $N' = \lambda z.[M/x]([z/y]Y)$ for fresh $z$
Example

- $\lambda p. (\lambda q. (\lambda p.p(p \ q))(\lambda r. (+ p r)))(+ p 4))$ 2
- $[(+ p 4)/q][(\lambda p.p(p \ q))(\lambda r. (+ p r))]$
- $[(+ p 4)/q](\lambda p.p(p \ q)))([(+ p 4)/q](\lambda r. (+ p r))$ (by case 2)
- $[(+ p 4)/q](\lambda p.p(p \ q)))(\lambda r.(+ p r))$ (by case 3.2.1 since q does not occur free in (+ p r)
- $\lambda a.[(+ p 4)/q][(a/p)(p(p \ q)))](\lambda r. (+ p r))$ (by case 3.3.2)
- $\lambda a.a (a (+ p 4)))(\lambda r. (+ p r))$
Operational Semantics

- **Values:**
  - \( \lambda x. \, t \)

- **Computation rule:**
  - \( ((\lambda x. \, t) \, v) \rightarrow t[v/x] \)

- **Congruence rules**
  - \( t_1 \rightarrow t_1' \)
    - \( (t_1 \, t_2) \rightarrow (t_1' \, t_2) \)
  
  - \( t_2 \rightarrow t_2' \)
    - \( (v \, t_2) \rightarrow (v \, t_2') \)

- \( x \) not free in \( t \)
  - \( \lambda x. \, (t \, x) \rightarrow t \)

The first computation rule is referred to as the \( \beta \)-substitution or \( \beta \)-conversion rule. \( ((\lambda x. \, t_1) \, t_2) \) is called a \( \beta \)-redex.

The last congruence rule is referred as the \( \eta \)-conversion rule.

\( (\lambda x. \, (t \, x)) \)
where \( x \) not in \( FV(t) \) is an \( \eta \)-redex

\( \eta \)-conversion related to notion of function extensionality. Why?
Multiple arguments

- The $\lambda$ calculus has no built-in support to handle multiple arguments.
- However, we can interpret $\lambda$ terms that when applied yield another $\lambda$ term as effectively providing the same effect:
- **Example:**
  - double $\equiv \lambda f. \lambda x. (f (f x))$
    - We can think of double as a two-argument function.
- Representing a multi-argument function in terms of single-argument higher-order functions is known as currying.
Programming Examples: Booleans

true ≡ λ t. λ f. t
false ≡ λ t. λ f. f

(true v w) → ((λ t.λ f. t) v) w) →
   ((λ f. v) w) →
     v

(false v w) → ((λ t.λ f. f) v) w) →
   ((λ f. f) w) →
     w
Booleans (cont)

- **not** $\equiv \lambda b. b \text{ false true}

  The function that returns true if $b$ is false, and false if $b$ is true.

- **and** $\equiv \lambda b. \lambda c. b \text{ c false}

  The function that given two Boolean values ($v$ and $w$) returns $w$ if $v$ is true and false if $v$ is false. Thus, $(\text{and } v \ w)$ yields true only if both $v$ and $w$ are true.
Pairs

We can encode common operations on pairs thus:

- $$\text{pair} \equiv \lambda f \cdot \lambda s \cdot \lambda b \cdot b f s$$
- $$\text{fst} \equiv \lambda p \cdot p \text{ true}$$
- $$\text{snd} \equiv \lambda p \cdot p \text{ false}$$

Example:

$$(\text{fst} (\text{pair} v w)) \rightarrow$$
$$(\text{fst} ((\lambda f \cdot \lambda s \cdot \lambda b \cdot b f s) v w)) \rightarrow$$
$$(\text{fst} (\lambda s \cdot \lambda b \cdot b v s) w)) \rightarrow$$
$$((\lambda p \cdot p \text{ true}) (\lambda b \cdot (b v w))) \rightarrow$$
$$((\lambda b \cdot (b v w)) \text{ true}) \rightarrow$$
$$(\text{true} v w) \rightarrow^* v$$
Numbers (Church Numerals)

- There are no explicit operations to manipulate numbers
- Encode numbers using higher-order functions:
  - zero \( \equiv \lambda s. \lambda z. z \)
  - one \( \equiv \lambda s. \lambda z. (s \ z) \)
  - two \( \equiv \lambda s. \lambda z. (s \ (s \ z)) \)

Read “s” as successor and “z” as zero
Numbers

- **succ**: \( \lambda n. \lambda s. \lambda z . s (n \ s \ z) \)
  
  A function that takes \( s \) and \( z \) and applies \( s \) repeatedly to \( z \).

- **plus**: \( \lambda m. \lambda n. \lambda s. \lambda z . m \ s (n \ s \ z) \)
  
  Takes two Church numerals and yields another Church numeral that given \( s \) and \( z \) applies \( s \) iterated \( n \) times to \( z \) and then applies \( s \) iterated \( m \) times to the result.
Example

\[(\text{plus one two succ zero}) \rightarrow\]

\[(\text{plus} \ (\lambda \ s. \lambda \ z. \ (s \ z)) \ (\lambda \ s. \lambda \ z. \ (s \ (s \ z))) \]
\[\text{succ zero} \rightarrow\]

\[(\lambda \ s. \lambda \ z. \ ((\lambda \ s. \lambda \ z. \ (s \ z)) \ s \]
\[\text{(s succ zero)} \rightarrow\]

\[((\lambda \ s. \lambda \ z. \ (s \ z)) \]
\[\text{succ}\]
\[((\lambda \ s. \lambda \ z. \ (s \ (s \ z)))) \text{ succ zero}) \rightarrow\]

\[((\lambda \ s. \lambda \ z. \ (s \ z)) \]
\[\text{succ}\]
\[(\text{succ (succ zero)})) \rightarrow\]

\[(\text{succ (succ (succ zero))})\]