Class Information

- Please check out the midterm sample solution. If you want to challenge the grading, please talk to us by April 16. Thanks.
- Second project has been posted on Saturday.
- Seventh homework will be posted tomorrow.

Lambda calculus

λ-terms (*wffs*) are inductively defined.

A λ-terms is:

- a variable x
- (λx.M) where x is a variable and M is λ-term (*abstraction*)
- (M N) where M and N are λ-terms (*application*)

Abbreviations (Notational conveniences):

- function application is left associative
  (f g z) is ((f g) z)
- function application has precedence over function abstraction — "function body" extends as far to the right as possible
  λx.yz is (λx.(yz))
- "multiple" arguments
  λxy.z is (λx.(λy.z))

Free and bound variables

Abstraction (λx. M) “binds” variable x in “body” M. You can think of this as a declaration of variable x with scope M.

(λ y. y z) y

free occurrence

binding occurrence

bound occurrence

Let M, N be λ-terms and x is a variable. The set of *free variables of M*, free(M), is defined inductively as follows:

- free(x) = {x}
- free(M N) = free(M) $\cup$ free(N)
- free(λx.M) = free(M) $-$ {x}

Free and bound variables

Note:

- a variable can occur free and bound in a λ- term.
  See example above

- $\lambda x.\, \overbrace{\lambda y.\, \underbrace{(\lambda z.xyz)}_{y\ is\ free}\, y}^{y\ is\ bound}$
  "free" is relative to a λ-subterm

Function application as substitution

The result of applying an abstraction (λx.M) to an argument N is formalized by a special form of textual substitution.

$$(\lambda x.M)N \quad \cong \quad [N/x]M$$

Informally: N replaces all free occurrences of x in M.

What can go wrong?

Example: Assume we have constants and arithmetic operation "+" in our lambda calculus

$(\lambda a.\lambda b.a+b)2\ x \quad \cong$
$(\lambda b.2+b)x \quad \cong$
$2+x$

What about:

$(\lambda a.\lambda b.a+b)b\ 3 \quad \cong$
$(\lambda b.b+b)3 \quad \cong$
$3+3 \quad \cong$
6

$\Rightarrow$ From now on, we assume **capture-free** substitution.

Function application

Computation in the lambda calculus is based on the concept or **reduction** (rewriting rules). The goal is to "simplify" an expression until it can no longer be further simplified.

$(\lambda x.M)N$	$\Rightarrow_\beta$	$[N/x]M$	(β–**reduction**)
$(\lambda x.M)$	$\Rightarrow_\alpha$	$\lambda y.[y/x]M$ if $y \notin free(M)$	(α–**reduction**)

Note:

- An equivalence relation can be defined based on $\cong$–convertable λ-terms. "Reduction" rules really work both ways, but we are interested in reducing the complexity of λ-term ($\rightarrow$ direction).
- α–reduction does not reduce the complexity.
- β–reduction: corresponds to application, models computation.

Reduction

- A subterm of the form (λx.M)N is called a ***redex*** (reduction expression).
- A reduction is any sequence of β–reductions and α–reductions.
- A term that cannot be β–reduced is said to be in β–normal form (normal form).
- A subterm that is an abstraction or a variable is said to be in head normal form.

Does a normal form always exist?

Examples:
((λx.(xx))(λx.(xx)))

Programming in lambda calculus

The lambda calculus has very few constructs and it is therefore easy to reason *about it*.

Question: Is the lambda calculus too simple, i.e., can we express all computable functions in the lambda calculus?

Remember: Computation in the lambda calculus is a sequence of applications of reduction rules (mostly β–reductions).

Logical constants and operations (incomplete list):

$$\mathbf{true} \equiv \lambda\text{a}.\lambda\text{b.a} \qquad \textit{select–first}$$
$$\mathbf{false} \equiv \lambda\text{a}.\lambda\text{b.b} \qquad \textit{select–second}$$

$$\mathbf{cond} \equiv \lambda\text{m}.\lambda\text{n}.\lambda\text{p}.((\text{p m})\text{n})$$
$$\mathbf{not} \equiv \lambda\text{x}.((\text{x false})\text{ true})$$
$$\mathbf{and} \equiv \boxed{\text{homework}}$$
$$\mathbf{or} \equiv \lambda\text{x}.\lambda\text{y}.\ ((\text{x true})\text{ y})$$

Programming in lambda calculus

What about data structures?

<u>*data structures*</u>:

pairs can be represented as

$$[M\ .\ N] \equiv \lambda \mathrm{z}.((\mathrm{z}\ \mathrm{M})\ \mathrm{N})$$

$$\begin{aligned} &\textbf{first} \equiv \lambda \mathrm{x}.(\mathrm{x}\ \mathrm{true}) && (\mathit{car}) \\ &\textbf{second} \equiv \lambda \mathrm{x}.(\mathrm{x}\ \mathrm{false}) && (\mathit{cdr}) \\ &\textbf{build} \equiv \lambda \mathrm{x}.\lambda \mathrm{y}.\lambda \mathrm{z}.((\mathrm{z}\ \mathrm{x})\ \mathrm{y}) && (\mathit{cons}) \end{aligned}$$

Programming in lambda calculus

What about arithmetic constants and operations?

There are many options here. Let's look at the system proposed by Church:

$$
\begin{aligned}
0 &\equiv \lambda \mathrm{fx.x} \\
1 &\equiv \lambda \mathrm{fx.(f\ x)} \\
2 &\equiv \lambda \mathrm{fx.(f\ (f\ x))} \\
\ldots & \\
\mathrm{n} &\equiv \lambda fx.(\underbrace{f(f(\ldots(f}_{n\ times}x)\ldots)) \quad \equiv \lambda fx.(f^n x)
\end{aligned}
$$

The natural number **n** is represented as a function that applies a function f n–times to its argument x.

$$
\begin{aligned}
\textbf{succ} &\equiv \lambda \mathrm{m.(\lambda fx.(f\ (m\ f\ x)))} \\
\textbf{add} &\equiv \lambda \mathrm{mn.(\lambda fx.((m\ f)\ (n\ f\ x)))} \\
\textbf{mult} &\equiv \lambda \mathrm{mn.(\lambda fx.((m\ (n\ f))\ x))} \\
\textbf{isZero?} &\equiv \lambda \mathrm{m.((m\ (true\ false))\ true)}
\end{aligned}
$$

Programming in lambda calculus

Examples:

(mult 2 3) $=$

((λmn.(λfx.((m (n f)) x))) 2 3) $=$

$\lambda f_0 x_0.((2\ \boxed{(3\ f_0)})\ x_0)\ =$

$\lambda f_0 x_0.((2\ ((\lambda\text{fx.(f (f (f x))))}\ f_0))\ x_0)\ =$

$\lambda f_0 x_0.((2\ (\lambda\text{x.}(f_0\ (f_0\ (f_0\ \text{x})))))\ x_0)\ =$

$\lambda f_0 x_0.(\boxed{(2\ (\lambda x_1.(f_0^3\ x_1)))}\ x_0)\ =$

$\lambda f_0 x_0.((\lambda\text{x.}((\lambda x_1.(f_0^3\ x_1))\ \boxed{(\ (\lambda x_1.(f_0^3\ x_1))\ \text{x})}\ ))\ x_0)\ =$

$\lambda f_0 x_0.((\lambda\text{x.}\boxed{(\ (\lambda x_1.(f_0^3\ x_1))\ (f_0^3\ \text{x})\ )})\ x_0)\ =$

$\lambda f_0 x_0.\boxed{((\lambda\text{x.}(f_0^3\ (f_0^3\ \text{x})))\ x_0)}\ =$

$\lambda f_0 x_0.(f_0^3\ (f_0^3\ x_0))\ =$

$\lambda\text{fx.}(f^6\ \text{x})\ =\ 6$

Recursion in lambda calculus

Does this make sense?

$$\mathbf{f} \equiv \ldots \mathbf{f} \ldots$$

In lambda calculus, such an equation does not define a term. How to find a λ- term that does "satisfy" the recursive definition?

Example:

add $\equiv$ λmn.
(cond m (**add** (succ m) (pred n)) (isZero? n))

Just to make things easier to read, we will write instead:

add $\equiv$ λmn.
if (isZero? n) then m else (**add** (succ m) (pred n))

This is not a valid definition of a λ– term. What about this one?

add $\equiv$ λ**f**.(λmn.
if (isZero? n) then m else (**f** (succ m) (pred n)))

<u>Claim</u>: The fixed point of the above function is what we are looking for.

Function fixed points

The fixed points of a function g is the set of values $fix_g = \{x | x = g(x)\}$.

Examples:

function **g**	fix_g
λx.6	{6}
λx.(6 - x)	{3}
λx.((x∗x) + (x-4))	{-2, 2}
λx.x	entire domain of f
λx.(x+1)	{ }

Is there a λ–term Y that "computes" a fixed point of a function $F = \lambda f.(\ldots f \ldots)$, i.e., YF = F(YF)?

YES. Y is called the fixed point combinator.

$$\mathrm{Y} \;\equiv\; \lambda \mathrm{f}.((\lambda \mathrm{x}.\mathrm{f}(\mathrm{x}\ \mathrm{x}))\ (\lambda \mathrm{x}.\mathrm{f}(\mathrm{x}\ \mathrm{x})))$$

$$\begin{aligned}
\mathrm{YF} \;&=\; ((\lambda \mathrm{f}.((\lambda \mathrm{x}.\mathrm{f}(\mathrm{x}\ \mathrm{x}))\ (\lambda \mathrm{x}.\mathrm{f}(\mathrm{x}\ \mathrm{x}))))\ \mathrm{F}) \\
&= (\lambda \mathrm{x}.\mathrm{F}(\mathrm{x}\ \mathrm{x}))\ (\lambda \mathrm{x}.\mathrm{F}(\mathrm{x}\ \mathrm{x})) \\
&= \mathrm{F}(\ (\lambda \mathrm{x}.\mathrm{F}(\mathrm{x}\ \mathrm{x}))\ (\lambda \mathrm{x}.\mathrm{F}(\mathrm{x}\ \mathrm{x}))) \\
&= \mathrm{F}(\mathrm{YF})
\end{aligned}$$

The Y–combinator

Example:

$$F \equiv \lambda \mathbf{f}.(\lambda \text{mn.} \\ \quad \text{if (isZero? n) then m else } (\mathbf{f} \text{ (succ m) (pred n)}))$$

$$((\text{YF})\ 3\ 2) =$$

$$(((\lambda \text{f.}((\lambda \text{x.f(x x)})\ (\lambda \text{x.f(x x)}))))\ \text{F})\ \ 3\ 2) =$$

$$(\boxed{(\text{F}((\lambda \text{x.F(x x)})\ (\lambda \text{x.F(x x)})))}\ \ 3\ 2) =$$

$$((\lambda \text{mn.if (isZero? n) then m else} \\ \quad (((\lambda \text{x.F(x x)})\ (\lambda \text{x.F(x x)}))\ \text{(succ m) (pred n)))}\ 3\ 2) =$$

$$\text{if (isZero? 2) then 3 else} \\ \quad (((\lambda \text{x.F(x x)})\ (\lambda \text{x.F(x x)}))\ \text{(succ 3) (pred 2))} =$$

$$(\boxed{((\lambda \text{x.F(x x)})\ (\lambda \text{x.F(x x)}))}\ 4\ 1) =$$

$$((\text{F}((\lambda \text{x.F(x x)})\ (\lambda \text{x.F(x x)})))\ 4\ 1) =$$

$$\text{if (isZero? 1) then 4 else} \\ \quad (((\lambda \text{x.F(x x)})\ (\lambda \text{x.F(x x)}))\ \text{(succ 4) (pred 1))} =$$

$$(\boxed{((\lambda \text{x.F(x x)})\ (\lambda \text{x.F(x x)}))}\ 5\ 0) =$$

$$((\text{F}(\ (\lambda \text{x.F(x x)})\ (\lambda \text{x.F(x x)})))\ 5\ 0) =$$

$$\text{if (isZero? 0) then 5 else} \\ \quad (((\lambda \text{x.F(x x)})\ (\lambda \text{x.F(x x)}))\ \text{(succ 5) (pred 0))} = \mathbf{5}$$

The Y–combinator example (cont.)

Note:

- Informally, the Y–combinator allows us to get as many copies of the recursive procedure body as we need. The computation "unrolls" recursive procedure calls one at a time.
- This notion of recursion is purely syntactic.

Lambda calculus — final remarks

- We can express all computable functions in our λ–calculus. However, nobody "programs" in lambda calculus. For that we have more "convenient" functional languages.
- All computable functions can be express by the following two combinators, referred to as **S** and **K**:
  - $\mathrm{K} \equiv \lambda\mathrm{xy.x}$
  - $\mathrm{S} \equiv \lambda\mathrm{xyz.xz(yz)}$

  Combinatory logic is as powerful as Turing Machines.