REMINDERS

• Project 1 grades have been posted. We are looking into “0” points cases.

• Midterm has been graded. Grades will be entered today and tomorrow.

• Project 2 has been posted. Due: November 30 (Monday after Thanksgiving)

• Homework 7 will be posted later today, or tomorrow morning. Due: December 1

• Last class: Tuesday, December 8 (December 10 last day of classes)
Scheme Project

A spell checker generator using Bloom filters in Scheme.
Review - Lambda calculus

$\lambda$-terms ($\textit{wffs}$) are inductively defined.
A $\lambda$-terms is:
- a variable $x$
- $(\lambda x . M)$ where $x$ is a variable and $M$ is $\lambda$-term
- $(M \ N)$ where $M$ and $N$ are $\lambda$-terms

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

- “multiple” arguments
  $\lambda x y z$ is $(\lambda x . (\lambda y . z))$
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.

\[
\begin{align*}
((\lambda x.M)N) & \Rightarrow_\beta [N/x]M \quad (\beta\text{-reduction}) \\
(\lambda x.M) & \Rightarrow_\alpha \lambda y.[y/x]M \quad (\alpha\text{-reduction}) \quad \text{if } y \notin \text{free}(M)
\end{align*}
\]

Note:

- An equivalence relation can be defined based on \( \cong \)-convertable \( \lambda \)-terms. “Reduction” rules really work both ways, but we are interested in reducing the complexity of \( \lambda \)-term (\( \rightarrow \) direction).

- \( \alpha \)-reduction does not reduce the complexity.

- \( \beta \)-reduction: corresponds to application, models computation.
• A subterm of the form $(\lambda x. M)N$ is called a **redex** (reduction expression).

• A reduction is any sequence of $\beta$–reductions and $\alpha$–reductions.

• A term that cannot be $\beta$–reduced is said to be in $\beta$–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:

$$((\lambda x. (xx)) (\lambda x. (xx)))$$
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 \( \beta \)-reductions).

Logical constants and operations (incomplete list):

\[
\begin{align*}
\text{true} & \equiv \lambda a.\lambda b.a \\
\text{false} & \equiv \lambda a.\lambda b.b \\
\text{cond} & \equiv \lambda m.\lambda n.\lambda p.((p\ m)\ n) \\
\text{not} & \equiv \lambda x.((x\ \text{false})\ \text{true}) \\
\text{and} & \equiv \underline{\text{homework}} \\
\text{or} & \equiv \lambda x.\lambda y.\ ((x\ \text{true})\ y)
\end{align*}
\]
Programming in lambda calculus

What about data structures?

data structures:
pairs can be represented as

\[ [M \cdot N] \equiv \lambda z.((z M) N) \]

first \( \equiv \lambda x.(x \text{ true}) \) \hspace{2cm} (car)
second \( \equiv \lambda x.(x \text{ false}) \) \hspace{2cm} (cdr)
build \( \equiv \lambda x.\lambda y.\lambda z.((z x) y) \) \hspace{2cm} (cons)
Programming in lambda calculus

What about arithmetic constants and operations?

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

\[ 0 \equiv \lambda f x. x \]
\[ 1 \equiv \lambda f x. (f \ x) \]
\[ 2 \equiv \lambda f x. (f \ (f \ x)) \]
\[ \ldots \]
\[ n \equiv \lambda f x. (f (\underbrace{f \ldots (f}_{n \ times} x \ldots})) \equiv \lambda f x. (f^n x) \]

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

\[
\begin{align*}
\text{succ} & \equiv \lambda m. (\lambda f x. (f (m \ f \ x))) \\
\text{add} & \equiv \lambda m n. (\lambda f x. ((m \ f) (n \ f \ x))) \\
\text{mult} & \equiv \lambda m n. (\lambda f x. ((m \ (n \ f)) \ x)) \\
\text{isZero}？ & \equiv \lambda m. ((m \ (\text{true} \ \text{false})) \ \text{true})
\end{align*}
\]
Programming in lambda calculus

Examples:

\[ (\texttt{mult} \ 2 \ 3) = \]
\[ (\lambda mn. (\lambda f x. ((m (n f)) x))) \ 2 \ 3) = \]
\[ \lambda f_0 x_0. ((2 [3 \ f_0]) \ x_0) = \]
\[ \lambda f_0 x_0. ((2 ((\lambda f x. (f (f (f x)))) \ f_0)) \ x_0) = \]
\[ \lambda f_0 x_0. ((2 ((\lambda x. (f_0 \ (f_0 \ (f_0 \ x)))))) \ x_0) = \]
\[ \lambda f_0 x_0. (((2 (\lambda x_1. (f_0^3 \ x_1))) \ x_0) = \]
\[ \lambda f_0 x_0. ((\lambda x. ((\lambda x_1. (f_0^3 \ x_1)) \ ((\lambda x_1. (f_0^3 \ x_1))) \ x_0)) \ x_0) = \]
\[ \lambda f_0 x_0. ((\lambda x. ((\lambda x_1. (f_0^3 \ x_1)) (f_0^3 \ x))) \ x_0) = \]
\[ \lambda f_0 x_0. (((\lambda x. (f_0^3 \ (f_0^3 \ x))) \ x_0) = \]
\[ \lambda f_0 x_0. (f_0^3 \ (f_0^3 \ x_0)) = \]
\[ \lambda f x. (f^6 \ x) = 6 \]
Recursion in lambda calculus

Does this make sense?

\[ f \equiv \ldots f \ldots \]

In lambda calculus, such an equation does not define a term. How to find a \( \lambda \)-term that does “satisfy” the recursive definition?

Example:
\[
\text{add} \equiv \lambda mn. \\
(\text{cond } m (\text{add} (\text{succ } m) (\text{pred } n)) (\text{isZero? } n))
\]

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

\[
\text{add} \equiv \lambda mn. \\
\text{if } (\text{isZero? } n) \text{ then } m \text{ else } (\text{add} (\text{succ } m) (\text{pred } n))
\]

This is not a valid definition of a \( \lambda \)-term. What about this one?

\[
\text{add} \equiv \lambda f.(\lambda mn. \\
\text{if } (\text{isZero? } n) \text{ then } m \text{ else } (f (\text{succ } m) (\text{pred } n)))
\]

**Claim:** 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:

<table>
<thead>
<tr>
<th>function $g$</th>
<th>$fix_g$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda x.6$</td>
<td>${ 6 }$</td>
</tr>
<tr>
<td>$\lambda x.(6 - x)$</td>
<td>${ 3 }$</td>
</tr>
<tr>
<td>$\lambda x.((x*x) + (x-4))$</td>
<td>${-2, 2}$</td>
</tr>
<tr>
<td>$\lambda x.x$</td>
<td>entire domain of $f$</td>
</tr>
<tr>
<td>$\lambda x.(x+1)$</td>
<td>${ }$</td>
</tr>
</tbody>
</table>

Is there a $\lambda$–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.

$$Y \equiv (\lambda f.((\lambda x.f(x x)) (\lambda x.f(x x))))$$

$$(YF) = ((\lambda f.((\lambda x.f(x x)) (\lambda x.f(x x)))) F)$$
$$= ((\lambda x.F(x x)) (\lambda x.F(x x)))$$
$$= (F( (\lambda x.F(x x)) (\lambda x.F(x x))))$$
$$= (F(YF))$$
The Y–combinator

Example:

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

\[
((\text{Y}F) \ 3 \ 2) = \\
(((\lambda f. ((\lambda x.f(x\ x)) (\lambda x.f(x\ x)))) \ F) \ 3 \ 2) = \\
((\lambda mn. \text{if (isZero? n) then m else } ((\lambda x.F(x\ x)) (\lambda x.F(x\ x))) (\text{succ m}) (\text{pred n}))) \ 3 \ 2) = \\
\text{if (isZero? 2) then 3 else} \\
(((\lambda x.F(x\ x)) (\lambda x.F(x\ x))) (\text{succ 3}) (\text{pred 2})) = \\
((\lambda x.F(x\ x)) (\lambda x.F(x\ x))) 4 \ 1) = \\
((\lambda x.F(x\ x)) (\lambda x.F(x\ x))) 4 \ 1) = \\
\text{if (isZero? 1) then 4 else} \\
(((\lambda x.F(x\ x)) (\lambda x.F(x\ x))) (\text{succ 4}) (\text{pred 1})) = \\
((\lambda x.F(x\ x)) (\lambda x.F(x\ x))) 5 \ 0) = \\
((\lambda x.F(x\ x)) (\lambda x.F(x\ x))) 5 \ 0) = \\
\text{if (isZero? 0) then 5 else} \\
(((\lambda x.F(x\ x)) (\lambda x.F(x\ x))) (\text{succ 5}) (\text{pred 0})) = 5
\]
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 $\lambda$-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 $K$:
  
  - $K \equiv \lambda xy.x$

  - $S \equiv \lambda yz.xz(yz)$

  Combinatory logic is as powerful as Turing Machines.
Programming with Concurrency

Why do we care about concurrency?

- Today, concurrency is nearly everywhere (peta-flops supercomputers to high-end smart phones).
- Necessary to keep “Moore’s Law” alive due to power/heat dissipation limits.
- Some form of parallel programming will be required, i.e., automatic tools have not been able to hide all aspects of concurrency.

⇒

Need to understand the basics of parallel programming
Next Lecture

- Dependence analysis
- More on automatic vectorization / parallelization
- Work on Project 2!