Class Information

INFORMATION and REMINDERS

• Homework 6 is due on Sunday.

• Project 2 has been posted; due Friday, April 21.

• Midterm

  1. Grader names have been posted.

  2. April 21 is deadline to challenge grade.
Scheme Project

A Bloom filter based spell checker generator.
Review: Lambda calculus

\textbf{\lambda-terms} \textit{(wffs)} are inductively defined.

A \lambda-terms is:
\begin{itemize}
  \item a variable \( x \)
  \item \((\lambda x. M)\) where \( x \) is a variable and \( M \) is \lambda-term \textit{(abstraction)}
  \item \((M N)\) where \( M \) and \( N \) are \lambda-terms \textit{(application)}
\end{itemize}

\textbf{Abbreviations} \textit{(Notational conveniences)}:

\begin{itemize}
  \item function application is left associative
        \((f g z)\) is \(((f g) z)\)
  \item function application has precedence over function abstraction — “function body” extends as far to the right as possible
        \(\lambda x. yz\) is \((\lambda x. (yz))\)
  \item “multiple” arguments
        \(\lambda x y. z\) is \((\lambda x. (\lambda y. z))\)
\end{itemize}
Review: Reduction

- 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**).

Does a normal form always exist?

Examples:
\(((\lambda x. (x x))(\lambda x. (x x)))\)

\((\lambda x.xxx)(\lambda x.xxx)\)
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):

- **true** $\equiv \lambda a.\lambda b.a$  
  \textit{select–first}

- **false** $\equiv \lambda a.\lambda b.b$  
  \textit{select–second}

- **cond** $\equiv \lambda m.\lambda n.\lambda p.((p m)n)$

- **not** $\equiv \lambda x.((x \text{ false}) \text{ true})$

- **and** $\equiv \underline{\text{homework}}$

- **or** $\equiv \lambda x.\lambda y. ((x \text{ true}) y)$
What about data structures?

_data structures:_

_pairs_ can be represented as

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

first \equiv \lambda x. (x \ true) \quad (\text{car})

second \equiv \lambda x. (x \ false) \quad (\text{cdr})

build \equiv \lambda x. \lambda y. \lambda z. ((z \ x) \ y) \quad (\text{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:

\[ 0 \equiv \lambda fx. x \]
\[ 1 \equiv \lambda fx. (f \ x) \]
\[ 2 \equiv \lambda fx. (f \ (f \ x)) \]
\[ \ldots \]
\[ n \equiv \lambda fx. (f(\ldots(fx)\ldots)) \equiv \lambda fx. (f^n x) \]

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

\[ \text{succ} \equiv \lambda m. (\lambda fx. (f \ (m \ f \ x))) \]
\[ \text{add} \equiv \lambda mn. (\lambda fx. ((m \ f) \ (n \ f \ x))) \]
\[ \text{mult} \equiv \lambda mn. (\lambda fx. ((m \ (n \ f)) \ x)) \]
\[ \text{isZero?} \equiv \lambda m. ((m \ (\text{true} \ \text{false})) \ \text{true}) \]
Examples:

\((\text{mult } 2 \ 3) =\)

\(((\lambda mn.(\lambda fx.((m \ (n \ f)) \ x))) \ 2 \ 3) =\)

\(\lambda f_0 x_0.((2 \ (3 \ f_0)) \ x_0) =\)

\(\lambda f_0 x_0.((2 \ ((\lambda f x.(f \ (f \ (f \ x)))) \ f_0)) \ x_0) =\)

\(\lambda f_0 x_0.((2 \ (\lambda x.((f_0 \ (f_0 \ x)))))) \ x_0) =\)

\(\lambda f_0 x_0.((2 \ (\lambda x_1.((f_0^3 \ x_1)))) \ x_0) =\)

\(\lambda f_0 x_0.((\lambda x.((\lambda x_1.((f_0^3 \ x_1))) \ ((\lambda x_1.((f_0^3 \ x_1))) \ x)))) \ x_0) =\)

\(\lambda f_0 x_0.((\lambda x.((\lambda x_1.((f_0^3 \ x_1))) \ (f_0^3 \ x)))) \ x_0) =\)

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

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

\(\lambda fx.(f^6 x) = 6\)
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:

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

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

\[ K \equiv \lambda xy.x \]
\[ S \equiv \lambda xyz.xz(yz) \]

Combinatory logic is as powerful as Turing Machines.
Next Lecture

Things to do:

• Programming with concurrency
• Dependence notion
• Dependence analysis
• OpenMP
• Automatic vectorization / parallelization