Announcements

- Reminder: **First project** deadline: March 4.
- Homework 1 sample solution is now available
- Homework 3 has been posted; deadline: March 2
- **Midterm: March 9**, in class, 80 minutes, closed book / notes;
- Spring break: March 12 - March 20.
Lambda calculus

- formalism for studying ways in which functions can be formed, combined, and used for computation

- computation is defined as rewriting rules (operational semantics)

- the syntactic notion of computation was developed first; a mathematical semantics followed much later

Examples:

\[ f(x) = x+2 \quad \lambda x.x+2 \quad \text{different notation} \]
\[ (\lambda x.x+2) \ 1 \quad 1+2 = 3 \quad \text{function application and substitution} \]
\[ (\lambda x.x) \ (\lambda y.y) \quad \text{arguments and returned “values” can be functions} \]
\[ \lambda x.xx \quad \text{untyped lambda calculus} \]
\[ f(x) = x(x) \]
Lambda calculus and functional programming

Lambda calculus is the theoretical foundation of pure functional programming (no side effects, *referential transparency*)

Functional programming: functions are *first class citizens*

- can be a return value
- can be passed as arguments
- can be put into a data structure
- value of an expression can be a function

\[(\lambda x.x) \ (\lambda x.1) \ (\lambda y.y)\]
Currying functions

\( f : D^n \rightarrow D \) can be transformed to

\( f : D \rightarrow (D \rightarrow \ldots (D \rightarrow D)) \ldots ) \)

Examples:

\[
\begin{align*}
f(x, y) &= x + y & f : \mathcal{N} \times \mathcal{N} \rightarrow \mathcal{N} \\
f'(x) &= g_x(y) & f' : \mathcal{N} \rightarrow (\mathcal{N} \rightarrow \mathcal{N}) \\
g_x : \mathcal{N} \rightarrow \mathcal{N} & g_2 : \mathcal{N} \rightarrow \mathcal{N} \\
g_x(y) &= x + y & g_2(y) = 2 + y
\end{align*}
\]

\( g_x(y) \) “freezes” the first argument value \( x \).
This is sometimes referred to as \textbf{partial evaluation}.

Here is an example:
\[
\begin{align*}
(\lambda x. \lambda y. x+y) \ 2 \ 3 &= \\
(\lambda y. 2+y) \ 3 &= \\
2+3 &= 5
\end{align*}
\]

To simplify discussion, all \( n \)-ary functions are curried, i.e., are represented by a sequence of one-place functions.
Lambda calculus

\(\lambda\)-terms (wffs) are inductively defined. A \(\lambda\)-terms is:

- a variable \(x\)
- \((\lambda x.M)\) where \(x\) is a variable and \(M\) is \(\lambda\)-term (abstraction)
- \((M \, N)\) where \(M\) and \(N\) are \(\lambda\)-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
  \(\lambda x.\, yz\) is \((\lambda x.\, (yz))\)
- “multiple” arguments
  \(\lambda xy.z\) is \((\lambda x.\, (\lambda y.z))\)
Examples

What are the λ-terms represented by

\[ \lambda xyz. x(yz)t \]

\[ (\lambda x. (\lambda y. yx) x) \]
Examples

What are the $\lambda$-terms represented by

$\lambda xyz.x(yz)t$ is

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

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

$(\lambda x. ((\lambda y. (yx)x))x)$
Free and bound variables

Abstraction \((\lambda x. M)\) "binds" variable \(x\) in "body" \(M\). You can think of this as a declaration of variable \(x\) with scope \(M\).

Let \(M, N\) be \(\lambda\)-terms and \(x\) is a variable. The set of free variables of \(M\), \(\text{free}(M)\), is defined inductively as follows:

- \(\text{free}(x) = \{x\}\)
- \(\text{free}(M \ N) = \text{free}(M) \cup \text{free}(N)\)
- \(\text{free}(\lambda x. M) = \text{free}(M) - \{x\}\)
Free and bound variables

Note:

- a variable can occur free and bound in a $\lambda$- term.
  See example above

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

"free" is relative to a $\lambda$-subterm

- formal definition of bound: exercise
Function application as substitution

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

\[(\lambda x. M)N \equiv [N/x]M\]

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

\[(\lambda a. \lambda b. a+b)2 \ x \equiv \ (\lambda b. 2+b)x \equiv 2+x\]

What about:

\[(\lambda a. \lambda b. a+b)b \ 3 \ \equiv \ (\lambda b. b+b)3 \ \equiv 3+3 \ \equiv 6\]
Function application as substitution

We need **capture free** substitution

(1) If the free variables of N have **no bound** occurrence in M, then \([N/x]M\) is formed by replacing all free occurrences of x in M by N.

Example: \((\lambda b.2+b)\ x \blacktriangleright [x/b]2+b \blacktriangleright 2+x\)

(2) Otherwise, if variable y is free in N and bound in M, replace the binding and bound occurrences of y in M by new (fresh) variable z. Repeat until case (1) applies.

\[(\lambda a.\lambda b.a+b)\ b \blacktriangleright \lambda a.\lambda z.a+z)\ b \blacktriangleright \lambda z.b+z\]
Function application as substitution

Examples:

\[
\begin{align*}
[u/x] x & \equiv u & \text{u not bound in } x \\
[u/x]\lambda x.xu & \equiv \lambda x.xu & \text{x not free in } \lambda x.xu \\
[u/x]\lambda u.x & \equiv [u/x]\lambda z.x & \text{u is bound in } \lambda u.x \\
[u/x]\lambda u.u & \equiv [u/x]\lambda z.z & \lambda z.z
\end{align*}
\]
Capture–free substitution

Let $x, y, z$ denote variables and $M, N, P$ arbitrary $\lambda$-terms.

$\left[ N/y \right] x = x$ if $y \neq x$
$\left[ N/y \right] x = N$ if $y = x$

$\left[ N/x \right] (\lambda x. M) = (\lambda x. M)$
$\left[ N/x \right] (\lambda y. M) = \lambda y.\left[ N/x \right] M$ where $x \neq y$,
$y \notin free(N)$,

$\left[ N/x \right] (\lambda y. M) = \lambda z.\left[ N/x \right] [z/y] M$ where $x \neq y$,
$y \in free(N)$,
$z \notin free(M)$
$z \notin free(N)$

$\left[ N/x \right](M P) = ([N/x]M [N/x]P)$
Substitution — Examples

\[ [y/x] ((\lambda z.zx)(\lambda x.x)) = \]

\[ [\lambda x.xy/x] ((\lambda y.xy)z) = \]
Substitution — Examples

\[ [y/x] ((\lambda z. zx)(\lambda x.x)) = \]
\[ (((y/x) \lambda z. zx)(y/x) \lambda x.x)) = \]
\[ (\lambda z.zy) (\lambda x.x) \]

\[ [\lambda x.xy/x] ((\lambda y.xy)z) = \]
\[ (((\lambda x.xy/x) \lambda y.xy) [\lambda x.xy/x] z) = \]
\[ ((\lambda z.[\lambda x.xy/x][z/y] xy) z) = \]
\[ ((\lambda z.[\lambda x.xy/x] xz) z) = \]
\[ ((\lambda z.(\lambda x.xy) z) z) \]
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.

\[(\lambda x.M)N \Rightarrow_\beta [N/x]M \quad (\beta\text{–reduction})\]
\[(\lambda x.M) \Rightarrow_\alpha \lambda y.[y/x]M \quad (\alpha\text{–reduction})\]
\[\text{if } y \notin free(M)\]
\[(\lambda x.M)x \Rightarrow_\eta M \quad (\eta\text{–reduction})\]
\[\text{if } x \notin free(M)\]

Note:

- An equivalence relation can be defined based on \(\cong\text{–convertable } \lambda\text{-terms. “Reduction” rules really work both ways, but we are interested in reducing the complexity of } \lambda\text{-term (\(\rightarrow\) direction).}\)
- \(\alpha\text{–reduction} \) does not reduce the complexity.
- \(\beta\text{–reduction}: \) corresponds to application, models computation.
- \(\eta\text{–reduction}: \) one can remove (or introduce) levels of indirections in function applications.
Reduction — example

$$(\lambda xyz. (xz)(yz))\ (\lambda xy. x)\ (\lambda xy. x)$$

$$(\lambda yz. ((\lambda xy.x) z)(yz))\ (\lambda xy. x)$$

$$(\lambda yz. (\lambda y.z)(yz))\ (\lambda xy. x)$$

$$\lambda z. ((\lambda xy.x) z)\ ((\lambda xy.x) z)$$

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

$$(\lambda yz.z)(\lambda xy. x)\quad\quad\lambda z.((\lambda xy.x) z)(\lambda y.z)$$

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

$$\lambda z. z$$
Reduction

- A subterm of the form \(((\lambda x.M)N)\) is called a \textit{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:
\((\lambda x.xx)(\lambda x.xx)\)

\((\lambda x.xxx)(\lambda x.xxx)\)
Church–Rosser Property

Informal: If an expression reduces to two normal forms, they must be identical modulo $\alpha$–conversion.

Theorem (CR–I)

Let $M$, $P$, $Q$, and $R$ be $\lambda$-terms.
If $M \Rightarrow^* P$ and $M \Rightarrow^* Q$, then
$\exists R$ such that $P \Rightarrow^* R$ and $Q \Rightarrow^* R$

Corollary: If a normal form exists for $M$, it must be unique. Proof is homework!
Reduction strategies

If P is not in normal form, what redex to choose?

Does it make a difference (remember CR–I)?

Yes, since there are infinite computations possible

\[(\lambda x.y)((\lambda x.xx)(\lambda x.xx))\]

Note:

- It is undecidable whether a \(\lambda\)-term has a normal form, i.e., whether it will reduce to a normal form.

- You can think of the normal form as the “value” to which a \(\lambda\)-term evaluates

- Two \(\lambda\)-terms are equal if they have the same normal form.
Reduction strategies & CR–II

• If A and B are two redexes in a \( \lambda \)-term M and the first occurrence of \( \lambda \) in A is to the left of the first occurrence of \( \lambda \) in B, then A is to the left of B.

• If A is a redex in M, and it is to the left of all other redexes, then A is the leftmost redex of M.

Theorem (CR–II)
Let X and Y be \( \lambda \)-terms.
If \( X \Rightarrow^* Y \) and Y is in normal form, then \( X \Rightarrow^* Y \) using only leftmost redexes.
Choosing always the left-most redex is called **normal order**.

**normal order** specifies the redex that is chosen next for a given \( \lambda \)-term. **c-b-n** and **c-b-v** specify what to do with a redex of the form \((\lambda x. M) N\).

- **c-b-n**: use \( \beta \)-reduction (don’t “touch” \( N \)).
- **c-b-v**: need a notion of value. Only after “\( N \)” is reduced to a value, the \( \beta \)-reduction is performed. What’s a value? Let’s pick head-normal form.

Question: Reduce the following \( \lambda \)-term using c-b-n and c-b-v

\[(\lambda z. y)(\lambda y. (\lambda x. xx)(\lambda x. xx))\]

Scott refers to **c-b-v** as **applicative-order evaluation**, **c-b-n** is called **normal-order evaluation** of a redex. **lazy evaluation** is a specific combination of both approaches, where arguments are evaluated at most once (memoization).
call–by–name vs. call–by–value

left-most redex, call–by–name:

- guaranteed to reach a normal form if it exists.
- can result in some parameter being evaluated several times or never.

left-most redex, call–by–value:

- efficient — assuming parameter is used at least once.
- can lead to nonterminating computation.

Functional languages typically use call–by–value because of its efficiency.
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:

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

What do we need to represent lists?

• empty list

• isEmpty? function
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

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

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

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

Note: \( \text{pred} \) function can also be encoded, but the encoding is rather complicated.
Programming in lambda calculus

Examples:

\[(\text{mult} \ 2 \ 3) = (\text{(\lambda} m n. (\text{m} (\text{n} \ f)) \ x)) \ 2 \ 3) = \]

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

\[\lambda f_0 x_0.((2 \ ((\text{\lambda} f x. (f \ (f \ f))) \ f_0)) \ x_0) = \]

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

\[\lambda f_0 x_0.((2 \ (\text{\lambda} x_1. (f_0^3 \ x_1))) \ x_0) = \]

\[\lambda f_0 x_0.((\text{\lambda} x. ((\text{\lambda} x_1. (f_0^3 \ x_1)) \ ((\text{\lambda} x_1. (f_0^3 \ x_1)) \ x))) \ x_0) = \]

\[\lambda f_0 x_0.((\text{\lambda} x. ((\text{\lambda} x_1. (f_0^3 \ x_1)) \ (f_0^3 \ x))) \ x_0) = \]

\[\lambda f_0 x_0.(((\text{\lambda} x. (f_0^3 \ (f_0^3 \ x))) \ x_0) = \]

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

\[\lambda f x. (f^6 \ x) = 6 \]

\[(\text{isZero?} \ 0) = ((\lambda f x. x) (\lambda y. \text{false})) \text{ true}) = \]

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

\[(\text{isZero?} \ n) \quad \text{where} \ n > 0 ? \]
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:

\[ \text{add} \equiv \lambda \text{mn}. \]

\[ (\text{cond m (add (succ m) (pred n)) (isZero? n)}) \]

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

\[ \text{add} \equiv \lambda \text{mn}. \]

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

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

\[ \text{add} \equiv \lambda f. (\lambda \text{mn}. \]

\[ \text{if (isZero? n) then m else (f (succ m) (pred n)))} \]

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:

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

\[
((YF) 3 2) = \\
(((\lambda f. ((\lambda x. f(x x)) (\lambda x. f(x x)))) F) 3 2) = \\
((\lambda m n. \begin{cases} 
  m & \text{if (isZero? } n) \text{ then } m \text{ else } \\
  ((\lambda x. f(x x)) (\lambda x. f(x x))) (\text{succ } m) (\text{pred } n) & \text{else}
\end{cases}) 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) = \\
((F((\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) = \\
((F((\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.

• Question: Would our Y–combinator work with call–by–value application order?
Call–by–value lambda calculus interpreter

Procedure $eval(P)$:

\[
\text{case } P:\ \\
\text{“x”: } x \\
\text{“(λ x.M)”: } λ x.M \\
\text{“(M N)”: \quad e_1 = eval(M);} \\
\text{\quad error if not } e_1 \equiv λx.M_1; \\
\text{\quad e_2 = eval(N);} \\
\text{\quad e_3 = [e_2/x]M_1; \quad //capture–free subst.} \\
\text{\quad eval(e_3)} \\
\text{endcase}
\]

Note:

- Values are variables or $λ$–abstractions
- You could think of this interpreter as an operational semantics for our call–by–value application order lambda calculus (defining interpreter)
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 \( K \):
  
  \[
  - K \equiv \lambda xy.x \\
  - S \equiv \lambda xyz.xz(yz)
  \]

  Combinatory logic is as powerful as Turing Machines.
Next Lecture

• **MIDTERM EXAM**, 80 minutes, in class

• Introduction to functional languages (Scheme) ; Scott: Chapter 10

• Second project