Class Announcements

- Fifth homework is due Tuesday, April 4
- Second project has been posted and is due Monday, April 17
- Midterms
  - Second midterm will be on April 11.
    Final (third midterm) exam on Thursday, May 4, noon - 3:00pm timeslot. Location: most likely this room.
    Any CONFLICTS with other classes?
Lexical Scoping and let, let*, and letrec

All are variable binding operations:

LET = let, let*, letrec

\[
\text{LET } ((v_1 \ e_1) \\
(v_2 \ e_2) \\
\ldots \\
(v_n \ e_n)) \\
e
\]

- **let**: binds variables to values (no specific order), and evaluates body \(e\) using the bindings; new bindings are not effective during evaluation of any \(e_i\).

- **let\***: binds variables to values in textual order of write-up (left to right, or here: top down); new binding is effective for next \(e_i\) (nested scopes).

- **letrec**: bindings of variables to values in no specific order; independent **evaluations of all** \(e_i\) **to values** have to be possible; new bindings effective for all \(e_i\); mainly used for recursive function definitions.
let and let* examples

(let ((a 5)
       (b 6))
  (+ a b))  ;; ==> 11

(let ((a 5)
       (b (+ a 6)))
  (+ a b))  ;; ==> ERROR: unbound variable: a

(let* ((a 5)
        (b (+ a 6)))
  (+ a b))  ;; ==> 16

Note: let and let* do not add anything to the expressiveness of the language, i.e., they are only a convenient shorthand. For instance,

(let ((x v1) (y v2)) e) can be rewritten as

((lambda (x y) e) v1 v2)
letrec examples

Typically used for local definitions of recursive functions

(letrec ((a 5)
           (b (+ a 6)))
    (+ a b)) ;; ==> ERROR: unbound variable: a

(letrec ((a 5)
           (b (lambda () (+ a 6))))
    (+ a (b))) ;; ==> 16

(letrec ((b (lambda () (+ a 6)))
           (a 5))
    (+ a (b))) ;; ==> 16

(letrec ((even? (lambda (x)
                   (or (= x 0)
                       (odd? (- x 1)))))
           (odd? (lambda (x)
                       (and (not (= x 0))
                            (even? (- x 1))))))
    (list (even? 3) (even? 20) (odd? 21)))
   ;; ==> (#f #t #t)
**Lambda calculus**

\textbf{λ-terms} (wffs) are inductively defined. A λ-terms is:

- a variable \( x \)
- \( (\lambda x. M) \) where \( x \) is a variable and \( M \) is λ-term

\textit{(abstraction)}

- \( (M N) \) where \( M \) and \( N \) are λ-terms

\textit{(application)}

**Abbreviations** (Notational conveniences):

- function application is left associative
  \[(f g z) \text{ is } ((f g) z)\]

- function application has precedence over function abstraction — “function body” extends as far to the right as possible
  \[\lambda x. yz \text{ is } (\lambda x. (yz))\]

- “multiple” arguments
  \[\lambda xy. z \text{ is } (\lambda x. (\lambda y. z))\]

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 \(M\), free(\(M\)), is defined inductively as follows:

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

Note:

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

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

“free” is relative to a λ-subterm
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.

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

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

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

What about:

\[
((\lambda a. \lambda b. a+b)b) \ 3) \ \cong \\
((\lambda b. b+b)3) \ \cong \\
3+3 \ \cong \\
6
\]

⇒ 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.

\[(\lambda x. M) N \Rightarrow_\beta [N/x]M \quad (\beta\text{-reduction})\]
\[\lambda x. M \Rightarrow_\alpha \lambda y. [y/x] M \quad (\alpha\text{-reduction})\]
\[\text{if } y \not\in \text{free}(M)\]

Note:

- An equivalence relation can be defined based on \(\cong\)-convertible \(\lambda\)-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.
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 (\textit{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):

\[
\begin{align*}
\text{true} & \equiv \lambda a. \lambda b. a \\
\text{false} & \equiv \lambda a. \lambda b. b \\
\text{cond} & \equiv \lambda m. \lambda n. \lambda p. ((p m) n) \\
\text{not} & \equiv \lambda x. ((x \text{ false}) \text{ true}) \\
\text{equiv} & \equiv \lambda x. \lambda y. ((x y) \text{ false}) \\
\text{or} & \equiv \lambda x. \lambda y. ((x \text{ true}) y)
\end{align*}
\]
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{1cm} (\text{car})
second \( \equiv \lambda x.(x \text{ false}) \) \hspace{1cm} (\text{cdr})
build \( \equiv \lambda x.\lambda y.\lambda z.((z x) y) \) \hspace{1cm} (\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:

\[
\begin{align*}
0 & \equiv \lambda fx.x \\
1 & \equiv \lambda fx.(f x) \\
2 & \equiv \lambda fx.(f (f x)) \\
\ldots \\
n & \equiv \lambda fx.\underbrace{(f(f(\ldots(f x)\ldots)))}_{n \text{ times}} \equiv \lambda fx.(f^n x)
\end{align*}
\]

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

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

\[(\text{mult} \ 2 \ 3) = \]
\[((\lambda mn.(\lambda fx.((m (n f)) x))) \ 2 \ 3) = \]
\[\lambda f_0 x_0.((2 \ ((3 f_0)) x_0) = \]
\[\lambda f_0 x_0.((2 \ ((\lambda f_0 ((f (f (f 0)))) f_0)) x_0) = \]
\[\lambda f_0 x_0.((2 \ ((\lambda x_0 ((f_0 (f_0 (f_0 x))) x_0))) x_0) = \]
\[\lambda f_0 x_0.(((2 \ ((\lambda x_1 (f_0^3 x_1))) x_0) = \]
\[\lambda f_0 x_0.((\lambda x_0 (((\lambda x_1 (f_0^3 x_1)) (f_0^3 x_1))) x_0) = \]
\[\lambda f_0 x_0.(((\lambda x (f_0^3 (f_0^3 x))) x_0) = \]
\[\lambda f_0 x_0.(f_0^3 (f_0^3 x_0)) = \]
\[\lambda f_0 x_0.((f_0^3 x) = 6 \]

\[\lambda f_0 x_0.((f_0^6 x) = 6 \]

CS 314

Spring’23 lecture 19, page 14
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>$\lambda x.((x\ast x) + (x-4))$</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:

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

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

Things to do:

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