Recursion in the Lambda Calculus

Monday, October 23, 2023
Recursion in the Lambda Calculus

Monday, October 23, 2023
Logistics

• HW 4 in
  • due: Sun 10/22 11:59 pm EST

• HW 5 out
  • due: Sun 10/29 11:59 pm EST
### From Lecture 1

- "high" level (easier for humans to understand)
- "Computation" = "arithmetic" of expressions
- "declarative"
- Core model: Lambda Calculus

- "low" level (runs on CPU)
- "Computation" = sequence of instructions / statements
- "imperative"
- Core model: Turing Machines

<table>
<thead>
<tr>
<th>English</th>
<th>Specification langs</th>
<th>Markup (html, markdown)</th>
<th>Database (SQL)</th>
<th>Logic Program (Prolog)</th>
<th>Lazy lang (Haskell, R)</th>
<th>Functional lang (Racket)</th>
<th>JavaScript, Python</th>
<th>C# / Java</th>
<th>C++</th>
<th>C</th>
<th>Assembly Language</th>
<th>Machine code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Types? pre/post cond?</td>
<td>tags</td>
<td>queries</td>
<td>relations</td>
<td>Delayed computation</td>
<td>Expressions (no stmts)</td>
<td>“eval”</td>
<td></td>
<td>GC (no alloc, ptrs)</td>
<td></td>
<td></td>
<td>Classes, objects</td>
<td>Scoped vars, fns</td>
</tr>
</tbody>
</table>

**NOTE:** This hierarchy is **approximate**

This class: how to program in a high-level more “human friendly” way

“Nicer” for humans to use
The Lambda ($\lambda$) Calculus

- A “programming language” consisting of only:
  - Lambda functions
  - Function application

- Equivalent in “computational power” to:
  - Turing Machines
  - Your favorite programming language!
Church Numerals

;; A ChurchNum is a function with two arguments:
;; “f” : a function to apply
;; “base” : a base ("zero") value to apply to
;;
;; For a specific number, its "Church" representation
;; applies the given function that number of times

(define czero
  (lambda (f base) base)) ; f applied zero times

(define cone
  (lambda (f base) (f base)))) ; f applied one time

(define ctwo
  (lambda (f base) (f (f base))))) ; f applied two times

(define cthree
  (lambda (f base) (f (f (f base))))))) ; f applied three times
Church “Add1”

;;; cplus1 : ChurchNum -> ChurchNum
;;; “Adds” 1 to the given Church num

(define cplus1
  (lambda (n)
    (lambda (f base)
      (f (n f base))))))

(define czero
  (lambda (f base) base))

(define cone
  (lambda (f base) (f base)))

(define ctwo
  (lambda (f base) (f (f base))))

(define cthree
  (lambda (f base) (f (f (f base)))))
Church Addition

;; cplus : ChurchNum ChurchNum -> ChurchNum
;;; “Adds” the given ChurchNums together

(define cplus
  (lambda (m n)
    (lambda (f base)
      (m f (n f base))))))

(define czero
  (lambda (f base) base))

(define cone
  (lambda (f base) (f base)))

(define ctwo
  (lambda (f base) (f (f base))))

(define cthree
  (lambda (f base) (f (f (f base)))))
A Church Bool is a function with two arguments, where the representation of:
“true” returns the first arg, and
“false” returns the second arg

(define ctrue
  (lambda (a b) a))

(define cfalse
  (lambda (a b) b))

Returns first arg
Returns second arg
Review: “And”

The truth table of $A \land B$:

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$A \land B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

- When $A = True$, then $\text{And}(A, B) = B$
- When $A = False$, then $\text{And}(A, B) = A$
The truth table of $A \land B$:

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$A \land B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

- When $A =$ True, want $\text{And}(A, B) = B$
- When $A =$ False, want $\text{And}(A, B) = A$

```scheme
(define cand
  (lambda (A B)
    (A B A)))

(define ctrue
  (lambda (a b) a))

(define cfalse
  (lambda (a b) b))
```

- (Returns first arg)
- (Returns second arg)

;; cand: ChurchBool ChurchBool-> ChurchBool
;; “ands” the given ChurchBools together

$$\text{cand}(A, B) = ?$$

If $A =$ $\text{ctrue}$, want $\text{cand}(A, B) = B$.

If $A =$ $\text{cfalse}$, want $\text{cand}(A, B) = A$. 

$A = \text{True}$, $B = \text{True}$. 

$A = \text{False}$, $B = \text{True}$. 

$A = \text{False}$, $B = \text{False}$. 

$A = \text{True}$, $B = \text{False}$. 

$A = \text{False}$, $B = \text{False}$.
Church Pairs (Lists)

;;;; A ChurchPair<X,Y> 1-arg function, where
;;;; the arg fn is applied to (i.e., "selects") the X and Y data values

;;;; ccons: X Y -> ChurchPair<X,Y>
(define ccons
  (lambda (x y)
    (lambda (get)
      (get x y)))))

(define cfirst
  (lambda (cc)
    (cc (lambda (x y) x)))))

(define csecond
  (lambda (cc)
    (cc (lambda (x y) y)))))

"Gets" the first item

"Gets" the second item
The Lambda (\(\lambda\)) Calculus

- A “programming language” consisting of only:
  - Lambda functions
  - Function application

- “Language” has:
  - Numbers
  - Booleans and conditionals
  - Lists
  - ...
  - Recursion?
Recursion in the Lambda Calculus

Q: How can we write recursive programs with no-name lambdas?

Q: Is there a way for a lambda program to reference itself?
Lambda Program that Knows “Itself”

• Program that runs “itself” repeatedly (i.e., it infinite loops):

**Function** (takes one argument)

\[
\left( \left( \lambda \ x \ x \ x \right) \left( \lambda \ x \ x \ x \right) \right)
\]

**Function applies argument (function) to itself**

**Argument** (is also function)

Result is: The same program (i.e., the program “itself”)

• Can we do something else besides loop?
Lambda Program that Prints “Itself”

- Program that prints “itself”:

\[ ((\lambda (x) (\text{print2x } x)) \text{ "}(\lambda (x) (\text{print2x } x))\text{")} \]

<table>
<thead>
<tr>
<th>Function (takes one argument)</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\lambda (x) (\text{print2x } x)))</td>
</tr>
<tr>
<td>\text{&quot;}(\lambda (x) (\text{print2x } x))\text{&quot;)}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument (string)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{define (print2x str)}</td>
</tr>
<tr>
<td>\text{(printf (&quot;\text{~a\n ~v\n}&quot; str str))}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Result is:</th>
</tr>
</thead>
<tbody>
<tr>
<td>The same program (i.e., the program “itself”)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line break</th>
</tr>
</thead>
<tbody>
<tr>
<td>(could have inlined this)</td>
</tr>
</tbody>
</table>
Lambda Program that Prints “Itself”

• Program that prints “itself”:

```
(((\(x\) (print2x x))
  "((\(x\) (print2x x)))")
```

• Q: Which part of the program is “itself”? 
Lambda Program that Knows “Itself”

- Program that runs “itself” repeatedly (i.e., it infinite loops):

\[
\left( \left( \lambda (x) \left( x \ x \right) \right) \left( \lambda (x) \left( x \ x \right) \right) \right)
\]

- “itself” = “the recursive call”

Q: Which part of the program is “itself”?  
Can we do something more useful with “the recursive call”?
Delay “the recursive call”

What function “needs” a recursive call? A Recursive function!

Add a function parameter

What do we do with this?

Delay “recursive call”

“the recursive call”

Give “the recursive call” to another function that needs it
A Recursive Function

(define (factorial n)
  (if (zero? n)
    1
    (* n (factorial (sub1 n)))))
A Recursive Function, as lambda

```
(define factorial
 (λ (n)
   (if (zero? n)
       1
       (* n (factorial (sub1 n))))))
```
A Recursive Function without recursion

```
(define factorial
  (λ (n)
    (if (zero? n)
        1
        (* n (THE-RECURSIVE-CALL (sub1 n))))))
```

Where does this come from?

Make it a parameter!
A Recursive Function without recursion

\[
(\text{define factorial} \\
(\lambda \text{ (THE-RECURSIVE-CALL)} \\
(\lambda \, n) \\
\quad (\text{if} \, (\text{zero?} \, n) \\
\quad \quad 1 \\
\quad \quad (* \, n \, (\text{THE-RECURSIVE-CALL} \, (\text{sub1} \, n))))))
\]

Make “the recursive call” a parameter
A Recursive Function without recursion

```
(define factorial factorial-maker
  (λ (THE-RECURSIVE-CALL)
    (λ (n)
      (if (zero? n)
        1
        (* n (THE-RECURSIVE-CALL (sub1 n)))))))
```

Make “the recursive call” a parameter
Delay “the recursive call”

```
(((\(x\) \(x\) \(x\)) \\
  (\(x\) (x x))))
```

```
(((\(x\) \(x\) \(\lambda\(v\) \((x \(x\) \(v\))\))) \\
  (\(\lambda\(x\) \(\lambda\(v\) \((x \(x\) \(v\))\)))))
```

```
(\(\lambda\(f\)\) \\
  (((\(x\) \(x\) \(f\) \(\lambda\(v\) \((x \(x\) \(v\))\))) \\
    \((\lambda\(x\) \(f\) \(\lambda\(v\) \((x \(x\) \(v\))\))))))
```

f could be “fact-maker”
Y Combinator

“the recursive call”

(((λ (x) (x x))
  (λ (x) (x x))))

“the recursive call”

(((λ (x) (λ (v) ((x x) v))))
  (λ (x) (λ (v) ((x x) v))))

“the recursive call”

((λ (f)
  (((λ (x) (f (λ (v) ((x x) v)))))
   (λ (x) (f (λ (v) ((x x) v)))))))

Y Combinator “creates” recursive functions

f could be “fact-maker”
Code Demo
Check-In Quiz 10/23
on gradescope

(due 1 minute before midnight)