Class Information

INFORMATION and REMINDERS

• Homework 5: extension until Monday, November 13, 11:59pm.

• Homework 6 will be posted by Saturday.

• Second programming project will be posted next week.

• Final exam: Wednesday, December 20, 4:00-7:00pm.

DO YOU HAVE A CONFLICT?

http://sasundergrad.rutgers.edu/forms/final-exam-conflict

– More than two (2) final exams on one calendar day
– More than two (2) final exams scheduled in consecutive periods
– Two final exams scheduled for the same exam period.
Functions as arguments:

(define f (lambda (g x) (g x)))

• (f number? 0)
  ⇒ (number? 0) ⇒ #t

• (f length ’(1 2))
  ⇒ (length ’(1 2)) ⇒ 2

• (f (lambda (x) (* 2 x)) 3)
  ⇒ ((lambda (x) (* 2 x)) 3)
  ⇒ (* 2 3) ⇒ 6

REMINDER: Computation, i.e., function application is performed by reducing the initial S-expression (program) to an S-expression that represents a value. Reduction is performed by substitution, i.e., replacing formal by actual arguments in the function body.

Examples for S-expressions that directly represent values, i.e., cannot be further reduced:

• function values (e.g.: (lambda(x) e))
• constants (e.g.: 3, #t)
Higher-order Functions (Cont.)

Functions as returned values:

(define plusn
  (lambda (n) (lambda (x) (+ n x))))

• (plusn 5) evaluates to a function that adds 5 to its argument

  Question: How would you write down the value of (plusn 5)?

• ((plusn 5) 6) ⇒ 11
In general, any $n$-ary function

$$(\lambda (x_1 \ x_2 \ldots \ x_n) \ e)$$

can be rewritten as a nest of $n$ unary functions:

$$(\lambda (x_1)$$

$$(\lambda (x_2)$$

$$(\ldots (\lambda (x_n) \ e) \ldots)))$$

This translation process is called *currying*. It means that having functions with multiple parameters do not add anything to the expressiveness of the language.

*Question*: How to write an application of the original vs. the curried version?

$$((\lambda (x_1 \ x_2 \ldots \ x_n) \ e) \ v_1 \ v_2 \ldots \ v_n)$$

$$((\ldots$$

$$((\lambda (x_1)$$

$$(\lambda (x_2)$$

$$\ldots$$

$$(\lambda (x_n) \ e)\ldots))) \ v_1) \ v_2 \ldots \ v_n)$$
Higher-order Functions: map

(define map
  (lambda (f l)
    (if (null? l)
        ()
        (cons (f (car l)) (map f (cdr l))))))

• map takes two arguments: a function and a list
• map builds a new list by applying the function to every element of the (old) list
Higher-order Functions: map

- Example:

  \((\text{map abs } '(-1 2 -3 4)) \Rightarrow\)
  
  \((1 2 3 4)\)

  \((\text{map (lambda (x) (+ 1 x)) '(-1 2 -3)) \Rightarrow}\)
  
  \((0 3 -2)\)

- Actually, the built-in map can take more than two arguments:

  \((\text{map + ' (1 2 3) ' (4 5 6)) \Rightarrow}\)
  
  \((5 7 9)\)
reduce

Higher order function that takes a binary, associative operation and uses it to “roll-up” a list

\[
\text{(define reduce}
\begin{align*}
&\quad (\text{lambda} \ (\text{op} \ l \ \text{id}) \\
&\quad \quad \quad (\text{if} \ (\text{null?} \ l) \\
&\quad \quad \quad \quad \text{id} \\
&\quad \quad \quad \quad (\text{op} \ (\text{car} \ l) \ (\text{reduce} \ \text{op} \ (\text{cdr} \ l) \ \text{id})))
\end{align*}
\]

Example:

\[
(\text{reduce} \ + \ '(10 \ 20 \ 30) \ 0) \Rightarrow \\
(+ 10 \ (\text{reduce} \ + \ '(20 \ 30) \ 0)) \Rightarrow \\
(+ 10 \ (+ 20 \ (\text{reduce} \ + \ '(30) \ 0))) \Rightarrow \\
(+ 10 \ (+ 20 \ (+ 30 \ (\text{reduce} \ + \ '() \ 0)))) \Rightarrow \\
(+ 10 \ (+ 20 \ (+ 30 \ 0))) \Rightarrow \\
60
\]
More on Higher Order Functions

Now we can compose higher order functions to form compact powerful functions

Examples:

\[
(\text{define sum}
\begin{align*}
  & (\text{lambda } (f \ l) \\
  & \quad (\text{reduce } + \ (\text{map } f \ l) \ 0))
\end{align*}
) \Rightarrow
\]

\[
(\text{sum } (\text{lambda } (x) \ (* \ 2 \ x)) \ '(1 \ 2 \ 3)) \Rightarrow
\]

\[
(\text{reduce } (\text{lambda } (x \ y) \ (+ \ 1 \ y)) \ '(a \ b \ c) \ 0) \Rightarrow
\]
Lexical Scoping and \texttt{let}, \texttt{let*}, and \texttt{letrec}

All are variable binding operations:

\L E T \ = \ \texttt{let}, \texttt{let*}, \texttt{letrec}

\[
\text{(LET } ((v1 \ e1) \\
(v2 \ e2) \\
\ldots \\
(vn \ en)) \\
\text{e)}
\]

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

- \texttt{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).

- \texttt{letrec}: bindings of variables to values in no specific order; independent \textbf{evaluations of all} \(e_i\) \textbf{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

**λ-terms** (wffs) are inductively defined. A λ-terms is:

- a variable $x$
- $(λx.M)$ where $x$ is a variable and $M$ is λ-term (abstraction)
- $(M N)$ where $M$ and $N$ are λ-terms (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
  $$(λx. yz) \text{ is } (λx. (yz))$$
- “multiple” arguments
  $$(λxy.z) \text{ is } (λx. (λy.z))$$
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$(M N) = \text{free}(M) \cup \text{free}(N)$
- free$(\lambda x. M) = \text{free}(M) - \{x\}$
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.

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

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

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

What about:

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

\( \Rightarrow \) 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.

\[(\lambda x.M)N \Rightarrow \beta \quad [N/x]M \quad (\beta\text{–reduction})\]
\[\lambda x.M \Rightarrow \alpha \quad \lambda y.[y/x]M \quad (\alpha\text{–reduction})\]

if \(y \notin \text{free}(M)\)

Note:

- An equivalence relation can be defined based on \(\equiv\)–convertible \(\lambda\)-terms. “Reduction” rules really work both ways, but we are interested in reducing the complexity of \(\lambda\)-term (\(\to\) 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 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):

\[
\begin{align*}
\text{true} & \equiv \lambda a. \lambda b. a & \text{select–first} \\
\text{false} & \equiv \lambda a. \lambda b. b & \text{select–second} \\
\text{cond} & \equiv \lambda m. \lambda n. \lambda p. ((p \ m) n) \\
\text{not} & \equiv \lambda x. ((x \ \text{false}) \ \text{true}) \\
\text{equiv} & \equiv \underline{\text{homework}} \\
\text{or} & \equiv \lambda x. \lambda y. ((x \ \text{true}) \ y)
\end{align*}
\]
Next Lecture

Next time:

- lambda calculus