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:

\[
\begin{align*}
  f(x) &= x+2 \\
  \lambda x.x+2 \\
  (\lambda x.x+2) 1 &= 1+2 = 3 \\
  (\lambda x.x) (\lambda y.y) \\
  \lambda x.xx &
\end{align*}
\]

different notation

function application

and substitution

arguments and returned

“values” can be functions

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

\[ f : D^n \rightarrow D \quad \text{can be transformed to} \quad f : D \rightarrow (D \rightarrow \ldots (D \rightarrow D)) \ldots \]

Examples:

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

\(g_x(y)\) “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 \(\lambda\)-term 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 $\lambda$-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

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

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

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

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

binding occurrence  bound occurrence
free occurrence

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

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

- "free" is relative to a \( \lambda \)-subterm

- formal definition of \textit{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: $(\lambda b.2+b) \ x \ \approx \ [x/b]2+b \ \approx \ 2+x$

(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.

$(\lambda a.\lambda b.a+b) \ b \ \approx \ \lambda a.\lambda z.a+z) \ b \ \approx \ (\lambda z.b+z)$
Function application as substitution

Examples:

\[ [u/x] x \equiv u \quad \text{u not bound in } x \]
\[ [u/x] \lambda x.xu \equiv \lambda x.xu \quad \text{x not free in } \lambda x.xu \]
\[ [u/x] \lambda u.x \equiv [u/x] \lambda z.x \quad \text{u is bound in } \lambda u.x \]
\[ [u/x] \lambda u.u \equiv [u/x] \lambda z.z \]
\[ \lambda z.u \]
\[ \lambda z.z \]
Capture–free substitution

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

\[
\begin{align*}
[N/y] x &= x & \text{if } y \neq x \\
&= N & \text{if } y = x
\end{align*}
\]

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

\[
[N/x](\lambda y. M) = \lambda y.[N/x]M
\]

where $x \neq y$, $y \notin \text{free}(N)$,

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

where $x \neq y$, $y \in \text{free}(N)$,
\[
\text{and } z \notin \text{free}(M)
\] \[
\text{and } z \notin \text{free}(N)
\]

\[
[N/x](M P) = ([N/x]M [N/x]P)
\]
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.

\((\lambda x.M)N \Rightarrow_\beta [N/x]M\) (\(\beta\)-reduction)

\((\lambda x.M) \Rightarrow_\alpha \lambda y.[y/x]M\) (\(\alpha\)-reduction)
if \(y \notin \text{free}(M)\)

\((\lambda x.M)x \Rightarrow_\eta M\) (\(\eta\)-reduction)
if \(x \notin \text{free}(M)\)

Note:

• An equivalence relation can be defined based on \(\cong\)-convertable \(\lambda\)-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.

• \(\eta\)-reduction: one can remove (or introduce) levels of indirections in function applications.
Reduction — example

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

\[(\lambda yz.((\lambda xy.x)z)(yz))(\lambda xy.x)\]

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

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

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

\[(\lambda yz.z)(\lambda xy.x)\]

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

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

\[\lambda z.z\]
Reduction

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

true \( \equiv \lambda a.\lambda b.a \quad \) select–first
false \( \equiv \lambda a.\lambda b.b \quad \) select–second

cond \( \equiv \lambda m.\lambda n.\lambda p.((p m)n) \)
not \( \equiv \lambda x.((x \text{ false}) \text{ true}) \)
and \( \equiv \lambda x.\lambda y.((x \text{ y}) \text{ false}) \)
or \( \equiv \lambda x.\lambda y. ((x \text{ true}) \text{ y}) \)
Programming in lambda calculus

What about data structures?

data structures:
pairs can be represented as

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

first \( \equiv \lambda x.(x \, \text{true}) \) \quad (car)
second \( \equiv \lambda x.(x \, \text{false}) \) \quad (cdr)
build \( \equiv \lambda x.\lambda y.\lambda z.((z \, x) \, y) \) \quad (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 ≡ λfx.x
1 ≡ λfx.(f x)
2 ≡ λfx.(f (f x))
...

n ≡ λfx.(f(f(...(f x)...)))  ≡ λfx.(f^n x)

The natural number n is represented as a function that applies a function f n–times to its argument x.

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

Note: pred function can also be encoded, but the encoding is rather complicated.
Programming in lambda calculus

Examples:

\[(\text{mult} \ 2 \ 3) = \]
\[((\lambda mn.((\lambda fx.((m (n f)) x))) \ 2 \ 3) = \]
\[\lambda f_0 x_0.((2 \ \boxed{3 \ f_0}) x_0) = \]
\[\lambda f_0 x_0.((2 \ ((\lambda fx.((f (f (f x)))) \ f_0))) x_0) = \]
\[\lambda f_0 x_0.((2 \ ((\lambda x_1.(f_0 (f_0 x)))) x_0)) = \]
\[\lambda f_0 x_0.((\lambda x_1.(f_0 (f_0 x))) x_0) = \]
\[\lambda f_0 x_0.((\lambda x_1.(f_0 (f_0 x))) (f_3 x)) x_0) = \]
\[\lambda f_0 x_0.((\lambda x_1.(f_0 (f_0 x))) x_0) = \]
\[\lambda f_0 x_0.((f_3 (f_3 x_0)) = \]
\[\lambda f x.(f^6 x) = 6 \]

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

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

\[ \text{add} \equiv \lambda mn. \]
\[ \quad (\text{cond} m (\text{add} (\text{succ} m) (\text{pred} n)) (\text{isZero?} n)) \]

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

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

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

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

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:

\[ F \equiv \lambda f.(\lambda m n.\]

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

\[
((YF) 3 2) =
(((\lambda f.(\lambda x f(x x)) (\lambda x f(x x))) F) 3 2) =

\[
\left(F((\lambda x F(x x)) (\lambda x F(x x)))\right) 3 2) =

((\lambda m n.\text{if (isZero? n) then m else}
\text{((} \lambda x F(x x) \text{)} (\lambda x F(x x))\text{)}} (\text{succ } m) (\text{pred } n))) 3 2) =

\text{if (isZero? 2) then 3 else}
\text{((} \lambda x F(x x) \text{)} (\lambda x F(x x))\text{)}} (\text{succ 3) (pred 2)}) =

\[
\left((\lambda x F(x x)) (\lambda x F(x x))\right) 4 1) =

((F((\lambda x F(x x)) (\lambda x F(x x))))) 4 1) =

\text{if (isZero? 1) then 4 else}
\text{((} \lambda x F(x x) \text{)} (\lambda x F(x x))\text{)}} (\text{succ 4) (pred 1)}) =

\[
\left((\lambda x F(x x)) (\lambda x F(x x))\right) 5 0) =

((F( (\lambda x F(x x)) (\lambda x F(x x))))) 5 0) =

\text{if (isZero? 0) then 5 else}
\text{((} \lambda x F(x x) \text{)} (\lambda x F(x x))\text{)}} (\text{succ 5) (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 $eval(P)$:

```plaintext
case P:
  “x”: x
  “(λ x.M)”: $\lambda x.M$
  “(M N)”: 
    $e_1 = eval(M)$;
    error if not $e_1 \equiv \lambda x.M_1$;
    $e_2 = eval(N)$;
    $e_3 = [e_2/x]M_1$; //capture–free subst.
    $eval(e_3)$
endcase
```

Note:

- Values are variables or $\lambda$–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.
Functional Programming

Pure Functional Languages

Scott Chapter 10.

Fundamental concept: application of (mathematical) functions to values

1. Referential transparency: The value of a function application is independent of the context in which it occurs

   - value of $f(a,b,c)$ depends only on the values of $f$, $a$, $b$ and $c$
   - It does not depend on the global state of computation

   ⇒ all vars in function must be local, or parameters
Pure Functional Languages

1. The concept of assignment is **not** part of functional programming
   - no explicit assignment statements
   - variables bound to values only through the association of actual parameters to formal parameters in function calls
   - function calls have no side effects
   - thus no need to consider global state

2. Control flow is governed by function calls and conditional expressions
   => no iteration
   => recursion is widely used
Pure Functional Languages

1. All storage management is implicit
   • needs garbage collection

2. Functions are *First Class Values*
   • Can be returned as the value of an expression
   • Can be passed as an argument
   • Can be put in a data structure as a value
   • (Unnamed) functions exist as values
Pure Functional Languages

A program includes:

1. A set of function definitions
2. An expression to be evaluated

E.g. in Scheme:

```scheme
> (define (length x)
   (if (null? x)
       0
       (+ 1 (length (rest x)))))

> (length '(A LIST OF 5 THINGS))
5
```
LISP

• Functional language developed by John McCarthy in the mid 50’s
• Semantics based on Lambda Calculus
• All functions operate on lists or symbols: (called “S-expressions”)
• Only five basic functions: list functions cons, car, cdr, equal, atom and one conditional construct: cond
• Useful for list-processing applications
• Programs and data have the same syntactic form: S-expressions
• Used in Artificial Intelligence
• SCHEME: Developed in 1975 by G. Sussman and G. Steele as a version of LISP

⇒ we are using SCHEME here

You can call SCHEME interpreters on the ilab cluster by saying: mzscheme or racket (command line interpreter); dracket (window based interpreter).
S-expressions

S-expression ::= Atom | ‘(’ { S-expression } ‘)’
Atom ::= Name | Number | #t | #f

#t

()

(a b c)

(a (b c) d)

((a b c) (d e (f)))

(1 (b) 2)

Lists have nested structure.
Special (Primitive) Functions

- **eq?**: identity on names (atoms)
- **null?**: is list empty?
- **car**: selects first element of list *(contents of address part of register)*
- **cdr**: selects rest of list *(contents of decrement part of register)*
- **(cons element list)**: constructs lists by adding element to front of list
- **quote** or **’**: produces constants
Special (Primitive) Functions

- ’() is the empty list
- (car ’(a b c)) =
- (car ’((a) b (c d))) =
- (cdr ’(a b c)) =
- (cdr ’((a) b (c d))) =
Special (Primitive) Functions

• **car** and **cdr** can break up any list:
  
  – \((\text{car} \ (\text{cdr} \ (\text{cdr} \ '((a) \ b \ (c \ d)))))) = \)

  – \((\text{caddr} \ '((a) \ b \ (c \ d)))\)

• **cons** can construct any list:
  
  – \((\text{cons} \ 'a \ '()) = \)

  – \((\text{cons} \ 'd \ '() = \)

  – \((\text{cons} \ 'd \ (e)) = \)

  – \((\text{cons} \ 'd \ (e)) = \)

  – \((\text{cons} \ '(a \ b) \ '(c \ d)) = \)

  – \((\text{cons} \ 'a \ b \ c) \ '((a) \ b)) = \)
Other Functions

• + − ∗ / numeric operators, e.g.,
  (+ 5 3) = 8, (- 5 3) = 2
  (∗ 5 3) = 15, (/ 5 3) = 1.6666666

• = < > comparison operators for numbers

• Explicit type determination and test functions:
  ⇒ All return Boolean values: #f and #t
  – (number? 5) evaluates to #t
  – (zero? 0) evaluates to #t
  – (symbol? 'sam) evaluates to #t
  – (list? '(a b)) evaluates to #t
  – (null? '()) evaluates to #t

Note: SCHEME is a strongly typed language.
Other Functions

- `(number? 'sam)` evaluates to `#f`
- `(null? '(a))` evaluates to `#f`
- `(zero? (- 3 3))` evaluates to `#t`
- `(zero? '(- 3 3))` ⇒ type error
- `(list? (+ 3 4))` evaluates to `#f`
- `(list? '(+ 3 4))` evaluates to `#t`
**READ-EVAL-PRINT Loop**

**READ:** Read input from user: a function application

**EVAL:** Evaluate input:

\[(f \ arg_1 \ arg_2 \ \ldots \ arg_n)\]

1. evaluate \(f\) to obtain a function
2. evaluate each \(arg_i\) to obtain a value
3. apply function to argument values

**PRINT:** Print resulting value:

the result of the function application
> (cons ’a (cons ’b ’(c d)))
(a b c d)

1. Read the function application
   (cons ’a (cons ’b ’(c d)))

2. Evaluate cons to obtain a function

3. Evaluate ’a to obtain a itself

4. Evaluate (cons ’b ’(c d)):
   (a) Evaluate cons to obtain a function
   (b) Evaluate ’b to obtain b itself
   (c) Evaluate ’(c d) to obtain (c d) itself
   (d) Apply the cons function to b and (c d) to obtain (b c d)

5. Apply the cons function to a and (b c d) to obtain (a b c d)

6. Print the result of the application:
   (a b c d)
Quotes Inhibit Evaluation

;; Same as before:
> (cons 'a (cons 'b '(c d)))
(a b c d)

;; Now quote the second argument:
> (cons 'a '(cons 'b '(c d)))
(a cons (quote b) (quote (c d)))

;; Instead, un-quote the first argument:
> (cons a (cons 'b '(c d)))
ERROR: unbound variable: a
Quotes Inhibit Evaluation

;; Some things evaluate to themselves:
> (list 1 2 #t #f)
(1 2 #t #f)

;; They can also be quoted:
> (list '1 '2 '#t '#f)
(1 2 #t #f)
READ-EVAL-PRINT Loop

Can also be used to define functions.

**READ:** Read input from user:
   a symbol definition

**EVAL:** Evaluate input:
   store function definition

**PRINT:** Print resulting value:
   the symbol defined

Example:

```
(define (square x) (* x x))
#<unspecified>
```
Defining Global Variables

> (define foo '(a b c))
#<unspecified>

> (define bar '(d e f))
#<unspecified>

> (append foo bar)
(a b c d e f)

> (cons foo bar)
((a b c) d e f)

> (cons 'foo bar)
(foo d e f)
Defining Scheme Functions

\[\text{(define <fcn-name> (lambda (<fcn-params>) <expression>))}\]

Example: Given function \texttt{pair?} (true for non-empty lists, false o/w) and function \texttt{not} (boolean negation):

\[\text{(define atom? (lambda (object) (not (pair? object))))}\]

Evaluating \texttt{(atom? 'a)}:
1. Obtain function value for \texttt{atom?}
2. Evaluate \texttt{'a} obtaining \texttt{(a)}
3. Evaluate \texttt{(not (pair? object))}
   a) Obtain function value for \texttt{not}
   b) Evaluate \texttt{(pair? object)}
      i. Obtain function value for \texttt{pair?}
      ii. Evaluate \texttt{object} obtaining \texttt{(a)}

Evaluates to \texttt{#t}

Evaluates to \texttt{#f}

Evaluates to \texttt{#f}
Function Definition

Two syntaxes for definition:

1. (define (<fcn-name> <fcn-params>)
   <expression>)

   (define (square x)
     (* x x))

   (define (mean x y)
     (/ (+ x y) 2))

2. (define <fcn-name> (lambda (fcn-params)
     <expression>))

   (define square
     (lambda (n) (* n n)))

   (define mean
     (lambda (x y) (/ (+ x y) 2)))
Conditional Execution: if

(if <condition> <result1> <result2>)

1. Evaluate <condition>

2. If the result is a “true value” (i.e., anything but #f), then evaluate and return <result1>

3. Otherwise, evaluate and return <result2>

(define abs-val
  (lambda (x)
    (if (>= x 0) x (- x))))

(define rest-if-first
  (lambda (e l)
    (if (eq? e (car l)) (cdr l) '())))
Conditional Execution: `cond`

```
(cond (<condition1> <result1>))
  (<condition2> <result2>)
  ...
  (<conditionN> <resultN>)
  (else <else-result>)) ; optional else clause
```

1. Evaluate conditions in order until obtaining one that returns a true value
2. Evaluate and return the corresponding result
3. If none of the conditions returns a true value, evaluate and return `<else-result>`
Conditional Execution: cond

(define abs-val
  (lambda (x)
    (cond ((>= x 0) x)
          (else (- x))))

(define rest-if-first
  (lambda (e l)
    (cond ((null? l) '())
          ((eq? e (car l)) (cdr l))
          (else '()))))
Recursive Scheme Functions: Length

(define (length x)
  (if (null? x)
      0
      (+ 1 (length (cdr x)))))

trace (length '(1 2)):

(length '(1 2))
  x = (1 2)
  (length '(2))
    x = (2)
    (length '())
      x = ()
      value: 0
      value: (+ 1 0) = 1
      value: (+ 1 1) = 2
Recursive Scheme Functions: Abs-List

- \((\text{abs-list} \ ('(1 \ -2 \ -3 \ 4 \ 0))) \Rightarrow (1 \ 2 \ 3 \ 4 \ 0)\)
- \((\text{abs-list} \ '()) \Rightarrow ()\)

\[
\text{(define abs-list}
\begin{align*}
\text{(lambda (l)} \\
\text{\quad (if (null? l) \} \\
\text\quad \quad '()) \\
\text\quad \quad (\text{cons (abs-val (car l)) (abs-list (cdr l))))}
\end{align*}
\)
\]
Recursive Scheme Functions: Append

(append '(1 2) '(3 4 5) ⇒ (1 2 3 4 5)
(append '(1 2) '(3 (4) 5) ⇒ (1 2 3 (4) 5)
(append '() '(1 4 5)) ⇒ (1 4 5)
(append '(1 4 5) '()) ⇒ (1 4 5)
(append '() '()) ⇒ ()

(define append
  (lambda (x y)
    (cond ((null? x) y)
          ((null? y) x)
          (else (cons (car x) (append (cdr x) y))))))
Equality Checking

The eq? predicate doesn’t work for lists. Why not?

1. (cons ’a ’()) produces a new list
2. (cons ’a ’()) produces another new list
3. eq? checks if its two arguments are the same
4. (eq? (cons ’a ’()) (cons ’a ’())) evaluates to #f

Lists are stored as pointers to the first element (car) and the rest of the list (cdr). This elementary “data structure”, the building block of lists, is called a pair.

Symbols are stored uniquely, so eq? works on them.
Lists in Scheme

The building blocks for lists are pairs or cons-cells. Lists use the empty list () as an “end-of-list” marker.

Note: (a.b) is not a list!
Equality Checking for Lists

For lists, need a comparison function to check for the same structure in two lists

\[
(\text{define equal?})
\]
\[
(\text{lambda} \ (x \ y))
\]
\[
(\text{or} \ (\text{and} \ (\text{atom?} \ x) \ (\text{atom?} \ y) \ (\text{eq?} \ x \ y)) \ (\text{and} \ (\text{not} \ (\text{atom?} \ x)) \ (\text{not} \ (\text{atom?} \ y)) \ (\text{equal?} \ (\text{car} \ x) \ (\text{car} \ y)) \ (\text{equal?} \ (\text{cdr} \ x) \ (\text{cdr} \ y))))
\]

- \((\text{equal?} \ 'a \ 'a)\) evaluates to \#t
- \((\text{equal?} \ 'a \ 'b)\) evaluates to \#f
- \((\text{equal?} \ '(a) \ '(a))\) evaluates to \#t
- \((\text{equal?} \ '((a)) \ '(a))\) evaluates to \#f
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:
  \[
  (\text{map + } ' (1 2 3) ' (4 5 6)) \Rightarrow \\
  (5 7 9)\]
Review – Constants and Quotes

- Constants denote particular values. These values cannot be changed. Examples: 1, 2, #t, #f

- Function `quote` can be used to inhibit evaluation of its argument, converting it into data (e.g.: symbol or list). Examples: (quote a) (quote (a b c 1))
  
  Abbreviation: (quote a) \equiv 'a

- Functions `quasiquote` and `unquote` allow the construction of data structures by allowing unquoted expressions (including symbols) to be evaluated, and the value be inserted into the data structure.

  Example: ((lambda (m) (quasiquote (n (unquote (+ 1 m)) o))) 5)
  \Rightarrow (n 6 o)

  Abbreviations:
  
  (quasiquote (a b)) \equiv '(a b)
  
  (unquote m) \equiv ,m

- `unquote` does not lead to any evaluation in a quoted data structure

  Examples: ((lambda (m) (quote (n (unquote (+ 1 m)) o))) 5)
  \Rightarrow (n (unquote (+ 1 m)) o)
The TINY language

\[ \begin{align*}
x & \in \text{Variables} \\
n & \in \text{Integers} \\
c & ::= n \mid \#t \mid \#f \mid + \mid - \mid * \mid / \quad \text{constants} \\
v & ::= c \mid (\text{lambda } (x \ldots) \ e) \quad \text{values} \\
e & ::= v \mid x \mid (e \ e_1 \ldots e_k) \mid (\text{if } e_1 \ e_2 \ e_3) \quad \text{expressions} \\
p & ::= e \quad \text{program}
\end{align*} \]

This simple functional language does not have constructs to define recursive functions.

Note: \((\text{lambda } (x \ldots) \ e)\) is a function with one or more arguments (that’s what the “\ldots” mean).

We want TINY to be a \textbf{lexically scoped} language.
\[
ev[ ((\lambda (x) \\
  ((\lambda (z) \\
   ((\lambda (x) (z \ x)) \\
    3)) \\
  (\lambda (y) (+ x y))) ) \\
 1) ] = 4
\]

“Computation” can be characterized by choosing an application, and substituting formal parameters by their actual arguments.

Properties of Substitution

- Only formal parameters that are \textit{free} in the function body
- Only capture–free substitution
Observations

Is there any difference between call-by-value and call-by-name in terms of

- efficiency – How many reduction steps?
- answers computed – Different answers?

Examples:

- $$((\lambda x (+ x x)) ((\lambda x x) 4))$$
- $$((\lambda x (+ 1 1)) ((\lambda x x) 4))$$
- $$((\lambda x 1) ((\lambda x x x) ((\lambda x x x))))$$

Summary:

1. We expect call-by-name to have more reduction steps than call-by-value (exception: see above).
2. Upon termination, both produce the same answer.
3. There are cases where call-by-name terminates, but call-by-value does not.
Another Interpreter for TINY

GOAL: Write an “efficient” lexically (statically) scoped interpreter for our example language TINY

What does efficient mean?

substitution is expensive since it requires scanning the redex and perform textual replacement each time a function is applied.

How can we avoid substitution without changing the “reduction” semantics?

Answer: Use closures and environments
Environments

• Defer substitution by recording the bindings for the variables we would substitute in a data structure called an environment. If we need the value that a variable denotes, we just look it up in the environment.

  An environment is a finite map from variables to values

  \[ \rho \in Env = \text{Variables} \rightarrow \text{Values} \]
An Efficient Interpreter for TINY

Closures

- Pair the environment for the evaluation of an expression with the expression! The environment must contain values for all free variables of the expression. The expression can only be evaluated in its attached environment, making capturing impossible.

Such a pairing is called a closure. In our TINY language, only a lambda abstraction is a value that may contain free variables.

\[ \text{A closure is a pair consisting of an environment and a lambda abstraction} \]

\[ cl \in \text{Closure} = \{ \langle \lambda, \rho \rangle | \text{FreeVar}(\lambda) \subseteq \text{DOM}(\rho) \} \]
NOTE:

- Our set of values has changed! Values are now constants and closures, i.e., lambda abstraction “values” are always “embedded” in closures.

- The definitions of environments and closures are mutually recursive. However, since we do not consider recursion, we are in good shape, i.e., can ignore this fact.

Note: Our closure interpreter $ev$ takes a TINY program and an initial environment as input.
Our example revisited

\(((\text{lambda}(x)) \((\text{lambda}(z) \((\text{lambda}(x)(z \ x)) \ 3)) \ (\text{lambda}(y)(+ \ x \ y)))) \ 1)\)

<table>
<thead>
<tr>
<th>substitution</th>
</tr>
</thead>
<tbody>
<tr>
<td>(((\text{lambda}(z)) ((\text{lambda}(x)(z \ x)) \ 3)))</td>
</tr>
<tr>
<td>((\text{lambda}(y)(+ \ 1 \ y))))</td>
</tr>
<tr>
<td>(((\text{lambda}(x)) ((\text{lambda}(y)(+ \ 1 \ y)) \ x)) \ 3))</td>
</tr>
<tr>
<td>(((\text{lambda}(y)(+ \ 1 \ y)) \ 3))</td>
</tr>
<tr>
<td>((+ \ 1 \ 3))</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>
Our example revisited

How to apply a closure value to actual argument values?

1. Let $c_v$ be the closure value $\langle (\text{lambda}(x) \ e), \rho \rangle$.

2. Apply $c_v$ to a value $a_v$ as follows:
   Evaluate the body $e$ of the function in the environment $\rho$ of the closure extended by the mapping of the formal parameter $x$ to the actual value $a_v (\rho[x \rightarrow a_v])$.

$((\text{lambda}(x)
    ((\text{lambda}(z) ((\text{lambda}(x)(z \ x)) \ 3)) \ (\text{lambda}(y)(+ x y)))) \ 1)$

<table>
<thead>
<tr>
<th>closure interpreter</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>{ }</td>
</tr>
<tr>
<td>((\text{lambda}(z)</td>
</tr>
<tr>
<td>{x\rightarrow1}</td>
</tr>
<tr>
<td>((\text{lambda}(x)(z \ x)) \ 3))</td>
</tr>
<tr>
<td>(\text{lambda}(y)(+ x y))</td>
</tr>
<tr>
<td>((\text{lambda}(x)</td>
</tr>
<tr>
<td>{x\rightarrow1,</td>
</tr>
<tr>
<td>(z \ x) \ 3)</td>
</tr>
<tr>
<td>z\rightarrow( (\text{lambda}(y)(+ x y)),{x\rightarrow1})}</td>
</tr>
<tr>
<td>(+ x y)</td>
</tr>
<tr>
<td>{x\rightarrow1, y\rightarrow3}</td>
</tr>
</tbody>
</table>