Outline

- Previous lecture
  - Functional Programming Lecture
  - LISP
  - Some basics of Scheme

- Today’s lecture
  - More on Scheme
    - Determine the meaning of scheme expressions by developing a precise model for interpreting them
    - model: Substitution Model
    - Linear recursive, Linear iterative, Tree recursive processes

Scheme Expressions

- Primitive expression
  - Example: Numbers

- Compound expression
  - Combination of primitive expressions
    (+ 123 456)
  - When you type an expression, the Scheme interpreter responds by displaying the result of its evaluating that expression

Scheme combinations

- Expressions formed by delimiting a list of expressions within parentheses in order to denote procedure application

  (fun op1 op2 op3 ...)

- delimited by parentheses
- first element is the Operator
- the rest are Operands
Some simple combinations

(+ 2 3)
(- 4 1)
(+ 2 3 4)
(abs -7)
(abs (+ (- x1 x2) (- y1 y2)))

Meaning?

- What does a scheme expression mean?
- How do we know what value will be calculated by an expression?

(abs (+ (- x1 x2) (- y1 y2)))

Substitution Model

To evaluate a scheme expression:
1. Evaluate each of the operands
2. Evaluate the operator
3. Apply the operator to the evaluated operands

(fun op1 op2 op3 ...)

Organization of Programming Languages - Cheng (Fall 2004)
Example of expression evaluation

\[ (+ \ 3 \ (* \ 4 \ 5) ) \]

operator

operand 1

operand 2

Example of expression evaluation

- example: \((+ 3 (* 4 5))\)
- evaluate \(3 \ ? \ 3\)
- evaluate \(* \ 4 \ 5\)
  - evaluate \(4 \ ? \ 4\)
  - evaluate \(5 \ ? \ 5\)
  - evaluate \(* \ ? \ *\) (multiplication)
  - apply \(*\) to \(4, 5\) \(= 20\)
- evaluate \(+ \ ? \ +\) (addition)
- apply \(+\) to \(3, 20\) \(= 23\)

Primitive expressions

- Numbers evaluate to themselves
  - \(105 \ = 105\)
- Primitive procedures evaluate to their corresponding internal procedures
  - \(+, -, *, /, abs\ ...\)
  - e.g. \(+\) evaluates to the procedure that adds numbers together
Another example

\[(+ 3 (- 4 (* 5 (expt 2 2))))\]

- evaluate 3 \(\approx 3\)
- evaluate \((- 4 (* 5 (expt 2 2)))\)
  - evaluate 4 \(\approx 4\)
    - evaluate \((\times 5 (expt 2 2))\)
      - evaluate 5 \(\approx 5\)
      - evaluate \((expt 2 2)\)
        - evaluate 2 \(\approx 2\)
        - evaluate 2 \(\approx 2\)
        - apply \(\times\) to 2, 2 \(\approx 4\)
      - apply \(\div\) to 5, 4 \(\approx 20\)
      - apply \(-\) to 4, 20 \(\approx -16\)
    - apply \(\times\) to 3, -16 \(\approx -13\)

Note: Operator evaluation step was skipped

Evaluation with variables

- Association between a variable and its value
  - \(\text{define } X \equiv 3\)

- To evaluate a variable:
  - look up the value associated with the variable
    - e.g., X has the value of 3
  - and replace the variable with its value

Simple expression evaluation

- \(\text{define } X \equiv 3\)
- \(\text{define } Y \equiv 4\)
- \((+ X Y)\)
  - evaluate \(X \equiv 3\)
  - evaluate \(Y \equiv 4\)
  - evaluate \(+ \equiv \text{primitive procedure } +\)
  - apply \(+\) to 3, 4 \(\approx 7\)
Special forms

- There are a few special forms that do not evaluate in quite the same way as we've described.
- define is one of them.
- `(define X 3)`
- We do not evaluate X before applying define to X and 3.
- Instead, we simply:
  - evaluate the second argument (3).
  - Associate the second argument (3) to the first argument (X).

Lambda Expressions

- lambda is also a special form:
  - Result of a lambda is always a function.
  - We do not evaluate its contents.
  - just "save them for later."

Scheme function definition

```
A(r) = pi * r^2
```

```
(define A (lambda (r) (* pi (expt r 2))))
```
Evaluate lambda

\[\text{(define } A \text{ (lambda } (r) \text{ (* } \pi \text{ (expt } r \text{ 2)}))\text{)}\]

- evaluate (lambda \((r) \text{ (* } \pi \text{ (expt } r \text{ 2)}))\text{)}
  - create procedure with one argument \(r\) and body \(" \pi \text{ (expt } r \text{ 2)}\)\text{)}
  - create association between \(A\) and the new procedure

Evaluate a function call

- To evaluate a function call:
  [use the standard rule]
  - evaluate the arguments
  - apply the function to the (evaluated) arguments

- To apply a function call:
  - Substitute each argument with its corresponding value everywhere in the function body
  - Evaluate the resulting expression

Example 1

\[\text{(define } f \text{ (lambda } (x) \text{ (+ } x \text{ 1)}))\text{)}\]

- Evaluate \((f \text{ 2)}\)
  - evaluate 2 \(\equiv 2\)
  - evaluate f \(\equiv \text{(lambda } (x) \text{ (+ } x \text{ 1)}))\)
  - apply \((\text{lambda } (x) \text{ (+ } x \text{ 1)}))\text{ to } 2\)
    - Substitute 2 for \(x\) in the expression \((+ x 1) \text{ ? (x 2 1)}\)
    - Evaluate \((+ 2 1) \text{ ? 3}\)
Example 2

(define f (lambda (x y)
  (+ (* 3 x) (* -4 y) 2))))

Evaluate (f 3 2)
  evaluate 3 ≡ 3
  evaluate 2 ≡ 2
  evaluate f ≡ (lambda (x y) (+ (* 3 x) (* -4 y) 2))
  apply (lambda (x y) ...) to 3, 2
    Substitute 3 for x, 2 for y
    Evaluate (+ (* 3 3) (* -4 2)) ≡ 3

Equivalent forms of expressions

Equivalent expressions:
(define f (lambda (x) (+ x 1))
(define (f x) (+ x 1))

To evaluate: (define (f x) (+ x 1))
1. convert it into lambda form:
   (define f (lambda (x) (+ x 1)))
2. evaluate like any define:
   create the function (lambda (x) (+ x 1))
   create an association between the name f and the function

Example

(define sq (lambda (x) (* x x))
(define d (lambda (x y) (+ (sq x) (sq y))))

evaluate: (d 3 4)
  evaluate 3 ≡ 3
  evaluate 4 ≡ 4
  evaluate d ≡ (lambda (x y) (+ (sq x) (sq y))))
  apply (lambda (x y) ...) to 3, 4
Example (cont’d)

- Apply (lambda (x y) (+ (sq x) (sq y))) to 3, 4
- substitute 3 for x, 4 for y in (+ (sq x) (sq y))
- evaluate (+ (sq 3) (sq 4))
- evaluate (sq 3)
  - evaluate 3 \equiv 3
  - evaluate sq \equiv (lambda (x) (* x x))
  - apply (lambda (x) (* x x)) to 3
    - substitute 3 for x in (* x x)
    - evaluate (* 3 3)
    - evaluate 9
  - evaluate 9
- evaluate sq
  - evaluate (lambda (x) (* x x))
  - apply (lambda (x) (* x x)) to 4
    - substitute 4 for x in (* x x)
    - evaluate (* 4 4)
    - evaluate 16
- apply + to 9 and 16 \equiv 25
  which is final result
- Therefore, (d 3 4) \equiv 25
Substitution Model

- Gives precise model for evaluation
- Can carry out mechanically
  - by you
  - by the computer
- The model is simple but tedious to evaluate all those steps!

Lambdas evaluating to booleans

- Already seen functions that evaluate to numbers:
  - (define (circle-area r) (* pi (expt r 2)))
  - (circle-area 1)
  - evaluates to 3.1415...
- Now, functions that evaluate to booleans
  - (define (at-least-two-true a b c)
      (or (and a b)
           (and b c)
           (and a c)))

How does If expression evaluate?

- Evaluate the <boolean-expression>
- If true, evaluate the <true-case-expression>
- Otherwise, evaluate the <false-case-expression>
- The whole if-expression then evaluates to the result of either <true-case-expr> or <false-case-expr>
Formal substitution with \textit{if}

\begin{itemize}
\item (define (max a b) (if (> a b) a b))
\item Evaluate (max 3 2)
\item Evaluate 3 \rightarrow 3
\item Evaluate 2 \rightarrow 2
\item Evaluate max \rightarrow (lambda (a b) (if (> a b) a b))
\item Apply (lambda (a b) (if (> a b) a b)) to 3,2
\item Substitute 3 for \(a\), 2 for \(b\) in (if (> a b) a b)
\item Evaluate (if (> 3 2) 3 2)
\item Evaluate (if > 3 2) \rightarrow #t
\item (if \#t 3 2)
\item Evaluate 3 \rightarrow 3
\end{itemize}

\textbf{If is an Expression}

\begin{itemize}
\item Since “if” is an expression, it “returns a value” or “evaluates to a value”
\item This is different from many programming languages
\item You can use that value
\end{itemize}

Example

\begin{itemize}
\item (define (scale a b) (/ (if (> a b) a b) (if (> a b) b a)))
\item Evaluate: (scale 4 2)
\item Evaluate 4 \rightarrow 4
\item Evaluate 2 \rightarrow 2
\item Evaluate scale \rightarrow (/ (if (> a b) a b) (if (> a b) b a))
\item Apply (lambda (a b) (/ (if (> a b) a b) (if (> a b) b a))) to 4,2
\item Substitute 4 for \(a\), 2 for \(b\)
\item (/ (if (> 4 2) 4 2) (if (> 4 2) 2 4))
\item (/ (if > 4 2) (if > 4 2))
\item (/ 4 2)
\item Evaluate 2 \rightarrow 2
\end{itemize}
Recursion

Example:
How do I compute the sum of the first N integers?

Sum of first N integers =
N + N-1 + N-2 + … + 1
N + (N-1 + N-2 + … + 1)
N + [Sum of first N-1 integers]

Convert to scheme:
(define (sum-integers n)
  (if (= n 0)
      0
      (+ n (sum-integers (- n 1)))))

Evaluating

Evaluates: (sum-integers 3)
(if (= 3 0) 0 (+ 3 (sum-integers (- 3 1))))
(if #f 0 (+ 3 (sum-integers (- 3 1))))
(+ 3 (sum-integers (- 2 1))))
(+ 3 (+ (sum-integers (- 2 1))))
(+ 3 (+ 2 (sum-integers (- 2 1))))
(+ 3 (+ 2 (+ 1 (sum-integers (- 1 1))))))
(+ 3 (+ 2 (+ 1 (+ 0 (sum-integers (- 0 1))))))
(+ 3 (+ 2 (+ 1 0)))
6

Linear Recursive

This is what we call a linear recursive process
Makes linear calls
Keeps a chain of deferred operations
linear w.r.t. input
Another strategy

- To sum integers...
- Add up as we go along:
  - 1 2 3 4 5 6 7 8 9 10 15 21 28 ....
  - 1 3 6 10 15 21 28 ....

<table>
<thead>
<tr>
<th>Current</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1 + 0 = 1</td>
</tr>
<tr>
<td>2</td>
<td>2 + 1 = 3</td>
</tr>
<tr>
<td>3</td>
<td>3 + 3 = 6</td>
</tr>
<tr>
<td>4</td>
<td>4 + 6 = 10</td>
</tr>
<tr>
<td>5</td>
<td>5 + 10 = 15</td>
</tr>
</tbody>
</table>

Alternate definition

```
(define (sum-int n)
  (sum-iter 0 n 0))
(define (sum-iter current max sum)
  (if (> current max)
   sum
   (sum-iter (+ 1 current) max
             (+ current sum))))
```

Evaluation of sum-int

```
(define (sum-int n)
  (sum-iter 0 n 0))
(define (sum-iter current max sum)
  (if (> current max)
   sum
   (sum-iter (+ 1 current) max
             (+ current sum))))
```

```
(sum-int 3)
(sum-iter 0 3 0)
(if (> 0 3) ...
(sum-int 3)
(if (> 1 3) ...
(sum-iter 2 3 1)
(if (> 2 3) ...
(sum-int 3)
(if (> 3 3) ...
(sum-iter 4 3 6)
(if (> 4 3) ...
6)
```
Difference from Linear Recursive

(linear recursive process)

```
(sum-integers 3)
(+ 3 (+ 2 (+ 1 (+ 0))))
```

(linear iterative process)

```
(sum-int 3)
(sum-iter 0 3 0)
(sum-iter 1 3 0)
(sum-iter 2 3 1)
(sum-iter 3 3 3)
(sum-iter 4 3 6)
```

Linear Iterative

- (sum-int 3)
- (sum-iter 0 3 0)
- (sum-iter 1 3 0)
- (sum-iter 2 3 1)
- (sum-iter 3 3 3)
- (sum-iter 4 3 6)
- 6

This is what we call a linear iterative process
- Makes linear calls
- State of computation kept in a constant number of state variables
- Does not grow with problem size

Fibonacci Numbers

Fibonacci sequence
0, 1, 1, 2, 3, 5, 8, 13, 21

What's the rule for this sequence?
- \( \text{Fib}(0) = 0 \)
- \( \text{Fib}(1) = 1 \)
- \( \text{Fib}(N) = \text{Fib}(N-1) + \text{Fib}(N-2) \)
Simple Scheme Translation

What’s the rule for this sequence?
- $Fib(0) = 0$
- $Fib(1) = 1$
- $Fib(N) = Fib(N-1) + Fib(N-2)$

**(define (fib n)**
  (if (< n 2)
    n
    (+ (fib (- n 1)) (fib (- n 2)))))

Has two recursive calls for each evaluation

Evolution of fib

(fib 5)
(+ (fib 4) (fib 3))
(+ (+ (fib 3) (fib 2)) (+ (fib 2) (fib 1)))
(+ (+ (+ (fib 2) (fib 1)) (fib 0))) (+ (+ (fib 1) (fib 0)) 1)
(+ (+ (+ (fib 1) (fib 0)) 1) (+ 1 0)) (+ (+ 1 0) 1)

Fib Evolution

Expands into a “tree” instead of linear chain

(fib n)
(fib (- n 1)) (fib (- n 2))
(fib (- n 2)) (fib (- n 3)) (fib (- n 3)) (fib (- n 4))
Tree Recursion

- `(fib 5)`
- `(+ (fib 4) (fib 3))`
- `(+ (fib 5) (fib 2))`
- `(+ (fib 2) (fib 1))`

This is what we call a **tree recursive process**

- Calls fan out in a tree
- How many calls to fib?
- Exponential in call argument
- (Not perfectly "balanced" tree)

Alternate Method

- Again, count up
- n: 0 1 2 3 4 5 6 7 ... 
- Fib(n): 0 1 1 2 3 5 8 13 ... 

Example: Fib(5)

<table>
<thead>
<tr>
<th>Count</th>
<th>Fnext</th>
<th>Fib</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1+1=2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2+1=3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3+2=5</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>5+3=8</td>
<td>8</td>
</tr>
</tbody>
</table>

Linear Iterative

- `(define (fib n)
  (fib-iter 1 0 n))`

- `(define (fib-iter fnext f cnt)
  (if (= cnt 0)
    f
    (fib-iter (+ fnext f) fnext (- cnt 1)))))`
Evolution of Iterative Fib

(define (fib-iter fnext f cnt)
  (if (= cnt 0)
      f
      (fib-iter (+ fnext f) fnext (- cnt 1))))