6.001--Another word of Introduction

If the automobile had followed the same development as the computer, a Rolls-Royce would today cost $100, get a million miles per gallon, and explode once a year killing everyone inside.

-- Robert Cringely, InfoWorld

Last lecture

• Basics of Scheme
  – Self-evaluating expressions
  – Names
  – Define
• Rules for evaluation

This lecture

• Adding procedures and procedural abstractions
• Using procedures to capture processes

Language elements -- abstractions

• Need to capture ways of doing things – use procedures

\( \text{Parameters} \rightarrow \text{Body} \)

To process something multiply it by itself

• Special form – creates a procedure and returns it as value

Language elements -- abstractions

• Use this anywhere you would use a procedure

\( \text{((lambda} (x) (\ast x x)) 5) \)

Scheme Basics

• Rules for evaluation
  1. If self-evaluating, return value.
  2. If a name, return value associated with name in environment.
  3. If a special form, do something special.
  4. If a combination, then
     a. Evaluate all of the subexpressions of combination (in any order)
     b. apply the operator to the values of the operands (arguments) and return result.

• Rules for application
  1. If procedure is \textit{primitive procedure}, just do it.
  2. If procedure is a \textit{compound procedure}, then evaluate the body of the procedure with each formal parameter replaced by the corresponding actual argument value.
Language elements -- abstractions

- Use this anywhere you would use a procedure

\[(\text{lambda} (x) (+ x x))\] 6
\[(* 5 5)\]

25

Can give it a name

\[(\text{define} \text{square} (\text{lambda} (x) (* x x)))\]

\[(\text{square} 5)\] \[\rightarrow 25\]

Lambda: just what is a procedure?

\[(+ 3 4)\]

7

\[(\text{lambda} (x) (* x x))\]

#\[compound-procedure 9\]

Remember the READ-EVAL-PRINT loop:

User types “\(+ 3 4\)”

Scheme prints “7”

Interaction of define and lambda

1. \[(\text{lambda} (x) (* x x))\] \[\rightarrow #\[compound-procedure 9\]\]
2. \[(\text{define} \text{square} (\text{lambda} (x) (* x x)))\] \[\rightarrow \text{undef}\]
3. \[(\text{square} 4)\] \[\rightarrow 16\]
4. \[(\text{lambda} (x) (* x x)) 4\] \[\rightarrow 16\]
5. \[(\text{define} \text{square} x) (* x x))\] \[\rightarrow \text{undef}\]

This is a convenient shorthand (called “syntactic sugar”) for 2 above -- this is a use of lambda!

Lambda special form

- lambda syntax \[(\text{lambda} (x y) (/ (+ x y) 2))\]

- 1st operand position: the parameter list \[(x y)\]
  - a list of names (perhaps empty)
  - determines the number of operands required

- 2nd operand position: the body \[(/ (+ x y) 2)\]
  - may be any expression
  - not evaluated when the lambda is evaluated
  - evaluated when the procedure is applied

- semantics of lambda:

THE VALUE OF A LAMBDA EXPRESSION IS A PROCEDURE
Corollaries

- Any time you evaluate a lambda-expression, you create a procedure
- The only way to create a procedure is to evaluate a lambda-expression

Using procedures to describe processes

- How can we use the idea of a procedure to capture a computational process?

What does a procedure describe?

- Capturing a common pattern
  - (* 3 3)
  - (* 25 25)
  - (* foobar foobar)

  (lambda (x) (* x x))

Common pattern to capture

Name for thing that changes

Modularity of common patterns

Here is a common pattern:

(sqr (* 3 3) (* 4 4))
(sqr (* 9 9) (* 16 16))
(sqr (* 4 4) (* 4 4))

Here is one way to capture this pattern:

(define pythagoras
  (lambda (x y) (sqr (+ (* x x) (* y y)))))

Why?

- Breaking computation into modules that capture commonality
  - Enables reuse in other places (e.g. square)
  - Isolates details of computation within a procedure from use of the procedure
  - May be many ways to divide up

(define square (lambda (x) (* x x)))
(define sum-squares
  (lambda (x y) (+ (square x) (square y))))
(define pythagoras
  (lambda (y x) (sqrt (sum-squares y x))))
Abstracting the process

• Stages in capturing common patterns of computation
  – Identify modules or stages of process
  – Capture each module within a procedural abstraction
  – Construct a procedure to control the interactions between the modules
  – Repeat the process within each module as necessary

A more complex example

• Remember our method for finding sqrts
  – To find the square root of X
    • Make a guess, called G
    • If G is close enough, stop
    • Else make a new guess by averaging G and X/G

The stages of “SQRT”

• When is something “close enough”
• How do we create a new guess
• How to we control the process of using the new guess in place of the old one

Procedural abstractions

For “close enough”:

\[
\text{(define close-enuf?}
\quad \text{(lambda (guess x)}
\quad \quad (> (abs (- (square guess) x)) 0.001)))
\]

Note use of procedural abstraction!

Procedural abstractions

For “improve”:

\[
\text{(define improve}
\quad \text{(lambda (guess x)}
\quad \quad \text{(average guess}
\quad \quad \quad \text{(average guess}
\quad \quad \quad \quad \text{(/ (+ a b) 2)})
\end{equation}

Why this modularity?

• “Average” is something we are likely to want in other computations, so only need to create once
• Abstraction lets us separate implementation details from use
  – E.g. could redefine as
    \[
    \text{(define average}
    \quad \text{(lambda (x y) (* (+ x y) 0.5)))}
    \]
  – No other changes needed to procedures that use average
  – Also note that variables (or parameters) are internal to procedure – cannot be referred to by name outside of scope of lambda

}
Controlling the process

• Basic idea:
  – Given X, G, want \((\text{improve } G \ X)\) as new guess
  – Need to make a decision – for this need a new special form
    \[(\text{if } <\text{predicate}> <\text{consequent}> <\text{alternative}>)
    \]

The \textbf{IF} special form

\[(\text{if } <\text{predicate}> <\text{consequent}> <\text{alternative}>)
\]

– Evaluator first evaluates the \(<\text{predicate}>\)
  expression.
– If it evaluates to a TRUE value, then the evaluator
  evaluates and returns the value of the
  \(<\text{consequent}>\)
  expression.
– Otherwise, it evaluates and returns the value of the
  \(<\text{alternative}>\)
  expression.
– Why must this be a special form?
  \[(\text{if } (\text{zero? } x) \ 42 \ (/ \ 1.0 \ x))\]

Controlling the process

• Basic idea:
  – Given X, G, want \((\text{improve } G \ X)\) as new guess
  – Need to make a decision – for this need a new special form
    \[(\text{if } <\text{predicate}> <\text{consequent}> <\text{alternative}>)
    \]
  – So heart of process should be:
    \[(\text{define } \sqrt\text{-loop} \ (\lambda G X) (\text{if } (\text{close-enuf? } G \ X) \ G \ (\text{improve } G \ X)) )\]
  – But somehow we want to use the value returned by “improving” things as the new guess, and repeat the process

Putting it together

• Then we can create our procedure, by simply starting with some initial guess:
  \[(\text{define } \sqrt (\lambda x) (\text{sqrt\text{-loop} } 1.0 \ x))\]

Checking that it does the “right thing”

• Next lecture, we will see a formal way of tracing evolution of evaluation process
• For now, just walk through basic steps
  \[(\sqrt 2)\]
  \[
  (\text{sqrt\text{-loop} } 1.0 \ 2)
  \]
  \[
  (\text{if } (\text{close-enuf? } 1.0 \ 2) \ ...)
  \]
  \[
  (\text{sqrt\text{-loop} } \text{improve } 1.0 \ 2)
  \]
  This is just like a normal combination
  \[
  (\text{sqrt\text{-loop} } 1.5 \ 2)
  \]
  \[
  (\text{if } (\text{close-enuf? } 1.5 \ 2) \ ...)
  \]
  \[
  (\text{sqrt\text{-loop} } 1.4166666 \ 2)
  \]
  • And so on...
Abstracting the process

- Stages in capturing common patterns of computation
  - Identify modules or stages of process
  - Capture each module within a procedural abstraction
  - Construct a procedure to control the interactions between the modules
  - Repeat the process within each module as necessary

Summarizing Scheme

- Primitives
  - Numbers
  - Strings
  - Booleans
  - Built-in procedures
- Means of Combination
  - (procedure argument1 argument2 ... argumentn)
- Means of Abstraction
  - Lambda
  - Define
- Other forms
  - if

1, -2.5, 3.67e25

Creates a loop in system
- allows abstraction of name for object

Enables means of combination

Create a procedure

Create names

Control order of evaluation