Previous lecture

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

This lecture

Adding procedures and procedural abstractions to capture processes

Language elements -- procedures

- Need to capture ways of doing things – use procedures
  \[ (\text{lambda} (x) (\ast x x)) \]
  - (lambda) \text{parameters} \text{body}
  - To process \text{something} \text{multiply it by itself}
  - Special form – creates a procedure and returns it as value

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 primitive procedure, just do it.
  2. If procedure is a compound procedure, then:
     a. Evaluate the body of the procedure with each formal parameter replaced by the corresponding actual argument value.

Lambda: making new procedures

expression \hspace{2cm} \text{printed representation of value}
\[ (\text{lambda} (x) (\ast x x)) \]
\#\text{[compound-procedure 9]}

A compound proc that squares its argument
Interaction of define and lambda

1. `(lambda (x) (* x x))`  
   ```  
   => #\[compound-procedure 9\]
   ```

2. `(define square (lambda (x) (* x x)))`  
   ```  
   => undef
   ```

3. `(square 4)`  
   ```  
   ==> 16
   ```

4. `((lambda (x) (* x x)) 4)`  
   ```  
   ==> 16
   ```

5. `(define (square x) (* x x))`  
   ```  
   => undef
   ```

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

Lambda special form

- lambda syntax  
  ```  
  (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

Using procedures to describe processes

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

THE VALUE OF A LAMBDA EXPRESSION IS A PROCEDURE

What does a procedure describe?

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

  ```  
  (lambda (x) (* x x))
  ```

  Name for thing that changes

Modularity of common patterns

Here is a common pattern:

```  
(sqrt (+ (* 3 3) (* 4 4)))
(sqrt (+ (* 3 9) (* 16 16)))
(sqrt (+ (* 4 4) (* 4 4)))
```  

Here is one way to capture this pattern:

```  
(define pythagoras
  (lambda (x y)
    (sqrt (+ (* x x) (* y y))))
)
Modularity of common patterns

Here is a common pattern:

\[
\text{sqrt} (+ (* 3 3) (* 4 4)) \\
\text{sqrt} (+ (* 9 9) (* 16 16)) \\
\text{sqrt} (+ (* 4 4) (* 4 4))
\]

But here is a cleaner way of capturing patterns:

\[
\begin{align*}
&\text{define square (lambda (x) (* x x))} \\
&\text{define pythagoras (lambda (x y) (sqrt (+ (square x) (square y))))}
\end{align*}
\]

Why?

- Breaking computation into modules that capture commonality
  - Enables reuse in other places (e.g. square)
- Isolates (abstracts away) details of computation within a procedure from use of the procedure
  - Useful even if used only once (i.e., a unique pattern)

\[
\begin{align*}
&\text{define comp (x y) (/ (+(* x y) 17) (-(+ x y) 4)))} \\
&\text{define comp (x y) (/ (prod+17 x y) (sum-4 x y))}
\end{align*}
\]

Why?

- May be many ways to divide up

\[
\begin{align*}
&\text{define square (lambda (x) (* x x))} \\
&\text{define pythagoras (lambda (x y) (sqrt (+ (square x) (square y))))}
\end{align*}
\]

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 do we control the process of using the new guess in place of the old one
Procedural abstractions

For “close enough”:

\[
\text{(define close-enuf?}
\begin{align*}
&\text{(lambda (guess x))} \\
&\text{(\textless (abs (- \text{(square guess)} x)) 0.001))}
\end{align*}
\]

Note use of procedural abstraction!

Procedural abstractions

For “improve”:

\[
\text{(define average}
\begin{align*}
&\text{(lambda \{a b\} (/ (+ a b) 2)))}
\end{align*}
\]

\[
\text{(define improve}
\begin{align*}
&\text{(lambda (guess x))} \\
&\text{(average guess (/ x guess))))}
\end{align*}
\]

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
- Originally:
  \[
  \text{(define average}
  \begin{align*}
  &\text{(lambda \{a b\} (/ (+ a b) 2)))}
  \end{align*}
  \]
- Could redefine as
- 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

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

The \textbf{IF} special form

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

- Evaluator first evaluates the <predicate> expression.
- If it evaluates to a TRUE value, then the evaluator evaluates and returns the value of the <consequence> expression.
- Otherwise, it evaluates and returns the value of the <alternative> expression.
- Why must this be a special form?

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
  \[
  \text{(if <predicate> <consequence> <alternative>)}
  \]
  - So heart of process should be:

\[
\text{(if (close-enuf? G X) G}
\begin{align*}
&\text{(improve G X)}
\end{align*}
\]

- But somehow we want to use the value returned by “improving” things as the new guess, and repeat the process
Controlling the process

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

Putting it together

- Then we can create our procedure, by simply starting with some initial guess:
  (define sqrt
    (lambda (x)
    (sqrt-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)
  - (sqrt-loop 1.0 2)
  - (if (close-enuf? 1.0 2) ... ...)
  - (sqrt-loop (improve 1.0 2) 2)
    This is just like a normal combination
  - (sqrt-loop 1.5 2)
  - (if (close-enuf? 1.5 2) ... ...)
  - (sqrt-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 1, -2.5, 3.67e25
  - Strings
  - Booleans
  - Built in procedures +, -, /, =, >, <, .
  - Names
- Means of Combination
  - (procedure argument1 argument2 ... argumentn)
- Means of Abstraction
  - Lambda
  - Define
- Other forms
  - if

Creates a loop in system – allows abstraction of name for object
Creates a procedure
Enables means of combination
Control order of evaluation