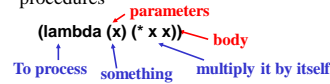


This lecture

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

Language elements -- abstractions

- Need to capture ways of doing things – use procedures



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

Language elements -- abstractions

- Use this anywhere you would use a procedure
((lambda (x) (* 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 **primitive procedure**, just do it.
 2. If procedure is a **compound procedure**, then:
evaluate the body of the procedure with each formal parameter replaced by the corresponding actual argument value.

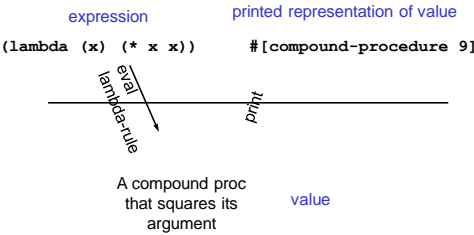
Language elements -- abstractions

- Use this anywhere you would use a procedure

((lambda (x) (* x x)) 5)
(* 5 5)
25

- Can give it a name
(define square (lambda (x) (* x x)))
(square 5) → 25

Lambda: making new procedures



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
 - evaluated when the procedure is applied
- semantics of lambda:

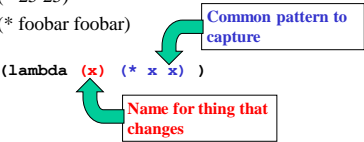
THE VALUE OF
A LAMBDA EXPRESSION
IS
A PROCEDURE

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)



Modularity of common patterns

Here is a common pattern:

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

But here is a cleaner way of capturing patterns:

```
(define square (lambda (x) (* x x)))  
(define pythagoras  
  (lambda (x y)  
    (sqrt (+ (square x) (square 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

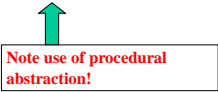
- Remember our method for finding sqrts
 - To find the square root of X
 - Make a guess, called G
 - If G is close enough, stop
 - Else make a new guess by averaging G and X/G

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”:
(define close-enuf?
  (lambda (guess x)
    (< (abs (- (square guess) x)) 0.001)))
```



Procedural abstractions

```
For “improve”:
(define average
  (lambda (a b) (/ (+ a b) 2)))
(define improve
  (lambda (guess x)
    (average guess (/ x guess)))))
```

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

```
(define average
  (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 (**improve G X**) as new guess
 - Need to make a decision – for this need a new *special form*

```
(if <predicate> <consequence> <alternative>)
```

The IF special form

```
(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 (**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:

```
(if (close-enuf? G X)
    G
    (improve G X))
```

- 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...

9/30/2015

Comp.101 SICP

25

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

9/30/2015

Comp.101 SICP

26