CSCI: 4500/6500 Programming Languages

Functional Programming Languages

Part 3: Evaluation and Application Cycle

Lazy Evaluation

LuisGuillermo.com

- **Meta circular evaluaturs**
- **Evaluate & Apply**
- **Lazy and Aggressive Evaluation**

Back to the Basics: Steps in Inventing a Language

- **Design the grammar**
  - » **What strings are in the language?**
  - » **Use BNF to describe all the strings in the language**
- **Make up the evaluation rules**
  - » **Describe what everything the grammar can produce means**
- **Build an *evaluator***
  - » **A procedure that evaluates expressions in the language**
    - – **The evaluator:**
      - determines the meaning of expressions in the programming language, is just another program.

Programming an Evaluator

- **If a language is just a program, what *language* should we program the language (evaluator) in?**

Definition: A Metacircular Evaluator

- **An evaluator that is written in the *same* language that it evaluates is said to be *metacircular***

  Sounds like recursion: It's circular recursion. There is no termination condition. It's a chicken-and-the-egg kind of thing. (There's actually a hidden termination condition: the bootstrapping process.)

- **One more requirement: The language interpreted does not need additional definitions of semantics other than that is defined for the evaluator (sounds circular).**
  - » **Example:**
    - – **The C compiler is written is C but is not meta circular because the compiler specifies extremely detailed and precise semantics for each and every construct that it interprets.**

Evaluation Basics

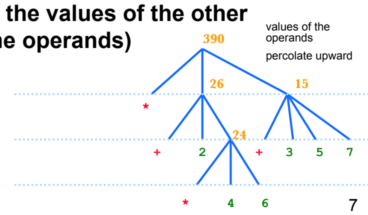
Observation: This is recursive

To **evaluate** a combination:

- **Evaluate** each element (all the **subexpressions**) of the combination
- **Apply** the procedure to the value of the left-most subexpression (the operator) to the arguments that are the values of the other subexpressions (the operands)

Evaluation rule is applied on 4 combinations:

```
(* (+ 2 (* 4 6))
   (+ 3 5 7) )
```



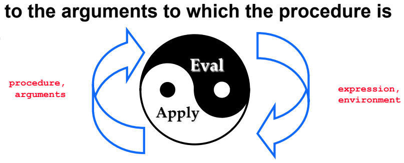
Example: Procedural Building Blocks

```
( define (square x) (* x x) )
   » ( square (+ 2 5) ) ⇒ 49
( define (sum-of-squares x y)     ; use square for
   (+ (square x) (square y) )     ; x² +  y²
   » ( sum-of-squares 3 4) ⇒ 25
( define (f a)
  ( sum-of-squares (+ a 1) (* a 2)))
   » (f 5) ⇒ 136
```

- **square** - is a compound procedure which is given the name square which is represents the operation of multiplying something by itself.
- **Evaluating** the definition creates the compound procedure and associates it with the name square (lookup)
- **Application**: To apply a compound procedure to arguments, evaluate the body of the procedure with each formal parameter replaced by the 'real' arguments. (substitution model -- an assignment model <-variable<-env)

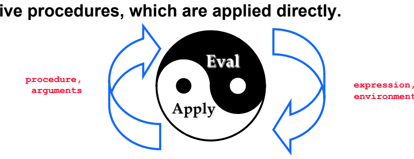
Environmental Model of Evaluation

1. To **evaluate** a combination (compound expression)
   - **evaluate** all the subexpressions and then
   - **apply** the value of the operator subexpression (first expression) to the values of the operand subexpressions (other expressions).
2. To **apply** a procedure to a list of arguments,
   - **evaluate** the body of the procedure in a **new *environment (by a frame)*** that **binds** the formal parameters of the procedure to the arguments to which the procedure is applied to.



Core of the Evaluator

- Basic cycle in which
  - » expressions to be evaluated in environments are
  - » reduced to procedures to be **applied** to arguments,
- Which in turn are reduced to new expressions
  - » to be **evaluated** in new environments, and so on,
  - » until we get down to
    - – symbols, whose values are **looked up** in the environment
    - – primitive procedures, which are applied directly.



The evaluator - metacircularity `( eval expression environment )`

- Evaluates the the `expression` relative to the `environment`
  - » **Examples**: environments (returns a specifies for the environment)
    - – `scheme-report-environment version`
    - – `null-environment version`
- Primitives:
  - » self-evaluating expressions, such as numbers, `eval` returns the expression itself
  - » variables, looks up variables in the **environment**
- Some special forms (`lambda`, `if`, `define` etc). `eval` provide direct implementation:
  - » Example: **quoted**: returns expression that was quoted
- Others lists:
  - » **eval calls itself recursively on each element and then calls apply,** passing as argument the value of the first element (which must be a function) and a list of the remaining elements. Finally, eval returns what apply returned

Eval

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type - EVAL" exp)))
```

Eval: Example

```
(eval '( * 7 3 ) (scheme-report-environment 5))
      => 21
(eval (cons '* (list 7 3)) (scheme-report-environment 5))
      => 21
```

Current Scheme doesn't recognize 'scheme-report-environment'

apply

- `apply` applies its first argument (a function) and applies it to its second argument (a list)
  `( apply max '(3 7 2 9) )      => 9`
- Primitive function, apply invokes the actual function.
- Non-primitive function (f),
  - » Retrieves the referencing environment in which the function's lambda expression was originally evaluated and adds the names of the function's parameters (the list) (call this resulting environment (e))
  - » Retrieves the list of expressions that make up the body of f.
  - » Passes the body's expression together with e one at a time to eval. Finally, apply returns what the eval of the last expression in the body of f returned.

Apply

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
           (procedure-body procedure)
           (extend-environment
             (procedure-parameters procedure)
             arguments
             (procedure-environment procedure))))
        (else
         (error
          "Unknown procedure type - APPLY" procedure))))
```

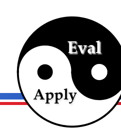

Example: Evaluating `( cadr p )`

- (define cadr (lambda (x) (car (cdr x))))
- Stored Internally as three element list C: (E (x) (car (cdr (x))))
  - – surrounding referencing environment (global)
  - – list of parameters (x)
  - – list of body expressions (one element: (car (cdr x)))
- Suppose: p is defined to be a list: (define p '(a b))
  - » (cadr p) => b
- Evaluating `( cadr p )` scheme interpreter executes:
  - » (eval '(cadr p) (scheme-report-environment 5))
    - – Note: assumes p is defined in scheme-report-environment 5
  1. Evaluate the car of it's car of the first argument,
     - » cadr via a recursive call returns function c to which cadr is bound, represented internally as a three element list C.
  2. Eval calls itself recursively on 'p' returning (a, b)
  3. Execute (apply c '(a b)) and return results

Example: Evaluating `( cadr p )`

- (define cadr (lambda (x) (car (cdr x))))
- Suppose: p is defined to be a list: (define p '(a b))
- Evaluating (cadr p) scheme interpreter executes:
  1. (eval '(cadr p) (scheme-report-environment 5))
     - – Note: assumes p is defined in scheme-report-environment 5
  2. Evaluate the car of it's car of the first argument,
     - » cadr via a recursive call returns function c to which cadr is bound, represented internally as a three element list C.
  3. Eval calls itself recursively on 'p' returning (a, b)
  4. Execute (apply c '(a b)) and return results
  5. Apply then notice the internal list representation cadr, C.
     (E (x) (car (cdr (x)))) and then apply would execute:
  6. (eval '(car (cdr (x))) (cons (cons 'x '(a b)) E)) and return the results

Summary of Scheme



- The core of a Scheme evaluator is `eval` and `apply`, procedures that are defined in terms of each other.
  - » The `eval` procedure takes an expression and an environment and evaluates to the value of the expression in the environment;
  - » The `apply` procedure takes a procedure and its operands and evaluates to the value of applying the procedure to its operands.

Evaluation Order

- Scheme uses **applicative order evaluation** (as most imperative languages, sometimes called eager or aggressive evaluation)
  - » Evaluate function arguments *before* passing them to functions

Example

- (define double (lambda (x) (+ x x)))
- Eager evaluation of (double (* 3 4))
  - ⇒ (double 12)
  - ⇒ (+ 12 12)
  - ⇒ 24

Evaluation Order

- Scheme uses **applicative order evaluation** (as most imperative languages, sometimes called eager or aggressive evaluation)
  - » Evaluate function arguments *before* passing them to functions
- We can change the evaluator to evaluate applications "*lazily*" instead, by only evaluating the value of an operand *when it is needed* (also called **normal order evaluation, call by need**).
  - » Miranda & Haskell evaluates lazily by default, call-by-name in imperative languages is a form of lazy evaluation.

Lazy Evaluation

- Don't evaluate expressions until their value is really needed.
  - » We *might* save work this way…
  - » We might change the meaning of some expressions, since the order of evaluation matters

Check: Is being Lazy any Good?

- (define double (lambda (x) (+ x x)))
- Eager evaluation of (double (* 3 4))
  - ⇒ (double 12)
  - ⇒ (+ 12 12)
  - ⇒ 24
- Lazy evaluation (double (* 3 4)) – delays computations
  - ⇒ (+ (* 3 4) (* 3 4))
  - ⇒ (+ 12 (* 3 4))
  - ⇒ (+ 12 12)
  - ⇒ 24
- QED (Quod Erat Demonstrandum): Proof that lazy is bad!
  - ⇒ Causes us to evaluate (* 3 4) twice!

Is *lazy* ever good!

```
( define switch ( lambda ( x a b c )
    ( cond
        ( ( < x 0 ) a )
        ( ( = x 0 ) b )
        ( ( > x 0 ) c ) ) ) )
```

Eager evaluation of (switch -1 (+ 1 2) (+2 3) (+ 3 4))

```
( switch -1 ( + 1 2 ) ( + 2 3 ) ( + 3 4 ) )
    ⇒  ( switch -1 3 ( + 2 3 ) ( + 3 4 ) )
    ⇒  ( switch -1 3 5  ( + 3 4 ) )
    ⇒  ( switch -1 3 5 7 )
    ⇒  ( cond
        ( ( < -1 0 ) 3 )
        ( ( = -1 0 ) 5 )
        ( ( > -1 0 ) 7 ) )
       ( cond ( #t 3 )
        ( ( = -1 0 ) 5 )
        ( ( > -1 0 ) 7 ) )
    ⇒  3
```

Is *lazy* ever good!

```
( define switch ( lambda ( x a b c )
    ( cond
        ( ( < x 0 ) a )
        ( ( = x 0 ) b )
        ( ( > x 0 ) c ) ) ) )
```

Lazy evaluation of (switch -1 (+ 1 2) (+2 3) (+ 3 4))

```
( switch -1 ( + 1 2 ) ( + 2 3 ) ( + 3 4 ) )
  ⇒  ( cond
          ( ( < -1 0 ) ( + 1 2 ) )
          ( ( = -1 0 ) ( + 2 3 ) )
          ( ( > -1 0 ) ( + 3 4 ) ) )
  ⇒  (    ( #t ( + 1 2 ) )
          ( ( = -1 0 ) ( + 2 3 )  )
          ( ( > -1 0 ) ( + 3 4 )  ) )
  ⇒  ( + 1 2 )
  ⇒  3
```

Lazy evaluation avoids evaluating both (+2 3) and (+ 3 4)

Yippie!

lazy is good!

```
( switch -1 ( + 1 2 ) ( + 2 3 ) ( + 3 4 ) )
  ⇒  ( switch -1 3 ( + 2 3 ) ( + 3 4 ) )
  ⇒  ( switch -1 3 5  ( + 3 4 ) )
  ⇒  ( switch -1 3 5 7 )
  ⇒  ( cond
     ( ( < -1 0 ) 3 )
     ( ( = -1 0 ) 5 )
     ( ( > -1 0 ) 7 ) )
  ⇒  ( cond ( #t 3 )
     ( ( = -1 0 ) 5 )
     ( ( > -1 0 ) 7 ) )
  ⇒  3
```


```
( switch -1 ( + 1 2 ) ( + 2 3 ) ( + 3 4 ) )
  ⇒  ( cond
          ( ( < -1 0 ) ( + 1 2 ) )
          ( ( = -1 0 ) ( + 2 3 ) )
          ( ( > -1 0 ) ( + 3 4 ) ) )
  ⇒  (    ( #t ( + 1 2 ) )
          ( ( = -1 0 ) ( + 2 3 )  )
          ( ( > -1 0 ) ( + 3 4 )  ) )
  ⇒  ( + 1 2 )
  ⇒  3
```


Check Scheme

- Secret is out: Scheme does use lazy evaluation for cond
  - » and special forms (aka macros)
- Functions use eager evaluation for functions defined with *lambda*

Evaluation Order

- We can also change the evaluator to evaluate applications "*lazily*" instead, by only evaluating the value of an operand ***when it is needed*** (also called normal order evaluation, call by need).
  - » In Scheme these can be done with the operator "delay".

Evaluation Order?

- First Review: What does Scheme return below?

```
(define ( try a a-expression )
(if (= a 0) 1 a-expression))
( define y 4 )
( define x 0 )

 (  try y ( / 1  y ) )    ; inverse
(  try x ( / 1  x ) )
```



```
; try with 2 arguments
(define ( try a a-expression )  ;  ( try a ( a-expression ) )   => evaluates
  (if (= a 0) 1 a-expression))  ;  inner expression first : problem if a = 0 even with if test.

( define y 4 )                  ;  ( try y ( / 1 y ) )
( define x 0 )                  ;  ( try x ( / 1 x ) )


; impact evaluation order by using lazy evaluation 'delay' in scheme
(define (delay-inverse x) (delay (/ 1 x ) ) )   ;  (try x (delay-inverse 0))
(define (aggressive-inverse x)  ( / 1 x ) )     ;  (try x (aggressive-inverse 0 ) )


( define double ( lambda (x) ( + x x ) ) )
```

- Lazy by default :
  - » Miranda & Haskell
- Lazy by demand:
  - » Scheme - using delay
  - » Ocaml – lazy
- The LAZY Advantage:
  - » http://en.wikipedia.org/wiki/Lazy_evaluation

Evaluation of Argument
Summary

- Applicative Order ("eager evaluation")
  - » Evaluate all subexpressions before apply
  - » The standard Scheme rule, Java
- Normal Order ("lazy evaluation")
  - » Evaluate arguments just before the value is needed
  - » Algol60 (sort of), Haskell, Miranda

"Normal" Scheme order is *not* "Normal Order"!

Strict and Non-Strict Languages

- A *strict* language requires all arguments to be well-defined, so applicative (eager) order can be used
- A *non-strict* language does not require all arguments to be well-defined; it requires normal-order (lazy) evaluation

Comparing Functional and Imperative Languages

- Imperative Languages:
  - » Efficient execution
  - » Complex semantics
  - » Complex syntax
  - » Concurrency is programmer designed
- Functional Languages:
  - » Simple semantics
  - » Simple syntax
  - » Inefficient execution
  - » Programs can automatically be made concurrent

Functional Programming in Perspective (pros)

- Advantages of functional languages
  - » lack of side effects makes programs easier to understand
  - » lack of explicit evaluation order (in some languages) offers possibility of parallel evaluation (e.g. MultiLisp)
  - » lack of side effects and explicit evaluation order simplifies some things for a compiler (provided you don't blow it in other ways)
  - » programs are often surprisingly short
  - » language can be extremely small and yet powerful

Functional Programming in Perspective (cons)

- Advantages of functional languages
  - » difficult (but not impossible!) to implement efficiently on von Neumann machines
    - – lots of copying of data through parameters
    - – (apparent) need to create a whole new array in order to change one element
    - – heavy use of pointers (space/time and locality problem)
    - – frequent procedure calls
    - – heavy space use for recursion
    - – requires garbage collection
    - – requires a different mode of thinking by the programmer
    - – difficult to integrate I/O into purely functional model