CS 565 August 23, 1996
Jens Palsberg Readings: EOPL, Chapter 4

Lecture Note 3:

Reasoning about Programs

The lecture has two parts:

1. Imperative programming in Scheme.
2. Reduction strategies: applicative-order and leftmost reduction.

Sequencing

(begin exp_1 ... exp_n)

is handy if one of more of exp_1 ... exp_n have side effects. It is a special form where exp_1 ... exp_n are evaluated in order, and the value of exp_n is the value of the entire `begin` expression.

```
(define for-each
  (lambda (proc lst)
    (if (null? lst)
        'done
        (begin
          (proc (car lst))
          (for-each proc (cdr lst))))))
```

The bodies of `lambda`, `let`, `letrec`, and the consequents of `cond` are implicitly contained in a `begin` expression.

```
(define displayln
   (lambda lst
     (for-each display lst)
     (newline)))
```


Example: factorial

```
> (define fact
     (lambda (n)
       (displayln "Entering fact with n = " n)
       (let ((ans (if (zero? n)
                      1
                      (* n (fact (- n 1))))))
         (displayln "Returning from fact with " ans)
 ans)))
> (fact 3)
Entering fact with n = 3
Entering fact with n = 2
Entering fact with n = 1
Entering fact with n = 0
Returning from fact with 1
Returning from fact with 1
Returning from fact with 2
Returning from fact with 6
6
```


Data Structure Mutation

The standard procedures `set-car!` and `set-cdr!` change the car and cdr fields of pairs!

```
> (define lst (list 1 2))
> (set-car! lst 3)
> lst
(3 2)
> (set-cdr! lst (list 4 5))
> lst
(3 4 5)
> (set-cdr! lst 6)
> lst
(3 . 6)
```

Data Structure Mutation

Here is a procedure `reverse!` which destroys the original list.

```
> (define reverse!
    (letrec ((loop
               (lambda (last ls)
                 (let ((next (cdr ls)))
                   (set-cdr! ls last)
                   (if (null? next)
                       ls
                       (loop ls next))))))
      (lambda (ls)
        (if (null? ls)
            ls
            (loop '() ls)))))
> (define ls (list 1 2 3))
> (reverse! ls)
(3 2 1)
> ls
(1)
```


Variable Assignment

Think of variables as being bound to *locations*.

Then a variable reference refer to the value contained in such a location.

The special form

`(set!` *var exp*`)`

evaluates the expression *exp*, places its value in the location that is the binding of variable *var*, and returns an unspecified value.

```
> (define x 1)
> (set! x 2)
> x
2
> (let ((y 3))
    (set! y 4)
    (+ y 1))
5
```


Sharing of Bindings: A Stack

```
> (define empty? '*)
> (define push! '*)
> (define pop! '*)
> (define top '*)
> (let ((stk '()))
    (set! empty?
      (lambda ()
        (null? stk)))
    (set! push!
      (lambda (x)
        (set! stk (cons x stk))))
    (set! pop!
      (lambda ()
        (if (empty?)
            (error "Stack empty")
            (set! stk (cdr stk)))))
    (set! top
      (lambda ()
        (if (empty?)
            (error "Stack empty")
            (car stk)))))

> (push! 1)
> (push! 2)
> (top)
2
> (pop!)
> (top)
1
> (pop!)
```

The four procedures `empty?`, `push!`, `pop!`, and `top` share the same `stk` binding.

Thus, the stack is a data abstraction.

Modifying a binding local to a procedure: Another stack

```
(define stack
  (let ((stk '()))
    (lambda (message)
      (case message
        ((empty?) (lambda ()
                    (null? stk)))
        ((push!) (lambda (x)
                     (set! stk (cons x stk))))
        ((pop!) (lambda ()
                   (if (null? stk)
                       (error "Stack empty")
                       (set! stk (cdr stk)))))
        ((top) (lambda ()
                  (if (null? stk)
                      (error "Stack empty")
                      (car stk))))
        (else (error "stack: does not understand : " message))))))

> ((stack 'push!) 1)
> ((stack 'push!) 2)
> ((stack 'top))
2
> ((stack 'pop!))
> ((stack 'top))
1
> ((stack 'pop!))
```

Note that the `lambda` is *inside* the `let`.

This `stack` is much like an object in object-oriented programming.

Towards classes and objects

The following procedure generate stacks.

```
(define make-stack
  (lambda ()
    (let ((stk '()))
      (lambda (message)
        (case message
          ((empty?) (lambda ()
                      (null? stk)))
          ((push!) (lambda (x)
                      (set! stk (cons x stk))))
          ((pop!) (lambda ()
                    (if (null? stk)
                        (error "Stack empty")
                        (set! stk (cdr stk)))))
          ((top) (lambda ()
                   (if (null? stk)
                       (error "Stack empty")
                       (car stk))))
          (else (error "stack: does not understand : " message))))))
```

```
  > (define s1 (make-stack))
  > (define s2 (make-stack))
  > ((s1 'push!) 1)
  > ((s2 'push!) 2)
  > ((s1 'top))
  1
  > ((s2 'top))
  2
```

Each time `make-stack` is invoked, a new stack procedure, with its own `stk` binding, is returned. This gives a simple form of classes, as known from object-oriented programming.

Implementing recursion by assignments

Here is how we can transform programs with `letrec` into program without `letrec`, but with `set!`.

(letrec ((var_1 exp_1) ...(var_n exp_n))
    body)

$\Rightarrow$

(let ((var_1 '*) ...(var_n '*))
    (set! var_1 exp_1)
    ...
    (set! var_n exp_n)
    body)

The let expression first introduces variable bindings that initially contain some irrelevant value (here: `'*`). The expressions $exp_1 \ldots exp_n$ are then evaluated *in the scope* of these bindings, and their values are assigned to the bindings. Finally, *body* is evaluated in the scope of the bindings.

Note that this implementation fixes the order in which $exp_1 \ldots exp_n$ are evaluated.

Reasoning about procedures

How do we determine the result of a procedure call?

```
  > (define foo
      (lambda (x y) (+ (* x 3) y)))
  > (foo (+ 4 1) 7)
  22
```

Let's calculate:

```
     (foo
       (+ 4 1) 7)

=>   ((lambda (x y) (+ (* x 3) y))
       (+ 4 1) 7)

=>   (+ (* (+ 4 1) 3) 7)

=>   22
```

We could also calculate like this

```
     (foo
       (+ 4 1) 7)

=>   (foo
       5 7)

=>   ((lambda (x y) (+ (* x 3) y))
       5 7)

=>   (+ (* 5 3) 7)

=>   22
```


A subset of Scheme

Let us look at a tiny subset of Scheme.

Syntax:

$\langle$exp$\rangle$ ::= $\langle$varref$\rangle$
    | (lambda ($\langle$var$\rangle$) $\langle$exp$\rangle$)
    | ($\langle$exp$\rangle$ $\langle$exp$\rangle$)

The only values that can be computed are *procedures*. No numbers, no booleans, only procedures!

The only computation that can take place is *procedure application*. No plus, no if, only procedure application!

Procedure application

Now consider an application:

(*rator* *rand*)

To actually perform a procedure application, we of course have to evaluate the operator (*rator*) to a procedure. This gives the following situation:

((lambda (*var*) *body*) *rand*)

The basic rule for this situation is:

The operand (*rand*) is evaluated to a value *before* the procedure call takes place.

Example:

```
    (((lambda (a) a) (lambda (x) x))
     ((lambda (y) y) (lambda (z) z)))

=>  ((lambda (x) x)
     ((lambda (y) y) (lambda (z) z)))

=>  ((lambda (x) x) (lambda (z) z))

=>  (lambda (z) z)
```

This is called *applicative-order* reduction.

The slogan is: "always evaluate the arguments first".

This is the evaluation principle for Scheme and ML.

An Alternative Rule

Consider again the situation:

((lambda (*var*) *body*) *rand*)

Let us now change the Semantics completely!

Here is a new evaluation principle:

The procedure call takes place without evaluating the operand *rand* at all.

Example:

```
    (((lambda (a) a) (lambda (x) x))
     ((lambda (y) y) (lambda (z) z)))

=>  ((lambda (x) x)
     ((lambda (y) y) (lambda (z) z)))

=> ((lambda (y) y) (lambda (z) z))

=>  (lambda (z) z)
```

This is called *leftmost* reduction.

The slogan is: "don't work if you can avoid it; be lazy!".

This is the evaluation principle for Miranda and Haskell.

Does it make a Difference?

Now we have one language with two different evaluation principles. That is: two semantics!

What if we take a program and first run it with applicative-order reduction (just run it in Scheme) and then run it again but this time with leftmost reduction (have to build a new system for this case ...).

Is there at least one program where the two runs will give different results?

Answer: the following is true of all programs:

- If the run with applicative-order reduction terminates, then the run with leftmost reduction terminates. (But the converse is in general false).
- If both runs terminate, then they give the same result.

A case where applicative-order reduction is too eager

Sometimes applicative-order reduction fails to terminate, even though leftmost reduction terminates.

Notice that:

```
  (define delta (lambda (x) (x x)))

    (delta delta)

=>  (delta delta)

=>  (delta delta)

=>  ...
```

Consider the program:

```
  ((lambda (y) 3) (delta delta))
```

Applicative-order reduction fails to terminate because it continues forever with trying to evaluate the operand.

Leftmost reduction terminates, simply because it throws away the argument:

```
    ((lambda (y) 3) (delta delta))

=>  3
```

Applicative-order reduction is often the most efficient

Consider this leftmost reduction:

```
    ((lambda (x) (x (x y)))
     ((lambda (w) w) z))

=>  (((lambda (w) w) z) (((lambda (w) w) z) y)))

=>  (z (((lambda (w) w) z) y)))

=>  (z (z y))
```

Let us then run the program with applicative order reduction:

```
    ((lambda (x) (x (x y)))
     ((lambda (w) w) z))

=>  ((lambda (x) (x (x y)))
     z)

=>  (z (z y))
```

The second computation is one step shorter!

Why? Because in the first computation, there were taken two copies of the argument before the procedure application `((lambda (w) w) z)` took place. That gave an extra step.

Redex

Notice that both applicative-order and leftmost reduction are *top-down*.

Both look for procedure applications that are *outside* `lambda`, that is, at the top-most level of the program.

They choose different applications, of course.

What is it they look for?

They look for patterns of the form:

`( (lambda (`*var*`)` *body* `)` *rand* `)`

Such a pattern is called a *redex*.

We can generalize applicative-order reduction and leftmost reduction by allowing the next redex chosen to be *any* redex, also one in the body of a `lambda`.

The process of doing a "procedure call" for some redex is called *beta-conversion*.

The idea is:

`( (lambda (`*var*`)` *body* `)` *rand* `)`

$\Rightarrow$ *body*`[`*rand*`/`*var*`]`

Here, *body*`[`*rand*`/`*var*`]` intuitively means "the result is like *body* except that all occurrences of *var* is replaced by *rand*".

When we allow arbitrary beta-version to take place, then the tiny subset of Scheme is called the *lambda calculus*.

Name clashes

Care must be taken to avoid name clashes.

Example:

```
  ((lambda (x)
     (lambda (y) (x y)))
   (y w))
```

should *not* be transformed into

```
   (lambda (y) ((y w) y))
```

The reference to `y` in `(y w)` should remain free!

The solution is to change the name of the inner variable name `y` to some name, say `z`, that does not occur free in the argument `y w`.

```
  ((lambda (x)
     (lambda (z) (x z)))
   (y w))
```

Now we get:

```
   (lambda (z) ((y w) z))
```


Beta-conversion

We need to define what $E[M/\mathtt{x}]$ means. We do this inductively on the form of E.

Variables

There are two subcases:

$\mathtt{x}[M/\mathtt{x}] = M$ Yes, substitute!

$\mathtt{y}[M/\mathtt{x}] = \mathtt{y}$ where $\mathtt{x} \neq \mathtt{y}$ No substitution here!

Applications

Do the substitution on both operator and operand:

$(F\ G)[M/\mathtt{x}] = (F[M/\mathtt{x}]\ G[M/\mathtt{x}])$
Just propagate the work!

Procedures

There are two subcases:

`(lambda (x)` E`)`$[M/\mathtt{x}]$ = `(lambda (x)` E`)`
No further work needed!

`(lambda (y)` E`)`$[M/\mathtt{x}]$ = `(lambda (z)` $(E[\mathtt{z}/\mathtt{y}])[M/\mathtt{x}]$ `)`
where $\mathtt{x} \neq \mathtt{y}$ and `z` does not occur free in E or M.

The second subcase creates a lot of new variables. We can save work by observing that if `x` does not occur free in E, then there is nothing to do. Moreover, if `y` does not occur free in M, then we need not change from `y` to `z`.