

6.001 Notes: Section 17.1

Slide 17.1.1

Over the past few lectures, we have been looking at evaluation, especially how to implement `eval` and `apply` in a language such as Scheme, in order to define a language. What we have seen is that by creating, or specifying, `eval` and its associated procedures, we actually define the semantics of the language: what it means to determine an expression's value, or to associate meanings with expressions in the language. We have also separated the semantics of `eval` and `apply` from the syntax of the language, meaning: how we choose to write an expression can be interfaced through a data abstraction into the deduction of meaning associated with the expression.

In this lecture, we are going to look at variations on Scheme. We are going to explore the idea of how, by making changes, in many cases very small changes, to `eval` and `apply`, we can cause the language to behave in a very different fashion. We are going to look at how we gain some benefits (sometimes with a little cost) by making those changes, trading off design issues in the language for performance issues in actually using the language.

Normal Order (Lazy) Evaluation



- 1/17

Normal Order (Lazy) Evaluation

Alternative models for computation:

- Applicative Order:
 - evaluate all arguments, then apply operator



- 2/17

Slide 17.1.2

To set the stage for what we are about to do, let's remind ourselves of what normal Scheme does. Remember that we said Scheme is an applicative order language. That means that when evaluating a combination, we first evaluate all the arguments, reducing them to actual values, including the first argument, which in our syntax is the operator. We then **apply** that operator (the procedure associated with that first argument) to the values of all the other arguments. Thus, we **apply** the procedure to the values.

Slide 17.1.3

Now, while we have been accepting that as our method of operation in Scheme, in fact it is a design choice. This was a choice made by the creators of Scheme. There is at least one other way of designing the system, called **normal order evaluation**.

In normal order evaluation, we do the following. When given a compound expression, we apply the operator (the value of the first subexpression) but to **unevaluated** argument subexpressions. Said another way, when given a compound expression, we can evaluate the first subexpression, but then can simply take the other pieces and substitute them into the

Normal Order (Lazy) Evaluation

Alternative models for computation:

- Applicative Order:
 - evaluate all arguments, then apply operator
- Normal Order:
 - go ahead and apply operator with unevaluated argument subexpressions
 - evaluate a subexpression only when value is *needed*
 - to print
 - by primitive procedure (that is, primitive procedures are "strict" in their arguments)



- 3/17

body of the procedure, and continue this process until we actually need the value of one of those subexpressions. In fact, we will evaluate a subexpression only when the value is needed in order to print something out, or because a primitive procedure is going to be applied to it. Thus, primitive procedures will be strict in requiring that their arguments are actual values.

Thus, normal order evaluation would have a very different substitution model than applicative order evaluation. In essence, we would take the subexpressions, substitute them into the body of the procedure, and keep doing that substitution until we have an expression that involves only primitive procedures and their application.

Our goal is to understand how that model leads to different evolution of the language, and how might we create such a language.

Applicative Order Example

```
(define (foo x)
  (write-line "inside foo")
  (+ x x))

(foo (begin (write-line "eval arg") 222))
```



- 4/17

Slide 17.1.4

To visualize the difference between these two kinds of evaluation, let's look at an example. Here is an example using applicative order (the normal kind of Scheme). We define `foo` to be a procedure as shown. What happens if we call `foo` with the sequence of expressions shown as argument?

Slide 17.1.5

In standard Scheme, the first thing we do is evaluate each of the subexpressions. We get the value of `foo`, which is a procedure object. We also evaluate the argument to `foo` at this stage.

Evaluating this `begin` statement leads to the following behavior.

Applicative Order Example

```
(define (foo x)
  (write-line "inside foo")
  (+ x x))

(foo (begin (write-line "eval arg") 222))

=> (begin (write-line "eval arg") 222)
```



- 5/17

Applicative Order Example

```
(define (foo x)
  (write-line "inside foo")
  (+ x x))

(foo (begin (write-line "eval arg") 222))

=> (begin (write-line "eval arg") 222)
```

```
eval arg
```



- 6/17

Slide 17.1.6

Evaluating the argument says we must evaluate this `begin` expression, and therefore we evaluate the first subexpression, which writes out on the screen: `eval arg`.

Slide 17.1.7

We then evaluate the second subexpression in the `begin` expression. Since this is the last subexpression in this `begin`, that is the value returned as the value of the argument to `foo`.

Applicative Order Example

```
(define (foo x)
  (write-line "inside foo")
  (+ x x))

(foo (begin (write-line "eval arg") 222))

=> (begin (write-line "eval arg") 222)
=> 222

eval arg
```

- 7/17

Applicative Order Example

```
(define (foo x)
  (write-line "inside foo")
  (+ x x))

(foo (begin (write-line "eval arg") 222))

=> (begin (write-line "eval arg") 222)
=> 222
=> (begin (write-line "inside foo")
         (+ 222 222))

eval arg
```

- 8/17

Slide 17.1.8

So now we apply the procedure named `foo` to that argument, `222`. By the substitution model, that reduces to evaluating the `begin` statement shown on the side in blue. We could, of course, use the environment model here, but the substitution model is sufficient for our purposes.

Slide 17.1.9

Of course, evaluating this `begin` expression's subexpressions in order means we first write out: `inside foo`; then we evaluate the actual addition, and return the value `444`.

Applicative Order Example

```
(define (foo x)
  (write-line "inside foo")
  (+ x x))

(foo (begin (write-line "eval arg") 222))

=> (begin (write-line "eval arg") 222)
=> 222
=> (begin (write-line "inside foo")
         (+ 222 222))

eval arg
inside foo
=> 444
```

- 9/17

Applicative Order Example

```
(define (foo x)
  (write-line "inside foo")
  (+ x x))

(foo (begin (write-line "eval arg") 222))

=> (begin (write-line "eval arg") 222)
=> 222
=> (begin (write-line "inside foo")
         (+ 222 222))

eval arg
inside foo
=> 444
```

We first evaluated argument, then substituted value into the body of the procedure

- 10/17

Slide 17.1.10

So let's summarize what we did. We first evaluated the argument, then substituted that value into the body of the procedure. This led to the observed behavior, as written out, that we evaluate the argument once, then proceeded inside the procedure, then we returned the value. Keep that in mind as we now go to the alternative model.

Slide 17.1.11

Now, let's think about the alternative model. This is a normal order model, in which we are going to first evaluate the procedure subexpression to get the procedure, but then the arguments we are going to substitute into the body of that procedure, and keep unwinding the evaluations until we get down to things that either involve printing or application of primitive procedures. Same definition of `foo`, same call, but is there a different behavior?

Normal Order Example

```
(define (foo x)
  (write-line "inside foo")
  (+ x x))

(foo (begin (write-line "eval arg") 222))
```



- 11/17

Normal Order Example

```
(define (foo x)
  (write-line "inside foo")
  (+ x x))

(foo (begin (write-line "eval arg") 222))

=> (begin (write-line "inside foo")
          (+ (begin (w-l "eval arg") 222)
             (begin (w-l "eval arg") 222))))
```



- 12/17

Slide 17.1.12

In this case, we get the value associated with `foo`, but we substitute the entire argument into the body of `foo` without evaluation. This leads to the evaluation of the `begin` expression shown on the slide. Notice that no evaluation of the argument has taken place yet, just substitution of tree structure. Thus, nothing has been printed out on the screen yet.

Slide 17.1.13

In this case, we evaluate the subexpressions to the top-level `begin` in order. Thus we first evaluate the line that says we are inside `foo`, which gets printed out as shown.

Normal Order Example

```
(define (foo x)
  (write-line "inside foo")
  (+ x x))

(foo (begin (write-line "eval arg") 222))

=> (begin (write-line "inside foo")
          (+ (begin (w-l "eval arg") 222)
             (begin (w-l "eval arg") 222))))

inside foo
```



- 13/17

Normal Order Example

```
(define (foo x)
  (write-line "inside foo")
  (+ x x))

(foo (begin (write-line "eval arg") 222))

=> (begin (write-line "inside foo")
          (+ (begin (w-l "eval arg") 222)
             (begin (w-l "eval arg") 222))))

inside foo
eval arg
```



- 14/17

Slide 17.1.14

Then, we turn to the next subexpression inside the top-level `begin`. Here we evaluate the first subexpression of this combination, namely `+`. This has as a value a primitive procedure. So in this case, we need to actually evaluate the argument expressions; there is no further substitution possible. Thus we evaluate the first of the two interior `begin` statements, which writes out: `eval arg`; then gets the value of `222` and returns it to `+`.

Slide 17.1.15

Then, do the same thing for the second interior `begin` expression, since `+` is a primitive procedure and requires its arguments be evaluated.


Normal Order Example

```
(define (foo x)
  (write-line "inside foo")
  (+ x x))

(foo (begin (write-line "eval arg") 222))

=> (begin (write-line "inside foo")
          (+ (begin (w-l "eval arg") 222)
             (begin (w-l "eval arg") 222))))
```

```
inside foo
eval arg
eval arg
```



- 15/17


Normal Order Example

```
(define (foo x)
  (write-line "inside foo")
  (+ x x))

(foo (begin (write-line "eval arg") 222))

=> (begin (write-line "inside foo")
          (+ (begin (w-l "eval arg") 222)
             (begin (w-l "eval arg") 222))))
```

```
inside foo
eval arg
eval arg
=> 444
```



- 16/17

Slide 17.1.16

Then actually apply the primitive procedure to the arguments, and return the value.

Slide 17.1.17

So now we see there **is** a difference in behavior. Remember in the applicative order case, we noted that we first evaluated the argument (once) and then we went inside the procedure.

Here, it's as if we substituted the unevaluated arguments into the body of the procedure. Thus we first went inside the procedure, then when required by a primitive procedure, we evaluate the argument. And since it has been substituted twice into the body, we evaluate it twice.

Our goal then is to note that normal order has a different behavior: substitute until required to evaluate. Applicative was: get value, then substitute. So why is this a useful change to make? And how do we change our evaluator to achieve normal order evaluation.

Normal Order Example


```
(define (foo x)
  (write-line "inside foo")
  (+ x x))

(foo (begin (write-line "eval arg") 222))

=> (begin (write-line "inside foo")
          (+ (begin (w-l "eval arg") 222)
             (begin (w-l "eval arg") 222))))
```

```
inside foo
eval arg
eval arg
=> 444
```

As if we substituted the
unevaluated expression in the
body of the procedure



- 17/17

Slide 17.2.1

Let's deal with the second issue first. What changes should we make to `eval` and `apply` in order to implement this new idea of normal order evaluation (sometimes also called **lazy** evaluation). Note that it is called **lazy** evaluation because we are only evaluating arguments when required, either because we need to print them out or because we are down to primitive procedures. So what changes are needed?

How can we implement lazy evaluation?

```
(define (l-apply procedure arguments env) ; changed
  (cond ((primitive-procedure? procedure)
        (apply-primitive-procedure
         procedure
         (list-of-arg-values arguments env)))
        ((compound-procedure? procedure)
         (l-eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           (list-of-delayed-args arguments env)
           (procedure-environment procedure))))
        (else (error "Unknown proc" procedure))))
```



- 1/16

How can we implement lazy evaluation?

```
(define (l-apply procedure arguments env) ; changed
  (cond ((primitive-procedure? procedure)
        (apply-primitive-procedure
         procedure
         (list-of-arg-values arguments env)))
        ((compound-procedure? procedure)
         (l-eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           (list-of-delayed-args arguments env)
           (procedure-environment procedure))))
        (else (error "Unknown proc" procedure))))
```



- 2/16

Slide 17.2.2

Let's start with the application of a compound procedure, something we have built with a `lambda`. We know we want to evaluate the body of that procedure in a new environment, but what should the environment do? `Extend-environment` should take the set of parameters of the procedure, but glue together with them, not with the actual values, but with a set of delayed arguments. And what is a **delayed argument**? Let's create a structure that let's us hold off on getting the actual value of an argument. That makes sense if you think of our example. When we want to apply a procedure, we want to take the argument expressions and substitute them

into the body without evaluation, but as unevaluated tree structure.

So our change in applying a compound procedure is to still evaluate the body of the procedure in a new environment, but rather than binding the procedure's parameters to their actual values, we will bind them to these special structures that keep track of the expression to be evaluated when required, plus we will have to keep track of the environment in which that evaluation should take place, when required.

Slide 17.2.3

That last point is worth stressing, because it is a change. Prior to this, `apply` didn't need to know about it's environment.

We simply applied a procedure to a set of argument **values**.

Here, because we delay getting the argument values, we need to keep track of the information needed to actually compute the values when needed, meaning the environment in which the expression should be evaluated. Thus we need to pass the environment in as an argument to `apply` so that we can pass it through to the construction of the delayed evaluation objects.

How can we implement lazy evaluation?

```
(define (l-apply procedure arguments env) ; changed
  (cond ((primitive-procedure? procedure)
        (apply-primitive-procedure
         procedure
         (list-of-arg-values arguments env)))
        ((compound-procedure? procedure)
         (l-eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           (list-of-delayed-args arguments env)
           (procedure-environment procedure))))
        (else (error "Unknown proc" procedure))))
```



- 3/16

How can we implement lazy evaluation?

```
(define (l-apply procedure arguments env) ; changed
  (cond ((primitive-procedure? procedure)
        (apply-primitive-procedure
         procedure
         (list-of-arg-values arguments env)))
        ((compound-procedure? procedure)
         (l-eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           (list-of-delayed-args arguments env)
           (procedure-environment procedure))))
        (else (error "Unknown proc" procedure))))
```



- 4/16

Slide 17.2.4

So with this change, compound procedure application will correctly delay the evaluation of the arguments. We will implement `list-of-delayed-arguments` shortly. Conceptually, we know that it should simply take a list of expressions, and glue onto these expressions a marker that identifies them as delayed.

Ultimately, however, we will reduce to a primitive application, and in that case, we want to do the actual work of evaluating the arguments. Thus, our other change will be to require when applying a primitive procedure, we need to ensure that the arguments are reduced to actual values. This means we will force any delayed evaluation to now take place. Here again we

will need to pass the environment into the procedure `list-of-arg-values` because we are going to have to walk our way down this list of expressions, asking each to be evaluated with respect to the environment.

Slide 17.2.5

Notice that we have made only a small number of changes:

`apply` now takes an environment as an argument;

application of a compound procedure constructs a set of delayed expressions; and application of a primitive procedure constructs a set of actual values by evaluating any delayed argument. We have to implement the procedures to delay and force evaluation, but otherwise we have a very small set of changes.

How can we implement lazy evaluation?

```
(define (l-apply procedure arguments env) ; changed
  (cond ((primitive-procedure? procedure)
        (apply-primitive-procedure
         procedure
         (list-of-arg-values arguments env)))
        ((compound-procedure? procedure)
         (l-eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           (list-of-delayed-args arguments env)
           (procedure-environment procedure))))
        (else (error "Unknown proc" procedure))))
```



- 5/16

Lazy Evaluation – l-eval

- Most of the work is in `l-apply`; need to call it with:
 - actual value for the operator
 - just expressions for the operands
 - the environment...

```
(define (l-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ...
        ((application? exp)
         (l-apply (actual-value (operator exp) env)
                  (operands exp)
                  env))
        (else (error "Unknown expression" exp))))
```



- 6/16

Slide 17.2.6

What else do we have to change? Since we have changed `apply` (which we here call `l-apply` for lazy apply), we need to change the use of that procedure in `eval` (which we here call `l-eval`). But in fact most of the work is now in `l-apply`. Here we call it with the actual value of the operator, but **just** the expressions for the other arguments, plus the environment.

So our change to `l-eval` is very simple. When we get to an application, we pass the actual value of the operator (which means we need to be sure to do the evaluation) to `l-apply`.

But notice that we just pass in the other expressions as tree structure, with no evaluation.

Notice that this is the **only** change to `l-eval`. All other expressions are still handled exactly as before.

Also notice that when `apply` takes in these arguments, it may further pass the delayed arguments along unevaluated (if the application involves another compound expression). Only when `apply` reaches a primitive application will it force the arguments to be evaluated.

Slide 17.2.7

What else do we need? Notice that we have now made a distinction between getting the actual value of expressions, versus simply delaying the evaluation of an expression. We need values for operators, and for primitive applications only. But this means we now have two different kinds of values: the actual value of the expression, versus a promise to get the value when asked for it. We need to implement those two kinds of values, which we do next.

Actual vs. Delayed Values

- 7/16

Actual vs. Delayed Values

```
(define (actual-value exp env)
  (force-it (l-eval exp env)))
```



- 8/16

Slide 17.2.8

First, to get the actual value of an expression with respect to an environment, we would expect to see a use of `l-eval`. We can see that inside of this procedure. But notice that evaluating that expression might itself return a delayed object. If it is a nested combination of procedures, for instance, evaluating the first level may give us back something that is still a delayed promise to get a value. So we will add one more piece. We will add a procedure called `force-it`, which takes an argument and ensures that it is fully evaluated, and not delayed. Thus, `actual-value` of an expressions will evaluate any delayed expression, and ensure that the returned value is

not delayed, but fully evaluated.

Slide 17.2.9

As we saw, we will use `actual-value` inside `l-eval` to get the value of the procedure. We do that because we need to know whether the procedure is primitive or compound, in order to keep unwrapping the substitution of unevaluated arguments into the bodies of procedures. We also need to use `actual-value` when a primitive application requires an actual set of values. Thus, `list-of-arg-values` should take a set of expressions and an environment, and should recursively evaluate all of the expressions in that environment. Note how we just construct a list using recursive evaluation, but here using `actual-value` to ensure that evaluation of the expression completely unwinds any delayed objects.

Actual vs. Delayed Values

```
(define (actual-value exp env)
  (force-it (l-eval exp env)))

(define (list-of-arg-values exps env)
  (if (no-operands? exps) '()
      (cons (actual-value (first-operand exps) env)
            (list-of-arg-values (rest-operands exps)
                               env))))
```



- 9/16

Actual vs. Delayed Values

```

(define (actual-value exp env)
  (force-it (l-eval exp env)))

(define (list-of-arg-values exps env)
  (if (no-operands? exps) '()
      (cons (actual-value (first-operand exps) env)
            (list-of-arg-values (rest-operands exps)
                               env))))

(define (list-of-delayed-args exps env)
  (if (no-operands? exps) '()
      (cons (delay-it (first-operand exps) env)
            (list-of-delayed-args (rest-operands exps)
                                  env))))

```



- 10/16

Slide 17.2.10

On the other hand, if we are applying a compound procedure to a set of arguments, we want to just take the argument expressions and substitute them directly into the body. But we need to label them as something that is a delayed evaluation. `List-of-delayed-args` does exactly the opposite of `list-of-arg-values`. It walks down the list of expressions and glues a label onto each expression, creating a promise for each expression to do the evaluation when required, where the promise consists of the tree structure of the expression and the environment in which to ultimately do the evaluation, assembled together in some data structure.

Slide 17.2.11

Notice, these are the **only** changes we need to make to `eval` and `apply` to implement lazy evaluation, so let's recap what we have done. `Apply` of a primitive procedure gets the actual values of the arguments by forcing delayed expressions. `Apply` of a compound procedure delays the evaluation of all other arguments. `Eval` simply evaluates the first subexpression of an application to get the procedure, but passes the rest of the arguments along as delayed, unevaluated objects. These will get forced into evaluation when a primitive application takes places.

All that is left is to implement the data abstraction for delayed objects and the operation of forcing such objects.

Actual vs. Delayed Values

```

(define (actual-value exp env)
  (force-it (l-eval exp env)))

(define (list-of-arg-values exps env)
  (if (no-operands? exps) '()
      (cons (actual-value (first-operand exps) env)
            (list-of-arg-values (rest-operands exps)
                               env))))

(define (list-of-delayed-args exps env)
  (if (no-operands? exps) '()
      (cons (delay-it (first-operand exps) env)
            (list-of-delayed-args (rest-operands exps)
                                  env))))

```



- 11/16

Representing Thunks

- *Abstractly* – a `thunk` is a "promise" to return a value when later needed ("forced")



- 12/16

Slide 17.2.12

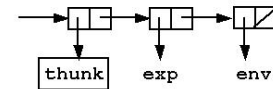
For historical reasons, we call one of these delayed things, one of these promises to do an evaluation, a **thunk**. Abstractly, a **thunk** is simply a promise to return a value when it is needed, i.e. when it is **forced**. Thus, inside a thunk, we need to represent the actual expression as tree structure, the environment that will serve as the context for interpreting symbols within the expression, and a label identifying this thing as a thunk.

Slide 17.2.13

To actually implement this, we can just build this as a "tagged" structure: a list with the label of a **thunk**, plus a pointer to the tree structure of the expression, plus a pointer to the environment.

Representing Thunks

- *Abstractly* – a **thunk** is a "promise" to return a value when later needed ("forced")
- *Concretely* – our representation:



- 13/16

Thunks – delay-it and force-it

```
(define (delay-it exp env) (list 'thunk exp env))
(define (thunk? obj) (tagged-list? obj 'thunk))
(define (thunk-exp thunk) (cadr thunk))
(define (thunk-env thunk) (caddr thunk))
```



- 14/16

Slide 17.2.14

With that in mind, we can now implement this idea. **Delay** just constructs this tagged list. Remember that since **exp** is just passed in as list structure from the parser, it is just glued together into new list structure, without evaluation. As with any data abstraction, we have a predicate for identifying a thunk, and selectors for getting out the parts of the thunk, the expression and the environment in which to evaluate the expression.

Slide 17.2.15

The corresponding piece is that when given a thunk, we can force it to be evaluated. Here, we can be careful. We will first check to see if the object is a delayed thing. If it is not a thunk, then we know that this is already reduced to a value, and we simply pass it along. This is exactly why we put the label on the object in the first place, allowing us to distinguish between tree structure that is part of the data structure being manipulated and tree structure that represents something waiting to be evaluated. On the other hand, if this is a thunk, then we extract the parts of the thunk (expression and environment), and then we do the actual evaluation by applying `actual-value` (which we know will force the evaluation of this expression). Notice as a consequence that if `obj` on first evaluation reduces to another delayed thing, `actual-value` will continue to force it, until a value is returned not a delayed object.

Thunks – delay-it and force-it

```
(define (delay-it exp env) (list 'thunk exp env))
(define (thunk? obj) (tagged-list? obj 'thunk))
(define (thunk-exp thunk) (cadr thunk))
(define (thunk-env thunk) (caddr thunk))
```

```
(define (force-it obj)
  (cond ((thunk? obj)
        (actual-value (thunk-exp obj)
                      (thunk-env obj)))
        (else obj)))
```



- 15/16

Thunks – delay-it and force-it

```

(define (delay-it exp env) (list 'thunk exp env))
(define (thunk? obj) (tagged-list? obj 'thunk))
(define (thunk-exp thunk) (cadr thunk))
(define (thunk-env thunk) (caddr thunk))

(define (force-it obj)
  (cond ((thunk? obj)
        (actual-value (thunk-exp obj)
                      (thunk-env obj)))
        (else obj)))

```



- 16/16

Slide 17.2.16

So there are the changes. Very simple changes to our evaluator have allowed us to build something that does lazy evaluation. We made a small change to `apply`, we made a small change to `eval`, we introduced one new kind of thing, a delayed object or thunk, but that's it. Thus we have a dramatic change in behavior based on a small change in the evaluator.

6.001 Notes: Section 17.3

Slide 17.3.1

So what have we accomplished to this point? We have converted our standard evaluator, our applicative order evaluator, into a normal order evaluator. And to do that we had to make only a small number of changes in the evaluator itself. However, we did need to introduce a couple of new things. The primary one was this idea of a delayed object: the idea of taking an expression, plus the environment in which it is to be evaluated, and sticking them together into a promise, a thunk, that says: "I am not going to evaluate myself now, but when you ask me for my value, I will give it to you". We use that to delay evaluation of any of the arguments to procedure applications, until we get down to things that need to be printed or to applications of primitive procedures.

Memo-izing evaluation

- 1/11

Memo-izing evaluation

- In lazy evaluation, if we reuse an argument, have to reevaluate each time
- In normal evaluation, argument is evaluated once, and just referenced
- Can we keep track of values once we've obtained them, and avoid cost of reevaluation?



- 2/11

Slide 17.3.2

One consequence of using these delayed objects as part of our lazy evaluation is that we end up doing, at least at present, some extra work. If you go back to our earlier example, when we compared normal order and applicative order using the `foo` procedure, you will see that in lazy evaluation, if we used the same argument multiple places inside a procedure body, we had to re-evaluate it each time. If this is an expensive computation, we could be wasting a lot of effort.

On the other hand, in applicative order evaluation, we evaluated the argument once, and then simply used it, or referenced it as part of the environment, inside the body of the procedure. So

can we trade this off? Is there some way of keeping track of values once we have forced a delayed object, in order to avoid the cost of re-evaluation?

Slide 17.3.3

The answer is **yes**. The basic idea is that we will **memoize** a thunk. This means that we will keep track, using a memo if you like, of when we have actually done the work. Thus, we start off with a thunk, a delayed expression or a promise to be evaluated when asked to. When we actually force that thunk, when we do the work to get the thunk's value, we will simply remember it. We will keep track of that by putting a tag on it that indicates that we have the value associated with a thunk. Thus if the value is ever needed again, we can just return that value, rather than recomputing it.

Memo-izing Thunks

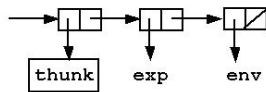
- *Idea*: once thunk `exp` has been evaluated, remember it
- If value is needed again, just return it rather than recompute



-3/11

Memo-izing Thunks

- *Idea*: once thunk `exp` has been evaluated, remember it
- If value is needed again, just return it rather than recompute



-4/11

Slide 17.3.4

Remember how we have been representing thunks as a tagged list with a label, an expression stored as tree structure and an environment to use for context when evaluating the expression. Given this representation for a thunk, what do we want to have happen after we have done the evaluation?

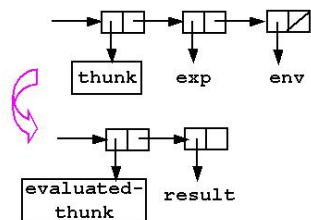
Slide 17.3.5

We simply want to take the result, the value that comes from that evaluation, and keep track of it, together with a label that indicates I have already done the work. So concretely, we can simply **mutate** the current thunk into an evaluated thunk, by putting that tag on the front together with the result.

Memo-izing Thunks

- *Idea*: once thunk `exp` has been evaluated, remember it
- If value is needed again, just return it rather than recompute

- *Concretely* – mutate a thunk into an evaluated-thunk

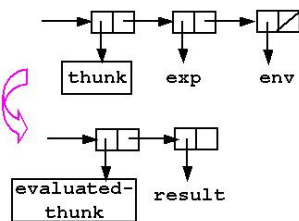


-5/11

Memo-izing Thunks

- *Idea*: once thunk `exp` has been evaluated, remember it
- If value is needed again, just return it rather than recompute

- *Concretely* – mutate a thunk into an evaluated-thunk



-6/11

Slide 17.3.6

Notice that we used an important word in describing this. We said we would **mutate** the thunk. That means we are going to take this list structure and actually change its contents. Why? Why not simply create a new object with the same information? The reason is that if some other part of the evaluation is pointing to this thunk, by mutating the thunk those things still point to the same object. This way I don't need to keep track of who needs this value, I have simply changed this value directly without having to change anything else.

Slide 17.3.7

How about implementing that idea? That is straightforward. We take a `thunk` and add some aspects to it. We now have a `thunk` that can be evaluated, so we need a predicate to check that. And we will have a way of getting the value out of an evaluated `thunk`, by extracting a part of the tagged list structure. This goes hand-in-hand with the parts we had earlier for normal `thunks`.

Thunks – Memoizing Implementation

```
(define (evaluated-thunk? obj)
  (tagged-list? obj 'evaluated-thunk))
(define (thunk-value evaluated-thunk)
  (cadr evaluated-thunk))
```



- 7/11

Thunks – Memoizing Implementation

```
(define (evaluated-thunk? obj)
  (tagged-list? obj 'evaluated-thunk))
(define (thunk-value evaluated-thunk)
  (cadr evaluated-thunk))

(define (force-it obj)
  (cond ((thunk? obj)
        (let ((result (actual-value (thunk-exp obj)
                                   (thunk-env obj))))
          (set-car! obj 'evaluated-thunk)
          (set-car! (cdr obj) result)
          (set-cdr! (cdr obj) '())
          result))
        ((evaluated-thunk? obj) (thunk-value obj))
        (else obj))))
```



- 8/11

Slide 17.3.8

You might expect that we should also have a constructor for an evaluated `thunk`. It seems like we are building a data abstraction, and one would normally expect a constructor to go with the selector. But remember what we said: we are going to mutate an existing structure, not create a new one. That means we are really just modifying the components of a `thunk` directly. Thus we won't have a specific constructor for an evaluated `thunk`, rather we will rely on changing what happens to a `thunk` when we force it.

Slide 17.3.9

Remember what `force-it` did. Given an object, it would first check to see if that object was a `thunk`. If it were, it would get the actual value of the expression part of the `thunk` with respect to the environment part of the `thunk`. If it were not a `thunk`, it would just return the object as the desired object. Now we need to change things slightly.

If this object is a `thunk`, i.e. is appropriately tagged as a `thunk`, we will again do the work to get the associated value of the `thunk` expression with respect to the environment, using `actual-value`. Now we will mutate the existing object.

We will change the first part of the object from the label `thunk` to the label `evaluated-thunk`. We will change the next component of the object to hold the result, replacing what used to be the expression. Finally, we will mutate the last part of the object to be the empty list, that is, we will drop the pointer to the environment because we no longer need it. Thus we have mutated a list of length 3 into a list of length 2, which now has the structure of an evaluated `thunk`.

What else do we need? If the object is itself an evaluated `thunk`, we will just return the value! We've already done the work so we can just return it. And if the object is neither kind of `thunk`, we will just return the object directly as the value. This gives us our memoized version of a `thunk`. This object only does the work once to evaluate a delayed object, keeping track of the computed value for future reference.

Thunks – Memoizing Implementation

```
(define (evaluated-thunk? obj)
  (tagged-list? obj 'evaluated-thunk))
(define (thunk-value evaluated-thunk)
  (cadr evaluated-thunk))

(define (force-it obj)
  (cond ((thunk? obj)
        (let ((result (actual-value (thunk-exp obj)
                                   (thunk-env obj))))
          (set-car! obj 'evaluated-thunk)
          (set-car! (cdr obj) result)
          (set-cdr! (cdr obj) '())
          result))
        ((evaluated-thunk? obj) (thunk-value obj))
        (else obj))))
```



- 9/11

Lazy Evaluation – other changes needed

- Example – need actual predicate value in conditional if...

```
(define (l-eval-if exp env)
  (if (true? (actual-value (if-predicate exp) env))
      (l-eval (if-consequent exp) env)
      (l-eval (if-alternative exp) env)))
```



- 10/11

Slide 17.3.10

Are any other changes needed to convert our evaluator into a lazy evaluator? Just a few. What have we had to change so far? Our primary change was in `apply`, which now gets actual values for arguments if using a primitive procedure, otherwise it delays the evaluation of the argument expressions. In terms of `eval` the change was in doing applications, in which we simply passed down the argument subexpressions without evaluation. The other changes were ways of getting the values of arguments when needed.

The other change that we need is in one of our special forms, in `if`. Think about what happens in an `if` expression. In order

to decide which branch to take, I need to get the actual value of the predicate. Thus within an `if`, I will get the actual value of the predicate, but the other subexpressions I can pass along as thunks, relying on the primitive applications to force these when needed.

Slide 17.3.11

But other special forms are fine. For example, in assignment the value associated with a variable can be evaluated in a lazy fashion. Until some procedure actually needs this new value for this variable, the value can be stored as a thunk, a promise to compute the value when needed.

This completes the conversion of our evaluator from applicative order to normal order or lazy evaluation.

Lazy Evaluation – other changes needed

- Example – need actual predicate value in conditional if...

```
(define (l-eval-if exp env)
  (if (true? (actual-value (if-predicate exp) env))
      (l-eval (if-consequent exp) env)
      (l-eval (if-alternative exp) env)))
```
- Example – don't need actual value in assignment...

```
(define (l-eval-assignment exp env)
  (set-variable-value!
   (assignment-variable exp)
   (l-eval (assignment-value exp) env)
   env)
  'ok)
```



- 11/11

6.001 Notes: Section 17.4**Slide 17.4.1**

One of the goals of this lecture was to show how we can change the behavior of a language by changing the evaluator, especially how small changes in the evaluator can have dramatic consequences on the evaluation of expressions in the language. In this lecture, we converted from applicative order to normal order evaluation as an example.

But what are the tradeoffs inherent in such a switch? It turns out that with lazy evaluation we have an interesting dilemma. On the positive side, we only do work when we need an actual value. This suggests that in principle we could devise very efficient methods for computing things. On the negative side, we are not always certain when an expression will be evaluated.

If we have a language with side effects, in which mutation takes place, this can be a serious issue. It can lead to conceptual errors as well as programming errors, and can cause some serious difficulties. It may also be the case

Laziness and Language Design

- We have a dilemma with lazy evaluation
 - Advantage: only do work when value actually needed
 - Disadvantages
 - not sure when expression will be evaluated; can be very big issue in a language with side effects
 - may evaluate same expression more than once



- 1/18

that we evaluate the same expression more than once. Since we delay the evaluation, we don't know when it is going to be evaluated, we may end up redoing work.

Laziness and Language Design

- We have a dilemma with lazy evaluation
 - Advantage: only do work when value actually needed
 - Disadvantages
 - not sure when expression will be evaluated; can be very big issue in a language with side effects
 - may evaluate same expression more than once
- Memoization doesn't fully resolve our dilemma
 - Advantage: Evaluate expression at most once
 - Disadvantage: What if we *want* evaluation on each use?



- 2/18

Slide 17.4.2

Some of this we can fix with memoization. It allows us to evaluate an expression at most once and resolves this issue of doing extra work. But it has the disadvantage that we don't have control over it. What if we want evaluation on each use, what if the evaluation of an expression involves mutation in which the side effect is important, and it is not the value that is returned but the effect that takes place that matters. In that case, we want the expression evaluated each time, not memoized. As we have built things so far, we don't have that ability.

Slide 17.4.3

This makes it sound like lazy evaluation is not a good idea. And of course that is not the case. We really have a trade off here. One way to approach this is to observe that if I am not going to have any side effects in my language, then lazy evaluation is a valuable tool. With memoization in place I can have efficient implementations by only evaluating expressions as needed and at most once.

But if I want to have side effects or mutations, then I have a disadvantage. I don't know how to control when I get evaluation to take place.

An alternative is to give the **programmer** direct control, letting him/her specify whether an expression should be a thunk, or a direct value. So how do we build this level of control into an evaluator?

Laziness and Language Design

- We have a dilemma with lazy evaluation
 - Advantage: only do work when value actually needed
 - Disadvantages
 - not sure when expression will be evaluated; can be very big issue in a language with side effects
 - may evaluate same expression more than once
- Memoization doesn't fully resolve our dilemma
 - Advantage: Evaluate expression at most once
 - Disadvantage: What if we *want* evaluation on each use?
- Alternative approach: **give programmer control!**



- 3/18

Variable Declarations: lazy and lazy-memo

- Handle lazy and lazy-memo extensions in an upward-compatible fashion.;
- ```
(lambda (a (b lazy) c (d lazy-memo)) ...)
```
- "a", "c" are normal variables (evaluated before procedure application)
  - "b" is lazy; it gets (re)-evaluated each time its value is actually needed
  - "d" is lazy-memo; it gets evaluated the first time its value is needed, and then that value is returned again any other time it is needed again.



- 4/18

### Slide 17.4.4

So let's change our evaluator to let the programmer directly specify, when they create a procedure, whether a variable should be treated as a normal variable, as a lazy variable, or as a lazy memo variable. Here is the form we will use.

We will now let `lambda`s have as their parameter lists, either variables, or expressions that indicate both a variable name and the kind of variable it should be. In the example shown, we want `a` and `c` to be treated by the evaluator as normal variables. When we apply the procedure, the arguments associated with these variables should be evaluated before we do the procedure application.

`b` is specified to be a lazy variable. That says that when an expression is passed in for this parameter, it should get re-evaluated each time it is needed, but not evaluated until it is needed in a primitive application.

On the other hand, `d` should be a memoized lazy variable. That means that an expression passed in as part of an application should be delayed until needed as part of a primitive application, but once evaluated, the value is saved and reused on each subsequent evaluation.

This will allow us to distinguish between things that we want evaluated right at application time, things that we

want to delay evaluating but re-evaluate each time we need them, and things that we want to delay evaluating but only evaluate once.

### Slide 17.4.5

Thus we are now giving the programmer the ability to put declarations on the variables or parameters of a procedure. This suggests that one of the changes we will need to make to our evaluator is to change the syntax, at least of a `lambda` expression.

#### Syntax Extensions – Parameter Declarations



- 6/18

#### Syntax Extensions – Parameter Declarations

```
(define (first-variable var-decls) (car var-decls))
(define (rest-variables var-decls) (cdr var-decls))
(define declaration? pair?)

(define (parameter-name var-decl)
 (if (pair? var-decl) (car var-decl) var-decl))

(define (lazy? var-decl)
 (and (pair? var-decl) (eq? 'lazy (cadr var-decl))))

(define (memo? var-decl)
 (and (pair? var-decl)
 (eq? 'lazy-memo (cadr var-decl))))
```



- 6/18

### Slide 17.4.6

So remember what happens in our lazy evaluator. When we go to apply a compound procedure, we evaluate the body of that procedure in an extended environment. That extended environment was created by taking the list of parameters of the procedure, plus a list of delayed arguments, and glued them together. Now we have to think a little more carefully about what it means to look at the list of parameters.

In the full version of a lazy evaluator, `list-of-delayed-arguments`, the thing that created the arguments, would simply delay everything in the list. Here, we want to walk down the parameter list at the same time we are

walking down the arguments, and decide for each parameter whether to evaluate or delay. So here is some structure to do this.

We will need a selector for the next variable in the list of parameters, and a selector for the rest of the variables. We will need a way of checking for each variable whether it is just a symbol (meaning I want direct evaluation of the expression) or whether it has been **declared** to be a different type of variable. Thus `declaration?` will tell me whether the variable is a special case or not.

To get the name of the variable, we will check: if the object is a list, then it is a declaration and I need to get the second element of the list. If it is not a list, it must just be the symbol I need. For variables that are declared, I can build tag-checking procedures to determine the type, as shown.



### Slide 17.4.7

So this will allow us to walk down the parameter list associated with a procedure, checking each element in turn to find the variable name, and any declaration that specifies how we want to evaluate the value to be bound to that variable.

#### Syntax Extensions – Parameter Declarations

```
(define (first-variable var-decls) (car var-decls))
(define (rest-variables var-decls) (cdr var-decls))
(define declaration? pair?)

(define (parameter-name var-decl)
 (if (pair? var-decl) (car var-decl) var-decl))

(define (lazy? var-decl)
 (and (pair? var-decl) (eq? 'lazy (cadr var-decl))))

(define (memo? var-decl)
 (and (pair? var-decl)
 (eq? 'lazy-memo (cadr var-decl))))
```



#### Controllably Memo-izing Thunks

- `thunk` – never gets memoized
- `thunk-memo` – first eval is remembered
- `evaluated-thunk` – memoized-thunk that has already been evaluated



### Slide 17.4.8

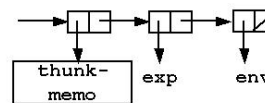
In terms of the values to be associated with variables, we now have several different flavors. We have standard values (numbers, symbols, etc.). We also have thunks, which we treat as a delayed expression that gets evaluated each time it is needed.

### Slide 17.4.9

We will also have a memoized thunk, and this is a different declaration. Here the idea is that when we first evaluate this thunk, we are going to remember its value, and we remember it in a particular structure ...

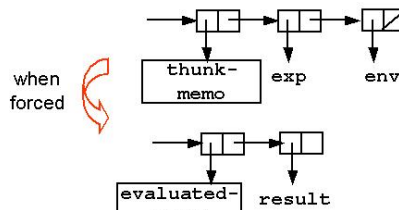
#### Controllably Memo-izing Thunks

- `thunk` – never gets memoized
- `thunk-memo` – first eval is remembered
- `evaluated-thunk` – memoized-thunk that has already been evaluated



#### Controllably Memo-izing Thunks

- `thunk` – never gets memoized
- `thunk-memo` – first eval is remembered
- `evaluated-thunk` – memoized-thunk that has already been evaluated



### Slide 17.4.10

As we did earlier, we will mutate a `thunk-memo` structure into an `evaluated-thunk`, keeping track of the result. So when we actually force the evaluation of one of these `thunk-memos`, we will mutate it into this new structure. Notice what we are doing: we are making a distinction between a `thunk` and a memoized `thunk`. This allows the programmer to specify whether a `thunk` should always remain as a delayed promise, to be re-evaluated each time the value is needed; or whether it should keep track of its value after the first evaluation, and just return that value when requested.

requested.

### Slide 17.4.11

Remember that in our full lazy evaluator, the key issue arose when we went to apply a compound procedure. There we glued together the parameters of the procedure with a list of delayed arguments. In that case, we delayed each argument, storing that promise in the environment, and proceeding to the evaluation of the body of the procedure. Because we are now going to let the programmer have control, we have to re-examine our mechanism for delaying arguments. We need to make a small change here.

#### A new version of delay-it

- Look at the variable declaration to do the right thing...

```
(define (delay-it decl exp env)
 (cond ((not (declaration? decl))
 (l-eval exp env))
 ((lazy? decl)
 (list 'thunk exp env))
 ((memo? decl)
 (list 'thunk-memo exp env))
 (else (error "unknown declaration:" decl))))
```



- 11/78

#### A new version of delay-it

- Look at the variable declaration to do the right thing...

```
(define (delay-it decl exp env)
 (cond ((not (declaration? decl))
 (l-eval exp env))
 ((lazy? decl)
 (list 'thunk exp env))
 ((memo? decl)
 (list 'thunk-memo exp env))
 (else (error "unknown declaration:" decl))))
```



- 12/78

### Slide 17.4.12

Previously, `delay-it` would simply have taken in an expression and an environment and converted the pair into a `thunk`. Now, we need to pass into `delay-it` not on the expression to be delayed, but also the declaration (the part of the parameter list that corresponds to this expression). If that parameter is **not** a declaration, i.e. it is just a normal variable, then we go ahead and get the value, right now, as we would in normal Scheme.

If, however, the programmer has specified this variable to be lazy, we will create a `thunk`, or one of these delayed evaluation objects, exactly as we did in the lazy evaluator.

If the programmer has specified this variable to be memoized, we will similarly delay that evaluation, but with a distinctive label to separate it from a normal `thunk`.

Thus, we are using the programmer's declarations to tell us how to handle evaluation of expressions to be bound to parameters during a procedure application: do we evaluate now or delay, and if we delay do we want to keep track of the value once we get it.

### Slide 17.4.13

Similarly, because we now have different kinds of expressions associated with variables, I need to change `force-it`, the procedure intended in the previous version to force the evaluation of any delayed object.

Now, we need to handle the wider range of possible expressions. First, if the object to be forced into evaluation has been labeled as a `thunk`, a delayed promise, then I get the expression and the environment parts of the object, and force the evaluation of the expression with respect to this environment. I return that value to the caller, but notice that I don't remember it. If some other caller asks for the value of this expression, I will redo the work of forcing its evaluation.

If the object has been labeled as a `memoized-thunk`, I will similarly get the expression and environment parts of the object, and force the evaluation of the expression with respect to the environment. But in this case, I will mutate this structure into an `evaluated-thunk`, keeping track of this computed value. Remember

#### Change to force-it

```
(define (force-it obj)
 (cond ((thunk? obj) ;eval, but don't remember it
 (actual-value (thunk-exp obj)
 (thunk-env obj)))
 ((memoized-thunk? obj) ;eval and remember
 (let ((result
 (actual-value (thunk-exp obj)
 (thunk-env obj))))
 (set-car! obj 'evaluated-thunk)
 (set-car! (cdr obj) result)
 (set-cdr! (cdr obj) '())
 result))
 ((evaluated-thunk? obj) (thunk-value obj))
 (else obj)))
```



- 13/78

that I am mutating in place, so that any variable that points to this object will continue to point to it, so that it can now access the computed value, without having to force the work.

If the object was a thunk but has already been evaluated, I can just extract and return the value. Everything else just returns the value of the object.

#### Changes to l-apply

- Key: in l-apply, only delay "lazy" or "lazy-memo" params
  - make thunks for "lazy" parameters
  - make memoized-thunks for "lazy-memo" parameters



• 14/18

#### Slide 17.4.14

With that infrastructure in place, I can now make the key change. In this version of `l-apply`, I only want to delay parameters that have been explicitly labeled as lazy or memoized lazy. For lazy parameters, we should make a thunk. For memoized lazy parameters, we should make a memoized thunk. Those are both handled by our new version of `delay-it`. And then when I go to apply a compound procedure, I need to do the following ....

#### Slide 17.4.15

Applying a compound procedure means that I want to evaluate the body of that procedure with respect to a new environment.

That environment I am going to get by extending the procedure's environment with a new frame. But in that frame, notice what we do. First, we get out the parameters of the procedure. However, I know that this list is not just a list of names, it may include declarations. So to extend the environment, I need to get just the names, which I do by mapping the selector down this list. Then, from the list of parameters I create a set of delayed arguments, as before. This means I take any declarations and the associated expressions and environment, and use `delay-it` to create the right expressions. If there is no declaration, I evaluate the expression and bind it to the variable. If there is a declaration, I create an appropriate delayed expression and bind it to the variable.

#### Changes to l-apply

- Key: in l-apply, only delay "lazy" or "lazy-memo" params
  - make thunks for "lazy" parameters
  - make memoized-thunks for "lazy-memo" parameters

```
(define (l-apply procedure arguments env)
 (cond ((primitive-procedure? procedure)
 ...) ; as before; apply on list-of-arg-values
 ((compound-procedure? procedure)
 (l-eval-sequence
 (procedure-body procedure)
 (let ((params (procedure-parameters procedure)))
 (extend-environment
 (map parameter-name params)
 (list-of-delayed-args params arguments env)
 (procedure-environment procedure))))))
 (else (error "Unknown proc" procedure))))
```



• 15/18

#### Deciding when to evaluate an argument...

- Process each variable declaration together with application subexpressions – delay as necessary:

```
(define (list-of-delayed-args var-decls exps env)
 (if (no-operands? exps)
 '()
 (cons (delay-it (first-variable var-decls)
 (first-operand exps)
 env)
 (list-of-delayed-args
 (rest-variables var-decls)
 (rest-operands exps)
 env))))
```



• 16/18

#### Slide 17.4.16

Now I can use this entire infrastructure to create the actual list of delayed arguments. I take the declarations, the list of variables and their declared types, plus the list of expressions (the arguments passed in) and I walk down those two lists in unison, delaying each expression. Remember that `delay-it` will use the declaration to decide which expressions to delay, and how, and which expressions to evaluate. This now allows the programmer to control when an expression to be used as an argument in a procedure gets evaluated: at time of application, or at time of primitive application; is it evaluated and remembered, or simply re-evaluated each time it is needed.

**Slide 17.4.17**

So what does this do for us? Now we have built in some programmer control. By changing how an application deals with its arguments, in terms of when to evaluate and what to keep track of, we have enabled the programmer to specify what he/she wants. When they create a `lambda` they have the opportunity to specify their choice concerning which parameters are evaluated under which rules.

The key issue is how, given this idea of lazy evaluation, we are able to easily change the behavior of the language. Some small syntactic changes in how we specify parameters, and some changes in the manipulation of evaluation gives us a very different behavior.

**Deciding when to evaluate an argument...**

- Process each variable declaration together with application subexpressions – delay as necessary:

```
(define (list-of-delayed-args var-decls exps env)
 (if (no-operands? exps)
 '()
 (cons (delay-it (first-variable var-decls)
 (first-operand exps)
 env)
 (list-of-delayed-args
 (rest-variables var-decls)
 (rest-operands exps)
 env))))
```



• 17/18

**Summary**

- Lazy evaluation – control over evaluation models
  - Convert entire language to normal order
  - Upward compatible extension
    - lazy & lazy-memo parameter declarations



• 18/18

**Slide 17.4.18**

To summarize, here are the key points about changes in evaluation based on our choice of evaluation model. The primary issue is how we have allowed a programmer to control evaluation behavior. We can control the order in which things are evaluated, from applicative order, to normal order, to compromises in between.