

Scheme is an applicative order language and the meta-circular evaluator we built also happens to applicative order.

If the body of a procedure is entered before an argument has been evaluated we say that the procedure is **non-strict** in that argument. If the argument is evaluated before the body of the procedure is entered we say that the procedure is **strict** in that argument.

Delaying evaluation of procedure arguments until the last possible moment (e.g., until they are required by a primitive operation) is called lazy evaluation.

- ▶ In a purely applicative order language, all procedures are strict in each argument.
- ▶ In a purely normal-order language, all compound procedures are non-strict in each argument, and primitive procedures may be either strict or non-strict.

The “strict” versus “non-strict” terminology means essentially the same thing as “applicative-order” versus “normal-order,” except that it refers to individual procedures and arguments rather than to the language as a whole.

`cons` (or, in general, almost any constructor for data structures) can be useful while being non-strict.

One can do useful computation, combining elements to form data structures and operating on the resulting data structures, even if the values of the elements are not known.

Example:- We can compute the length of a list without knowing the values of the individual elements in the list.

if expression has the form

(if <predicate> <consequent> <alternative>).

What if we have knowledge that the predicate in most cases evaluates to true and the <alternative> is very resource intensive and at the same time most likely to not be evaluated.

if expression has the form

(if <predicate> <consequent> <alternative>).

What if we have knowledge that the predicate in most cases evaluates to true and the <alternative> is very resource intensive and at the same time most likely to not be evaluated.

It is also possible that the <alternative> throws an error when evaluated.

In such cases, it would be very reasonable to delay the evaluation of the <alternative>.

```
(define (try a b) (if (> a 0) a b))  
(define (fry a b c) (+ a b))
```

```
; (try 5 (/ 1 0))  
; (fry 1 2 (/ 3 0))
```

```
; This would throw an error  
; Although one could expect them to  
; produce 5 and 3 as their outputs
```

```
; Reason for the error:-  
; arguments are being evaluated  
; before entering the body
```

We will implement a normal-order language that is the same as Scheme except that compound procedures are non-strict in each argument.

Primitive procedures will still be strict.

Almost all the required changes center around procedure application.
Basic Idea:- When applying a procedure, the interpreter must determine which arguments are to be evaluated and which are to be delayed.

The delayed arguments are not evaluated and are transformed into objects called **thunks**.

The thunk must contain the information required to produce the value of the argument when it is needed, as if it had been evaluated at the time of the application.

⇒ The thunk must contain the argument expression and the environment in which the procedure application is being evaluated. The process of evaluating the expression in a thunk is called **forcing**.

In general, a thunk will be forced only when its value is needed:

1. when it is passed to a primitive procedure that will use the value of the thunk;
2. when it is the value of a predicate of a conditional; and
3. when it is the value of an operator that is about to be applied as a procedure.

One design choice we have available is whether or not to memoize thunks.

In general, a thunk will be forced only when its value is needed:

1. when it is passed to a primitive procedure that will use the value of the thunk;
2. when it is the value of a predicate of a conditional; and
3. when it is the value of an operator that is about to be applied as a procedure.

One design choice we have available is whether or not to memoize thunks.

The critical difference between what we are doing here and what we did while studying streams is that we are building delaying and forcing into the evaluator, and thus making this uniform and automatic throughout the language.

Lazy evaluation combined with memoization is sometimes referred to as call-by-need argument passing, in contrast to call-by-name argument passing.

(Call-by-name, introduced in Algol 60, is similar to non-memoized lazy evaluation.)

```
((application? exp)
 (apply (eval (operator exp) env)
        (list-of-values (operands exp) env)))
```

Old vs the portion of the `eval` procedure of the metacircular evaluator.

```
((application? exp)
 (apply (actual-value (operator exp) env)
        (operands exp)
        env))
```

`actual-value` is a procedure which forces the evaluation of an expression in its associated environment, so that if the expression is a thunk, it will be forced.

Implementing Non-strict compound procedures

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

Old vs New apply

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

The new version of `apply` is also almost the same as the first version of the metacircular evaluator.

The difference is that `eval` has passed in unevaluated operand expressions:

1. Primitive procedures (which are strict):- evaluate all the arguments before applying the primitive.
2. Compound procedures (which are non-strict):- delay all the arguments before applying the procedure.

Two new procedures are now used by `apply`.

- ▶ `list-of-delayed-args` delays the arguments instead of evaluating them, and
- ▶ `list-of-arg-values` uses `actual-value` procedure instead of `eval` procedure:

We must change the evaluator in its handling of `if`, where we must use `actual-value` instead of `eval` to get the value of the predicate expression before testing whether it is true or false:

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

Finally, we must change the REPL procedure (details in the textbook) to use `actual-value` instead of `eval`, so that if a delayed value is propagated back to the REPL, it will be forced before being printed.

Our evaluator must be able to create thunks when procedures are applied to arguments and to force these thunks later.

A thunk must package an expression together with the environment, so that the argument can be produced later.

To force the thunk, we simply extract the expression and environment from the thunk and evaluate the expression in the environment.

Our evaluator must be able to create thunks when procedures are applied to arguments and to force these thunks later.

A thunk must package an expression together with the environment, so that the argument can be produced later.

To force the thunk, we simply extract the expression and environment from the thunk and evaluate the expression in the environment.

We use **actual-value** rather than **eval** so that in case the value of the expression is itself a thunk, we will force that, and so on, until we reach something that is not a thunk:

Our evaluator must arrange

- ▶ to create thunks when procedures are applied to arguments and
- ▶ to force these thunks later when needed.

Our evaluator must arrange

- ▶ to create thunks when procedures are applied to arguments and
- ▶ to force these thunks later when needed.

One easy way to package an expression with an environment is to make a list containing the expression and the environment.

```
(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))
```

We would ideally want the thunks to be memoized.


Thus when a thunk is forced, we will turn it into an evaluated thunk by

- ▶ replacing the stored expression with its value and
- ▶ changing the *thunk* tag to *evaluated-thunk* tag so that it can be recognized as already evaluated.

While studying streams, we were introduced to special forms `delay` and `stream-cons`, which allowed us to construct a “promise” to compute the cdr of a stream, without actually fulfilling that promise until later.

A special form is not a first-class object like a procedure, so we cannot use it together with higher-order procedures.

Special forms are merely syntax and not procedures that can be used in conjunction with other higher-order procedures.

```
> (map square '(4 5 6))  
(16 25 36)  
> (map quote '(4 5 6))  
 quote: bad syntax in: quote
```

Special forms are exceptions to the general evaluation rule.

Like `(define x 3)` is a special form, because it deviates from the general evaluation rule of applying `define` to `x` and `3`.

The rules for evaluating a special form depend on the particular special function being called.

A special form is a primitive function specially marked so that its arguments are not all evaluated. Most special forms create control structures (like `if`) or perform variable bindings—things (like `define`) which functions cannot do.

Each special form has its own rules for which arguments are evaluated and which are used without evaluation. Whether a particular argument is evaluated may depend on the results of evaluating other arguments.

- ▶ In the absence of streams, we had access to the primitive procedure cons which was strict.

We were forced to create streams as a new kind of data object similar but not identical to lists, we were required to reimplement many ordinary list operations like map/filter for use with streams.

- ▶ In the absence of streams, we had access to the primitive procedure `cons` which was strict.

We were forced to create streams as a new kind of data object similar but not identical to lists, we were required to reimplement many ordinary list operations like `map/filter` for use with streams.

- ▶ With lazy evaluation for arguments, streams and lists can be identical, so there is no need for special forms or for separate list and stream operations.

All we need to do is make `cons` non-strict.

One way to accomplish this is to extend the lazy evaluator to allow for non-strict primitives.

Alternatively, an easier way is to recall that there is no fundamental need to implement `cons` as a primitive at all.

We have already seen how we can represent pairs as procedures:

```
(define (cons x y) (lambda (m) (m x y)))  
(define (car z) (z (lambda (p q) p)))  
(define (cdr z) (z (lambda (p q) q)))
```

The difference would be that we would create a thunk for x and y .

`(car z)` would also return a thunk if it is being passed on to some other procedure as input, but `car` would force the evaluation of the thunk if we were trying to print `(car z)`.

Now the lists would be lazy by default and we would not two different procedures with same functionality for lists and streams.

```
;procedure for lists
(define (list-ref items n)
  (if (= n 0)
      (car items)
      (list-ref (cdr items) (- n 1))))

;procedure for streams
(define ones (cons 1 ones))
(define integers (cons 1
                       (add-lists ones integers)))
```

Note that these lazy lists are even lazier than the streams. The car of the list, as well as the cdr, are both delayed, whereas in case of streams which we had studied earlier the car was not delayed.