

19. Introduction to evaluation order

At this point in the material we start our coverage of *evaluation order*. We start by discussing the idea of referential transparency.

19.1. Referential transparency

Lecture 5 - slide 2

The main idea behind the concept of referential transparency is captured by the point below.

Two equal expressions can substitute each other without affecting the meaning of a functional program

As formulated above, the possibility of substituting one expression by another depends on whether or not the two expressions are considered as being equal. As noticed in Section 4.5 there are a number of different interpretations of equality around in Scheme, as well as in other programming languages.

As we observe in the items below, we can use even the weakest form of equality, namely structural, deep equality, for our observations about referential transparency. In other words, if two structures are structurally equal, the expressions and values involved may substitute each other.

- Referential transparency
 - provides for easy equational reasoning about a program
 - does not rely on a particular notion of equality
 - Reference equality, shallow equality and deep equality cannot be distinguished by functional means
 - is a major contrast to imperative programming

The idea of referential transparency can be stated very briefly in the following way:

Equals can be replaced by equals

19.2. An illustration of referential transparency

Lecture 5 - slide 3

Before we proceed we will illustrate some practical uses of referential transparency.

With referential transparency it is possible to perform natural program transformations without any concern of side effects

The two expressions in the top left and the top right boxes of Figure 19.1 may substitute each other, provided that the function F is a pure function (without side effects).

In the case where F is a value returning procedure, as illustrated in the bottom box of Figure 19.1 it is clear that it is important how many times F is actually evaluated. The reason is that an evaluation of F affects the value of the variable c , which is used in the top level expressions. (Thus, F is an imperative abstraction - a procedure).

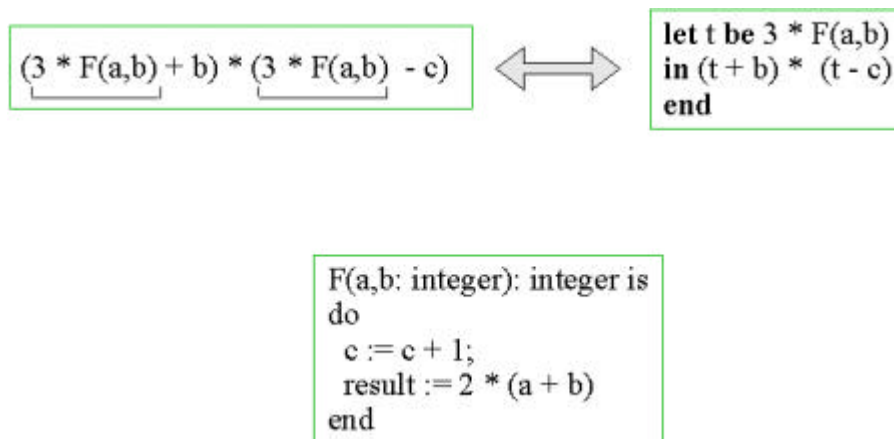


Figure 19.1 It is possible to rewrite one of the expressions above to the other, provided that F is a function. Below, we have illustrated an example where F is of procedural nature. Notice that F assigns the variable c , such that it becomes critical to know how many times F is called.

On the ground of this example it is worth observing that equational reasoning about functional programs is relatively straightforward. If procedures are involved, as F in the bottom box of Figure 19.1, it is much harder to reason about the expression, for instance with the purpose of simplifying it (as it is the case when substituting the expression to the left with the expression to the right in the top-part of Figure 19.1).

19.3. Arbitrary evaluation order - with some limits

Lecture 5 - slide 5

In this section we will discuss the order of subexpression evaluation in a composite expression.

In a functional program an expression is evaluated with the purpose of producing a value

An expression is composed of subexpressions

Take a look at one of the expressions in Figure 19.2. The underlying syntax trees are shown below the expressions in the figure. The question is which of the subexpressions to evaluate first, which comes next, and which one is to be the last.

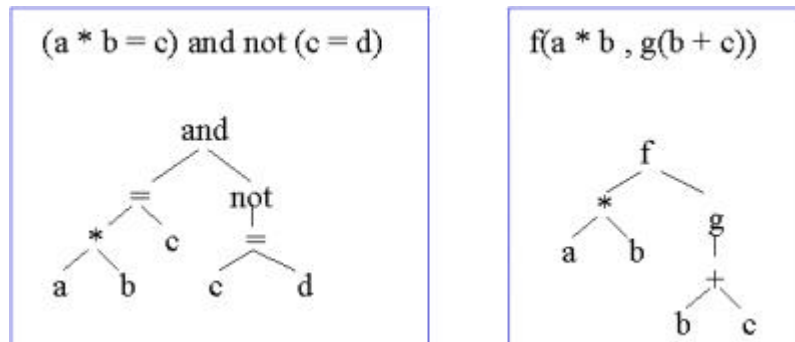


Figure 19.2 An illustration of an expression with subexpressions.

To be concrete, we can propose to start from the left leaf, from the right leaf, from the top, or perhaps from some location in the middle of the expression.

In the functional programming context, with expressions and pure functions, we will probably expect that an arbitrary evaluation order is possible. If we should devise a practical recipe we will probably start from one of leafs (say the leftmost leaf) and work our way to the expression in the root.

As noticed below, we can actually use an arbitrary evaluation order, provided that there are no errors in any of the subexpressions, and provided that all of the involved evaluations terminate.

- Subexpressions can be evaluated in an arbitrary order provided that
 - no errors occur in subexpressions
 - the evaluation of all subexpressions terminates
- It is possible, and without problems, to evaluate subexpressions in parallel

In the rest of this section, as well as in Chapter 20 we will study and understand the premises and the limits of 'arbitrary evaluation order'.

19.4. A motivating example

Lecture 5 - slide 6

It is valuable to understand the problems and the quirks of evaluation order by looking at a very simple program example.

What is the value of the following expression?

The lambda expression in Program 19.1 shows a pseudo application of the constant function, which returns 1 for every possible input x . The tricky part is, however, that we pass an actual parameter expression which never terminates.

```
((lambda (x) 1) some-infinite-calculation)
```

Program 19.1 *A constant function with an actual parameter expression, the evaluation of which never terminates.*

It is not difficult to write a concrete Scheme program, which behave in the same way as Program 19.1. Such a program is shown in Program 19.2. The parameter less function `infinite-calculation` just calls itself forever recursively.

```
(define (infinite-calculation)
  (infinite-calculation))

((lambda (x) 1) (infinite-calculation))
```

Program 19.2 *A more concrete version of the expression from above. The function `infinite-calculation` just calls itself without ever terminating.*

19.5. A motivating example - clarification

Lecture 5 - slide 7

As noticed below, it can be argued that the value of the expression in Program 19.1 is 1, due to the reasoning that the the result of the function `(lambda (x) 1)` is independent of the formal parameter x .

It can also be argued that an evaluation of the actual parameter expression `(infinite-calculation)` stalls the evaluation of the surrounding expression, such that the expression in Program 19.1 does not terminate. In Scheme, as well as in most other programming languages, this will be the outcome.

The items below summarizes these two possibilities, and they introduce two names of the two different function semantics, which are involved.

- Different evaluation orders give different 'results'
 - The number 1
 - A non-terminating calculation
- Two different semantics of function application are involved:
 - **Strict**: A function call is well-defined if and only if all actual parameters are well-defined
 - **Non-strict**: A function call can be well-defined even if one or more actual parameters cause an error or an infinite calculation

In most languages, functions are strict. This is also the case in Scheme. In some languages, however, such as Haskell and Miranda, functions are non-strict. As we will see in the following, languages with non-strict functions are very interesting, and they open up new computational possibilities.

19.6. References

- [1] Foldoc: referential transparency
<http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?query=referential+transparency>

20. Rewrite rules, reduction, and normal forms

At this point in the material it will be assumed that the reader is motivated to study evaluation order in some detail.

An evaluation of an expression can be understood as a transformation of the expressions which preserves its meaning. In this chapter we will see that transformations can be done incrementally in *rewriting steps*. A rewriting of an expression gives a new expression which is semantically equivalent to the original one. Usually, we go for rewritings which simplify an expression. In theory, however, we could also rewrite an expression to more complicated expression.

In this chapter we will formally characterize the *value of an expression*, using the concept of a normal form. We will see that the value of an expression is an expression in itself that cannot be rewritten to simpler forms by use of any rewriting rules.

As a key insight in this chapter, we will also see that an expression can be reduced to a value (a normal form) in many different ways. We will identify and name a couple of these, and we will discuss which of the evaluation strategies is the 'best'.

20.1. Rewrite rules

Lecture 5 - slide 9

This section gives an overview of the rewrite rules, we will study in the subsequent sections.

The *rewrite rules* define semantics preserving transformations of expressions

The goal of applying the rewrite rules is normally to reduce an expression to the simplest possible form, called a *normal form*.

- Overview of rewrite rules
 - **Alpha conversion:** rewrites a lambda expression
 - **Beta conversion:** rewrites general function calls
 - Expresses the idea of substitution, as described by *the substitution model*
 - **Eta conversion:** rewrites certain lambda expressions to a simpler form

The Beta conversion corresponds to the *substitution model* of function calls, which is explained in [Abelson96]. (See section of 1.1.5 of [Abelson96] for the details).

20.2. The alpha rewrite rule

Lecture 5 - slide 10

The first rewrite rule we encounter is called the alpha rewrite rule. From a practical point of view this rule is not very interesting, however. The rule tells under which circumstances it is possible to use other names of the formal parameters. Recall in this context that the formal parameter names are binding name occurrences, cf. Section 8.8.

An alpha conversion changes the names of lambda expression formal parameters

Here comes the formulation of the alpha rewrite rule.

Formal parameters of a lambda expression can be substituted by other names, which are not used as free names in the body

Recall in the context of this discussion that a free name in a construct is applied, but not bound (or defined) in the construct. See Section 8.11 for additional details about free names.

In Table 20.1 we see an example of a legal use of the alpha rewrite rule. The formal names x and y of the lambda expression are changed to a and b , respectively. It is fairly obvious that this causes no problems nor harm. The resulting lambda expression is fully equivalent with the original one.

Expression	Converted Expression
<code>(lambda (x y) (f x y))</code>	<code>(lambda (a b) (f a b))</code>

Table 20.1 *An example of an alpha rewriting. The name a replaces x and the name y replaces y .*

More interesting, we show an example of an illegal use of the alpha rewrite rule in Table 20.2. Again we change the name x to a . The name of the other formal parameter is changed to f . But f is a free name in the lambda expression. It is easy to see that the converted expression in Table 20.2 has changed its meaning. The name f is now bound in the formal parameter list. Thus, the rewriting in Table 20.2 is illegal.

Expression	Converted Expression
<code>(lambda (x y) (f x y))</code>	<code>(lambda (a f) (f a f))</code>

Table 20.2 *Examples of an illegal alpha conversion. f is a free name in the lambda expression. A free name is used, but not defined (bound) in the lambda expression. In case we rename one of the parameters to f , the free name will be bound, hereby causing an erroneous name binding.*

20.3. The beta rewrite rule

Lecture 5 - slide 11

The beta rewrite rule is the one to watch carefully, due to its central role in any evaluation process that involves the calling of functions.

A beta conversion tells how to evaluate a function call

The beta rewrite rules goes as follows.

An application of a function can be substituted by the function body, in which formal parameters are substituted by the corresponding actual parameters

It is worth noticing that there are no special conditions for the application of the beta rewrite rule. In that way the rule is different from both the alpha rewrite rule, which we studied in Section 20.2, and it is also different from the eta rewrite rule which we encounter in Section 20.4 below. All the examples in Table 20.3 are legal examples of beta rewritings.

Expression	Converted Expression
<code>((lambda(x) (f x)) a)</code>	<code>(f a)</code>
<code>((lambda(x y) (* x (+ x y))) (+ 3 4) 5)</code>	<code>(* 7 (+ 7 5))</code>
<code>((lambda(x y) (* x (+ x y))) (+ 3 4) 5)</code>	<code>(* (+ 3 4) (+ (+ 3 4) 5))</code>

Table 20.3 *Examples of beta conversions. In all the three examples the function calls are replaced by the bodies. In the bodies, the formal parameters are replaced by actual parameters.*

Be sure to understand that the beta rewrite rule tells us how to implement a function call, at least in a principled way. In a practical implementation, however, the substitution of formal parameters by (more or less evaluated) actual parameters is not efficient. Therefore, in reality, the bindings of the formal parameters are organized in name binding frames, in so-called environments. Thus, instead of name substitution (as called for in the beta rewrite rules), the formal names are looked up in a name binding environment, when they are needed in the body of the lambda expression.

The implementation of `eval` in a Scheme interpreter describes the details of a practical and real life use of the beta rewrite rule. See Section 24.3 for additional details.

20.4. The eta rewrite rule

Lecture 5 - slide 12

The eta rewrite rules transforms certain lambda expressions. As such the eta rewrite rule is similar to the alpha rewrite rule, but radically different from the beta rewrite rule.

An eta conversion lifts certain function calls out of a lambda convolute

In loose terms, the eta rewrite rule can be formulated in the following way. Be aware, however, that there is a condition associated with applications of the eta rewrite rule. The condition is described below.

A function f , which only passes its parameters on to another function e , can be substituted by e

Here is a slightly more formal - and more precise - description of the eta rewrite rule:

$$(\lambda(x) (e\ x)) \Leftrightarrow e \text{ provided that } x \text{ is not free in the expression } e$$

In the same way as above for alpha conversions in Section 20.2 we will give examples of legal and illegal uses of the eta rule.

The example in Table 20.4 shows that the lambda expression around `square` is superfluous. In the eta-rewritten expression, the lambda surround of `square` is simply discarded.

Expression	Converted Expression
<code>(lambda (x) (square x))</code>	<code>square</code>

Table 20.4 *An example of an eta rewriting.*

It is slightly more complicated to illustrate an illegal use of the rule. In the expression of the left cell in Table 20.5 we are attempting to eliminate the outer lambda expression by use of the eta rewrite rule. Notice, however, that x is free in the inner blue lambda expression. Therefore the eta rewriting illustrated in Table 20.5 is not legal. By applying the rewriting rule on the left part of Table 20.5 anyway we lose the binding of x , and therefore the rewriting does not preserve the semantics of the left cell expression.

Expression	Converted Expression
<code>(lambda (x) ((lambda (y) (f x y)) x))</code>	<code>(lambda (y) (f x y))</code>

Table 20.5 *An example of an illegal eta conversion. The eta conversion rule says in general how 'e' is lifted out of the lambda expressions. In this example, e corresponds to the emphasized inner lambda expression (which is blue on a color medium.) However, x is a free name in the inner lambda expression, and therefore the application of the eta rewrite rule is illegal.*

This completes our discussion of rewriting rules, and we will now look at the concept of normal forms.

20.5. Normal forms

Lecture 5 - slide 13

As already mentioned above, the value v of an expression e is a particular simple expression which is semantically equivalent with e . The expression v is obtained from e by a number of rewriting steps.

Normal forms represent our intuition of the value of an expression

Here is the definition of a normal form.

An expression is in *normal form* if it cannot be reduced further by use of beta and eta conversions

Notice in the definition that we talk about reduction. By this is meant application of the rewrite rules 'from left to right'.

- About normal forms
 - Alpha conversions can be used infinitely, and as such they do not play any role in the formulation of a normal form
 - A normal form is a particular simple expression, which is equivalent to the original expression, due to the application of the conversions

Normal forms are simple to understand. But there are a number of interesting and important questions that need to be addressed. One of them is formulated below.

Is a normal form always unique?

The answer to the question will be found in Section 20.9.

20.6. The ordering of reductions

Lecture 5 - slide 14

As discussed in Section 19.3 we can expect that the concrete order of evaluation steps will matter, especially in the cases where errors or infinite calculations are around in some of the subexpressions.

Evaluation steps are now understood as reductions with the beta or eta rewrite rule.

In this section we will identify and name a couple of evaluation strategies or plans. Such a strategy determines the order of use of the beta and eta reduction rules.

Given a complex expression, there are many different orderings of the applicable reductions

Using *normal-order reduction*, the first reduction to perform is the one at the outer level of the expression

Using *applicative-order reduction*, the first reduction to perform is the inner leftmost reduction

- Normal-order reduction represents *evaluation by need*
- Applicative-order reduction evaluates all constituent expressions, some of which are unnecessary or perhaps even harmful. As such, there is often a need to *control the evaluation process with special forms* that use a non-standard evaluation strategy

Let it be clear here, that many other evaluation strategies could be imagined. The practical relevance of additional strategies is another story, however.

Applicative-order reduction represents 'the usual' evaluation strategy, used for expressions in most programming languages. Normal-order reduction represents a new approach, which is used in a few contemporary functional programming languages.

In Section 20.10 we will discuss examples of the special forms mentioned in the item discussing the applicative-order reduction.

20.7. An example of normal versus applicative evaluation

Lecture 5 - slide 15

Let us illustrate the difference between normal-order reduction and applicative-order reduction via a concrete example.

Reduction of the expression `((lambda(x y) (+ (* x x) (* y y))) (fak 5) (fib 10))`

The example involves an application of the blue function `(lambda(x y) (+ (* x x) (* y y)))` on the actual parameters `(fak 5)` and `(fib 10)`. The functions `fak` and `fib` are shown in Program 20.1.

In Program 20.1 we show definitions of `fak` and `fib`, together with the example expression.

```

(define (fak n)
  (if (= n 0) 1 (* n (fak (- n 1)))))

(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1)) (fib (- n 2))))))

((lambda(x y) (+ (* x x) (* y y))) (fak 5) (fib 10))

```

Program 20.1 *The necessary Scheme stuff to evaluate the expression.*

In Figure 20.1 applicative-order reduction is outlined in the leftmost path of the graph. With applicative-order reduction we first evaluate the lambda expression, then `(fak 5)` and `(fib 10)`. The evaluation of the lambda expression gives a function object. Notice that the expensive calculations of `(fak 5)` and `(fib 10)` are only made once. The last step before the addition and the multiplications is a beta reduction, with which the function is called.

The normal order reduction is illustrated with the path to the right in Figure 20.1. The outer reduction is a beta reduction, in which we substitute the non-reduced parameter expressions `(fak 5)` and `(fib 10)`. Notice that the calculation of `(fak 5)` and `(fib 10)` are made twice.

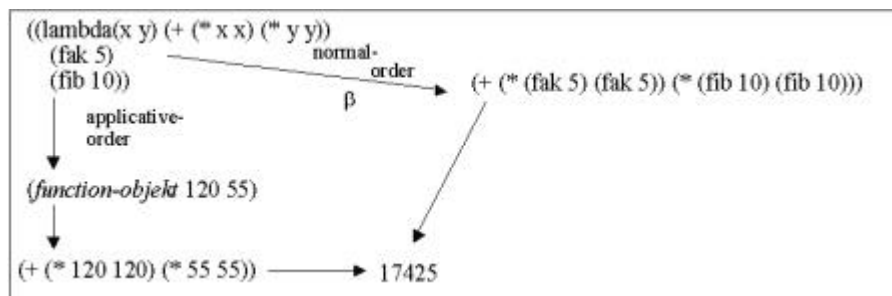


Figure 20.1 *Normal vs. applicative reduction of a Scheme expression*

As an immediate insight from the example we will emphasize the following:

It appears to be the case that normal order reduction can lead to repeated evaluation of the same subexpression

20.8. Theoretical results

Lecture 5 - slide 16

We will now cite some theoretical results of great importance to the field.

The theoretical results mentioned on this page assure some very satisfactory properties of functional programming

The results are based on a definition of *confluence*, which appears in the figure below.

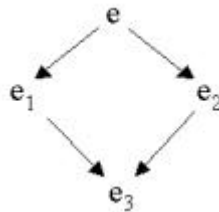


Figure 20.2 *The rewriting \Rightarrow is confluent if for all e, e_1 and e_2 , for which $e \Rightarrow e_1$ and $e \Rightarrow e_2$, there exists an e_3 such that $e_1 \Rightarrow e_3$ and $e_2 \Rightarrow e_3$*

The results which we will use below are the following:

The first Church-Rosser theorem. Rewriting with beta and eta conversions are confluent.

The second Church-Rosser theorem If $e_0 \Rightarrow \dots \Rightarrow e_1$, and if e_1 is on normal form, then there exists a normal order reduction of e_0 to e_1

The practical consequences of the results will be discussed in the following section.

20.9. Practical implications

Lecture 5 - slide 17

We will here describe the practical consequences of the theoretical results mentioned on the previous page

- During the evaluation of an expression, it will never be necessary to backtrack the evaluation process in order to reach a normal form.
- An expression cannot be converted to two different normal forms (modulo alpha conversions, of course).
- If an expression e somehow can be reduced to f in one or more steps, f can be reached by normal order reduction - but not necessarily by applicative order reduction

Because rewriting with beta and eta reduction is confluent, according to the first Church-Rosser theorem in Section 20.8, we see that there can be no dead ends in an evaluation process. Assume there is, and you will get an immediate contradiction.

The middle item is of particular importance because it guaranties that a normal form is unique. Assume that two different normal forms exist, and get a contraction with the first of the theorems.

The last result is a direct consequence of the second Church-Rosser theorem. It says more or less that normal-order reduction is the most powerful evaluation strategy. Notice, however, the

efficiency penalties with are involved, due to repeated evaluation of expressions. This is the theme of Section 20.11.

We can summarize as follows.

Normal-order reduction is more powerful than the applicative-order reduction

Scheme and ML uses applicative-order reduction

Haskell is an example of a functional programming language with normal-order reduction

20.10. Conditionals and sequential boolean operators

Lecture 5 - slide 18

In languages with applicative-order reduction there is a need to control the evaluation process in order to avoid the traps of erroneous and infinite calculations. In this section we review a couple of widely used and important forms from Scheme and Lisp. The *evaluation control* of these should in particular be noticed.

There are functional language constructs - special forms - for which applicative order reduction would not make sense

- `(if b x y)`
 - Depending on the value of `b`, either `x` or `y` are evaluated
 - It would often be harmful to evaluate both `x` and `y` before the selection
 - `(define (fak n) (if (= n 0) 1 (* n (fak (- n 1)))))`
- `(and x y z)`
 - `and` evaluates its parameter from left to right
 - In case `x` is false, there is no need to evaluate `y` and `z`
 - Often, it would be harmful to evaluate `y` and `z`
 - `(and (not (= y 0)) (even? (quotient x y)))`

In the items above we discuss the general semantics of `if` and `and`. In the deepest items we give a concrete examples of `if` and `and` where the evaluation order matters.

20.11. Lazy evaluation

Lecture 5 - slide 19

Lazy evaluation is a particular implementation of normal-order reduction which takes care of the lurking multiple evaluations identified in Section 20.7.

We will now deal with a practical variant of normal-order reduction

Lazy evaluation is an implementation of normal-order reduction which avoids repeated calculation of subexpressions

In Figure 20.3 we show an evaluation idea which is based on normal-order reduction without multiple evaluation of parameters, which are used two or more times in the body of a function.

It is not our intention in this material to go deeper into the realization of an interpreter that supports lazy evaluation.

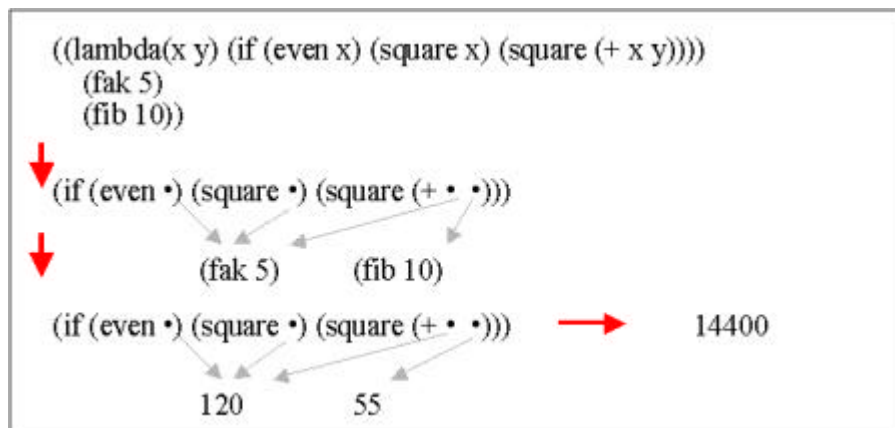


Figure 20.3 An illustration of lazy evaluation of a Scheme expression. Notice, that Scheme does not evaluate the expression in this way. Scheme uses applicative-order reduction.

This ends the general coverage of evaluation order. In the next chapter we will see how to explore the insights from this chapter in Scheme, which is a language with traditional, applicative-order reduction.

20.12. References

- [-] Foldoc: lazy evaluation
<http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?query=lazy+evaluation>
- [-] Foldoc: church-rosser
<http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?query=church-rosser>

- [-] **Foldoc: normal form**
<http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?query=normal+form>
- [-] **Foldoc: eta conversion**
<http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?query=eta+conversion>
- [-] **Foldoc: beta conversion**
<http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?query=beta+conversion>
- [-] **Foldoc: alpha conversion**
<http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?query=alpha+conversion>
- [abelson96] **Abelson, H., Sussman G.J. and Sussman J., *Structure and Interpretation of Computer Programs, second edition.* The MIT Press, 1996.**

21. Delayed evaluation and infinite lists in Scheme

As noticed in Chapter 19 and Chapter 20 the evaluation strategy in Scheme is the one called *applicative order*, cf. Section 20.6.

In contrast to normal-order reduction and lazy evaluation - as described in Section 20.6 and Section 20.11 - we can think of Scheme as *eager* in the evaluation of the function parameters.

In this chapter we will see how to make use of a new evaluation idea in terms of explicitly delaying the evaluation of certain expressions. This is the topic of Section 21.1 and Section 21.2.

21.1. Delayed evaluation in Scheme

Lecture 5 - slide 21

The starting point of our discussion is now clear.

Scheme does not support normal-order reduction nor lazy evaluation
Scheme has an explicit primitive which delays an evaluation

The `delay` and `force` primitives are described in Syntax 21.1 and Syntax 21.2. The `delay` primitive returns a so-called *promise*, which can be redeemed by the `force` primitive. Thus, the composition of `delay` and `force` carry out a normal evaluation step.

```
(delay expr) => promise
```

Syntax 21.1

```
(force promise) => value
```

Syntax 21.2

In Program 21.1 we show simple implementations of `delay` and `force`. In Program 21.2 we show possible implementations of `delay` by means of Scheme macros.

```
(delay expr) ~ (lambda() expr)
(define (force promise) (promise))
```

Program 21.1 *A principled implementation of delay and force in Scheme.*

The thing to notice is the semantic idea behind the implementation of `delay`. The expression `(delay expr)` is equivalent to the expression `(lambda () expr)`. The first expression is supposed to replace the other expression at program source level. The value of the lambda expression is a closure, cf. Section 8.11, which captures free names in its context together with the syntactic form of `expr`. As it appears from the definition of the function `force` in Program 21.1 the promise returned by the `delay` form is redeemed by calling the parameter less function object. It is easy to see that this carries out the evaluation of `expr`.

Be sure to observe that `force` can be implemented by a function, whereas `delay` cannot. The reason is, of course, that we cannot allow a functional implementation of `delay` to evaluate the parameter of `delay`. The whole point of `delay` is to avoid such evaluation. This rules out an implementation of `delay` as a function. The `force` primitive, on the other hand, can be implemented by a function, because it works on the value of a lambda expression.

Please notice that other implementations of `delay` and `force` can easily be imagined. The Scheme Report describes language implementations of `delay` and `force`, which may use other means than described above to obtain the same semantic effect, cf. [delay-primitive] and [force-primitive].

```
; R5RS syntactic abstraction:
(define-syntax my-delay
  (syntax-rules ()
    ((delay expr)
     (lambda ()
       expr))))

; MzScheme syntactic abstraction:
(define-macro my-delay
  (lambda (expr)
    `(lambda () ,expr)))
```

Program 21.2 *Real implementations of delay. The first definition uses the R5RS macro facility, whereas the last one uses a more primitive macro facility, which happens to be supported in MzScheme.*

21.2. Examples of delayed evaluation

Lecture 5 - slide 22

Let us look at a few very simple examples of using `delay` and `force`. In the first line of the table below we delay the expression `(+ 5 6)`. The value is a promise that enables us to evaluate the sum when necessary, i.e, when we choose to force it. The next line shows that we cannot force a non-promise value. The last line shows an immediate forcing of the promise, which we bind to the name `delayed` in the `let` construct.

Expression	Value
<code>(delay (+ 5 6))</code>	<code>#<promise></code>
<code>(force 11)</code>	<i>error</i>
<code>(let ((delayed (delay (+ 5 6)))) (force delayed))</code>	11

Table 21.1 Examples of use of *delay* and *force*.

21.3. Infinite lists in Scheme: Streams

Lecture 5 - slide 23

We are now done with the toy examples. It is time to use delayed evaluation in Scheme to something of real value. In this material we focus on *streams*. A stream is an infinite list. The inspiration to our coverage of streams comes directly from the book *Structure and Interpretation of Computer Programs* [Abelson98].

The crucial observation is the following.

We can work with lists of infinite length by delaying the evaluation of every list tail using `delay`

As an invariant, every list tail will be delayed

Every tail of a list is a promise. The promise covers an evaluation which gives a new cons cell, in which the tail contains another promise.

It is simple to define a vocabulary of stream functions. There is an obvious relationship between list functions (see Section 6.1) and the stream functions shown below in Program 21.3.

```
(cons-stream a b) ~ (cons a (delay b))

(define head car)

(define (tail stream) (force (cdr stream)))

(define empty-stream? null?)

(define the-empty-stream '())
```

Program 21.3 Stream primitives. Notice the way *head* is defined to be an alias of *car*.

In that same way as we defined `delay` as a macro in Program 21.2, we also need to define `cons-stream` as a macro. The reason is that we are not allowed to evaluate the second parameter; The second parameter of `cons-cell` is going to be delayed, and as such it must be passed unevaluated to `cons-stream`.

```
(define-macro cons-stream
  (lambda (a b)
    `(cons ,a (delay ,b))))
```

Program 21.4 A *MzScheme* implementation of `cons-stream`.

In the following sections we will study a number of interesting examples of streams from the numerical domain.

21.4. Example streams

Lecture 5 - slide 24

In the first example line in Table 21.2 we define a stream of ones. In other words, the name `ones` is bound to an infinite list of ones: `(1 1 1 ...)`.

Please notice the very direct use of recursion in the definition of `ones`. We are used to a conditional such as `cond` or `if` when we deal with recursion, in order to identify a basis case which stops the recursive evaluation process. We do not have such a construction here. The reason is that we never reach any basis (or terminating case) of the reduction. Due to the use of delayed evaluation we never attempt to expand the entire list. Instead, there is a promise in the end of the list which can deliver more elements if needed.

In the second row of the example we use the function `stream-section` to extract a certain prefix of the list (determined by the first parameter of `stream-section`). The function `stream-section` is defined in Program 21.5 together with another useful stream function called `add-streams` which adds elements of two numeric streams together.

In the third row we define a stream of all natural numbers, using the function `integers-starting-from`.

The fourth row shows an alternative definition of `nat-nums`. We use `add-streams` on `nat-nums` and `ones` to produce `nat-nums`. Please notice the recursion which is involved.

In the bottom row of the table we define the Fibonacci numbers, in a way similar to the definition of `nat-nums` just above. `fibs` is defined by adding `fibs` to its own tail. This works out because we provide enough starting numbers `(0 1)` to get the process started.

Expression	Value
<code>(define ones (cons-stream 1 ones))</code>	<code>(1 . #<promise>)</code>
<code>(stream-section 7 ones)</code>	<code>(1 1 1 1 1 1 1)</code>
<code>(define (integers-starting-from n)</code> <code> (cons-stream n</code> <code> (integers-starting-from (+ n 1))))</code> <code>(define nat-nums</code> <code> (integers-starting-from 1))</code> <code>(stream-section 10 nat-nums)</code>	<code>(1 2 3 4 5 6 7 8 9 10)</code>
<code>(define nat-nums</code> <code> (cons-stream 1</code> <code> (add-streams ones nat-nums)))</code> <code>(stream-section 10 nat-nums)</code>	<code>(1 2 3 4 5 6 7 8 9 10)</code>
<code>(define fibs</code> <code> (cons-stream 0</code> <code> (cons-stream 1</code> <code> (add-streams (tail fibs) fibs))))</code> <code>(stream-section 15 fibs)</code>	<code>(0 1 1 2 3 5 8 13 21 34 55 89 144</code> <code> 233 377)</code>

Table 21.2 *Examples of streams. ones is an infinite streams of the element 1. stream-section is a function that returns a finite section of a potentially infinite stream. nat-nums is stream of all the natural numbers, made by use of the recursive function integers-starting-from. The fourth row shows an alternative definition of nat-nums. Finally, fibs is the infinite stream of Fibonacci numbers.*

As mentioned above, the functions `stream-section` and `add-streams` in Program 21.5 are used in Table 21.2.

In the web version of the material (slide and annotated slide view) there is an additional program with all the necessary definitions which allow you to play with streams in MzScheme or DrScheme.

```
(define (stream-section n stream)
  (cond ((= n 0) '())
        (else
         (cons
          (head stream)
          (stream-section
           (- n 1)
           (tail stream))))))

(define (add-streams s1 s2)
  (let ((h1 (head s1))
        (h2 (head s2)))
    (cons-stream
     (+ h1 h2)
     (add-streams (tail s1) (tail s2)))))
```

Program 21.5 *The functions stream-section and add-streams.*

21.5. Stream example: The sieve of Eratosthenes

Lecture 5 - slide 25

Still with direct inspiration from the book *Structure and Interpretation of Computer Programs* [Abelson98] we will look at a slightly more complicated example, namely generation of the stream of prime numbers. This is an infinite list, because the set of prime numbers is not finite.

The algorithmic idea behind the generation of prime numbers, see Program 21.6 was originally conceived by Eratosthenes (a Greek mathematician, astronomer, and geographer who devised a map of the world and estimated the circumference of the earth and the distance to the moon and the sun - according to the American Heritage Dictionary of the English Language).

The input of the function `sieve` in Program 21.6 is the natural numbers starting from 2. See also the example in Table 21.3. The first element in the input is taken to be a prime number. Let us say the first such number is p . No number $p \cdot n$, where n is a natural number greater than one, can then be a prime number. Program 21.6 sets up a sieve which disregards such numbers.

Recursively, the first number which comes out of the actual chain of sieves is a prime number, and it is used set up a new filter. This is due to the simple fact that the `sieve` function calls itself.

The Sieve of Eratosthenes is a more sophisticated example of the use of streams

```
(define (sieve stream)
  (cons-stream
    (head stream)
    (sieve
      (filter-stream
        (lambda (x) (not (divisible? x (head stream))))
        (tail stream))))))
```

Program 21.6 *The sieve stream function.*

Program 21.6 uses the functions `cons-stream`, `head` and `tail` from Program 21.3. The functions `filter-stream` and `divisible?` are defined in Program 21.7.

Figure Figure 21.1 shows a number of sieves, and it sketches the way the numbers (2 3 4 ...) are sieved. Notice that an infinite numbers of sieves are set up - on demand - when we in the end requests prime numbers.

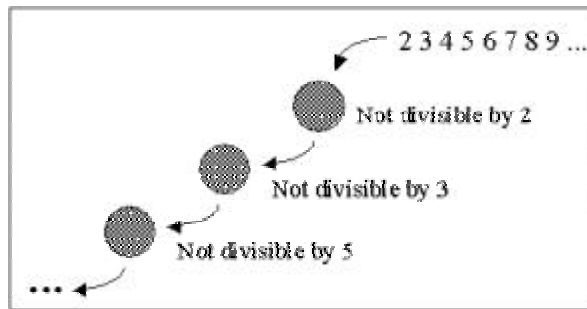


Figure 21.1 An illustration of the generation of prime numbers in The Sieve of Eratosthenes

21.6. Applications of The sieve of Eratosthenes

Lecture 5 - slide 26

In this section we show an example of prime number generation with the sieve function from Program 21.6.

Notice that the prime numbers are really generated on demand. In the call `(stream-section 25 primes)` we are requesting 25 prime numbers. This triggers generation of sufficient natural numbers via `(integers-starting-from 2)`, and it triggers the set up of sufficient sieves to produce the result.

We see that the evaluations are done *on demand*.

The sieve process produces the stream of all prime numbers

Expression	Value
<pre>(define primes (sieve (integers-starting-from 2))) (stream-section 25 primes)</pre>	<pre>(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97)</pre>

Table 21.3 The first 25 prime numbers made by sieving a sufficiently long prefix of the integers starting from 2.

You can use the definitions in Program 21.7 to play with the `sieve` function. You should first load the stream stuff discussed in Section 21.4. More specifically, you should load the definitions on the last program clause in the slide view of Section 21.4. Then load the definitions in Program 21.7.

```

(define (sieve stream)
  (cons-stream
    (head stream)
    (sieve
      (filter-stream
        (lambda (x) (not (divisible? x (head stream))))
        (tail stream)))))

(define (divisible? x y)
  (= (remainder x y) 0))

(define (filter-stream p lst)
  (cond ((empty-stream? lst) the-empty-stream)
        ((p (head lst)) (cons-stream (head lst) (filter-stream p (tail lst))))
        (else (filter-stream p (tail lst)))))

(define (integers-starting-from n)
  (cons-stream n
    (integers-starting-from (+ n 1))))

(define primes (sieve (integers-starting-from 2)))

```

Program 21.7 *All the functions necessary to use the Sieve of Eratosthenes. In addition, however, you must load the Scheme stream stuff. The most remarkable function is `filter-streams`, which illustrates that it is necessary to rewrite all our classical higher order function to stream variants. This is clearly a drawback!*

21.7. References

- [force-primitive] R5RS: force
http://www.cs.auc.dk/~normark/prog3-03/external-material/r5rs/r5rs-html/r5rs_63.html#SEC65
- [delay-primitive] R5RS: delay
http://www.cs.auc.dk/~normark/prog3-03/external-material/r5rs/r5rs-html/r5rs_36.html#SEC38
- [abelson98] Richard Kelsey, William Clinger and Jonathan Rees, "Revised⁵ Report on the Algorithmic Language Scheme", *Higher-Order and Symbolic Computation*, Vol. 11, No. 1, August 1998, pp. 7--105.