Introduction to Lambda Calculus through redex

By Jacob J. A. Koot

Last update: 11/10/2011 12:45:58, Mataró, Barcelona, Spain

Contents

Introduction

Contents

Every Racket [1] programmer knows the word ‘lambda’, which is the name of the Greek letter ‘λ’. Not every Racket programmer knows the beautiful field of mathematics from which the use of the word ‘lambda’ originates. This piece of mathematics is called ‘Lambda Calculus’ [2]. Knowledge of the Lambda Calculus is not a prerequisite in order to become a good programmer. However, for a Racket programmer with only some basic mathematical knowledge and some curiosity to know the ideas on which Racket and other functional programming languages are built, it is not difficult to grasp the basics of the mathematical theory. In this essay I attempt to provide a short and simple introduction into the Lambda Calculus to the aforementioned audience. See Barendregt for a comprehensive treatment of the Lambda Calculus.

Formal languages

Contents

The Lambda Calculus is a formal mathematical system that, very much like a programming language, has a grammar and a set of semantic rules. The grammar tells us which forms a program can take. The semantics tell us what a program means, in particular how a program (interpreted or compiled and run) produces its results. The Lambda Calculus can be regarded as the ultimate mathematical abstraction of all programming languages. It consists of abstraction (*lambda*), variables and application only, nothing else, no numbers, no characters, no lists, no strings, no vectors and not even one single primitive procedure nor any syntactic form other than *lambda*. Therefore the Lambda Calculus is not suited for real life programming. Nevertheless, the Lambda Calculus allows the description of every feasible computation. For example, one can describe Racket and a universal Turing machine in terms of the Lambda Calculus. It will be shown how to represent data such as pairs and numbers and how to create self reference and recursion without any of the self-referential Racket forms such as *define*, named *let* and *letrec*. Because the Lambda Calculus is a mathematical theory, one can reason about it with mathematical methods. Therefore the Lambda Calculus is an important tool in scientific studies of the properties of real life programming languages.

Tools and source texts

Contents

This essay comes with a set of files providing tools and all source texts used and shown in this essay. They depend on PLT’s Racket, [4] particularly on its library redex. See the appendix for instructions how to download and use PLT Racket and the source texts. It takes a few mouse clicks only. Make sure to use a copy of this essay downloaded and stored as described in the appendix.

Reasoning *within* or *about* a formal system [5]

Contents

The grammar of a language tells us which forms a program can take. A correctly composed program is called a ‘well formed sentence’. The semantics consist of the rules of reduction and tell us how a program is evaluated. In order to find the properties and the results of a language or one of its programs, one can slavishly follow the rules of reduction just like an interpreter would do. This can be called: “reasoning *within* the system”. However, one may step out of the system and deduce properties of a language or program by contemplating the properties of its grammar and its semantics. This can be called: “reasoning *about* the system”. The MIU puzzle provides a good illustration of the difference between the two approaches. The language of this puzzle consists of all (proper) lists of the symbols `M`, `I` and `U`. [6] The language and its rules are:

1 Almost everywhere you read “Racket”, you may also read “Scheme”.

2 There are several Lambda Calculi. Here λKβ and λKβη are treated and are simply referred to as *the* Lambda Calculus.

4 PLT’s Racket includes Scheme and much more.

5 With thanks to Douglas R. Hofstadter who permitted me to borrow and adapt the MIU puzzle and to borrow the distinction between *within* and *about* from his book “Gödel, Escher, Bach, an Eternal Golden Braid”. This book is a must!

6 In order to avoid an abundance of quotes, data or programs or fragments thereof, often are implicitly quoted. In general `this font` is to be read as implicitly quoted within running text.

	Grammar:		
a	‹sentence› ::= (‹symbol› **...**)		
b	‹symbol› ::= `M` | `I` | `U`		
	Semantics:		
0	(`M I`)		is given as an axiom.
1	(‹symbol› **...** `I`)	⇒	(‹symbol› **...** `I U`)
2	(`M` ‹symbol› **...**)	⇒	(`M` ‹symbol› **...** ‹symbol› **...**)
3	(‹symbol›$_0$ **...** `I I I` ‹symbol›$_1$ **...**)	⇒	(‹symbol›$_0$ **...** `U` ‹symbol›$_1$ **...**)

We regard (`M I`) as an axiom. Every sentence that can be deduced from the axiom by means of zero or more applications of rules 1, 2 and 3 can be regarded as a theorem, the axiom included. The left hand side of a rule is a pattern to be matched. The right hand side a template. Each rule indicates that if a sentence matching the pattern is a theorem, the template is a theorem as well. Every sequence '‹symbol› **...**' matches an arbitrary sequence of the symbols `M`, `I` and `U`, but within an application of a rule every occurrence of '‹symbol› **...**' represents the same sequence. Subscripts are used where it is necessary to distinguish between possibly different sequences (as in rule 3). Rule 1 states that symbol `U` may be added at the end of a sentence that ends with symbol `I`. Rule 2 indicates that a sentence starting with `M` may be replaced by the sentence formed by appending all that follows the starting `M`, at the end of the original sentence. Rule 3 indicates that three consecutive symbols `I` may be replaced by one single symbol `U`. The question is: "Is (`M U`) a theorem?" The puzzle can easily be described in PLT's redex:

```
#lang racket                                                   ; File miu-puzzle.rkt
(require redex)
(provide MIU-rules traces)

(define-language MIU-language           ; Grammar.
 (‹sentence› (‹symbol› ...))
 (‹symbol› M I U))

(define MIU-rules                       ; Semantics.
 (reduction-relation MIU-language
  (--> (‹symbol› ... I)                 (‹symbol› ... I U)                "rule 1")
  (--> (M ‹symbol› ...)                 (M ‹symbol› ... ‹symbol› ...)     "rule 2")
  (--> (‹symbol›_0 ... I I I ‹symbol›_1 ...)                              (‹symbol›_0 ... U
‹symbol›_1 ...)                         "rule 3")))
```

In redex a subscript is written as an underscore followed by a natural number. In order to see part of the graph of deductions starting from the axiom, try:

```
#lang racket                                                   ; File miu-traces.rkt
(require "miu-puzzle.rkt")
(traces MIU-rules '(M I))
```

Take a look at the graphical user interface opened by procedure *traces* and play with it (you may want to expand it to full screen) Procedure *traces* comes from PLT's library redex. In this example it forms a graph of all theorems that can be deduced from axiom (`M I`). However, the procedure is protected such as to halt when the graph outgrows some maximum size. Click the reduce button in order to find more theorems. Notice that many theorems can be deduced in more than one way. If proposition (`M U`) is a theorem, then we may repeatedly press the reduce button until (`M U`) appears in the graph. If proposition (`M U`) is a theorem, procedure *traces* ultimately will find it. If (`M U`) is not a theorem, it will never be found, of course, regardless of the number of times the graph is extended.

Question

Is (`M U`) a theorem? Think about it before turning to the next page.

Answer
The number of different theorems that can be deduced from the axiom is infinite. For example, the second rule implies that from axiom `(M I)` we can deduce every theorem `(M I ...)` whose number of occurrences of symbol `I` is a power of two. If the number of different theorems deducible from `(M I)` would be finite, then procedure *traces* eventually would signal that no more theorems can be found.

`(M U)` is not a theorem. If the graph of deductions would be finite, this could be checked by inspecting the whole graph and looking whether or not `(M U)` appears in the graph. However, the graph is infinite. In order to prove that `(M U)` is not a theorem, we must step out of the system and look at the rules from a higher point of view. Now the proof is easily found. The number of occurrences of symbol `I` in axiom `(M I)` is not a multiple of 3. This property is conserved by the three rules of deduction. Hence every theorem has the property. Because proposition `(M U)` does not have the property, it is not a theorem.

When studying the Lambda Calculus, it is important to be aware in which mode we are working: reasoning *within* the system or reasoning *about* the system.

The grammar of the Lambda Calculus

Contents

In the Lambda Calculus a well formed sentence or program is called a ‘lambda term’ or shortly a ‘term’. In Backus-Naur notation the grammar of the Lambda Calculus as used in this essay is: [7]

‹term›::= ‹var›	Variable reference
‹term›::= (λ (‹var›) ‹term›)	Abstraction
‹term›::= (‹term› ‹term›)	Application
‹var› ::= identifier, but not λ	We don’t allow λ to be used as the name of a variable.

That’s all. There are no semantic restrictions. Notice that in Racket identifier ‘`λ`’ is a synonym of ‘`lambda`’. In this essay ‘`λ`’ is used (typed as control backslash) Free, id est unbound, variables will sometimes be used. In Racket they are not permitted or at least give rise to problems when referenced. However, one might imagine a Racket dialect in which the value of an unbound variable is the very same symbol that names the variable. There are, or at least have been, some Lisp dialects with this property. Because the Lambda Calculus has no define-forms, there are no globally bound variables. When talking about lambda terms it is handy to have names for its components:

When occurring as an instantiation of a ‹term›, ‹var› is a subterm, particularly a **variable reference**.
In (λ (‹var›) ‹term›), the ‹term› is a subterm and is referred to as the **body** of the abstraction
In (λ (‹var›) ‹term›), the ‹var› is a ***formal argument***. This occurrence of the ‹var› is ***not a subterm***!
In (‹term›$_0$ ‹term›$_1$), ‹term›$_0$ is a subterm, particularly the **operator** of an application.
In (‹term›$_0$ ‹term›$_1$), ‹term›$_1$ is a subterm, particularly the **actual argument** of an application.

The above grammar is easily rewritten for redex. In addition we prepare some metapredicates and auxiliary non terminals like variable lists and environments. Later they will appear to be useful.

7 The conventional notation (without shorthand) for the Lambda Calculus is:

‹term› ::= ‹var›	Use variables that do not require separators.
‹term› ::= (λ‹var›‹term›)	Abstraction.
‹term› ::= (‹term›‹term›)	Application.
For example: ((λx(xx))(λx(xx)))	which is called ‘Ω’.

We use a notation closer to that of Racket: ((λ (x) (x x)) (λ (x) (x x))). Mark the spaces between variables. By convention this example also has some spaces before or after a parenthesis, although Racket does not require these spaces.

```
#lang racket                                                    ; File curried-lc-grammar.rkt
(require redex)
(require "define-language-with-metapredicates.rkt")
(provide curried-lc-grammar Term? Abstr? Appl? Var? Varlist? Env? Binding?)

#| define-language-with-metapredicates is like define-language of redex, but allows each
clause to be adorned with a predicate by means of keyword #:pred. |#

(define-language-with-metapredicates curried-lc-grammar
 (‹term›     ‹var› ‹abstr› ‹appl›   #:pred Term?     )
 (‹abstr›    (λ (‹var›) ‹term›)     #:pred Abstr?    )
 (‹appl›     (‹term› ‹term›)        #:pred Appl?     )
 (‹var›      (variable-except λ)    #:pred Var?      )

; Additional forms.
 (‹varlist›  (‹var› ...)            #:pred Varlist?  )
 (‹env›      (‹binding› ...)        #:pred Env?      )
 (‹binding›  (‹var› number)         #:pred Binding?  )
 (‹bool›     #f #t                                   )
 (‹subterm›  (λ (‹var›) ‹subterm›) (‹subterm› ‹term›) (‹term› ‹subterm›) hole))
```

Within the context of *curried-lc-grammar*, the identifiers that appear as the first one in a clause, like ‹term› and ‹var›, are non terminals. To obtain a well formed lambda term, start with ‹term› and apply the rules of the grammar one or more times such as to obtain a sentence without non terminals. The word ‘hole’ in the last clause indicates that ‹subterm›s are indeed to be recognized as subterms within a context. *Redex* provides many tools to check the consistency of its use, for example:

```
#lang racket                                               ; File test-curried-lc-grammar.rkt
(require redex "curried-lc-grammar.rkt")
(redex-check curried-lc-grammar ‹term›
 (= ; Here redex-check randomly forms 10000 terms and checks that the condition is satisfied.
  (apply +
   (map (λ (x) (if x 1 0))
    (list                          ; Here we check that every ‹term› is either
     (term (Var?    ‹term›))       ; a variable reference,
     (term (Abstr? ‹term›))        ; an abstraction or
     (term (Appl?   ‹term›)))))    ; an application.
  1) #:attempts 10000 #:retries 100)

(test-false (term (Term?  123)))
(test-false (term (Var?   123)))
(test-false (term (Abstr? 123)))
(test-false (term (Appl?  123)))
(test-true  (term (Term?  x)))
(test-true  (term (Var?   x)))
(test-false (term (Abstr? x)))
(test-false (term (Appl?  x)))
(test-true  (term (Term?  (x y))))
(test-false (term (Var?   (x y))))
(test-false (term (Abstr? (x y))))
(test-true  (term (Appl?  (x y))))
(test-true  (term (Term?  (λ (x) y))))
(test-false (term (Var?   (λ (x) y))))
(test-true  (term (Abstr? (λ (x) y))))
```

```
(test-false (term (Appl?  (λ (x) y))))
(test-false (term (Term?  λ)))
(test-false (term (Term?  (λ (x y) z))))
(test-false (term (Term?  (λ (x)))))
(test-false (term (Term?  (x y z))))
(test-results)
```

Scope

Contents

Apart from occasional free variables and the absence of global bindings, the Lambda Calculus has the same rules of scope as Racket (in fact rather reversely). The formal argument of an abstraction is bound within the body of that abstraction and bindings may be shadowed. In the example below, x_1 creates a binding. x_2 creates a binding that shadows that of x_1. x_3 is a variable reference related to the binding of x_2. Finally, x_4 is a free occurrence of variable x.

```
((λ (x) (λ (x) x)) x)
     1      2  3   4
```

Free variables

Contents

Metafunction *Free-vars* finds the list of free variables of a lambda term. The produced list is sorted and does not contain duplicates. Metafunction *Var-free-in?* returns `#t` or `#f` depending on whether or not a given variable occurs free in a given term.

```
#lang racket                                                    ; File free-vars.rkt
(require redex "curried-lc-grammar.rkt")
(provide Free-vars Var-free-in?)

(define-metafunction curried-lc-grammar Free-vars : ‹term› -> ‹varlist›
 ((Free-vars ‹var›) (‹var›))
 ((Free-vars (‹term›_0 ‹term›_1)) (Merge (Free-vars ‹term›_0) (Free-vars ‹term›_1)))
 ((Free-vars (λ (‹var›) ‹term›)) (Remove-var ‹var› (Free-vars ‹term›))))

(define-metafunction curried-lc-grammar Var-free-in? : ‹var› ‹term› -> ‹bool›
 ((Var-free-in? ‹var›_0 ‹var›_0) #t)
 ((Var-free-in? ‹var›_0 ‹var›_1) #f)
 ((Var-free-in? ‹var›_0 (λ (‹var›_0) ‹term›)) #f)
 ((Var-free-in? ‹var›_0 (λ (‹var›_1) ‹term›)) (Var-free-in? ‹var›_0 ‹term›))
 ((Var-free-in? ‹var› (‹term›_0 ‹term›_1))
 ,(or (term (Var-free-in? ‹var› ‹term›_0))
      (term (Var-free-in? ‹var› ‹term›_1)))))

(define-metafunction curried-lc-grammar Merge : ‹varlist› ‹varlist› -> ‹varlist›
 ((Merge (‹var›_0 ...) (‹var›_1 ...))
 ,(sort (remove-duplicates (term (‹var›_0 ... ‹var›_1 ...))) symbol<?)))

(define (symbol<? x y) (string<? (symbol->string x) (symbol->string y)))

(define-metafunction curried-lc-grammar Remove-var : ‹var› (‹var› ...) -> (‹var› ...)
 ((Remove-var ‹var›_0 (‹var›_1 ...)) ,(remove (term ‹var›_0) (term (‹var›_1 ...)))))
```

The clauses in the definition of a metafunction have the form (‹pattern› ‹template›), where the pattern must always begin with the name of the metafunction and the template is implicitly wrapped as in (`term` ‹template›). Syntax *term* is like a quasiquotation, allowing *unquote* and *unquote-splicing*, but also recognizing calls to metafunctions and the bindings introduced in the pattern. As in a cond-form, a metafunction selects the first clause that matches. This means that later clauses may depend on the fact that previous clauses did not apply.

```
#lang racket                                                   ; File test-free-vars.rkt
(require "free-vars.rkt" redex)
(test-equal (term (Free-vars (λ (x) (x x)))) (term ( )))
(test-equal (term (Free-vars (λ (x) (x y)))) (term (y)))
(test-equal (term (Free-vars (λ (x) ((x y) z)))) (term (y z)))
(test-equal (term (Free-vars ((λ (x) (x y)) (λ (y) (x y))))) (term (x y)))
(test-equal (term (Var-free-in? x (λ (y) z))) #f)
(test-equal (term (Var-free-in? x (λ (x) x))) #f)
(test-equal (term (Var-free-in? x (λ (y) x))) #t)
(test-results) ; Displays: All 7 tests passed
```

α-congruence

Contents

Consider the lambda terms: (λ (x) x) and (λ (y) y). When interpreted as representations of the (un-typed) identity function, both terms represent the very same function. The two terms are said to be α-congruent. This applies to renaming of bound variables only. It has nothing to do with the question whether or not two terms have the same meaning. For example, (λ (x) ((λ (y) y) x)) is an identity function like (λ (y) y), because for every argument ‹arg› the expression ((λ (x) ((λ (y) y) x)) ‹arg›) has the same value as ‹arg›. However, the two representations are not α-congruent. The α-congruence of two terms can be decided by a metafunction as shown in the next module. We use environments in order to keep track of the bindings of variables.

```
#lang racket                                                ; File alpha-congruence.rkt
(require redex "curried-lc-grammar.rkt")
(provide α-congruent?)

(define-metafunction curried-lc-grammar α-congruent? : ‹term› ‹term› -> ‹bool›
 ((α-congruent? ‹term›_1 ‹term›_2) (α-aux? (‹term›_1 ( )) (‹term›_2 ( )))))

(define-metafunction curried-lc-grammar α-aux? : (‹term› ‹env›) (‹term› ‹env›) -> ‹bool›
 ((α-aux? (‹var›_0 ‹env›_0) (‹var›_1 ‹env›_1))
 ,(let ((e0 (assq (term ‹var›_0) (term ‹env›_0))) (e1 (assq (term ‹var›_1) (term ‹env›_1))))
   (or       (and e0 e1 (= (cadr e0) (cadr e1)))
      (and (not e0) (not e1) (eq? (term ‹var›_0) (term ‹var›_1))))))
 ((α-aux? ((λ (‹var›_0) ‹term›_0) ‹env›_0) ((λ (‹var›_1) ‹term›_1) ‹env›_1))
 ,(term-let ((g (new-scope)))
   (term (α-aux?(‹term›_0 ((‹var›_0 g) ,@(term ‹env›_0)))
                  (‹term›_1 ((‹var›_1 g) ,@(term ‹env›_1)))))))
 ((α-aux? ((‹term›_0 ‹term›_1) ‹env›_0) ((‹term›_2 ‹term›_3) ‹env›_2))
 ,(and       (term (α-aux? (‹term›_0 ‹env›_0) (‹term›_2 ‹env›_2)))
      (term (α-aux? (‹term›_1 ‹env›_0) (‹term›_3 ‹env›_2)))))
 ((α-aux? any_1 any_2) #f))

(define new-scope ; Use semaphore because the guis of traces may run concurrently.
 (let ((semaphore (make-semaphore 1)) (scope 0))
  (λ ( )
   (begin0 (begin (semaphore-wait semaphore) (set! scope (add1 scope)) scope)
    (semaphore-post semaphore)))))
```

```
#lang racket                                            ; File test-alpha-congruence.rkt
(require redex "alpha-congruence.rkt")

(define-syntax tester ; Does concurrent tests.
 (syntax-rules ()
  ((tester ‹term› expected)
    (void
     (set! nr-of-tests (add1 nr-of-tests))
     (thread (λ () (test-equal (term ‹term›) expected) (semaphore-post sema))))))

(define nr-of-tests 0)
(define sema (make-semaphore 0))

(tester
 (α-congruent?
  ((λ (x) (λ (y) (λ (z) ((x y) z)))) a)
  ((λ (p) (λ (q) (λ (r) ((p q) r)))) a))
 #t) ; Because there is a one to one correspondence between all variables.

(tester
 (α-congruent?
  ((λ (x) (λ (y) (λ (z) ((x y) z)))) a)
  ((λ (x) (λ (y) (λ (z) ((x y) z)))) b))
 #f) ; Because free variable a ≠ free variable b.

(tester
 (α-congruent?
  (λ (x) (λ (y) (λ (z) ((z z) z))))
  (λ (x) (λ (x) (λ (x) ((x x) x)))))
 #t) ; Because the underlined abstractions have no references to the other two bindings.

(tester
 (α-congruent?
  (λ (x) (λ (y) (λ (z) ((z z) z))))
  (λ (x) (λ (z) (λ (y) ((z z) z)))))
 #f) ; Because z has different binding levels.

(let loop ((nr-of-tests nr-of-tests))      ; Wait until all threads have finished
 (if (zero? nr-of-tests) (test-results) ; before checking the test results.
  (begin (semaphore-wait sema)
   (loop (sub1 nr-of-tests))))) ; Displays: All 4 tests passed.
```

Curry

Contents

Although the observation was first made by M. Schönfinkel, the name of H. B. Curry is attached to the idea that every function of a fixed number of one or more arguments can be written in terms of functions of one single argument. Applications must be adapted accordingly. For example:

Uncurried	**Curried**
(λ (x y) x)	(λ (x) (λ (y) x))
(a b c d)	(((a b) c) d)
((λ (x y) x) p q)	(((λ (x) (λ (y) x)) p) q)

In order to improve readability, uncurried notation and superfluous parentheses will be allowed, but should always be regarded as shorthand for a fully curried form. This applies to both abstractions and applications. To emphasize this, we write ‹uncurried-term› ≡ ‹corresponding-curried-term›. Later a relation = will be defined for lambda terms, but with quite another meaning.

```
#lang racket                                              ; File uncurried-lc-grammar.rkt
(require redex "curried-lc-grammar.rkt")
(provide uncurried-lc-grammar)

(define-extended-language uncurried-lc-grammar curried-lc-grammar
(‹term› ‹var› (λ (‹var› ...) ‹term›) (‹term› ‹term› ...)))
```

The new language is like the original one, but with the definition of the non terminal ‹term› replaced by the new one. The process of transforming abstractions and applications of more than one argument to nested abstractions and applications of one argument is called 'currying'. The uncurried shorthand of application is left associative, id est, (a b c) means ((a b) c), not (a (b c)). As a Racketeer you already know that these two expressions usually have different behaviour. The formal arguments of an abstraction in uncurried shorthand are not required to be distinct. A formal argument may shadow a formal argument of the same name to its left. For example: (λ (x y x) z) ≡ (λ (x) (λ (y) (λ (x) z))). We shall need a metafunction that transforms an uncurried or partially curried lambda term into a fully curried one.

```
#lang racket                                                            ; File curry.rkt
(require redex "uncurried-lc-grammar.rkt")
(provide Curry)

(define-metafunction uncurried-lc-grammar Curry : ‹term› -> ‹term›
 ((Curry (λ () ‹term›))                      (Curry ‹term›))
 ((Curry (λ (‹var›_0 ‹var›_1 ...) ‹term›))  (λ (‹var›_0) (Curry (λ (‹var›_1 ...) ‹term›))))
 ((Curry (‹term›))                           (Curry ‹term›))
 ((Curry (‹term›_0 ‹term›_1))                ((Curry ‹term›_0) (Curry ‹term›_1)))
 ((Curry (‹term›_0 ‹term›_1 ‹term›_2 ...)) (Curry ((‹term›_0 ‹term›_1) ‹term›_2 ...)))
 ((Curry ‹var›)                              ‹var›))
```

The identifications (‹term›) ≡ ‹term› and (λ ( ) ‹term›) ≡ ‹term› cannot be made in Racket for it does not require that a function has an argument. For example ((λ ( ) 3)) → 3, whereas (λ ( ) 3) → #<procedure>. In the Lambda Calculus the identifications are quite natural, for (‹term›) and (λ ( ) ‹term›) are not allowed as curried lambda terms.

```
#lang racket                                                       ; File test-curry.rkt
(require redex "curry.rkt" "uncurried-lc-grammar.rkt" "curried-lc-grammar.rkt")

(test-equal
 (term (Curry ((λ (x x x) (x x x)) (λ (x x x) (x x x)) (λ (x x x) (x x x)))))
 (term (((λ (x) (λ (x) (λ (x) ((x x) x)))) (λ (x) (λ (x) (λ (x) ((x x) x))))) (λ (x) (λ (x) (λ (x) ((x x) x)))))))

(test-equal (term (Curry ((((x)))))) (term x))
(test-equal (term (Curry ((((λ () (((((x) (y)))))))))))) (term (x y)))

(redex-check uncurried-lc-grammar ‹term› ; Check that Curry indeed forms curried terms.
 (term (Term? (Curry ‹term›))) #:attempts 2000 #:retries 100)

(test-results) ; Displays:
; redex-check: ...\test-curry.rkt:13 no counterexamples in 2000 attempts
; All 3 tests passed.
```

Reduction

Contents

Up to now the grammar of the Lambda Calculus has been presented, together with some metafunctions and meta-predicates for curried terms. Nothing has been said about the semantics yet. In the Lambda Calculus evaluation is called 'reduction'. A lambda term is reduced by contraction of redices until no more redices are left. There are two types of redices, β-redices and η-redices.

β-redices

A β-redex is a term or subterm of the form:

((λ (‹formal-arg›) ‹body›) ‹actual-arg›), where ‹body› and ‹actual-arg› are curried terms.

A β-redex is an application of an abstraction to an actual argument. Somehow the formal argument must be associated with the actual argument while the body of the abstraction is evaluated. In Racket this is done by evaluation of the actual argument and binding the formal argument to the value of the actual argument. Subsequently the value of the body is computed in the environment of the abstraction to which the new binding is added, possibly shadowing another binding of a variable of the same name. In the Lambda Calculus substitution is used. In principle this means that the above term can be contracted to the body, but with all free (id est non shadowed) occurrences of the formal argument replaced by the actual argument.

For example: ((λ (x) ((x x) (λ (x) x))) p) $\rightarrow^{\beta}$ ((p p) (λ (x) x)), where the free occurrences of x in body ((**x x**) (λ (x) x)), id est the boldface ones, are replaced by actual argument p. However, there is a nasty pitfall:

((λ (x) (λ (y) x)) y) $\rightarrow^{\beta}$ (λ (y) y), wrong.
((λ (x) (λ (z) x)) y) $\rightarrow^{\beta}$ (λ (z) y), ok.

The problem is revealed by writing the free variable in boldface:

((λ (x) (λ (y) x)) **y**) $\rightarrow^{\beta}$ (λ (y) **y**)

A variable must not be substituted in a context where it would obtain another binding or would become bound whereas it was free. In this case correct substitution requires α-conversion:

((λ (x) (λ (y) x)) y) $\rightarrow^{\alpha}$ ((λ (x) (λ (z) x)) y) $\rightarrow^{\beta}$ (λ (z) y), ok.

Of course, we use *redex* for the implementation of the β-reduction:

```
#lang racket                                                ; File beta-reductor.rkt
(require redex "curried-lc-grammar.rkt" "free-vars.rkt")
(provide β-reductor Subst βnf?)

(define β-reductor
 (reduction-relation curried-lc-grammar
  (-->      (in-hole ‹subterm› ((λ (‹var›) ‹term›_0) ‹term›_1))
      (in-hole ‹subterm› (Subst (‹var› ‹term›_1) ‹term›_0)) "β")))

; The word ‘in-hole’ indicates that subterms must be looked for within a possibly more
; complicated term and that the rule applies to each such subterm within its context.
; Rules without the word ‘in-hole’ apply to whole terms only.

(define-metafunction curried-lc-grammar βnf? : ‹term› -> ‹bool›
 ((βnf? ‹term›) ,(null? (apply-reduction-relation β-reductor (term ‹term›)))))
```

```
; Metafunction Subst does the substitution for β-contraction
; It is almost identical to the one found in beginner friendly introduction of redex.

(define-metafunction curried-lc-grammar Subst : (‹var› ‹term›) ‹term› -> ‹term›
  ; Matching var: substitute ‹term›_0 for ‹var›_0.
  ((Subst (‹var›_0 ‹term›_0) ‹var›_0) ‹term›_0)
  ; Non matching var: no substitution.
  ((Subst (‹var›_0 ‹term›_0) ‹var›_1) ‹var›_1)
  ; Subst in abstr that shadows the var: no substitution in body.
  ((Subst (‹var›_0 ‹term›_0) (λ (‹var›_0) ‹term›_1)) (λ (‹var›_0) ‹term›_1))
  ; Subst in abstr that does not shadow the var.
  ((Subst (‹var›_0 ‹term›_0) (λ (‹var›_1) ‹term›_1))
  ,(term-let ((Free (term (Free-vars ‹term›_0))))
     (if (member (term ‹var›_1) (term Free))
       (term-let ; α-conversion required.
         ((‹new-var›
             (variable-not-in (term (‹var›_0 Free (Free-vars ‹term›_1))) (term ‹var›_1))))
         (term
           (λ (‹new-var›) ; ‹new-var› is the new name for ‹var›_1.
             (Subst (‹var›_0 ‹term›_0) (Subst (‹var›_1 ‹new-var›) ‹term›_1)))))
       (term (λ (‹var›_1) (Subst (‹var›_0 ‹term›_0) ‹term›_1))))))) ; α-conversion not required.
  ; Application: recur on operator and actual argument.
  ((Subst (‹var›_0 ‹term›_0) (‹term›_1 ‹term›_2))
   ((Subst (‹var›_0 ‹term›_0) ‹term›_1) (Subst (‹var›_0 ‹term›_0) ‹term›_2))))
```

```
#lang racket                                                  ; File test-beta-reductor.rkt
(require redex "beta-reductor.rkt" "tracer.rkt")

(parameterize ((reduction-steps-cutoff 100))
 (tracer β-reductor
  ((x y z)                                   (x y z))
  ((λ (x) x)                                 (λ (y) y))
  ((λ (x) (F x))                             (λ (x) (F x)))
  (((λ (x) (y x)) x)                         (y x))
  (((λ (x y) x) y)                           (λ (p) y))
  (((λ (x) (λ (y) (x y))) z)                 (λ (y) (z y)))
  (((λ (x y z) (x y z)) a b c)               (a b c))
  (((λ (x) (x x)) (λ (x) (x x)))             #f)
  ((λ (p) ((λ (x) (x x)) (λ (x) (x x))))     #f)
  (((((λ (z) (z z)) (λ (x y) (x (x y)))) a) z)  (a (a (a (a z))))))

(parameterize ((reduction-steps-cutoff 10))
 (tracer β-reductor (((λ (x) (x x x)) (λ (x) (x x x))) #f)))
```

η-redices

An η-redex is a lambda term of the form (λ (‹var›$_0$) (‹term›$_0$ ‹var›$_0$)), where ‹term›$_0$ does not contain any free occurrence of its formal argument. With this condition, we have:

((λ (‹var›$_0$) (‹term›$_0$ ‹var›$_0$)) ‹term›$_1$) $\rightarrow^{\beta}$ (‹term›$_0$ ‹term›$_1$)

Therefore we can make the following contraction:

(λ (‹var›$_0$) (‹term›$_0$ ‹var›$_0$)) $\rightarrow^{\eta}$ ‹term›$_0$, provided ‹var›$_0$ has no free occurrence in ‹term›$_0$.

This is an observation made by talking *about* the Lambda Calculus. It is not possible to derive this observation *within* the rules of the *β-reductor* shown above. If we want to include η-contraction, the above rule must be added to the semantics of the Lambda Calculus. We extend the *β-reductor* such as to include η-contraction

```
#lang racket                                                ; File beta-eta-reductor.rkt
(require "curried-lc-grammar.rkt" "beta-reductor.rkt" "free-vars.rkt" redex)
(provide βη-reductor βηnf?)

(define βη-reductor
 (extend-reduction-relation β-reductor curried-lc-grammar
  (--> (in-hole ‹subterm› (λ (‹var›_0) (‹term› ‹var›_0)))
       (in-hole ‹subterm› ‹term›)
       (side-condition (not (term (Var-free-in? ‹var›_0 ‹term›))))
       "η")))

(define-metafunction curried-lc-grammar βηnf? : ‹term› -> ‹bool›
 ((βηnf? ‹term›) ,(null? (apply-reduction-relation βη-reductor (term ‹term›)))))
```

```
#lang racket                                           ; File test-beta-eta-reductor.rkt
(require redex "beta-eta-reductor.rkt" "tracer.rkt")

(parameterize ((reduction-steps-cutoff 100))
 (tracer βη-reductor
  ((x y z)                                (x y z))
  ((λ (x) x)                              (λ (y) y))
  ((λ (x) (x x))                          (λ (y) (y y)))
  ((λ (x) (F x))                          F)
  ((λ (x x) (F x))                        (λ (x) F))
  ((λ (x) (F x x))                        (λ (x) (F x x)))
  (((λ (x) (y x)) x)                      (y x))
  (((λ (x y) x) y) (λ (p)                 y))
  (((λ (x) (λ (y) (x y))) z)              z)
  (((λ (x y z) (x y z)) a b c)            (a b c))
  (((λ (x) (x x)) (λ (x) (x x)))          #f)
  ((λ (p) ((λ (x) (x x)) (λ (x) (x x))))  #f)
  (((((λ (z) (z z)) (λ (x y) (x (x y)))) a) z)  (a (a (a (a z)))))))

(parameterize ((reduction-steps-cutoff 10))
 (tracer βη-reductor (((λ (x) (x x x)) (λ (x) (x x x))) #f)))
```

Normal forms and reducibility

Contents

A normal form is a lambda term without redices, id est, a term of which no subterm is a redex. Simpler said, a normal form is a lambda term that cannot be reduced any further. It can be regarded as the final result of a computation. β-reduction consists of zero or more subsequent β-contractions, where needed with α-conversion, until no more β-redices are left. The result is a β-normal form. It may still contain η-redices. βη-reduction consists of zero or more subsequent β- and η-contractions, where needed with α-conversion. The result is a βη-normal form. Not all lambda terms have a normal form (like some programs in real life programs never produce a result) A term is called reducible if it is a normal form or can be reduced to a normal form. If a lambda term has a normal form, the latter is uniquely defined modulo α-congruence. This is a consequence of the Church-Rosser theorem, which holds for both β-reduction and βη-reduction. When running test-beta-eta-reductor.rkt, you see that a lambda term can have zero one or more redices. The Church-Rosser theorem is: if a lambda term M can be reduced to a lambda term N_1 and also to a lambda term N_2, then there is a lambda term *L* such that both N_1 and N_2 can be reduced to *L*. See figure 1. See Barendregt for a proof, or rather material that provides the knowledge to do an exercise that asks to provide the proof.

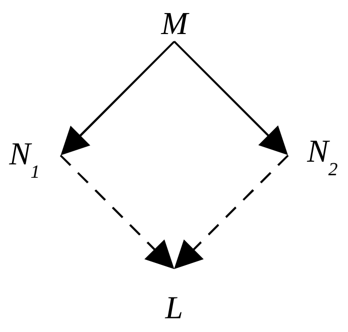


Figure 1: The Church-Rosser theorem

The two solid arrows imply the existence of *L* and the two dotted arrows. Each arrow represents a sequence of zero or more contractions.

The Church-Rosser theorem applies to common arithmetic too. It gives us freedom in the order of computing the value of an arithmetic expression, provided the hierarchy of operators is respected. E.g: (3+4)(5+6) can be computed in many different ways. See file: common-arithmetic.rkt.

Reduction strategies

Contents

The Church-Rosser theorem implies that, in principle, the order of reduction is irrelevant. However, when contracting the underlined redex of the lambda term shown in figure 2, the original lambda term is obtained again, while the lambda term as a whole is a redex too and immediately reduces to the normal form y. The lambda term of figure 3 has the same normal form, but an infinite reduction graph for the underlined subterm. In both figure 2 and figure 3 a strategy is required that sooner or later takes the outer redex that branches down to y.

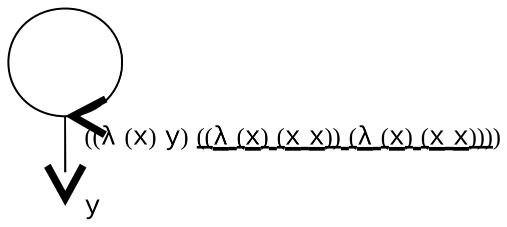
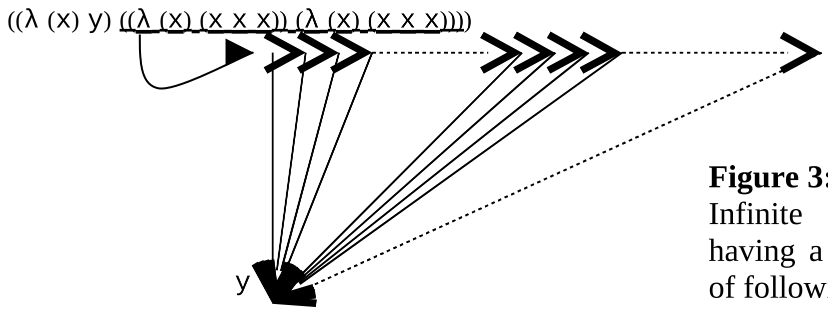


Figure 2: Simple reduction graph

Contraction of the underlined redex results in the very same term and makes the graph contain a cycle. Contraction of the outer redex immediately produces the normal form y.



Ever expanding term when trying to first normalize the underlined irreducible subterm.

Figure 3:
Infinite reduction graph for a term having a normal form. Hence the order of following the branches does matter.

Non deterministic or width first traversal

Contents

Procedure *traces* follows all possible reduction paths by traversing the graph in width first order. See graph traversal and shortest paths.pfd for a simple example of this method. This strategy ensures that given a reducible lambda term, the normal form will be found. It is as though *traces*, when inspecting a lambda term, contracts all its redices in parallel and follows all branches to the reducts simultaneously.

One step or deterministic reduction strategies [Contents]

A one step or deterministic strategy is one that chooses one branch only. The result is a linear path without bifurcations. The branch is determined by a function, say *F*, such that given a non normal form ‹term›, *F*(‹term›) can be obtained from ‹term› by one single contraction. Examples are: normal order, applicative order and random order. Given an initial term, the followed path is: ‹term› → *F*(‹term›) → *F*(*F*(‹term›)) ≡ F^2(‹term›) → *F*(F^2(‹term›)) ≡ F^3‹term› → **...** → F^n‹term› **...**, until a normal form is obtained (or non ending if the reduction strategy fails or the term is irreducible)

Normal order reduction strategy [Contents]

For the normal order strategy function *F* selects the leftmost redex, id est, the redex whose left parenthesis is left of those of all other redices. When given a reducible term, the normal order strategy does lead to the normal form within a finite number of reduction steps. Let *N* be the normal form of a reducible term *M*. If this strategy would not find the normal form, then a reduction from *M* to *N* might require one or more steps involving the contraction of a non leftmost redex. Let $M_i \rightarrow M_{i+1}$ be such a non leftmost reduction step. Then M_{i+1} still has the leftmost redex of M_i, possibly internally contracted. More non leftmost contractions may follow, possibly affecting the subterms of the still not contracted leftmost redex. A non leftmost contraction cannot contract the still uncontracted, possibly internally modified leftmost redex. Eventually we must contract this leftmost redex in order to obtain the normal form. We may as well do so for the step $M_i \rightarrow M_{i+1}$.

Applicative order reduction strategy [Contents]

The normal order reduction strategy is a safe one, but not always efficient. For example:

$\underline{\text{((λ (x) (x x)) ((λ (x) x) (λ (x) x)))}} \rightarrow$
$\text{(}\underline{\text{((λ (x) x) (λ (x) x))}}\text{ ((λ (x) x) (λ (x) x)))} \rightarrow$
$\underline{\text{((λ (x) x) ((λ (x) x) (λ (x) x)))}} \rightarrow$
$\underline{\text{((λ (x) x) (λ (x) x))}} \rightarrow \text{(λ (x) x)}$

In each line the redex being contracted is underlined. We see that subterm ((λ (x) x) (λ (x) x)) is reduced twice. This is avoided by the applicative order strategy. For this strategy we shall use β-reduction only (without η-contractions). We still consider the leftmost redex, but before contracting it, the actual argument is reduced. This gives:

$\text{((λ (x) (x x)) }\underline{\text{((λ (x) x) (λ (x) x))}}\text{)} \rightarrow \text{((λ (x) (x x)) (λ (x) x))} \rightarrow \text{((λ (x) x) (λ (x) x))} \rightarrow \text{(λ (x) x)}$

This is only one step less than for normal order reduction, but in more complicated examples the gain of efficiency can be huge. A drawback of applicative order reduction is that it does not always find the normal form. Figures 2 and 3 show examples of reducible terms that cannot be reduced to their normal forms in applicative order. The latter requires that an abstraction is never applied to an irreducible actual argument.

Random order reduction strategy

The next redex to be contracted can be chosen randomly from all redices. With a good random selector this strategy almost surely leads to the normal form, provided the initial term is reducible. The reason is that with a good random selector, there is a fair chance that after ignoring a leftmost redex for a while, it will eventually be chosen.

Normal order β-reduction [Contents]

In this section a program is shown for normal order β-reduction.

```
#lang racket                                        ; File normal-order-beta-reductor.rkt

(require redex "tracer.rkt")
(require redex "curried-lc-grammar.rkt")
(require (only-in "beta-reductor.rkt" Subst))
(provide normal-order-β-reductor βNf? β-redex?)
```

```
(define normal-order-β-reductor
 (reduction-relation curried-lc-grammar
  (--> ‹term› (Contract-once ‹term›)
   (side-condition (not (term (βNf? ‹term›)))))))

(define-metafunction curried-lc-grammar βNf? : ‹term› -> ‹bool›
 ((βNf? ‹var›) #t)
 ((βNf? (λ (‹var›) ‹term›)) (βNf? ‹term›))
 ((βNf? (‹term›_0 ‹term›_1))
 ,(and
   (not (term (Abstr? ‹term›_0)))
   (term (βNf? ‹term›_0))
   (term (βNf? ‹term›_1)))))

(define-metafunction curried-lc-grammar β-redex? : ‹term› -> ‹bool›
 ((β-redex? ((λ (‹var›) ‹term›_0) ‹term›_1)) #t)
 ((β-redex? any) #f))

(define-metafunction curried-lc-grammar Contract-once : ‹term› -> ‹term›
 ((Contract-once (side-condition ‹term› (term (β-redex? ‹term›))))
  (Contract-β-redex ‹term›))
 ((Contract-once (λ (‹var›) ‹term›)) (λ (‹var›) (Contract-once ‹term›)))
 ((Contract-once
   ((side-condition ‹term›_0 (not (term (βNf? ‹term›_0)))) ‹term›_1))
  ((Contract-once ‹term›_0) ‹term›_1))
 ((Contract-once (‹term›_0 ‹term›_1)) (‹term›_0 (Contract-once ‹term›_1))))

(define-metafunction curried-lc-grammar Contract-β-redex : ‹term› -> ‹term›
 ((Contract-β-redex ((λ (‹var›_0) ‹term›_0) ‹term›_1))
  (Subst (‹var›_0 ‹term›_1) ‹term›_0)))

(printf "~a~n" "test-normal-order-β-reductor")

(tracer normal-order-β-reductor
 (((((λ (x y) x) y) z) ((((λ (z) (z z)) (λ (x y) (x (x y)))) a) z))
  (y (a (a (a (a z))))))
 (((((λ (z) (z z)) (λ (x y) (x (x y)))) a) z) (a (a (a (a z)))))
 ((λ (x) (y x)) (λ (x) (y x)))
 (((λ (x) x) y) y)
 (((λ (x y z) (x y z)) a b c) (a b c))
 (((λ (z) ((a b) z)) c) (a b c))
 (((λ (x) (x x)) (λ (x) (x x))) #f))
```

```
(redex-check curried-lc-grammar ‹term›
 (let*
  ((nf? (term (βNf? ‹term›)))
   (redices
    (apply-reduction-relation
     normal-order-β-reductor
     (term ‹term›))))
  (cond
   ((and (pair? redices) (pair? (cdr redices)))
    (error 'test "more than one left most redex for term ~s"
     (term ‹term›)))
   ((and nf? (pair? redices))
    (error 'test "(βNf? ~s)->#t but reductor found redex ~s."
     (term ‹term›) (car redices)))
   ((and (not nf?) (null? redices))
    (error 'test "(βNf? ~s)->#f but reductor found no redex ~s."
     (term ‹term›)))))
 #:attempts 100000 #:retries 100)
```

Equality relation on lambda terms

Contents

The equality relation on lambda terms is the compatible, reflexive, symmetric and transitive closure of α-congruence and the β- or βη-reduction-relation. This means the following list of axioms:

0	‹term›$_0$ $\rightarrow^{\alpha}$ ‹term›$_1$ $\Rightarrow$ ‹term›$_0$ = ‹term›$_1$	α-congruence.
1	‹term›$_0$ $\rightarrow^{\beta}$ ‹term›$_1$ $\Rightarrow$ ‹term›$_0$ = ‹term›$_1$	Single step β-contraction.
2	‹term›$_0$ = ‹term›$_0$	Reflexive closure.
3	‹term›$_0$ = ‹term›$_1$ $\Rightarrow$ ‹term›$_1$ = ‹term›$_0$	Symmetric closure.
4	‹term›$_0$ = ‹term›$_1$ $\wedge$ ‹term›$_1$ = ‹term›$_2$ $\Rightarrow$ ‹term›$_0$ = ‹term›$_2$	Transitive closure.
5	‹term›$_0$ = ‹term›$_1$ $\Rightarrow$ (λ (‹var›$_0$) ‹term›$_0$) = (λ (‹var›$_0$) ‹term›$_1$)	Compatibility with abstraction.
6	‹term›$_0$ = ‹term›$_1$ $\Rightarrow$ (‹term›$_0$ ‹term›$_2$) = (‹term›$_1$ ‹term›$_2$)	Compatibility with application.
7	‹term›$_0$ = ‹term›$_1$ $\Rightarrow$ (‹term›$_2$ ‹term›$_0$) = (‹term›$_2$ ‹term›$_1$)	Compatibility with application.

Axiom 5 is also called 'rule ξ'. We may also add:

8 (‹term›$_0$ ‹var›$_0$) = (‹term›$_1$ ‹var›$_0$) $\Rightarrow$ ‹term›$_0$ = ‹term›$_1$ Extensionality.
provided ‹var›$_0$ has no free occurrences in ‹term›$_0$ nor in ‹term›$_1$.

Addition of extensionality (axiom 8) is equivalent with adding η-conversion.

Proof: extensionality $\Rightarrow$ η-conversion
If ‹var›$_0$ has no free occurrences in ‹term›$_0$, then we have ((λ (‹var›$_0$) (‹term›$_0$ ‹var›$_0$)) ‹var›$_1$) = (‹term›$_0$ ‹var›$_1$). Now extensionality gives (λ (‹var›$_0$) (‹term›$_0$ ‹var›$_0$)) = ‹term›$_0$. QED.

Proof: η-conversion $\Rightarrow$ extensionality
Suppose (‹term›$_0$) ‹var›$_0$) = (‹term›$_1$) ‹var›$_0$), where ‹var›$_0$ has no free occurrences in ‹term›$_0$ nor in ‹term›$_1$. Then by rule ξ: (λ (‹var›$_0$) (‹term›$_0$ ‹var›$_0$)) = (λ (‹var›$_0$) (‹term›$_1$ ‹var›$_0$)). η-contraction of the left and right hand sides gives: ‹term›$_0$ = ‹term›$_1$. QED.

With addition of extensionality, the Lambda Calculus has the following property of completeness: let *M* and *N* be two reducible lambda terms and *M*≠*N*, which can be decided by checking that their normal forms are not α-congruent. Then addition of *M*=*N* to axioms 1-8 produces a degenerated calculus in which all lambda terms are equal to each other. It is like adding the axiom 0=1 to number theory, making all numbers equal to each other. See Barendregt for proof.

Applicative order Y combinator in Racket

Contents

Y combinators are used for the creation of recursive functions without the use of forms like *define*, *letrec*, named *let* or other forms that bind variables before their values have been established. Before presenting a more formal introduction to Y combinators, this section shows a derivation in pure Racket. This derivation requires more text than the introduction in the next section. Nevertheless it may be a good preparation. [(8)]

```
#lang racket                                        ; File y-derivation-in-Racket.rkt
```

Preparation of a recursive function without *define*, named *let*, *letrec* or any form alike. First look at a simple way to define function factorial **using *define***:

```
; Version 1
(define factorial-1 (λ (n) (if (zero? n) 1 (* n (factorial-1 (sub1 n))))))
(factorial-1 4)
```

Add a layer of abstraction (why do we want an extra layer of abstraction? Read on and see. Moreover, in Lambda Calculus there is no other way to proceed)

```
; Version 2
(define make-factorial-2
 (λ ()
  (λ (n) (if (zero? n) 1 (* n ((make-factorial-2) (sub1 n)))))))

(define factorial-2 (make-factorial-2))
(factorial-2 4)
```

Function *make-factorial-2* has no argument, but we can give it a dummy argument. Why this is done will become clear within three steps. Moreover, Lambda Calculus requires every function to have an argument.

```
; Version 3
(define make-factorial-3
 (λ (dummy-arg)
  (λ (n) (if (zero? n) 1 (* n ((make-factorial-3 dummy-arg) (sub1 n)))))))

(define factorial-3 (make-factorial-3 "dummy"))
(factorial-3 4)
```

In version 3 the dummy argument is never used. It is just passed around. It does not matter what actual argument we provide for the dummy formal argument. We may as well pass *make-factorial-3* to itself. Why we want to do that? This will become clear within two more steps. The dummy argument can be renamed such as to indicate that its value will be the factorial-maker itself.

```
; Version 4
(define make-factorial-4
 (λ (make-factorial)
  (λ (n) (if (zero? n) 1 (* n ((make-factorial-4 make-factorial) (sub1 n)))))))

(define factorial-4 (make-factorial-4 make-factorial-4))
(factorial-4 4)
```

In *make-factorial-4* we have subexpression `(make-factorial-4 make-factorial)` but *make-factorial-4* is called with itself for its actual argument `make-factorial` The two variables have the same value. Hence we can replace the subexpression by `(make-factorial make-factorial)`.

8 With thanks to Daniel P. Friedman and Matthias Felleisen permitting me to use an adapted version of their approach in 'The Little LISPer' c.q. 'The Little Schemer'.

```
; Version 5
(define make-factorial-5
 (λ (make-factorial)
  (λ (n) (if (zero? n) 1 (* n ((make-factorial make-factorial) (sub1 n)))))))

(define factorial-5 (make-factorial-5 make-factorial-5))
(factorial-5 4)
```

Now we see why it made sense to apply *make-factorial-n* to itself. *Make-factorial-5* no longer is recursive, or is it? Anyway, function `(λ (make-factorial) ...)` no longer has a recursive reference to variable `make-factorial-5`.

Function `(λ (n) ...)` contains the subexpression `(make-factorial make-factorial)`, where we would like to write `factorial`. This can be done by binding the value of the subexpression to a variable of this name.

File make-factorial-6.rkt

```
; Version 6
;(define make-factorial-6
; (λ (make-factorial)
;  ((λ (factorial)
;     (λ (n) (if (zero? n) 1 (* n (factorial (sub1 n))))))
;   (make-factorial make-factorial))))

;(define factorial-6 (make-factorial-6 make-factorial-6))
;(factorial-6 4)
```

Version 6 has been commented out for good reason. Open the file and use Debug to see what happens. Do not run it if you don't want memory to be set in fire. Function *make-factorial-6* immediately applies *make-factorial* to itself. But we have given *make-factorial-6* as the value for `make-factorial`. Hence *make-factorial-6* is applied to itself, but this application makes *make-factorial-6* apply itself to itself immediately again and again and again and again **...** . What has happened during the transformation from version 5 to version 6? In version 5 the self application `(make-factorial make-factorial)` is in the else-part of an if-form. It is evaluated only when needed. At the bottom of the recursion, it is not evaluated. However, in version 6 the self-application is evaluated unconditionally for it has been lifted out of the if-form. We have to postpone the self application until it is needed in the else-part of the if-form. This can be done by wrapping it in a lambda-form. If we have an expression *F* that produces a function of one argument, the function (λ (x) (*F* x)) exactly does the same, provided evaluation of *F* produces no side effects and *F* does not refer to variable `x`. If *F* does depend on variable `x`, we can take (λ (y) (*F* y)) or choose any other formal argument that does not conflict with the variables used in *F*. *F* is not evaluated before (λ (x) (*F* x)) is called. That is precisely what a function is for. It is a recipe. We don't have to start cooking right away. We can wait until we are hungry. The appetite is in the else-part of the if-form.

```
; Version 7
(define make-factorial-7
 (λ (make-factorial)
  ((λ (factorial) (λ (n) (if (zero? n) 1 (* n (factorial (sub1 n))))))
   (λ (n) ((make-factorial make-factorial) n)))))

(define factorial-7 (make-factorial-7 make-factorial-7))
(factorial-7 4)
```

Now we can put the definition of *make-factorial-7* into the self application that defines *factorial-7*:

```
; Version 8
(define factorial-8
 ((λ (f) (f f)) ; does the self application.
  (λ (make-factorial)
   ((λ (factorial) (λ (n) (if (zero? n) 1 (* n (factorial (sub1 n))))))
    (λ (n) ((make-factorial make-factorial) n))))))

(factorial-8 4)
```

Function (λ (factorial) **...**) does not refer to make-factorial. Therefore it can be lifted out. We call it m.

```
; Version 9
(define factorial-9
 ((λ (m)
   ((λ (f) (f f))
    (λ (make-factorial) (m (λ (n) ((make-factorial make-factorial) n))))))
  (λ (factorial) (λ (n) (if (zero? n) 1 (* n (factorial (sub1 n))))))))

(factorial-9 4)
```

Now function (λ (m) **...**) is independent from the algorithm used for the factorial function. It can be used for other functions too. Therefore it is better to change the name make-factorial into a more general one. We choose g.

```
; Version 10
(define factorial-10
 ((λ (m) ((λ (f) (f f)) (λ (g) (m (λ (n) ((g g) n))))))
  (λ (factorial) (λ (n) (if (zero? n) 1 (* n (factorial (sub1 n))))))))

(factorial-10 4)

(define length-10
 ((λ (m) ((λ (f) (f f)) (λ (g) (m (λ (n) ((g g) n))))))
  (λ (length) (λ (lyst) (if (null? lyst) 0 (add1 (length (cdr lyst))))))))

(length-10 '(a b c d e f g h i j k l m n o p q r s t u v w x))
```

We no longer need *define*:

```
; Version 11
(((λ (m) ((λ (f) (f f)) (λ (g) (m (λ (n) ((g g) n))))))
  (λ (factorial) (λ (n) (if (zero? n) 1 (* n (factorial (sub1 n)))))))
 4)

(((λ (m) ((λ (f) (f f)) (λ (g) (m (λ (n) ((g g) n))))))
  (λ (length) (λ (lyst) (if (null? lyst) 0 (add1 (length (cdr lyst)))))))
'(a b c d e f g h i j k l m n o p q r s t u v w x))
```

Function (λ (m) **...**) is usually called ‘Y’, in this case an ‘applicative-order Y-combinator’. Applicative-order, because it is suited to applicative order evaluation as in Racket. This means that an actual argument is evaluated before its value is passed to the function that needs it. It also is important that a function does not evaluate its body until it is called. Well, it can’t, because it needs values for its formal arguments. However, it must not even evaluate parts of its body that do not depend on the values provided for the formal arguments. Racket satisfies this requirement, although an optimizer may pre-evaluate parts of the body, but only when it can prove that the optimization does no harm. Y is a combinator because it only contains references to variables that are bound within Y. That is the definition of a combinator: a lambda term without free variables.

```
; Version 12
(let*
 ((Y (λ (m) ((λ (f) (f f)) (λ (g) (m (λ (n) ((g g) n)))))))
  (factorial (Y (λ (factorial) (λ (n) (if (zero? n) 1 (* n (factorial (sub1 n))))))))))
 (factorial 4))

(let*
 ((Y (λ (m) ((λ (f) (f f)) (λ (g) (m (λ (n) ((g g) n)))))))
  (length (Y (λ (length) (λ (lst) (if (null? lst) 0 (add1 (length (cdr lst))))))))))
 (length '(a b c d e f g h i j k l m n o p q r s t u v w x)))
```

Y of version 12 is not the only possible way to write it. For example it can also be written as:

```
; Version 13
(define Y-13
 (λ (m)
  ( (λ (f) (m (λ (x) ((f f) x))))
    (λ (f) (m (λ (x) ((f f) x)))))))

((Y-13 (λ (factorial) (λ (n) (if (zero? n) 1 (* n (factorial (sub1 n))))))) 4)

((Y-13 (λ (length) (λ (lst) (if (null? lst) 0 (add1 (length (cdr lst)))))))
'(a b c d e f g h i j k l m n o p q r s t u v w x))
```

or as:

```
; Version 14 (9)
(define Y-14
 ((λ (x) (x x x x x x x x x x x x x x x x x x x x x x x x x x x))
  (λ (a b c d e f g h i j k l m n o p q s t u v w x y z) ; Notice that r is not included in this alpha-
bet.
    (λ (r) (λ (u) ((r ((t h i s i s a f i x e d p o i n t c o m b i n a t o) r)) u))))))

((Y-14 (λ (factorial) (λ (n) (if (zero? n) 1 (* n (factorial (sub1 n))))))) 4)
((Y-14 (λ (length) (λ (lst) (if (null? lst) 0 (add1 (length (cdr lst)))))))
'(a b c d e f g h i j k l m n o p q r s t u v w x))
```

or as:

```
; Version 15
(define Y-15
 ((λ (x) (x x x x x x x x x x x x x x x x x x x x x x x x x x x))
  (λ (a b c d e f g h i j k l m n o p q s t u v w x y z)
    (λ (r) (λ (u) ((r ((x x x x x x x x x x x x x x x x x x x x x x x x x x x) r)) u))))))

((Y-15 (λ (factorial) (λ (n) (if (zero? n) 1 (* n (factorial (sub1 n))))))) 4)

((Y-15 (λ (length) (λ (lst) (if (null? lst) 0 (add1 (length (cdr lst)))))))
'(a b c d e f g h i j k l m n o p q r s t u v w x))
```

When you run this program, all results should be `24`. Notice that an applicative-order Y-combinator can be used in normal order evaluators too. Such an evaluator does not evaluate an actual argument before passing it to a function. The reverse is not true. Normal-order Y-combinators cannot be used for an applicative order evaluator.

In the above derivation we have used some constants and primitive functions like *zero?*, *null?*, *sub1*, *add1*, etc. Later we shall see how they can be expressed in Lambda Calculus. We also used if-forms. *If* too can be expressed in Lambda Calculus.

9 This version is an adapted one of an example in Barendregt.

Fixed point combinators, self-reference and recursion Contents

We return to Lambda Calculus. Consider the term (λ (m) ((λ (f) (m (f f))) (λ (f) (m (f f))))). We abbreviate it as '**Y**'. Let ‹term› be an arbitrary lambda term without free occurrences of m and n. If it has, we can α-convert **Y** such that it does not contain any variable occurring free in ‹term›. Below bold underlined is substituted in thin underlined:

1	(**Y** ‹term›) =	Write **Y** in full.
2	((λ (m) ((λ (f) (m (f f))) (λ (f) (m (f f))))) ‹term›) =	Substitute ‹term› for m.
3	((λ (f) (‹term› (f f))) (λ (f) (‹term› (f f)))) =	Substitute (λ (f) (‹term› (f f))) for f.
4	(‹term› ((λ (f) (‹term› (f f))) (λ (f) (‹term› (f f))))) =	Because line 3 = line 2.
5	(‹term› ((λ (m) ((λ (f) (m (f f))) (λ (f) (m (f f))))) ‹term›)) =	Because line 2 = line 1.
6	(‹term› (**Y** ‹term›))	

For this reason **Y** is called a fixed point combinator. A combinator is a lambda term without free variables. Indeed, **Y** has no free variables. Given a term *M*, **Y** produces a term *N* ≡ (**Y** *M*) such that (*M N*) = *N*. Therefore *N* is called a fixed point of *M*. In general: if *f*(*x*) = *x*, then *x* is called a fixed point of *f*. Now: (**Y** *M*) = (*M* (**Y** *M*)) = (*M* (*M* (**Y** *M*))) = (*M* (*M* (*M* (**Y** *M*)))) = (*M* (*M* (*M* (*M* (**Y** *M*))))), etc. Hence (**Y** *M*) is equal to a term that applies *M* as many times to (**Y** *M*) as we may wish. Now:

7 (**Y** (λ (‹var›) ‹body›)) =
8 ((λ (‹var›) ‹body›) (**Y** (λ (‹var›) ‹body›))) =
9 ‹body› in which (**Y** (λ (‹var›) ‹body›)) is substituted for ‹var› =
10 ‹body› in which ‹var› is replaced by a term equal to the updated ‹body› because lines 7-9 tell us that (**Y** (λ (‹var›) ‹body›)) is equal to the updated ‹body›: self-reference!

We can use this fact for the construction of lambda terms representing recursive functions:

11 (**Y** (λ (‹var›$_0$) (λ (‹var›$_1$) ‹body›))) =
12 (λ (‹var›$_1$) ‹body›) in which ‹var›$_0$ is bound to a term equal to (λ (‹var›$_1$) ‹body›): recursion!

Hence in order to prepare a recursive function of the name ‹name›, formal argument ‹formal-arg› and body ‹body›, we write (**Y** (λ (‹name›) (λ (‹formal-arg›) ‹body›))). **Y** cannot be used in Racket. For example:

```
(((λ (m) ((λ (f) (m (f f))) (λ (f) (m (f f))))) (λ (!) (λ (n) (if (zero? n) 1 (* n (! (sub1 n))))))) 4)
```

as an attempt to compute the factorial of 4, does not work. Copy the above expression into DrRacket's definition window and use Debug to see what happens. The reason that the above factorial function does not work is that Racket evaluates in applicative order. **Y** is not suited for this strategy of reduction. This has already been discussed in the previous section. For applicative order reduction use (λ (m) ((λ (f) (m (f f))) (λ (f) (m (λ (x) ((f f) x)))))).

Undecidability Contents

Write **T** ≡ (λ (x y) x) and **F** ≡ (λ (x y) y). We use **T** to represent true and **F** to represent false. A lambda term, say *P*, is a predicate if for every arbitrary lambda term *M*, either (*P M*) = **T** or (*P M*) = **F**, or (*P M*) has no normal form. *P* is complete if (*P M*) has a normal form for every arbitrary term *M*, meaning that (*P M*) is either **T** or **F**. *P* is not trivial if there are two terms *T* and *F* such that (*P T*) = **T** and (*P F*) = **F**.

Theorem

There is no non trivial complete predicate.

Proof

The theorem is proven by reductio ad absurdum. Suppose that *P* is a non trivial complete predicate. Then there should be two terms *T* and *F* such that (*P T*) = **T** and (*P F*) = **F**. Now define:

$N \equiv (\lambda\ (x)\ (P\ x\ F\ T))$ *N* for 'negation'
$A \equiv (\mathbf{Y}\ N) = (N\ (\mathbf{Y}\ N)) \equiv (N\ A)$ *A* for 'absurdum' How can *A* be equal to its negation?

Because *P* is supposed to be complete we expect either: $(P\ A) = \mathbf{T}$ or $(P\ A) = \mathbf{F}$. However:

$(P\ A) = \mathbf{T} \Rightarrow \mathbf{T} = (P\ A) = (P\ (N\ A)) = (P\ (P\ A\ F\ T)) = (P\ (\mathbf{T}\ F\ T)) = (P\ F) = \mathbf{F}$, a contradiction.
$(P\ A) = \mathbf{F} \Rightarrow \mathbf{F} = (P\ A) = (P\ (N\ A)) = (P\ (P\ A\ F\ T)) = (P\ (\mathbf{F}\ F\ T)) = (P\ T) = \mathbf{T}$, a contradiction.

Hence *P* does not exist. QED.

Corollary
From the above theorem it immediately follows that there is no complete non trivial predicate that can decide whether or not its argument has a normal form (halting problem) It means that there is no program that for every arbitrary program can decide whether or not the latter will produce a result within a finite number of steps. For programs that need data, read 'combination of program and data' or 'combination of program with arbitrary data'.

Corollary
There is no term *E* such that $(E\ M\ N) = \mathbf{T}$ if $N = M$ and $(E\ M\ N) = \mathbf{F}$ if $N \neq M$, for this would mean that for every *M* the term $(E\ M)$ would have to be a complete non trivial predicate. If we restrict the domain to reducible terms *M* and *N*, then it is possible to construct *E* such that $(E\ M\ N) = \mathbf{T}$ if $N = M$ and $(E\ M\ N) = \mathbf{F}$ if $N \neq M$, for example by reducing *M* and *N* and checking whether or not their normal forms are α-congruent. However, the set of reducible terms is not well defined for we have no predicate for that property.

One can even prove something stronger than the above theorem. The range of a lambda term *M* is the set $\{(M\ N){:}\ N \in \mathbf{\Lambda}\}$, where $\mathbf{\Lambda}$ is the set of all lambda terms. A range either consists of one single element or is infinite. Hence the above theorem does not depend on the actual choices of **T** and **F**. See Barendregt for a proof.

Applicative order reduction

Contents

Normal order reduction is effective, but not efficient. For example: ((λ (x) (x x)) ‹argument›) → (‹argument› ‹argument›). This means that both occurrences of the ‹argument› must yet be reduced. This can be avoided by reducing the ‹argument› before contracting the outermost redex. However, this strategy does not always find the normal form of reducible terms. For example:

$((\lambda\ (x)\ y)\ \underline{((\lambda\ (x)\ (x\ x))\ (\lambda\ (x)\ (x\ x)))}) \xrightarrow{\text{applicative-order}} ((\lambda\ (x)\ y)\ \underline{((\lambda\ (x)\ (x\ x))\ (\lambda\ (x)\ (x\ x)))})$ because
$\underline{((\lambda\ (x)\ (x\ x))\ (\lambda\ (x)\ (x\ x)))} \rightarrow \underline{((\lambda\ (x)\ (x\ x))\ (\lambda\ (x)\ (x\ x)))}$

Yet the term is reducible. Its normal form is `y`. See figure 2. This means that the applicative order strategy does not find the normal forms of all reducible terms. We have to limit the terms to be reduced to those in which a function is never applied to an irreducible argument. Another problem is the if-form. It is essential that in (`if` ‹test› ‹then-part› ‹else-part›) the two alternatives are not evaluated before the test has decided which one to choose. Therefore *if* is necessarily a syntactic form in Racket. Nevertheless we can write the if-form in terms of functions only. We only have to postpone the evaluation of the two alternatives. This can be done by wrapping them in abstractions:

```
((‹test› (λ (‹ignore›) ‹then-part›) (λ (‹ignore›) ‹else-part›)) ‹to-be-ignored›)
```

The test, which must yield either **T** or **F**, selects one of the two abstractions. The selected one is applied to ‹to-be-ignored›, which indeed is ignored.

Lazy evaluator

Contents

Lazy reduction is yet another deterministic reduction strategy. When a function is applied to an actual argument, as in ((λ (‹formal-arg›) ‹body›) ‹actual-arg›), we can attach a label to each free occurrence of the formal argument within the body. The label refers to the still uncontracted actual argument. For each application a fresh label must be chosen. Now the body can be reduced. If during the reduction of the body the formal argument is referenced, the corresponding actual argument is reduced and its normal form substituted for every variable that has the same label. This has the advantage of postponing the reduction of actual arguments that will never be needed, but avoids multiple reduction of the same actual argument. For reasons of efficiency a lazy evaluator will be used in the remainder of this essay. It is like Racket, but with the following modifications:

0 Each uncurried term is evaluated as though fully curried. This allows uncurried shorthand.
1 Free variables are allowed and are self-evaluating.
2 Abstractions are evaluated to Racket procedures.
3 Evaluation of an application does not necessarily involve the evaluation of the actual argument. The actual argument is wrapped in a promise and the latter is passed to the procedure. The Racket procedures mentioned in item 2 know that their argument may be a promise rather than the value itself.

File lazy-reductor.rkt shows the implementation. It is not a reductor in proper sense, because it may return a Racket procedure, which is not a lambda term.

Syntax *ev* accepts a term, evaluates it and returns the result. Syntax *def* allows the preparation of environments to be used during subsequent calls to syntax *ev*. (def ‹var› ‹term›) differs from (define ‹var› ‹term›) because *def* promises to evaluate the term in an environment that does not include the new binding. It does not allow forward or recursive references to any variable. *Def* cannot construct self reference or recursion as can be done with *define* or *letrec*. *Def* may be used to redefine a variable. The new definition shadows the old one, but does not affect any definition that was made before the variable was shadowed. A sequence of def-forms preceding an ev-form has the same effect as (ev (let* ((‹var› ‹term›) **...**) ‹term›)), where the let*-form is expanded in terms of lambda, which can be done as follows:

```
#lang racket                                    ; File let-star-in-terms-of-lambda.rkt

(define-syntax let*
 (syntax-rules ()
  ((let* () . body) ((λ () . body)))
  ((let* ((var expr) binding ...) . body)
   ((λ (var) (let* (binding ...) . body)) expr))))
```

```
#lang racket                                    ; File lazy-evaluator.rkt
```

Procedure: (ev-proc ‹term›) → value of the ‹term›.
Curries the term and evaluates it in the current environment. Free variables are allowed and self-evaluating. The operator of an application must be a function. The actual argument is not evaluated, but wrapped in a promise. The value of an abstraction is a procedure of one argument. If this procedure receives a promise for its argument, the promise will be forced when needed.

Syntax: (ev ‹term›) ⇒(ev-proc (term ‹term›))

Procedure: (def-proc ‹var› ‹term›)
Adds a binding (‹var› ‹value›) to the current environment, where the value is a promise to evaluate the term in the environment that was the current one before adding the binding. It is allowed to shadow an existing binding. This does not affect the values of any bindings that may already exist.

Syntax: (def ‹var› ‹term›) ⇒(def-proc '‹var› '‹term›)

Procedure (list-env) → list of variables
Returns the list of all variables bound in the current environment. Shadowed variables are included.

Procedure: `(clear-env)` → void
Clears the current environment.

```
(provide ev-proc ev def-proc def list-env list-vars clear-env)
(define (ev-proc x) (ev-aux x env))
(define (def-proc var x) (extend-env var (let ((e env)) (lazy (ev-aux x e)))))
(define-syntax ev (syntax-rules ( ) ((_ x) (ev-proc 'x))))
(define-syntax def (syntax-rules ( ) ((_ var x) (def-proc 'var 'x))))

(define (ev-aux x env)
  (cond
    ((var? x) (force (lookup x env)))
    ((abstr? x) (curry-fun (cadr x) (caddr x) env))
    ((appl? x) (curry-appl (ev-aux (car x) env) (cdr x) env))
    (else (error 'ev-proc "incorrect (sub)term ~s" x))))

(define (abstr? x)
  (and
    (list? x)
    (= (length x) 3)
    (eq? (car x) 'λ)
    (formals? (cadr x))))

(define (formals? x)
  (and
    (list? x)
    (andmap var? x)))

(define (appl? x)
  (and
    (list? x)
    (pair? x)
    (not (eq? (car x) 'λ))))

(define (lookup var env)
  (let ((binding (assq var env)))
    (if binding (cdr binding) var)))

(define (curry-fun vars body env)
  (if (null? vars) (ev-aux body env)
    (let ((var (car vars)) (vars (cdr vars)))
      (λ (x) (curry-fun vars body (cons (cons var x) env))))))

(define (curry-appl op args env)
  (if (null? args) op
    (let ((arg (car args)) (args (cdr args)))
      (curry-appl (op (lazy (ev-aux arg env))) args env))))

(define env '( ))
(define (clear-env) (set! env '( )))
(define (list-vars) (map car env))
(define (list-env) env)
(define (extend-env var value) (set! env (cons (cons var value) env)))
(define (var? x) (and (symbol? x) (not (eq? x 'λ))))
```

Booleans

Contents

The main use of Booleans in a programming language is for Boolean logic and for the test in an if-form in order to choose one out of two options. Therefore we represent true and false by functions of two arguments returning the first c.q. the second argument. Every function with this property is a Boolean. Other values cannot be used as Booleans. In fact the functions will be curried, of course. In our terminology a function of two arguments is a function of one argument producing another function of one argument.

```
#lang racket                                                  ; File booleans.rkt
(require "lazy-evaluator.rkt")

(def True  (λ (x y) x)) ; ≡ (λ (x) (λ (y) x))
(def False (λ (x y) y)) ; ≡ (λ (x) (λ (y) y))
(def If    (λ (x y z) (x y z))) ; A function! Unnecessary, but it may improve readability.
(def And   (λ (x y) (If x y False)))
(def Or    (λ (x y) (If x x y)))
(def Not   (λ (x) (If x False True)))
```

In fact we have `If` ≡ (λ (x y z) (x y z))) ≡ (λ (x) (λ (y) (λ (z) ((x y) z) $=^\eta$ (λ (x) (λ (y) (x y))) $=^\eta$ (λ (x) x). We can as well write:

```
(def Identity (λ (x) x))
(def If Identity) ; Amazing: not only if can be written as a function but even as an identity!
```

```
#lang racket                                                  ; File test-booleans.rkt
(require redex "booleans.rkt" "curried-Racket.rkt")
(test-equal (curried-Racket (True yes no))                   'yes)
(test-equal (curried-Racket (False yes no))                  'no )
(test-equal (curried-Racket (Not False yes no))              'yes)
(test-equal (curried-Racket (Not True yes no))               'no )
(test-equal (curried-Racket (And True True yes no))          'yes)
(test-equal (curried-Racket (And True False yes no))         'no )
(test-equal (curried-Racket (And False True yes no))         'no )
(test-equal (curried-Racket (And False False yes no))        'no )
(test-equal (curried-Racket (Or True True yes no))           'yes)
(test-equal (curried-Racket (Or True False yes no))          'yes)
(test-equal (curried-Racket (Or False True yes no))          'yes)
(test-equal (curried-Racket (Or False False yes no))         'no )
(test-equal (curried-Racket (If True yes no))                'yes)
(test-equal (curried-Racket (If False yes no))               'no )
(test-equal (curried-Racket (If (Not False) yes no))         'yes)
(test-equal (curried-Racket (If (Not True) yes no))          'no )
(test-equal (curried-Racket (If (And True True) yes no))     'yes)
(test-equal (curried-Racket (If (And True False) yes no))    'no )
(test-equal (curried-Racket (If (And False True) yes no))    'no )
(test-equal (curried-Racket (If (And False False) yes no))   'no )
(test-equal (curried-Racket (If (Or True True) yes no))      'yes)
(test-equal (curried-Racket (If (Or True False) yes no))     'yes)
(test-equal (curried-Racket (If (Or False True) yes no))     'yes)
(test-equal (curried-Racket (If (Or False False) yes no))    'no )
(test-results) ; Display: All 24 tests passed.
```

Pairs

Contents

How shall we represent pairs? By functions of course, for we have no other equipment available. The essential property of a pair is that it must allow recollection of its car and its cdr. Therefore a pair will be a function that expects a Boolean for its argument and returns the car or cdr depending on the Boolean.

```
#lang racket                                                  ; File pairs.rkt
(require "booleans.rkt" "lazy-evaluator.rkt")
(def Cons (λ (x y z) (If z x y))) ; ≡ (λ (x y) (λ (z) (If z x y)))
(def Car  (λ (x) (x True  )))
(def Cdr  (λ (x) (x False)))
```

Now indeed:

(Car (Cons ‹x› ‹y›) = (Car (λ (z) (z ‹x› ‹y›)) = ((λ (z) (z ‹x› ‹y›) True) = (True ‹x› ‹y›) = ‹x›
(Cdr (Cons ‹x› ‹y›) = (Cdr (λ (z) (z ‹x› ‹y›)) = ((λ (z) (z ‹x› ‹y›) False) = (False ‹x› ‹y›) = ‹y›

```
#lang racket                                             ; File test-pairs.rkt
(require redex "pairs.rkt" "lazy-evaluator.rkt")
(test-equal (ev (Car (Cons yes no))) 'yes)
(test-equal (ev (Cdr (Cons yes no))) 'no)
(test-results)
```

Numbers

Contents

There are several ways to represent natural numbers in Lambda Calculus. The representation of a number is called a numeral.

Numbers represented by pairs

Contents

Natural number n represented by [n], where: [0] ≡ (λ (x) x) and [n+1] ≡ (Cons False [n])

```
#lang racket                                        ; File pair-numerals.rkt
(require "pairs.rkt" "booleans.rkt" "lazy-evaluator.rkt")

(def Zero (λ (x) x))
(def Zero? Car)
(def Add1 (Cons False))
(def Sub1 Cdr)
```

Now we must show that *Zero*, *Zero?*, *Add1* and *Sub1* form an adequate number system. We do this by proving that the axioms of Peano are satisfied. Here we are reasoning *about* the system!

0 Consider Zero to be a numeral. For every numeral [n], consider (Add1 [n]) to be a numeral too. Also assume that numerals are not made in any other way. This ensures that the axiom of mathematical induction applies.

1 (Zero? [0]) = (Zero? Zero) = (Car Zero) = (Zero True) = ((λ (x) x) True) = True and
(Zero? (Add1 [n])) = (Car (Cons False [n])) = False.
Hence: (Add1 [n]) ≠ Zero.

2 (Sub1 (Add1 [n])) = (Cdr (Cons False [n])) = [n].
Hence: (Add1 [n]) = (Add1 [m]) ⇒[n] = (Sub1 (Add1 [n])) = (Sub1 (Add1 [m])) = [m], id est:
(Add1 [n]) = (Add1 [m]) ⇒[n] = [m]

```
(def Y (λ (m) ((λ (f) (m (f f))) (λ (f) (m (f f))))))
(def + (Y (λ (+ x y) (If (Zero? x) y (If (Zero? y) x (Add1 (Add1 (+ (Sub1 x) (Sub1 y)))))))))
(def - (Y (λ (- x y) (If (Zero? x) Zero (If (Zero? y) x (- (Sub1 x) (Sub1 y)))))))
(def * (Y (λ (* x y) (If (Zero? x) Zero (If (Zero? y) Zero (Sub1 (+ (+ (* (Sub1 x) (Sub1 y)) x) y)))))))
(def = (Y (λ (= x y) (If (Zero? x) (Zero? y) (If (Zero? y) False (= (Sub1 x) (Sub1 y)))))))
```

```
(def > (Y (λ (> x y) (If (Zero? x) False (If (Zero? y) True (> (Sub1 x) (Sub1 y)))))))
(def < (λ (x y) (> y x)))
(def >= (λ (x y) (Not (< x y))))
(def <= (λ (x y) (Not (> x y))))
(def Quotient (Y (λ (Quotient x y) (If (< x y) Zero (Add1 (Quotient (- x y) y))))))
(def Modulo (λ (x y) (- x (* (Quotient x y) y))))
(def Divisor? (λ (x y) (Zero? (Modulo x y))))

(def Even?+Odd? ; a pair of mutually recursive functions.
 (Y
  (λ (Even?+Odd?)
   ((λ (Even? Odd?)
     (Cons (λ (n) (If (Zero? n) True (Odd? (Sub1 n))))
           (λ (n) (If (Zero? n) False (Even? (Sub1 n))))))
    (λ (n) (Car Even?+Odd? n))
    (λ (n) (Cdr Even?+Odd? n))))))

(def Even? (Car Even?+Odd?))       (def Odd? (Cdr Even?+Odd?))

; The following two functions provide conversion between natural numbers of Racket
; and numerals represented by pairs.

(define (number->pair-numeral n)
 (if (zero? n) (ev Zero)
  ((ev (Cons False)) (number->pair-numeral (sub1 n)))))

(define (pair-numeral->number numeral)
 (if (eq? ((ev (λ (numeral) (If (Zero? numeral) yes no))) numeral) 'yes) 0
  (add1 (pair-numeral->number ((ev Sub1) numeral)))))

(provide number->pair-numeral pair-numeral->number)
```

```
#lang racket                                          ; File test-pair-numerals.rkt
(require redex "pair-numerals.rkt" "lazy-evaluator.rkt" redex)

(for ((n (in-range 0 25))) (test-equal (pair-numeral->number (number->pair-numeral n)) n))

(for ((m (in-range 1 11)) #:when #t (n (in-range 1 11)))
 (for-each
  (λ (function check-function)
   (test-equal
    (pair-numeral->number
     ((function (number->pair-numeral m)) (number->pair-numeral n)))
    (check-function m n)))
  (list (ev +) (ev -) (ev *) (ev Modulo) (ev Quotient))
  (list + (λ (m n) (max 0 (- m n))) * modulo quotient)))

(for ((n (in-range 0 10)))
 (test-equal ((((ev Even?) (number->pair-numeral n)) #t) #f) (even? n))
 (test-equal ((((ev Odd?) (number->pair-numeral n)) #t) #f) (odd? n)))

(test-results)
```

```
#lang racket                                                    ; File primes.rkt
(require "lazy-evaluator.rkt" "pair-numerals.rkt" redex)
(def Naturals ((Y (λ (make-naturals n) (Cons n (make-naturals (Add1 n))))) Zero))
```

```
(def filter
 (Y
  (λ (filter pred list)
   (If (pred (Car list)) (Cons (Car list) (filter pred (Cdr list))) (filter pred (Cdr list))))))

(def Primes
 ((Y
   (λ (make-primes number-list)
    (Cons (Car number-list)
     (make-primes
      (filter (λ (n) (Not (Divisor? n (Car number-list)))) (Cdr number-list))))))
  (Cdr (Cdr Naturals))))

(define (list-numbers numbers n)
 (if (zero? n) '()
  (cons (pair-numeral->number ((ev Car) numbers))
   (list-numbers ((ev Cdr) numbers) (sub1 n)))))

(test-equal (list-numbers (ev Naturals) 20)
'(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19))

(define (primes n) ; For testing the result of Primes.
 (let loop ((primes '()) (n 2) (k n))
  (cond
   ((zero? k) (reverse primes))
   ((ormap (λ (p) (zero? (modulo n p))) primes) (loop primes (add1 n) k))
   (else (loop (cons n primes) (add1 n) (sub1 k))))))

(test-equal (list-numbers (ev Primes) 25) (primes 25))
(test-results)
```

Church numerals

Contents

n represented by $\mathbf{c}_n$ ≡ (λ (f) (λ (x) (f^n x))), where (f^0 x) ≡ x and (f^{n+1} x) ≡ (f (f^n x)). Hence a Church numeral is a function that given a function f returns a function f^n of argument x that applies f *n* times to argument x. For example:

$\mathbf{c}_0$ ≡ (λ (f) (λ (x) x))
$\mathbf{c}_1$ ≡ (λ (f) (λ (x) (f x)))
$\mathbf{c}_2$ ≡ (λ (f) (λ (x) (f (f x))))
$\mathbf{c}_3$ ≡ (λ (f) (λ (x) (f (f (f x)))))
$\mathbf{c}_4$ ≡ (λ (f) (λ (x) (f (f (f (f x))))))
$\mathbf{c}_5$ ≡ (λ (f) (λ (x) (f (f (f (f (f x)))))))

Notice that (f (f^n x)) ≡ (f^n (f x)) and more generally (f^n (f^m x)) ≡ (f^{n+m} x). Also notice that in file church-numerals.rkt no Y combinator is used for the preparation of the basic numerical functions.

```
#lang racket                                        ; File church-numerals.rkt
(require redex "lazy-evaluator.rkt" "booleans.rkt")

(define-syntax Church-numeral->number
 (syntax-rules ()
  ((_ numeral) (((ev numeral) (λ (n) (add1 (force n)))) 0))))

(define-syntax test-Church
 (syntax-rules ()
  ((_ numeral expect)
   (test-equal (((ev numeral) (λ (n) (add1 (force n)))) 0) expect))))
```

```
(def C-add1 (λ (n f x) (f (n f x))))
(def C-sub1 (λ (n f x) (n (λ (g h) (h (g f))) (λ (f) x) (λ (f) f))))
(def C-plus (λ (m n f x) (m f (n f x))))
(def C-mult (λ (m n f) (n (m f))))
(def C-expt (λ (m n) (n m)))
(def C-zero? (λ (n) (n (λ (x) false) True)))
(def C-minus (λ (m n) (n C-sub1 m)))
(def C-eq? (λ (m n) (And (C-zero? (C-minus m n)) (C-zero? (C-minus n m)))))
(def C0 (λ (f x) x))
(def C1 (λ (f x) (f x)))
(def C2 (λ (f x) (f (f x))))
(def C3 (λ (f x) (f (f (f x)))))
(def C4 (λ (f x) (f (f (f (f x))))))
(def C5 (λ (f x) (f (f (f (f (f x)))))))
(def C6 (λ (f x) (f (f (f (f (f (f x))))))))
(def C7 (λ (f x) (f (f (f (f (f (f (f x)))))))))
(def C8 (λ (f x) (f (f (f (f (f (f (f (f x))))))))))
(test-Church C0 0)
(test-Church C1 1)
(test-Church C2 2)
(test-Church C3 3)
(test-Church C4 4)
(test-Church C5 5)
(test-Church C6 6)
(test-Church C7 7)
(test-Church C8 8)
(test-Church C9 9)
(test-Church (C-add1 C7) 8)
(test-Church (C-sub1 C7) 6)
(test-Church (C-sub1 C0) 0)
(test-Church (C-plus C5 C7) 12)
(test-Church (C-mult C5 C7) 35)
(test-Church (C-minus C5 C5) 0)
(test-Church (C-minus C7 C5) 2)
(test-Church (C-expt C7 C3) 343)
(test-Church (C-expt C5 C4) 625)
(test-equal (ev (If (C-eq? C7 C5) yes no)) 'no)
(test-equal (ev (If (C-eq? C5 C7) yes no)) 'no)
(test-equal (ev (If (C-eq? C0 C0) yes no)) 'yes)
(test-equal (ev (If (C-eq? C1 C1) yes no)) 'yes)
(test-equal (ev (If (C-eq? C7 C7) yes no)) 'yes)
(test-equal (ev (If (C-zero? (C-minus C5 C5)) yes no)) 'yes)
(test-equal (ev (If (C-zero? (C-minus C5 C7)) yes no)) 'yes)
(test-equal (ev (If (C-zero? (C-minus C5 C4)) yes no)) 'no)

(test-results) ; Displays: All 28 tests passed.
```

Combinatory logic

Contents

Consider the following combinators (lambda terms without free variables):

I ≡ (λ (x) x)		(**I** *A*) = *A*
K ≡ (λ (x y) x)	Hence:	(**K** *A* *B*) = *A*
S ≡ (λ (x y z) ((x z) (y z)))		(**S** *A* *B* *C*) = ((*A* *C*) (*B* *C*))
X ≡ (λ (x) (x **K** **S** **K**))		(**X** *A*) = (*A* **K** **S** **K**)

Hence:

(**X X X**) = ((**X K S K**) **X**) = (((**K K S K**) **S K**) **X**) = (((**K K**) **S K**) **X**) = (**K K X**) = **K**
(**X** (**X X**)) = (**X X K S K**) = (**X K S K K S K**) = (**K K S K S K K S K**) = (**K K S K K S K**) = (**K K K S K**) = (**K S K**) = **S**
(**S K** *A*) = (λ (‹var›) (**S K** *A* ‹var›)) = (λ (‹var›) ((**K** ‹var›) (*A* ‹var›))) = (λ (‹var›) ‹var›) ≡ **I**, assuming that ‹var› is chosen such as to have no free occurrence in *A* and using extensionality accordingly. Hence:

I = (**S K K**) = ((**X** (**X X**)) (**X X X**) (**X X X**))
K = (**X X X**)
S = (**X** (**X X**))

We only need **X** in order to make **K**, **S** and **I** as well. {**X**} is a one point basis for the set of all combinators modulo equality. There are more terms like **X** that form a one point basis. The following scheme shows how abstractions can be eliminated from every term.

(λ (‹var›$_0$) ‹var›$_0$))	= **I**
(λ (‹var›$_0$) ‹term›)	= (**K** ‹term›) if ‹var›$_0$ has no free occurrence in ‹term›
(λ (‹var›) (‹term›$_0$ ‹term›$_1$))	= (**S** (λ (‹var›) (‹term›$_0$ ‹var›)) (λ (‹var›) (‹term›$_1$ ‹var›)))

By repeating these rules on a given term, an equal term is obtained in terms of **I**, **K** and **S**, free variables and application. Furthermore **I**, **K** and **S** can be expressed in **X** such as to obtain a term with **X**, free variables and application only, id est, without abstractions. The transformation is shown in the following program:

```
#lang racket                                                ; File one-point-basis.rkt

(require redex "curry.rkt" "curried-lc-grammar.rkt" "free-vars.rkt")
(require "lazy-evaluator.rkt")
(printf "~a~n" "one-point-basis")

(def-proc 'X let ((K '(λ (x y) x)) (S '(λ (x y z) ((x z) (y z))))) (term (Curry (λ (x) (x ,K ,S ,K)))))
(define-language x-term (‹xterm› (variable-except λ) (‹xterm› ‹xterm›)))
(define-metafunction x-term check-xterm : ‹xterm› -> #t ((check-xterm ‹xterm›) #t))

(define-metafunction curried-lc-grammar Trafo : ‹term› -> ‹term›
  ((Trafo I) (Trafo ((S K) K)))
  ((Trafo K) ((X X) X))
  ((Trafo S) (X (X X)))
  ((Trafo X) X)
  ((Trafo ‹var›) ‹var›)
  ((Trafo (λ (‹var›_0) ‹var›_0)) (Trafo I))
  ((Trafo
     (side-condition (λ (‹var›_0) ‹term›_0) (not (term (Var-free-in? ‹var›_0 ‹term›_0)))))
    ((Trafo K) (Trafo ‹term›_0)))
  ((Trafo (λ (‹var›) (‹term›_0 ‹term›_1)))
    (((Trafo S) (Trafo (λ (‹var›) ‹term›_0))) (Trafo (λ (‹var›) ‹term›_1))))
  ((Trafo (λ (‹var›) ‹term›)) (Trafo (λ (‹var›) (Trafo ‹term›))))
  ((Trafo (‹term›_0 ‹term›_1)) ((Trafo ‹term›_0) (Trafo ‹term›_1))))
```

```
(define-syntax test
 (syntax-rules ()
  ((_ p q)
   (let* ((x (term (Trafo (Curry p)))))
    (printf "~s ->~n" 'p) (pretty-display x) (printf "-> ~s~n~n" 'q)
    (term (check-xterm ,x)) ; Checks Trafo to produce an ‹xterm›.
    (test-equal (ev-proc x) 'q)))))

(test ((λ (x) x) yes)                 yes)
(test ((λ (x y) x) yes no)            yes)
(test ((λ (x y) y) yes no)            no)
(test (X X a b c)                     b)
(test (S K X (S K K) b)               b)
(test (S K K b)                       b)
(test (S K S b)                       b)
(test (K yes no)                      yes)
(test (K I yes no)                    no)
(test (K I I yes)                     yes)
(test (K I I I I yes)                 yes)
(test (S I I K a b c)                 b)
(test (S S S S S S S I I I yes)       yes)
(test (S (S S) (S S) (S S) S S K yes) yes)
(test (X X X X X X X I I yes)         yes)
(test (X (X X) (X X) (X X) X X I I yes)yes)

(test-results)
```

Listing all combinators modulo equality

Contents

Because every combinator (term without free variables) can be expressed in **X** with application but without abstractions and without free variables, it is possible to prepare a combinator that lists all combinators modulo equality. The list is infinite of course. A finite solution is a combinator **E** such that (**E** [n]) or (**E** $\mathbf{c}_n$) is the nth term of the list. The list certainly has duplicates in the sense that it is very well possible that (**E** [n]) = (**E** [m]) or (**E** $\mathbf{c}_n$) = (**E** $\mathbf{c}_m$) for some n≠m. Because we have no predicate for the equality of terms, it is not possible to remove these duplicates. The construction of term **E** is not shown here (See Barendregt) Below Racket and *redex* are used. Let the number of occurrences of **X** be the length of an xterm. We make a function that lists all xterms with length *n*. Concatenation of all these lists results in a countably infinite list of all combinators.

```
#lang racket                                              ; File term-generator.rkt
(require redex)
(printf "~a~n" "term-generator")

(define-language x-lang
 (‹xterm› X (‹xterm› ‹xterm›))
 (‹hole› (‹hole› ‹xterm›) (‹xterm› ‹hole›) hole))

(define extend
 (reduction-relation x-lang
  (--> (in-hole ‹hole› X) (in-hole ‹hole› (X X)))))

(define (list-xterms n) ; Lists all xterms with n occurrences of X.
 (case n
  ((0) '()) ((1) '(X)) ((2) '((X X)))
  (else (remove-duplicates (apply append (map apply-extend (list-xterms (sub1 n)))))))
```

```
(define-metafunction x-lang Term<? : ‹xterm› ‹xterm› -> any
  ((Term<? X X) #f)
  ((Term<? X ‹xterm›) #t)
  ((Term<? ‹xterm› X) #f)
  ((Term<? (‹xterm›_0 ‹xterm›_1) (‹xterm›_0 ‹xterm›_2)) (Term<? ‹xterm›_1 ‹xterm›_2))
  ((Term<? (‹xterm›_0 ‹xterm›_1) (‹xterm›_2 ‹xterm›_3)) (Term<? ‹xterm›_0 ‹xterm›_2)))

(define (Catalan n) (quotient (! (* 2 n)) (* (! n) (! (add1 n))))) ; (2n)!/(n!(n+1)!)
; These are Catalan numbers after the mathematician Eugène Charles Catalan.
; They solve many counting problems. Here they are used to count the number of fully
; curried xterms with n+1 occurrences of X, n≥0.

(define (sort-xterms x) (sort x xterm<?))
(define (xterm<? x y) (term (Term<? ,x ,y)))
(define (apply-extend x) (apply-reduction-relation extend x))
(define (! n) (if (zero? n) 1 (* n (! (sub1 n)))))

(define (test n #:print (print-x-terms #t))
  (for ((n (in-range 1 n)))
    (let* ((r (list-xterms n)) (c (length r)) (C (Catalan (sub1 n))))
      (printf "nr of X: ~s, nr of xterms: ~s~n" n c)
      (when print-x-terms (for-each (λ (r) (printf "~s~n" r)) (sort-xterms r)) (newline))
      (test-equal c C)))
  (test-results))

(test 10 #:print #f) ; Replace #f by #t if you want to see the lists of xterms.
```

Results:

```
nr of X: 1, nr of xterms: 1
X

nr of X: 2, nr of xterms: 1
(X X)

nr of X: 3, nr of xterms: 2
(X (X X))
((X X) X)

nr of X: 4, nr of xterms: 5
(X (X (X X)))
(X ((X X) X))
((X X) (X X))
((X (X X)) X)
(((X X) X) X)
```

```
nr of X: 5, nr of xterms: 14
(X (X (X (X X))))
(X (X ((X X) X)))
(X ((X X) (X X)))
(X ((X (X X)) X))
(X (((X X) X) X))
((X X) (X (X X)))
((X X) ((X X) X))
((X (X X)) (X X))
((X (X (X X))) X)
((X ((X X) X)) X)
(((X X) X) (X X))
(((X X) (X X)) X)
(((X (X X)) X) X)
((((X X) X) X) X)

nr of X: 6, nr of xterms: 42
...

nr of X: 7, nr of terms: 132
...

nr of X: 8, nr of terms: 429
...

nr of X: 9, nr of terms: 1430
...

All 9 tests passed.
```

Appendix

Contents

If you don't yet have DrRacket, download it from the PLT Racket download page, execute the downloaded installer and click 'next' until the installer starts. DrRacket is free. After the installer completes, click 'finish' and open DrRacket. Select 'Language Module' in menu item 'Language/Choose Language'. Also select menu item 'Edit/Preferences/General' and enable 'Automatically switch to module language when opening a module'. Personally I also prefer the options 'Open files in separate tabs' and 'Put interactions window beside the definitions window'. It may be handy to enable the option 'Automatically compile source files' in menu item 'Language/Choose Language'. Show details when hidden in order to get the option visible. Setting the language and the preferences is a one time job. DrRacket memorizes the settings, although you may have to quit DrRacket and restart it before all selected options become active. If you are not yet familiar with DrRacket, you may want to have a look at DrRacket: PLT Programming Environment, although in my opinion, the user interface hardly requires any explanation. In the present essay frequent use is made of PLT's *redex*. It has a beginner friendly introduction, but the latter requires some knowledge of Lambda Calculus. In the present essay the first examples using *redex* are self evident. Take a look at the introduction when you feel you are up to it. I make frequent use of Check Syntax. It provides a lot of information about the structure of the program. Many typos can be traced by means of Check Syntax (nowadays, Check Syntax is implied automatically within DrRacket)

The tools and source codes are available as a zip file: lc-with-redex.zip. Download it and unzip. All files must be kept in the same directory. The directory contains the original copy of the present essay. Always use the copy in the directory. Many hyperlinks in this essay assume the referenced material to be present in the same directory as the essay itself. Run examples by following a hyperlink in this essay or by opening one of the '.rkt' files in the directory. Run DrRacket.exe for your own programs. Enter your program in the definitions window. Results are shown in the interactions window after clicking the Run button and in some cases a little bit of patience. The interactions window provides a read-eval-print-loop, but I rarely use it (back to section Tools and source texts)

Acknowledgements

Contents

In arbitrary order:

Thanks to Daniel P. Friedman and Matthias Felleisen permitting me to use an adapted version of their Y-combinator approach in ‘The Little LISPer’ c.q. ‘The Little Schemer’. Also thanks for encouraging me to present this essay to the public, which in casu I did on the PLT Racket discussion list.

Thanks to Douglas R. Hofstadter who permitted me to borrow and adapt the MIU puzzle and to borrow the distinction between *within* and *about* from his book “Gödel, Escher, Bach, an Eternal Golden Braid”. This book is a must!

Thanks to Robert Bruce Findler and Casey Klein for their hints while I was studying how to make module normal-order-beta-reductor-version-2.rkt. This version is not included in this essay because I did not implement it correctly. It appeared not to be a deterministic reductor after all.

Thanks to Eli Barzilay for repeatedly converting this essay to pdf. Notwithstanding the fact that I use Eli Barzilay’s conversions, also thanks to others who did take the trouble of helping me to convert my essay from MsWord doc file to a pdf file.

Thanks for all responses and to all who copied the essay and the associated material to other sites.

Copyright

Contents