Semantics Engineering with Redex in Racket

Adrian Manea
Coordinator: Assoc. Prof. Traian Şerbănuţă

March 6, 2020

Contents

TO ADD/CLARIFY

INTRODUCTION AND MOTIVATION

The starting point for the present work was my motivation of learning the programming language Racket. Having a background in mathematics, the lambda calculus and the function programming languages appealed to me instantly and once I got acquainted with Emacs and the Lisps, my interest grew. Furthermore, the excellent theoretical foundations laid by J. McCarthy in [McC62] and [McC61], described by some computer scientists as having introduced a paradigm shift in theoretical computer science that's comparable to the non-Euclidean geometry revolution proved to be an excellent introduction and motivation for wanting to learn more about the Lisp family of programming language. Before long, I discovered the influential [AS96] and saw how an apparently simple programming language such as Scheme can reveal wonderful constructions of various degrees of abstraction.

Doing some more reading in the world of Lisp dialects, I have discovered Racket, whose appeal was instantaneous to me, since it is so often described not as a general purpose programming language derived from Scheme, but rather as a toolbox for constructing languages to solve various problems. In fact, as I saw it, it offers the necessary items for one to properly understand, craft and teach languages that exhibit particular behaviour.

This is precisely the aim of the current work. Given the inherent shortcomings of Scheme, assumed by its creators for the sake of simplicity and ease of use and extension, and not trying to delve into the whole "zoo" of Lisp dialects, I found Racket to be the most appropriate object of study for my purposes. These are to study semantic aspects of (mostly) functional programming languages, as well as type theory and related formal methods, which Racket has the flexibility to allow, all while using a mild variation of the syntax that was established ever since McCarthy's (Common) Lisp.

However, this work is not a Racket manual and given its sheer flexibility and array of features, it is beyond my scope to thoroughly explore this language-toolbox, at least not when confined within the scope of this dissertation. Therefore, the precise topic I chose to focus on this work is that of **PLT Redex** (henceforth called **Redex** in short, since PLT is the name of the research group that started it, continuing efforts from PLT Scheme, which was to become Racket).

Inspired by the great article and presentation of B. Findler at POPL 2012 ([FF+12]), I will be presenting the main features of Redex (in Racket, as it is implemented), provide some examples and try to show how it can be used not only to create toy languages, but also some which come with included formal methods of checking correctness (a rather vague term which will be detailed in due time).

The plan of the work is as follows. A short historical presentation and preliminaries make up the first part of the dissertation. Then a Scheme and Racket *crash course* will follow, focusing on specific aspects that will come in handy when discussing Redex, such as `call/cc` and `amb`. The main part of the work will then contain the actual presentation of Redex, along with some standard examples that the authors provided ([amb20, lon20, FF09]). Finally, further examples are provided, which exhibit features of interest, related mostly to type theory.

The work is in fact part of a more elaborate plan, which will be detailed in the final section of this dissertation.

It is also worth mentioning that the PLT group ([plt20]) used Redex as a starting point for their work on a language-creating toolbox and their current focus is on Pyret, described as *a programming language designed to serve as an outstanding choice for programming education while exploring the confluence of scripting and functional programming* ([Gro20]). However, since as we will detail, the wider focus is on Racket, we will not touch on this subject here.

change the bib style (especially to go well with websites)

CHAPTER 1

HISTORY AND PRELIMINARIES

index!

`marginpars`?

Ever since the introduction of the lambda calculus formalism for representing functions, by A. Church and others in the 1930s, various uses of it in the foundations of mathematics and in theoretical computer science arose. As it is mentioned in [McC61] and [McC62], while unsuited for recursive functions representation, lambda calculus can be extended to help represent a generous amount of key concepts of programming.

Once the technology started to develop to support the theoretical advances in computer science, approaches such as Landin's in [Lan64] and McCarthy's in [McC60] showed how one can "mechanically" implement some variants of lambda calculus in order to create what were to become prototypes of functional programming languages. This happened in an era when the procedural and imperative ALGOL was one of the greatest achievements in programming languages.[1]

Essentially, some of the features that McCarthy emphasized and which became included later in Common Lisp and Scheme were *continuations*, as a means of capturing the computational content of a lambda-expression, thus making it more useful for recursive functions and *ambiguous "functions"*. The latter were thought of as a means of representing non-determinism and are not actual functions, since they can return either nothing or one or many results. Without getting more into the historical details, we will detail these features in the Scheme crash course a bit later.

reference the section

Making a 50-year time leap, at the POPL conference in 2012, the PLT group, voiced by B. Findler and M. Felleisen presented a tool (rather, a tool*box*) called **Redex** for programming language

[1] McCarthy explains in some detail how his Lisp language (so called by shrinking "list processing") is intended to differ and be better than ALGOL, COBOL and UNCOL in his introduction of [McC61], pp. 1–4.

creation which proposes an approach that is similar to any software development. They called the approach *semantics engineering* and expanded on the idea in their book [FF09]. Basically, what the presentation of [FF+12] outlined and was detailed in the book cited previously were two key aspects which they proposed:

- firstly, they emphasized the empowering of programmers with *language creation tools* that would follow a cycle not dissimilar to general software development, as represented in figure 1.1;
- secondly and equally important, as it is implied by the title of their article and talk, *Run Your Research: On the Effectiveness of Lightweight Mechanization* the aim was that not only can a programmer create their own language, but they can also abstract features of existing languages and then use Redex to *verify* their work. Such "lightweight mechanized verification" can be used even for simple tasks such as LaTeX typesetting.

The demo in their talk was focused on nine ICFP 2009 articles where they discovered various errors, ranging from simple LaTeX typesetting and typos to false theorems which most likely were due to missing hypotheses (probably deemed obvious, but not explicitly stated).

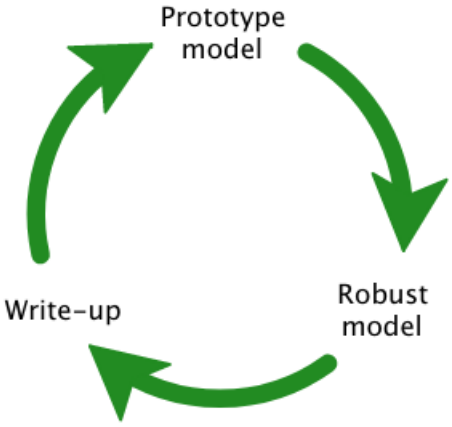

Figure 1.1: Semantics engineering cycle proposed in [FF+12]

In this dissertation, we will skip many of the theoretical aspects of Lisp and Scheme, stopping only at some details on `amb` and `call/cc` and assume the reader can catch most of the syntax on the go.

Without further ado, we will now skim through some features of Scheme and Racket that we will use throughout the rest of the dissertation. For further details, the standard references are [AS96] and [S+13] for Scheme and [rac20] and [FF14] for Racket.

1.1 Scheme and Racket crash course

Since both languages emerged from a common ancestor, Lisp, they use (as the name implies) a *list syntax*. Furthermore, most of the functions and predicates use a prefix notation and the mathematical operations use the so-called Polish (Łukasiewicz) notation. Some basic examples, including specific functions follow.

- `(+ (* 5 3) 1)` computes as $5 \cdot 3 + 1$;
- `(lambda (x) (rem x 2))` returns a function that computes the remainder of its argument modulo 2;
- `(cond (a instruction-a) (b instruction-b) (c instruction-c))` is a branching conditional expression. If `a` is true, then `instruction-a` is executed, else if `b` is true, `instruction-b` is executed and most of the times, `c` is taken to be `t` (true) such that the last branch is executed if everything else fails.

We should remark from the beginning that Racket is almost a superset of Scheme, and as such, its syntax contains that of Scheme. To be precise, Racket expands on the standards R5RS (1998) and R6RS (2009) of Scheme, diverging more significantly from R7RS (2013), but for the purpose of this dissertation, unless otherwise specified, all Scheme syntax presented is assumed to be valid Racket syntax.

While it is not typed in the modern sense of the word, Scheme differentiates between three basic types: *lists*, *functions* and *symbols*, the latter encompassing "everything else".

A specific syntax with its corresponding semantics is the *quoting mechanism*. As such, a *quoted expression*, denoted either by `(quote (expr))` or by `('(expr))` (another valid variation is `(#'(expr))`) is one that evaluates to itself, regardless of what `expr` contains. In brief, it turns `expr` into (a list of) symbols. Examples:

- `(lambda (x) '(+ 1 2 3))` is a function that associates to its argument the (literal) list `'(+ 1 2 3)` (notice that the sum is not evaluated, but it is returned literally);
- `(define animals '(cond cat dog mouse))` defines the variable `animals` to be the (literal) list `'(cond cat dog mouse)`.[2]

The opposite of quoting expressions is *unquoting*, which forces the evaluation and returns the final value in-place. The syntax for unquoting is a comma that precedes the expression that is evaluated. Examples:

- `(lambda (x) ,(+ 1 2 3))` is a function that returns 6 for all its arguments, since it associates to `x` the *final value* of `(+ 1 2 3)`, which is 6;

[2] We are aware that there's no "cond" animal, we included it to make it clear that it is not evaluated, i.e. the list is not seen as a branching conditional, but instead returned literally.

- `(define sum '(+ 1 2 3)) (define x (* ,sum 2))` makes `x` to be 12, since in its definition, the initially quoted `sum` is now forcibly evaluated.

In fact, the unquoting mechanism is an artifice of *metaprogramming*, more precisely an instance of a macro, which is one of the features that Lisps excel in.

Some more elements of syntax that will provide useful are listed briefly below.

- As seen above, functions and variables are defined with `(define ...)`. In fact, usually `define` is used for functions and variables are defined and given a value with `set`. There's also the variation `set!`, which forcibly overwrites whatever value the variable had;
- `cons` is the common function for appending to a list;
- `car` is the function that returns the head of the list and `cdr` returns its tail, so `(car '(1 2 3))` evaluates to `1` and `(cdr '(1 2 3))` evaluates to `(2 3)`;
- Comments are preceded by semicolons (`;`) and the convention is to use single semicolon for in-line comments and double semicolons for full-line comments. Also, in-line comments are used for writing the result of evaluating an expression, with an added arrow, e.g. `(+ 2 3) ; => 5`.

1.2 `amb` and `call/cc`

We will now give special attention to *ambiguous evaluation* using the function `amb` and also the *continuation* support included in Scheme, as they are specific features that will be used later on. Furthermore, from a historical perspective, Lisps were the first to support both of them. More languages followed, such as:

- `amb` implementation in many programming languages is comprehensively presented at [ros20];
- *continuations* are supported, for example, in Ruby's *Continuation* class ([rub20]), C++'s *context switching* ([Kow14]) and Haskell's `Control` monad ([has20]).

We will only present here the implementation of the two functions in Scheme, using mostly [AS96] and [non20]. Furthermore, the specifications for both are included in [S+13]. Since `amb` is implemented using `call/cc`, we will start with the latter.

By definition, the *continuation* of a computation is what will happen after the current computation is performed. For example, in the computation `(+ (* 2 3) 5)`, the continuation of the multiplication is the addition of 5 to the result. As such, one can abstract the continuation from this example with a lambda expression:

```
(define cont1
  (lambda (x) (+ x 5)))
```

Then, we get the same result as above by calling:

```
(cont1 (* 2 3))     ; => 11
```

While this example may not seem like much, the basic idea that one can capture "a part" of a computation and reuse it for some other purpose is important.

This concept is captured in the standard `call/cc` function in Scheme (sometimes implemented as `call-with-current-continuation`). Some basic examples are the following:

```
;; Nothing to capture
(define id              ; the identity function
  (lambda (k) k))
(call/cc id)            ; returns the function
                        ; since there's nothing left to do

;; Continuation is just the argument
(define apply-to-zero
  (lambda (k) (k 0)))
(call/cc apply-to-zero) ; returns 0
                        ; since that continues the function

;; Capture a further operation
(+ (call/cc
     (lambda (k) (k (* 1 2))))
      3)
;; is equivalent to
((lambda (v) (+ v 3)) (* 1 2))

;; Basically, the continuation only is (+ _ 3):
(lambda (x) (+ x 3))
```

The `call/cc` function can be used for saving a certain point of a computation. For example, we can save a backup copy of a function than can be used later:

```
(define z #f)                   ; initialize z with false
(+ (call/cc
     (lambda (k)
        (set! z k)              ; clear what's stored in z and put k
        (* 1 2)))
    3)                          ; => 5
```

In the example above, we have stored only the lambda-function in `z`, so even if the whole computation returns `5`, we have like a "bookmark" at the `lambda` definition which is stored in `z`. Henceforth, we can do:

```
(z 1)               ; => 4
```

The `amb` operator is a bit more complicated and although it is implemented in the standard library in some Scheme distributions, we will define it manually. We will follow the presentation in [non20]. This will define `amb` as a macro, which ensures that the arguments are not evaluated and as we will see, it is implemented using `call/cc`.

The basic idea, as outlined first in [McC61], is that the `amb` operator works as an ambiguous evaluation, with a variable number of arguments, in the following sense:

- if none of its arguments can be computed or if no arguments are given, it returns some kind of failure (we will use `# f` for `false`);
- if any (one or more) of its arguments can be computed, return all the possible values.

For example, McCarthy defines in pseudocode:

```
less(n) = amb(n-1, less(n-1)),
```

which is a "function" (quotes because it can return multiple values) that returns all the numbers that are less than the argument `n`. Thus, with `n` assumed to be a positive integer, `amb` tries to evaluate both `n-1` and recursively compute `less(n-1)`. If both can be computed, it returns both, which will happen until `n = 1`, when the final call will be `amb(0, less(0))`, that returns `0` and the computation stops, since none of the arguments won't work at the next iteration.

Here's how `amb` can be implemented in Scheme, using `call/cc`:

```
(define f #f)                                   ; 1

(define-syntax amb                              ; 2
  (syntax-rules ()                              ; 3
    ((_) (f))                                   ; 4
    ((_ a) a)                                   ; 5
    ((_ a b ...)                                ; 6
      (let ((s f))                              ; 7
        (call/cc                                ; 8
          (lambda (k)                           ; 9
            (set! f (lambda ()                  ; 10
                      (set! f s)                ; 11
                      (k (amb b ...))))         ; 12
          (k a)))))))                           ; 13
```

The basic ideas are the following:

- In the `syntax-rules` declaration, the underscore denotes the syntax (function) that is defined, `amb`;
- The first case (line 4) is when `amb` is called with no arguments, `(amb)`, case in which it returns `(f)`, which is `false`;

- Then, in line 5, if it is called with some argument, it will return that;
- If it is called with more than one argument (the ellipse being special syntax for zero or more arguments that could follow), then the actual work is done:
  - first, we introduce a variable `s` and bind it to `f`, which is `false`;
  - then, `call/cc` is called to capture the current continuation in `k`;
  - inside this `lambda`, `f` is reset and a recursive call is made, invoking `k` (the previous computation) to `amb` called with the rest of the arguments (starting with `b`);
  - finally, in line 13, the captured outermost `lambda` is applied to `a`, which makes the innermost `lambda` be called only after this computation, so first it tries to compute `(amb a)`

Remark 1.1: Note that by default, many Scheme and Racket implementations make `amb` behave step by step. That is, when called with multiple arguments, of which more than one can be computed, the first evaluation returns only the first computation. Calling `amb` again returns the second possible computation etc. Hence, for example, Racket 7.5 REPL, using the `#lang scheme` pragma, gives the following session, using the implementation above:

```
REPL> (amb "dog" "cat")
"dog"
REPL> (amb)
"cat"
REPL> (amb)
#f
```

Remark 1.2: One last word regarding the source code and the entire contents of this dissertation. While Racket comes with an exceptionally powerful and user-friendly IDE called `DrRacket` which is automatically installed when installing the language and for which the book [FF14] can be seen as an extensive user manual, we have chosen to write and test everything contained in this dissertation in GNU Emacs 26.3 running under Manjaro Linux. Scheme support is built in and for Racket, we used the full-featured REPL included in racket-mode.

CHAPTER 2

PLT REDEX

BIBLIOGRAPHY

[amb20] Amb: A redex tutorial. https://docs.racket-lang.org/redex/tutorial.html, 2020. Accessed: March 2020.

[AS96] Harold Abelson and Gerald Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1996.

[FF09] Matthias Felleisen and Robert Findler. *Semantics Engineering with PLT Redex*. MIT, 2009.

[FF+12] Matthias Felleisen, Robert Findler, et al. Run your research: On the effectiveness of lightweight mechanization. *POPL*, 2012.

[FF14] Matthias Felleisen and Bruce Findler. *How to Design Programs*. MIT Press, 2014.

[Gro20] The PLT Group. Pyret. https://www.pyret.org/, 2020. Accessed: March 2020.

[has20] The control monad. http://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-Cont.html, 2020. Accessed: March 2020.

[Kow14] Oliver Kowalke. Context switching with call/cc. https://www.boost.org/doc/libs/1_72_0/libs/context/doc/html/context/cc.html, 2014. Accessed: March 2020.

[Lan64] Peter Landin. The mechanical evaluation of expressions. *Comput. J.*, 6:308–320, 1964.

[lon20] Long tutorial. https://docs.racket-lang.org/redex/redex2015.html, 2020. Accessed: March 2020.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Commun. ACM*, 3(4):184–195, April 1960.

[McC61] John McCarthy. A basis for a mathematical theory of computation. *Western Joint Computer Conference*, 1961.

[McC62] John McCarthy. Towards a mathematical science of computation. *IFIP-62*, 1962.

[non20] A gentle introduction to non-determinism in scheme. https://ebzzry.io/en/amb/, 2020. Accessed: March 2020.

[Plo75] Gordon Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[plt20] The PLT group. https://racket-lang.org/people.html, 2020. Accessed: March 2020.

[rac20] Racket. https://racket-lang.org/, 2020. Accessed: March 2020.

[ros20] Amb implementation. http://rosettacode.org/wiki/Amb, 2020. Accessed: March 2020.

[rub20] Continuation in ruby 2.5.0. https://ruby-doc.org/core-2.5.0/Continuation.html, 2020. Accessed: March 2020.

[S+13] Gerald Sussman et al. Revised7 Report on the Algorithmic Language Scheme. Technical report, Scheme Working Group, 7 2013.