Making Languages: An Exploration of the Ebook Beautiful Racket

Presented to the Meetup Group FunctionalKnox by

Mark Carringer

On January 2, 2018

What can I get out of this presentation?

- An abrupt introduction to Lisp-style programming.
- An awareness of Racket as a vehicle for creating languages.
- A general idea of how Racket creates new languages.
- A notion of the potential for using macros to extend a language at compile-time.
- A curiosity about the utility and feasibility of domain-specific languages.

History of Racket

Lambda Calculus (1936)

Lisp (1958)

Scheme (1970)

PLT Scheme (1994)

Racket (2010)

Features of Racket (1 of 2)

- Like all Lisps, a distinctive parenthesized syntax
- Primarily but not purely functional
- Highly developed macro system
- Dynamic typing, with a contracts system

Features of Racket (2 of 2)

There are three core flavors of Racket:

- #lang racket
- #lang racket/base
- #lang typed/racket

But the goal of the language, as we will see, is to be capable of producing any language one can imagine. The variety of language names one may see following a #lang is quite large.

Really, Really Fast Racket Tutorial (1 of 4)

- Lists are everywhere. Every program is a list. You make a list…
- Using the list function:

```
(list 2 3 4)
(list 1 "apple" #t)
(list (list 1 2 3) (list 3 4 5))
```

- Or by using the quote function:

```
(quote "apple" "banana" "cherry")
```

  which may also be written:

```
'("apple" "banana" "cherry")
```

- The difference is that list evaluates its arguments as far as possible, while quote does not. This feature makes quote useful for passing around code within a program.

Really, Really Fast Racket Tutorial (2 of 4)

- Functions are defined like this:

```
(define (add-one n)
  (+ n 1))
```

- The last expression of a function body is the value to which it evaluates.
- This function may now be invoked as follows:

```
> (add-one 3)
```

- Or the same function could be passed to or returned from another function without being invoked.

Really, Really Fast Racket Tutorial (3 of 4)

- You use a PDE (notably, not an IDE) named DrRacket.
- Single-line comments are the most common. Everything to the right of a semicolon is a comment.
- Units of source code are called *modules*. Modules correspond to single files, as far as I've seen.
- In Racket parlance, you "provide" a module or function to export it, and you "require" a module or function to import it.
- There are also *packages* and *collections*.

Really, Really Fast Racket Tutorial (4 of 4)

Racket programs are made up of S-expressions. An S-expression may be either:

1. An atomic value, like a string, number or symbol.
2. A parenthesized list of values.

This second category is constituted of either:

a) Functions
b) Special Forms

Racket Culture (with respect to Naming)

- Pascal and camel case are frowned upon, as is snake case (use of underscores).
- Uppercase letters are frowned upon, although there are special uses of ALL CAPS.
- Hyphens are used extensively.
- A mutating function ends with a !.
- A predicate (Boolean-valued function) ends with a ?.
- Pairs of brackets [] and braces {} may be used in place of parentheses for readability. There is no difference in meaning.

The Racket Manifesto (1 of 4)

Three Principles:

1. "Racket is a programming language for creating new programming languages."
2. "Racket offers protection mechanisms to implement a full language spectrum."
3. "Racket also turns extra-linguistic mechanisms into linguistic constructs."

The Racket Manifesto: First Principle

- “Programming is about stating and solving problems, and this activity normally takes place in a context with its own language of discourse; good programmers ought to formulate this language as a programming language.”
- In other words, we should make languages for specific problem domains rather than using general-purpose languages for multiple problem domains.
- The approach is termed "language-oriented programming".

The Racket Manifesto: Second Principle

- "An abstraction enforces invariants. Languages are abstractions, and their creators must have the means to build the necessary enforcement mechanisms—especially when components in these languages end up in interconnected, multi-lingual contexts."

The Racket Manifesto: Third Principle

- "Program development is also a problem domain."
- To facilitate rapid development, tools normally external to a programming language are internalized in Racket.

Why is Racket Suited to Making Languages?

- Racket is sometimes said to have “no syntax”.
- “If you understand how compilers work, what's really going on is not so much that Lisp has a strange syntax as that Lisp has no syntax.” http://www.paulgraham.com/avg.html
- Which means that the programs you write are already in the form of parse trees which the compiler uses.
- The other reason is Racket's sophisticated macro system.

The Macro System (1 of 2)

- Macros are like a regular expression system for source code.
- Macros are run at compile-time. Code is “expanded” (or rewritten), but expressions are not evaluated.
- Obviously, macro expansion always precedes run-time evaluation.
- Macros could be thought of as custom extensions of the Racket compiler.
- Macros in general cannot be treated like first-class functions. That is, they cannot be passed and returned.

The Macro System (2 of 2)

- #lang racket comes with various special forms which look like functions, but are actually macros.
- Examples include let, cond, if, and, or, lambda, require, provide, module, and many more.
- These special forms are (to the best of my understanding) rewritten at compile time using macros.

Necessity of Macros (1 of 3)

- One advantage of macros is that the expressions they rewrite are not evaluated. This fact can make a difference when expressions have side effects.
- Say we want to write our own version of if:

```
(define (our-if condition true-expr false-expr)
  (cond [condition true-expr]
        [else false-expr]))
```

- It seems to work fine:

```
> (our-if #t "true" "false")
"true"
```

Necessity of Macros (2 of 3)

- However, what if we pass an expression with side effects to our-if ?

```
(define (display-and-return x)
  (displayln x)
  x)
```

```
> (our-if #t
          (display-and-return "true")
          (display-and-return "false"))

true
false
"true"
```

Both expressions were evaluated, which was probably not the intent. The issue is that the runtime evaluates all expressions, regardless of control flow.

Necessity of Macros (3 of 3)

- We can solve this problem using a macro:

```
(define-syntax (our-if-v2 stx)
  (define xs (syntax->list stx))
  (datum->syntax stx `(cond [,(second xs) ,(third xs)]   ; see
                            [else ,(fourth xs)])))       ; footnote*
```

- Now the function works as expected:

```
> (our-if-v2 #t
             (display-and-return "true")
             (display-and-return "false"))

true
"true"
```

* To write the above macro, you would need to make the following substitutions: cadr for second, caddr for third, cadddr for fourth. The classic Lisp list functions (car, cdr, cons, and their variants) are ubiquitous in Racket, but are perhaps too obscure for this presentation. Alternatively, you can put the following expression before define-syntax expression: (require (for-syntax racket/list)) Only racket/base is available by default at the macro stage, which is an easy fact to stumble over.

In fact…

We can rewrite *any* sort of code with macros, not just Racket code.

And that is where this adventure begins.

Beautiful Racket

- Online book by Matthew Butterick
- Guy is primarily a typographer.
  Has a BA in Visual and Environmental Studies from Harvard, and a law degree from UCLA.
- Created Pollen publishing system for online books:
  http://docs.racket-lang.org/pollen/

Key Idea

- We are creating languages by specifying them and then transcompiling them into Racket. And then executing them in Racket.
- Much like other languages are compiled into JavaScript.
- The languages br, br/quicklang and brag which were made for the book make language building easier.

The Racket Approach to Making a Language

- Every language has a reader and an expander.
- The **reader** takes the source code and formats it into what are called S-expressions. S-expressions are the concentric parenthesized expressions that we see in ordinary Racket.

The Racket Approach (2 of 8)

- So let's say we are writing a simple, stack-based calculator language, with one argument per line. The input could look like this:

```
4
8
+
3
*
```

- This input corresponds, in ordinary math notation, to the expression ((4 + 8) * 3), which evaluates to 36.

The Racket Approach (3 of 8)

The reader *might* take this input and wrap it in S-expressions as follows:

```
(handle 4)
(handle 8)
(handle +)
(handle 3)
(handle *)
```

- Where did this handle function come from? From nowhere. The reader simply declared that some function named handle would deal with all the arguments. (Someone had better define it.)

The Racket Approach (4 of 8)

```
(define (handle [arg #f])
  (cond
    [(number? arg) (push-stack! arg)]
    [(or (equal? + arg) (equal? * arg))
     (push-stack! (arg (pop-stack!) (pop-stack!)))]))
(provide handle)
```

- We also define functions like push-stack! and pop-stack!, while things like cond and equal? are already available for use.

The Racket Approach (5 of 8)

- More technically, every reader must define a function named read-syntax. Here is the read-syntax for the example we have:

```
;; the reader
(define (read-syntax path port)
  (define src-lines (port->lines port))
  (define src-datums (format-datums '(handle ~a) src-lines))
  (define module-datum `(module stacker-mod "stacker.rkt"
                          ,@src-datums))
  (datum->syntax #f module-datum))
(provide read-syntax)
```

The Racket Approach (6 of 8)

- The most important thing to understand about what is happening is that the read-syntax function (not macro) is returning a syntax object.
- This syntax object is a unit of code that is structured in the way that real Racket code is structured. However, in this case, it isn't real Racket code.
- Another view of the syntax object is that it is a data structure, containing the source code, along with other information such as the lexical context and the source location.

The Racket Approach (7 of 8)

- The foregoing syntax object is then passed to the expander, which decides how this syntax object shall be executed as ordinary Racket code.
- Every expander must define a macro named #%module-begin. Ours looks like this:

```
(define-macro (stacker-module-begin HANDLE-EXPR ...)
  #'(#%module-begin
     HANDLE-EXPR ...
     (display (first stack))))
(provide (rename-out [stacker-module-begin #%module-begin]))
```

The Racket Approach (8 of 8)

- So, given the following input:

```
4
8
+
3
*
```

- The output will be:

```
36
```

- Broadly speaking, this is a programming language, with one argument per line, and each argument being either a number or an arithmetic operator. A stack is also maintained behind the scenes.

Summary of the Racket Approach

- "The reader converts the source code of our language from a string of characters into Racket-style parenthesized forms, also known as S-expressions."*
- Every reader provides a read-syntax function.
- "The expander determines how the code produced by the reader corresponds to real Racket expressions, which are then evaluated to produce a result. The expander works by adding *bindings* to identifiers in the code."*
- Every expander provides a #%module-begin macro.

* These quotes appear several times throughout the book Beautiful Racket.

Demonstration

- Of course, the previous example merely used pattern matching, which we know is not sufficient for making many languages.
- Recall that pattern matching is enough to give us the regular languages, but for many things we need a context-free grammar.
- Also, we should probably figure out how to tokenize the input, which will simplify our grammar considerably.

Spec for a Spreadsheet Language (1 of 2)

- Take an intuitive notion of a spreadsheet, like Excel.
- The language has a set of single-line commands for input, and an output consisting of a list of cell reference/value pairs.

```
set a1 3
set a2 "apple"
set a3 a1
```

- The input syntax includes a means of setting a mode either allowing or disallowing the mutation of envalued cells. The default is to allow mutation, and this mode may be changed any number of times during the course of a program. Unenvalued cells may always be initialized.

Spec for a Spreadsheet Language (2 of 2)

Commands:

set cell-ref value

mode (static | dynamic)

Only one command per line of input is permitted.

Comments will be everything to the right of a //.

Building a Quick Language (1 of 2)

We will take the above program:

```
set a1 3
set a2 "apple"
set a3 a1
```

Which maps to the output:

```
A1 3
A2 "apple"
A3 3
```

And build a general spreadsheet specifying language.

Building a Quick Language (2 of 2)

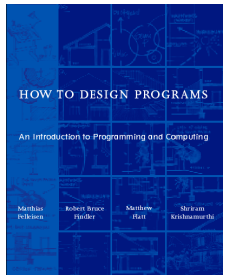
- First we make a reader, which uses a tokenizer to wrap the input in Racket-style s-expressions.
- Then we make a parser, simply by specifying a grammar in extended Backus-Naur form. (Here is a use of a DSL.)
- Then we make an expander, which consumes the syntax tree produced by the parser, rewriting the nodes of the tree and evaluating them.
- When finished, we are able to put **#lang spreadsheet** at the top of a Racket file and work in a DSL that looks nothing like Racket.

Shortcomings of This spreadsheet Language

- Does not support a rich attribute grammar.
- Feels more like a utility.
- The implementation is not completely functional.

The Racket community is actually doing it:

HtDP	*teaching language for the textbook How to Design Programs*
brag	*parser generator*
datalog	*logic programming*
web-server	*(guess)*
shill	*scripting "with least privilege"*
Pollen	*making e-books, like Beautiful Racket*
Pyret	*teaching language*
Rex	*specifying regular languages*
Rosette	*program synthesis and verification (wow!)*
Scribble	*all Racket documentation is written in Scribble*
Slideshow	*(guess)*



Different Kinds of DSLs

- From-Scratch: A new language from the ground up. (But probably adding and subtracting features from existing languages.)
- Embedded: Making a new language as a superset of an existing language. Like, a language that is JSON, with the added ability to embed Racket expressions within JSON. The expressions are evaluated so that the final output is still JSON. (This example is actually carried out in Beautiful Racket.)
- Ad-Hoc: Creating a new language as the shortest route to solving a specific problem, and then perhaps never using the language again.

Discussion Points

- Do you agree with the premise of "language-oriented programming" implied by the Manifesto?
- Do you like that the author provides a language that makes language-writing easier, or would you rather suffer through the core language and get the "real story"?
- Is language creation feasible as a problem-solving approach? Can DSLs be sufficiently correct, scalable, extensible? Can they be designed in advance of the problems they are intended to be used for?

References and Acknowledgments

- The Racket Manifesto may be found here: http://www.ccis.northeastern.edu/home/matthias/manifesto/
- The example about writing our-if comes from the tutorial Fear of Macros, and may be found here: http://www.greghendershott.com/fear-of-macros/Transform_.html
- Beautiful Racket is found here: https://beautifulracket.com/

Contact Info

- My source code:

https://github.com/markdcarringer/spreadsheet

- My email:

Mark.D.Carringer@gmail.com