No Escapes Needed: A Regular Expression Sublanguage for Rhombus

Cooper Coradeschi

Matthew Flatt

James St Germain

{u1310898, mflatt, u0035613}@utah.edu

4/…/2024

**Abstract:** Rhombus is meant to present the macro extensibility of Racket and other Lisp-style languages in a new, non-s-expression, syntax, and, in addition, allows for sublanguages to have context specific macros. This extension to the power of macros is going to be exploited in the development of a regular expression sublanguage for Rhombus, that puts an emphasis on readability and works as a proof of concept for Rhombus’s macro doctrine.

**Introduction**: Regular expressions, first introduced as regular events by Kleene in 1956, have been an integral part of pattern matching and string processing since their use in Ken Thompson’s ‘ed’ text editor. Since then, Perl’s regular expression engine, in unison with the Unix 8th Edition, has remained the standard, syntactically as well as functionally. This, however, presents a host of issues as Perl’s regex documentation self admittedly “varies from difficult to understand to completely and utterly opaque…” [7] and the syntax itself has been found to be increasingly inconvenient, backslash overload, obscure keywords, and clunky posix integration being the main concerns. To try and alleviate these issues the “re” sublanguage for Rhombus will implement non-string based regular expressions that exploit the domain specific macro extensibility inherent to the language.

While a regular expression language is the end product, the goal is to explore the efficacy of Rhombus’s language-extension abilities. Rhombus’s focus on providing a framework that allows for easy, macro-based language production will be put to the test by adapting a near 40-year-old syntax into one that is readable and integrates into the host language well. The re language is meant to capitalize upon and explore the benefits and limitations of both Rhombus’s macro facilities and syntactic constructs and constraints.

**Previous Work:** One of the issues with developing a new syntax for regular expressions is that the focus on regular expressions is, understandably, focused on optimization and cybersecurity rather than presentation. As a result, much of the development in this subject has been pulled from contemporary languages’ documentation pages.

As we see it, two main issues plague the world of regular expressions. The first is that documentation is often unhelpful and regular expressions have inherently undefined behavior, and the second is the issue of readability.While Racket has addressed the concerns presented by the former by way of a more constrained type system, the issue of readability remains an open question which languages are still grappling with. While Python opted to introduce a verbose mode which allows for inline comments, it seems that there is a shift towards verbosity and the depreciation of string contained regular expressions entirely. Swift’s RegexBuilder is an incredibly verbose method of producing regular expressions, but they have eliminated the backslash in its entirety. This format has helped take away some of the confusion that comes from the many different “esoteric symbols and escaped characters,” [8], however, the language trades one esoteric set of symbols for another, with a frequent abuse of curly braces, indents, and accessor dots. Another language, Pomsky, has taken maybe the most readable approach, its syntax is clean and readable and presents some interesting ideas like number ranges, e.g. `range ‘0’-‘255’` compiles to a regular expression that matches only [0, 255]. The language, however, currently only offers an online playground, a cli, and integration into JavaScript and Rust.

On the other hand, research into the world of macros is extensive. Beginning with Lisp and Scheme, the idea of developing a core notation that supported syntactic expansion to develop new languages are target specific domains transformed into macro extensibility. In contrast to the simpler, substitution-based macros of C and C-style languages, macro expansion is a separate stage of evaluation before compilation. However, the s-expression style of these languages has led to confusion of where core notation ends and macros begin. Some modern languages like Rust, Scala, and Lean have implemented more sound macro systems than their C-style predecessors but their implementations are often post-hoc and don’t provide the full expressiveness that languages like Scheme or Racket can provide. By developing Rhombus with macros explicitly in mind and as a core tenet of the language, Rhombus’s macro extensibility is attempting to integrate a sound, consistent syntactic extension that still maintains the full power of Racket’s macro system. It is through the shrubbery notation form and support for multiple spaces, that is the ability to use context to differentiate between overloaded macros in the same module, that Rhombus fulfills these goals.

**Proposed Work:** The research begins with an analysis of what does and does not work in contemporary representations of regular expressions. The first thing that most people will complain about is the confluence of two major factors, the use of strings to represent regular expressions and the many backslashes that denote keywords and character classes. Does `\\n` mean the literal `\` followed by `n`? Or is it an escape to represent newline? The confusion here, especially when reading a regular expression already present in the code base, can lead to misunderstandings and improper use.

To help negate this, the new regular expression language is going to put an emphasis on readability, both through the use of new keywords, which aim to alleviate confusion, and the exploitation of Rhombus’s Shrubbery notation for things like inline comments, readable formatting, and the inclusion of previously defined regular expressions. Additionally, named capture groups will help alleviate confusion when using backreferences. A lexical syntax for the re sublanguage has already been developed that fulfills these requirements, placing an emphasis on matching the theme of existing Rhombus syntax to ensure as little dissonance between the super- and sublanguage as possible. This syntax, in its current state, can be seen in **fig. 1**.

After the sublanguage has been constructed the next step is compilation. Currently, the plan is to use Rhombus’s context specific macros to compile down the regular expression to a paired form, the first part being a Racket regular expression string and the second being a secondary data structure that may be used for operations like variable substitution. This latter structure ideally being statically compiled as to minimize our runtime debt. The exploration of producing both the strings and the data structure, as well as the form of that structure will be the next step, alongside revisions to the language as conflicts and new ideas develop.

Afterwards the plan is to implement an API that allows for all of the standard regular expression calls, this likely will be using the escape s-expression form provided by Rhombus to call the standard regex library provided by Racket. Then, a series of tests, using industry standard and sufficiently complex regular expressions like ones used to validated JSON validation, email addresses and so forth. Additionally, the re sublanguage will be compared to languages less constrained by external factors, such as Pomsky [2] and demonstrating its use in Rhombus programs to show the ease of integration that the sublanguage, through Rhombus’s framework, provides.

regexp ::= <atom>  
 | <regexp>|<regexp>  
pieces ::= ; matches empty  
 | <piece><pieces>  
piece ::= <repeat>  
 | <repeat> lazy ; lazy operator  
 | <atom>  
repeat ::= <atom>\*  
 | <atom>+  
 | <atom>?  
atom ::= (<regexp>) ; match & report  
 | [<range>] ; match range inside  
 | [^<range>] ; match 'range inside  
 | . ; match any ('\n in ML Mode)  
 | ^ ; match start (")  
 | $ ; match end (")  
 | <character> ; any character (NO ESCAPES NEEDED)  
 | (mode:<mode>:<regexp>) ; match using given mode  
 | (first <regexp>) ; match only first   
 | <look\_to> ; match empty if <look\_to> matches  
 | (if <test> | <pieces> | <pieces>)  
 | (if <test> <pieces>)  
 | <name> = (<regexp>) ; named capture group  
 | newline  
 | null ; match null character  
 | tab  
 | range <num\_range>  
range ::= <literal> ;   
 | <literal><literal> ; concatenation  
 | <literal>-<literal> ; unicode range  
 | <range>&&<range> ; intersection of ranges  
 | <range>--<range> ; direct subtraction  
 | <range><range>  
look\_to ::= (matches <regexp>) ; match if regexp matches  
 | (not matches <regexp>) ; match if regexp does NOT match  
 | (matches prev <regexp>) ; match if regexp matches preceding  
 | (not matches prev <regexp>) ; match if preceding does not match  
test ::= (<n>) ; true if nth '(' has matches   
 | <look\_to> ; true if look\_to matches  
 | (name) ; true if named capture group matches  
mode ::= ; empty - same as surrounding mode  
 | <mode>insensitive ; case insensitive  
 | <mode>sensitive ; case sensitive  
 | <mode>multiline ; multiline mode  
 | <mode>monoline ; not multi mode  
literal ::= any character   
num\_range ::= "<n>"-"<m>" ; matches any and all numbers between n and m  
 ; range "0"-"244" will match "0" and "000" etc.  
  
Pregexp:  
repeat ::= ...  
 | <atom>{<n>}  
 | <atom>{<n>, <m>}  
 | <atom>{,<m>}  
 | <atom>{}  
atom ::= ...  
 | backref<n>  
 | <class>  
 | word bound ; match wordbound  
 | in word ; matches not in word (word\*)  
 | property {<property>}  
 | not property {<property>}  
range ::= ...  
 | <posix>  
class ::= digit ; digit  
 | not digit  
 | word ; a-z A-Z 0-9  
 | not word  
 | whitespace ;space, tab, newline, formfeed, return  
 | not space  
posix ::= alpha ; a-z A-z  
 | upper ; A-Z  
 | lower ; a-z  
 | digit ; 0-9  
 | hex ; 0-9, a-f, A-F  
 | alnum ; a-z, A-Z, 0-9  
 | word ; a-z, A-Z, 0-9, \_  
 | blank ; space and tab  
 | space  
 | graph ;Contains all ASCII characters that use ink  
 | print ;Contains space, tab, and ASCII ink users  
 | cntrl ;Contains all characters with scalar value < 32  
 | ascii ; any ASCII char  
**Fig. 1**

**Timeline**: The rx sublanguage is to be completed between the spring and summer semesters of 2024, the first semester has so far been dedicated to researching representations of regular expressions and developing a syntax that fulfills the principles laid out above. Over the next few weeks experimentation with different data structures is going to be the main focus, and the summer semester will be dedicated to fleshing out the sublanguage completely, implementing an API, and bug testing to ensure a complete and sound regular expression experience. Ideally the data structures will be experimented on and finalized by the end of the semester, however complications, most likely coming from character classes, may necessitate changes later into the next semester. The Summer semester will ideally be split into 3 week periods, each for finishing the language, implementing the API, bug testing, and writing up the final thesis paper.

**References:**

[1] Flatt, M. Et al. Rhombus: A New Spin on Macros without All the Parentheses  
[2] https://pomsky-lang.org/docs/get-started/introduction/  
[3] https://docs.python.org/3/library/re.html   
[4] https://developer.apple.com/documentation/regexbuilder   
[5] https://docs.rs/regex/latest/regex/   
[6] https://www.rand.org/content/dam/rand/pubs/research\_memoranda/2008/RM704.pdf  
[7] https://perldoc.perl.org/perlre#BUGS  
[8] https://quickbirdstudios.com/blog/regexbuilder/