High-Performance Grammar-Based Text Search

Paul MacNichol, Jamie Jennings
pemacnic@ncsu.edu, jjennings@ncsu.edu

1 Introduction

Regular expression implementations departed long ago from the formal model of regular languages, and have a bewildering variety of features. Regexes are hard to write [3, 5] and under-tested [4], giving rise to both correctness and performance issues. Recent interest in Parsing Expression Grammars (PEGs) [1] as an alternative to regex partly derives from their simple formal model. PEGs can recognize a larger set of languages on the Chompsky hierarchy than the regular languages: any regular language, most context-free languages, and a few context-sensitive languages can be recognized [1].

The Rosie Pattern Language (RPL) (https://rosie-lang.org) is a language-independent "external" Domain Specific Language (DSL) for pattern-based matching and searching utilizing an *lpeg*-derived bytecode interpreter [6]. RPL is a PEG-like combinator language that syntactically resembles a traditional programming language, for improved readability and ease of development. Unlike modern regexes, RPL patterns are composable, allowing complex RPL patterns to be built from simpler ones.

In this paper, we focus on the pattern compiler in the Rosie project, which takes RPL as input and emits bytecode for a small pattern-matching virtual machine (VM). In a new codebase that will become Rosie 2.0, we have re-architected the Rosie pattern compiler to support ahead-of-time compilation, separate compilation of pattern packages, tail recursion elimination, function inlining, loop unrolling, character search optimization, and also some novel language-level features [7]. Most of the features in this list are well-known to compiler writers since the 1960s [2], except for *character search optimization*, which is specific to our domain and the subject of the next section.

Rosie 1.4 is implemented in a mix of Lua and C, and its origins as a layer on top of *lpeg* have become an obstacle to adding new functionality and to improving time and space performance. Rosie 1.4 incorporates a heavily modified *lpeg,* but further re-architecting and rewriting became necessary to support new language features and the list of compiler enhancements given above, with their corresponding runtime implementations.

The Rosie 2.0 codebase is a work in progress that incorporates lessons learned through feedback collected from users of Rosie 1.x over the last six years. Rosie 2.0 is written entirely in C resulting in markedly reduced memory usage compared to Rosie 1.4 (and *lpeg*). The new code is generally much faster than both *lpeg* and Rosie 1.4 while maintaining a larger feature set.

A primary factor in the performance improvement (time to search for an occurrence of a pattern) is the introduction of a new pattern matching instruction in the VM. The new instruction allows a single VM instruction to skip ahead in the input until it encounters a character that could be the first character in an occurrence of a given pattern.

2 Character Search Optimization

The two primary use cases for Rosie are character matching and searching. Matching is used, e.g. for input validation. We note that input validation, often done with regex, is a perennial source of software vulnerabilities. Here we focus on searching and present a *character search optimization* that exists in neither *lpeg* nor Rosie 1.4. To search an input text for a span matching pattern P, we might write any one of the following semantically equivalent RPL definitions, where S is the grammar start symbol, "/" is the PEG ordered choice, the dot matches any character, the exclamation point is RPL syntax for negative lookahead, and curly braces are used for grouping:

```
1. S = { P / { . S } }
2. S = { { !P . }* P }
3. S = find:P
```

The first definition gives regex-like backtracking behavior. Users of regex observe poor performance with such patterns and are encouraged to use "atomic" or "possessive" constructions instead. The choice and repetition operators in RPL are possessive by default. Thus, definition 2 above is preferred because it will not backtrack. Definition 3 uses the RPL macro `Find`, which generates a pattern that searches for the target pattern `P` instead of matches `P`. It is implemented, like other Rosie macros, as an abstract syntax tree (AST) transformation.

Compiling definition 2 in RPL using Rosie 1.4 produces the bytecode in Figure 1, which will search the input for the first occurrence of the pattern `"Om"`:

```
 0  testchar O - on failure JMP to 8    // peek without consuming input
 2  choice - on failure JMP to 8        // push onto backtracking stack
 4  any                                 // consume one character (the 'O')
 5  char m                              // consume 'm'
 6  commit and JMP to 11                // commit to choice (success)
 8  any                                 // consume one character
 9  call to 0                           // try again
11  ret                                 // end of pattern
```

Figure 1: Instruction vector produced by Rosie 1.4. The RPL code in definition 2 produces these bytecode instructions.

Using three VM instructions to search for the first character of a pattern is inefficient. The solution is a form of instruction fusion: combine the parts of the loop into a single instruction, `Find1`, which skips ahead in the input to the target character, and fails if none is found.

Compiling definition 3 in RPL using Rosie 1.4 produces the same bytecode shown in Figure 1, because `find:P` macro-expands into definition 2. However, in Rosie 2.0, `find:P` is no longer a macro. It is a built-in operator that, when compiled, produces bytecode containing our new VM instruction.

Compiling definition 3 in Rosie 2.0 produces the following bytecode listing, Figure 2, for the same task: searching the input for the first occurrence of pattern `"Om"`:

```
 0  find1 (1 char) O                    // skip ahead to 'O' (or fail)
 1  choice - on failure JMP to 7        // push onto backtracking stack
 3  char O                              // consume 'O'
 4  char m                              // consume 'm'
 5  commit and JMP to 10                // commit to choice (success)
 7  any                                 // consume one character
 8  jmp to 0                            // try again
10  ret                                 // end of pattern
```

Figure 2: Instruction vector produced by Rosie 2.0. RPL definition 3, when compiled in Rosie 2.0, produces these bytecode instructions. Note the `Find1` instruction at address 0.

The VM instruction `Find1` uses `memchr` to efficiently search for a pattern. When a pattern does not start with a fixed single character, our compiler produces a conservative *first-set* of the pattern, a set containing only characters that may start an occurrence of the pattern. An analogous instruction `Find` advances to the next input character in the *first-set* of the pattern.

Our benchmark suite includes eight patterns, some of which are literal strings like `"Om"`, while others include bounded and unbounded repetitions of single characters and character sets, as well as other operators, like ordered choice, from the PEG specification. While searching for the first occurrence of a pattern is a common task, we show below the performance results from *capturing* all occurrences. Here, *capture* is used as it is in regex to mean that the start and end positions of each match are recorded in a data structure that grows during matching. In Rosie, capturing is handled by constructing a parse tree,

whereas a regex implementation builds a fixed-length list of occurrences. By searching and capturing all occurrences, we avoid any influence on performance of whether the first match occurs early or late in the input. All tests are executed against a 4.1MB file of English-language text.

Typical results are shown in Figure 3, where Rosie 2.0 (code-named *pexl*) is benchmarked against Lua's native PEG library (*lpeg*) and Rosie 1.4 (*rosie)*. Searching for all occurrences of eight benchmarked patterns (named along the x-axis) takes significantly less time using Rosie 2.0 than the other systems. The patterns named *the* and *omega* and *tubalcain* are notable because they each start with a single character and there are relatively few matches to these patterns in our test input. Our new VM instruction performs especially well here.

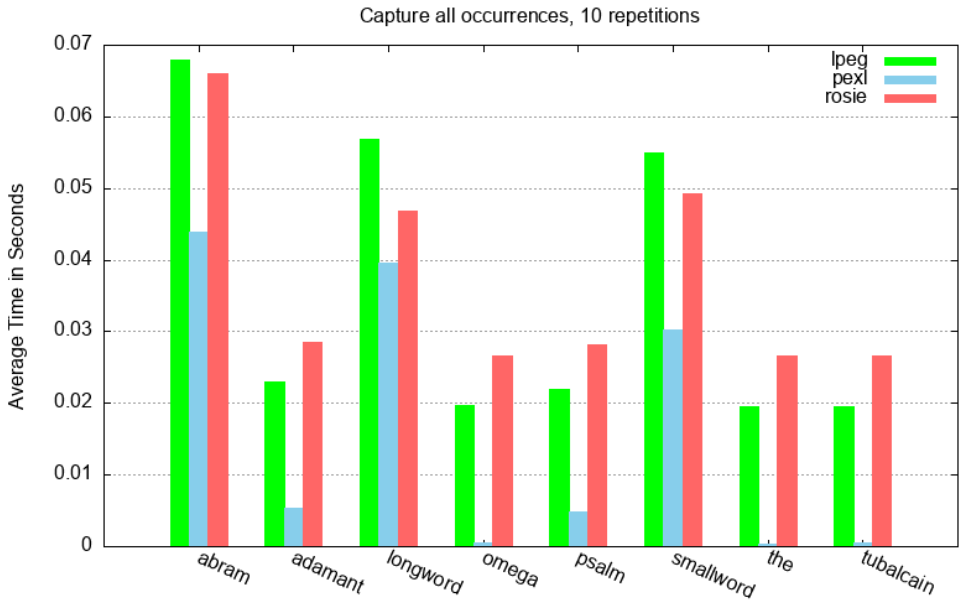

Figure 3: Performance on macOS, arm64 laptop. Lower is better. *Pexl* is the code name for Rosie 2.0. Pattern names from our benchmark suite are shown on the X-axis. The patterns are defined in [6].

Benchmarking is an art whose nuances are beyond the scope of this paper. In Figure 3, we show the average of 10 executions of 3 different systems (*lpeg, pexl,* and *rosie*) each searching for all occurrences of a pattern in the same input text. Results for eight patterns are shown. Three warm-up executions were executed before measurements began. The time reported is the total of system and user time for the process as reported by the operating system. Finally, each benchmarked program performs a complete search of the input text 10 times, so that the total run time measured for an execution consists largely of pattern matching and only a small amount of file I/O and argument processing.

3 Conclusion and Future Work

DSL implementations can leverage design and implementation techniques for general programming languages to great benefits. The fused `Find` instruction produces significant performance increases across our benchmark suite. In ongoing work, we will transform AST shapes corresponding to the three definitions from Section 2, above, into semantically equivalent ones that utilize `Find`, so that the developer need not use the `Find` operator explicitly.

In ongoing work, we are exploring the impact of using single instruction, multiple data (SIMD) (vector CPU) instructions to speed up the `Find` implementation. Early results suggest that SIMD-based character set searches perform well when the frequency of matches is low. For example, searching for punctuation marks in English text performs well with SIMD instructions, as opposed to alphanumeric characters. Inspired by the simple saturating counter used in CPU branch prediction, we intend to explore

dynamically switch between SIMD and non-SIMD character searches depending on the frequency of character matches.

A frequent request is that we benchmark Rosie against regex implementations. We are interested in doing this, though the endeavor is complicated by the variety of readily available regex dialects, each of which provides a unique feature set. Furthermore, most regex implementations provide less power than Rosie, in the sense that they recognize a smaller set of formal languages. This situation requires that benchmarks use only patterns that can be represented in all systems being compared. Indeed, this is the approach taken in [6], where Ierusalimschy demonstrates in Section 5 that *lpeg* is faster (by several orders of magnitude) than a handful of popular regex systems. In our current work, we compare our work with *lpeg*, so we hypothesize a favorable comparison between Rosie 2.0 and popular regex systems, though, importantly, we have not yet run such comparisons.

Acknowledgements

We wish to thank Alex Boots, Luke Jenquin, Ramón Sanchez, and Meles Meles for the development of our benchmark suite, which enables us to easily execute a wide array of comparisons. Their work enabled automatic generation of Figure 1, from test suite execution to statistical calculations to graphing.

References

[1] Bryan Ford. 2004. Parsing expression grammars. *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (January 2004). DOI:http://dx.doi.org/10.1145/964001.964011

[2] Frances E. Allen and John Cocke. 1971. A Catalogue of Optimizing Transformations. *IBM Thomas J. Watson Research Center* (1971), 1–30.

[3] Louis G. Michael, James Donohue, James C. Davis, Dongyoon Lee, and Francisco Servant. 2019. Regexes are hard: Decision-making, difficulties, and risks in programming regular expressions. *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2019), 415–426. DOI:http://dx.doi.org/10.1109/ase.2019.00047

[4] Peipei Wang and Kathryn T. Stolee. 2018. How well are regular expressions tested in the wild? *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2018). DOI:http://dx.doi.org/10.1145/3236024.3236072

[5] Peipei Wang, Jamie A. Jennings, Chris Brown, and Kathryn T. Stolee. 2020. An Empirical Study on Regular Expression Bugs. *International Conference on Mining Software Repositories (MSR)* (2020), 1–11. DOI:https://doi.org/10.1145/3379597.3387464

[6] Roberto Ierusalimschy. 2009. A text pattern-matching tool based on parsing expression grammars. *Software: Practice and Experience* 39, 3 (2009), 221–258. DOI:http://dx.doi.org/10.1002/spe.892

[7] Kayla Sanderson and Jamie Jennings. 2023. Moving Beyond Parsing Expression Grammars. *The Southeast Regional Programming Languages Seminar 2023* (October 2023).