Moving Beyond Parsing Expression Grammars

Kalya Sanderson, Jamie Jennings
kgsande2@ncsu.edu, jjennings@ncsu.edu

1 Introduction

Regular Expressions (regexes) are a powerful tool, commonly used [1, 4] for text search and input validation (ensuring the entire input matches). Input validation is a key use case because it regularly appears near the top of annual rankings of software vulnerabilities. But regexes have many drawbacks. They are under-tested [3], hard to write and maintain [1, 5], and are not composable (unlike the regular languages that inspired them). Even worse, they lack portability [4] across programming languages.

Applying techniques from the design and implementation of programming languages can yield a pattern-based search/match technology that avoids the regex pitfalls. We propose a regex replacement which is designed like a modern programming language to be better suited to software development "in the large": the Rosie Pattern Language (RPL) (https://rosie-lang.org). RPL is a Domain-Specific Language (DSL) for pattern-based matching and searching that is loosely based on Parsing Expression Grammars (PEG) [2]. RPL has a concrete syntax, composable patterns, lexical scopes, shareable pattern packages, and Unicode support[1]. A built-in unit test framework helps document patterns and catch regressions, both vital features when patterns are developed by many people over the lifetime of a software project.

Rosie 1.x has been in use outside of IBM (where the project began) since 2018. In the Rosie 2.0 project, we are addressing lessons learned about language design; we are also re-architecting the implementation to look more like a traditional compiler. Our compiler transforms patterns into bytecode for a pattern matching virtual machine like that of [6]. The new architecture enables adoption of classic compiler optimizations to our DSL, including tail recursion elimination, function inlining, and loop unrolling.

Importantly, as we develop Rosie 2.0, we are experimenting with new language features. In this paper, we present two operators that are not part of the PEG specification: a *back reference* feature that operates on (partial) parse trees at runtime, and a *match predicate* feature that conditions the success of a match on a recursive application of the matching algorithm to text matched by a prior non-terminal.

2 Back-references

Back-references appeared in 1979 when Al Aho added captures to Unix regex. A capture is the span of input matched by a designated sub-expression of the regex. The sub-expression, marked by enclosing it in parentheses, is called a *capture group*. In regex implementations, captures form a flat list. If, due to repetition in a regex, a capture group matches more than once, the new match overwrites the old one. While perhaps disadvantageous to the regex user, this semantics is friendly to the implementer of the regex matcher, who may pre-allocate an array to hold captures before matching begins. The number of captures is fixed syntactically by the regex itself. Building a capture list is a side-effect of regex matching.

The reference part of a *back reference* is merely an index into the list of captures. Back-references in regex are often denoted syntactically as, e.g. `\2`, which causes the regex engine to match the contents of the second capture. Information about captures, back references, and other regex

[1] Our current Unicode supports both full and simple case folding, and the ability to match characters that have any standard Unicode property, e.g. Greek script, or whitespace, or upper-case. In Rosie 1.4, the only supported encoding is UTF-8, though Rosie 2.0 is designed to handle the other encodings as well.

features is plentiful online; we caution readers to be aware that the semantics of many regex features can vary from one implementation to the next [4].

In a grammar-based system like Rosie, a parse tree node is the analog for a regex capture, storing the input span that matched a sub-expression – specifically, a non-terminal in the grammar. We created an experimental back-reference feature in Rosie 1.2, which we are now refining. Since parse tree nodes are named for non-terminals, a back-reference in RPL looks like `backref:P`, where `P` names the target non-terminal of the back-reference operator. The RPL definitions in Figure 1, below, can be used to match a simplified form (for ease of exposition) of HTML, with nested tag pairs[2]. Note that the slash "/" denotes possessive ordered choice in Parsing Expression Grammars.

```
tagname = [:alpha:]+                               // one or more letters
starttag = { "<" tagname ">" }                     // <tagname>
endtag = { "</" backref:tagname ">" }              // </tagname>
grammar html = { starttag html endtag } / "" end   // ϵ is denoted "" in RPL
```

Figure 1: RPL grammar for a simplified version of HTML.

The key question in the back-reference design was how to find the appropriate instance of the target non-terminal in a parse tree that is (1) under construction, and (2) may contain multiple occurrences of the target non-terminal, possibly even nested (recursive) ones. When Rosie encounters `backref:P`, its VM must first locate a prior capture of non-terminal `P`. That is, the VM looks through the partially-completed parse tree for a `P` node that is *complete*. In the absence of recursion, the VM simply finds the most recent *complete* match for the specified non-terminal. Unlike in regex, where captures with the same name (or number) overwrite prior ones, a parse tree may contain many instances of the same non-terminal.

Note that recursion must be handled carefully[3]. Consider the case in which the VM is currently building a subtree rooted at `P`, then it does not know whether `P` will match at all, let alone the input span it will match. So the VM must ignore ancestors named `P` of the parse tree node it is currently building, as the matching for those non-terminals is *incomplete.* But there may instances of `P` nodes elsewhere.

Suppose the parse tree search does find a complete node named `P`. That node is annotated with the input span (e.g. start/end positions) that `P` matched. This span is a sequence of symbols that `backref:P` must match (consume) ahead of the current input position. Our PEG back reference feature thus provides a function analogous to its counterpart in regex languages.

We hope that the brief overview of the back reference feature given here is sufficient to motivate consideration of other pattern matching features that utilize the parse tree at run-time, while it is under construction. In the next section, we propose a novel feature that does just this, the *match predicate*.

[2] Any Dyck language would suffice for an example; the tags of HTML form one that is familiar to many.
[3] See https://jamiejennings.com/posts/2023-10-01-dont-look-back-3/ for a more detailed treatment of the algorithm.

3 Match predicates

Parsers for programming languages often support "semantic actions" which are executed during parsing and can be effectful, e.g. building a symbol table or halting parsing because some test fails. Our DSL is not embedded, and we are not building a parser generator for programming language implementation. Yet, we have identified a use case for conditioning the success of a match on a separate predicate, namely to facilitate pattern reuse. Generally, developers prefer to reuse patterns from libraries over having to modify their definitions (as they do with code), and match predicates facilitate that.

A `where` predicate is a pattern with three arguments, a pattern X, a path Y, and a pattern Z, such that `X where(Y, Z)` means, operationally, *"match X, then test that the input span matched by non-terminal Y is itself matched by Z"*. The argument Y is a parse tree path, with implicit root X, that uniquely identifies a non-terminal within the (completed) parse tree of X. The ability to specify an arbitrary pattern for Z means that our matcher is invoked recursively in concept, though the implementation itself does not use the C runtime stack for this.

While deciding the format of Y, the tree path, we chose a variant of JSONPath [7]. Rosie has the option to output a match (parse tree) as a JSON object and users may decide on a correct path for their predicate easily by previewing the structure of the JSON objects returned by pattern X.

Another design decision involved how we might apply the `where` predicate while the parse tree is still under construction. By applying the predicate immediately after matching X, and requiring Y to be rooted at X, we can ensure that either the node referred to by Y exists and is unique, or that this particular parse for X did not contain Y, in which case the `where` predicate will fail for lack of a referent.

Consider this example: The RPL `net` library contains patterns for network-related items such as URLs, email addresses, and ip (v4 and v6) addresses. A `where` predicate allows users to search for URLs that meet certain criteria, without altering or duplicating the definition in the `net` library. Thus a user might write this pattern to search for URLs where the registered-name part is "catalog.data.gov":

```
P = net.url where(authpath/authority/host/registered-name, "catalog.data.gov")
```

Figure 2: Example of a match predicate, indicated by the keyword `where`.

Note that the string "catalog.data.gov" is in fact the syntax for an RPL pattern that matches that string. The pattern `P` will match, e.g. "https://catalog.data.gov/dataset/electric-vehicle-population-data". While in this example, and in our current implementation, we restrict the pattern Z to literal strings, we plan to allow for an arbitrary pattern to be matched.

3 Conclusion and Future Work

Our approach to the design and implementation of Rosie and RPL is heavily based on the history of programming language development. Scheme inspired our choice of lexical scope, our package model, and hygienic macros that operate on ASTs. Static analysis, ahead-of-time compilation, and an automated test framework are especially important for large projects developed by many people over a long period of time, such as open source and commercial software. All are common traits of Turing-complete language implementations, and often lacking in DSLs where they confer many of the same benefits.

Focusing specifically on our *matching predicates,* we note that the ability to match a particular span of input twice (or more) suggests a connection to formal language intersection. Regular languages are closed under intersection, and context-free languages are not. What about Parsing Expression Grammars, and the classes of grammars that we, like others, have been developing after starting with PEGs and finding them insufficient? We are keenly interested in having as simple a formal model as possible for our system, which is meant to facilitate development of reliable, performant software. Our work continues with implementation; solving problems like input validation and text search; and building formal models and semantics.

Acknowledgements

We wish to thank Andrew Farkas for his work implementing the re-entrant feature of our VM in 2022, on which our current implementation effort to implement matching predicates relies.

References

[1] L. G. Michael, J. Donohue, J. C. Davis, D. Lee and F. Servant, "Regexes are Hard: Decision-Making, Difficulties, and Risks in Programming Regular Expressions," *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, San Diego, CA, USA, 2019, pp. 415-426, https://doi.org/10.1109/ASE.2019.00047

[2] Bryan Ford. 2004. Parsing expression grammars: a recognition-based syntactic foundation. In Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '04). Association for Computing Machinery, New York, NY, USA, 111–122. https://doi.org/10.1145/964001.964011

[3] Peipei Wang and Kathryn T. Stolee. 2018. How Well Are Regular Expressions Tested in the Wild?. In Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3236024.3236072

[4] James C. Davis, Louis G. Michael IV, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2019. Why Aren't Regular Expressions a Lingua Franca? An Empirical Study on the Re-use and Portability of Regular Expressions. In Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26, 2019, Tallinn, Estonia. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3338906.3338909

[5] Peipei Wang, Chris Brown, Jamie A. Jennings, and Kathryn T. Stolee. 2020.
An Empirical Study on Regular Expression Bugs. In 17th International Conference on Mining Software Repositories (MSR '20), October 5–6, 2020, Seoul, South Korea. 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. https://doi.org/10.1002/spe.892

[7] See, for example: Stefan Goessner. 2007. JSONPath - XPath for JSON. Retrieved August 8, 2023 from: https://goessner.net/articles/JsonPath/