Parser Generation in Perl: an Overview and Available Tools

Hugo Areias[1], Alberto Simões[2], Pedro Henriques[1], and Daniela da Cruz[1]

[1]Departamento de Informática, Universidade do Minho
[2]Escola Superior de Estudos Industriais e de Gestão, Instituto Politécnico do Porto
hugomsareias@gmail.com,pedrorangelhenriques@gmail.com
alberto.simoes@eu.ipp.pt,danieladacruz@di.uminho.pt

Abstract. There are some modules on Comprehensive Perl Archive Network to help with the parser generation process in Perl. Unfortunately, some are limited, only supporting a particular parser algorithm and do not covering some developer needs. In this document we will analyse some of these modules, testing them in terms of performance and usability, allowing a proper evaluation of the results and careful considerations about the state of art of parser generation using Perl.

Keywords: Parser generators, Perl, grammars

1 Introduction

The primary aim of this paper is to provide an overview of the particular condition of parser generation in Perl and analyse some of the available tools.
In the Comprehensive Perl Archive Network (CPAN[1]) there are some modules available to automate the process of generating a parser. However, the user should choose carefully according to his needs and because the lack of maintenance and efficiency of some of them.
We chose four tools, the most used, the more robust, the more elaborate and the more recent:

- **Parse::RecDescent** (v 1.962.2) – one of the most used tools, generate on-the-fly a recursive-descent parser;
- **Parse::Yapp** (v 1.05) – can be compared with the well known `yacc` parser generator tool in terms of algorithm and syntax;
- **Parse::Eyapp** (v 1.154) – an extended version of `Parse::Yapp` including new recursive constructs;
- **Regexp::Grammars** (v 1.002) – an implementation of the future Perl 6 grammars[2]

[1] http://www.cpan.org/
[2] This tool is only supported in recent Perl versions (> 5.10).

1.1 Parse::RecDescent

`Parse::RecDescent` [3] supports $LL(1)$ parsers [1] and generates recursive-descent parsers on-the-fly. It is a powerful module that provides useful mechanisms to create parsers with ease, such as auto-actions (automatically adding pre-defined actions) and named access to semantic rule values (allowing the retrieve of data from an associative array using the symbol name instead of the usual array indexes). To create the parser, `Parse::RecDescent` generates routines in runtime, doing the lexical and syntactic analysis, and achieving the results on the fly. The drawbacks are the incapacity to deal with left recursion and its efficiency when dealing with large inputs. Therefore, it is not recommendable for cases where the performance is an issue.

1.2 Parse::Yapp

`Parse::Yapp` [5] is one of the oldest parser generators in Perl and probably still one of the most robust. It is based on `yacc` [2]. Just like `yacc`, it is well known for supporting $LALR$ parsers [1] and for its parsing speed. Such traits makes it an obvious choice for the users. As an addition, it also provides a command-line script that, when executed over an input grammar file, generates a Perl Object Oriented (OO) parser. This module only supports *Backus-Naur Form* (BNF) rules to write the grammar. Also, `Parse::Yapp` does not include lexical analyser features, forcing the user to provide one. Gratefully, there are some useful modules on CPAN to help in this process, such as Text::RewriteRules [8].

1.3 Parse::Eyapp

`Parse::Eyapp` [6] is an extension of `Parse::Yapp`. Just like `yapp`, it only supports $LALR$ parsers, but is able to parse extended BFN rules. While it introduces a lot of new useful features, it still keeps the same structure of `Parse::Yapp` allowing parsers made for the second to run when executed by the first. The most relevant features from `Parse::RecDescent` implemented in this module include auto-actions and named access to semantic rule values.

1.4 Regexp::Grammars

`Regexp::Grammars` [4] is a module that tries to implement Perl 6 grammar support with Perl 5. This is possible given the new recursive regular expressions introduced in Perl 5.10. The module extends the regular expressions in a way that makes them similar to typical grammars. While it is easy to use, it has some efficiency problems, very similar to the presented for `Parse::RecDescent`, given that it also generates recursive-descent parsers. Also, `Regexp::Grammars` creates automatically abstract data structures for the grammar, reducing the number of visible semantic actions.

2 Analysis and Tests

Three different grammars[3] were chosen to help testing the four modules described earlier: The Swedish Chef, a simple grammar but relatively large with an high number of semantic actions; The Lavanda, a Domain Specific Language (DSL) to describe the laundry bags daily sent to wash by a launderette company; and an highly recursive grammar to match s-expressions. These tests were performed by a machine with an *Intel Pentium 4* with a clock rate of 3.4 GHz and 3Gb of *RAM*.
Looking to the following tables it is possible to understand the most efficient modules. `Parse::RecDescent` and `Regexp::Grammars` both use regular expressions to perform the lexical analysis but they store the parsing functions in memory as they are generated on-the-fly. So, even with the advantages of using regular expressions, these modules take too long. This also has to do with a few recursive-descent parser limitations in Perl.

Table 1. User time evolution of the four approaches for the Lavanda grammar.

Nr. Lines	Parse::Yapp	Parse::Eyapp	Parse::RecDescent	Regexp::Grammars
10	0.031 s	0.090 s	0.123 s	0.069 s
100	0.115 s	0.184 s	0.258 s	0.163 s
1000	1.240 s	1.380 s	4.041 s	1.399 s
10000	34.896 s	37.640 s	331.814 s	out of memory
100000	> 2488.348 s	> 4973.639 s		
1000000				

Table 2. Memory consumption (in megabytes) of the four approaches for the Lavanda grammar.

Nr Lines	Parse::Yapp	Parse::Eyapp	Parse::RecDescent	Regexp::Grammars
10	0.933	3.866	3.583	3.490
100	1.934	4.867	4.607	22.545
1000	12.141	15.214	15.175	181.809
10000	108.697	113.242	115.383	out of memory
100000				
1000000				

Table 3 show a final analysis of the modules. From these results, it is easy to realise that `Parse::Yapp` is the most efficient module available for Perl, mainly due to the fact that it is based on *LALR* grammars, slightly more powerful than *LL* algorithms. It also offers the best support for integration of the parser with other code. In the other hand, it does not offer any support for attribute grammars and for the construction of AST. The lack of documentation makes it not very easy to start with, increasing the development time. Also, it does not provides the best support for semantic actions when compared to the other modules and it requires the lexical analyser to be provided by the user.

[3] All grammars, test files and generated parsers are available at `http://www.di.uminho.pt/~gepl/PARSINGinPERL`

Table 3. Module Analysis.

Module	Debugging	Generated Parser Readability	Integration with External Code	Development Time
Parse::Yapp	+/−	+/−	++	+/−
Parse::Eyapp	+/−	+/−	+	+/−
Parse::RecDescent	+	NA	+	−
Regexp::Grammars	++	NA	+/−	−−

3 Conclusions

Parser generators in Perl still lacks valuable mechanisms to make them challengeable when compared with other languages, like C. There is no valid support for attribute grammars and, according to the research made, there is only one module on CPAN that supports attribute grammars that, however, lacks of maintenance for several years now.
The modules that support recursive-descent parsers provide several useful mechanisms but due to the lack of efficiency, they are not recommendable for processing large input streams. *LALR* parsers provide a more efficient solution, however the lexical analyser must be provided by the user and their efficiency is not the best when compared with other language solutions [7].
An alternative solution could be combining the Perl modules with other tools written in another languages to achieve better results. This solution would require a bridge between both tools and its evaluation would be dependable on the effort and difficulty level of implementing this bridge. This is precisely the objective of a master thesis that aims at retargeting AnTLR (a well known LL(K) compiler generator from attribute grammars) to generate Perl compilers.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley, 1986.
2. Stephen Johnson Bell and Stephen C. Johnson. Yacc: Yet another compiler-compiler. Technical report, 1979.
3. Damian Conway. Parse::recdescent. `http://search.cpan.org/dist/Parse-RecDescent/lib/Parse/RecDescent.pm`, 1997.
4. Damian Conway. Regexp::grammars. `http://search.cpan.org/~dconway/Regexp-Grammars-1.001005/lib/Regexp/Grammars.pm`, 2009.
5. Francois Desarmenien. Parse::yapp. `http://search.cpan.org/dist/Parse-Yapp/lib/Parse/Yapp.pm`, 1998.
6. Casiano Rodriguez-Leon. Parse::eyapp. `http://search.cpan.org/dist/Parse-Eyapp/lib/Parse/Eyapp.pod`, 2006.
7. Alberto Simoes. Parsing with perl. Copenhaga, Aug 2008. Yet Another Perl Conference Europe.
8. Alberto Simoes and José Joao Almeida. Text::rewriterules. `http://search.cpan.org/~ambs/Text-RewriteRules-0.21/lib/Text/RewriteRules.pm`, 2004.