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. Perl programming language is well known for its flexibility and for providing powerful text processing mechanisms, along with a powerful regular expressions engine. Although regular expressions are quite helpful in the construction of parsers and Perl 5.10 already supports recursive regular expressions, these mechanisms are not enough to recognise complex languages. This makes the process of constructing parsers painful and time consuming. But, as for other languages, Perl also have some parser generator tools.
There are some modules on CPAN 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

In the early days of language recognition, parser algorithms were very complex and, the parser checks and the semantic actions were, typically, mixed together, making the parser extremely hard to maintain. Also writing a parser by hand was a painful and time consuming job. For this reason a few parser generators started to be developed with the goal of simplifying this process.

A parser generator is a program that, based on a grammar, creates a parser for that syntax. It is believed that using a parser generator makes it easier to achieve a more correct, simpler, efficient and easier to read parser, than programming the parser manually [1].

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.

This paper is organised in five sections. First, section 2 gives a quick overview of the state of the art of parser generation in Perl. Follows section 3 presenting a few of the available Perl modules main functionalities. Section 4 evaluates the chosen modules in terms of efficiency. Lastly, some conclusions and comments on parser generation in Perl are discussed in section 5.

2 Parser Generation in Perl

The Perl language provides a few powerful mechanisms that can be proven very useful on the process of generating parsers. The expressiveness of regular expressions can be very helpful when performing lexical analysis, or even to help on the parsing process, when the user is dealing with complex or hierarchical input. Regular expressions also improve readability and flexibility of the parser, decreasing its complexity.

Despite the regular expressions use advantages, parser generation in Perl requires more mechanisms, mostly to decrease the development time, by generating, based on a given grammar, the parser algorithms that are too complex to be hand made and to decrease the complexity and consequently the effort of writing a parser for a specific language [1].

Perl has some modules developed for parser generation. However, there are still a few efficiency barriers to be overcame, such as the high memory dependability by some parser algorithms (mainly recursive-descent) or the large amount of memory consumed by the parsers with huge input streams.

Although Perl 5.10 introduced recursive regular expressions, the same problems pointed above still arise.

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

3.1 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

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

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. Figure 1 shows a sample grammar.

```
my $grammar = new Parse::RecDescent q{
    lisp: sExp                      { $return = $item{sExp} }

    sExp: (/\d+/)                   { $return = $item[1] }
        | (/[A-Za-z]+/)             { $return = $item[1] }
        | '(' sExp(s?) ')'          { $return = $item[2] }
};
```

Fig. 1. `Parse::RecDescent` parser for s-expressions.

3.2 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.

```
lisp     : sExp             { return $_[1][0] }
         ;
sExp     : NUM              { return [ $_[1] ] }
         | PAL              { return [ $_[1] ] }
         | '(' sExplist ')' { return [ $_[2] ] }
         ;
sExplist : sExp sExplist    { push @{$_[1]}, @{$_[2]}; return $_[1] }
         |                  { return [] }
         ;
```

Fig. 2. `Parse::Yapp` parser for s-expressions.

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 [10].

3.3 Parse::Eyapp

`Parse::Eyapp` [7] is an extension of `Parse::Yapp`. Just like `yapp`, it only supports *LALR* parsers, but is able to parse extended BNF 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.

```
        lisp  : sExp           { return $_[1] }
              ;
        sExp  : NUM            { return $_[1] }
              | PAL            { return $_[1] }
              | '(' sExp * ')'  { return $_[2] }
              ;
```

Fig. 3. `Parse::Eyapp` parser for s-expressions.

3.4 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.

```
  my $parser = qr{
    <lisp>

    <rule:lisp>    <MATCH=sExp>

    <rule:sExp>    (\d+)              (?{ $MATCH = $CAPTURE })
                 | ([A-Za-z]+)        (?{ $MATCH = $CAPTURE })
                 | \( <[MATCH=sExp]>* \)
  }x;
```

Fig. 4. `Regexp::Grammars` parser for s-expressions.

In the other hand, `Regexp::Grammars` creates automatically abstract data structures for the grammar, reducing the number of visible semantic actions.

4 Analysis and Tests

Three different grammars were chosen to help testing the four modules described earlier: The Swedish Chef, a simple but relatively large grammar, 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.

4.1 Readability of the Grammars and Generated Parser

The readability of a grammar is crucial for a better comprehension of the target language and for maintenance purposes. To ensure a satisfactory readability a parser generator should provide, at least, a simplistic, organised and well structured syntax.

All the four modules have a similar syntax, however there are some relevant differences between them. `Parse::Yapp` is the only module that does not support extended-BNF rules. Therefore, grammars containing lists will be larger in length when compared to the grammars of the remaining modules.

Also, `Parse::Yapp` is the only of the four modules that does not support named access to semantic rule values, a functionality that helps improving the readability of semantic actions, allowing the access to the parsed data through the identifier name of the terminal or non-terminal symbol. Notice how the data was obtained in the first production o example 1.

`Regexp::Grammars` has two interesting characteristics that can help improve the readability of the grammar. First, the module alone generates an abstract syntax tree, so the user does not need to add semantical actions to return the data. Keep in mind that this can also be done with `Parse::Eyapp` and `Parse::RecDescent` by adding auto-actions. Secondly, the user can specify different types of productions to differentiate them, as for example, to distinguish rules and tokens.

The readability of the generated parser is also important, especially if the user does not have access to the grammar, as happens with standalone parser applications. A confusing parser can lead to language misunderstandings and waste of time with maintenance.

Only the two modules (`Parser::Yapp` and `Parser::Eyapp`) were analysed for parser readability purposes, as the other two generate code in run-time that is not possible to serialise for future use (note that this is also an efficiency problem).

The parser generated by `Parser::Yapp` is very extensive and not easy to understand for a common user. It is also hard to identify the original grammar and to comprehend the target language if the user does not have knowledge of Perl. As `Parse::Eyapp` is a `Parser::Yapp` extension, the generated code is also not easy to understand. They are mostly comprised of look-ahead tables. However, `Parse::Eyapp` generates an auxiliary data structure where the grammar can be found.

4.2 Lexical Analyser

The lexical analyser is responsible for matching the input text with the tokens, or terminal symbols, of a grammar, supplying to the syntactic analyser a sequence of tokens. The syntactic analyser, will check if the sequence is disposed of in accordance with a given grammar [1].

When it comes to the lexical analysis, the `Parse::Yapp` and `Parse::Eyapp` modules have some disadvantages when compared to `Parse::RecDescent` and `Regexp::Grammars`, as they do not provide the lexical analyser. Therefore, it must be hand made by the user. The second two modules use regular expressions to perform the lexical analysis saving time to the user. Moreover, they mix the lexical analyser with the grammar, making the code more readable.

Not having an embedded lexical analyser can be seen as an advantage, in case the user is looking for more flexibility or better efficiency. For example, combining the well known Unix lexical analyser `flex` with `Parse::Yapp` can lead to an increase of the parser efficiency and decrease of required memory [8].

4.3 Support for Semantic Actions

After the parser has been generated and the syntactic analysis finished, the user usually defines semantic actions that instruct the parser on what operations should be performed during the process of parsing the input text. While these actions can perform lateral effects, the most common behaviour is the construction of an Abstract Syntax Tree (AST) that can be traversed and analysed after the parsing process.

The four modules provide some mechanisms to support semantic actions, however some are still lacking of satisfactory support. For example, none of these modules provide any kind of mechanisms or support for attribute grammars [11], and only `Parse::Eyapp` gives the possibility of creating a standard AST.

Given Perl built-in data structures (lists and associative arrays), all the modules provide easy mechanisms to manipulate the parsed data, and the construction of complex data structures.

`Regexp::Grammars` provides a list and a hash to access to the semantic rules values, by using indexes or named access, and also provides a collection of useful variables with associated actions, like access to the last parsed token, debugging or get the index of the next token to be matched.

`Parse::RecDescent` allows the definition of auto-actions, and auto-trees that automatically creates a parsing tree for the input text.

4.4 Flexibility of Integration with Other Code

After generating a parser, a user might want to use it in more than one program. The user could replicate the code and use it in every program he wishes, however this will lead to code duplication and time consumption. A good practice is to separate the parser from the other programs [6], storing it as a distinct program or as a library, importing it only when it is needed. For this purpose, Perl supports modularity.

The parsers generated by `Parse::Yapp` and `Parse::Eyapp` are both easy to manage and integrate with other Perl code, since both store the generated parser in a stand-alone Perl module. They also provide an option to allow the parser to be generated as a single application, making it executable and independent of the original code.

`Parse::RecDescent` does not create a module by default like `Parse::Yapp` or `Parse::Eyapp`, however it provides mechanisms to allow it if the user desires so. Activating the `Parse::RecDescent` pre-compiling option will give the user the ability of creating a module containing the generated parser.

Finally, `Regexp::Grammars` takes advantage of the new functionalities of Perl 5.10 and its disposal (only affects the regular expressions constructors) makes very easy its integration into another code. However it does not offers support for modularisation.

4.5 Debugging

Debug tools are always useful to search for errors or conflicts in the grammars, and a good debugger could save a lot of time to the user when solving complex problems. All the analysed modules have debug mechanisms to help solving errors by giving useful information about them or even giving correction suggestions.

With `Parse::Yapp` it is possible to debug the grammar by filling a parameter when invoking the parser. This parameter can assume different values depending on the wanted information, such as error recovery tracing or syntax analysis.

The `Parse::Eyapp` provides a list of methods to output all the relevant information about the grammar, including rules, warnings, errors, conflicts, ambiguities and even the parsing tables. These methods, after collecting the necessary data according to the called method, presents a detailed output with the desired information. In opposite to the `Parse::Yapp`, the output can also be sent to a file.

`Parse::RecDescent` provides a set of global switches that can be set to debug the grammar. The most relevant works like a trace, reporting the parser behaviour and progress to the *standard error* or to a file. Besides, most of the information that every parser generator usually shows, this module has a little functionality that cannot be found in the other modules, as it gives suggestions or hints for possible solutions for the problems found, if any.

The `Regexp::Grammars` has, based on documentation and practical experience, the most powerful and elucidative debugging mechanisms and pleasant output of all four analysed modules. This debugger is based on directives that control the output information sent to *standard error* by default or to a file.

This module, besides the static debugging, also provides an interactive debugger. The available options and their descriptions transcribed below were taken directly from the `Regexp::Grammars` documentation:

<debug: on> - Enables debugging, stops when entire grammar matches
<debug: match> - Enables debugging, stops when a rule matches
<debug: try> - Enables debugging, stops when a rule is tried
<debug: off> - Disables debugging and continues parsing silently

Using these directives, the user can control completely which rules he wants to debug and define when to start and finish the debug process. Both interactive and static debugging present the data in a pleasant, easy to read and well structured tree.

4.6 Performance tests

A parser needs to be, at least, correct, readable and efficient. Readability and sometimes correctness can be discussed due to the diversity of opinions. This hardly happens with efficiency due to the fact that results from tests are often accurate and elucidative. Efficiency is crucial when parsing the input requires the consumption of a large amount of resources.

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.

`Parse::Yapp` and `Parse::Eyapp`, based on LALR parsers, are faster, helped by the fact that the lexical analysers are developed by the user and provided to the modules. However, this introduce a few consequences. The easier lexical analyser that one can write in Perl load all its input to memory, making the parser efficiency to drop. A solution to minimise these effects is to provide a lexical analyser that discards all the input already checked, diminishing the use of memory and increasing the efficiency of both parsers. This solution was applied in the tests. Another option, already mentioned, is to couple the syntactic analyser with a C lexical analyser. This was not done in the context of this article as it would lead to unfair comparisons.

As shown on table 1, none of the modules gave a positive response to an input of *1 000 000* lines of input when parsing the Swedish chef language, and the same happened when parsing the *Lavanda* language (table 2). The huge amount of time consumed by all the generated parsers was crucial to interrupt the tests.

Regarding the parsing of the Swedish chef language, `Parse::Yapp` obtained the best times and finished parsing a file with *100 000* lines after 526.649 seconds (almost 9 minutes!). This is not an impressive time, but it is not bad, also considering that only `Parse::Eyapp` was able to parse the same file. Only these two modules were able to attempt parsing, without a favourable outcome, a file with *1 000 000* lines, since `Parse::RecDescent` was interrupted when parsing a file with *10 000* lines after spent more than an hour trying it, and `Regexp::Grammars` returned an *out of memory* warning when parsing a file with *100* lines!

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

Input Lines	Parse::Yapp	Parse::Eyapp	Parse::RecDescent	Regexp::Grammars
10	0.076 s	0.130 s	0.519 s	0.267 s
100	0.549 s	0.643 s	4.496 s	out of memory
1000	5.290 s	5.994 s	268.803 s	
10000	52.749 s	59.734 s	$>$ 3675.743 s	
100000	526.649 s	602.193 s		
1000000	$>$ 5096.551 s	out of memory		

Table 2. User time evolution of the four approaches for the Lavanda grammar.

Input 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				

Almost the same happens when parsing the *Lavanda* language as can be visualised in table 2. `Parse::Yapp` and `Parse::Eyapp`, again, are the only two modules to attempt the parsing of the file with *100 000* lines. In opposite to the latest test, the tests are interrupted because of the large time consumption. `Regexp::Grammars` obtained better results in this test, but still not near to be considered satisfactory due to the fact that only attained to parse a file with *1 000* lines. `Parse::RecDescent` did better than `Regexp::Grammars`, however, obtained weaker results when compared with the other two modules.

These two tests clearly show that the parsing times of the parsers generated by the tested modules tend to highly increase when dealing with large input streams. This can be seen in table 2, as example, for the `Parse::Yapp` case, it took 34.896 second(s) to parse a file with *10 000* lines, but when parsing a file *10 times* larger, this value suffered a considerable increase to 2 488.348 second(s) (around *73* times larger than the first), before being interrupted.

`Regexp::Grammars` was not properly evaluated in these tests since Perl 5.10 still has some bugs that hindered the analysis. It has some good features and achieved good times when compared to the other modules (it was the second faster to parse the *Lavanda* language, after `Parse::Yapp`), however it lacks efficiency when dealing with large input streams, like the other $LL(1)$ based modules.

To parse the s-expressions language (results in table 3) it was only used one test file containing a large expression (32 377 characters and 7 131 tokens). This grammar is highly recursive, however all the modules achieved acceptable times when performing the task. Like in the other tests, the parser generated by

`Parse::Yapp` was the fastest, followed by `Parse::Eyapp`. `Parse::RecDescent` finished parsing after 4.041 seconds, closely four times more than `Parse::Yapp`. `Regexp::Grammars` could not finish parsing because of a Perl 5.10 bug (already reported).

Table 3. User time spent by the four approaches parsing a large s-expression.

	Parse::Yapp	Parse::Eyapp	Parse::RecDescent	Regexp::Grammars
Time	0.949 s	1.148 s	4.041 s	bus error

Table 4 presents the memory consumption during the parsing of the *Lavanda* test files. According to table 2, the parsers could only undertake the *10 000* or less lines test files, so the sizes above have been discarded. Also it has to be considered the amount of memory that the tool was consuming along with the parser execution.

As the table 4 shows, the parser generated by `Regexp::Grammars` quickly becomes out of memory to parse the files. `Parse::RecDescent` follows the same strategy but it results in less memory consumption. As verified during all the tests, the parsers generated by `Parse::Yapp` achieve the best efficiency results. To do the memory tests it was used the `massif` tool from `valgrind`[3].

Table 4. Memory consumption (in megabytes) of the four approaches for the Lavanda grammar.

Input 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				

Tables 5 and 6 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] http://valgrind.org/

The `Parser::Eyapp` offers useful support for semantic actions, such as auto-actions and AST construction. It is very similar to `Parse::Yapp`, however it is not that efficient, mainly because of the extra features introduced. It also requires the lexical analyser to be provided by the user.

Table 5. Grammar Analysis.

Module	Supported Grammars	Grammar Readability	AGs	AST	Semantic Actions	Lexical Analyser
Parse::Yapp	LALR	+	No	No	+	No
Parse::Eyapp	LALR	+	No	Yes	++	No
Parse::RecDescent	LL(1)	++	No	No	++	Yes
Regexp::Grammars	LL(1)	++	No	No	++	Yes

The `Parse::RecDescent` lacks of support for attribute grammars and efficiency when dealing with large input streams. In the other hand, the lexical analyser is integrated and it offers a large number of useful features to help with the semantic actions.

The `Regexp::Grammars` it has a lot of good features, however it is very poor in efficiency. It cannot be said much of this module since it was not tested properly at the same stage than the other modules.

Table 6. Module Analysis.

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

5 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 [9].

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.

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. Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.
7. Casiano Rodriguez-Leon. Parse::eyapp. `http://search.cpan.org/dist/Parse-Eyapp/lib/Parse/Eyapp.pod`, 2006.
8. Alberto Simoes. Cooking perl with flex. *The Perl Review*, 0(3), May 2002.
9. Alberto Simoes. Parsing with perl. Copenhaga, Aug 2008. Yet Another Perl Conference Europe.
10. Alberto Simoes and José Joao Almeida. Text::rewriterules. `http://search.cpan.org/~ambs/Text-RewriteRules-0.21/lib/Text/RewriteRules.pm`, 2004.
11. William M. Waite. Use of attribute grammars in compiler construction. In *WAGA*, pages 255–265, 1990.