Generating flex Lexical Scanners for Perl Parse::Yapp*

Alberto Simões[1], Nuno Carvalho[2], and José João Almeida[3]

1 Centro de Estudos Humanísticos, Universidade do Minho
Campus de Gualtar, Braga, Portugal
ambs@ilch.uminho.pt

2 Departmento de Informática, Universidade do Minho
Campus de Gualtar, Braga, Portugal
narcarvalho@di.uminho.pt

3 Departmento de Informática, Universidade do Minho
Campus de Gualtar, Braga, Portugal
jj@di.uminho.pt

Abstract

Perl is known for its versatile regular expressions. Nevertheless, using Perl regular expressions for creating fast lexical analyzer is not easy. As an alternative, the authors defend the automated generation of the lexical analyzer in a well known fast application (flex) based on a simple Perl definition in the syntactic analyzer.

In this paper we extend the syntax used by `Parse::Yapp`, one of the most used parser generators for Perl, making the automatic generation of flex lexical scanners possible. We explain how this is performed and conclude with some benchmarks that show the relevance of the approach.

1998 ACM Subject Classification D.3.4 Processors

Keywords and phrases flex, Perl, yapp, lexical analyzer

Digital Object Identifier 10.4230/OASIcs.SLATE.2012.41

1 Introduction

There are diverse tools to write syntactic parsers in Perl. [1] describes some of these tools (like `Parse::Yapp`, `Parse::Eyapp` and `Parse::RecDescent`) and explain how they work. Also, [2] describes work in progress for an ANTLR generator for the Perl language. And recently a new module, named Marpa[1] also appeared. All these tools have a common denominator: they are syntactic parsers and little attention is given to the lexical analyzer.

While Perl is great with regular expressions, it is not that good when these expressions need to match a text file. Regular expressions were designed to match in a string, and to make them work with a file there are only two options:

- read the entire file to memory as a huge string, and apply regular expressions there: has the disadvantage of the memory used (as all text is loaded to memory), and if regular expressions are not written with efficiency in mind, they can take some time to match on huge strings.

* This work was partially supported by grant SFRH/BPD/73011/2010 funded by Science and Technology Foundation, Portugal.

[1] Available as a pure Perl implementation at `https://metacpan.org/release/Marpa` and as an hybrid Perl and C implementation at `https://metacpan.org/release/Marpa-XS`.

- alternatively one can load chunks of text from the file, and try to use regular expressions in those chunks. If the string does not match one need to check why, as it might required another chunk to match.
  Of course most languages are line-oriented, making it easier to write this approach without exhausting memory. Nevertheless, this is not a generic solution.

This document proposes the use of the well known fast lexical analyzer flex [11] as the lexical analyzer together with one of the above mentioned Perl syntactic parser (namely `Parse::Yapp` and `Parse::Eyapp`), as described in [12][2]. For that, the `Parse::Yapp` language was enriched with extra information, and an automatic flex code and Perl–C glue code generators.

We start by discussing the state of the art on lexical scanners (section 2) and on syntax analyzers (section 3) in Perl.

Section 4 explains how C and Perl code can work together, and then specifies on how flex and `Parse::Yapp` can be glued together. The code generator (`Parse::Flapp`) is presented in section 5, followed by some tests and evaluations (section 6). Finally we draw some conclusions.

2 Perl Lexical Analyzers

Although most Perl written parsers use regular expressions as lexical analyzers, there are a couple of available modules that can be used as lexical analyzers, but most are too rudimentary, and that functionality is a side effect of their main goal. In this section we will compare on pure Perl approach and the well known flex. This comparison is mostly as a motivation for the usage of flex together with Perl, as we would not expect for a Perl implementation to be near as efficient as a C implementation.

- **Parse::Lex** is the only lexical analyzer available written entirely in Perl, and developed for that purpose;
- **flex**, while not a Perl approach, is the tool we are gluing in with Perl, and therefore a comparison is relevant.

We will first explain briefly how `Parse::Lex` works (we will skip flex on this explanation, as it is well known), and then present some comparison values in terms of memory usage and time efficiency.

2.1 Writing Lexical Analyzers in Perl

The common approach to write a lexical analyzer in Perl is to use regular expressions. The text to be analyzed is loaded into memory, and regular expressions are used to match and detect tokens. As tokens should be supplied to the syntactic analyzer in the same order they appear, these regular expressions can be anchored to the beginning of the string, making the matching faster. Also, given that strings are stored in memory, every time a token is found, the matching string can be replaced by the empty string, reducing the size of the string, and freeing some memory (although this can be a good solution for reducing memory usage, it will take some extra time).

[2] This article being cited, published in The Perl Review, is a tutorial of gluing a flex generated scanner with Perl code. It is a step by step guide of the needed code. The current work automatizes that process generating automatically the needed source code.

Unfortunately this approach has a big drawback, that is the memory used. Given this, we decided not to compare this vanilla technique. It is true that for some specific rules, and some specific input data, lexical analyzers like the ones generated with flex will need to read the entire document to memory. But that is not the usual situation, and therefore, loading before hand the entire document into memory is not appropriate.

Parse::Lex

The `Parse::Lex` module tries to abstract the lexical analyzer creation. Its lexer object is initialized with a flattened list of pairs (thus, a list with an even number of elements). Each pair include the token name and the regular expression that should be matched. An iterator is available, but unfortunately it does not detect when the end of input it reached. Therefore, a method to check for end of input needs to be used. Although the interface is very clean (see listing 1), this module creates very slow lexical analyzers. The next section will present details on its timings.

Listing 1 `Parse::Lex` lexical analyzer in Perl.

```
use Parse::Lex;

@tokens = ('PLUS'  => '\+',
           'MINUS' => '-'
           'NUM'   => '\d+', ...);
my $Lexer = Parse::Lex->new(@tokens);
$Lexer->configure(Skip => qr/[ \t\n]+/);

while (my $token = $Lexer->next) {
  last if $Lexer->eoi;

  ## do something with $token->name and $token->text
}
```

Unlike common lexical analyzers, where all rules are checked all the time and the rule that matches the longest data string is fired, in `Parse::Lex` the rules are analysed in order (much like as the vanilla technique pointed above would work). Therefore, if you have two rules that overlap (think of `a` and `a+b`) you need to specify the more generic first.

2.2 Lexers Efficiency Comparison

As stated before, generalized lexical analyzers written in Perl are prone to be based on regular expressions, which can produce inefficient analyzers. The use of *flex* based analyzers commonly produces faster and more efficient analyzers.

To sustain this claim several tests were made, one to benchmark CPU speed, and another to benchmark memory usage. Note that we are not stating that every flex based analyzer is faster than one written in pure Perl. But in general this is true.

Our tests compared the recognition of simple tokens: the four basic arithmetic characters, integer numbers and basic words (`/[a-zA-z]+/`). Spaces and newlines were recognized also, and ignored.

To perform the benchmarks three different lexical analyzers using different tools were implemented, just like detailed in previous section: `Parse::Lex` and flex.

The benchmark performed was to measure the lexical analyzers speed. Each analyzer was given the same input file to tokenize. The number of lines was increased and the time measured[3]. Table 1 shows the average execution time, in seconds, for each implementation for each given input file size.

Table 1 Average time, in seconds, to run lexical analyzers versus number of input lines (Intel Core 2 Duo, 2.4MHz).

Input lines	**Parse::Lex**	**Flex**
100	0.065454	0.015611
1 000	0.103696	0.017376
10 000	0.447719	0.024170
100 000	4.133743	0.118859

The performed benchmark help to sustain the claim that flex based lexical analyzers are prone to be faster than plain regular expressions based Perl analyzers.

This motivates the development of the approach described in this document, which plugs a flex based lexical analyzer to parser building frameworks.

3 Perl Syntactic Analyzers Overview

This section describes `Parse::Yapp` and `Parse::Eyapp`[4] because these tools will be used in the remainder of this work. Please refer to [1] for an overview on other syntactic analyzers in Perl. This section also includes a brief overview about Marpa because this tool was not included in the cited survey.

3.1 Perl Modules for Syntactic Analyzer construction

This section describes the tools that will be used for the construction of our parsers in Perl.

`Parse::Yapp` and `Parse::Eyapp`

`Parse::Yapp` (v1.05) is a pure Perl implementation of the well known `yacc/bison` algorithm [8]. This module reads a `yacc` specification with actions written in Perl, but is, otherwise, identical.

Unlike `yacc`, `Parse::Yapp` does not use a shared global structure (like `yylval`). Instead, the lexer function returns a pair that includes the identifier of the recognized token and the recognized string (or any other value you might want to pass to the syntactic analyzer). Note that inside the syntactic analyzer there is no need to define what types each production returns: semantic actions return Perl references to data structures, making it easy and clean.

The generated parser is reentrant, and it is possible to supply user data to the `yyparse` function.

`Parse::Eyapp` (v1.181) is an extended version of `Parse::Yapp`. It extends the `yapp` syntax with named attributes, extended BNF expressions, automatic abstract syntax tree

[3] Since time can be hard to measure accurately, each test was executed ten times. Then, the worst and best value were discarded, and the average time was calculated for the remaining values.

[4] In the remaining of the article we will refer to `Parse::Yapp` when a feature is both available on `Parse::Yapp` and `Parse::Eyapp`, and will explicitly refer to `Parse::Eyapp` when it is a specific feature of the extended module.

building, syntax directed data generation, tree regular expressions, tree transformations, directed acyclic graphs and it also includes a built-in lexical analyzer.

Marpa

Marpa is a parsing algorithm [9] for the recognition, parsing and evaluation of context-free grammars. The algorithm supports ambiguous grammars, and efficiently handles both left and right recursion. The algorithm is an evolution of Earleys' parsing algorithm [6], combined with improvements by *Joop Leo* described in [10] and by *Aycock et al* described in [3].

`Marpa::XS` (v1.006) is the latest stable implementation of this algorithm. This tool is able to create a parser for any grammar that can be described using the BNF notation [14]. But still the tokens recognizer must be written outside of the scope of the tool. Once the *tokenizer* is written it can be used to feed tokens, and corresponding values if required, to the parser, and the input can be parsed.

3.2 Efficiency Comparison

To compare `Yapp`, `Eyapp` and `Marpa` we implemented a simple calculator grammar as described on listing 2[5]. As semantic actions, instead of calculating the real value an abstract parsing tree was constructed. To test the three implementations we used a random generator.

Listing 2 Grammar used for syntactic analyzers benchmark.

```
gram: lines

lines: lines exp
     | exp

exp: exp ’+’ exp
   | exp ’-’ exp
   | exp ’/’ exp
   | exp ’*’ exp
   | ’(’ exp ’)’
   | NUM
   | VAR
   | VAR ’=’ exp
```

As `Parse::Eyapp` is based on `Parse::Yapp` and we did not use any of the new features, we decided to use `Parse::Eyapp` built-in lexical analyzer and, for `Parse::Yapp` and `Marpa::XS`, use `Parse::Lex`, but both lexical analyzers are implemented in Perl. Table 2 presents running times for different input sizes, and table 3 shows memory usage. Regarding memory usage it is relevant to say that as we are building a parse tree it is natural the amount of used memory grows. Also, as at the end we are generating a dump for the tree structure, still more memory gets used.

Note that this comparison is not completely fair regarding `Marpa::XS`. Marpa computes all possible parse trees, making it able to parse ambiguous grammars, but also making it slower and exhausting more memory.

As a final note, our Marpa parser uses a ranking approach that is the Marpa feature that better mimic the precedence (and associativity) information given to `Parse::Yapp` and

[5] The precedence details are omitted in the grammar.

Table 2 Average time, in seconds, to parse input file versus number of input lines (Intel Core 2 Duo, 2.4MHz).

Input lines	**Parse::Yapp**	**Parse::Eyapp**	**Marpa**
100	0.108692	0.116950	0.375665
1 000	0.459240	0.469139	2.801585
10 000	4.436363	5.010674	70.969120

Table 3 Memory footprint for parsers versus number of input lines.

Input lines	**Parse::Yapp**	**Parse::Eyapp**	**Marpa**
100	3.86 MB	4.82 MB	8 202.98 MB
1 000	14.13 MB	15.07 MB	8 258.57 MB
10 000	117.04 MB	118.10 MB	9 008.07 MB

`Parse::Eyapp`. After some discussion with its author we found that a non ambiguous version of the grammar after some code refactoring gets some more interesting times and memory consumption. The time to process 10 000 lines drops to 9.218295 seconds, and the memory used to 279 MB.

4 flex and `Parse::Yapp`: How It Works

This section resumes the approach presented in [12] for gluing a flex syntactic analyzer with Perl.

As with any other language, the Perl community does not intend to implement everything in Perl. There are good libraries available (mainly written in C and C++) that should be used, and only an abstract interface written (also known as bindings). This means that the approach here described can be simulated using any other scripting language. It is just a matter of rewriting the glue approach.

The binding between a C library (or simply an object file) and Perl is written in a syntax known as XS [7]. It is a Domain Specific Language (DSL) [13] written on top of the C syntax, with some syntactical sugar that is recognized by the `ExtUtils::ParseXS` module, that generates a complete C (or C++) file that can be linked against Perl.

To glue flex with Perl we need to generate the standard flex input file. We decided to implement the flex file just like a standard implementation as if it was working with bison or yacc. For characters, the `yylex` function returns the character ASCII code (unfortunately flex is not supporting Unicode at the moment). For other tokens, the function returns an integer bigger than 256. This same flex files implements a function to access the `yytext` variable content. Listing 3 shows part of the flex generated lexical scanner.

Listing 3 Generated flex scanner.

```
%%
[0-9]+          { return 256; }
[A-Za-z]+       { return 257; }
[ \t\n\r]+      { /* ignore */ }
.               { return flapp_yytext[0]; }
%%
int flapp_yywrap(void) { return 1; }
char* flapp_yylextext(void) { return flapp_yytext; }
```

In the generated Perl lexer module, the method yylex is called to check for the next token type. If it is a character, it is returned both as token name and token content. If not, an array is accessed to check for the name of the token (note that this array is generated automatically), and the token name is returned together with the result of invoking the method to access the yytext variable content. Listing 4 presents the lexer interface for our calculator grammar[6].

Listing 4 Perl interface to a flex scanner.

```
sub flapp_lex {
   my $token = flapp_yylex();
   my @tokens = ('Num', 'Var');
   if ($token) {
      if ($token >= 256) {
         return ($tokens[$token - 256], flapp_yylextext())
      } else {
         return (chr($token), chr($token))
      }
   } else {
      return (undef, "")
   }
}
```

Other than these two files, all other generated files are related with the Perl module construction and compilation. These details are specific to Perl and we do not think they are relevant to be discussed in this article. Nevertheless, the next section will briefly introduce those files.

5 Parse::Flapp: The Code Generator

Parse::Flapp is a module that helps creating syntactic analyzers in Perl, glued with flex. At the time of writing the module only supports Parse::Yapp, but support for Parse::Eyapp and Marpa::XS is planned.

The module includes a command line tool, named flapp, that given a Parse::Yapp grammar with some extra minimal syntactic sugar, and a module name, creates a complete Perl module that implements the syntactic parser, for example:

```
$ flapp -module=My::Parser example.fyp
```

The syntactic sugar added to the Parse::Yapp (and Eyapp) grammar files is inspired in the way Eyapp defines terminal symbols. When listing the tokens that will be used in the grammar, the regular expressions that matches each token can also be defined. For example:

```
%token NUM = /[0-9]+/
%token VAR = /[A-Za-z]+/
```

In fact, Eyapp requires that expressions are between parenthesis, to instruct Perl to capture the matching string. For Flapp this is not required. But as the regular expressions are

[6] Instead of yylex the method is calling flapp_yylex as the Perl language parser is written using flex, and uses the default prefix. So, if we did not use a different name, the linkage process would not work.

used directly in the flex file, they need to be flex compatible, and not standard Perl regular expressions. This fact limits the token definition, namely if the user needs to interpolate string variables, or needs to use any construct not recognized by flex. In a future version `Flapp` might include a regular expression parser to validate if the expression is compatible with flex, or even rewrite portions of it. `Flapp` rewrites the grammar file removing the regular expressions. This makes the grammar file compatible with both `Parse::Yapp` and `Parse:Eyapp`.

The `flapp` script generates a Perl module that includes the following files (assuming the module name is `My::Parser`):

- `fl_Parser.l` — the flex lexical scanner, as described in the previous section (see listing 3);
- `lib/My/Grammar.yp` — the grammar received as input, with the `Parse::Flapp` specific syntax removed (this is a standard `Parse::Yapp` grammar);
- `lib/My/Parser.pm` — the Perl module which includes the lexical analyzer method, that interfaces with the flex lexical scanner (see listing 4);
- `Makefile.PL` — a standard Perl module makefile, that compiles and links the C code with Perl;
- `MANIFEST` — a manifest file, that lists all the files included in the module;
- `parse.pl` — a small Perl script to use the parser *right out of the box* (see listing 5);
- `Parser.xs` — the interface file that explains Perl how to deal with the C functions (see listing 6);

Listing 5 Small Perl script using the generated module (`parse.pl`).

```
use My::Grammar;
use My::Parser;

use Data::Dumper;
my $output = flapp();
print Dumper($output);

sub flapp {
  my $parser = My::Grammar->new;
  $parser->YYParse(yylex   => \&My::Parser::flapp_lex,
                   yyerror => \&My::Parser::flapp_error);
}
```

To make this module general [5] (at least in a near future), the module implementation was based on templates [4], one for each generated file. The use of templates makes the tool easier to extend and maintain.

6 Tests and Evaluation

To compare with the other approaches we ran the resulting `Flapp` parser for the same kind of data used before. The results were similar to `Parse::Yapp` both in time and memory (check table 4). Nevertheless, we can lower the memory consumption in approximately 25 MB, and the execution time in one second. Although these differences are not extremely high, they will be relevant for bigger input data.

Listing 6 C glue code, written in the XS syntax (`Parser.xs`).

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

#include "lex.flapp_yy.c"

MODULE = My::Parser             PACKAGE = My::Parser

int
flapp_yylex()
    OUTPUT:
         RETVAL

char*
flapp_yylextext()
    OUTPUT:
        RETVAL
```

Table 4 Running times and memory footprint for `Parse::Yapp` and flex.

Input lines	Running time	Memory footprint
100	0.098 628 sec.	1 922.30 KB
1 000	0.375 966 sec.	9 127.16 KB
10 000	3.300 446 sec.	82 595.58 KB
100 000	34.743 845 sec.	857 262.58 KB

7 Conclusions

There is no doubt that flex is more efficient than any other lexical scanner implemented in Perl. Although there is not a pure Perl implementation of the flex algorithm, we do not think it would beat its C counterpart.

Nevertheless, Perl is great for dynamic data structures. We can argue that C data structure are faster, but they are harder to implement (a flex scanner is not that hard to use). Therefore, the composition of Perl with flex is relevant. But the process of using C from Perl is not that easy, and that can be an obstacle in the use of flex from within Perl.

The `Parse::Flapp` module solves this problem by making that task completely automatic, generating a Perl module ready to use and edit for further features.

The current `Parse::Flapp` implementation, still a prototype, generates new modules to parse with `Parse::Yapp` and `Parse::Eyapp` syntactic analyzers, but is easy to extend for other parser generators. In the future `Parse::Flapp` should also be able to update a generated module (and not only bootstrap). That would be indispensable for standard software development methodologies.

References

1 Hugo Areias, Alberto Simões, Pedro Henriques, and Daniela da Cruz. Parser generation in Perl: an overview and available tools. In Luis S. Barbosa and Miguel P. Correia, editors, *INForum'10 — Simpósio de Informática (CoRTA2010 track)*, pages 209–212, Braga, Portugal, Setembro 2010. Universidade do Minho.

2 Hugo Areias, Alberto Simões, Pedro Henriques, and Daniela da Cruz. Parser generation in Perl: Crafting an AnTLR back-end. In Raul Barbosa and Luis Caires, editors, *INForum'11 — Simpósio de Informática (CoRTA2011 track)*, pages 258–269, Coimbra, Portugal, Setembro 2011. Dep. de Eng. Informática da Universidade de Coimbra.

3 John Aycock and R. Nigel Horspool. Practical earley parsing. *The Computer Journal*, 45(6):620–630, 2002.

4 Darren Chamberlain, David Cross, and Andy Wardley. *Perl Template Toolkit*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2003.

5 Mark Jason Dominus. *Higher-Order Perl: Transforming Programs with Programs*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

6 Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

7 Tim Jenness and Simon Cozens. *Extending and Embedding Perl*. Manning Publications, August 2002.

8 Stephen C. Johnson. Yacc: Yet Another Compiler-Compiler. Technical report, AT&T Bell Laboratories, Murray Hill, New Jersey 07974.

9 Jeffrey Kegler. Marpa. Web site: http://www.jeffreykegler.com/marpa [Last accessed: 19-03-2012].

10 Joop M. I. M. Leo. A general context-free parsing algorithm running in linear time on every lr (k) grammar without using lookahead. *Theoretical computer science*, 82(1):165–176, 1991.

11 Vern Paxson, Will Estes, and John Millaway. *The flex Manual*. The Flex Project, version 2.5.35 edition, September, 10 2007. Available at `http://flex.sourceforge.net/manual/index.html`.

12 Alberto Simões. Cooking Perl with flex. *The Perl Review*, 0(3), May 2002.

13 Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages. *Centrum voor Wiskunde en Informatika*, 2000.

14 Wikipedia. Backus–naur form. Web site: http://en.wikipedia.org/wiki/Backus-Naur_Form [Last accessed: 19-03-2012].