Parser Generation in Perl: Crafting an AnTLR Back-end

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,alberto.simoes@eu.ipp.pt
pedrorangelhenriques@gmail.com,danieladacruz@di.uminho.pt

Abstract. Completely convinced of the benefits of Perl for the implementation of language processors and after doing a deep review of the state of the art on parser generation in Perl, we have identified a clear need for a powerful tool that accepts attribute grammars and builds compilers in Perl.
The objective of this paper is to present a solution based on the developing of a back-end for ANTLR to generate attribute based language processors in Perl, to overcome the lack of this kind tools described in the preliminary study referred above.
To achieve the intended objective, the parser generator tool ANTLR was studied in detail to understand its components and to plan the best strategies to perform the necessary retargeting of its back-end. In this process, the Java compilers generated by ANTLR were analysed carefully to learn the algorithms adopted and the language resources employed.
After that, the general scheme for Perl compiler was sketched. This task has required a comparative study of object-oriented extensions to the standard Perl in order to decide how to provide the basic features available in Java.
We also discuss in the paper the performance tests carried out to prove that the generated processors, produced by our new tool, are efficient and provide indeed a reliable and competitive solution.

Keywords: Parser generators, Perl, grammars, ANTLR, language processors

1 Introduction

This project[1] has emerged in the context of language processing and parser generation, motivated by the need to create an efficient and viable alternative to the existing *Perl* [8] parser generators as addressed in further detail in a preliminary study [3] done last year.

Although there are some tools to generate parsers in *Perl*, most of them reveal some drawbacks — such as the non-support for attribute grammars [1,5,7] and

[1] https://github.com/HugoAreias/AnTLR-Back-end-for-Perl

lack of efficiency — which causes *Perl* to be discarded from projects that require high levels of efficiency [3].

It was intended with this project to workaround the known flaws when generating *Perl* parsers. The starting point was to port the parser algorithms generated by other tools to *Perl* language.

To accomplish this task, the compiler generator AnTLR[2][6] was chosen. AnTLR is a tool developed in *Java* that provides support for attribute grammars, well known for its professional contribution to $LL(k)$ (recursive-descent) [1,2] parser generation[3]. This tool was chosen instead of other tools, such as LISA, due to its wide use on parser generation and amount of documentation available; moreover AnTLR was developed taking into account the possibility of creating code generators for specific target languages and easily fold them with AnTLR. AnTLR already has a few code generators associated with it, for languages such as *C*, *Java*, *Ruby*, *JavaScript*, *C#*, *Python*, amongst others. The internal structure of AnTLR and the fact that the tool provide a template engine to assist on the retargeting, stimulates the creation of new output modules to produce parsers in other languages, such as *Perl*.

With the assistance of this tool, it was possible to develop a *Perl* code generator folded into AnTLR, with the purpose of translating the generated parsers to *Perl* language taking advantage of AnTLR algorithms and grammars support. Besides, the use of grammars would turn *Perl* parsers more readable, easy to write and maintain. To obtain the desired results, an exhaustive study over ANTLR was made to correctly understand its functioning and to obtain the maximum knowledge about the tool.

The main goal was to develop a new AnTLR back-end to output *Perl* parsers, with support for attribute grammars, that evidences slightly better efficiency levels than the existing alternatives for the *Perl* language. It is intended to generate *Perl* parsers that can be used as a valid alternative to parsers generated in other languages.

Lavanda[4] grammar [4] was chosen to serve as example throughout this paper.

This paper is organised in six sections. First, section 2 gives a quick explanation of the importance of StringTemplate in the development of the tool. section 3 explains the strategies adopted to implement the Perl Code Generator. The Runtime Library is explained in section 4. In section 5 analysis are made about the tool features, and performance tests are carried out to prove the generated language processors are reliable and efficient. Lastly, conclusions and comments about the tool and this work will be drawn in section 6.

2 The StringTemplate

The StringTemplate engine was introduced by ANTLR to convert the code generation procedures to a template based model. It provides its own template cre-

[2] ANother Tool for Language Recognition
[3] Language processor generation, to be more accurate.
[4] `http://epl.di.uminho.pt/~{}gepl/LP/`

ation language, used by the code generator templates. Basically, the templates in charge of generating the parsers for each target programming language, are implemented with StringTemplate language, and are interpreted by StringTemplate engine. These templates are included in the Code Generator component, along with a target class for each language available.

The integration of the template engine StringTemplate with ANTLR proved to be worthwhile and a catalyst for code generation, because it strengthened the ANTLR position among parser generators, increased the employment of this tool on serious projects and acted as an encouragement for programmers to retarget code generators for other languages, such as *Java*, *Python*, *C*, *C#* and others.

StringTemplate is a library that operates separately from ANTLR. This separation makes possible the use of template construction actions in the rules of a grammar implemented in ANTLR and to specify output templates directly in the grammar rules, taking advantage of a specific notation provided by ANTLR for this purpose.

These features contribute to a more clean and readable code, and encourages the separation of the output specifications from the remaining code and the reuse of replicated segments of code to emit output. However, the most significant advantage of StringTemplate integration with ANTLR is the easiness of generating a parser in different languages without the need of rewritten it for each target language. Furthermore, it saves time and avoids unexpected bugs.

These features made it easier to develop the Perl Code Generator since it was not required to know how exactly ANTLR compiles the grammars and how it handles the extracted data. The StringTemplate integration was of a considerable help and made us concentrate mostly on the generation of the language processors in *Perl*.

The StringTemplate language was crucial in this process since it functioned as a kind of bridge between the generated language processor and the data extracted from the compiled grammar required to generate its code.

3 The Perl Code Generator

The Perl Code Generator is constituted by two major components, the runtime library and the template group. Although these are the essential components to generate a *Perl* Language Processor with ANTLR, there are complement components, responsible for performing the tests and installation of the Runtime library, and providing examples of the implementation of grammars in ANTLR using *Perl* syntax. The functional specification of the Perl Code Generator can be seen in figure 1. This functional specification was initially designed by Ronald Blaschke[5].

The Perl Code Generator was implemented with two priority aspects in mind, readability and efficiency. The *Perl* code, generated by ANTLR, should be as easy as possible to understand, with simple processes and similar in structure

[5] `http://www.rblasch.org/`

with the *Java* solution, but at the same time the generated Language Processor (LP) should be able to parse input text efficiently, especially when dealing with large input streams. To achieve these goals it was required a careful analysis of the code generated by ANTLR and how it was generated taking into account the specification of the grammar.

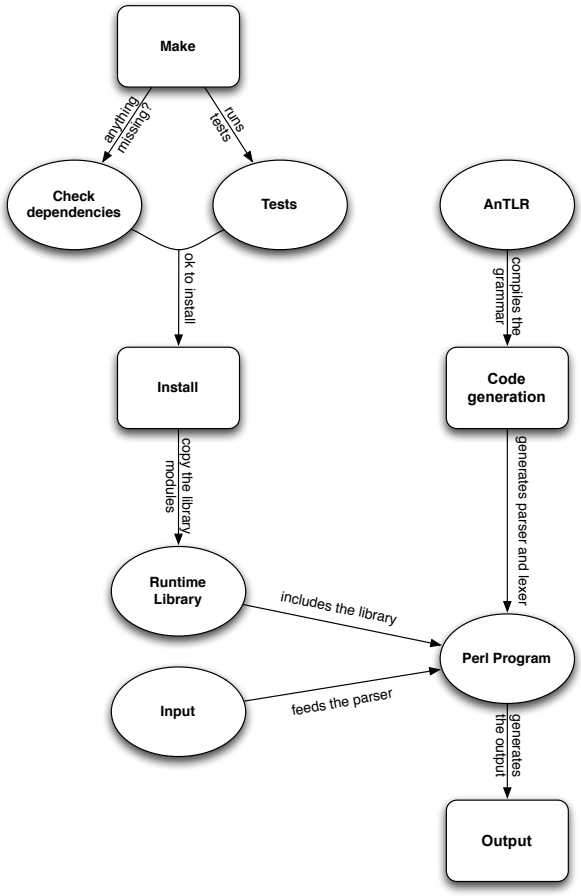

Fig. 1. Functional specification of the Perl Code Generator.

The first step to implement the Code Generator was to analyse the generated code for *Java*. *Java* was chosen mainly because it is the default generated language and therefore it keeps the original hierarchy and structure of ANTLR

template files intact. With these analysis it was possible to obtain a general idea of how the parser and lexer works and the algorithms implemented by them.

After having a precise idea of the expected *Perl* code, the next step was to create the template group for *Perl* language. The template group was not created from scratch since it would be easier to replicate the *Java* template group and replace the code fragments with the corresponding in *Perl*. However, to make the changes it was essential to understand how ANTLR generates the code and how the Code Generator works internally, including the data structures where the information about the grammar and the defined language are stored.

With all this in mind, the development of the Code Generator was a slow and careful process assisted closely by a battery of tests to assure that the code was being generated correctly and that it actually works as expected when executed.

The strategy used to achieve a runnable LP was to work on one block of the grammar at a time, studying the templates responsible for generating the corresponding code and then change them to generate *Perl*. Lastly, ANTLR was executed to generate the code in *Perl*, to allow the verification of the generated code, checking whether it was syntactically correct or not.

The tests were based on small grammars in the first stages of the project. After obtaining a correct *Perl* processor, for a specific grammar, more complex grammars were used to test other templates, until producing valid pieces of *Perl* code at the end. During the tests, all the templates not generating *Perl* were yet to be translated and tested. This way it was easier to keep track of which ones were already implemented and which not.

4 ANTLR Perl Runtime Library

The organisation of the files of the Perl Runtime library is very similar to the *Java* library. Each class in the *Java* library corresponds to a *Perl module*. Since *Perl* and *Java* have a large number of differences, a few classes have not been retargeted for *Perl* and one new has been created.

One of the most important decisions before starting to implement the *Perl* library was which would be the approach to implement it with object-oriented *Perl*, knowing that our model (*Java*) is an *OO* language and that *OO* is a relevant topic in modern *Perl* nowadays. Since *Perl* was not originally designed to support the *OO* paradigm, it was not easy to find a way of include it in the language.

Currently, *Perl* deals with objects in the same way it deals with hashes; moreover, an object in *Perl* is nothing more than a hash that holds all the information in it. This solution was not desirable in this project since it would negatively affect the readability of the code (Runtime library and generated language processor). It was intended that the solution would have to make a comprehensive differentiation between hashes and objects usage.

The solution was to integrate *Moose*[6] with the runtime classes and include also its constructs in the generation of the code. By using *Moose* to implement

[6] `http://search.cpan.org/~doy/Moose-1.08/lib/Moose.pm`

the Runtime library and the generated Language Processors, the code becomes easier to read and understand, and we can obtain *modules* similar in structure with *Java* classes. Besides that, *Moose* introduces some concepts that are also present in *Java*, such as *interfaces* (*roles* in *Moose*), that proved to be useful during the Perl Runtime library implementation.

Although *Moose* efficiently solved the *Perl OO* limitations, there were still some language drawbacks that needed to be addressed. A small example is the keyword *final* in *Java*, not found in *Perl*, which determines when the value of a variable or object is immutable. In the *Perl* solution it was included a *module*, *Data::Lock*[7], to deal with this kind of variables or objects.

One of the most arduous tasks to accomplish during the developing process of the Runtime library was how to handle exceptions. Again, *Perl* has no native solution for this problem, so a *module* was required to solve it, in this case, *Exception::Class*[8]. However, the *module* alone was not enough, since ANTLR as its own set of exception classes to handle mismatch or any other kind of exception, that inherit the properties of *Java Exception* class.

For the equivalent exception classes in *Perl* to have the ability of throwing exceptions, throughout the code, it was required a new class, named *Exception*, which inherits *Exception::Class* properties, including the "throw" method. This class would be later extended by all the exception classes present in the Perl Runtime library.

Exception::Class also provides a method to catch and identify an exception, however it does not implement a "try" block as found in *Java*. Therefore, it was required another *Perl module* to obtain the same behaviour. The module chosen was *Try::Tiny*[9].

There were also found significant differences at the string encoding level between both languages, especially when dealing with *unicode* characters. This differences have been overcome by creating a method in the *Perl* target class found in the Code Generator, to address and convert the *unicode* representation from *Java* to *Perl*.

Lastly, one of the major differences between both languages is the type system used. *Java* is a strongly typed language, on the other hand, *Perl* is a weakly typed language, this means a variable in *Perl* can assume any value and can be used in evaluations of different types. For example, an instance object of *CommonToken* can be assigned to a variable holding an instance object of *BitSet* at the time of the assignment without raising an error.

In this project it was required a more tight control over *Perl* variables, to assure that variables would not assume other types under specific situations. *Moose* was extremely helpful on this matter, since it raises constraints when the user is manipulating variables instantiated in a class implemented with it. This

[7] `http://search.cpan.org/~dankogai/Attribute-Constant-0.02/lib/Data/Lock.pm`

[8] `http://search.cpan.org/~drolsky/Exception-Class-1.32/lib/Exception/Class.pm`

[9] `http://search.cpan.org/~nuffin/Try-Tiny-0.06/lib/Try/Tiny.pm`

means that values assigned to a variable must correlate with its type, just like happens in *Java*.

5 Evaluation of the Generated Parsers

With the developing process of the tool finished it is important to assure that the tool actually works and behaves as expected. Also, it should be able to execute correctly and efficiently. It is crucial for a tool of this nature to be submitted to a number of optimisations to be as much efficient as possible. To prove the tool is reliable and efficient, performance tests and comparisons with other available solutions have been done in this section.

5.1 Optimisation and Profiling

When dealing with large input data, the generated parsers must evidence, at least, a reasonable performance. So, it is important to have some concerns about the efficiency of the code execution when writing it. However, since the code is very extensive and integrates multiple components, the refining process can be arduous to accomplish since it is harder to identify the critical parts.

With this in mind, a profiler was used to assist in the optimisation process of the tool. The profiler eased the evaluation of the performance of the tool components and helped to identify exactly which code instructions should be improved to achieve better results.

The profiler chosen was the *Perl module Devel::NYTProf*[10], one of the most used for *Perl* projects. This *module* is very useful to know which are the most inefficient subroutines, how many times and where they are called, etc. It is also possible to verify the execution times of every subroutine instructions. This property is extremely useful to know where to start the optimisations of the tool.

Executing the profiler in parallel with the generated LP for the *Lavanda* language, it was possible to identify the weaknesses of the Code Generator and Runtime library.

The code refining process was done by consulting the profiler results for the parsing process and replacing whenever possible the instructions with the worst execution time by equivalent instructions.

One of these cases was the "For-Each" loop, that has been replaced by a "While" loop in most of its occurrences. This happens because the "For-Each" statement must know how many elements are in the list before iterating over it. This means the complete list must be pushed to memory at the same time to calculate its size. On the other hand, the "While" loop does not requires the size of the list since it iterates until reaches the end of the list. Since parsers deal with huge amounts of data, the "For-Each" loop would decrease its efficiency.

The continuous call of subroutines in loop conditions is another case that can decrease the efficiency of the code. These situations were reduced to a minimum,

[10] `http://search.cpan.org/~timb/Devel-NYTProf-4.06/lib/Devel/NYTProf.pm`

even when dealing with accessors. In these cases, the variables were set to public and accessed directly to avoid calling the accessor methods several times.

To measure the impact of these changes on the parsing times, the execution times of the generated parser for the *Lavanda* language, when parsing an input file with *1000* lines, will be compared below.

Before these changes, the parser spent more than one minute to correctly recognise the input. After applying the changes described before, the parser spent around *46* seconds to parse the same input. The changes clearly had a small impact on the parsing time.

On a second round (*Opt 2*) of optimisations, replicated computations were reduced to a single one. For instance, the method "length" would be called each time the input size was needed. When dealing with large input streams, calculate its length several times can be a costly process. Since the input cannot be modified during runtime, its value will be preserved until reaching the end of execution, and therefore its length will remain the same at any point of the parsing process. This means that the input size calculation can be reduced to a single call of the "length" method. This change implies the inclusion of a new instance variable to store the input size when the object is created.

Applying this optimisation decreases the parsing time from *46* seconds to approximately *34* seconds. A better result but still with room for improvement.

As explained in the preliminary study, one of the main drawbacks of the parser generators available for *Perl* was the fact that the parsers generated by these tools push the input into memory treating it as a single string, decreasing considerably their efficiency when operating over large input texts.

To workaround this issue, the class that handles the input text, included in the Runtime library, turns the string into an array of chars. This way it is easy to perform operations over the input by accessing it via indexes, such as obtaining substrings.

This last change (*Opt 3*) decreased the parsing time, for the same input, to approximately *20* seconds. Before starting the code refining process, the generated parser for *Lavanda* language was taking more than *60* seconds to parse a file with *1 000* lines, and after it, the execution time was approximately *20* seconds, near *66%* less than the first attempt. The accurate times can be seen in Figure 2 along with the parsing time evolution.

5.2 Evaluation

After refining the code generator to obtain a better performance, specific tests were planned to evaluate the efficiency of the generated LPs by comparing the obtained results with the results of the tools evaluated in the preliminary study mentioned before in this paper.

The same metrics will be used in the following analysis. These comparison parameters are:

1. the readability of grammars and generated parsers.
2. integration of the lexical analyser.

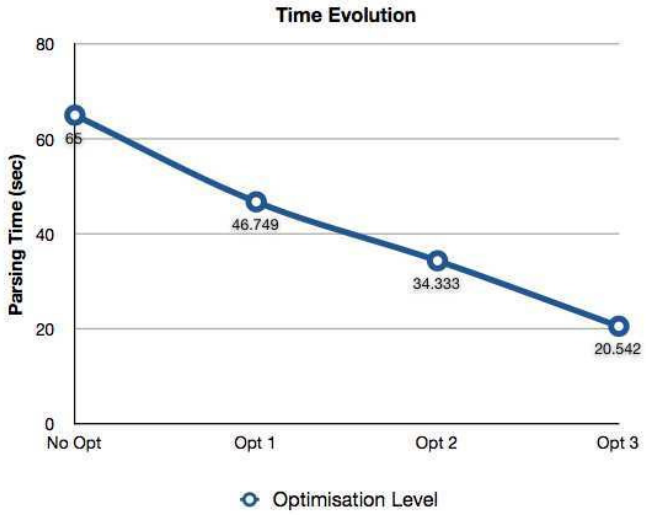

Fig. 2. Time evolution after performing each optimisation level.

3. support for semantic actions.
4. flexibility of integration of the generated LP with other code.
5. debugging mechanisms.
6. efficiency of the generated parser (parsing time and memory consumption).

Good readability is a key aspect of a grammar and generated parser, especially for language comprehension and maintenance purposes. The developed tool offers the same grammar readability as ANTLR because the grammar is written in ANTLR metalanguage. Also, ANTLR metalanguage offers support for EBNF notation and named access, which helps improving the readability of the grammars.

Regarding the generated parser readability, the code is not easy to understand since the user should have a basic knowledge of the Perl Runtime library. However, the identification of the grammars rules throughout the parser is straightforward, since each subroutine of the generated parser corresponds to a production of the grammar.

As far as the lexical analyser is concerned, the developed tool benefits, once again, from the organisation and algorithms implemented by ANTLR. When ANTLR generates a parser in *Perl* based on a given grammar it also generates a lexical analyser structured in the same way as the parser.

The main difference between the lexer generated by ANTLR and the ones used by the tools tested before is that the first treats the input as an array of chars while the second treat it as a string. The fact that to perform operations over the input may require it to be processed more than once, instead of accessing

the data via indexes, may have severe consequences on the efficiency of the tool when dealing with large blocks of data.

As mentioned before, none of the available parser generators for *Perl* offers any kind of support to generate parsers based on attribute grammars and only `Parse::Eyapp` gives the possibility of creating an AST. On the other hand, the Perl ANTLR back-end supports attribute grammars, and creates an AST and a DFA during the parsing process. Also, all the semantic information gathered during the parsing process can be obtained by the user at the end of it.

The generated parser is easily integrated in any *Perl* project, however, the target machine (the one where it is intended to execute the generated code) must have the Perl Runtime library installed. This is one of the main drawbacks of the Perl Code Generator when compared to other solutions, such as `Parse::Yapp` (can generate standalone parsers).

Most of the parser debugging mechanisms are not implemented yet, since the debug library has not been included in the Perl Runtime library. However, the Runtime library offers mechanisms to catch and throw the parsing errors or trace the parsing process.

For the efficiency tests, only three *Perl* parser generators were selected, the best *LALR* [1,2] (*Parse::Yapp*) and *LL* [1,2] (*Parse::RecDescent*) based parser generators of the earlier tests [3] and the *Perl Code Generator*.

LR [1,2] based parsers tend to achieve better parsing times than *LL* so it is not a surprise that a parser generated by *Parse::Yapp* (*LALR*) achieves better results that the other generated parsers. However, this only happens for input streams somewhere below *100 000* lines, since the parser generated by the *Perl Code Generator* spent around *1 409* seconds for an input with *100 000* lines, against the *1 796* spent by the one generated by *Parse::Yapp*, as seen in Table 1.

Regarding the *LL* based parser generators, *Parse::RecDescent* parsers spend less time parsing small inputs than the *Perl Code Generator*, but its efficiency highly decreases when dealing with large input streams.

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

Input Lines	Parse::Yapp	Parse::RecDescent	Perl Code Generator
10	0.017 s	0.073 s	0.726 s
100	0.073 s	0.183 s	2.525 s
1000	0.763 s	2.796 s	20.542 s
10000	18.915 s	290.914 s	204.679 s
100000	1796.26 s	> 14206.187 s	1409.238 s

The *Perl Code Generator* parsers spend more time to finish the parsing process when dealing with small input streams. However, it must be considered that it generates a Language Processor instead of only a parser, like the other gener-

ators tested, so it is acceptable that it executes a larger amount of instructions than the others.

Despite the LPs generated by our tool present parsing times greater than the other solutions, when dealing with small input streams, they have proven to be the best solution to deal with large input streams, even against generated $LALR$ based parsers, known for being faster than recursive-descent parsers. This was one of the identified drawbacks of the parsers in *Perl* in the beginning of this work.

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

Input Lines	Parse::Yapp	Parse::RecDescent	Perl Code Generator
10	0.933	3.733	10.440
100	1.934	4.764	11.646
1000	12.142	15.335	23.743
10000	108.701	115.539	145.299

The main drawback of the parsers generated by the *Perl Code Generator* when compared to the other solutions is the slightly larger use of memory, as seen in Table 2. Since the generated LP generates or obtains plenty of information about the grammar during the parsing process, it was obvious that it would use a larger slice of memory than the other generated parsers. However, the difference tends to stabilise as the input size increases.

6 Conclusion

This study made possible the implementation of a code generator for *Perl* that was fully integrated with ANTLR. Some of its implementation strategies and decisions were explained throughout this paper.

The Perl Code Generator main goal was to generate parsers in *Perl*, based on attribute grammars, that prove to be more efficient than the solutions already available for *Perl*, especially when dealing with large input streams. The tests performed, as reported in subsection 5.2, show that the developed tool is slower to parse small input streams (*1 000* lines or less) but it reveals to be the best for inputs of *100 000* or more lines, since the other tools lose efficiency as the input size increases.

Based on the tests performed during this project, the Perl Code Generator is the best solution, to our knowledge, from the LL-based parser generators, and its generated parsers are faster than the $LALR$-based tested when dealing with large blocks of data, which is something important to notice. Regarding the use

of memory, the Perl Code Generator was the tool that shown higher consumption, mainly due to the fact that it generates a language processor instead of only a parser, as happens with the other tools.

Although the AST it is not yet available for display and the debug facilities are not totally usable, most of the features available on the other *Perl* parser generators are also available on the developed tool, mostly because ANTLR already has native support for them.

The optimisations described in subsection 5.1 need to be extended to more levels to improve the efficiency of the tool, to achieve better parsing times and reduce the memory consumption. With these desired optimisations it would be possible to make the Perl Code Generator a valid alternative to other language solutions.

For a first version, the developed tool is already in a valid state to be considered a good solution for parser generation in *Perl*, mainly due to its performance behaviour and the advantages of using ANTLR metalanguage to write the grammars, especially attribute grammars.

On the overall, the developed tool is in an operational state and the generated parsers behave as expected. *Perl* has now a valid LP generator with good results, despite needing more improvements to become a strong alternative to other solutions available.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley, 1986.
2. Alfred V. Aho and Jeffrey D. Ullman. *The theory of parsing, translation, and compiling*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1972.
3. Hugo Areias, Alberto Simoes, Pedro Henriques, and Daniela da Cruz. Parser generation in perl: an overview and available tools. *Luis S. Barbosa and Miguel P. Correia, editors, INForum'2010 Simpósio de Informática*, pages 209–212, September 2010.
4. Daniela da Cruz and Pedro Rangel Henriques. Lavanda, an exercise with attribute grammars and a case-study to compare ag-based compiler-generators. Cctc technical report, Dep.Informática / Univ. do Minho, Dec. 2006.
5. Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.
6. Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007.
7. William M. Waite. Use of attribute grammars in compiler construction. In *WAGA*, pages 255–265, 1990.
8. Larry Wall. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.