Resumo

No âmbito do segundo ano de mestrado em engenharia informática, este documento apresenta uma dissertação na área de processamento de linguagens, mais propriamente na geração de compiladores baseados em gramáticas de atributos.

A linguagem Perl é bastante conhecida, principalmente, pela sua flexibilidade e por disponibilizar poderosos mechanismos para processamento de texto, juntamente com expressões regulares bastante expressivas e suas funcionalidades, para facilitar a extracção e manipulação de informação. Embora as expressões regulares sejam bastante úteis no processamento de linguagens, e Perl 5.10 já suportar parsers recursivo-descendentes, estas nem sempre são suficientes para reconhecer e processar textos com uma estrutura complexa. No entanto, o processo de desenvolver parsers é difícil e demorado. Para facilitar este processo começaram a ser desenvolvidos geradores de parsers para várias linguagens alvo, independente da linguagem de implementação.

Existem alguns módulos disponíveis no CPAN para assistir na geração de parsers em Perl, mas alguns são limitados, apenas suportando um único algoritmo de parsing e não abrangendo algumas necessidades do utilizador. A eficiência e a usabilidade destes módulos irão ser analisadas e testadas, permitindo assim a avaliação dos resultados e cuidadosas considerações sobre o estado da arte da geração de parsers em Perl.

Este trabalho vai-se focar no estudo da relevância de geradores de parsers como o AnTLR[1], e na necessidade de produzir processadores em diferentes linguagens. Vai-se proceder também a uma análise sobre alguns dos back-ends já existentes para o AnTLR, principalmente o de Java. No final, irá ser desenvolvido um back-end escrito em Perl para AnTLR, baseado no estudo efectuado, de modo a consiguir-se retirar algumas conclusões. O principal objectivo da ferramenta é gerar parsers em Perl automaticamente através da assisência do AnTLR.

Para se obter esta ferramenta, irá ser feito um estudo sobre o AnTLR e seus componentes para ser possível delinear as melhores estratégias para implementá-la. Sendo o StringTemplate o componente responsável pela geração do código nas linguages alvo, este será devidamente analisado juntamente com alguns processadores de linguagens em Java *gerados pelo AnTLR para se ter uma ideia concreta de como obter o mesmo resultado em* Perl.

Para provar que a ferramenta é uma solução válida, e que os processadores de lin-

[1] ANother Tool for Language Recognition

guagens gerados são eficientes, serão efectuados testes

Abstract

In the scope of the second year of my master course in computer engineering, this document is a dissertation in the area of language processing, more specifically, in compilers generation based on attribute grammars.

The Perl language is well known mainly for its flexibility and for providing powerful text processing mechanisms, along with powerful regular expressions and related functionalities to facilitate data extraction and manipulation. Although regular expressions are quite helpful for language processing, and Perl 5.10 already supports recursive-descent parsers, these mechanisms are not always enough to recognise and process texts with a complex structure. However, the process of constructing such LPs is painful and time consuming. To ease this process some parser generator tools were developed; independently of the implementation language, different target languages can be considered.

There are a few modules in CPAN to help with the generation of parsers in Perl; however some are limited, only supporting a particular parser algorithm and not covering desirable needs of the user. These modules are going to be analysed and tested for performance and usability. A proper evaluation of the results will allow careful considerations about the state of art of parser generation in Perl.

This work will focus on the study of the relevance of parser generators like AnTLR[1]*, and the need for producing processors in different languages. It will also be presented an analysis of some of the already existent back-ends for AnTLR, mainly the one that generates the Java parser. At the end, a* Perl *back-end for AnTLR will be developed based on the studies done in order to draw some conclusions. The main goal of this back-end is to automatically generate attribute based language processors in Perl using AnTLR assistance.*

To achieve this goal it will be undertaken a reverse engineering process over AnTLR to better understand its components and plan the best strategies to implement the tool. Being the StringTemplate engine the component responsible for generating the code in the target languages, it will be carefully analysed altogether with some Java *parsers generated by AnTLR, to perceive how the retargeting for* Perl *should be performed.*

To prove the tool is indeed a reliable solution and the generated parsers are efficient, performance tests will be done for this purpose.

[1] ANother Tool for Language Recognition

Acknowledgements

It is during our most special moments or hardest times that we realise friends play an indispensable role in our lives, sharing all the emotions of our fragile lives. This is why I firmly believe I would never accomplish what I did without their support, contributions, kind words and, most importantly, their unconditional love.

Firstly, I would like to thank *Professor Pedro Rangel Henriques* for all his wise words and for sharing his knowledge with me. For all the encouraging words, and for always being so patient with me. I am aware I have not been the easiest student to guide, however, he stood with me until the end of this project, helping and supporting me even when I was not motivated. Also for all the good advises he gave me throughout my master course, it was a pleasure to work with such a teacher and person. Thank you very much.

To *Alberto Simões* for all the knowledge he shared with me, especially about *Perl*. If it was not him, I would not love *Perl* as I do now. He taught and guided me in my *Perl* learning process and was always available to help me overcome problems that arose during the development of the back-end. I could not agree more with what my friend *Nuno Oliveira* once said about him. *"If there is a solution for your problem, he knows the answer!"*. Also, thank you for your funny jokes.

I would like to give a special thanks to *Daniela da Cruz* since, although she was not directly involved in this project, she was always available to help and give her contribution. She is, undoubtedly, a great and knowledgeable person. Thank you.

To *Ronald Blaschke* for his enormous efforts to come up with the AnTLR back-end for *Perl*. His impressive work was a valuable start up for my dissertation. Also for his support and availability to clarify any of my doubts. Thank you.

To *Terence Parr* and all the development team of AnTLR for combining their efforts to build a tool that gave a great contribution to the generation of language processors. Without AnTLR this project would not have been possible. Thank you.

To *Casiano Rodriguez-Leon* for his contribution to parser generation in *Perl* and for the knowledge exchanged in *CORTA 2010*.

I would like to thank *Nuno Oliveira* for his suggestions, advises and patience, since I lost count of how many times I interrupted his work, during my dissertation writing period, to ask for help or just to loosen up from work. Also for all the priceless and funny moments shared in the *gEPL* lab.

To *Daniela Fonte*, *Márcio Coelho* and *Mario Berón* for their help and contributions for this dissertation. Also for their continuous support. Thank you both.

I would like to take the opportunity to also thank my classmates from the *2004/2005* class for all the moments we shared together during our academic course. Thank each one of you.

I would like to extend my thanks to all my friends, for their support and love, especially *Ana Silva* for being always by my side when I desperately needed and such an amazing friend, and *Ana Macedo* for her unconditional support and encouraging words during the last year. Thank you everyone.

Lastly, I would like to thank the most impressive and important people I have ever met, my family. As pessoas que me criaram, educaram e fizeram de mim aquilo que eu sou hoje. Se agora represento alguma coisa neste mundo, devo-lhes tudo a eles. Obrigado por tudo o que fizeram por mim, por todos os sacrifícios que suportaram para me oferecer uma vida melhor, por toda a paciência que tiveram comigo ao longo de todos estes anos e pelos meus últimos anos de ausência. Obrigado por tudo pais e irmãos. Deixo uma palavra de especial apreço pelo meu irmão Sérgio, o meu melhor amigo, que me acompanhou e ajudou durante todos estes anos. Por tudo o que fez por mim ao longo destes anos, especialmente neste último, ficarei eternamente em dívida para com ele. Muito Obrigado!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This master thesis 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* [25] parser generators.

To build parsers by hand is a tedious task that, usually, takes a lot of time. So there are some available compiler generators to generate the parser algorithms coded in different programming languages (target languages), being the most popular target languages *C* and *Java*. *Perl* language has a considerable amount of useful mechanisms that encourage the development of parsers, positioning it as a tempting target for parser generators. However, *Perl* has some disadvantages that can lead to the deterioration of parsers efficiency, therefore placing them behind others implemented in more traditional alternative languages.

Although there is some tools to generate parsers in *Perl*, most of them reveal some drawbacks — such as the non-support for attribute grammars and lack of efficiency — which causes *Perl* to be discarded from projects that require high levels of efficiency. Some of these drawbacks are explained in further detail in Section 3.3 of Chapter 3.

It is intended with this master project to workaround the known flaws when generating *Perl* parsers. The starting point will be porting the parser algorithms generated by other tools to *Perl* language.

To accomplish this task, the compiler generator AnTLR[1][17] was chosen. AnTLR is a tool developed in *Java* that provides support for attribute grammars, well known

[1] ANother Tool for Language Recognition

for its professional contribution to LL(k) parser generation[2]. 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*.

Although *Perl* is a powerful language to recognise structured text through its regular expressions [12], it has some difficulties to recognise complex data that comply with a set of derivation rules. In these cases it is recommended to use a grammar to minimise programmer effort and maximise the parser efficiency [17].

With the assistance of this tool, it will be 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 desire results, it will be required to perform a reverse engineering process over AnTLR, to correctly understand its functioning and to obtain the maximum knowledge about the tool. On the other hand, it is mandatory to study other *Perl* parser generators, to be able to identify the best approach to take profit from *Perl* language when implementing a parser.

During the development process of the *Perl* code generator, it will be required to take a few precautions to prevent delays or future problems. To assure a faultless tool, it will be imperative to automatise a suit of tests that will coexist with the tool along its progress, allowing an agile and secure development.

The main goal is 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[3] grammar [8] (Appendix A.1) was chosen to serve as example throughout this document. *Lavanda* is a Domain Specific Language (DSL [16, 14]) to calculate

[2]Language processor generation, to be more accurate.
[3]http://epl.di.uminho.pt/~gepl/LP/

the number of laundry bags daily sent to wash by a launderette company frequently used by the *gEPL*[4].

Since this project aims the implementation of a language processor generator for *Perl* based on attribute grammars, any reference to grammars in this document will imply attribute grammars as well, unless a differentiation is made.

This document is organised in six different chapters. In Chapter 2, the problem in hands is studied and divided into parts for an easier understanding. It is also covered the first stage of the project, the grammar specification. In Chapter 3 it will be given a brief introduction of the basic concepts of languages, grammars and parser generation[5], and the role of attribute grammars in the generation process. It will also be presented a survey on the state of art of parser generation in *Perl*, enumerating some of the most common parser generators existing for *Perl*. These parser generators will be examined and their main features discussed. At the end, the reviewed tools will be carefully analysed and tested so conclusions can be drawn concerning their efficiency. In Chapter 4 AnTLR will be examined, discussing its architecture and main features. It will be presented the conclusions of a reverse engineering process over the tool, to study its main components: StringTemplate, the Runtime libraries and the Code Generator. This will help to understand the best practices to develop the back-end for *Perl*. Also it will be performed a detailed analysis over the generated parsers and their retargeting for other language, to have an idea of the expected code for the *Perl* back-end. The architecture and development process of the aimed tool will be discussed in Chapter 5, along with efficiency assessments and conclusions regarding what was achieved. In Chapter 7 conclusions will be drawn concerning the overall outcomes of this master thesis work.

[4]grupo de Especificação e Processamento de Linguagens at University of Minho

[5]many times along the document this term "parser generation" should be read as "LP generation"

Chapter 2

Generation of Perl LPs

This project aims at the construction of a code generator able to produce parsers in *Perl*. To achieve this goal, there are a few solutions which could be undertaken:

1. implement a new *Perl module* from scratch;

2. combine an already existing *Perl module* with an external (non-*Perl*) program unit (e.g. lexical analyser) to achieve better efficiency results;

3. take advantage of the parser generation algorithms used by a well-known tool, with the purpose of retargeting the generated parsers to *Perl*.

The first step would be, to choose a solution, from the few mentioned above, to deal with the task in hand.

The chosen solution was the latter, since the first two would, probably, share the same inefficiencies already found in the solutions currently available for *Perl* language. By creating a new module, the efficiency levels would remain almost equivalent to the levels of the other available modules, since it would end using some of the mechanisms used by them. Combining one of these modules with a non *Perl* tool would be a better approach than the latter, however, it would still lack some efficiency to be considered as an alternative to other programming languages available solutions. On the other hand, the chosen solution, besides taking advantage of the efficiency of reliable generation algorithms, it also benefits from the support of the large community that is involved in the development of the tool or using it for projects. From a number of available tools that could serve this project purposes,

AnTLR was the chosen tool since, besides the advantages mentioned above, this solution would profit from the favourable acceptance of this tool by the users and the systematic code generator provided by AnTLR.

Considering that it was chosen AnTLR to workaround the parser generation problem, a process of reverse engineering will need to be performed over this tool, to understand the mechanisms and all algorithms used to generate the parsers, along with its architecture, to realise which components need to be added or changed, and what will be the effects of those changes. These effects must be foreseen to prevent unexpected behaviours from the generated parsers or even from the AnTLR tool. The main question is, how to make AnTLR generate parsers written in *Perl*.

AnTLR provides its own metalanguage to write grammars and define the associated semantic actions. After compiling the grammar, AnTLR generates a parser in *Java*, by default, to recognise the language defined by that grammar. To generate the parser, AnTLR makes use of a template engine to avoid introducing any kind of print statement in the core of the internal code of AnTLR. The template engine is integrated with AnTLR, however, it is completely independent, having its own classes and an operating process unrelated to AnTLR. Nevertheless, it is required a bridge between both tools, to allow AnTLR to communicate with the template engine, providing all the required data for the generation of parsers. Although this bridge is already prepared to support the production of parsers in other languages, it is still required to notify the tool about the existence of a new target language.

So, besides providing a new set of templates, some changes are required in the internal structure of AnTLR to specify that a new language is available. To create these templates, it is crucial to have knowledge of what is expected to be generated in the end. Therefore, it is critical to define from the beginning which features are intended to be supported by the tool being developed, avoiding multiple redesigns of the tool. To prevent software design common mistakes, an extensive study of which features are to be supported by the tool is imperative. This study should cover all the requirements to build a parser in *Perl* with the assistance of AnTLR.

2.1 General Overview

As mentioned before, when introducing this chapter, it is crucial to define what is the focus of this project and specify all the features to be developed, analysed and

translated, to plan an implementation which realises the intended behaviour. These features include the grammar specification, the parser and the semantic, amongst others.

The specification of the problem in hands can be represented as a sequence of three stages: the language specification via grammar; the language processor (LP) generation; and the use of the generated program to process sentences of the specified language. All these stages depend on each other to be successfully planned and completed. This process (generating and using parsers in *Perl*) defined by the three stages above, is illustrated in the Figure 2.1.

In the first stage, a particular grammar must be specified, using the AnTLR metalanguage defined for this purpose; this grammar shall define the syntactic and the static and dynamic semantics of the intended new language. From the syntax specification included in the grammar, it is possible to write the expected parser which recognises the input, i.e. the sentences of the language defined by that grammar.

The expected parser is the starting point for the semantic analysis and code generator. Both modules can also be generated from the attribute evaluation rules included in the grammar. The core of all the three components required to generate a processor for the specified language are already provided by AnTLR. The internal functioning of the code generator and all its components are described in Section 4.4.

With a LP generator like AnTLR, it is possible to generate correct processors in *Perl* language, however, it is required a runtime library to implement the procedures invoked in the generated LP. The runtime library is responsible for implementing the data structures and classes required in the parsing process. This means that, without the runtime library, the generated parsers are syntactically correct but not runnable as supposed.

After having the grammar implemented in AnTLR metalanguage, generated the LP and implemented the runtime library, all that remains to be done is to test the generated parser to inspect its behaviour and outputs. For a parser to be considered correct, it must recognise an input which respects the syntax and semantics of the language defined by the grammar implemented with AnTLR. This process is explained in more detail in Section 5.1.

To create a parser generator tool, it is imperative to define, previously, which parsing

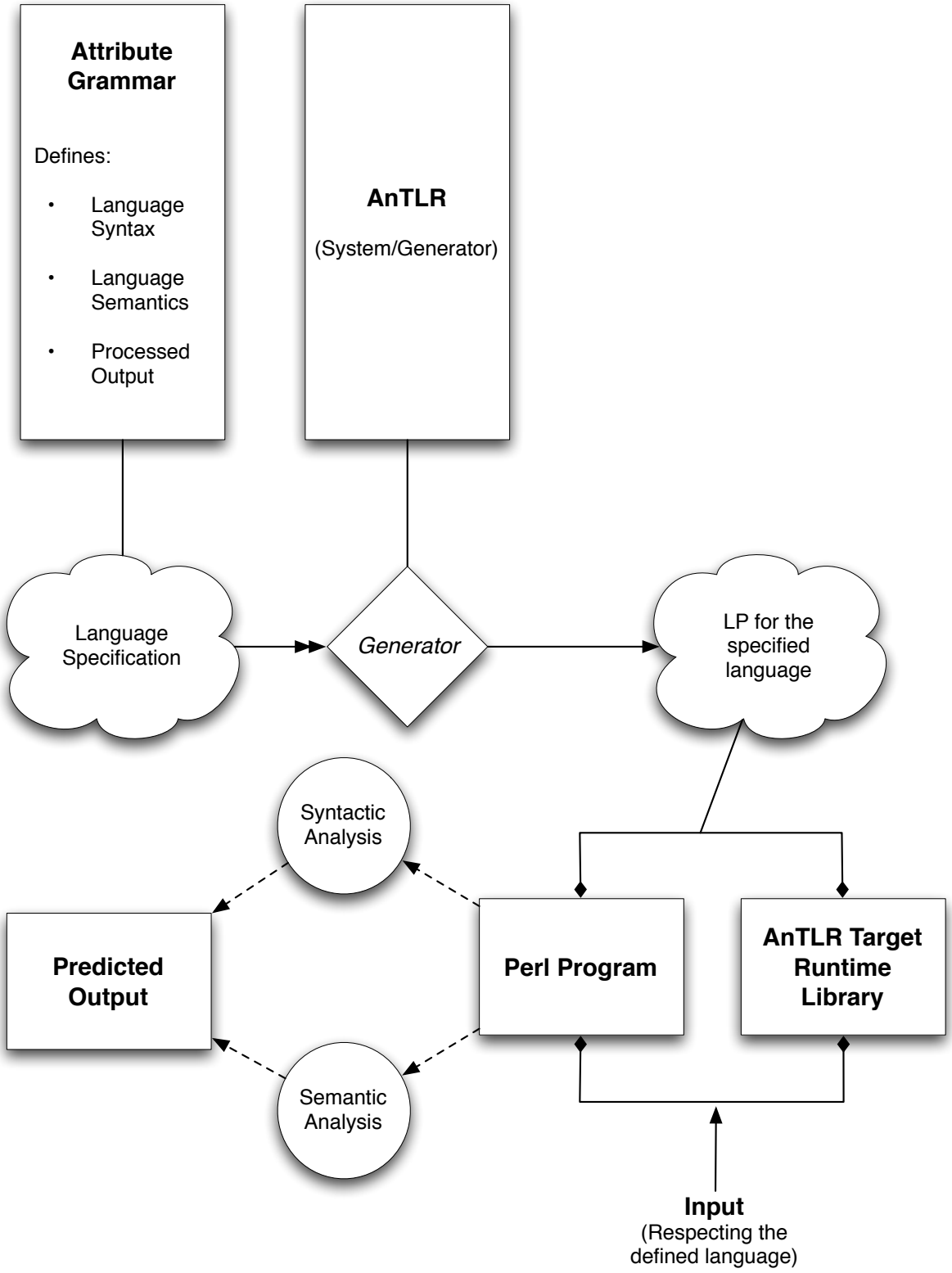


Figure 2.1: Automatic generation of Perl language processor.

algorithms to support, grammar features and target language. As already mentioned in the introduction, in the context of this Master Thesis we aim at generating parsers in *Perl*. Since AnTLR was chosen to assist in this task, the parser algorithm will be a k-lookahead recursive-descent one.

AnTLR also offers support for attribute grammars so, it is possible to say that the tool will generate language processors based on attribute grammars using the attribute evaluation rules to implement the semantic analysis and translation. All the remaining features implemented by AnTLR will also be present in the *Perl* code generated, since it is only a matter of transcribing the templates for the new target language. The other dependencies are implemented in the runtime library.

Defining all the tool requirements at the beginning will ease its development, avoiding major changes in future maintenance.

To generate a LP in *Perl* for a particular language is only a matter of providing the grammar that defines the language and compile it with AnTLR. The grammar should also contain all the semantic rules associated with its productions. However it is important to understand the relationship between the specification (the grammar) language and the LP code generated (in *Perl*).

AnTLR provides its metalanguage to write grammars. A grammar, when compiled by AnTLR, gives rise, by default, to a syntactic analyser in *Java* to parse the language defined by it. This *Java* parser can be extremely useful as the basis for the *Perl* parser.

Taking as example the production "lavanda" from *Lavanda* grammar (Figure 2.2), the respective *Java* code generated for it is shown in Figure 2.3. Note that, the explanation of the code is given in Section 4.5.2, since it is not important at this stage. What is important to retain at this moment from this code fragment is that it gives a significant idea of the complete generated language processor code; it is possible to see how a production is represented, by a method; its main algorithm and how it invokes other production methods; how attributes associated with local symbols are handled, etc.

Analysing the complete *Java* code produced by AnTLR for *Lavanda* grammar is very helpful in understanding how the grammar components are treated and transformed by AnTLR; this comprehension is an actual aid to craft the code generator for *Perl*.

```
lavanda : cabec sacos
        {
         System.out.println("Total␣sacos:␣" + $sacos.nSacos);
         for(Object o : $sacos.outEnv.keySet())
           System.out.println((String) o + "\t" + (Integer)
              $sacos.outEnv.get(o));
        }
;
```

Figure 2.2: Production "lavanda" extracted from *Lavanda* grammar, in AnTLR notation.

```
public final void lavanda() throws RecognitionException {
    Lavanda2Parser.sacos_return sacos1 = null;
    try {
        {
        pushFollow(FOLLOW_cabec_in_lavanda23);
        cabec();

        state._fsp--;

        pushFollow(FOLLOW_sacos_in_lavanda25);
        sacos1=sacos();

        state._fsp--;

        System.out.println("Total␣sacos:␣" + (sacos1!=null?
           sacos1.nSacos:0));
        for(Object o : (sacos1!=null?sacos1.outEnv:null).
           keySet())
        System.out.println((String) o + "\t" + (Integer) (
           sacos1!=null?sacos1.outEnv:null).get(o));
        }
    }
    catch (RecognitionException re) {
        reportError(re);
        recover(input,re);
    }
    finally {
    }
    return ;
}
```

Figure 2.3: Method "lavanda" extracted from *Lavanda* LP, generated in *Java*.

For instance, the "lavanda" production would be translated into the method illustrated in Figure 2.4, when targeted to *Perl*.

Besides being helpful to know what to generate when dealing with a particular component of the grammars implemented in AnTLR, the grammar specification is also important for the generation of tests; actually, input text can be drawn from it, to be used in tests to evaluate the correctness and performance of the parsers generated.

```
1  sub lavanda {
2      my ($self) = @_;
3      my $sacos1 = undef;
4      try {
5          {
6           $self->push_follow($FOLLOW_cabec_in_lavanda30);
7           $self->cabec();
8
9           $self->state->_fsp($self->state->_fsp - 1);
10
11          $self->push_follow($FOLLOW_sacos_in_lavanda32);
12          $sacos1 = $self->sacos();
13
14          $self->state->_fsp($self->state->_fsp - 1);
15
16          print 'Total␣sacos:␣' . $sacos1->{nSacos} . "\n";
17          foreach ( keys %{$sacos1->{outEnv}} ) {
18             print "$_\t" . $sacos1->{outEnv}->{$_} . "\n";
19          }
20         }
21     }
22     catch {
23         my $re;
24         $EVAL_ERROR = $_;
25         if ( $re = Exception::Class->caught('ANTLR::Runtime::
               RecognitionException') ) {
26             $self->report_error($re);
27             $self->recover($self->input, $re);
28         }
29     }
30     finally {
31     };
32     return ;
33 }
```

Figure 2.4: Block of *Perl* code equivalent to the method "lavanda".

Chapter 3

Comparative Study of Parser Generators in Perl

Computer programs often need to process some input. To do that in a systematic and safely way, this input should be a valid sentence of a given language (defined by a grammar), otherwise it is discarded by the program. This task is performed by a parser, an important component of several computer programs [13].

In the early days of language recognition, parser algorithms were very complex and the semantic actions were, typically, mixed together with the parser actions, making the parser extremely hard to maintain. Also writing a parser by hand was a tedious, bug-prone and time consuming task, a task that experienced developers would avoid [10]. Since parsers play an important role in language recognition, such as interact with the lexical analyser, report parsing errors, and call semantic actions defined by the user, considerable effort has been made in the development of parser generators, with the goal of automatising and simplifying the process of creating parsers [13]. This would allow tedious tasks to be automated and generated parsers to be easily folded into target applications, decreasing considerably the amount of time wasted writing a parser by hand for a specific task, and strongly increasing its performance.

3.1 Languages and Grammars

A language, in computer science, is a system of symbolic communication between programmers and machines, designed to help solve computation problems. These languages are denominated programming languages. Programming languages provide the users the necessary constructs to express the algorithms that solve their problems in a terminology which the machine is able to understand and execute [1, 5, 23]. Depending on the nature of the problem, the user may choose between traditional (General-Purpose programming Languages, GPL for short) or domain oriented programming languages, usually designated as Domain-Specific Languages (DSL [22]).

Languages have a finite alphabet associated, which allows the creation of sentences. The set of these sentences are either finite or infinite and it stipulates the type of language. To attempt a finite specification of an infinite language, a set of grammatical rules is defined. The set of rules denote a formal grammar and from them it is possible to derive or produce sentences for the defined language [1, 23].

A grammar is a finite set of derivation rules, or productions, which define the structure of a language. Grammars can also be used in the reverse derivation, being possible to determine if a sentence belongs to a particular language [5, 23].

Noam Chomsky[1] has categorised the formal grammars in four different classes (the *Chomsky Hierarchy*). According to the *Chomsky Hierarchy*, grammars can be classified as unrestricted, context-sensitive, context-free and regular, being the third the basis for the structure or syntax of most programming languages. However, the grammars only specify the syntax of a language, they do not contemplate expression semantics. There are some methods to assign meaning to the sentences, such as attributes [5, 15, 23].

Attribute Grammars

Attribute grammar is a formalism to define a set of attributes for each symbol (nonterminal or terminal symbol) of a specific grammar, acting as records for holding information about the associated symbols. Attributes are useful to define semantic actions to be processed by the parser in the nodes of the parse tree. These attributes

[1] http://www.chomsky.info/

can be *inherited* or *synthesized* [1, 15].

Inherited attributes are passed down the parse tree from parent nodes to theirs children and siblings, extending the semantic information to the nodes on the bottom of the parse tree. In code implementation, attributes of this class are defined as parameters of the methods. Synthesized attributes are consequence of the evaluation of attribute semantic rules — functions synthesized or inherited applied to attributes of the local symbols — in the subtree of the production applied in a specific node. Such attributes are useful to carry semantic information up the parse tree [1, 15]. When dealing with code implementation, usually, they are equivalent to the methods return values.

Attributes grant the deliverance of semantic information from any node of the parse tree to anywhere else, abiding by the constraints in a formal way.

Attribute grammars are mostly useful for providing mechanisms to specify the relationships between semantic rules. This relationships help to obtain the evaluation order of semantic rules and, determine and define the storage requirements for the attribute values [24]. Generally, these relationships are represented by a dependency graph, from where the evaluation order is inferred [1, 15].

Defining these relationships evince a noteworthy advantage when changes need to be done on a grammar design. Due to attribute grammars abstraction level of the evaluation order of productions or semantic rules, changes on the relationships amongst the semantic rules are needed, however no changes are registered on the order of the semantic actions or variable declarations [24].

Like any other generated parser, parsers based on attribute grammars are expected to offer, at the worst case scenario, a similar performance and complexity than hand-built parsers.

3.2 Parsing

As mentioned before, the structure or syntax of a language are defined by a formal grammar. To perform the syntactic analysis of a sequence of tokens, a syntactic analyser is required. The syntactic analyser consumes the sequence of tokens and determines if they correlate with the grammatical structure of the language defined by the grammar. This process is often referred as parsing [2, 23].

During this process, the syntactic analyser, or parser, generates a syntax tree — *Abstract Syntax Tree (AST)* — to assist in the recognition process. This tree represents the structural relationships between the symbols of a particular input. The leaves of the tree are the terminal symbols (tokens) of the grammar and the nodes correspond to the nonterminal symbols [2, 23].

There are a few algorithms to construct these trees, but, usually, parsers are either *top-down* or *bottom-up*, determined by the construction order of the parsing tree nodes [1, 2].

A parser can be created for any grammar, but its efficiency will always rely on the parsing method chosen for its implementation and how the grammar is built.

Top-down and Bottom-up Parsing

Top-down

The *top-down* method (LL) starts the construction process of the parse tree at the most abstract level of the language, the grammar axiom, proceeding to the leaves, by trying to derive the input string from the grammar. During the transition, a subtree is constructed for each node of the tree. This method can be defined as goal-oriented, as it starts with one rule and then tries to match the alternatives [1].

The most common implementation of LL parsers is called *recursive-descent*, where a procedure is defined for each nonterminal present in the grammar [13].

Although, efficient *top-down parsers* are easier to write and understand when compared to *bottom-up*, they have some grammar restrictions. Left recursive grammars can cause LL parsers to go into infinite loop, by trying to expand a rule indefinitely without consuming any input [1].

Bottom-up

On the opposite of *top-down* strategy that attempts to find a leftmost derivation for an input string, *bottom-up* (LR), also known as shift-reduce parsing, attempts to find a rightmost derivation for the same input. *Bottom-up parsing* attempts to

create a parse tree for a given input, by starting to match at the leaves, trying to reduce the input to the grammar axiom [1].

LALR is one possible implementation of *bottom-up parsing*, being used by the parser generator Yacc [4]. LALR parsers are not as powerful as LR but are simpler to write and implement [17].

Bottom-up based parsers are harder to build, however they are more powerful and have a larger class of acceptable grammars than *top-down parsers*.

3.3 Parser Generation in Perl

A parser generator is a program that, based on a grammar, creates a parser for the language defined by that grammar. It is believed that having a grammatical description of a language and a parser generator, it is easier to achieve a correct and better encoded (more elegant) parser, than doing the parser by hand [1]. Also, usually, they are more efficient and readable, since it is supposed the generated parser to be, at least, as good as the hand-built.

A parser generator, besides generating a parser for the language defined by the given grammar, guarantees that grammar is unambiguous and prevents conflicts that can arise during parsing. It also prevents interruption of the parsing process when an error is detected in the input, by recovering from that situation [13].

Nowadays, software systems that require a lot of input processing, always need a powerful parser that can take the task and solve it without too many resource consumptions. In these cases, parser generators can be extremely valuable and save a lot of development and processing time.

Perl language provides powerful mechanisms that can be proven very useful on the implementation of 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 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 that have the ability of generating parsers by outputting the parser algorithms so they can be used with simplicity. 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 amounts 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. It was already announced that *Perl 6*, besides regular expressions, will also support full grammars, helping to improve readability and to decrease the syntax and parser complexity.

3.3.1 Available Tools

In the *Comprehensive Perl Archive Network* (CPAN[2]) 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. Four tools were chosen to be carefully analysed and tested for performance and resource consumptions. The most used, the most robust, the most elaborated and the most recent:

- **Parse::RecDescent** – one of the most used tools, that generates 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[3]

Keep in mind that this list could be larger, since there a few more parser generators for *Perl*, such as the back-ends for *Byacc*[4] or *GNU Bison*[5], however those listed

[2]http://www.cpan.org/
[3]This tool is only supported in recent *Perl* versions (> 5.10).
[4]http://free-compilers.sharnoff.org/TOOL/BNF-23.html
[5]http://linux.maruhn.com/sec/bison.html

above are the most popular used, exception made for the `Regexp::Grammars` that is still under appreciation.

3.3.1.1 Parse::RecDescent

`Parse::RecDescent` [6] 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 the 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, performing the lexical and syntactic analysis, and achieving the results on the fly. The drawbacks are the incapacity to deal with left recursive parsers and its efficiency when dealing with large inputs. Therefore, it is not recommendable for cases where the performance is an issue. Figure 3.1 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] }
};
```

Figure 3.1: `Parse::RecDescent` parser for s-expressions.

3.3.1.2 Parse::Yapp

`Parse::Yapp` [9] is one of the oldest parser generators in *Perl* and probably still one of the most robust. It is based on `yacc` [4]. 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 (*yapp*) that, when executed over an input grammar file, generates a *Perl Object Oriented* (OO) parser. This script also provides a flag that, if activated, allows the generation of a standalone parser.

This module only supports *Backus-Naur Form* (BNF) rules to write the parser grammar. Also, `Parse::Yapp` does not include lexical analyser features, forcing the user

to provide one. Gratefully, there are some useful modules in CPAN to help in this process, such as `Text::RewriteRules` [21].

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

Figure 3.2: `Parse::Yapp` parser for s-expressions.

3.3.1.3 Parse::Eyapp

`Parse::Eyapp` [19] 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, making it a possible alternative to `Parse::Yapp`.

The most relevant features from `Parse::RecDescent` implemented in this module include auto-actions and the named access to semantic rule values.

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

Figure 3.3: `Parse::Eyapp` parser for s-expressions.

3.3.1.4 Regexp::Grammars

`Regexp::Grammars` [7] 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;
```

Figure 3.4: `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.

3.3.2 Tool Analysis

Three different grammars were chosen to allow the analysis of the four modules described earlier: The Swedish Chef (Appendix A.2), a simple but relatively large grammar, with an high number of semantic actions; The *Lavanda*, a DSL to describe the laundry bags daily sent to wash by a launderette company; and an highly recursive grammar to match s-expressions (Appendix A.3).

```
The Perl language is well known mainly for its flexibility
and for providing powerful text processing mechanisms, along
with powerful regular expressions and related functionalities
to facilitate data extraction and manipulation. Although
regular expressions are quite helpful in the construction of
parsers and Perl 5.10 already supports recursive descent
parsers, these mechanisms are not always enough to recognise
complex textual structures and parse it into useable data
structures.
```

Figure 3.5: Example of Swedish Chef input.

Though The Swedish Chef language was chosen to assist on the tests and evaluations

```
13-04-2007
PstVVerde
1 sergio(casa-br-la 3, corpo-br-alg 3, casa-cor-la 1).
2 ruben(casa-cor-alg 5, corpo-br-la 3, casa-cor-alg 1).
3 dani(casa-cor-fib 3, corpo-br-alg 5, casa-br-fib 3).
4 hugo(casa-cor-fib 1, corpo-cor-la 3, casa-br-fib 3).
```

Figure 3.6: Example of *Lavanda* input.

```
(let res (add (mul 1 2)(div 8 4) ) )
```

Figure 3.7: Example of a s-expression.

of the tools, it is not excluded that there are better solutions to write a parser to this language. Since most of the work is done at the lexical analyser level, it would be easier to achieve a more readable parser using, solely, *Perl* native regular expressions without losing any efficiency.

Each of the chosen grammars were implemented in all the modules, so that careful analysis could be performed. Concerning the usability, it was performed a evaluation of the grammars and generated parsers, and concerning the efficiency, there were carried evaluations of time and memory consumption.

The main goal of these analysis is to compare the most used parser generators for *Perl* from the available tools, and evaluate a few important criteria and their efficiency when targeting different languages. This will make it possible to have an overview of the state of parser generation in *Perl*, how the available tools deal in the presence of some adversities and to support the need of a new tool for *Perl* to be considered a valid alternative to other languages. To be a valid alternative, the tool must aim for more efficiency than the already available tools and must meet some design requirements.

3.3.2.1 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 to improve 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 of example 3.1. With named access the behaviour is exactly the same, but it is easier to understand the semantic actions. Besides `Parse::RecDescent`, `Parse::Eyapp` and `Regexp::Grammars` also supports this functionality.

`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 parser grammar, as happens with standalone 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 runtime that is not possible to serialise for future use (note that this is also an efficiency problem), although it is possible to generate a standalone parser with `Parse::RecDescent`.

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 `Parser::Eyapp` is a `Parse::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.

3.3.2.2 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 the `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 also mix the lexical analyser with the application 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 know Unix lexical analyser `flex` with `Parse::Yapp` can lead to an increase of the parser efficiency and decrease of required memory [20].

3.3.2.3 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 an 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 (AGs) [24], 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 rule 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.

3.3.2.4 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 [11], 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 standalone *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.

3.3.2.5 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 behaves 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, 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 description 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.

3.3.3 Tests and Comparisons

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 because test results 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 work as it would lead to unfair comparisons.

As shown on Table 3.1, 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 3.2). 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!

Almost the same happens when parsing the *Lavanda* language as can be visualised in Table 3.2. `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 since it 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.

Table 3.1: 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 3.2: 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				

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 3.2, 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 3.3) it was only used one test

file containing a large expression (32 377 chars 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 s, closely four times more than `Parse::Yapp`. `Regexp::Grammars` could not finish parsing because of a *Perl 5.10* bug (already reported).

Table 3.3: 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 3.4 presents the memory consumption during the parsing of the *Lavanda* test files. According to Table 3.2, 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 3.4 shows, the parser generated by `Regexp::Grammars` quickly becomes out of memory to parse the files. Notice that, this out of memory interrupt message does not means, however, that the machine is out of memory to run the program, but, that the memory allocated by the operating system to execute the program has been exhausted. `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`[6].

Table 3.4: 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				

[6]http://valgrind.org/

Tables 3.5 and 3.6 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 because 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.

`Parser::Eyapp` offers useful support for semantic actions, such as auto-actions or 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 3.5: 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

Table 3.6: Module Analysis.

Module	Debugging	Generated Parser Readability	Integration with External Code	Development Time
Parse::Yapp	+/−	+/−	++	+/−
Parse::Eyapp	+/−	+/−	+	+/−
Parse::RecDescent	+	NA[1]	+	−
Regexp::Grammars	++	NA	+/−	−−[2]

`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.

`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.

Note 1: `Parse::RecDescent` gives support for integration with other code, however the parser is not stored by default in a separate file.

Note 2: `Regexp::Grammars` it has a poor development time mainly due to the *Perl 5.10* bug. The parsers, however, are easy to create and integrate with other code blocks.

Chapter 4

AnTLR

AnTLR is a tool designed and developed in *Java*, by Terence Parr[1], which automates the construction of compilers and parsers based on $LL(k)$ grammars. It provides support to predicates and attribute grammars, and it is well-known for its professional contribution to $LL(k)$ parser generation. With AnTLR it is possible to generate recognisers in a variety of languages, such as *Java*, *C*, *C++* and *Python*, and it was developed taking into account the possibility of extending the original tool with new code generators for specific target languages. There are some other $LL(k)$ parser generators besides AnTLR, so why use it?

It is common to find some available parsing tools which are mainly focused on the language recognition process, ignoring other important tasks, such as, the proper integration of an efficient lexical analyser or the construction of an AST to assist in the syntactic and semantic analysis. Also, programmers tend to seek powerful and easy to use language tools to solve their problems, which should be able to provide understandable mechanisms and some features to decrease the development time of syntactic analysers, avoiding redundancy of procedures and tasks. Besides, AnTLR generates output that is easily folded into users applications or targeted to other programming languages [18].

Despite the fact that *LR* or *LALR*, as the one generated by *YACC*, are more powerful parsing algorithms than *LL*, programmers still choose recursive-descent parsers in order to gain more flexibility, better error handling and recovery, since *LL* based recursive-descent parsers tend to be more intuitive and easy to understand [24].

[1] http://www.cs.usfca.edu/~parrt/

These features, along with the integration of the template engine StringTemplate (*AnTLRv3* only) to assist in the code generation, made AnTLR to be reckon as a versatile tool on the construction of recognisers in many different languages such as *C*, *Python* and *Java*.

4.1 Architecture

AnTLR, when installed, builds an hierarchy of files divided into some components. These components are responsible for the compilation of the grammars or tree grammars, and the generation of the lexers and parsers. Each of these components is a combination of classes implemented in *Java*, exception made for the Code Generator templates, that are written in distinct languages. In Figure 4.1 it can be observed the architecture of AnTLR, showing all the involved components and how they interact with each other.

From the starting stage of running the AnTLR tool over a specific grammar, to the point where the lexer and parser are generated, a few processes are executed to perform every task required to obtain the expected results. The main class, "Tool", is the responsible for starting the whole process of compiling the grammar and, accordingly with the analysis performed, generate a parser able to recognise texts belonging to the language defined by that grammar. This class belongs to a set of classes which constitute the core component of AnTLR.

Besides this component, there are other four important components. They are, the Analysis component, StringTemplate, Runtime library and the Code Generator. These five components, along with some miscellaneous classes, interact and depend on each other to accomplish their tasks, and share access to the data stored through the compilation process of the given grammar.

The Analysis component, as the name indicates, is responsible for the analysis of the grammar. It creates its own data structures to store all the information obtained while analysing the grammar, namely, the productions and the associated semantic information. During the analysis of the productions and respective predicates, information for the construction of a DFA is gathered or generated, to assist in the parsing process. The data required by the parser, to build a DFA during the parsing process, is stored in objects and later used by the Code Generator. The Code Generator outputs this data, to a parser or a lexer, in a structured form, enabling

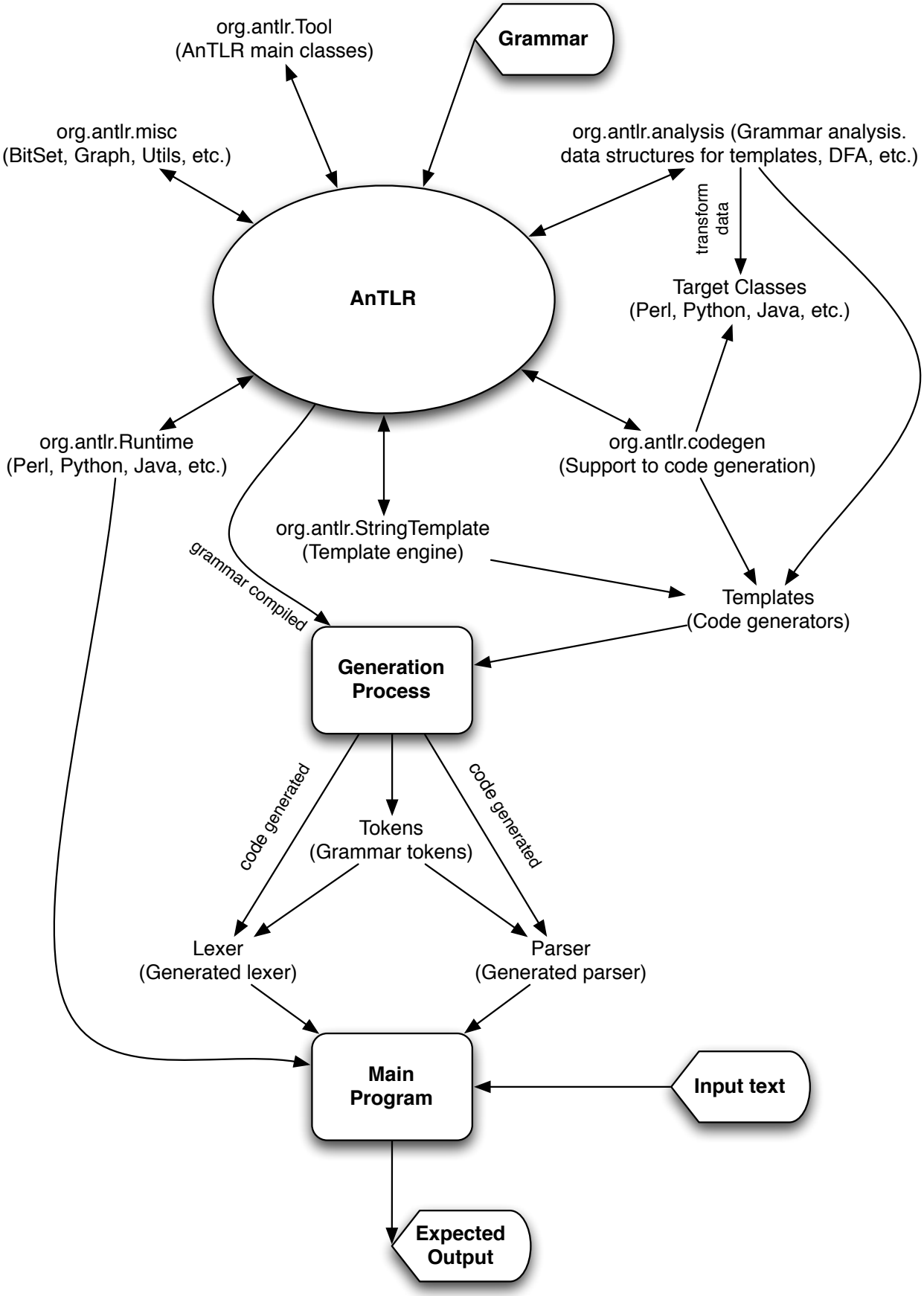


Figure 4.1: AnTLR Architecture.

the creation of a new object of "org.antlr.DFA" type (class found in the Java Runtime library). The classes belonging to this component are often imported by other component classes, especially by the Code Generator templates.

The Runtime libraries are of utmost importance for the correct execution of the generated parsers, but play no role in any other AnTLR task. The Java Runtime library is an exception to this, since it is also of great importance for the compilation process of AnTLR grammars. The creation and *modus operandi* of the DFA, AST, token streams, parser and lexical analyser, for each supported language, are described and implemented by this library. The Java Runtime library is described with further detail in Section 4.3.

The StringTemplate engine was introduced by AnTLR to convert the code generation procedures to a template based model. It provides its own template creation 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 target classes are called by the analysis component to transform specific data that needs to be converted to meet some target programming language requirements. The StringTemplate engine is explained in Section 4.2 and the Code Generator in Section 4.4.

These five components, all together, are responsible for compiling the grammars and generate all the information required by the code generator to output the parser and lexer correctly. Though, the StringTemplate and the Code Generator are present in the compiling process with less preponderance than the remaining components, they are of major importance for the code generation process.

After AnTLR compiles the grammar provided by the user, the data is stored in the data structures created or instantiated by the analysis classes. These data structures are imported by the templates and used to fill its gaps with the corresponding information.

The template files are called template groups, since they contain multiple templates inside, covering all the possible parser and lexer implementations of the AnTLR supported features. The mapping between the template gaps and the data delivered by the analysis classes, is accomplished by StringTemplate, which, along with *AnTLR*, are responsible for generating the lexer, tokens and parser files.

With these files created, it is only a matter of implementing a main program that manages and treats the information returned by the lexer and parser. This program must import the corresponding target language Runtime library, and should receive the input text from a char stream (*stdin*, text or binary file, buffer, etc.) to work properly and obtain the desired results.

A scheme of a Language Processor, generated by AnTLR, functional specification can be observed in Figure 4.2.

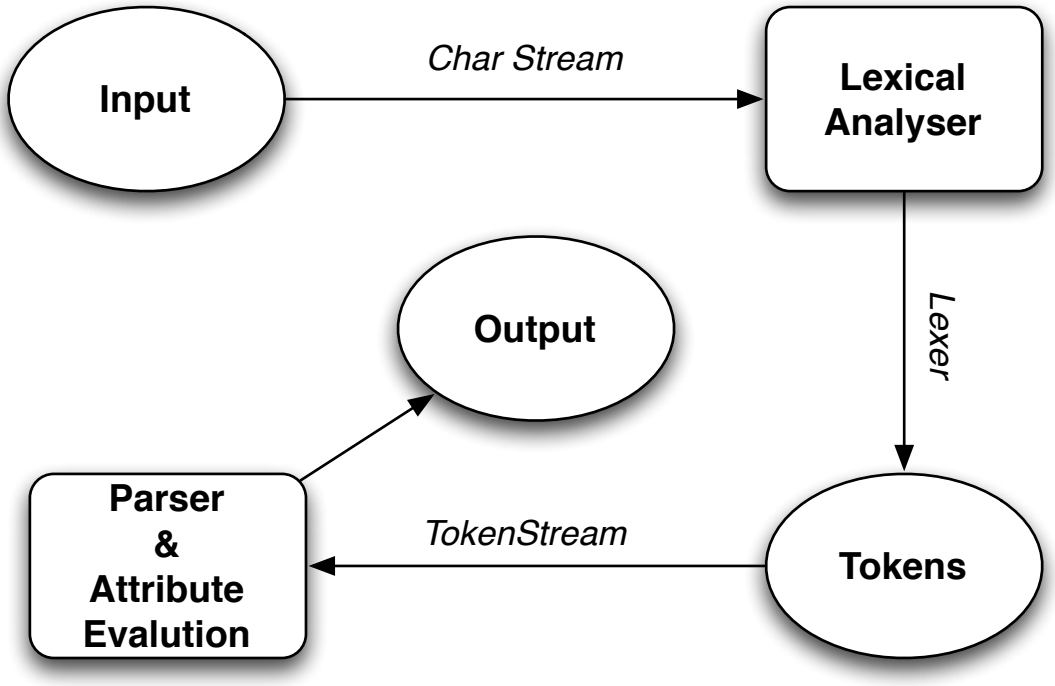


Figure 4.2: Functional Specification of a Language Processor.

4.2 StringTemplate Engine

Structured output was usually obtained through the use of the traditional print statements practice. Obviously, this strategy leads to a considerably increase of the code size and it is error prone. These drawbacks are present on the parser generation, where direct output of the parser being generated can deteriorate the readability or increase the complexity of writing the grammar that defines the language to be recognised by the parser. Furthermore, it increases the complexity of maintenance, since any change in the grammar syntax implies the reformulation of the computations and logic. This is also an obstacle on the process of re-targeting

a parser generator to other programming languages.

A viable alternative to this method is the use of templates. Templates are, in its most abstract form, a unit of source code that defines the output structure of a specific document, which can be combined with data stored in databases, documents, data structures, among others, and processed by a template engine, originating a single or a set of documents. Template engines are often used to generate web documents — such as XSLT for XML — but they can also be found in other fields. The introduction of template engines on parser generator tools, at the beginning, raised some doubts and started some interesting discussions, among experts, whether templates were a viable solution.

Templates are beneficial for maintenance of parser generators, as they isolate the generated code structure from the grammar recognition and automata construction. These steps can be performed one after the other, and isolated from each other.

When groups of templates responsible for the generation of the output code are isolated from the grammar actions, they are easier to maintain and understand. This not only makes it easier to maintain, but also easier to re-target. Programmers dealing with re-targeting do not need to create a different code generator for each target language, they just need to provide a new group of templates for each programming language [17].

The template engine StringTemplate integrated with AnTLR provides all these template features which makes AnTLR a suitable and recommended tool for parser generation.

4.2.1 StringTemplate Templating System

StringTemplate is a template engine available, in the form of a library, for *Java*, *C#*, and *Python*. Basically it is an output generator (called a code generator on the language recognition discipline) based on templates. These templates are strings, which can be found in the code or in a separate file, that can be combined with template expressions. Template expressions are similar to variables or functions, that can be invoked with an associated set of parameters designated as *attributes*. Note that these attributes are not related to grammar attributes.

StringTemplate templates provide useful mechanisms for output generation such as auto-indentation, map operation (automatic generation based on the iteration over

lists), pretty-print features, among others. They also simplify and considerably improve the readability of a grammar when it is required to buffer up rule elements for later output, mostly because it decreases the number of semantic actions embedded in a rule [17].

Templates play a determinant role on code generation, not only for improving the readability of grammars, by isolating the output structure from the generation logic and computations, but also for stimulating the reuse of components and decreasing the complexity — such as the code size — of the implemented grammars. This is achieved by grouping templates in a single file and loading it when required. Moreover, this aims at avoiding code replication in rules with output alike, allowing a straightforward edit of reused templates, and to distinguish output actions from computations.

One of the most interesting features of StringTemplate is the lazy evaluation when a reference is made to an attribute. StringTemplate only evaluates the attributes and passes them to the templates before generating the output, not when the attributes are referenced for the first time. This property removes any kind of computation of order dependencies since attributes can be mentioned anywhere in the code even if the template that references those attributes has not been created. Furthermore, it prevents dependency bugs and programmers from wasting time trying to anticipate any order dependency that could rise [17].

4.2.2 AnTLR StringTemplate Interface

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, formed by a few classes, 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.

Generate a parser in several languages with AnTLR is a matter of specifying the target language, by using the directive *option* (assuming that the group of templates for the language is available). This directive specifies which group of templates should be loaded by AnTLR to generate the output [17]. This is really useful and practical since there is no need to modify or recompile the code to retarget it to a new language target, it is only a matter of providing a new group of templates. Moreover, to generate a different output for the same language it is just a matter of replacing the current group of templates for the new one. StringTemplate is seen by AnTLR as a component just like the Runtime library or the Code Generator. It is, basically, a group of classes that interact with AnTLR and has an important role in the generation of the code, since the Code Generator is a set of templates recognised by StringTemplate, which handles the information that results from the analysis of the grammar implemented with AnTLR, and returns it in a structured form to generate a program able to respond correctly to the needs of the user.

Though it is possible to write templates directly into the grammar designed with AnTLR metalanguage, the following example will not cover that cases since this practice was never put to use in this project. The idea is to preserve, as most as possible, the transparency of the code generation process.

The template in Figure 4.3 was taken directly from the *Perl* Code Generator, and it is responsible for the generation of the block of code that handles the cases of one or more alternatives, often represented by the symbol "+" in *EBNF* notation.

The name of the template observed in the Figure 4.3 is "positiveClosureBlock". When invoked, during the construction of the parser, it requires some information to generate the block of code for which it is responsible. These variables set as parameters in the prototype of the template are denominated attributes. The attributes hold information related with the grammar compiled by AnTLR.

The information required to build a parser is analysed by the analysis classes of AnTLR and is stored in data structures (also classes) for later use. These classes are then included in the templates so, the attributes can fetch the information required

to build the parser and keep it consistent with the specification of the implemented grammar.

The attributes are found in the templates surrounded by chevrons, as it can be verified by observing the last example. These attributes can vary from plain strings, attributes with properties or even lists. Attributes "fileName" and "description" are two examples of plain strings. This type of attribute has only the attribute name surrounded by chevrons and evaluates to the value of the associated method "toString" (*e.g. fileName.toString()*). The "alt" attribute is an example of a list of values. The "<alt:altSwitchCase>" works as a *"foreach"* in *Perl*. It applies the template "altSwitchCase" to each of the values contained in "alt".

```
1  positiveClosureBlock(alts,decls,decision,enclosingBlockLevel,
       blockLevel,decisionNumber,maxK,maxAlt,description) ::= <<
2  // <fileName>:<description>
3  int cnt<decisionNumber> = 0;
4  <decls>
5  <@preloop()>
6
7  loop<decisionNumber>:
8  do {
9      int alt<decisionNumber> = <maxAlt>;
10     <@predecision()>
11     <decision>
12     <@postdecision()>
13
14     switch (alt<decisionNumber>) {
15         <alts:altSwitchCase()>
16
17         default :
18            if ( cnt<decisionNumber> >= 1 )
19                          break loop<decisionNumber>;
20            <ruleBacktrackFailure()>
21            EarlyExitException eee =
22              new EarlyExitException(<decisionNumber>, input);
23            <@earlyExitException()>
24            throw eee;
25     }
26     cnt<decisionNumber>++;
27 } while (true);
28
29 <@postloop()>
30 >>
```

Figure 4.3: Template to generate cases of one or more alternative (+) in *Java*.

Besides attributes, also templates can be found surrounded by chevrons. Templates are always followed by curve brackets surrounding the list of required parameters. "ruleBracktrackFailure" is an example of template invocation inside another template, "positiveClosureBlock". Everything else not surrounded by chevrons is treated as plain text by StringTemplate, and it is sent to the output parser as it is displayed.

The layout of what is generated by this template is presented and explained in Section 4.4.

4.3 AnTLR Java Runtime Library

The property of generating parsers in several languages is possible at the exchange of some easiness of portability. The parsers generated by AnTLR are not standalone, since they depend on a Runtime library to achieve any desired result.

The Runtime library is a component of AnTLR, responsible for assisting in the execution of the generated parsers. The code generators integrated with AnTLR, each have a corresponding library, essential to the correct functioning of the target language, which control the parser behaviour, and implement the data structures for storage purposes, the several mechanisms of AnTLR, such as backtracking, the structures to handle the input, the debugging mechanisms, the parsing tree construction and other components required in the parsing process, such as, the interface between the lexical analyser and the parser.

From all the Runtime libraries available for AnTLR, only the library for *Java* has some extra roles in the construction of parsers, no matter which the target programming language. The *Java* library is used, internally, by *AnTLR*, in the process of generating the resulting parsers, assuming an assisting role in the analysis and extraction of data from the implemented grammars. A few classes from the Java Runtime library are imported by the classes responsible for the analysis and compilation of the grammars, especially the exception and input handling classes.

The Java Runtime library is composed by the following classes:

IntStream An interface that defines the required features to implement a stream of integers. This kind of streams are used by the parser when attempting to match token types.

CharStream This interface inherits *IntStream* properties, but extends them for characters. A char stream is the source of characters for an AnTLR lexer.

ANTLRStringStream Implements a char stream that gets all the data directly from an array. It inherits all the *CharStream* properties, and consequently, all the properties of *IntStream* as well. It is mainly used during the lexical analysis, and it is responsible for storing the input text in array context. The lexer works over an object of this type to obtain and recognise the words found in the text, and to generate the corresponding tokens — accordingly to the grammar that defines the language in question — required for the parsing process.

ANTLRFileStream Similar to *AnTLRStringStream*. However, it loads the input text from a file stream.

ANTLRReaderStream This class is also similar to *AnTLRStringStream*, but it loads the input text from any buffer, such as the *stdin* or a binary file.

ANTLRInputStream It inherits the properties of *ANTLRReaderStream* and has an identical functioning. However, it contemplates the encoding of the input stream.

Token Interface responsible for defining the behaviour of a token.

TokenSource An interface that defines the properties a token source must have. The *Lexer* implements all the properties of *TokenSource*, since it draws symbols, from an input stream, which abide with a specific language.

TokenStream An interface that defines the properties of a stream of tokens. A class that implements *TokenStream* grabs the tokens from a token source, usually a lexer, and feeds them to the parser.

CommonToken Class that stores all the information about a specific token. It implements the *Token* interface.

ClassicToken Similar in functioning to the *CommonToken* class, though it was created to serve other purposes. This kind of tokens are needed by imaginary tree nodes. The main difference between *CommonToken* and *ClassicToken* is that, the first uses indexes into a char stream to identify the token text, instead of using a string, like the latter.

CommonTokenStream This class implements a *TokenStream*. The stream filters the tokens by channels.

CharStreamState This class works as a state of the input stream, making possible

to return a previous state after scanning ahead. It is especially useful when traversing over a DFA, since it makes possible to rewind the state after jumping for a state ahead.

RuleReturnScope Class that defines which type of information rules can return, besides user defined types.

ParserRuleReturnScope This class is needed by the rules that synthesize more than one attribute. Since more than one attribute is returned, the values of these must be stored in an object. The object created for this purpose inherits all the properties of *ParserRuleReturnScope*. This class also defines the minimum properties that are always available, even when the user defines the return values.

Lexer A lexer is basically a token source. It works over an input text (*CharStream*), from where it draws symbols that correlate with a specific language, defined by a particular grammar. These symbols are then stored in a *TokenStream* and passed to the parser. The lexers generated by AnTLR inherit all the properties of this class.

BaseRecognizer It is one of the most important classes in the Runtime library. Most of the parsing mechanisms are implemented in this class, such as backtracking, error recovery and memoization. Both *Lexer* and *Parser* extend this class since, *BaseRecognizer* is a generic recogniser that implements most of the support code that assists in the parsing process.

Parser The class *Parser* defines all the properties that a parser generated by AnTLR should inherit. It grabs tokens provided by a *TokenStream* and checks if the input belongs to a previously defined language.

DFA Class that implements all the support code for the construction of DFAs. The DFA is implemented as a set of transition tables. Any DFA has an associated recogniser to enable backtracking. Considering a specific input stream, this class predicts the next stage to succeed using the DFA.

RecognizerSharedState This class contains all the fields required by an abstract recogniser to match the input and recover from errors. All the recognisers (*Lexer* and *Parser*) handle an object of this nature.

BitSet A sort of stack used for automatic error recovery.

EarlyExitException This class handles exceptions that occur when the recogniser does not match anything successfully for a one or more (+) loop.

MismatchedSetException This class handles exceptions that occur when a match that succeeds before fails later.

MismatchedNotSetException Similar to the latter class, however it does not occur the first match.

MismatchedRangeException It handles exceptions that occur when an attempt to match fails between boundaries.

MismatchedTokenException It handles exceptions that occur when fails an attempt to match a token, a character or a tree node.

MissingTokenException This class handles exceptions that occur when an expected token or tree node is not matched next.

NoViableAltException This class handles exceptions thrown when no other viable alternative is suitable to follow.

MismatchedTreeNodeException This class handles exceptions thrown when fails an attempt to match a tree node.

UnwantedTokenException It handles exceptions thrown when a token is found, while other token is expected by the parser.

FailedPredicateException This class handles exceptions that occur when it fails the validation of a rule predicate.

RecognitionException Class from which most of the exception classes depend on. It defines the properties common to most of the exceptions, which allow them to track the errors occurred and which was the expected input, if any.

There are some other classes, besides those explained in this section, but they are not required for the proper functioning of the parser. These classes include some ancillary data structures, debugging mechanisms and parsing tree construction.

4.4 Code Generator

The Code Generator is where the templates, that assist on the generation of the parsers, are located. This AnTLR component is mostly constituted with code developed by contributors that made possible for the tool to support the generation of

parsers in several programming languages. The Code Generator templates for each programming language available with AnTLR and the corresponding target classes for data transformation are included in this component.

The Code Generator is directly connected with StringTemplate in the generation process, as it requires the template engine to interpret the templates underneath the generator. These templates, known as template groups, contain the necessary code to write a parser or a lexer in any of the programming languages available.

A template group is structured in smaller templates, each one being similar to a function. AnTLR invokes the templates when attempting to generate the code. The called template is then interpreted by StringTemplate that, according to the input parameters of the template, extracts the data from the classes constructed at the analysis phase and builds an output fragment applying the template constructs.

Eventually, some of the data may require some mutations, depending on the target language. Take as example the fact that *Perl* representation of *unicode* characters differs from the *Java*. It would be incorrect to pass the data in its original form to the Perl Code Generator. The solution lies in creating a class (named a target class) for each programming language, to handle these isolated cases.

These target classes are called during the analysis process to adapt the input data to a particular language needs and then return it, to be stored in the proper data structures, for later use by the templates.

To have a general idea of how these templates work, the example 4.3 analysed in Section 4.2.2 will be reviewed but now with focus on its behaviour and results instead of the StringTemplate constructs.

As mentioned in Section 4.2.2, the example template shown (Figure 4.4) is responsible for generating the code corresponding to one or more predicates surrounded by the symbol "+". Notice that this block appears surrounded by curly brackets inside a *try* block in a production method.

At the top it is found the name of the template along with the input parameters, just like a common function. These parameters contain all the data required to generate the code fragment for which the template is responsible. However, there are some exceptions such as the global attributes that are accessible by all the templates included in the template group.

```
positiveClosureBlock(alts,decls,decision,enclosingBlockLevel,
   blockLevel,decisionNumber,maxK,maxAlt,description) ::= <<
// <fileName>:<description>
int cnt<decisionNumber> = 0;
<decls>
<@preloop()>

loop<decisionNumber>:
do {
    int alt<decisionNumber> = <maxAlt>;
    <@predecision()>
    <decision>
    <@postdecision()>

    switch (alt<decisionNumber>) {
        <alts:altSwitchCase()>

        default :
           if ( cnt<decisionNumber> >= 1 )
                        break loop<decisionNumber>;
           <ruleBacktrackFailure()>
           EarlyExitException eee =
             new EarlyExitException(<decisionNumber>, input);
           <@earlyExitException()>
           throw eee;
    }
    cnt<decisionNumber>++;
} while (true);

<@postloop()>
>>
```

Figure 4.4: Template to generate cases of one or more alternative (+) in *Java*.

The first line of the template outputs a comment in *Java*, with the path for the compiled grammar and the description of the production in question, separated by a colon. The following lines until the loop outputs the variable responsible for controlling the success of the predicates covered by the one or more symbol, and some other declarations, if any.

Remember that, as explained in Section 4.2.2, the templates and attributes are surrounded by chevrons, so, anything else is sent to the output as it is found in the template. Therefore, most of the instructions of the loop observed in the example would appear in the output file as they are seen in the example, hence requiring no explanation.

The label "loop" is followed by an attribute "decisionNumber". This attribute stores an integer which is incremented each time a decision must be made by the parser or lexer. Decisions include the prediction of alternatives or loop control ("+" and "*").

The first instruction of the loop defines the number of alternatives that can be chosen by the parser or lexer. The "alt" variable is concatenated with the decision number and the number of alternatives is assigned to it. Consider the "sacos" production, from the *Lavanda* grammar found as Appendix (B.2), for a practical example.

sacos: saco+;

Since the "+" symbol only covers "saco", the maximum number of alternatives would be *1*, plus the stop alternative, when the input no longer satisfies the production. Therefore, the value of the attribute "maxAlt" would be *2*. The resulting code is shown in the following figure.

```
int alt1 = 2;
```

Figure 4.5: Resulting value for the "maxAlt" attribute.

The "predecision" and "prodecision" templates, and the "decision" attribute generate the conditions and instructions required to select the next alternative to follow. To generate these instructions, the templates must obtain some data stored during the compilation of the grammar (analysis phase).

```
int LA1_0 = input.LA(1);

if ( (LA1_0 == NUM) ) {
    alt1 = 1;
}
```

Figure 4.6: Resulting code for the decisions templates.

In "sacos" case, it is necessary to check if the next token satisfies the production "saco", to determine the next state. In case of success, it jumps to "saco" otherwise,

it no longer satisfies "sacos" and it must jump to the follow of "sacos". To compare the next token with the first token that satisfies "saco" production ("NUM"), the Code Generator must output the match instruction. However, to accomplish this task, the Code Generator must know which token should output. This information is, again, retrieved from the analysis classes of AnTLR. The corresponding output code for this portion of the template can be seen in Figure 4.6.

The last important block in the example template is the "switch" statement. To create an undetermined number of similar blocks is a task that cannot be planned and performed by anticipation, since it is impossible to predict how many blocks are necessary to create. A task of this nature should be automatised and abide by a template. This is exactly what happens in the example template, when outputting the "switch" alternatives.

For each value found in the attribute "alts", a template "altSwitchCase" is called to output a "case" statement. This template can be observed in Figure 4.7. The output of "altSwitchCase" for the valid alternative in "sacos" production (only one, "saco") is shown in Figure 4.8.

```
1 altSwitchCase() ::= <<
2 case <i> :
3     <@prealt()>
4     <it>
5     break;<\n>
6 >>
```

Figure 4.7: "altSwitchCase" template.

```
1  case 1 :
2      // [grammar_path]:[line]:[col]: saco[inEnv, clientIds,
           bagIds]
3      {
4      pushFollow(FOLLOW_saco_in_sacos70);
5      saco2 = saco(inEnv, clientIds, bagIds);
6      state._fsp--;
7      retval.nSacos++; // semantic action
8      retval.outEnv = saco2; // semantic action
9      }
10     break;
```

Figure 4.8: "altSwitchCase" output for the "sacos" production.

The remaining code would be almost the same as in the template, as observed in Figure 4.9.

```
1  do {
2      ...
3        default :
4           if ( cnt1 >= 1 ) break loop1;
5                EarlyExitException eee =
6                     new EarlyExitException(1, input);
7                throw eee;
8        }
9        cnt1++;
10 } while (true);
```

Figure 4.9: Remaining code generated by "positiveClosureBlock" template.

4.5 Structure of the Generated Parser

With the help of the template engine *StringTemplates*, AnTLR generates a parser and a lexical analyser for a target language, if the back-end for the same language is available. It also generates an additional file with the mapping of the tokens and their respective type. The *Java* generated code for the lexical analyser and parser will be analysed in the next sections. Both method of implementing the *Lavanda* grammar in AnTLR metalanguage, using *global variables* or *attributes*, can be found in this document as Appendix (B.1 and B.2).

4.5.1 Lexical Analyser

The generated code of the lexical analysers is exactly the same for the two implemented grammar approaches, due to both dealing with the same terminal symbols.

Considering that AnTLR generates recursive-descent parsers, the lexical analyser also shares this property. So, for each terminal symbol a specific method is created with the purpose of attempting matching with the input to verify if it can be identified as a known token.

The generated code of the lexical analyser is human-readable and straightforward to read.

Considering the terminal symbol *"ID"* as an example,

ID: LETTER+;

the resulting method for this symbol would be something like the following:

```
public final void mID() throws RecognitionException {
  try {
    ...
    int cnt1=0;
    loop1:
    do {
      int alt1=2;
      int LA1_0 = input.LA(1);

      if ( ((LA1_0>='A' && LA1_0<='Z')
                        ||
            (LA1_0>='a' && LA1_0<='z')) ) {
         alt1=1;
      }

      switch (alt1) {
         case 1 :
         {
          mLETTER();
         }
         break;

         default :
          if ( cnt1 >= 1 ) break loop1;
          EarlyExitException eee =
                  new EarlyExitException(1, input);
          throw eee;
      }
      cnt1++;
    } while (true);
  }
  ...
}
```

Notice that the name on the prototype of the method is a concatenation of the char "m", from "match", with the name of the symbol. In this example it is called "mID".

Given that "ID" is equivalent to one or more "LETTER", the subrule "LETTER" is enforced to be satisfied at least once. To assure this constraint, a loop counter variable "cnt1" (the number following the name of the variable varies) was declared. At the line *22*, if "cnt1" has a value equal or bigger than 1 the loop breaks as normal, otherwise an exception is thrown and the production is skipped. If before a zero or more situation, this counter is omitted since it is not required to satisfy the subrule.

The variable "alt1" provides the lexical analyser the ability of predicting the next alternative to jump to. An exception is thrown in situations where is enforced to predict the next alternative to succeed, yet the lookahead is not consistent with any of the alternatives available.

The lookahead symbol is stored into the variable "LA1_0" and it can be attain through the call of the *LA()* method, originally associated with the "input" object *(CharStream)*. Therefore, "input.LA(1)" will look at the next symbol to be consumed. The result of this operation will affect the decision of the next jump, as shown in the "if" statement at line *10*. If the next symbol belongs to the alphabet, or "LETTER" to be more precisely, the first alternative is chosen as next jump. The "mLETTER" method is called to consume a symbol that satisfies "LETTER", using the "input.consume" method.

In a brief clarification, the running process of this method would start with the loop. After, it would look for the next symbol to consume, this would allow the lexer to predict the next jump relying on the lookahead. After consuming a symbol, another iteration of the loop would start. This process would continue until the loop breaks normally or with an exception.

This method provides an intelligible idea of how the methods of the lexical analyser are generated, however it is composed by other important components, such as the method "mTokens" that predicts the next symbol to be consumed, assisted by a *DFA*[2], followed by the call of the appropriate method.

[2]Deterministic Finite Automaton

4.5.2 Generated Parser

Unlike the lexical analyser, the two approaches generate different parsers, though they still share some similarities.

4.5.2.1 Approach Based on Global Variables

In this approach, global variables are used to store the data evaluated in the semantic actions defined in each node of the parser tree. The data structures that hold the semantic information are available and reachable by all nodes or methods of the parser. Therefore, the data structures and scalar variables are common among the methods, excluding the need of setting input parameters when calling them. This approach forces the user to change all the occurrences of any of these variables in case its declarations are modified (identifier, type or even initialised value).

For a better understanding of the parser behaviour, the generated parser for the *Lavanda* grammar (B.3.1 and B.1 respectively) will be analysed with some particular detail.

This parser can be seen as a joint of five different blocks. The header, the tokens, the variables, the productions (the nodes of the parser tree) and lastly, the error recovery block.

```
1 import java.util.HashMap;
2 import java.util.LinkedList;
```

Figure 4.10: Parser includes.

It is in the header block that all the libraries required by the parser are imported. The libraries imported at the top of the parser, are the same defined in the "header" block of the AnTLR grammar. The remaining libraries are incorporated automatically by the AnTLR engine, to guarantee the normal functioning of the parser. It is, at this point, that the Runtime library is imported.

In the tokens block, a list of tokens is declared, containing all the tokens found in the previously generated file, referred at the beginning of this section, along with a few control tokens. All the tokens present in the tokens file are also instantiated

and assigned a token type. With a closer look over the example 4.11, it is easy to notice that, the token type is equivalent to the index where the token is positioned in the earlier declared list.

Notice that, the token representing the terminal symbol "," assumes an identifier composed by the character "T" and its token type *12*, resulting in "T__12". This happens to prevent variables from having forbidden characters (commonly found in terminal symbols) in its name. Also, this can be verified in the tokens file as seen in the example 4.12.

```
1 public static final String[] tokenNames = new String[] {
2     "<invalid>", "<EOR>", "<DOWN>", "<UP>", "DATA", "ID", "
         NUM",
3     "LETTER", "DIGIT", "WS", "'('", "')'", "','", "'-'", "'corpo
         '",
4     "'casa'", "'br'", "'cor'", "'alg'", "'la'", "'fib'"
5 };
6
7 public static final int LETTER = 7;
8 public static final int T__12 = 12;
9 ...
```

Figure 4.11: Tokens instantiation block.

```
'la'=19
'br'=16
','=12
'('=10
'casa'=15
')'=11
'cor'=17
'corpo'=14
'alg'=18
'fib'=20
'-'=13
```

Figure 4.12: Mapping of the tokens.

The tokens block is followed by the class constructors, some auxiliary methods and the variables block, where are defined and initialised the instance variables of the class. These variables are exactly the same variables defined in the "members" block

on the AnTLR grammar. This can be verified by comparing the "members" block of the *Lavanda* grammar implemented in AnTLR metalanguage (Appendix B.2), with the example 4.13, taken from the generated parser for the same language (Appendix B.3.2).

These variables are global in the parser class and can be accessed or mutated by all the parser tree nodes. This has a few effects on the implementation of the methods. No input or output parameters need to be declared most of the times, since most of the data structures are already instantiated in this block, and it also reduces the number of variables in scope of the method, including auxiliary variables. On the other hand, it leads to declaration dependencies, as mentioned before in this section. Any amend on the declaration of these variables would evince a change on the behaviour of all the methods that use them and force a revision of an high portion of the code, depending on the number of methods directly affected. This happens because global variables usually are in scope of all the code instead of being limited to a particular block, such as methods, loops or statements.

```
1 HashMap inTable = new HashMap();
2 HashMap env     = new HashMap();
3 int nSacos      = 0;
4 int custoTotal  = 0;
5 int nLotes      = 0;
6 LinkedList clientIds = new LinkedList();
7 LinkedList bagIds    = new LinkedList();
```

Figure 4.13: Global variables defined in the members block.

The method of the parser can be found in the productions block. Each of these methods is equivalent to the rule of the grammar, implemented in AnTLR, which has the same name as the method identifier. These methods are the backbone of the parser and, as expected, the most extensive block, as well. To better understand the working process and roles of these methods, two methods, "sacos" and "saco" were chosen to be analysed. The choice fell upon these two methods since they are the appropriate example to identify the most significant differences between the two approaches, *global variables* and *attributes*.

The methods following this approach have a few structural similarities with the lexical analyser methods. These similarities can be drawn by comparing the example

given in the Section 4.5.1 and the method "sacos" found in the generated parser for the *Lavanda* language, using global variables (Appendix B.1). After a careful analysis over the "sacos" method, along with some knowledge of the code generation process of AnTLR, it is reasonable to state that this method follows a straightforward implementation, since the Code Generator engine does not need to consider almost any special case to generate the method. This happens, essentially, because of the use of global variables. The previous declaration of most of the variables at the members block, reduces drastically the number of local variables in the methods. Therefore, new variables are declared only when confronted with special situations, such as attributes or terminal symbols with data associated. Since global variables identifiers are preserved when in scope of a code block, all the semantic actions within that same block can use them directly instead of creating auxiliary variables. Notice, this does not applies for every method in the parser, since a few productions use synthesized attributes.

Another visible effect of global variables in the "sacos" method, is the absence of input and output parameters provided to or returned by it, as seen in the Figure 4.14. As in the other cases, this happens, mainly because the variables are available and reachable through the whole parser, granting the direct use of these variables in any computation of a specific method, without any scope restriction, the need to receive input data as parameters or to return the value of local variables.

```
public final void sacos() throws RecognitionException
```

Figure 4.14: Prototype of the "sacos" method.

The "sacos" method is very similar to the lexer method analysed in the Section 4.5.1, since the productions of the *Lavanda* grammar, implemented in AnTLR, that originated these methods also share very similarities. This can be easily verified in figures 4.15 and 4.16.

```
ID: (LETTER)+ ;
```

Figure 4.15: *ID* rule

```
sacos: (saco)+ ;
```

Figure 4.16: *sacos* rule

The local variables "cntl1", "alt1" and "LA1_0" share the same purpose of the variables instantiated in the "mID" method found on the lexical analyser. The "cntl1" variable assures the one or more ("+") precondition is satisfied, the "alt1" determines the next jump, assuring that a decision is made by the parser, otherwise the matching process fails, and the "LA1_0" holds the next token to be consumed, contemplating one symbol ahead (lookahead equals to *1*).

The main distinction between the parser and the lexer methods is, the fact that, unlike the lexer, the parser enforces backtracking when it fails to match a specific symbol. The backtracking is performed by a base recogniser object. The instructions responsible for the control and recover of errors are shown in the example 4.17.

```
pushFollow(FOLLOW_saco_in_sacos60);
saco();
state._fsp-;
```

Figure 4.17: Error control and recovery.

The method "pushFollow" is responsible for pushing, into the state stack, the bit set relative to the next rule invocation. This stack keeps track of every jump conducted by the generated parser, extending the possibility to recover from errors, by returning early states stored in it. The "_fsp" variable is useful to keep track of how many tokens have been already consumed by the parser. In this case, no token is consumed by the parser (a rule is called instead), however, the "pushFollow" method still, automatically, increments the counter. This is why the variable decrements after the rule call.

The "saco" method is responsible for reproducing the behaviour of the production with the same name in the AnTLR implementation of the *Lavanda* grammar, shown in the Figure 4.18.

The production consists of four terminal symbols and one nonterminal. Two of the terminal symbols, "NUM" and "ID", have data associated, so, some extra computations are required, in the respective method, to get this data. Besides the

computations, it is also required to declare a variable for each symbol, to store the data, as shown in example 4.19.

```
saco : NUM ID '(' lotes ')' ;
```

Figure 4.18: *saco* rule

```
1 Token NUM1 = null;
2 Token ID2 = null;
```

Figure 4.19: Declaration of auxiliary variables.

Accordingly to the rule "saco" found in the *Lavanda* grammar, "NUM" is the first symbol being matched. Observing the *try* block in the Figure 4.20, it can be seen that this is exactly what happens in the method as well. The method "match" grabs the next token to be consumed on the stream "input" and compares it with the type of token expected, "NUM" in this case. Also, it recovers, automatically, from errors. This is the reason why a bit set ("FOLLOW_NUM_in_saco79") is set as parameter. Unlike in "sacos" method, the semantic actions suffer a few modifications. Knowing that the "NUM" symbol has some data associated, an extra computation must be added when generating the parser, to assure the call of the token method that delivers the data.

The same process is used for the token "ID", as seen from line *12* to *15* of Figure 4.20. The curved brackets are matched, at lines *18* and *25*, like the other terminal symbols, but the returned tokens are ignored since they do not have any associated semantic actions. The remaining instructions have already been covered previously in the lexer or in "sacos" method.

Lastly, the error control and recovery block, where the bit sets are declared for each rule invocation or symbol matching. These bit sets are pushed into a stack to control the state of the parsing process and to allow backtracking, by recovering previous states when a symbol fails to match.

```
...
try {
  ...
  {
  NUM1=(Token)match(input,NUM,FOLLOW_NUM_in_saco79);
  if(bagIds.contains(Integer.parseInt(
             (NUM1!=null?NUM1.getText():null))))
      ...
  bagIds.add(Integer.parseInt(
             (NUM1!=null?NUM1.getText():null)));

  ID2=(Token)match(input,ID,FOLLOW_ID_in_saco88);
  if(clientIds.contains((ID2!=null?ID2.getText():null)))
      ...
  clientIds.add((ID2!=null?ID2.getText():null));

  nLotes = 0; custoTotal = 0;
  match(input,10,FOLLOW_10_in_saco97);

  pushFollow(FOLLOW_lotes_in_saco99);
  lotes();

  state._fsp--;

  match(input,11,FOLLOW_11_in_saco101);
  System.out.print("Numero␣de␣lotes␣para␣o␣ID␣" +
             (ID2!=null?ID2.getText():null) + ":␣" + nLotes);
  System.out.println("␣Custo:␣" + custoTotal);
  }
}
...
```

Figure 4.20: Matching block of *saco* method.

```
public static final BitSet FOLLOW_saco_in_sacos60 = new
   BitSet(new long[]{0x0000000000000042L});
public static final BitSet FOLLOW_NUM_in_saco79 = new BitSet(
   new long[]{0x0000000000000020L});
public static final BitSet FOLLOW_ID_in_saco88 = new BitSet(
   new long[]{0x0000000000000400L});
public static final BitSet FOLLOW_10_in_saco97 = new BitSet(
   new long[]{0x000000000000C000L});
public static final BitSet FOLLOW_lotes_in_saco99 = new
   BitSet(new long[]{0x0000000000000800L});
public static final BitSet FOLLOW_11_in_saco101 = new BitSet(
   new long[]{0x0000000000000002L});
```

Figure 4.21: Example of bit set instantiation.

4.5.2.2 Approach Based on Attributes

This approach consists in associating variables to the nodes of the parse tree, to provide them all the semantic information required for the semantic actions or computations. This method eliminates the dependencies between variable identifiers and restraints the access to the data structures used to store the semantic information. By eliminating most or all, since it is still possible to declare global variables, the variable dependencies, it is easier to deal with maintenance situations or code changes. Changing an attribute would only affect the node that uses it, unlike in the other approach, where it would be necessary to amend all the occurrences of the target variable.

The main difference between the two approaches, in the implementation of the parser, is the prototype of the methods. Methods using global data for computations or storing operations do not require input parameters or to return any data, respectively. On the other hand, parsers using attributes require its methods to specify the input parameters (inherited attributes), when needed, and the return data type (synthesized attributes) in theirs prototypes.

To better understand the differences between the two generated parsers, it will be undertaken a brief analysis over the generated parser for the *Lavanda* language, defined by an attribute grammar (Appendix B.2).

The *Lavanda* parser, based on an attribute grammar, is analogous to the parser generated for the global variables approach, and both can be divided in five blocks. These blocks are the same for both parsers and also share the same purposes. At a first glance, a user may assume that, since this approach uses attributes in substitution of the global variables, the latter is not needed, so the "members" block could be discarded. However, this is not necessarily true, having in account that global variables can still be instantiated in this block, as verified in the variables block of the parser based on attributes (Figure 4.22). The same happens in the opposite way, where attributes were used in the global variables approach, as seen previously in Section 4.5.2.1.

```
HashMap inTable = new HashMap();
```

Figure 4.22: Variables block of the attribute approach.

The code of the blocks is identical, exception made for the variables block, since the use of attributes reduces the number of global variables needed, and the productions block. Is in this last block that the most significant differences between the two approaches can be found. The code repercussions of changing the variables block are visible in the methods "saco" and "sacos" of the parser generated for the *Lavanda* language (Appendix B.3.2).

Analysing the AnTLR implementation of the rules that produce these two methods (Figure 4.23), it can be drawn that the "sacos" production deals with two synthesized attributes, "nSacos" and "outEnv", and "saco" deals with one synthesized attributes, "outEnv", and three inherited, "inEnv", "clientIds" and "bagIds".

```
sacos returns [ int nSacos = 0, HashMap outEnv = new HashMap
   () ]
@init { ... }
        : ( saco[inEnv, clientIds, bagIds] { ... } )+
        ;

saco [ HashMap inEnv, LinkedList clientIds, LinkedList bagIds
    ] returns [ HashMap outEnv ]
        :  NUM { ... }
           ID  { ... }
           '(' lotes[inEnv] ')' { ... }
        ;
```

Figure 4.23: "saco" and "sacos" productions.

In the parser implementation, the synthesized attributes turn into variables with the end of returning semantic information from the methods, and the inherited attributes are equivalent to the input parameters of the methods. Since the "saco" method has only one synthesized attribute, it is predictable that it will return data of a particular type, the same as the attribute. However, this raises a question. What would happen if the method returns more than one attribute? This is "sacos" case, where the grammar, which defines the language, specifies that it must return two distinct types of data, a scalar integer and a "HashMap" object. Due to programming languages constraints, this cannot be achieved by simply returning the two variables. The solution lies on defining a nested class within the parser, containing all the variables needed to store the semantic information associated to this node. This way, the "sacos" method will return an instance of the newly created

nested class, as seen in the Figure 4.24.

To return this instance, a new object must be created and its type must match the nested class identifier. This object is generated by the Code Generator under the name "retval". It is possible to verify this property by inspecting the methods "sacos", "lotes" and "lote". Also, it has another particular characteristic, it extends the class "ParserRuleReturnScope" from the Runtime library, which provides access to some methods and a few properties. Such as, the *start* and the *stop*, to keep track of the consumed tokens that correlate with the target production.

```
1 public static class sacos_return extends
     ParserRuleReturnScope {
2     public int nSacos = 0;
3     public HashMap outEnv = new HashMap();
4 };
5
6 public final Lavanda2Parser.sacos_return sacos() throws
     RecognitionException {}
```

Figure 4.24: Class "sacos_return" and the prototype of "sacos" method.

```
1 retval.start = input.LT(1);
2 ...
3 try {
4         ...
5         do {
6                 ...
7         } while (true);
8         ...
9         retval.stop = input.LT(-1);
10 }
11 catch (RecognitionException re) {
12         ...
13 }
14 finally {}
15 return retval;
```

Figure 4.25: *start* and *stop* consuming tokens.

Notice that, most of the initial instructions of the method "sacos" are, on the opposite of what occurs in the global variables approach, initialisations of variables.

These variables are either to store the result of the computations, the data returned from the invoked methods or the initialisations defined in the "init" block of the production in the AnTLR implementation of the grammar. Since the method "saco" returns only a "HashMap" object, it is not required the creation of a new nested class. Therefore, the variable "saco2", in the method "sacos", it is instantiated as a "HashMap", as seen in Figure 4.26. The same does not happen in the method "saco", because "lotes" has three synthesized attributes and, obviously, it requires a nested class to be created. Consequently, the variable "lotes5" will be an instance of this nested class, "lotes_return". This can be verified in Figure 4.27.

Regarding the "try" block, only a few instructions change from the first approach to the attributes approach. These changes are verified, mostly, on the instructions that compute the method output objects or in the method invocations that have any kind of attribute associated.

```
// variable declaration
HashMap saco2 = null;

// method call
saco2 = saco(inEnv, clientIds, bagIds);

// method prototype
public final HashMap saco(HashMap inEnv, LinkedList clientIds
    , LinkedList bagIds);
```

Figure 4.26: Call of "saco" method.

```
// variable declaration
Lavanda2Parser.lotes_return lotes5 = null;

// method call
lotes5 = lotes(inEnv);

// method prototype
public final Lavanda2Parser.lotes_return lotes(HashMap inEnv)
    ;
```

Figure 4.27: Call of "lotes" method.

In the "sacos" method, where usually would appear a direct assignment between the

object returned by a method and the result of the semantic actions, is now replaced by a new assignment that aims the storage of the resulting values into the "retval" object, as observed in the Figure 4.28.

```
retval.nSacos++;
retval.outEnv = saco2;
```

Figure 4.28: Computations using "retval".

Something similar occurs in method "saco", however, the instance of "lotes_return", "lotes5", is not the returning object of the method. The variable "lotes5" is used to store the semantic information returned by the method "lotes", invoked in the "try" block of the method "saco" (Figure 4.29). The main difference between both variables is that "retval" is used only in mutate operations and "lotes5" only in access operations.

```
System.out.println("␣Custo:␣" + (lotes5 != null?lotes5.
   custoTotal:0));
outEnv = (lotes5 != null?lotes5.outEnv:null);
```

Figure 4.29: Computations using "lotes5".

Methods that have attributes associated present some changes in their invocations. If the node has synthesized attributes associated, the result of the method call is assigned to a predefined variable. If the node inherits attributes, the call is made with the proper parameters. The inherited attributes are defined in the prototype of the called method. An example of the two adjustments can be observed in the method "sacos", at "saco" call, in Figure 4.26.

The remaining code is pretty much the same, with a few minor adjustments. All these changes make this approach harder to understand and more extensive, but easier to maintain.

Chapter 5

The Perl Code Generator

With the reverse engineering process over AnTLR finished, it is possible to start idealising and implementing the Perl Code Generator, having in mind all the information gathered during the process. In this chapter it is described how this generator is implemented, which are their components and its working processes. In the end, it is performed an evaluation of the tool, considering all the important aspects.

5.1 Expected Perl Program

Before describing and analysing the *Perl* code expected to be generated by the Code Generator, it is recommended to be aware of what changed in the specification of the grammar, to be able to predict how these changes would affect the resulting code. The changes made in the grammar specification were all syntactical due to some *Perl* requirements and one conflict with AnTLR metalanguage syntax.

AnTLR uses the dollar sign ("$") to identify attributes, inherited and synthesized, and intermediate production results used in semantic calculations, while *Perl* introduces it as a precedence for most variable identifications. This raises a conflict between the two languages when writing the semantic rules associated with the productions of the grammar.

These conflicts can occur at the "members" block of the grammar; the "init" block and semantic rules associated with the productions; the attributes sent as parameters in a predicate, etc.

Since AnTLR metalanguage cannot be changed without authors consent, the changes

must be verified in the *Perl* code spread throughout the grammar. As usually in this cases, the language provides a way of escaping the restricted symbols (Figure 5.1). Therefore, every occurrence of the dollar sign in *Perl* context, when in scope of attributes using the same symbol, must be escaped to avoid colliding with AnTLR syntax.

Also, in the declaration of the inherited and synthesized attributes associated with a production, the type of the attributes were dropped since *Perl* is a weakly typed language. Nevertheless, it is required the inclusion of the symbol that determines if the attribute is or not in scalar context ("$", "@", "%"). This way it is possible to declare variables in scalar context.

```
1  sacos returns [ $nSacos = 0, $outEnv = {} ]
2  @init {
3  my \$clientIds = {};
4  my \$bagIds    = {};
5  my \$inEnv     = {};
6  }
7       : ( saco[\$inEnv, \$clientIds, \$bagIds] {
8                       $nSacos++;
9                       $outEnv = $saco.outEnv;
10           } )+
11      ;
```

Figure 5.1: Production "sacos" to be generated in *Perl*, in AnTLR notation.

This was the main reason why the *Perl* variables included in the semantic rules were escaped, instead of removing the precedence symbol and let the Code Generator output the dollar sign by default into the generated Language Processors. By enabling the possibility of the user define the nature of the variable, it is possible to program *Perl* with no variable restrictions of whatsoever, as seen below.

```
production [ %hash, @array, $invar ] returns [ $outvar ]
```

These changes came with a cost, the decreasing of the readability of the grammar specification when compared with the *Java* solution. Moreover, these changes make it harder, for a Perl programmer, to write *Perl* code while implementing a grammar

in AnTLR. Therefore, it has been implemented a line command program to automatically escape the conflicting symbols present in a given grammar and then execute AnTLR to compile it. This program is available along with the Runtime library.

```
1 WS    : ('\r'|'\n'|' '|'\t')+ {$self->skip();} ;
```

Figure 5.2: Invocation of method *skip* in *Perl*, in AnTLR notation.

There is also a minor change in the methods call. To address to an object in *Perl* inside a subroutine (method) of its own, it is required to identify it as "$self", so, if a method associated with a class is called by an instance of this class, it should be preceded by "$self", as seen in Figure 5.2.

After these changes, to use the Perl Code Generator instead of the one defined by default, it is imperative to notify *AnTLR* of this intention. The change is reflected on the header of the grammar implementation. For instance, the header of the *Lavanda* grammar would become the following:

```
grammar Lavanda;
options {
  language = Perl5;
}
```

5.1.1 Generated Lexer

It will not be given to much emphasis to the generated lexical analyser since it shares various similarities with the parser. Therefore, they will be addressed in the parser analysis.

The better way of planing the generation of the code for the lexer and the parser is to translate the code generated in *Java* and then analyse it to extract the best of it and replace the fragments that need improvements, to obtain the expected code for *Perl* solution. After achieving the mapping between what can be specified in the grammar and the respective expected code, it is only a matter of implement the Code Generator in accordance with it.

```
1 ID     : LETTER+;
```

Figure 5.3: Production "ID" extracted from *Lavanda* grammar, in AnTLR notation.

For a better analysis of the expected code for the lexer, it was chosen the production "ID" extracted from *Lavanda* grammar, observed in Figure 5.3. This way it is possible to compare the *Java* lexer found in Section 4.5.1 against the expected *Perl* one, seen in Figure 5.4.

By comparing the implementation in *Java* against the expected code for the *Perl* solution, it is verifiable that they are similar in structure, there are not much changes from one solution to the other besides syntax. Thanks to the *Perl modules* available in CPAN, it was possible to keep the same structure of the *Java* solution and most important, the same behaviour, including the "try" block, the exceptions handling and the method invocations.

The main differences in the generation of method "m_ID" are the use of the object "$self" each time a method associated with the lexer is called, and the creation of new objects as seen in the creation of an instance of *ANTLR::Runtime::EarlyExitException* from line *25* to *29*. This happens because *Perl* objects are treated as hashes internally, thus why each value, the arguments of the method, is associated with a corresponding key, the instance variable of the class.

Notice that the object is grabbed as an argument in the first line of the subroutine, and stored in "$self" variable (line *2*). This "$self" is analogous in use to "this" in *Java*. Also, the variables declared in the *Java* method lose the type identifier and are now preceded by the keyword "my", used to declare variables in *Perl*.

The remaining differences, not covered by this method, will be described during the parser analysis in next section.

5.1.2 Generated Parser

The strategy used to analyse the lexer is the same used to analyse the expected parser in *Perl*, by dividing it in the same blocks as it was *Java* generated parser in Section 4.5.2. This makes possible to analyse one block at a time and identify the major differences between both language solutions. The order of the performed analysis is the same verified in Section 4.5.2, though, the order of appearance in the

```
sub m_ID {
  my ($self) = @_;
  try {
    my $_type = ID;
    my $_channel = $self->DEFAULT_TOKEN_CHANNEL;
    {
     my $cnt1 = 0;
     LOOP1:
     while (1) {
        my $alt1 = 2;
        my $LA1_0 = $self->input->LA(1);

        if ( (($LA1_0 ge 'A' && $LA1_0 le 'Z')
                              ||
              ($LA1_0 ge 'a' && $LA1_0 le 'z')) ) {
                $alt1 = 1;
        }

        given ($alt1) {
            when (1) {
                { $self->m_LETTER(); }
            }
            default {
                last LOOP1 if ( $cnt1 >= 1 );
                my $eee =
                   ANTLR::Runtime::EarlyExitException->new({
                        decision_number => 1,
                        input => $self->input,
                   });
                $eee->throw();
            }
        }
        ++$cnt1;
     }
    }

    $self->state->type($_type);
    $self->state->channel($_channel);
  }
  finally {};
  return;
}
```

Figure 5.4: Production "ID" extracted from *Lavanda grammar*, expected in *Perl*.

code is different, since the error recovery block is located between the tokens and the global variables in the *Perl* generated parser.

```
use Moose;
use Data::Lock qw( dlock );
use Try::Tiny;
use Exception::Class;
use feature qw( switch );
use ANTLR::Runtime::BitSet;
use ANTLR::Runtime::MismatchedSetException;
...
```

Figure 5.5: Header of the generated parser in *Perl*.

The first block, the header (Figure 5.5), is slightly larger than the *Java* one, since the runtime library is not all imported into the parser, only the *Perl* runtime *modules* required by the parser to run as expected. Besides the runtime *modules*, some other *Perl modules* are included to deal with particular situations.

In the tokens block, the main adversity was to keep the variables immutable as in the *Java* solution ("final" keyword). As seen in Figure 5.6, the solution was to use the subroutine "dlock", which belongs to *Data::Lock*[1], to preserve the immutable property of the array of tokens, and the *Perl pragma* "constant", to cover the tokens declarations.

```
dlock ( my $token_names = [
    "<invalid>", "<EOR>", "<DOWN>", "<UP>", "DATA", "ID", ...
] );
use constant {
        LETTER => 7,
        T__12 => 12,
        T__20 => 20,
        WS => 9,
        ...
};
```

Figure 5.6: Tokens block of the generated parser in *Perl*.

The reason why it is used the *pragma* "constant" instead of "dlock" is to keep

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

the declaration of the variables as much clean and similar to the *Java* solution as possible. Variables within the *pragma* "constant" do not need to be preceded by any of the symbol $, @ or %.

The global variables block offers a sort of freedom not experienced in any other fragment of the code. As explained earlier in this document, this block corresponds to the "members" block in the grammar specification in AnTLR metalanguage, so it is a fragment provided by the user. The *Perl* solution uses *Moose*[2] to implement the parser in *OO* paradigm, later explained in this chapter. So, the user is able to write code in the block "members" of the grammar using *Moose* constructs, apart from the traditional object implementation in *Perl*.

The user is free to choose the method he has preference for, as long the use of the variables respects the constraints of the chosen method, since they have distinct syntax for objects representation. Both methods can be seen in Figure 5.7.

```
1 Without Moose
2 my $inTable = {};
3
4 Equivalent with Moose
5 has 'inTable' => (
6     is       => 'rw',
7     isa      => 'HashRef[Any]',
8     default  => sub{ {} }
9 );
```

Figure 5.7: Comparison between the two possible ways of declaring variables in the generated parser in *Perl*.

As seen in Section 5.1.1, the methods generated in *Perl* are very similar to the *Java* ones. However, there are some important differences that arise in the parser generation.

By looking to the prototype of the method "saco", in *Perl*, presented in Figure 5.8, it can be verified that it has no reference to the arguments and return type of the method (subroutine). This happens because, in *Perl*, a subroutine receives a list as parameter, containing all the arguments passed to the subroutine at the time of the call. The arguments are usually grabbed in the first line of a subroutine, including the object itself.

[2]http://search.cpan.org/ doy/Moose-1.08/lib/Moose.pm

Another distinction between both solutions is that, since *Moose* is in use, the access to the value of a variable is made through an accessor method. To assign a new value to a variable, the respective mutator method must be used (Figure 5.9).

```
1 ''saco'' prototype in Java
2 public final HashMap saco(HashMap inEnv, LinkedList clientIds
    , LinkedList bagIds) throws RecognitionException
3
4 Equivalent prototype in Perl
5 sub saco
```

Figure 5.8: Comparison between the prototypes of method "saco" in *Java* and *Perl*.

```
1 Java
2 retval.stop = input.LT(-1);
3
4 Equivalent in Perl using Moose
5 $retval->stop($self->input->LT(-1));
```

Figure 5.9: Comparison between assignments in *Java* and *Perl*.

Although *Moose* being in use, the object representation of the subclasses are made in traditional *Perl*, since it would be very hard to predict all the possible declarations and initialisation assignments, and represent them in *Moose* notation (Figure 5.10).

```
1 {
2     package sacos_return;
3     use Moose;
4
5     extends 'ANTLR::Runtime::ParserRuleReturnScope';
6
7     my $nSacos = 0;
8     my $outEnv = {};
9 }
```

Figure 5.10: Example of subclass implemented in the parser generated in *Perl*.

The "try" fragment (Figure 5.11) is very similar in structure to the *Java* one. In the *Perl* solution it is used a module, *Try::Tiny*[3], to achieve the "try/catch" behaviour

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

from *Java*, however, it has some differences that need to be addressed.

On the opposite of what happens in *Java*, the "catch" in *Perl* do not automatically recognises the exception throw within the "try", requiring the user to do the necessary validations. So, to accomplish this task it was required a *module* that was able to catch and recognise the exceptions. Amongst a few available *modules* in *CPAN*, it was chosen *Exception::Class*[4].

```
1  try {
2      ...
3  }
4  catch {
5      my $re;
6      $EVAL_ERROR = $_;
7      if ( $re = Exception::Class->caught('ANTLR::Runtime::
           RecognitionException') ) {
8          $self->report_error($re);
9          $self->recover($self->input, $re);
10     }
11 }
12 finally {};
```

Figure 5.11: "Try/catch" in *Perl* extracted from the generated parser.

Although *Exception::Class* ability to recognise the exceptions thrown, there was still an undesirable behaviour of *Try::Tiny*. When an exception is thrown within the "try", the error is stored into *Perl* default input special variable ("$_"). The problem is that the method "caught" of *Exception::Class* looks for an error in the special variable "$@". The solution was to assign "$_" value to "$@" at the beginning of the "catch" block.

```
1 dlock( my $FOLLOW_cabec_in_lavanda30 = ANTLR::Runtime::BitSet
     ->new({ words64 => [ '0x0000000000000040' ] }) );
2 dlock( my $FOLLOW_sacos_in_lavanda32 = ANTLR::Runtime::BitSet
     ->new({ words64 => [ '0x0000000000000002' ] }) );
```

Figure 5.12: Error recovery block.

Lastly, the error recovery block suffers some changes, but they have all been cov-

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

ered already. A snippet of the resulting code in *Perl* for this block can be seen in Figure 5.12.

5.2 Architecture

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. These tests will be covered in Chapter 6.

The Perl Code Generator is stored under the *code generator* folder of AnTLR, along with the corresponding *Perl* target class. The complement components are all included in the *runtime* folder, because they directly influence the correct installation of the library. The *Perl runtime* folder includes the root of installation of the library, a set of examples of implemented grammars, the runtime classes, the tests executed before installing the library, and tools to execute AnTLR or to check if any primary file has been changed. This hierarchy of files was initially designed by Ronald Blaschke[5].

The examples set contain some grammars implemented with AnTLR using the *Perl* syntax, along with the resultant parsers in *Perl*. Some of the examples also include an input file and the expected output file.

The tests were divided in three different categories. *Author* tests, to help the developers of the back-end, by identifying developing errors. *Class* tests, to guarantee that each of the Runtime library classes are correctly implemented and meet the expected behaviours. Lastly, the *Grammar* tests, responsible for verifying if the parsers are being correctly generated and, if the output for a particular input matches the expected output.

The tests are executed during the installation of the Perl runtime library. Since the generated Language Processors depend on the library to work properly, the Runtime library must be present in any machine that executes the LP. To install the Perl Runtime library is simply a matter of executing a make file.

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

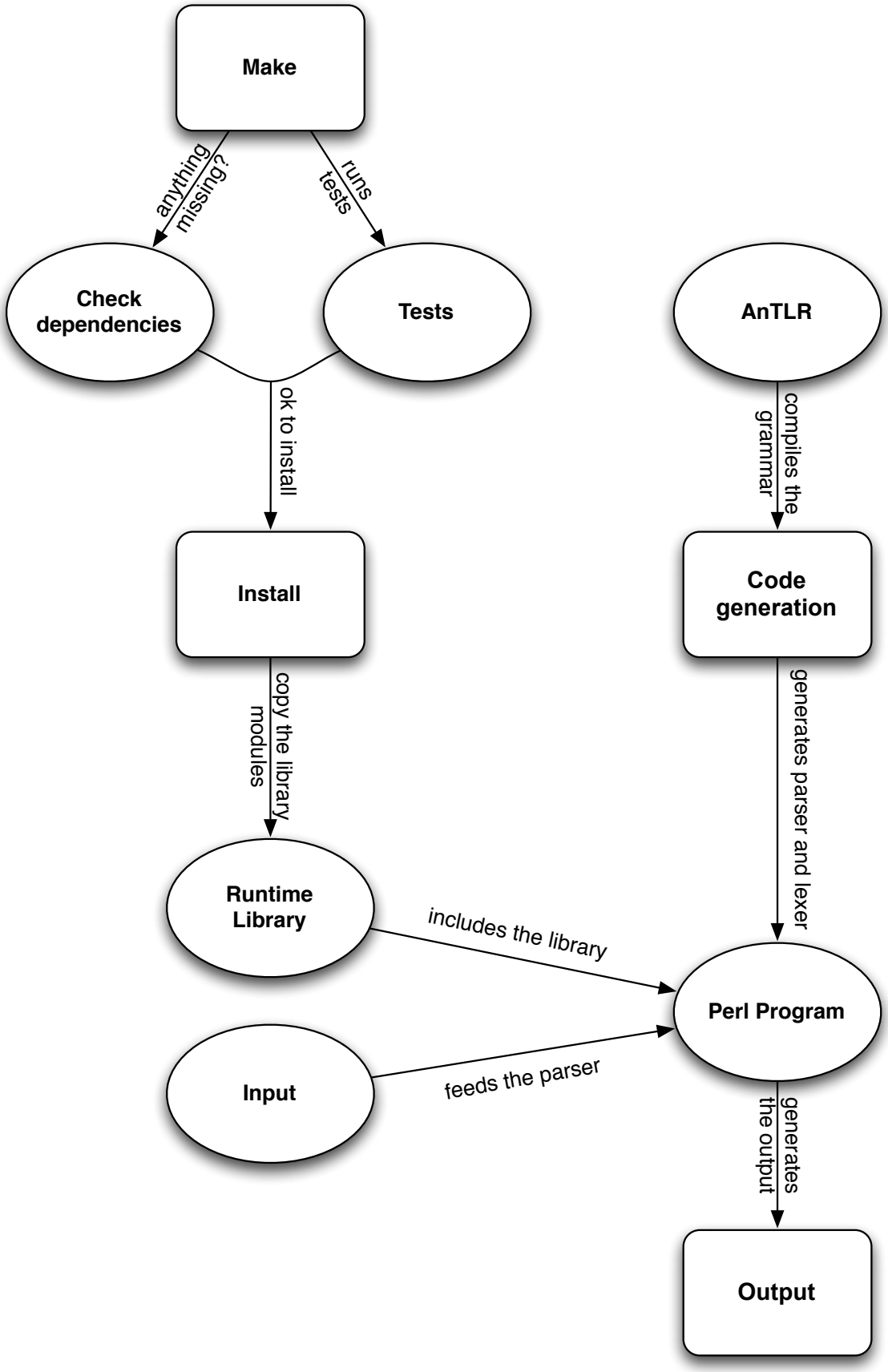


Figure 5.13: Runtime library installation process.

The library has some dependencies that should be installed before its installation. These dependencies are basically *Perl modules* required to implement the classes that constitute the Runtime library, and some others to perform the tests over these classes. *Moose*, which allows the library to be implemented in object-oriented *Perl*, is the only one present in all library classes. *Try::Tiny*, to allow the creation of *try* blocks and to catch exceptions during instructions evaluation. *Data::Lock*, to instantiate read only variables, and *Exception::Class* to handle exceptions. For the tests, the only requirement, besides the Runtime library, is the *module Test::More*[6]

After the library being installed, the generated parsers can be executed as any other *Perl* program. The installation process of the runtime library is illustrated in Figure 5.13. The Perl Code Generator works under AnTLR so, it has the same functional processes than the Java Code Generator. Therefore, its behaviour has already been covered in Section 4.4.

5.3 Perl Code Generator

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 with the *Java* solution, but at the same time the LP generated 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.

The first step to implement the Code Generator was to analyse the generated code for *Java*, generated by default, and *Python*, a language with some similarities with *Perl*. With these analysis it was possible to obtain a general idea of how the parser and lexer works and the algorithms implemented by them. This step was explained in Section 5.1.

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 easy 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

[6]http://search.cpan.org/ mschwern/Test-Simple-0.96/lib/Test/More.pm

works internally, including the data structures where the information about the grammar and the defined language are stored.

With all this in mind, the developing 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 *Java* code generated for it, and then, translating it to *Perl*. Then, it was just a matter of finding the template responsible for the generation of that fragment of code, and replace it. Lastly, AnTLR was executed to generate the code in *Perl*, to allow the verification of the generated code, checking whether is syntactically correct or not.

The tests were based on small grammars at the first stage of this 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 generating *Java* were yet to be translated and tested. This way it was easier to keep track of which ones were already implemented and which not.

To better understand this translation process, it will be given a brief explanation of how a template in *Java* was translated into *Perl*. The example used will be the one analysed in Section 4.2.2.

Looking to Figure 5.14 it is easy to recognise, for those who have some knowledge with *Java* language, the *Java* instructions and the StringTemplate. The *Java* instructions must be removed or replaced by the corresponding *Perl* instructions, if any, while the StringTemplate just needs to be positioned in the right order.

In the "positiveClosureBlock" template, all the variable types would be dropped, and would be replaced by the keyword "my". Also, the variable name, in *Perl*, would be preceded by the dollar sign ("$"). The order of the instructions would remain the same since it is a priority to keep the structure of the code when possible.

Other changes would be the name of the loop and case statements. The "do" would become a "while" and the "switch" would turn into "given" in *Perl*. Lastly, the only fragment of the template that suffers significant changes is the "default" block, where the creation of a new object "eee" would be translated to *Moose* notation, as would the method "throw" invocation.

The resulting template included in the Perl Code Generator can be seen in the Figure 5.14.

```
1  positiveClosureBlock(alts,decls,decision,enclosingBlockLevel,
       blockLevel,decisionNumber,maxK,maxAlt,description) ::= <<
2  # <fileName>:<description>
3  my $cnt<decisionNumber> = 0;
4  <decls>
5  <@preloop()>
6  LOOP<decisionNumber>:
7  while (1) {
8      my $alt<decisionNumber> = <maxAlt>;
9      <@predecision()>
10     <decision>
11     <@postdecision()>
12     given ($alt<decisionNumber>) {
13         <alts:altSwitchCase()>
14         default {
15             last LOOP<decisionNumber>
16                          if ( $cnt<decisionNumber> >= 1 );
17             <ruleBacktrackFailure()>
18             my $eee =
19                 ANTLR::Runtime::EarlyExitException->new({
20                     decision_number => <decisionNumber>,
21                     input => $self->input,
22                 });
23             <@earlyExitException()>
24             $eee->throw();
25         }
26     }
27     ++$cnt<decisionNumber>;
28 }
29 <@postloop()>
30 >>
```

Figure 5.14: Template to generate cases of one or more alternative (+) in *Perl*.

5.4 AnTLR Perl Runtime Library

The organization 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 (as explained below) for *Perl* and one new has been created.

ANTLRInputStream and *ANTLRReaderStream* are not present in *Perl* library, mainly, due to the flexibility of the language when dealing with input streams. However, these classes are going to be implemented in a future work, since they have particular characteristics that might be useful if implemented in *Perl*. Although the useful features, these classes are not essential in the parsing process, thus why they have been discarded from the Perl Runtime library in a first stage.

ClassicToken has also been discarded from the *Perl* library since the *Perl* back-end does not yet supports the display of AST, where this class plays its role. A few more classes have been discarded, primarily those used internally by AnTLR which are not required to run the generated parsers.

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 *Java* is an *OO* language, and its library is based in classes. 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* with the runtime classes and include also its constructs in the generation of the code. By using *Moose* to implement 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*, 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*. 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*.

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 instance, an *integer* in *Perl* can be used in *integer* calculations and string operations as well, 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 means that values assigned to a variable must correlate with its type, just like happens in *Java*.

Chapter 6

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 chapter.

6.1 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.

For a better optimisation of the tool it was required to evaluate the performance of each of its components using a profiler, to know exactly which code instructions should be improved to achieve better results.

The profiler chosen was the *Perl module Devel::NYTProf*, 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 from where they are called, etc. It is also possible

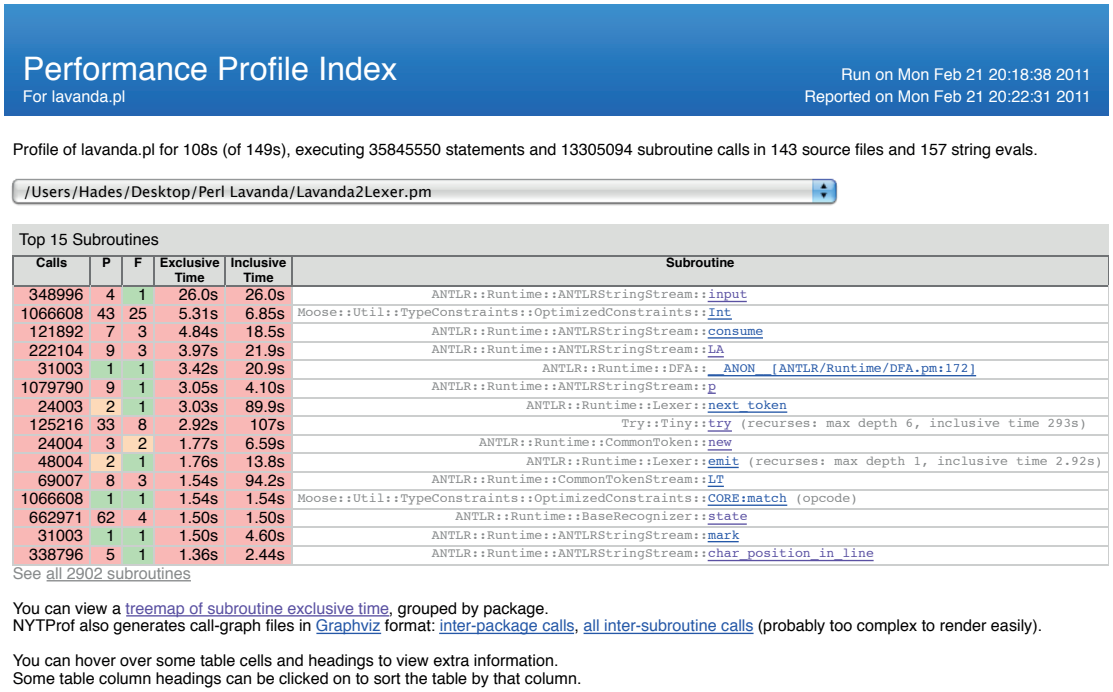
to verify the execution times of every subroutine instructions. This property is extremely useful to know where to start the optimisations of the tool.

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.

An example of the *HTML* visualisation for the generated LP for the *Lavanda* language can be seen in Figure 6.1.

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.



Performance Profile Index
For lavanda.pl

Run on Mon Feb 21 20:18:38 2011
Reported on Mon Feb 21 20:22:31 2011

Profile of lavanda.pl for 108s (of 149s), executing 35845550 statements and 13305094 subroutine calls in 143 source files and 157 string evals.

/Users/Hades/Desktop/Perl Lavanda/Lavanda2Lexer.pm

Top 15 Subroutines

Calls	P	F	Exclusive Time	Inclusive Time	Subroutine
348996	4	1	26.0s	26.0s	ANTLR::Runtime::ANTLRStringStream::input
1066608	43	25	5.31s	6.85s	Moose::Util::TypeConstraints::OptimizedConstraints::Int
121892	7	3	4.84s	18.5s	ANTLR::Runtime::ANTLRStringStream::consume
222104	9	3	3.97s	21.9s	ANTLR::Runtime::ANTLRStringStream::LA
31003	1	1	3.42s	20.9s	ANTLR::Runtime::DFA::__ANON__[ANTLR/Runtime/DFA.pm:172]
1079790	9	1	3.05s	4.10s	ANTLR::Runtime::ANTLRStringStream::p
24003	2	1	3.03s	89.9s	ANTLR::Runtime::Lexer::next_token
125216	33	8	2.92s	107s	Try::Tiny::try (recurses: max depth 6, inclusive time 293s)
24004	3	2	1.77s	6.59s	ANTLR::Runtime::CommonToken::new
48004	2	1	1.76s	13.8s	ANTLR::Runtime::Lexer::emit (recurses: max depth 1, inclusive time 2.92s)
69007	8	3	1.54s	94.2s	ANTLR::Runtime::CommonTokenStream::LT
1066608	1	1	1.54s	1.54s	Moose::Util::TypeConstraints::OptimizedConstraints::CORE:match (opcode)
662971	62	4	1.50s	1.50s	ANTLR::Runtime::BaseRecognizer::state
31003	1	1	1.50s	4.60s	ANTLR::Runtime::ANTLRStringStream::mark
338796	5	1	1.36s	2.44s	ANTLR::Runtime::ANTLRStringStream::char_position_in_line

See all 2902 subroutines

You can view a treemap of subroutine exclusive time, grouped by package.
NYTProf also generates call-graph files in Graphviz format: inter-package calls, all inter-subroutine calls (probably too complex to render easily).

You can hover over some table cells and headings to view extra information.
Some table column headings can be clicked on to sort the table by that column.

Figure 6.1: *HTML* visualisation of the result of the profiler Devel::NYTProf.

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, 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 the 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 Section 3.3.3, 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 as a single string, decreasing considerably their efficiency when dealing with large input texts.

To workaround this issue, the class which 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, 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 *1000* 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 6.2 along with the parsing time evolution.

More cases have been identified, some of them have been completely or partially solved, the remaining will be addressed on future work.

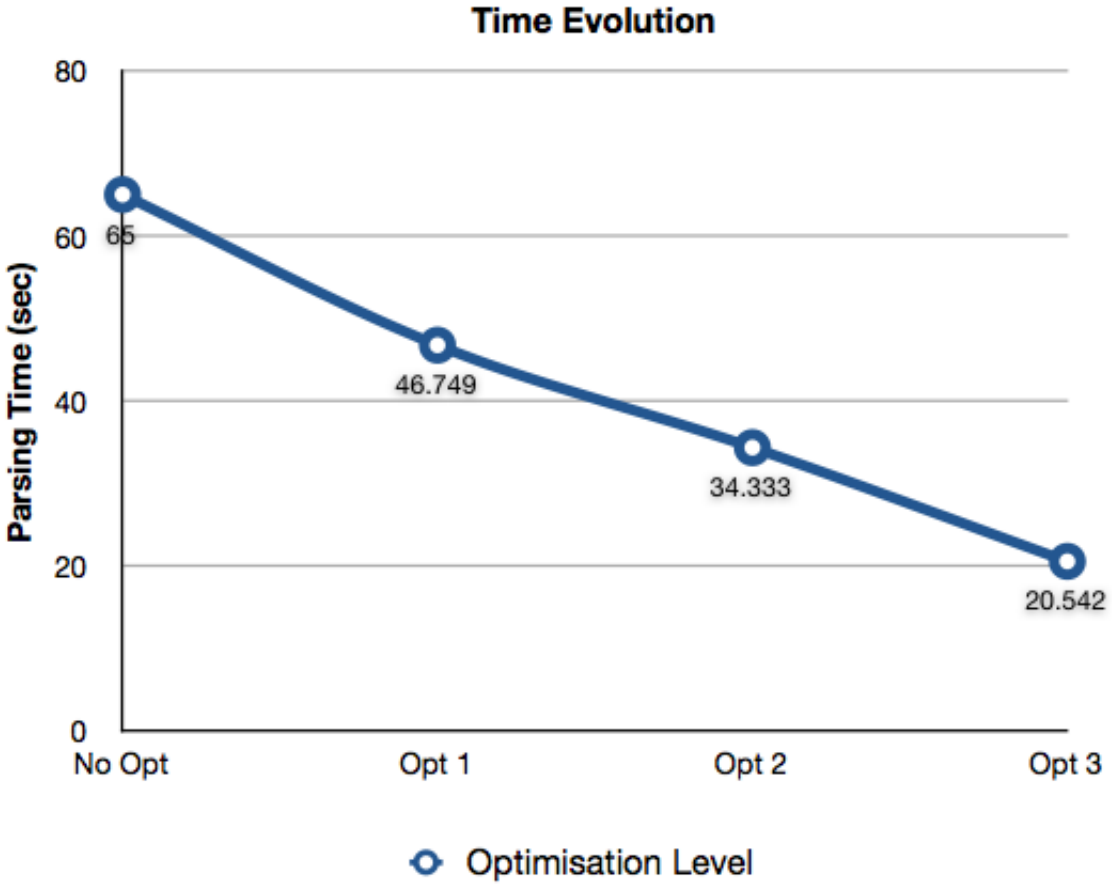


Figure 6.2: Time evolution after performing each optimisation level.

6.2 Evaluation

After developing the tool and refining the code to obtain better behaviour performance, specific tests have been planned to evaluate its correctness and performance. Besides the tests performed during the tool development process, some tests have been executed to evaluate the tool efficiency (parsing time and memory consumption) and verify whether the Runtime library classes behave as expected or not.

For a better analysis and evaluation of the proposed tool, comparisons will be made

with similar available tools for *Perl*. These tools were tested and analysed, as already discussed in Chapter 3; 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.
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 function 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. This means the latter needs to push the complete string into memory to perform an operation over the input. When dealing with large blocks of input data, this strategy can have severe consequences on the efficiency of the tool.

As explained in Section 3.3.2, 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.

A particular aspect of the tool is that, the parsing process continues executing instead of being interrupted in case of unrecoverable parsing errors. The errors are logged and accessible at the end of the parsing process.

Adding this analysis to the tables introduced in Section 3.3.3 (see Tables 6.1 and 6.2), it is possible to realise that, in general, the Perl Code Generator is the tool that presents better results.

Table 6.1: Tool Analysis.

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

The performance tests were ran in a different machine than the one used for the tests made in Section 3.3.3 in an attempt to see if the tested parsers were able to finish parsing in a faster machine. The RAM size is the same, but the CPU is relatively

Table 6.2: 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
Perl Code Generator	LL(k)	++	Yes	Yes	++	Yes

faster than before. Only three *Perl* parser generators were selected for these tests, the best *LALR* and *LL* based parser generators of the earlier tests and the tool developed in this work.

1. *Parse::Yapp* — The best *LALR* based parser generator.
2. *Parse::RecDescent* — The best $LL(1)$ based parser generator.
3. *Perl Code Generator* — The tool developed in this work.

As seen before, *LR* 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 small input streams, 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 6.3.

Table 6.3: 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
1000000			

Regarding the *LL* based parser generators, *Parse::RecDescent* parsers spend less

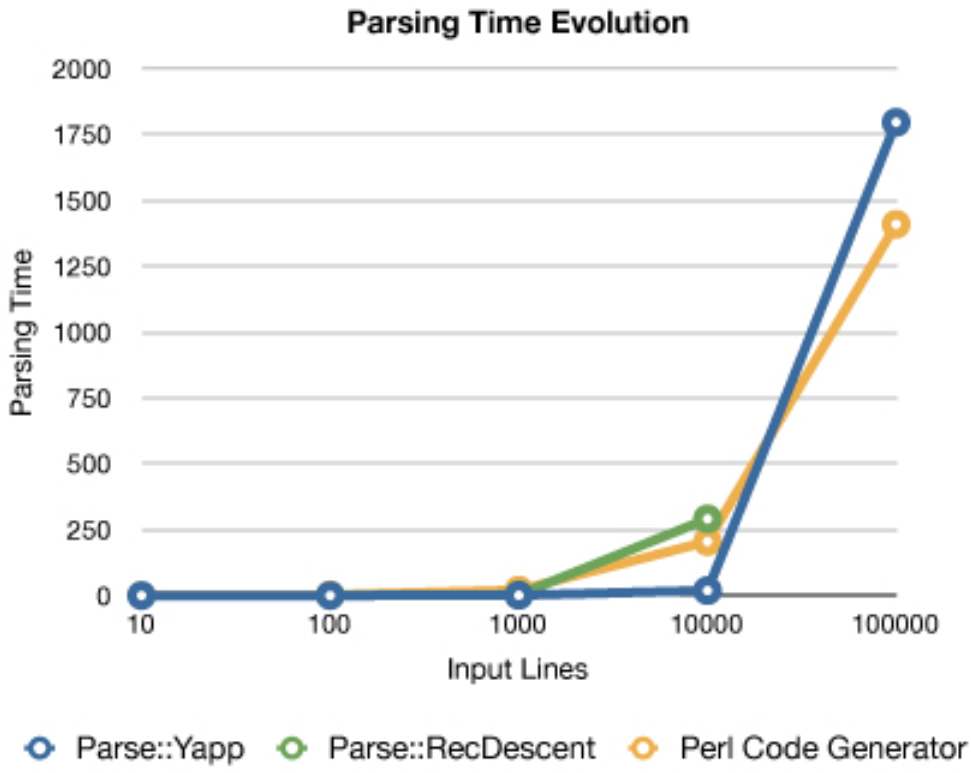


Figure 6.3: Parsing time evolution for the tested parsers.

time parsing small inputs than the *Perl Code Generator*, but its efficiency highly decreases when dealing with large input streams.

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 generators 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.

The time evolution of the parsing processes can be observed in Figure 6.3.

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 6.4. 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. The memory used by the parsers can be seen in Figure 6.4.

Table 6.4: 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
100000			

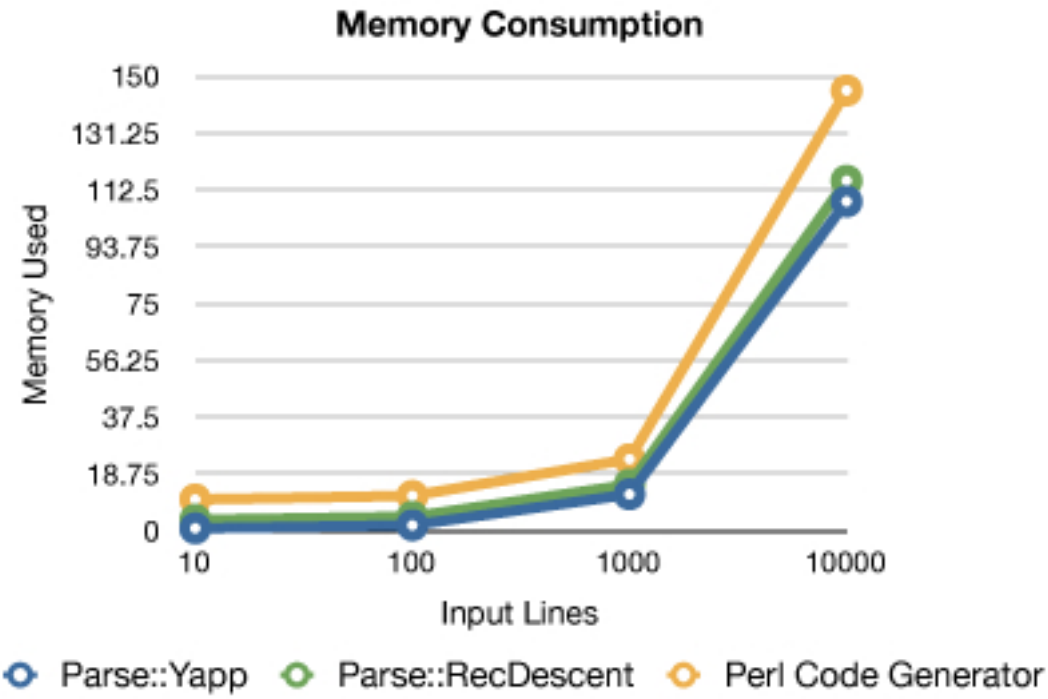


Figure 6.4: Memory used by the tested parsers.

Chapter 7

Conclusion

In this dissertation, it was shown that *Perl* lacks a parser generator able to generate efficient attribute based language processors, especially when dealing with large input streams. Therefore, a new solution that combines the best of the existing generators and overcomes their limitations was needed, as described in Chapter 2.

The limitations of the more efficient parser generators available for *Perl* have been identified [3] after a careful analysis and a set of performance tests over the tools, as shown in Chapter 3.

From the possible solutions, it was chosen the integration of a *Perl* back-end into AnTLR, a tool with proven results in the language recognition field. Besides profiting from AnTLR acceptance and support, this choice also takes advantage from the algorithms used and features provided by AnTLR. To our eyes, one of the main advantages is the support to attribute grammars implementation, something that was still missing in *Perl*.

To create a Perl Code Generator it was first required an exhaustive study about AnTLR architecture, its internal components and their modus operandi. To describe this study, in Chapter 4, it was presented each of AnTLR components, Analysis, Runtime Library, Code Generator, StringTemplate and Tool main classes, their roles and how they interact with each other.

This study made possible the implementation of a code generator for *Perl* that was fully integrated with AnTLR. Its implementation strategies and decisions were explained in Chapter 5.

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 Chapter 5, 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 large input streams since the other tools lose efficiency as the input size increases.

Based on the tests performed during this project, the Perl Code Generator is clearly the best solution 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.

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. Though, the AST it is not yet available for display and the debug facilities are not totally usable.

The optimisations described in Chapter 5 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, despite the *Perl* limitations.

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.

Since this is an initial version of the Perl Code Generator, there are still future work expected to be done:

- The implementation of the *tree library* (part of the Runtime Library) responsible for giving support to the creation of ASTs and execution of parsers based on tree grammars.

- The implementation of the debug mechanisms to assist on the debug of the parsing process.

- The translation of the remaining templates from *Java* to *Perl*, including the output of ASTs.
- Extend the functioning of the Perl Code Generator to *ANTLRWorks*[1], the grammar development environment for AnTLRv3.
- Implement the StringTemplate engine in *Perl*, making possible the rewrite of the templates with *Perl* syntax or even make use of the templates practice in other *Perl* projects.

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.

[1] http://www.antlr.org/works/index.html

Bibliography

[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] Stephen Johnson Bell and Stephen C. Johnson. Yacc: Yet another compiler-compiler. Technical report, 1979.

[5] Noami Chomsky. Three models for the description of languages. *IRE Transactions on Information Theory*, IT-2(3):113–124, Sep. 1956.

[6] Damian Conway. Parse::recdescent. `http://search.cpan.org/dist/Parse-RecDescent/lib/Parse/RecDescent.pm`, 1997.

[7] Damian Conway. Regexp::grammars. `http://search.cpan.org/~dconway/Regexp-Grammars-1.001005/lib/Regexp/Grammars.pm`, 2009.

[8] 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.

[9] Francois Desarmenien. Parse::yapp. `http://search.cpan.org/dist/Parse-Yapp/lib/Parse/Yapp.pm`, 1998.

[10] Gabriel J. Ferrer. Simplifying parser generation, May 2007.

[11] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA, 1999.

[12] Jeffrey E. F. Friedl. *Mastering Regular Expressions.* O'Reilly Media, Inc., 2006.

[13] Ceriel J. H. Jacobs. Some topics in parser generation.

[14] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation.* Wiley-IEEE Computer Society Pr, March 2008.

[15] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[16] Tomaz Kosar, Pablo Martínez Lopez, Pablo Barrientos, and Marjan Mernik. A preliminary study on various implementation approaches of domain-specific languages. *IST – Information and Software Technology*, 50(5):390–405, 2008.

[17] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages.* Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007.

[18] Terence Parr and Russell W. Quong. Antlr: A predicated-ll(k) parser generator. *Software Practice and Experience*, 25(7):789–810, July 1995.

[19] Casiano Rodriguez-Leon. Parse::eyapp. `http://search.cpan.org/dist/Parse-Eyapp/lib/Parse/Eyapp.pod`, 2006.

[20] Alberto Simões. Cooking perl with flex. *The Perl Review*, 0(3), May 2002.

[21] Alberto Simões and José João Almeida. Text::rewriterules. `http://search.cpan.org/~ambs/Text-RewriteRules-0.21/lib/Text/RewriteRules.pm`, 2004.

[22] T. Sloane, M. Mernik, and J. Heering. When and how to develop domain-specific languages. Technical Report SEN-EE0309, CWI – Centre for Mathematics and Computer Science, 2003.

[23] Jean-Paul Tremblay and Paul G. Sorenson. *The Theory and Practice of Compiler Writing.* McGraw-Hill, Inc., New York, NY, USA, 1st edition, 1985.

[24] William M. Waite. Use of attribute grammars in compiler construction. In *WAGA*, pages 255–265, 1990.

[25] Larry Wall. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.

Appendices

Appendix A

Grammars

In this chapter are shown the grammars used to implement the tests used during this work. The *Lavanda* was the selected grammar in the presentations of the results of the tests done to the tool developed in the scope of this master thesis.

A.1 Lavanda Grammar

```
1  p1:       Lavanda --> Cabec Sacos
2  p2:       Cabec   --> data IdPR
3  p3:       Sacos   --> Saco '.'
4  p4:                 | Sacos Saco '.'
5  p5:       Saco    --> num IdCli Lotes
6  p6:       Lotes   --> Lote Outros
7  p7:       Lote    --> Tipo Qt
8  p8:       Tipo    --> Classe Tinto Fio
9  p9:       Outros  --> &
10 p10:                | ';' Lotes
11 p11:      IdPR    --> id
12 p12:      IdCli   --> id
13 p13:      Qt      --> num
14 p14:      Classe  --> corpo
15 p15:                | casa
16 p16:      Tinto   --> br
17 p17:                | cor
18 p18:      Fio     --> alg
19 p19:                | la
20 p20:                | fib
```

A.2 Swedish Chef Grammar

```
1  p1:        chef    --> token
2  p2:                  | chef token
3  p3:        token   --> EOS
4  p4:                  | Bbork
5  p5:                  | an
6  p6:                  | An
7  p7:                  | au
8  p8:                  | Au
9  p9:                  | ax
10 p10:                 | Ax
11 p11:                 | en
12 p12:                 | ew
13 p13:                 | edone
14 p14:                 | ex
15 p15:                 | Ex
16 p16:                 | f
17 p17:                 | ir
18 p18:                 | i
19 p19:                 | ow
20 p20:                 | o
21 p21:                 | O
22 p22:                 | xo
23 p23:                 | the
24 p24:                 | The
25 p25:                 | th
26 p26:                 | tion
27 p27:                 | u
28 p28:                 | U
29 p29:                 | v
30 p30:                 | V
31 p31:                 | w
32 p32:                 | W
33 p33:                 | NW
34 p34:                 | WC
35 p35:                 | NL
```

A.3 S-expression Grammar

```
1 p1:        lisp     --> sExp
2 p2:        sExp     --> NUM
3 p3:                   | PAL
4 p4:                   | '(' sExplist ')'
5 p5:        sExplist --> &
6 p6:                   | sExp sExplist
```

Appendix B

ANTLR Java Code Generator

In this chapter are presented the parsers generated in *Java* by AnTLR for the *Lavanda* grammar. The *Lavanda* implementation in AnTLR metalanguage includes two approaches, the first based in global variables and other based in attributes. Both implementations in AnTLR are also shown in this chapter.

B.1 ANTLR Implementation of Lavanda Grammar using Global Variables

```
1  grammar Lavanda1;
2
3  @header {
4          import java.util.HashMap;
5          import java.util.LinkedList;
6  }
7
8  @members {
9          HashMap inTable = new HashMap();
10         HashMap env     = new HashMap();
11         int nSacos      = 0;
12         int custoTotal  = 0;
13         int nLotes      = 0;
14         LinkedList clientIds = new LinkedList();
15         LinkedList bagIds    = new LinkedList();
16 }
17
18 lavanda
19         : cabec sacos
20             { System.out.println("Total sacos: " + nSacos);
21               for(Object o : env.keySet())
```

```
22                      System.out.println((String) o + "\t" + (Integer) env.get(o)); }
23        ;
24
25 cabec
26        : DATA ID
27        ;
28
29 sacos
30        : ( saco { nSacos++; } )+
31        ;
32
33 saco
34        : NUM { if(bagIds.contains(Integer.parseInt($NUM.text)))
35                   System.err.println("Bag ID already exists!");
36                bagIds.add(Integer.parseInt($NUM.text)); }
37          ID  { if(clientIds.contains($ID.text))
38                   System.err.println("Client ID already exists!");
39                clientIds.add($ID.text);
40                nLotes = 0; custoTotal = 0; }
41          '(' lotes ')' {
42                  System.out.print("Numero de lotes para o ID " + $ID.text + ": " + nLotes);
43                  System.out.println(" Custo: " + custoTotal); }
44        ;
45
46 lotes
47        : l1=lote { nLotes++; }
48          ( ',' l2=lote { nLotes++; } )*
49        ;
50
51 lote
52        : tipo NUM { custoTotal += (Integer) inTable.get($tipo.name) * Integer.parseInt($NUM.text);
53                      if(!env.containsKey($tipo.name))
54                         env.put($tipo.name, 0);
55                      env.put($tipo.name, (Integer) env.get($tipo.name) + 1); }
56        ;
57
58 tipo returns [ String name ]
59        : classe '-' tinto '-' fio { $name = $classe.name + '-' + $tinto.name + '-' + $fio.name; }
60        ;
61
62 classe returns [ String name ]
63        : 'corpo' { $name = "corpo"; }
64        | 'casa'  { $name = "casa";  }
65        ;
66
67 tinto returns [ String name ]
68        : 'br'   { $name = "br";  }
69        | 'cor'  { $name = "cor"; }
70        ;
71
72 fio returns [ String name ]
73        : 'alg'  { $name = "alg"; }
74        | 'la'   { $name = "la";  }
75        | 'fib'  { $name = "fib"; }
```

```
76         ;
77
78 fragment
79 LETTER  : 'a'..'z'|'A'..'Z' ;
80 fragment
81 DIGIT   : '0'..'9' ;
82
83 ID   : LETTER+;
84 NUM  : DIGIT+;
85 DATA : DIGIT DIGIT '-' DIGIT DIGIT '-' DIGIT DIGIT DIGIT DIGIT ;
86 WS   : ('\r'|'\n'|' '|'\t')+ {skip();} ;
```

B.2 ANTLR Implementation of Lavanda Grammar using Attributes

```
1  grammar Lavanda2;
2
3  @header {
4          import java.util.HashMap;
5          import java.util.LinkedList;
6  }
7
8  @members {
9          HashMap inTable = new HashMap();
10 }
11
12 lavanda
13         : cabec sacos
14             { System.out.println("Total sacos: " + $sacos.nSacos);
15               for(Object o : $sacos.outEnv.keySet())
16                       System.out.println((String) o + "\t" + (Integer) $sacos.outEnv.get(o));
17         }
18         ;
19
20 cabec
21         : DATA ID
22         ;
23
24 sacos returns [ int nSacos = 0, HashMap outEnv = new HashMap() ]
25 @init { LinkedList clientIds = new LinkedList();
26         LinkedList bagIds    = new LinkedList();
27         HashMap inEnv = new HashMap();            }
28         : ( saco[inEnv, clientIds, bagIds] { $nSacos++; $outEnv = $saco.outEnv; } )+
29         ;
30
31 saco [ HashMap inEnv, LinkedList clientIds, LinkedList bagIds ] returns [ HashMap outEnv ]
32         :  NUM { if(bagIds.contains(Integer.parseInt($NUM.text)))
33                    System.err.println("Bag ID already exists!");
34                  bagIds.add(Integer.parseInt($NUM.text)); }
```

```
35            ID  { if(clientIds.contains($ID.text))
36                     System.err.println("Client ID already exists!");
37                  clientIds.add($ID.text); }
38            '(' lotes[inEnv] ')' {
39                  System.out.print("Numero de lotes para o ID " + $ID.text + ": " + $lotes.nLotes);
40                  System.out.println(" Custo: " + $lotes.custoTotal);
41                  $outEnv = $lotes.outEnv; }
42         ;
43
44 lotes [ HashMap inEnv ] returns [ HashMap outEnv, int nLotes = 0, int custoTotal = 0]
45         : l1=lote[$inEnv]  { $nLotes++; $custoTotal += $l1.custoTotal; $outEnv = $l1.outEnv; }
46           (',' l2=lote[$outEnv] { $nLotes++; $custoTotal += $l2.custoTotal; $outEnv = $l2.outEnv; } )*
47         ;
48
49 lote [ HashMap inEnv ] returns [ int custoTotal, HashMap outEnv ]
50         : tipo NUM { $custoTotal = (Integer) inTable.get($tipo.name) * Integer.parseInt($NUM.text);
51                  if(!$inEnv.containsKey($tipo.name))
52                     $inEnv.put($tipo.name, 0);
53                  $inEnv.put($tipo.name, (Integer) $inEnv.get($tipo.name) + 1);
54                  $outEnv = $inEnv; }
55         ;
56
57 tipo returns [ String name ]
58         : classe '-' tinto '-' fio { $name = $classe.name + '-' + $tinto.name + '-' + $fio.name; }
59         ;
60
61 classe returns [ String name ]
62         : 'corpo' { $name = "corpo"; }
63         | 'casa'  { $name = "casa";  }
64         ;
65
66 tinto returns [ String name ]
67         : 'br'   { $name = "br";  }
68         | 'cor'  { $name = "cor"; }
69         ;
70
71 fio returns [ String name ]
72         : 'alg'  { $name = "alg"; }
73         | 'la'   { $name = "la";  }
74         | 'fib'  { $name = "fib"; }
75         ;
76
77 fragment
78 LETTER  : 'a'..'z'|'A'..'Z' ;
79 fragment
80 DIGIT   : '0'..'9' ;
81
82 ID    : LETTER+;
83 NUM   : DIGIT+;
84 DATA  : DIGIT DIGIT '-' DIGIT DIGIT '-' DIGIT DIGIT DIGIT DIGIT ;
85 WS    : ('\r'|'\n'|' '|'\t')+ {skip();} ;
```

B.3 Generated Parser

B.3.1 Approach using Global Variables

```
1  // $ANTLR 3.2 Sep 23, 2009 12:02:23 /Users/Hades/Desktop/Lavanda1.g 2010-03-16 16:23:45
2
3  import java.util.HashMap;
4  import java.util.LinkedList;
5
6
7  import org.antlr.runtime.*;
8  import java.util.Stack;
9  import java.util.List;
10 import java.util.ArrayList;
11
12 public class Lavanda1Parser extends Parser {
13     public static final String[] tokenNames = new String[] {
14         "<invalid>", "<EOR>", "<DOWN>", "<UP>", "DATA", "ID", "NUM", "LETTER", "DIGIT", "WS",
15         "'('", "')'", "','", "'-'", "'corpo'", "'casa'", "'br'", "'cor'", "'alg'", "'la'", "'fib'"
16     };
17     public static final int LETTER=7;
18     public static final int T__12=12;
19     public static final int T__20=20;
20     public static final int WS=9;
21     public static final int T__13=13;
22     public static final int T__19=19;
23     public static final int DATA=4;
24     public static final int T__14=14;
25     public static final int T__11=11;
26     public static final int T__17=17;
27     public static final int EOF=-1;
28     public static final int T__16=16;
29     public static final int NUM=6;
30     public static final int T__10=10;
31     public static final int DIGIT=8;
32     public static final int T__18=18;
33     public static final int T__15=15;
34     public static final int ID=5;
35
36     public Lavanda1Parser(TokenStream input) {
37         this(input, new RecognizerSharedState());
38     }
39     public Lavanda1Parser(TokenStream input, RecognizerSharedState state) {
40         super(input, state);
41         }
42
43     public String[] getTokenNames() { return Lavanda1Parser.tokenNames; }
44     public String getGrammarFileName() { return "/Users/Hades/Desktop/Lavanda1.g"; }
45
46     HashMap inTable = new HashMap();
47     HashMap env     = new HashMap();
48     int nSacos      = 0;
```

```
49      int custoTotal  = 0;
50      int nLotes      = 0;
51      LinkedList clientIds = new LinkedList();
52      LinkedList bagIds    = new LinkedList();
53
54          ...
55
56      // /Users/Hades/Desktop/Lavanda1.g:28:1: sacos : ( saco )+ ;
57      public final void sacos() throws RecognitionException {
58          try {
59              {
60              // /Users/Hades/Desktop/Lavanda1.g:29:4: ( saco )+
61              int cnt1=0;
62              loop1:
63              do {
64                  int alt1=2;
65                  int LA1_0 = input.LA(1);
66
67                  if ( (LA1_0==NUM) ) {
68                      alt1=1;
69                  }
70
71                                  switch (alt1) {
72                      case 1 :
73                          // /Users/Hades/Desktop/Lavanda1.g:29:6: saco
74                          {
75                          pushFollow(FOLLOW_saco_in_sacos60);
76                          saco();
77
78                          state._fsp--;
79
80                          nSacos++;
81
82                          }
83                          break;
84
85                      default :
86                          if ( cnt1 >= 1 ) break loop1;
87                          EarlyExitException eee = new EarlyExitException(1, input);
88                          throw eee;
89                  }
90                  cnt1++;
91              } while (true);
92              }
93
94          }
95          catch (RecognitionException re) {
96              reportError(re);
97              recover(input,re);
98          }
99          finally {
100         }
101         return ;
102     }
```

```
103
104     // /Users/Hades/Desktop/Lavanda1.g:32:1: saco : NUM ID '(' lotes ')' ;
105     public final void saco() throws RecognitionException {
106         Token NUM1=null;
107         Token ID2=null;
108
109         try {
110             // /Users/Hades/Desktop/Lavanda1.g:33:6: NUM ID '(' lotes ')'
111             {
112             NUM1=(Token)match(input,NUM,FOLLOW_NUM_in_saco79);
113             if(bagIds.contains(Integer.parseInt((NUM1!=null?NUM1.getText():null))))
114                             System.err.println("Bag ID already exists!");
115                     bagIds.add(Integer.parseInt((NUM1!=null?NUM1.getText():null)));
116             ID2=(Token)match(input,ID,FOLLOW_ID_in_saco88);
117             if(clientIds.contains((ID2!=null?ID2.getText():null)))
118                             System.err.println("Client ID already exists!");
119             clientIds.add((ID2!=null?ID2.getText():null));
120             nLotes = 0; custoTotal = 0;
121             match(input,10,FOLLOW_10_in_saco97);
122             pushFollow(FOLLOW_lotes_in_saco99);
123             lotes();
124
125             state._fsp--;
126
127             match(input,11,FOLLOW_11_in_saco101);
128             System.out.print("Numero de lotes para o ID " +
129                             (ID2!=null?ID2.getText():null) + ": " + nLotes);
130             System.out.println(" Custo: " + custoTotal);
131
132             }
133
134         }
135         catch (RecognitionException re) {
136             reportError(re);
137             recover(input,re);
138         }
139         finally {
140         }
141         return ;
142     }
143
144     // /Users/Hades/Desktop/Lavanda1.g:44:1: lotes : l1= lote ( ',' l2= lote )* ;
145     public final void lotes() throws RecognitionException {
146         try {
147             // /Users/Hades/Desktop/Lavanda1.g:44:13: l1= lote ( ',' l2= lote )*
148             {
149             pushFollow(FOLLOW_lote_in_lotes120);
150             lote();
151
152             state._fsp--;
153
154             nLotes++;
155             // /Users/Hades/Desktop/Lavanda1.g:45:6: ( ',' l2= lote )*
156             loop2:
```

```
157             do {
158                 int alt2=2;
159                 int LA2_0 = input.LA(1);
160 
161                 if ( (LA2_0==12) ) {
162                     alt2=1;
163                 }
164 
165                 switch (alt2) {
166                     case 1 :
167                         // /Users/Hades/Desktop/Lavanda1.g:45:8: ',' l2= lote
168                         {
169                         match(input,12,FOLLOW_12_in_lotes131);
170                         pushFollow(FOLLOW_lote_in_lotes135);
171                         lote();
172 
173                         state._fsp--;
174 
175                         nLotes++;
176 
177                         }
178                         break;
179 
180                     default :
181                         break loop2;
182                 }
183             } while (true);
184             }
185 
186         }
187         catch (RecognitionException re) {
188             reportError(re);
189             recover(input,re);
190         }
191         finally {
192         }
193         return ;
194     }
195 
196         ...
197 
198     public static final BitSet FOLLOW_cabec_in_lavanda23 = new BitSet(new long[]{0x0000000000000040L});
199     public static final BitSet FOLLOW_sacos_in_lavanda25 = new BitSet(new long[]{0x0000000000000002L});
200     public static final BitSet FOLLOW_DATA_in_cabec44 = new BitSet(new long[]{0x0000000000000020L});
201     public static final BitSet FOLLOW_ID_in_cabec46 = new BitSet(new long[]{0x0000000000000002L});
202     public static final BitSet FOLLOW_saco_in_sacos60 = new BitSet(new long[]{0x0000000000000042L});
203         ...
204 
205 }
```

B.3.2 Approach using Attributes

```
1  // $ANTLR 3.2 Sep 23, 2009 12:02:23 /Users/Hades/Desktop/Lavanda2.g 2010-04-27 16:21:12
2
3  import java.util.HashMap;
4  import java.util.LinkedList;
5
6
7  import org.antlr.runtime.*;
8  import java.util.Stack;
9  import java.util.List;
10 import java.util.ArrayList;
11
12 public class Lavanda2Parser extends Parser {
13     public static final String[] tokenNames = new String[] {
14         "<invalid>", "<EOR>", "<DOWN>", "<UP>", "DATA", "ID", "NUM", "LETTER", "DIGIT", "WS",
15         "'('", "')'", "','", "'-'", "'corpo'", "'casa'", "'br'", "'cor'", "'alg'", "'la'", "'fib'"
16     };
17     public static final int LETTER=7;
18     public static final int T__12=12;
19     public static final int T__20=20;
20     public static final int WS=9;
21     public static final int T__13=13;
22     public static final int T__19=19;
23     public static final int DATA=4;
24     public static final int T__14=14;
25     public static final int T__11=11;
26     public static final int T__17=17;
27     public static final int EOF=-1;
28     public static final int T__16=16;
29     public static final int NUM=6;
30     public static final int T__10=10;
31     public static final int DIGIT=8;
32     public static final int T__18=18;
33     public static final int T__15=15;
34     public static final int ID=5;
35
36     public Lavanda2Parser(TokenStream input) {
37         this(input, new RecognizerSharedState());
38     }
39     public Lavanda2Parser(TokenStream input, RecognizerSharedState state) {
40         super(input, state);
41     }
42
43     public String[] getTokenNames() { return Lavanda2Parser.tokenNames; }
44     public String getGrammarFileName() { return "/Users/Hades/Desktop/Lavanda2.g"; }
45
46     HashMap inTable = new HashMap();
47
48         ...
49
50     public static class sacos_return extends ParserRuleReturnScope {
51         public int nSacos = 0;
```

```
52          public HashMap outEnv = new HashMap();
53      };
54
55      // /Users/Hades/Desktop/Lavanda2.g:23:1: sacos returns
56      // [ int nSacos = 0, HashMap outEnv = new HashMap() ] : ( saco[inEnv, clientIds, bagIds] )+ ;
57      public final Lavanda2Parser.sacos_return sacos() throws RecognitionException {
58          Lavanda2Parser.sacos_return retval = new Lavanda2Parser.sacos_return();
59          retval.start = input.LT(1);
60
61          HashMap saco2 = null;
62
63
64           LinkedList clientIds = new LinkedList();
65                  LinkedList bagIds    = new LinkedList();
66                  HashMap inEnv = new HashMap();
67          try {
68              {
69              // /Users/Hades/Desktop/Lavanda2.g:27:4: ( saco[inEnv, clientIds, bagIds] )+
70              int cnt1=0;
71              loop1:
72              do {
73                  int alt1=2;
74                  int LA1_0 = input.LA(1);
75
76                  if ( (LA1_0==NUM) ) {
77                      alt1=1;
78                  }
79
80
81                  switch (alt1) {
82                      case 1 :
83                          // /Users/Hades/Desktop/Lavanda2.g:27:7: saco[inEnv, clientIds, bagIds]
84                          {
85                          pushFollow(FOLLOW_saco_in_sacos70);
86                          saco2=saco(inEnv, clientIds, bagIds);
87
88                          state._fsp--;
89
90                           retval.nSacos++; retval.outEnv = saco2;
91
92                          }
93                          break;
94
95                      default :
96                          if ( cnt1 >= 1 ) break loop1;
97                          EarlyExitException eee =
98                              new EarlyExitException(1, input);
99                          throw eee;
100                 }
101                 cnt1++;
102             } while (true);
103
104
105             }
```

```
106
107             retval.stop = input.LT(-1);
108
109         }
110         catch (RecognitionException re) {
111             reportError(re);
112             recover(input,re);
113         }
114         finally {
115         }
116         return retval;
117     }
118
119     // /Users/Hades/Desktop/Lavanda2.g:30:1: saco[ HashMap inEnv, LinkedList clientIds,
120     // LinkedList bagIds ] returns [ HashMap outEnv ] : NUM ID '(' lotes[inEnv] ')' ;
121     public final HashMap saco(HashMap inEnv, LinkedList clientIds, LinkedList bagIds)
122     throws RecognitionException {
123         HashMap outEnv = null;
124
125         Token NUM3=null;
126         Token ID4=null;
127         Lavanda2Parser.lotes_return lotes5 = null;
128
129
130         try {
131             // /Users/Hades/Desktop/Lavanda2.g:31:5: NUM ID '(' lotes[inEnv] ')'
132             {
133             NUM3=(Token)match(input,NUM,FOLLOW_NUM_in_saco95);
134              if(bagIds.contains(Integer.parseInt((NUM3!=null?NUM3.getText():null))))
135                             System.err.println("Bag ID already exists!");
136                             bagIds.add(Integer.parseInt((NUM3!=null?NUM3.getText():null)));
137             ID4=(Token)match(input,ID,FOLLOW_ID_in_saco104);
138              if(clientIds.contains((ID4!=null?ID4.getText():null)))
139                             System.err.println("Client ID already exists!");
140                             clientIds.add((ID4!=null?ID4.getText():null));
141             match(input,10,FOLLOW_10_in_saco113);
142             pushFollow(FOLLOW_lotes_in_saco115);
143             lotes5=lotes(inEnv);
144
145             state._fsp--;
146
147             match(input,11,FOLLOW_11_in_saco118);
148             System.out.print("Numero de lotes para o ID " + (ID4!=null?ID4.getText():null) +
149                             ": " + (lotes5!=null?lotes5.nLotes:0));
150             System.out.println(" Custo: " + (lotes5!=null?lotes5.custoTotal:0));
151             outEnv = (lotes5!=null?lotes5.outEnv:null);
152
153             }
154
155         }
156         catch (RecognitionException re) {
157             reportError(re);
158             recover(input,re);
159         }
```

```
160         finally {
161         }
162         return outEnv;
163     }
164 
165     public static class lotes_return extends ParserRuleReturnScope {
166         public HashMap outEnv;
167         public int nLotes = 0;
168         public int custoTotal = 0;
169     };
170 
171     // /Users/Hades/Desktop/Lavanda2.g:42:1: lotes[ HashMap inEnv ]
172     // returns [ HashMap outEnv, int nLotes = 0, int custoTotal = 0]
173     // : l1= lote[$inEnv] ( ',' l2= lote[$outEnv] )* ;
174     public final Lavanda2Parser.lotes_return lotes(HashMap inEnv) throws RecognitionException {
175         Lavanda2Parser.lotes_return retval = new Lavanda2Parser.lotes_return();
176         retval.start = input.LT(1);
177 
178         Lavanda2Parser.lote_return l1 = null;
179 
180         Lavanda2Parser.lote_return l2 = null;
181 
182 
183         try {
184             // /Users/Hades/Desktop/Lavanda2.g:43:11: l1= lote[$inEnv] ( ',' l2= lote[$outEnv] )*
185             {
186             pushFollow(FOLLOW_lote_in_lotes147);
187             l1=lote(inEnv);
188 
189             state._fsp--;
190 
191             retval.nLotes++;
192             retval.custoTotal += (l1!=null?l1.custoTotal:0);
193             retval.outEnv = (l1!=null?l1.outEnv:null);
194             // /Users/Hades/Desktop/Lavanda2.g:44:6: ( ',' l2= lote[$outEnv] )*
195             loop2:
196             do {
197                 int alt2=2;
198                 int LA2_0 = input.LA(1);
199 
200                 if ( (LA2_0==12) ) {
201                     alt2=1;
202                 }
203 
204 
205                 switch (alt2) {
206                     case 1 :
207                         // /Users/Hades/Desktop/Lavanda2.g:44:7: ',' l2= lote[$outEnv]
208                         {
209                         match(input,12,FOLLOW_12_in_lotes159);
210                         pushFollow(FOLLOW_lote_in_lotes163);
211                         l2=lote(retval.outEnv);
212 
213                         state._fsp--;
```

```
214
215                         retval.nLotes++;
216                         retval.custoTotal += (l2!=null?l2.custoTotal:0);
217                         retval.outEnv = (l2!=null?l2.outEnv:null);
218
219                         }
220                         break;
221
222                     default :
223                         break loop2;
224                 }
225             } while (true);
226
227
228             }
229
230             retval.stop = input.LT(-1);
231
232         }
233         catch (RecognitionException re) {
234             reportError(re);
235             recover(input,re);
236         }
237         finally {
238         }
239         return retval;
240     }
241
242         ...
243
244     public static final BitSet FOLLOW_cabec_in_lavanda23 = new BitSet(new long[]{0x0000000000000040L});
245     public static final BitSet FOLLOW_sacos_in_lavanda25 = new BitSet(new long[]{0x0000000000000002L});
246     public static final BitSet FOLLOW_DATA_in_cabec44 = new BitSet(new long[]{0x0000000000000020L});
247     public static final BitSet FOLLOW_ID_in_cabec46 = new BitSet(new long[]{0x0000000000000002L});
248     public static final BitSet FOLLOW_saco_in_sacos70 = new BitSet(new long[]{0x0000000000000042L});
249         ...
250
251 }
```