Universidade do Minho
Escola de Engenharia
Departamento de Informática

Master Course in Computing Engineering

Pedro Miguel de Jesus Ventura Carvalho

Unfuzzying Fuzzy Parsing

Master dissertation

Supervised by: Pedro Rangel Henriques
Nuno Oliveira

Braga, January 14, 2015
ACKNOWLEDGEMENTS

Firstly, I would like to thank my supervisor Pedro Rangel Henriques and co-supervisor Nuno Oliveira. Without their constant support, guidance and friendship this dissertation would have not been possible. Secondly, I would like to thank my mother for the constant encouragement and belief throughout all my academic life.

To all my friends, a heartfelt thank you for always being there for me and for their constant support.

Lastly, I would like to dedicate this dissertation to my father. Faith had it in such a way that you could not be here to watch me conclude my studies, as you so dearly wanted. Wherever you are, I hope that you are proud. This is for you.
Recognizing sentences of a language in an efficient and precise manner has always been a strong subject within computer science. Many theories, algorithms and techniques have been proposed along computing history, but at the end it all comes down to performing lexical and syntactic analysis of the source, originating a parse tree as the result.

Sometimes there is no need for full precision or even a full parse tree. A good example of one of these cases is architecture extraction from source code. In this case only a small portion of the code is of interest. Another good example is recognizing handwritten expressions, because it is entirely impossible to predict the kind of calligraphy that will be analyzed, it is also impossible to perform an one hundred percent precise recognition. This need for tolerant parsing lead to the development of many forms of tolerant parsing along the years.

This master work will focus on one form of tolerant parsing in particular, Fuzzy Parsing. From this work it is expected the emergence of a new Fuzzy Parsing technique based on automata, where automata states would represent context and edges would represent potential matches inside that context. The hypothesis of this work is that such an approach reduces uncertainty and recognition time. It is also expected the creation of a tool suit that facilitates the process of developing fuzzy parsers. We believe that such a tool will be a great addition to areas such as Program Comprehension or IDE construction.
RESUMO

Reconhecer frases de uma linguagem, de um forma eficiente e precisa, tem sido sempre um tópico de interesse dentro da Informática. Diversas teorias, algoritmos e técnicas foram propostas ao longo dos anos, mas no fim de contas tudo se baseia em análises léxicas, sintáticas e semânticas da fonte, originando uma árvore de parsing como resultado.

Por vezes não há necessidade de um reconhecimento cem por cento preciso, nem de uma árvore de parsing completa. Um bom exemplo de um desses casos, pode ser encontrado na extracção de modelos arquiteturais a partir de código fonte. Neste caso, apenas uma parte reduzida do código tem interesse. Outro bom exemplo pode ser encontrado no reconhecimento de expressões matemáticas escritas manualmente. Uma vez que é impossível prever o tipo de caligrafia a ser analisada, é também impossível realizar um reconhecimento cem por cento preciso. Esta necessidade por tolerância no reconhecimento levou ao desenvolvimento de várias técnicas de parsing tolerante ao longo dos anos.

Esta dissertação irá focar-se particularmente numa forma de parsing tolerante, Fuzzy Parsing. Deste trabalho é esperado que surja uma nova técnica de Fuzzy Parsing baseada em autómatos, onde os estados representarão contexto e os arcos representarão possíveis correspondências dentro desse contexto. Espera-se que esta abordagem reduza a incerteza e o tempo de reconhecimento. É também esperado que seja criada uma ferramenta que facilite o processo de criação de outros fuzzy parsers. Pensamos que uma ferramenta do gênero será uma bela adição para áreas como análise de programas e construção de ambientes de desenvolvimento de software.
## CONTENTS

<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 INTRODUCTION</strong></td>
<td></td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>4</td>
</tr>
<tr>
<td>1.2 Objectives</td>
<td>5</td>
</tr>
<tr>
<td>1.3 Document Structure</td>
<td>5</td>
</tr>
<tr>
<td><strong>2 STATE OF THE ART</strong></td>
<td></td>
</tr>
<tr>
<td>2.1 Context-Free Grammars</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Robust Parsing</td>
<td>7</td>
</tr>
<tr>
<td>2.2.1 Island Grammars</td>
<td>8</td>
</tr>
<tr>
<td>2.2.2 Fuzzy Parsing</td>
<td>9</td>
</tr>
<tr>
<td>2.3 Fuzzy Parsing Implementations</td>
<td>10</td>
</tr>
<tr>
<td>2.3.1 Cocke-Younger-Kasami’s algorithm</td>
<td>10</td>
</tr>
<tr>
<td>2.3.2 Recursive Descent Recognizers</td>
<td>13</td>
</tr>
<tr>
<td>2.3.3 Shift Reduce Algorithm</td>
<td>15</td>
</tr>
<tr>
<td><strong>3 UNFUZZING FUZZY PARSING</strong></td>
<td></td>
</tr>
<tr>
<td>3.1 The Language Conception</td>
<td>18</td>
</tr>
<tr>
<td>3.1.1 Contexts and Constructs of interest</td>
<td>20</td>
</tr>
<tr>
<td>3.1.2 Semantic Actions</td>
<td>23</td>
</tr>
<tr>
<td>3.2 The Language’s Syntax</td>
<td>24</td>
</tr>
<tr>
<td>3.3 ANTLR</td>
<td>26</td>
</tr>
<tr>
<td>3.3.1 ANTLR’s Lexical Modes</td>
<td>27</td>
</tr>
<tr>
<td>3.4 The Solution Workflow</td>
<td>28</td>
</tr>
<tr>
<td>3.4.1 Preprocessor</td>
<td>29</td>
</tr>
<tr>
<td>3.4.2 Generator</td>
<td>37</td>
</tr>
<tr>
<td><strong>4 CASE STUDY</strong></td>
<td></td>
</tr>
<tr>
<td>4.1 The running environment</td>
<td>48</td>
</tr>
<tr>
<td>4.2 Data Domain</td>
<td>49</td>
</tr>
<tr>
<td>4.3 Using our tool</td>
<td>50</td>
</tr>
<tr>
<td><strong>5 CONCLUSIONS AND FUTURE WORK</strong></td>
<td></td>
</tr>
<tr>
<td>5.1 Conclusions</td>
<td>64</td>
</tr>
<tr>
<td>5.2 Prospect for future work</td>
<td>65</td>
</tr>
<tr>
<td><strong>A LISTINGS OF CODE</strong></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>69</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 1  Filling the bottom row ........................................ 11
Figure 2  Filling the first cell of the second row ..................... 12
Figure 3  The entire second row filled ................................. 12
Figure 4  The triangular table completely filled ..................... 12
Figure 5  Extension to the CYK functional algorithm ................. 13
Figure 6  Shift Reduce Parser ........................................... 15
Figure 7  Pushing and Popping of Contexts ............................. 20
Figure 8  Push transition to inner_class ............................... 21
Figure 9  Push transition to inner_class ............................... 21
Figure 10 Pop transition to class ...................................... 22
Figure 11 Pop transition to class ...................................... 22
Figure 12 Pop transition to class ...................................... 23
Figure 13 Pop transition to class ...................................... 23
Figure 14 ANTLR Lexer commands .................................... 27
Figure 15 First workflow approach .................................... 28
Figure 16 Reviewed workflow approach ................................. 28
Figure 17 Final step in the process .................................... 29
Figure 18 Inner brackets replaced by keywords ....................... 30
Figure 19 Modifications made in the spec file ....................... 36
Figure 20 Mapping the name of the grammar .......................... 38
Figure 21 Mapping the imports section ................................. 38
Figure 22 Mapping the methods section ............................... 39
Figure 23 Mapping a push transition ................................... 40
Figure 24 Mapping a pop transition ................................... 40
Figure 25 Mapping a pop push transition .............................. 40
Figure 26 Injecting whitespaces into a production .................... 40
Figure 27 Mapping our default context ............................... 41
Figure 28 Mapping a custom context .................................. 42
Figure 29 Mapping semantic actions ................................... 44
Figure 30 Complete translation ....................................... 46
Figure 31 Input file class diagram ...................................... 48
Figure 32 Data Domain Class Diagram ................................. 50
Figure 33 Passing the Fuzzy Grammar to the Preprocessor ........ 51
| Figure 34 | Passing the modified Fuzzy Grammar to the Generator | 51 |
| Figure 35 | The generated Fuzzy Parser | 52 |
| Figure 36 | Generating results | 52 |
| Figure 37 | Context Automaton after the definition of the default state | 54 |
| Figure 38 | Context Automaton after defining class header | 56 |
| Figure 39 | Context Automaton after defining interface header | 57 |
| Figure 40 | Automaton after class body definition | 60 |
| Figure 41 | Automaton after method args and method body contexts | 61 |
| Figure 42 | Final class diagram | 63 |
# List of Listings

2.1 Grammar for simple arithmetic expressions ........................................ 7
2.2 Base for Island Grammars ................................................................. 8
2.3 Adding an island for COBOL program calls ....................................... 9
2.4 Fuzzy Parser able to recognize COBOL calls ..................................... 10
2.5 Abstract context-free grammar for the CYK algorithm example ............. 11
2.6 Adding incorrect nonterminals to the grammar .................................. 14

3.1 Example of a Fuzzy Grammar in our notation .................................... 19
3.2 Defining a production ......................................................................... 20
3.3 Definition of two contexts ............................................................... 20
3.4 Definition a push transition ............................................................. 21
3.5 Definition a pop transition .............................................................. 21
3.6 Definition a push transition ............................................................. 22
3.7 Example of a semantic action ............................................................ 23
3.8 Definition of the 4 sections ............................................................. 24
3.9 Definition of a context .................................................................... 24
3.10 Example of a production inside a context ......................................... 24
3.11 Terminal, Nonterminal and Regex definitions ................................... 25
3.12 Context transition operations ......................................................... 25
3.13 Semantic Actions ............................................................................ 26
3.14 Structure of ANTLR’s Lexical Modes .............................................. 27
3.15 Reaching a semantic action ............................................................. 29
3.16 Reaching an opening bracket inside the Bracket context .................. 29
3.17 The rest of the text is mapped into bracket_body ................................ 30
3.18 Production to recognize public classes ............................................ 31
3.19 Method to return a subset of a string ............................................... 31
3.20 Production to recognize public classes ............................................ 31
3.21 Production to recognize public classes ............................................ 32
3.22 Production to recognize public classes ............................................ 32
3.23 Example of label usage .................................................................... 32
3.24 Invoking the variable expansion method ......................................... 32
3.25 Calculating the variable’s position .................................................. 33
3.26 Iterating through the production ..................................................... 34
INTRODUCTION

A common topic of interest in computer science is Parsing. Parsing is the process of recognizing sentences of a language so that some context or meaning can be inferred.

In our daily routine we are constantly in need to interpret all kinds of inputs like images, texts or sounds. When someone speaks to us our brain parses the information that is being received by giving it context and making it understandable. Another good example can be observed in visual languages such as traffic and road signs. In Computer Science that need is pretty much evident as computers need to recognize what they are being fed.

Concerning Classical Parsing Theory, Aho et al. (1986) the recognition process is done accordingly to the rules of a formal grammar. This process is divided in three different types of analysis (Lexical, Syntactic and Semantic). Lexical analysis consists in reading the input that is submitted and grouping the characters into meaningful sequences called lexemes then producing tokens based on those lexemes. In syntactic analysis a tree-like intermediate representation is built, based on the aforementioned tokens, called syntax tree. Finally in Semantic analysis, there is a check for semantic consistency between the source and the language definition (eg., type checking).

Having laid some background information about parsing, it is now important to highlight the deterministic nature of this endeavor. In Classical Parsing Theory a sentence either belongs or not to the language. This is, in fact, the basic membership concept found on Set Theory, Kechris (1995). Thus, if we consider a set $\Sigma$ of characters as an alphabet, and $\Sigma^*$ a set of strings (or sentences) over $\Sigma$ we can then derive the following membership function $\mu : \Sigma^* \rightarrow \{0, 1\}$, meaning that a sentence either belongs or not to set $\Sigma^*$. In plain words, this means that for parsing to be successful all sentences must be derived from the grammar following a structured set of rules without uncertainty, making classical parsing precise.

However in certain situations precise parsing is not the best option. Some common scenarios where this statement applies can be easily found in areas as diverse as IDE Construction Bischofberger (1992), Program Comprehension Murphy and Notkin (1996); Van Deursen and Kuipers (1999), Recognition of Handwritten Expressions Fitzgerald et al. (2006); MacLean and Labahn (2010); Fitzgerald et al. (2004) or even Speech Recognition Vidal et al. (1985). As we go deeper into these areas we can easily find practical examples where classical parsing is not suitable. In IDE Construction for example, code-completion features are a must have component; this requires parsing unfinished sentences, this is, recognizing sentences that are not in the language. In Program Comprehension
it is very common to extract high level information from source code (eg., extracting class relationships); in these situations only a small portion of the code is needed and generating a full parse tree is unnecessary work. Another good example comes from Handwritten Symbol Recognition (eg., mathematical formulas); because of the unpredictable nature of human calligraphy it is impossible to create a grammar containing all possible cases, making precise parsing in itself an impossibility.

This need for something that classical parsing can not provide was the basis for the emergence of Robust Parsing Theories. The main difference between this and Classical Parsing resides in the breaking of the do or die situation that previously existed, were a sentence either belonged or not to the language; and replacing that concept with a new one where an element may belong or not to the language. If previously we linked the deterministic nature of Classical Parsing to Set Theory we can now do the same for Robust Parsing and Fuzzy Set Theory, Zimmermann (2001, 2010). If we consider the same \( \Sigma^* \) set as before we can now derive a new membership function \( \eta : \Sigma^* \rightarrow [0,1] \). Note that the boolean set \( \{0,1\} \) has been replaced by the continuous interval \( [0,1] \). However, with this extension of dual logic comes a new layer of ambiguity or uncertainty. Although there are many variants of Robust Parsing this work will focus on Fuzzy Parsing.

A formal definition for fuzzy parsing is difficult to propose because fuzzy parsers and the strategies they employ depend on language- and application-specific properties Asveld (1995). The term fuzzy parsing was firstly introduced by Bischofberger (1992) and referred to a parser which had only a partial understanding of C++ that could deal with incomplete software systems that contained errors. Koppler (1996) presents a more precise definition for fuzzy parsing saying that it consists in the recognition of parts of a language by means of an unstructured set of rules. In this work we will simply employ the notion that fuzzy parsing consists on parsing source code sentences that can be incomplete or not totally valid according to the grammar language, but agreeing to a set of rules that, in a certain extent, partially represent the complete grammar. This kind of parsing produces a derivation tree with some uncertainty which may be measured on a \( [0,1] \) interval in the same sense as a probability. Asveld (1995) goes a step further by using this interval to categorize sentences according to their validity. That way he was able to create a scale that went from tiny mistakes to capital blunders.

1.1 MOTIVATION

As discussed above, there are many areas where Robust Parsing, and more specifically Fuzzy Parsing, deliver a better performance than Traditional Parsing. A simple example would be an application intended to attach code inspectors to functions, like Berón et al. (2007a,b); in such scenarios the only information relevant is for instance, the header of the function, or the beginning of a control structure, the remaining code may be irrelevant and should be ignored; That being said, it’s easy to understand that Fuzzy Parsing has a performance advantage, in comparison to Traditional Parsing, because parts of the function will be ignored, resulting in a smaller parse tree.
Another example where Fuzzy Parsing is a better option is the recognition of Handwritten Mathematical Expressions. Symbol recognition is arguably more difficult in mathematical expressions than in cursive text, as the symbol set is larger (over 100 symbols). As if this setback was not enough, we also have to consider the unpredictable nature of human calligraphy. Although we are talking about a common symbol set, every individual has its own quirks when writing those symbols. In fact, we can even say that the same individual may write the same symbol in more than one way. Not only the format of the symbol is important here, even the spacing between symbols has to be predicted. For all these reasons it is safe to say that precise parsing in this scenario is impossible and Fuzzy Logic is the best approach.

Unfortunately there are no specific tools for fuzzy parsing construction and one has to rely on compiler compilers Parr and Quong (1995); Johnson (1975); Aaby (2003), in particular, on the generated lexers (since fuzzy parsing is all about lexical analysis). But these traditional tools tend to generate lexical analyzers that are rather sequential, consuming more time than what would be desired for fuzzy parsing. Moreover, a great amount of recognition uncertainty will remain due to the lack of mechanisms suitable to reduce fuzzyness. Conditional recognition of regular expressions may reduce uncertainty, but may come at a price because with an increase in condition testing comes a decrease in performance. For these reasons fuzzy parsing has not, to this date, been as broadly adopted by the computing community as would be expected, the development of new fuzzy parsers is still very costly and consequently there is still a gap in this area that needs to be filled.

1.2 OBJECTIVES

In this work, we will start by proposing a new technique for Fuzzy Parsing based on automata, where automata states would represent context and edges would represent matches inside that context. It is our belief that this technique will reduce fuzzyness as well as increase performance.

We intend to create a tool suit that facilitates the process of creating Fuzzy Parsers, even abstracting the developer from the fuzzyness issue.

1.3 DOCUMENT STRUCTURE

This document can be divided in 5 chapters. After the introduction, chapter 2 explores the Robust Parsing world, with special focus in Fuzzy Parsing. Different techniques of Robust Parsing are presented, and multiple techniques for Fuzzy Parsing, in particular, are shown. In chapter 3 we start presenting our solution. A new formal language was created, along with a new workflow based on the ANTLR parsing engine. Chapter 4, presents a case study where we utilize our tool suit, in a tutorial fashion. In chapter 5 we can find our final remarks and prospects for future work.
STATE OF THE ART

In order to fully understand the problem at hand, we did a deep research in the robust parsing area. We searched for the main ideas behind the fuzzy parsing rationale, as well as state of the art implementations of robust parsing techniques.

To have a better grasp of robust parsing concepts it is necessary to have a strong understanding of traditional parsing techniques. That understanding allows us to make a better parallel between the those two worlds. For that reason, our research was also expanded to include the traditional parsing domain. Most of the main traditional parsing techniques were studied and that was the basis for the latter research for new robust parsing techniques.

In this chapter, we will present the most significant findings of our research. What we consider to be the current pinnacle of robust parsing techniques. Because this dissertation focuses on a subset of robust parsing (Fuzzy Parsing), this chapter will also focus on that subset.

The chapter starts with a brief recap of Context-Free Grammars. This concept is used in most parsing techniques, and so, understanding it is crucial in order to advance to new, more complex, parsing notions.

We will follow it with what we found to be the two more significant robust parsing techniques, Island Grammars and Fuzzy Parsing. Finally, three state of the art implementations of Fuzzy Parsing are presented, Cocke-Younger-Kasami, Recursive Descent and Shift Reduce.

2.1 CONTEXT-FREE GRAMMARS

A context-free grammar, firstly introduced by Chomsky (1956), is a set of production rules that syntactically derive sentences from a formal language. The rules describe how to form sentences from the language’s alphabet that are valid according to the language’s syntax. A grammar is considered context free when its production rules can be applied regardless of the context of a nonterminal.

Formally, a context-free grammar $G$ is defined by the 4-tuple $G = (V, \Sigma, R, S)$ where:

- $V$ is a set of nonterminals, which are syntactic variables denoting sets of sentences that can be derived by successive applications of the production rules.
• Σ is the alphabet of the language, and each symbol σ in Σ is called a terminal (or a token). Terminals are the basic immutable symbols of the grammar (i.e., they are never rewritten by production rules), and together form the sentences of the language.

• R is a set of production rules that specify the concrete way in which terminals and nonterminals may be combined to form sentences. A production rule p is formally given as a rewrite rule of the form \( V \rightarrow (V \cup E)^* \), meaning that the left-hand side (LHS) symbol (a nonterminal) derives into a sub-language given by the symbols in the right hand side (RHS). The latter may be the empty set or any combination of symbols form V or E.

• \( S \in V \) is the start symbol (or axiom).

In listing 2.1 we can see a simple context-free grammar for arithmetic expressions. In this grammar the terminals are id, +, -, *, /, (, ) and the nonterminals are expression (also the start symbol), term and factor.

<table>
<thead>
<tr>
<th>expression</th>
<th>: expression '+' term</th>
<th>expression '-' term</th>
<th>term</th>
</tr>
</thead>
<tbody>
<tr>
<td>term</td>
<td>: term '*' factor</td>
<td>term '/' factor</td>
<td>factor</td>
</tr>
<tr>
<td>factor</td>
<td>: '(' factor ')'</td>
<td>id</td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>: [A-Z][A-Z0-9] *</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Listing 2.1: Grammar for simple arithmetic expressions

2.2 ROBUST PARSING

We have already discussed some of the benefits that can be obtained by utilizing robust parsing. If we start looking for documented cases where some form of robust parsing has been applied we reach the conclusion that there still isn’t a de facto standard for it. Lexical approaches are usually the way to go, software developers usually create a set of regular expressions that are intended to match the desired constructs of interest, such approaches can be observed in Murphy and Notkin (1996) or Wong et al. (1995). However, there are two techniques that can be considered as the leading standards on the area, these are Island Grammars and Fuzzy Parsing. These two concepts will be detailed in the following sections.
2.2.1 Island Grammars

An Island Grammar is a Grammar that consists of two distinct parts: (i) detailed productions that describe certain constructs of interest (so called islands), and (ii) liberal productions intended to catch the rest of the input (so called water).

From a formal point of view, given a language \( L_0 \), a context-free grammar \( G = (V, \Sigma, R, S) \) such that \( L(G) = L_0 \) and a set of constructs of interest \( I \subseteq \Sigma^* \) such that \( \forall i \in I \colon \exists s_1, s_2 \in \Sigma^* : s_1 is_2 \in L(G) \). An Island Grammar \( G_I = (V_I, \Sigma_I, R_I, S_I) \) for \( L_0 \) has the following properties:

1. \( L(G) \subset L(G_I) \)
   
   \( G_I \) generates an extension of \( L(G) \).

2. \( \forall i \in I \colon \exists v \in V_I : v \rightarrow i \land \exists s_3, s_4 \in \Sigma^* : s_3 is_4 \not\in L(G) \land s_3 is_4 \in L(G_I) \)

   \( G_I \) can recognize constructs of interest from \( I \) in at least one sentence that is not recognized by \( G \).

In listing 2.2 we can see the core specification for an Island Grammar. We start by defining our start symbol input that may derive in a sequence of chunks. All chunks derive in water, this means that at this phase we don’t have any constructs of interest.

```plaintext
input
  : chunk*

chunk
  : water

water
  : .*
```

Listing 2.2: Base for Island Grammars

The grammar here specified is a very robust one, in the sense that all sentences will be recognized. However, it is almost useless as we can not distinguish the parsed symbols.

We can turn this into a useful grammar by adding specifications of constructs of interest, in other words, adding islands. In listing 2.3 we add such an island by specifying that a chunk can also derive in an island; furthermore we specify that island derives in the literal ‘Call’ followed by an identifier; identifiers are sequences of alphanumeric characters starting with capital letters.
Incredibly or not, this grammar is able to recognize function calls in COBOL source code. In Program Comprehension areas such grammars may be useful as one could, for example, generate a system call diagram from it, while ignoring the majority of the code. Depending on the complexity of the task at hand, other islands may be added, producing an almost tailored made parser.

2.2.2 Fuzzy Parsing

The term Fuzzy Parsing is semi-formally defined in Koppler (1996). A fuzzy parser is a parser that is able to recognize only parts of a language according to some kind of specification, or set of rules, that are established by the programmer. It ignores the input that is being consumed until it reaches a mark that symbolizes the start of a substring that is meant to be recognized. These special inputs are commonly referred as anchors.

From a more formal stand, considering a context-free grammar $G = (V, \Sigma, R, S)$, as introduced in Section 2.1, a fuzzy parser $F(G)$ is defined as $G = (V', \Sigma, R', A)$, where $V'$ is a set of anchor nonterminals, $\Sigma$ remains the alphabet of the language, $R'$ is the provided rule set and $A \subseteq \Sigma$ is the set of anchor symbols.

As explained before, an anchor flags the beginning of a substring that is meant to be recognized by $F(G)$; this means each $s \in L(G)$ contains at least one anchor that is accepted by $F(G)$. For each anchor there is a production rule $r_a \in R'$ that specifies the substring of $s$ that is meant to be recognized.

In conclusion, the language $L_F(G)$ that is partially accepted by $F(G)$ can be described as follows: $L_F(G) = \{ s \in L(G) : s = \omega_1 a \omega_2 \land \omega_1 \in \Sigma^* \land \omega_2 \in \Sigma^* \land a \in A \}$ and $s$ has been derived by the application of a production rule $r \in R$.

The example in listing 2.3 can be re-engineered to conform to this notion of Fuzzy Parsing:

```
input
  : chunk*
chunk
  : island | water
island
  : 'Call' id
water
  : .*
id
  : [A-Z][A-Z0-9]*
```

Listing 2.3: Adding an island for COBOL program calls
A clear distinction between what is recognized by the fuzzy parser and what is ignored (the text matching the trash nonterminal) was made only to facilitate the understanding of this concept; in fact, a fuzzy parser does not need this explicit specification, as it will only recognize the substrings matching the rules provided by the programmer. This grammar, although again very simple, is able to identify function calls in COBOL source code. To precisely follow the fuzzy-grammar specification, as introduced in this section, it would only be required to define the call rule. In this rule, the anchor is `Call` and `id` is only part of the sentence to match. Notice that `id` is defined as a regular expression, and thus it shall not be regarded as a nonterminal anchor.

### 2.3 Fuzzy Parsing Implementations

In this section state-of-the-art fuzzy-parsing algorithms are presented. It is shown how they are adapted to a fuzzy context, from common parsing algorithms. In essence, three algorithms are addressed: Cocke-Younger-Kasami; Recursive-Descent and Shift-Reduce. Asveld (1995, 2005a); Fitzgerald et al. (2006)

#### 2.3.1 Cocke-Younger-Kasami's algorithm

Cocke-Younger-Kasami’s (CYK) algorithm intends to validate the presence of a word in a context-free grammar. In other words, considering a string $s$ and a context-free grammar $G = (V, \Sigma, R, S)$, introduced in Section 2.1, CYK verifies whether $s \in L(G)$ by using a dynamic programming or table

```
input
  : chunk*
chunk
  : anchor | trash
anchor
  : call
trash
  : .*
call
  : 'Call' id
id
  : [A-Z][A-Z0-9]*
```

Listing 2.4: Fuzzy Parser able to recognize COBOL calls
**filling algorithm** approach. CYK starts by constructing a triangular table where each row correspond to the length of a substring; the bottom row corresponds to substrings of length 1 and the top row corresponds to strings of length of \( s \).

As an example, consider the string \( s = baaba \) and the following grammar:

\[
\begin{align*}
S & : AB | BC \\
A & : BA | a \\
B & : CC | b \\
C & : AB | a
\end{align*}
\]

Listing 2.5: Abstract context-free grammar for the CYK algorithm example

The CYK algorithm starts by calculating the bottom row. Each cell of this bottom row is assigned a production rule that may derive directly into each symbol \( \sigma \) of \( s \), where \( \sigma \in \Sigma \). The following shows this first step. Notice that, since symbol \( a \) can be derived directly from rules A and C, then both rules are associated to a single cell.

![Figure 1.: Filling the bottom row](image)

The subsequent steps are concerned with filling the top rows of the triangular table, by taking substrings of \( s \), successively bigger. For instance, in the second step, cells of the second row are filled with nonterminals that allow the derivation of substrings of length 2. For the first cell, the first substring of length 2, \( ba \), is considered. From the first row, it is known that \( b \) is derived from B and \( a \) from A or C. This means that to derive \( ba \), sequences BA or BC of nonterminals shall be consumed. Finally, these sequences can be derived from rules S and A.
For the second cell, it is considered the second substring of length 2: \(aa\). Figure 3 the same approach, \(aa\) could be obtained by AA, AC, CA or CC sequences of nonterminals. Since only CC is a recognizable sequence, than the second cell is associated with rule B (that derives in CC). The remainder of the row follows the same rationale. The following shows the result of the second step.

By applying CYK algorithm until reaching a substring of length \(s\) (this is, \(s\) itself), a table like the one showed in figure 14 is obtained.

Because the start symbol \(S\) is one of nonterminals on the top cell we can say that \(s \in L(G)\) with all certainty.

After observing the way CYK performs, we can now see how can this algorithm be adapted to a fuzzy context.

In Asveld (2005a) the author proposes a functional version of this algorithm that reads as follows.

Let \(G = (V, \Sigma, R, S)\) be a context-free grammar in CNF and let \(s\) be a nonempty string over \(\Sigma\), also let \(f(\omega)\) be the initialization phase (the filling of the first row), and \(g(\omega)\) be the iteration phase (the filling of the following rows). Compute \(g(f(s))\) and verify if \(S\) belongs to \(g(f(s))\). We have
s ∈ L(G) only if S ∈ g(f(s)). Note that Asveld substituted the normal iterative approach with a recursive one.

Asveld then proceeds to build on this functional algorithm adapting it to fuzzy context-free languages. In Asveld (2005b), the notions of fuzzy context-free languages and fuzzy context-free grammars are introduced in a very detailed manner. Roughly speaking, the notions we can find there are the same that have already been discussed in the previous sections of this dissertation, more specifically the may belong to the language concept that rises up some ambiguity. However, it is important to note that while a regular context-free grammar environment, $L(G)$ is defined by using the concept of derivation, in a fuzzy context-free grammar framework, $L(G)$ is defined in terms of set-theoretical operations. These operations and their definitions are, as one can imagine, quite extensive and can be found in Asveld (2005b).

That being said, Asveld extends the functional algorithm by adding the following four new rules, regarding the computation of the membership function $\mu$ of the input.

\[
\begin{align*}
\mu(A; f(a)) &= \mu(a; P(A)), \\
\mu(A; g(A)) &= 1 \quad (A \in N), \\
\mu(A; g(\omega)) &= \sqrt{\{\mu(A; g(\chi) \otimes g(\eta)) \mid \chi, \eta \in N^+, \omega = \chi\eta\}} \quad (\omega \in N^+, |\omega| \geq 2), \\
\mu(A; X \otimes Y) &= \sqrt{\{\mu(B; P(A)) \ast \mu(B; X) \ast \mu(C; Y) \mid B, C \in N\}},
\end{align*}
\]

Figure 5.: Extension to the CYK functional algorithm

As we can see, by utilizing set theoretical operations like Kleen closure, Asveld was able to calculate the membership function of each string, and also fill the table with such information. A practical example of this algorithm in action can be found in Asveld (2005a).

2.3.2 Recursive Descent Recognizers

CYK is a bottom-up algorithm for recognizing context-free languages. In Asveld (1993) a top-down alternative is presented.

First, it is necessary to associate to each nonterminal $v$ of the grammar a rule $v'$ (application-specific) defined for inputs $\sigma \in \Sigma^*$ and designating a set (possibly empty) $\sigma' \subseteq \Sigma$ as result of consuming $\sigma$. Then the algorithm takes an input string $s$ and pursues all possible paths beginning from the axiom of the grammar (i.e., assuming all the optional productions of the axiom). It then proceeds to break the input into substrings of lengths from $n - 1$ to 1, considering length of $s$ is $n$. Then, for each production $p$, the rule $v_1'$ of its first nonterminal, $v_1$, takes each substring $s'$ and the rule $v_2'$ of the second nonterminal, $v_2$, takes the reminder of $s$ (i.e., $s - s'$). The results of each $v_i'$ are joined with the usual set union operation. After expanding to all possible paths, it starts consuming the tokens, according to the rules associated to the nonterminals, trying to reduce the input to the empty string $\lambda$.

For a concrete example, let’s consider the grammar presented in 2.5 and an input string $s = baa$. Applying the rationale explained before the algorithm develops as follows:
\[ S(baa) = A'(ba)B'(a) \cup A'(b)B'(aa) \cup B'(ba)C'(a) \cup B'(b)C'(aa) \]

Now we have to define how do we reduce this input, so let's add the following rules:

- \( A'(\bot) = \emptyset \) where the symbol \( \bot \) means \textit{undefined}.
- \( A'(\lambda) = \{ \lambda \} \)
- \( A'(x) = \{ \lambda \} \) when \( x \in \Sigma \) and \( |x| = 1 \)
- \( A'(x) = \emptyset \) when \( x \in \Sigma \) and \( |x| \geq 2 \)
- \( A . \emptyset = A \)

Now continuing our derivation:

\[
... = \emptyset . \emptyset \cup \emptyset . \emptyset \cup \emptyset . C'(a) \cup B'(b) . \emptyset \\
= C'(a) \cup B'(b) \\
\{\lambda\} \cup \{\lambda\} = \{\lambda\}
\]

Having reached the conclusion that \( \lambda \in S(s) \) we can say with certainty \( s \in L(G) \).

To adapt this algorithm to a fuzzy context we need to somehow define the acceptance value for each possible acceptable error. We also need to relate these acceptance values, in this example let’s consider that the number of mistakes that are made is irrelevant. That being said, the acceptance value that will be considered will be the lowest one, in other words, the degree of confidence that one can assign to input \( s \) is equal to the biggest error that is encountered.

Considering our previous example, let’s add two \textit{incorrect} nonterminals to our production \( S \), and let’s say their acceptance values are 0.9 and 0.1, let’s also assume that all other nonterminals have 1 as their acceptance value.

<table>
<thead>
<tr>
<th>Production</th>
<th>Acceptance Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
<td>( AB \mid BC \mid AA/0.9 \mid BB/0.8 )</td>
</tr>
<tr>
<td>( A )</td>
<td>( BA \mid a )</td>
</tr>
<tr>
<td>( B )</td>
<td>( CC \mid b )</td>
</tr>
<tr>
<td>( C )</td>
<td>( AB \mid a )</td>
</tr>
</tbody>
</table>

Listing 2.6: Adding incorrect nonterminals to the grammar
So if we apply this new rationale, the algorithm develops as follows:

\[
S(baa) = A'(ba)B'(a) ∪ A'(b)B'(aa) ∪ B'(ba)C'(a) \\
∪ B'(b)C'(aa) ∪ A'(ba)A'(a)[0.9] ∪ A'(b)A'(aa)[0.9] ∪ B'(ba)B'(a)[0.8] ∪ B'(b)B'(ba)[0.8] \\
= ∅.∅ ∪ ∅.∅ ∪ ∅.∅ ∪ ∅.∅ ∪ ∅.∅ ∪ ∅.∅ ∪ ∅.∅ ∪ ∅.∅ ∪ ∅.∅ ∪ ∅.∅ ∪ ∅.∅ ∪ ∅.∅ \\
= C'(a) ∪ B'(b) ∪ A'(a)[0.9] ∪ B'(b)[0.8] \\
= \{λ\} ∪ \{λ\} ∪ \{λ\}[0.9] ∪ \{λ\}[0.8] = \{λ\}
\]

Considering all the nonterminals that derived into \(λ\), we now calculate the minimum possible acceptance value for them (not forgetting the implicit 1 value to all correct productions). That value is the degree of confidence that one can say that \(s ∈ L(G)\), in this case \(μ(s, L(G)) = min(1, 1, 0.9, 0.8) = 0.8\).

2.3.3 Shift Reduce Algorithm

Shift Reduce Parsing (SRP) uses the input string \(w\) to the start symbol of a grammar \(G\). Additionally to this, SRP also uses a parsing table and a stack; the parsing table calculates the next action in a deterministic manner, according to the current state and the next input; the parser then shifts a state and the new symbol onto the stack until the right side of a grammar production \(A → B\) is on top. A reduction will then take place, as \(B\) is then replaced on the stack by the nonterminal \(A\) and a state chosen by a predefined function (lets call it goto). The production \(A → B\) is then added to the parse tree. This process is then repeated until \(w\) has been reduced to the start symbol of \(G\) or if an error occurs.

In Fitzgerald et al. (2006) a new approach is proposed, based on attribute grammars. Attribute Grammars were firstly introduced in Knuth (1968) and consist in a context-free grammar extended
with attributes for the nonterminals and semantic rules for the productions. By using attributes, each
token in the input can be represented as an object with attributes, in the traditional sense. In the work
where this approach was introduced, attributes were used to store the token location and it’s identity
associated with a fuzzy likelihood. However, other kinds of attributes can be explored, depending on
the task at hand.

Another notion that is introduced, is the notion of a fuzzy constraint, this is in fact, the same concept
of the membership function introduced in previous sections. Whenever a reduction occurs according
to a production $p_i$ a function is applied, receiving as arguments the attribute values of the tokens on
the right side of $p_i$, and returning a value representing the acceptability of those attribute values.

The use of attributes and constraints introduced a new layer of information to the SRP framework.
This lead to Fuzzy Shift Reduced Parsing (FSRP), which explores multiple parses whenever strong
ambiguities arise. Of course the sheer possibility of analyzing various paths makes for a more certain
decision, but a balance must be stroke as multiple parses mean lower efficiency.

Initially there will only be one parse, but when a strong ambiguity arises during a parse $P_i$, $P_i$ is
copied to a new parse $P_z$. The most likely path is then pursued in $P_i$ firstly and the least likely path
is pursued in $P_z$ later. A function $\text{worthExploring()}$ has to be defined, to determine whether the least
likely option should be pursued. This function is defined according to the task at hand.

When all parses are complete, the most likely parse tree is selected according to a predefined metric.
This metric is application specific, but usually takes into account the attribute values as well as the
results returned by the constraints.
As introduced in the previous chapter, it is our intention to create a new Fuzzy Parsing technique, as well as a tool suit that enables a faster and easier creation of fuzzy parsers.

Our approach springs from fuzzy grammars and it is intended to precisely recognize constructs of interest of some language, while dealing with the fuzziness (incompleteness, incorrectness, etc.) of the inputs. We do this by adding the notion of contexts that unfuzzy the parsing and reduce the search space for constructs of interest.

The approach is based on deterministic finite automata notions. The rationale is to recognize constructs of interest defined within specific contexts of a language. Anchors and associated rules define the syntax of such constructs of interest and their recognition may or may not define transitions between contexts. To be precise, in a context, any sequence of characters is consumed and ignored unless a rule for such a sequence is defined. The normal behavior after matching a rule is to remain in the same context (defining a loop transition in the automaton); the exception is to transit to a different context.

Contexts are (final) states in an automaton with particular notion of hierarchy. This means that a context is inside other contexts, facilitating transitions from a child context to its parent, when explicit transitions are not suitable. In a sense, the overall behavior of the context space can be regarded as a non-linear stack machine, where pushing contexts is as usual, but popping contexts may depend on the existence of a transition (defined in the respective automaton) to a deeper context in the stack. Bare in mind that this is different to backtracking in parsing strategies: it is intended behavior jumping from contexts to contexts.

Because our approach requires the generation of a deterministic automaton, ambiguity will not be an issue (as in many fuzzy parsing approaches). Consequently, a unique parsing tree is produced that only contemplates the recognized constructs of interest (it is not a full parsing tree). Considering these claims, performance gains seem to be evident.

Not only was a new technique devised, we also created a language to support it. Developers are able to specify contexts, transitions and semantic behavior in a more high level and user friendly way. From this specification a new Fuzzy Parser will be generated.
3.1 THE LANGUAGE CONCEPTION

As previously discussed, there still is no *de facto* standard to implement Fuzzy Parsing solutions. As so, it was our intention to create an easy and lightweight way to specify a Fuzzy Parser from scratch. Building on our proposed rationale, we identified 2 main activities that where the basis for our language.

- Being able to define contexts;
- Being able to define constructs of interest inside those contexts;

A first attempt to create such a language was informally introduced in Carvalho et al. (2014) and was the basis for the current state of the language.

But defining contexts and constructs of interest clearly isn’t enough. We also wanted this language to offer the main functionalities that one can encounter in the compiler compilers that are usually out there.

The language is divided in 4 distinct sections:

- A mandatory section where one must define a name for our Fuzzy Grammar;
- An optional section where one can define files to be imported in the final product;
- An optional section where one can define custom methods to be used in the final product;
- The core section where one can define contexts and constructs of interest;

The keyword **BEGIN** was chosen to identify the beginning of each section, and the keyword **END** was designated to identify the end of each section. We can see an example on the following listing.
The first section starts with the keyword BEGIN NAME, end with END NAME, and defines the name of the Fuzzy Parser that is being produced.

The second section begins with the keyword BEGIN IMPORTS, ends with END IMPORTS, and defines a section where external modules can be loaded into the final Fuzzy Parser. In the example above we can see two Java classes being imported.

The third section begins with BEGIN METHODS, ends with END METHODS, and defines a section where developers can create custom methods that will be mapped to the final Fuzzy Parser.

As already pointed out before, the core of the language specification resides in the definition of contexts. The section reserved for context definitions starts with the keyword BEGIN CONTEXTS, ends with END CONTEXTS, and it’s content will be detailed in the following section.
3.1.1 Contexts and Constructs of interest

Contexts are defined with the symbol @ plus the intended name of that context. If we wanted to define a context named **ExampleContext** we would write `@ExampleContext`. When we define a context, all the productions that follow it are limited to the scope of that context.

We can define constructs of interest just like in any other grammar. We just declare a production, and define inside that production what we want to recognize, just like in the example below:

```plaintext
class: "class";
```

Listing 3.2: Defining a production

The key difference here is that if we are defining constructs of interest inside the scope of a context, those constructs of interest will only be recognized when the parser is inside that context. In the listing below we can see two contexts, each with two productions defined inside of them. If the parser is currently in the `class` context then it will only recognize the constructs of interest defined in the `public_class` and `private_class` productions. If a public or private method appears, it will be ignored.

```plaintext
@class
public_class: 'public class' ID;
private_class: 'private class' ID;

@method
public_method: 'public' TYPE ID '(' ARGS ')' ;
private_method: 'private' TYPE ID '(' ARGS ')' ;
```

Listing 3.3: Definition of two contexts

The rationale behind our context concept is based on a Stack Machine. We start with a default context (A mandatory context) and when we reach an anchor that identifies the start of a new context we push that context into the stack. In the other direction, when we reach an anchor that identifies the end of the current state, we pop it from the stack.

![Figure 7.: Pushing and Popping of Contexts](image-url)
Three types of operators where reserved to identify transitions between contexts:
The $\gg$ operator identifies a transition from context A to context B, **maintaining** context A in the stack (Push B).

```plaintext
@class
public_class: 'public class' ID;
enter_class: '{' $\gg$ inner_class;

@inner_class
//constructs of interest here
```

Listing 3.4: Definition a push transition

In the listing above, we can see that inside our `class` context a new anchor was defined (`{`), whenever this anchor is recognized the parser pushes the `inner_class` context into the stack, changing the current context of the recognition process.

![Push transition to inner_class](image)

Figure 8.: Push transition to inner_class

And in our context stack a new context is pushed.

![Push transition to inner_class](image)

Figure 9.: Push transition to inner_class

The $\wedge$ operator identifies a transition to the parent context in the stack order, **removing** the current context from the stack (Pop A).

```plaintext
@class
public_class: ‘public class’ ID;
enter_method: ‘public’ TYPE ID ’(’ ARGS ’)’ {’ $\gg$ method;

@method
exit_method: ’)’ $\wedge$;
```

Listing 3.5: Definition a pop transition
In the listing above, we can see that a push transition was defined from a class context to a method context whenever a method header is reached. Also, in our method context a pop transition is defined whenever a ’}’ anchor is reached. This means that whenever the parser is in the method context and recognizes a ’}’ symbol, it will pop the method context from the stack and shift its parent context onto the top of the stack, meaning it will be now the new current context. Of course this is just a simple example to illustrate this rationale, as we assume that the only time a ’}’ symbol appears on a method definition will be to close it.

![Figure 10.: Pop transition to class](image)

And of course our stack is updated.

![Figure 11.: Pop transition to class](image)

The `>` operator identifies a transition from context A to context B, removing context A from the stack (Pop A + Push B).

```plaintext
@class
public_class: 'public class' ID;
enter_class: '{' `>` inner_class;

@inner_class
//constructs of interest here
```

Listing 3.6: Definition a push transition
This is in fact a composition of the two former transitions. Considering the example above, whenever we are in the class context and reach a '{' anchor a pop and a push with the new inner_class context is made onto the stack. In practical terms, this is a direct replacement of the stack head.

In our automaton notion this transition is equal to the one in 3.1.1.

![Figure 12: Pop transition to class](image)

But we have a different behavior in our stack.

![Figure 13: Pop transition to class](image)

3.1.2 **Semantic Actions**

In every parser generator tools, semantic actions are a key feature. A semantic action block is contained inside brackets.

```plaintext
BEGIN NAME
Print_Class
END NAME

BEGIN CONTEXTS

@default
ID: [A-Z][a-zA-Z]*;
class: "public class" >> class_header;

@class_header
c1_name: name1=ID name2=ID ^
{  //Semantic action block here
   System.out.println($name1.text + $name2.text);
};

END CONTEXTS
```

Listing 3.7: Example of a semantic action
3.2 THE LANGUAGE’S SYNTAX

In the previous section we made a more high level approach to the language that was created. In this section a more formal presentation will be made, regarding it’s formal grammar.

As we said before, this language has 4 distinct sections:

```
spec: GRAMMAR_NAME IMPORTS? METHODS? context+

GRAMMAR_NAME: ‘BEGIN NAME’ [A-Za-z0-9_]+ ‘END NAME’;

IMPORTS: ‘BEGIN IMPORTS’ .*? ‘END IMPORTS’;

METHODS: ‘BEGIN METHODS’ .*? ‘END METHODS’;

context: ‘BEGIN CONTEXTS’ CONTEXT_ID production+ ‘END CONTEXTS’;
```

Listing 3.8: Definition of the 4 sections

Each section starts with the keyword BEGIN plus the name of the section, and ends with the keyword END plus the name of the section. In the axiom of the grammar we see the first reference to contexts, specifying that multiple contexts may be present.

A context is defined by it’s ID and it’s productions. Inside a production we can define our constructs of interest and the procedures we want to apply on them.

```
CONTEXT_ID: '@'[A-Za-z0-9_]+;

production: PRODUCTION_ID (ACTION | operation | terminal | nonterminal | REGEX)+ SEMI_COLLON

PRODUCTION_ID: [A-Za-z_]+ ‘:’;

SEMI_COLLON: ‘;’;
```

Listing 3.9: Definition of a context

A CONTEXT_ID starts with the symbol @ and it’s name. A production starts with a PRODUCTION_ID and ends with a SEMI_COLLON.

```
@my_context

my_production: ... ;
```

Listing 3.10: Example of a production inside a context
Inside a production we can find:

- Semantic Actions;
- Context transition operations;
- Terminal symbols;
- Nonterminal symbols;
- Regular Expressions;

Starting with the simpler ones:

```
terminal: STRING;
nonterminal: ID;
REGEX: [\+\-\*\^\(\)\[\]\{\}\A-Za-z0-9\\\'^]+;
STRING: '"' (\"')* '"';
ID: [A-Za-z_]+;
```

Listing 3.11: Terminal, Nonterminal and Regex definitions

Terminal symbols will be literals enclosed in quotes ("). Nonterminal symbols will be defined by their ID. And Regular Expressions will permit the use of special characters.

There are three context transition operations:

```
operation: PUSH ID | MODE ID | POP;
PUSH: '>>';
MODE: '^>';
POP: '^';;
```

Listing 3.12: Context transition operations

As detailed in 3.1.1, we can see that:

- **PUSH ID** defines a push transition to context ID;
- **POP** defines a pop transition to the parent context;
- **MODE ID** defines a push + pop transition to context ID;

  Semantic actions start with a `{` symbol and end with a `}` symbol.

  ```latex
  ACTION: `({.*?});`
  ```

  Listing 3.13: Semantic Actions

  Inside a semantic action block other brackets may appear, something that is not covered by our ACTION production. That issue is solved by pre-processing the input file and replacing any brackets that appear inside a semantic action block by a predefined keyword. That process will be detailed in section 3.4.1.

  In the next sections we will see how, building on this Fuzzy Grammar specification language, we can obtain our Stack like context driven parsing engine.

3.3 **ANTLR**

ANTLR *Parr (2013); Parr and Quong (1995) (Another Tool for Language Recognition)* is a powerful parser generator intended for reading, processing, executing or translating structured text or binary files.

ANTLR takes as input a grammar that specifies a formal language and generates as output source code for recognizing that language. ANTLR allows generating *lexers, parsers, tree parsers*, and combined *lexer-parsers*.

When searching for applications that would allow for a faster implementation of our Fuzzy Parsing idea, we discovered that ANTLR’s *lexer grammars* had a feature, called Lexical Modes, that could translate our ideas of contexts and reduced search space.
3.3.1 **ANTLR’s Lexical Modes**

Modes allow us to group rules inside contexts, which is exactly the basis for the technique we devised. The *lexer* can only recognize rules inside the current mode. *Lexers* start in the default mode and are treated like a stack, with pushing and popping actions.

```
rules in lexical mode;
...
mode MODE1;
rules in MODE1;
...
mode MODE2;
rules in MODE2;
...
```

Listing 3.14: Structure of ANTLR’s Lexical Modes

ANTLR supports Lexer commands, these commands follow specific syntax and are limited to a few common commands. Three commands are especially important for our implementation. The `pushMode(MODE1)` command inserts MODE1 at the head of the stack, the `popMode` command removes the head of the stack and the `mode(MODE1)` command changes the head of the stack to MODE1 (the same as `popMode` immediately followed by `pushMode(MODE1)`).

![ANTLR Lexer commands](image)

Figure 14.: ANTLR Lexer commands

It’s easy to see that ANTLR’S Lexical Modes already provide us with the tools to implement our context based rationale. For that reason, it was our choice to make a direct translation from our Fuzzy Grammar specification to an ANTLR Lexer Grammar, using the Lexical Modes feature. In other words, ANTLR’S Lexical Modes are the engine that will propel our idea.
3.4  THE SOLUTION WORKFLOW

Having our specification language defined and having found out that ANTLR’s Lexical Modes had the desired behavior to implement our Fuzzy Parsing rationale, the next step was to map our specification language to ANTLR’s syntax.

Because both languages have many similarities, the translation seemed pretty much straightforward and so our first workflow approach was to pass our specification file through a generator. This generator would, from our specification language, create a new ANTLR Lexer Grammar, utilizing ANTLR’s Lexical Modes. From this Lexer Grammar, a new Parser would be generated. Due to the nature of the Lexer Grammar created, this generated parser would be a Fuzzy Parser.

Figure 15.: First workflow approach

While implementing the translation step between the spec file and the lexer grammar, some technical problems lead us to believe that in order to ease this translation, a preprocessment of the spec file would be necessary (the reasons for this extra step will be detailed in 3.4.1). With this in mind, a new step was added to our workflow.

Figure 16.: Reviewed workflow approach
To the user this extra step is invisible. He only needs to define a specification file like in the first approach.

Having our final Fuzzy Parser generated, the user can then feed it an input file, just like in any regular parsing tool.

![Diagram showing the process](image)

**Figure 17.: Final step in the process**

Although ANTLR’s Lexical modes revealed themselves powerful enough to implement our idea, we feel that this layer of abstraction created with our specification language benefits the developer, as he now has a less complex way to create a Fuzzy Parser. More than that, because ANTLR’s Lexical Grammar operates at a lexical level (just like it’s name indicates) developers could not operate directly over the token values. Our tool abstracts the user of this limitation just like we will see in 3.4.2.

### 3.4.1 Preprocessor

As we addressed before, while implementing our translator to ANTLR’s Lexical Modes, we stumbled upon some issues. First of all, we needed a way to recognize `{ and `}` symbols inside semantic actions. Our solution was to create a Lexer Grammar that used ANTLR’s Lexical Modes to identify a semantic action context.

Whenever we are outside the Bracket context and a `{` appears we jump to the Bracket context and initialize a bracket counter, and an empty string that will be filled with the content inside the brackets.

```java
//OUTSIDE THE BRACKET CONTEXT
BRACKET: '{' { bracket_body = ""; bracket_counter = 1; pushMode(BRACKETS); };}
```

**Listing 3.15: Reaching a semantic action**

Inside the Bracket context whenever a `{ symbol appears, our bracket counter is incremented, and we inject the bracket_body string with the keyword __BRACKETO.**

```java
mode BRACKETS;
OPEN: '{'
{  
    bracket_counter++;  
    bracket_body += "__BRACKETO";
}
```

**Listing 3.16: Reaching an opening bracket inside the Bracket context**
When a } symbol is reached we decrement the counter and check if it has reached 0. If it has, then it is the bracket that closes the semantic action block, and so we have to pop context. Otherwise it is just an ordinary bracket inside the semantic action block, and we inject the keyword __BRACKETC. The rest of the text will be directly mapped into the final string.

```java
OPEN: '{'
{
    bracket_counter++;  
    bracket_body += "__BRACKETO";
};

CLOSE: '}'
{
    bracket_counter--;  
    if (bracket_counter == 0)  
    {
        bracket_body += '}';
        popMode();
    } ...
    else  
    {
        bracket_body += "__BRACKETC";
    };

TEXT: .
{
    bracket_body += getText();
};
```

Listing 3.17: The rest of the text is mapped into bracket_body

The final result of this transformation is:

```
@class
public: public class' ID
{
    if(true)
    {
        doSomething();
    }
}
```

```
@class
public: public class' ID
{
    if(true)
    {
        __BRACKETO
        doSomething();
    }
    __BRACKETC
}
```

Figure 18.: Inner brackets replaced by keywords

Doing this transformation guarantees that no inner brackets will appear inside semantic actions while our generator processes our specification files. The generator will, obviously, have to do the inverse transformation in order to correctly map the semantic actions into the final Lexer Grammar.
Another issue that appeared during the implementation phase is due to ANTLR’S Lexer Grammar’s nature. Because we are operating only at a Lexical level, it is impossible to inspect tokens in order to use their values, as there are no tokens yet. ANTLR provides us with a `getText()` method that returns us the full matched string. For example, if we have the following production defined:

```
public_class: 'public class' ID;
{
  getText();
}
```

Listing 3.18: Production to recognize public classes

When processing the following string

```
public class Example
```

The whole string will be returned by the `getText` method. If we wanted to use only the value contained in ID we would have to break the string in words and return it’s third element.

In order to abstract the user of this process, we decided to create a mechanism that permits the user to directly refer to the text value of an element.

First of all, we created the following method that will be automatically mapped into the final Lexer Grammar.

```
public String getTextAt(String text, int position)
{
  \Splits the string by spaces
  String[] split = text.split(" ");
  \returns the element at ‘position’
  return split[position-1];
}
```

Listing 3.19: Method to return a subset of a string

This method enables us to collect only the text of a specific value, instead of the whole string. So, if we replaced our previous code with:

```
public_class: ‘public class’ ID;
{
  getTextAt(getText(),3);
}
```

Listing 3.20: Production to recognize public classes
With the same input string as before, instead of the whole string, only the text value of ID would be returned. We then defined a specific syntax to let the user refer an element, we call these elements variables.

```
VAR: '$' ID '.text'
```

Listing 3.21: Production to recognize public classes

So, translating this to our previous example, in order to access the text value of ID, we would write:

```
public_class: 'public class' ID;
{
  $ID.text
}
```

Listing 3.22: Production to recognize public classes

Our preprocessor would then calculate the position of ID in the production text, and replace it with 3.20. But, there are cases were the same identifier is found more than once in the same production. For those cases, we created a labeling mechanism in order eliminate conflicts:

```
extends: 'public class' class=ID 'extends' super=ID;
{
  System.out.println($class.text + " extends " + super.text);
}
```

Listing 3.23: Example of label usage

When a variable is recognized we have to calculate it’s position, considering the whole string. Then we invoke the getTextAt method, passing the previously calculated position. We call this process expanding the variable. The first problem in this logic, is that terminal symbols may, or may not, have spaces in them. To the eyes of the developer, however, a terminal symbol is just one symbol, whether it contains spaces or not. For that reason, before expanding the variable we remove all white spaces from all terminal symbols, and use this new auxiliary string when calculating position. That way, in our logic, white spaces inside terminals are disregarded.

```
VAR: '$' ID '.text'
{
  //Remove whitespaces from terminals
  String aux_production = replacePattern("(["\"\s]\s*\=\s*[^"\s ]])*", \
     production,"\s", "");

  //get the expanded text
  bracket_body += expandVar(aux_production,getID(getText()));
}
```

Listing 3.24: Invoking the variable expansion method
To make the translation to the `getTextAt` method, we need to calculate the position of the labeled variable regarding the whole text.

We start by breaking the text by spaces.

```java
public String expandVar(String production, String id)
{
    int index = 0;
    int has_operation = 0;
    //Splits the production by spaces
    String[] split = production.split(" ");
    //List to keep the labels ids
    ArrayList<String> aux = new ArrayList<String>();
    //flag is activated when a label is recognized
    boolean label;

    Listing 3.25: Calculating the variable’s position
```

Then, we iterate through the strings returned by the split method. If the string is only composed by whitespaces or if it is a terminal we ignore it. This step prevents any trouble with terminals that may contain the `=` symbol and could mistakenly be recognized as a label.

If the string passed the previous two tests than we check if it contains the `=` symbol. If it does, then we are in the presence of a label. In that case, we split the string by the `=` symbol and store the label/symbol relationship in a map. We need to store this relationship in order to replace the production definition with a label free one. This, of course, is needed because ANTLR’s Lexer Grammars do not support labels.

We store all the symbols that are not whitespaces in an auxiliary list. We will need the size of this list in order to calculate the position of the variable.
Finally we also need to check if the symbol is an operation. These symbols will not be mapped to the final productions, and as so, when calculating the variables position we need to also decrement the number of operation symbols encountered.

```java
for(String s : split)
{
    label = false;
    if(!s.matches("^\s*$"))
    {
        if(!isTerminal(s))
        {
            if(s.contains("")
            {
                String[] split_label = s.split("=");
                labels.put(split_label[0],split_label[1]);
                label = true;
                aux.add(split_label[0]);
            }
        }
        if(isOperation(s))
            has_operation++;
    }
    if(label == false)
        aux.add(s);
}
index++;
```

Listing 3.26: Iterating through the production
The last step is to calculate the position of the variable and return the `getTextAt` invocation syntax that will be mapped into the final Lexer Grammar.

```java
index = 0;
for(String s : aux) {
    if(s.equals(id)) {
        // calculates its position
        int position = aux.size() - index - has_operation;
        return "getTextAt(getText()," + position + ")";
    }
    index++;
}
```

Listing 3.27: Calculating the variable's position

This way our semantic code block will be replaced by:

```java
extends: 'public class' class-ID 'extends' super-ID;
{
    System.out.println(getTextAt(getText(),2) + " extends " + getTextAt(getText(),4));
}
```

Listing 3.28: Definition of the production with labels

Just one last step is needed in this process, we still have to eliminate the labels in the definition of the production, because there are no labels in Lexer Grammars.

For that, at the end of our production, we iterate through the labels map that we previously constructed and replace the original string with a label free one.

```java
PROD_EXIT: ';' {
    for(String s : labels.keySet()) {
        production = production.replace(s + "=" + labels.get(s), labels.get(s));
    }
    // writes the new production to the modified spec file
    writeToFile(production + bracket_body + ";",true);
}
```

Listing 3.29: Invoking the variable expansion method
After all these steps our production will look like this in the end of the processing.

```
extends: 'public class' ID 'extends' ID;
{
    System.out.println(getTextAt(getText(), 2) + " extends " + getTextAt(getText(), 4));
}
```

Listing 3.30: Definition of the production without labels

Below, we can see a simple example of a specification file before and after the preprocess.

![Diagram showing modifications made in the spec file]

Figure 19.: Modifications made in the spec file

For the next step in our workflow, the resulting file from the preprocess will be fed to our generator, in order to create a new ANTLR Lexer Grammar.
3.4.2 Generator

At this point in our workflow, our initial Fuzzy Grammar specification file has already gone through a preprocessor, originating a modified specification file.

```java
BEGIN NAME
Example
END NAME

BEGIN IMPORTS
import java.util.HashMap;
import java.util.ArrayList;
END IMPORTS

BEGIN METHODS
public String myCustomMethod()
{
    return "My custom string";
}
END METHODS

BEGIN CONTEXTS

@default
open_class: 'public class' >> class_header;
ID: [A-Za-z];
@class_header
extends: ID 'extends' ID;
{
    //Just to illustrate brackets
    if(true)
    __BRACKETO
        System.out.println(getTextAt(getText(),1) + "extends" + getTextAt(getText(),3));
    __BRACKETC
}
open_body: '{' ˆ> class_body;
exit_class: '}' ˆ;
@class_body
exit_body: '}' ˆ;

END CONTEXTS
```

Listing 3.31: Example of a modified specification file
The next step in our workflow is to transform this modified file into an ANTLR Lexer Grammar. Because the more complex translations were dealt with in our previous step (preprocessor), we can now make a more straightforward translation here in our generator.

We will now make a step by step description of the mapping of our main sections, considering the modified file in 3.31, to their counterparts in ANTLR’s Lexer Grammar syntax.

First of all, ANTLR’s Lexer Grammars start with the declaration of the grammar’s name. It is easy to see that this means a direct mapping from our Name section to their syntax.

```
BEGIN NAME
Example
END NAME
```

Figure 20.: Mapping the name of the grammar

So our final Lexer Grammar would have the following declaration:

```
lexer grammar Example;
```

Listing 3.32: Lexer Grammar name declaration

ANTLR also has a section reserved for imports, called `@header`. So, in this case, the mapping is also direct.

```
BEGIN IMPORTS
...  
END IMPORTS
```

@header {

```
import java.util.HashMap;
import java.util.ArrayList;
```

Listing 3.33: Lexer Grammar with imports section

Another section of our Lexer Grammar is complete.

```
lexer grammar Example;

@header {
    import java.util.HashMap;
    import java.util.ArrayList;
}
```
Mapping the methods section is also straightforward, as ANTLR has a section for it, called `@members`. The difference here is that, not only do we map the user defined methods, we add our `getTextAt` method, that was introduced in section 3.4.1.

![Mapping the methods section](image)

And so, we have another piece of our Lexer Grammar puzzle.

```java
lexer grammar Example;

@header {
    import java.util.HashMap;
    import java.util.ArrayList;
}

@members{

    public String getTextAt(String text, int position) {
        String[] split = text.split(" ");
        return split[position-1];
    }

    public String myCustomMethod()
    {
        return "My custom string";
    }
}
```

Listing 3.34: Lexer Grammar with members section

So far, all of our mapping has been pretty much straightforward. In our context section thing start to get a bit tricky.

First of all, our `@default` context doesn’t have a direct counterpart in ANTLR’s syntax. ANTLR’s convention is that everything defined before the declaration of the first mode is the `default` mode. That way, we only need to do a direct mapping of what is defined in our `@default` context to the top of ANTLR’s production declaration space.

Of course, we have to translate any context transition operations that may appear. As we have seen before, three types of transition operations may appear.
Firstly, a push transition is mapped into the `pushMode` command in ANTLR’s syntax.

A pop transition is mapped into the `popMode` command.

And finally, a pop push transition is mapped into the `mode` command.

In order to abstract the user from the declarations of whitespaces, we declare a whitespace production in the `default` context.

```
WS: [ \t\n\r ];
```

We assume that between every symbol multiple whitespaces may appear. This means that we automatically inject whitespaces into our productions.
A key point to this Fuzzy Parsing rationale, is ignoring everything that is not defined in the specification file. To reach this goal, we add in every context a *trash* production, whose purpose is to collect everything that doesn’t fall in any other production. ANTLR tries to recognize productions by the order on which they were defined, and so, the *trash* production needs to be the last one in its context.

```
TRASH: .;
```

Listing 3.36: Whitespace production

Because we add this production to every context, the user has no need to specify anything, and thus, remains abstracted from this process. Our naming convention for these trash productions is TRASH_CONTEXT_NAME. We adopted this convention because productions need to have a unique name.

So, resuming our example, our `@default` context would be mapped to:

```
@default
open_class: 'public class' >> class_header;
ID: [A-Za-z];

OPEN_CLASS: 'public class'
{
    pushMode(CLASS_HEADER);
}
ID: [A-Za-z]
TRASH_DEFAULT: .;
```

Figure 27.: Mapping our default context
And looking at our Lexer Grammar:

```java
lexer grammar Example;

@header {
import java.util.HashMap;
import java.util.ArrayList;
}

@members{

public String getTextAt(String text, int position) {
    String[] split = text.split(" ");
    return split[position-1];
}

public String myCustomMethod() {
    return "My custom string";
}

OPEN_CLASS: 'public class'
{
    pushMode(CLASS_HEADER);
}

ID: [A-Za-z];

TRASH_DEFAULT: .;
```

Listing 3.37: Lexer Grammar with default section

Mapping the rest of the contexts isn’t much different from mapping the default context. In fact, the only thing that changes, is that we need to map the name of the context. In ANTLR’s syntax, the counterpart to our @ context keyword is `mode`.

![Figure 28.: Mapping a custom context](image-url)
And our Lexer Grammar:

lexer grammar Example;

@header {
    import java.util.HashMap;
    import java.util.ArrayList;
}

@members{

    public String getTextAt(String text, int position) {
        String[] split = text.split(" ");
        return split[position-1];
    }

    public String myCustomMethod() {
        return "My custom string";
    }
}

OPEN_CLASS: 'public class'
{
    pushMode(CLASS_HEADER);
}

ID: [A-Za-z];

TRASH_DEFAULT: .;

mode CLASS_HEADER;

EXTENDS: WS* ID WS* 'extends' WS* ID;
{
    //We will address semantic actions next
};

OPEN_BODY: '{'
{
    mode(CLASS_BODY);
};

EXIT_CLASS: '}'
{
    popMode();
};
mode CLASS_BODY;
EXIT_BODY: '}
{
  popMode();
};

Listing 3.38: Lexer Grammar with default section

The only thing left to address, is what would be, in theory, the most complex task, mapping semantic actions. However, because we already simplified this task in the preprocessor (3.4.1) step, this ends up to be a fairly simple task. In fact, the only thing that needs to be done, is the translation of the __BRACKETO and __BRACKETC keywords to their respective bracket symbols, the rest is a direct mapping.

Our final Lexer Grammar looks like:

```java
@header {
  import java.util.HashMap;
  import java.util.ArrayList;
}

@members{
  public String getTextAt(String text, int position) {
    String[] split = text.split(" ");
    return split[position-1];
  }
  public String myCustomMethod() {
    return "My custom string";
  }
}
```
OPEN_CLASS: 'public class'
{
    pushMode(CLASS_HEADER);
}

ID: [A-Za-z];

TRASH_DEFAULT: .;

mode CLASS_HEADER;

EXTENDS: WS* ID WS* 'extends' WS* ID;
{
    //Just to illustrate brackets
    if(true)
    {
        System.out.println(getTextAt(getText(),1) + "extends" + getTextAt(getText(),3));
    }
};

OPEN_BODY: '{'
{
    mode(CLASS_BODY);
};

EXIT_CLASS: '}'
{
    popMode();
};

mode CLASS_BODY;

EXIT_BODY: '}'
{
    popMode();
};

Listing 3.39: Lexer Grammar with default section
We can see the complete transformation in the following image:

![ANTLR Lexer Grammar Example](image)

We now have a complete ANTLR Lexer Grammar, that specifies a Fuzzy Grammar through its Lexical Modes functionality. Now, using ANTLR we could generate a Fuzzy Parser, ready for use.

In the next section we will see a practical example of how we can use this workflow to create something.
CASE STUDY

In this chapter, we will introduce a practical example of the usage of our tool. The purpose is to start with a simple example and, building on it, increase complexity in order to show the main features of the tool.

Our goal in this example is to recognize Java classes and their inheritance and implementation relationships. We also want to recognize inner methods and attributes of the classes. Also, we will recognize method arguments, distinguishing their scope from class attributes through contexts.

The final product of this process would be a diagram, were we could see all the relationships that were recognized.

In order to maintain simplicity and readability in this example, we will not define code to generate diagrams. Instead we will use pseudo code invocations of the following methods:

- drawClass(class) - A method that receives a class and generates a drawing of that class;
- drawInterface(interface) - A method that receives an interface and generates a drawing of that interface;
- drawExtendsRelationship(parent, child) - A method that draws a hierarchical relationship between 2 classes;
- drawImplementsRelationship(implementee, implementor) - A method that draws an implementation relationship between an Interface and a class;
- drawClassMethod(class,method) - A method that draws a methods in a class;
- drawInterfaceMethod(interface,method) - A method that draws a methods in an interface;
- drawClassArg(class,arg) - A method that draws a class attribute;
- drawMethodArg(method,arg) - A method that draws a method argument;

We will also print to the standard output all classes, relationships and methods that are being recognized. Just so we can see some results.
This example, although lightweight, can be somewhat useful to Program Comprehension based utilities. When generating a class diagram from source code the parsing logic would be similar, just the implementation of the drawing methods would change. Due to substantial volume and complexity of the Java programming language, we will focus on a small subset of the language.

4.1 THE RUNNING ENVIRONMENT

This example was created using the following working environment:

- Ubuntu 14.04 LTS (Trusty Tahr) 64x - Ubuntu is a Debian based Linux operating system developed by Canonical.
- NetBeans 7.4 - NetBeans is an integrated development environment (IDE) for developing primarily with Java. It has integration with AntlrWorks2 through a plugin.
- ANTLR 4.1 - ANTLR is a parser generator developed by Terence Parr. At the time of this work the 4.1 version was the only one that was supported by AntlrWorks2.
- AntlrWorks2 - ANTLRWorks is an ANTLR GUI Development Environment.
- Java 7 (open jdk) - Open-source implementation of the Java Platform, Standard Edition.

We will use a small input file with an Animal class that will be extended by 3 other classes and a Mascot interface that will be implemented by 2 other classes (one of them is also an animal).

![Figure 31.: Input file class diagram](image)

The full input file can be found at A.1
In order to support our class diagram example, 5 classes were created.

The **RE.Class** class represents a Java class with the following components:

- **id** - The identifier of the class;
- **methods** - A list of methods defined inside the class;
- **extends** - The Superclass of this class;
- **implements** - The interface this class implements;

The **RE.Interface** class represents a Java interface. For simplicity we will consider that a interface may not implement other interfaces. **RE.Interface** has the following attributes:

- **id** - The identifier of the interface;
- **methods** - A list of methods defined inside the interface;

The **RE.Method** class represents the declaration of a Java method. It has the following attributes:

- **id** - The identifier of the method;
- **return_type** - The return type of the method;
- **args** - A list of method arguments;

The **RE.Arg** class represents the declaration of a Java parameter. It has the following attributes:

- **type** - The type of the argument;
- **id** - the id of the argument;

The **RE.DiagramManager** class is responsible for storing and drawing all recognized classes. This will be a Singleton element, with the following attributes:

- **classes** - List of recognized classes;
- **interfaces** - List of recognized interfaces;
- **current_class** - Last recognized class;
- **current_interface** - Last recognized interface;
- **current_method** - Last recognized method;
4.3 USING OUR TOOL

First of all, let's just make a brief recap of our workflow process. We define a new Fuzzy grammar in a regular text file; this file goes through a preprocessor; The preprocessor creates a new file that contains our original specification, modified by the preprocessor; This new file then goes through our generator that creates an ANTLR lexer grammar that is a reflection of our specification file; Through ANTLR we can now generate a new parser based on the lexer grammar, this parser is a Fuzzy Parser due to the nature of our specification.

Unfortunately, at the moment of this writing, this process was not encapsulated in a single command fashion, and so, all steps have to be done separately. Our preprocessor and generator are Java classes that depend on ANTLR, so to compile them we have to specify the path to ANTLR's JAR file.

> javac -cp "PATH_TO_JAR:$CLASSPATH" PreProcessor.java Generator.java

Then, considering that our Fuzzy Grammar is called my_fuzzy_grammar.txt, we pass it to the preprocessor. The preprocessor will create a new file called my_fuzzy_grammar_modified.txt.

> java PreProcessor.java my_fuzzy_grammar.txt
The next step is to run our new modified file through our generator. The generator will then create a lexer grammar.

> java Generator.java my_fuzzy_grammar_modified.txt

We can then use ANTLR to generate our final Fuzzy Parser. This can be done using the Antlr-Works2 IDE or by command line.

> antlr my_fuzzy_grammar.g4

ANTLR will then generate a Fuzzy Parser that we can use by running the newly generated Run.java file.
Considering we have an input file called `Animals.java`, then we just need to carry the final step:

```bash
generate Run.java Animals.java
```

Now that we revised the work process, let’s start by defining a new Fuzzy Grammar. We start by specifying our data domain inside our methods section. We will use Java’s `ArrayList` and `HashSet`, so we have to import it. For this example we will use `Running_Example` as the grammar name. The full source for domain definition and its Lexer Grammar transformation can be found at A.2 and A.3 respectively.

When specifying contexts and productions we will follow the following naming convention to ease readability:

```plaintext
<context>
  CONTEXT_PRODUCTIONNAME: ...
```
In our default section we will specify 2 constructs of interest. One to recognize public classes, and another to recognize public interfaces.

BEGIN CONTEXTS
@default
DEFAULT_CLASS: 'public class';
DEFAULT_INTERFACE: 'public interface';
END CONTEXTS

Listing 4.1: class and interface constructs of interest

At this point, public class and public interface terms will be recognized, though nothing will be done with them.

When we reach a new class or interface, we will want to inspect its header and see what kind of information we can extract. We could do this in the default context, but defining a new context just for the header, gives us a more secure environment. Inside a class or interface header only a small number of constructs of interest may appear. By limiting our context, we are reducing search space and diminishing ambiguity issues.

In order to go to this new states, we need to specify our transitions.

BEGIN CONTEXTS
@default

//push transition from default to class_header
DEFAULT_CLASS: 'public class' >> class_header;

//push transition from default to interface_header
DEFAULT_INTERFACE: 'public interface' >> interface_header;

END CONTEXTS

Listing 4.2: Defining transitions to headers
If we think on how our context automaton look like at this point, we reach the conclusion that we have a start state with two possible transitions.

![Figure 37.: Context Automaton after the definition of the default state](image)

Inspecting our lexer grammar at this point we will be able to see the above transitions mapped:

```plaintext
DEFAULT_CLASS: WS* 'public class'  
{  
pushMode(CLASS_HEADER);  
};

DEFAULT_INTERFACE: WS* 'public interface'  
{  
pushMode(INTERFACE_HEADER);  
};

WS: [ \t\n\r]; //Automatic Whitespace definition

DEFAULT_TRASH: .; //Automatic production to ignore everything that is not defined
```

Listing 4.3: Default context in the lexer grammar

Inside our `class_header` context, we want to consider 4 constructs of interest. Firstly, we want to recognize the class id, in order to generate the class drawing. We also want to recognize `extends` and `implements` keywords, in order to generate its relationships. Finally, we want to recognize the `{` symbol, that identifies the end of the class header and the beginning of the class body.

```plaintext
@class_header
CLASS_HEADER_ID: CLASS_ID;
CLASS_HEADER_EXTENDS: 'extends' CLASS_ID;
CLASS_HEADER.IMPLEMENTS: 'implements' CLASS_ID;
CLASS_HEADER_INNER: '{';
CLASS_ID: [A-Z][A-Za-z];
```

Listing 4.4: Constructs of interest in class header
Whenever a `class_id` is recognized we store it in the `current_class` variable. This variable is needed when reaching other constructs of interest. We also store the class in the diagram manager, and draw it. Just for a proof of concept, we will also print to the standard output the class id.

```java
@class_header
CLASS_HEADER_ID: CLASS_ID;
{
   //In the final grammar $CLASS_ID.text will be transformed into getTextAt(
   //getText(),1)
   RE_Class new_class = new RE_Class($CLASS_ID.TEXT);
   diagram_manager.current_class = new_class;
   diagram_manager.classes.add(new_class);
   diagram_manager.drawClass(new_class);
   System.out.println(new_class.id);
}
```

Listing 4.5: Recognizing a class id

When a `extends` anchor is reached, we create a new `RE_Class` passing the following `class_id`, and save it in the `current_class extends` attribute. We also draw the `extends` relationship between the two classes.

```java
CLASS_HEADER_EXTENDS: 'extends' CLASS_ID;
{
   //In the final grammar $CLASS_ID.text will be transformed into getTextAt(
   //getText(),2)
   RE_Class super = diagram_manager.getClass($CLASS_ID.text);
   diagram_manager.current_class.extends = super;
   diagram_manager.drawExtendsRelationship(super, current_class);
   System.out.println(current_class.id + " extends " + super.id);
}
```

Listing 4.6: Recognizing a hierarchical relationship

Now, when a `implements` anchor is reached, we create a new `RE_Interface` with the next `class_id`, and save it in the `current_class implements` attribute. We also draw the `implements` relationship between them.

```java
CLASS_HEADER_IMPLMENTS: 'implements' CLASS_ID;
{
   RE_Interface interface = diagram_manager.getInterface($CLASS_ID.text);
   diagram_manager.current_class.implement = interface;
   diagram_manager.drawImplementsRelationship(interface, current_class);
   System.out.println(current_class.id + " implements " + interface.id);
}
```

Listing 4.7: Recognizing an implementation relationship
In the next production can see a practical example of a pop push transition. When we reach the opening bracket we make a transition to the class\_body context. Because we can guarantee that the \{ symbol will be the last input in the class header, it makes no sense popping from class\_body back to class\_header. By making a pop push transition, we change the top of the stack, making sure that when we pop from class\_body we end up in the predecessor of class\_header.

```
CLASS_HEADER_INNER: '{' ^> class_body;
```

Listing 4.8: Pop push transition to class body

![Diagram of Context Automaton after defining class header](image)

Figure 38.: Context Automaton after defining class header

In our lexer grammar, we can see the resulting class header context:

```
mode CLASS_HEADER;

CLASS_HEADER_ID: WS* CLASS_ID;
{  
    RE_Class new_class = new RE_Class(getTextAt(getText(),1));
    diagram_manager.current_class = new_class;
    diagram_manager.classes.add(new_class);
    diagram_manager.drawClass(new_class);
    System.out.println(new_class.id);
}

CLASS_HEADER_EXTENDS: WS* 'extends' WS* CLASS_ID;
{  
    RE_Class super = diagram_manager.getClass(getTextAt(getText(),2));
    diagram_manager.current_class.extends = super;
    diagram_manager.drawExtendsRelationship(super, current_class);
    System.out.println(current_class.id + " extends " + super.id);
}
```

CLASS_HEADER_IMPLEMENT: WS* 'implements' WS* CLASS_ID;
{
    RE_Interface interface = diagram_manager.getInterface(getTextAt(getText(), 2));
    diagram_manager.current_class.implements = interface;
    diagram_manager.drawImplementsRelationship(interface, current_class);
    System.out.println(current_class.id + " implements " + interface.id);
};

CLASS_HEADER_INNER: WS* '{'
{
    mode(CLASS_BODY);
};

CLASS_ID: [A-Z][A-Za-z];

CLASS_HEADER_TRASH: .;

Listing 4.9: Constructs of interest in class header

Our interface header is a bit simpler as there are no relationships to consider. We are only looking to recognize the interface id, and any methods inside its definition.

@interface_header
INTERFACE_HEADER_ID: INTERFACE_ID;
INTERFACE_HEADER_INNER: '{';
INTERFACE_ID: [A-Z][A-Za-z];

Listing 4.10: Constructs of interest in interface header

The implementation is similar to class_header context. The full transformation of the context to the lexer grammar notation can be found at A.4.

Figure 39.: Context Automaton after defining interface header
At this point we are already able to recognize and draw classes, interfaces, and their relationships. Running our generated Fuzzy Parser with our the input file presented in A.1 we get the following results:

Animal
Mascot
Dog
Dog extends Animal
Cat
Cat extends Animal
Penguin
Penguin extends Animal
Penguin implements Mascot
Clown
Clown implements Mascot

Listing 4.11: Generated output

Now to take this a step further, lets inspect the body of our classes and interfaces and try to recognize methods. Whenever a method is recognized, we will draw it and print it to the standard output.

Our class_body context will have 4 constructs of interest. We will want to recognize class attributes and methods. Also, we will want to jump to a Method context, when one is reached, and we will also want to pop context at the end of the class definition.

@class_body

CLASS_BODY_ARG: CLASS_BODY_TYPE CLASS_BODY_ID;
CLASS_BODY_METHOD: 'public' CLASS_BODY_TYPE CLASS_BODY_ID '(' >> method_args;
CLASS_BODY_INNER: '{'; \Beginning of a method
CLASS_BODY_EXIT: '}'; \End of class definition
CLASS_BODY_ID: [a-zA-Z]+;
CLASS_BODY_TYPE: 'void'|'String'|'int'|'boolean';

Listing 4.12: Constructs of interest in class body
Whenever a class attribute is found we will draw it in the current class, and print it to the standard output.

```java
CLASS_BODY_ARG: CLASS_BODY_TYPE CLASS_BODY_ID
{
    RE_Arg new_arg = new RE_Arg($CLASS_BODY_TYPE.text, $CLASS_BODY_ID.text);
    diagram_manager.current_class.args.add(new_arg);
    diagram_manager.drawClassArg(current_class, new_arg);
    System.out.println("Class " + current_class.id + " has an attribute of type "
    + new_arg.type + " called " new_arg.id
};
```

Listing 4.13: Recognizing a class attribute

When a class method is recognized, we store it in the current_method variable, and insert it in the methods list contained in the current_class variable. Afterwards, we draw the method and print a confirmation message to the standard output. Finally, we make a transition to the method_args context, in order to recognize any eventual method argument.

```java
CLASS_BODY_METHOD: 'public' CLASS_BODY_TYPE CLASS_BODY_ID '(' >> method_args
{
    RE_Method new_method = new RE_Method($CLASS_BODY_ID.text, $CLASS_BODY_TYPE.text
    );
    diagram_manager.current_method = new_method;
    diagram_manager.current_class.methods.add(new_method);
    diagram_manager.drawClassMethod(current_class, new_method);
    System.out.println("Class " + current_class.id + " has a method of return type "
    + new_method.type + " called " new_method.id
};
```

Listing 4.14: Recognizing a class method

When a { symbol is recognized, it means that we already have passed in the method_args context, and so we make a transition to method_body context.

```java
CLASS_BODY_INNER: '{' >> method_body;
```

Listing 4.15: Entering a class method

And whenever a } is recognized, it means that our class definition has ended (any inner method } symbols will be caught in the method_body context), and so we pop the current context;

```java
CLASS_BODY_EXIT: '{' ˆ;
```

Listing 4.16: Exiting the class body context
The translation to our final lexer grammar can be found at A.5.
The method_args and method_body states are added to our automaton.

Figure 40.: Automaton after class body definition

The interface_body context definition is similar to class_body. For simplicity, we will assume that interfaces may only contain methods with empty bodies and no arguments. That means that only method headers may appear in an interface definition.

Listing 4.17: Constructs of interest in interface body

The lexer grammar equivalent can be found at A.6.
Our *method_args* context will store each argument in the *current_method* variable, and draw it in the diagram. Whenever a ')' is recognized, we pop back to our parent context (in this case *method_header*).

```
@method_args

METHOD_ARGS_ARG: METHOD_ARGS_TYPE METHOD_ARGS_ID ',' ;
{
    RE_Arg new_arg = new RE_Arg($METHOD_ARGS_TYPE.text, $METHOD_ARGS_ID);
    diagram_manager.current_method.args.add(new_arg);
    diagram_manager.drawMethodArg(current_method, new_arg);
    System.out.println("Argument " + new_arg.id + " has return type "
        + new_arg.type + " and belongs to " + current_method.id);
}

METHOD_ARGS_EXIT: '}' ^;
```

Listing 4.18: Constructs of interest in method args

Our method logic *per se* does not interest us for this diagram exercise. For that reason, our only construct of interest in the *method_body* is the } symbol, and we will pop context whenever it is reached.

```
@method_body

METHOD_BODY_EXIT: '}' ^;
```

Listing 4.19: Constructs of interest in method body

Our lexer grammar equivalent can be found at A.7:

Our automaton after these two contexts:

```
Figure 41.: Automaton after method args and method body contexts
```
At this moment we are already able to recognize all of our constructs of interest and generate our class diagram. Running our Fuzzy parser again we get the following results on the standard output:

```java
Animal
Class Animal has an attribute of type int called age
Class Animal has an attribute of type boolean called sex
Class Animal has a method of return type void called eat
Argument food has return type String and belongs to eat
Mascot
Interface Mascot has a method of return type void called company
Dog
Dog extends Animal
Class Dog has an attribute of type String called breed
Class Dog has an attribute of type String called name
Class Dog has a method of return type void called eat
Argument food has return type String and belongs to eat
Cat
Cat extends Animal
Class Cat has an attribute of type String called breed
Class Cat has a method of return type void called eat
Argument food has return type String and belongs to eat
Penguin
Penguin extends Animal
Penguin implements Mascot
Class Penguin has an attribute of type String called company
Class Penguin has a method of return type void called company
Class Penguin has a method of return type void called eat
Argument food has return type String and belongs to eat
Clown
Clown implements Mascot
Class Clown has an attribute of type String called company
```

Listing 4.20: Recognizing a class id
Using a standard diagram notation and generator (like DOT and GraphViz), we are able to generate the following diagram.

![Diagram](image)

Figure 42.: Final class diagram

From this example, we can see that from a very lightweight and high level language we are able to use all of ANTLR’s lexer grammar power. By abstracting the user from tasks as whitespace definition, and giving the developer a high level way to use the text values of the input that is processed, we are simplifying all this process and making it much more user friendly. The fact that we also abstract the user from the fuzzyness of the parsing is also important, as developers need not to adapt to this new paradigm. They specify a Fuzzy Grammar just like a regular grammar.
5

CONCLUSIONS AND FUTURE WORK

5.1 CONCLUSIONS

From the beginning of this dissertation, we understood that defining a new Fuzzy Parsing technique was not a simple task. The mere definition of Fuzzy Parsing is a bit abstract and open to interpretation by nature. How do we parse something that is incorrect? That was the main question that drove this idea further. While searching for clues out on the scientific community, we found out that multiple definitions and opinions existed on the same matter, and that there was still no de facto standard on the subject.

That gave even more motivation to devise and propose a new technique. We envisioned a new technique based on automata notions, based on contexts and transitions between contexts. We were lucky enough to find out that ANTLR already had a mechanism that would let us implement our ideas, and so, we took that opportunity.

But the whole scheme of things still seemed a bit overwhelming. Again, the mere notion of having to parse incorrect sources still baffles developers that are starting to pave their way into the area, as this is a completely different rationale as opposed to traditional parsing techniques. That lead us to think that we needed to narrow the bridge between Traditional Parsing and Fuzzy Parsing techniques.

The next step would be to let developers specify fuzzy parsers, but, maintaining the familiarity of regular context free grammar approaches and traditional compiler compiler environments.

This layer of abstraction was obtained by introducing our Fuzzy Parsing specification language and the whole workflow behind it. This way a developer is able to fully concentrate on what they want to recognize, and not be distracted with implementation details. The ability to manipulate the source itself is a big step forward, as we are able to jump through some limitations that come attached with lexical analysis.

It is this friendly environment that is produced by the usage of labels, semantic actions, token attributes, etc, that give this tool a bit of life and make it stand out from the rest.
There is still a lot to improve in the future. This is just a first draft of the tool. For future work, we could start by creating a GUI that used this engine, that way improving the user experience.

Errors still aren’t collected and presented to the developer. For example, a developer may only define a transition to a context that is already defined. If that does not happen, the developer should be informed of this error.

At a more functional level, the set of token attributes should be expanded. At this moment only the text attribute is implemented, but to this set we could add attributes like line or column that are regular presences in most compiler compilers and are, without doubt, of great importance and usage for the developer.

The benefits of Fuzzy Parsing in a wide variety of contexts has already been discussed in the previous chapters. We hope that this new approach helps to fill the gap between Traditional Parsing and Fuzzy Parsing usage. We feel that abstracting the developer from the extra complexity that comes with Fuzzy Parsing is a key step to a wider adoption of this kind of parsing and we are confident that this approach is a step in the right direction.
BIBLIOGRAPHY


Mario Berón, Pedro R. Henriques, Maria J. V. Pereira, and Roberto Uzal. Static and dynamic strategies to understand c programs by code annotation. In OpenCert’07, 1st Int. Workshop on Fondations and Techniques for Open Source Software Certification (collocated with ETAPS’07), 2007b.


LISTINGS OF CODE

In order to maintain a fluid and easy document reading, it was decided not to include all the source code in some of the chapters. However, these snippets of code are meaningful and do contribute to a better comprehension of the solution as a whole.

For that reason, we decided to create this chapter, where we can find the snippets that were not included before.

```java
public class Animal {
    int age;
    boolean sex;

    public void eat(String food) {
        System.out.println( "Animal Eating..." );
    }
}

public interface Mascot {
    public void company;
}

public class Dog extends Animal {
    String breed;
    String name;

    public void eat(String food) {
        System.out.println( "Dog Eating " + food );
    }
}

public class Cat extends Animal {
    // Cat class code
}
```
String breed;

    public void eat(String food)
    {
        System.out.println( "Cat Eating" + food );
    }
}

public class Penguin extends Animal implements Mascot
{
    String company;

    public void company()
    {
        System.out.println("I am a penguin and also " + company + "'s mascot")
    }

    public void eat(String food)
    {
        System.out.println( "Penguin Eating" + food );
    }
}

public class Clown implements Mascot
{
    String company;

    public void company()
    {
        System.out.println("I am a clown and also " + company + "'s mascot")
    }
}

Listing A.1: input file used on the case study

BEGIN NAME
Running_Example
END NAME
BEGIN IMPORTS
import java.util.ArrayList;
import java.util.HashSet;

public class RE_Class
{
    public String id;
    public ArrayList<RE_Method> methods;
    public ArrayList<RE_Arg> args;
    public RE_Interface implements;
    public RE_Class extends;

    public RE_Class(String id)
    {
        this.id = id;
        this.methods = new ArrayList<RE_Method>();
        this.args = new ArrayList<RE_Arg>();
    }
}

public class RE_Method
{
    public String id;
    public String return_type;
    public ArrayList<RE_Arg> args;

    public RE_Method(String id, String type)
    {
        this.id = id;
        this.type = type;
        this.args = new ArrayList<RE_Arg>();
    }
}

public class RE_Arg
{
    String type;
    String id;

    public RE_Arg(String type, String id)
    {
        this.type = type;
        this.id = id;
    }
}

public class RE_Interface
public String id;
public ArrayList<RE_Method> methods;

public RE_Interface(String id)
{
    this.id = id;
    this.methods = new ArrayList<RE_Method>();
}

public class RE_DiagramManager
{
    HashSet<RE_Class> classes;
    HashSet<RE_Interface> interfaces;
    RE_Class current_class;
    RE_Interface current_interface;
    RE_Method current_method;

    public RE_DiagramManager()
    {
        this.classes = new HashSet<RE_Class>();
        this.interfaces = new HashSet<RE_Interface>();
    }

    public RE_Class getClass(String id)
    {
        for(RE_Class stored_class : this.classes)
        {
            if(stored_class.id == id)
                return stored_class;
        }

        return null;
    }

    public RE_Interface getInterface(String id)
    {
        for(RE_Interface stored_interface : this.interfaces)
        {
            if(stored_interface.id == id)
                return stored_interface;
        }

        return null;
    }
}
Listing A.2: Case Study’s data domain definition

```java
lexer grammar Running_Example;

@header{
import java.util.ArrayList;
import java.util.HashSet;
}

@members{

public String getTextAt(String text, int position) {
    String[] split = text.split(" ");
    return split[position-1];
}

public class RE_Class {
    public String id;
    public ArrayList<RE_Method> methods;
    public ArrayList<RE_Method> args
    public RE_Interface implements;
    public RE_Class extends;

    public RE_Class(String id) {
        this.id = id;
        this.methods = new ArrayList<RE_Method>();
        this.args = new ArrayList<RE_Arg>();
    }
}

public class RE_Method {
    public String id;
```
public String return_type;
public ArrayList<RE_Arg> args;

public RE_Method(String id, String type)
{
    this.id = id;
    this.type = type;
    this.args = new ArrayList<RE_Arg>();
}

public class RE_Arg
{
    String type;
    String id;

    public RE_Arg(String type, String id)
    {
        this.type = type;
        this.id = id;
    }
}

public class RE_Interface
{
    public String id;
    public ArrayList<RE_Method> methods;

    public RE_Interface(String id)
    {
        this.id = id;
        this.methods = new ArrayList<RE_Method>();
    }
}

public class RE_DiagramManager
{
    public HashSet<RE_Class> classes;
    public HashSet<RE_Interface> interfaces;
    public RE_Class current_class;
    public RE_Interface current_interface;
    public RE_Method current_method;

    public RE_DiagramManager()
    {
        this.classes = new HashSet<RE_Class>();
    }
Listing A.3: Case Study’s data domain definition in the lexer grammar

```java
this.interfaces = new HashSet<RE_Interface>();
}

public RE_Class getClass(String id)
{
    for(RE_Class stored_class : this.classes)
    {
        if(stored_class.id == id)
            return stored_class;
    }
    return null;
}

public RE_Class getInterface(String id)
{
    for(RE_Interface stored_interface : this.interfaces)
    {
        if(stored_interface.id == id)
            return stored_interface;
    }
    return null;
}

//Drawing Logic here
{
    RE_Diagram diagram_manager = new RE_DiagramManager();
    }
```

```grammar
mode INTERFACE_HEADER;
INTERFACE_HEADER_ID: WS* INTERFACE_ID;
{
    RE_Interface new_interface = new RE_Interface(getTextAt(),1);
diagram_manager.current_interface = new_interface;
diagram_manager.interfaces.add(new_interface);
diagram_manager.drawInterface(new_interface);
System.out.println(new_interface.id);
};

INTERFACE_HEADER_INNER: '{'
```
mode INTERFACE_BODY;
};

INTERFACE_ID: [A-Z][A-Za-z];

INTERFACE_HEADER_TRASH: .;

Listing A.4: Interface header in the lexer grammar

mode CLASS_BODY;

CLASS_BODY_ARG: WS* CLASS_BODY_TYPE WS* CLASS_BODY_ID
{
    RE_Arg new_arg = new RE_Arg(getTextAt(getText(),1), getTextAt(getText(),2));
    diagram_manager.current_class.args.add(new_arg);
    diagram_manager.drawClassArg(current_class, new_arg);
    System.out.println("Class "+ current_class.id + " has an attribute of type 
     + new_arg.type + " called " new_arg.id
    ");
}

CLASS_BODY_METHOD: WS* 'public' WS* CLASS_BODY_TYPE WS* CLASS_BODY_ID WS* '('
{
    RE_Method new_method = new RE_Method(getTextAt(getText(),2), getTextAt(
     getText(),1));
    diagram_manager.current_method = new_method;
    diagram_manager.current_class.methods.add(new_method);
    diagram_manager.drawClassMethod(current_class, new_method);
    System.out.println("Class "+ current_class.id + " has a method of return 
     type 
     + new_method.type + " called " new_method.id
    ");
    pushMode(METHOD_ARGS);
}

CLASS_BODY_INNER: '{'
{
    pushMode(METHOD_BODY);
}

CLASS_BODY_EXIT: '{'
{
    popMode();
}

mode INTERFACE_BODY

INTERFACE_BODY_METHOD: WS* 'public' WS* INTERFACE_BODY_TYPE WS* INTERFACE_BODY_ID WS* ';'
{
    RE_Method new_method = new RE_Method(getTextAt(getText(),3), getTextAt(getText(),2));
    diagram_manager.current_interface.methods.add(new_method);
    diagram_manager.drawInterfaceMethod(current_interface, new_method);
    System.out.println("Interface " + current_interface.id + " has a method of return type " + new_method.type + " called " new_method.id);
}

INTERFACE_BODY_EXIT: WS* '{'
{
    popMode();
}

INTERFACE_BODY_TRASH: .;

mode METHODARGS;

METHODARGS_ARG: WS* METHODARGS_TYPE WS* METHODARGS_ID WS* '','
{
    RE_Arg new_arg = new RE_Arg(getTextAt(getText(),1), getTextAt(getText(),2));
    diagram_manager.current_method.args.add(new_arg);
    diagram_manager.drawMethodArg(current_method, new_arg);
    System.out.println("Argument " new_arg.id + " has return type " + new_arg.type + " and belongs to " current_method.id);
}

METHODARGS_EXIT: WS* '}'
{
    popMode();
}
Listing A.7: Method args and method body contexts in the lexer grammar