Unfuzzying Fuzzy Parsing

Pedro Carvalho, Nuno Oliveira, and Pedro Rangel Henriques

Departamento de Informática, Universidade do Minho, Braga, Portugal
{pedrocarvalho,nunooliveira,prh}@di.uminho.pt

Abstract

Traditional parsing has always been a focus of discussion among the computer science community. Numerous techniques and algorithms have been proposed along these years, but they require that input texts are correct according to a specific grammar. However, in some cases it's necessary to cope with incorrect or unpredicted inputs that raise ambiguities, making traditional parsing unsuitable. These situations led to the emergence of robust parsing theories, where fuzzy parsing gains relevance. Robust parsing comes with a price by losing precision and decaying performance, as multiple parses of the input may be necessary while looking for an optimal one.

In this short paper we briefly describe the main robust parsing techniques and end up proposing a different solution to deal with fuzziness of input texts. It is based on automata where states represent contexts and edges represent potential matches (of constructs of interest) inside those contexts. It is expected that such an approach reduces recognition time and ambiguity as contexts reduce the search space by defining a smaller domain for constructs of interest. Such benefits may be a great addition to the robust parsing area with application on program comprehension, among other research fields.

1998 ACM Subject Classification D.3.4 Processors

Keywords and phrases robust parsing, fuzzy parsing, automata

Digital Object Identifier 10.4230/OASIcs.SLATE.2014.101

1 Introduction

Recognizing sentences of a language has always been an interesting topic in computer science. This need can be easily transposed to our common life, as in our daily routine we are constantly in need to interpret all kind of inputs like images, texts or sounds. A good example of *natural parsing* is our ability to listen to human's speech and collect useful information from those sounds (sometimes imperceptible because of surrounding noise). Computer Science builds on this, as computers must understand each other or (primarily) the commands given by humans under the shape of programming language sentences.

To transpose this abstract notion of parsing to a formal domain, like computer science, rules and methodologies had to be engineered. Concerning classical parsing theory [1], the recognition process is done in accordance to rules of a formal grammar. This process is divided into three different types of analysis: Lexical, Syntactic and Semantic. Lexical analysis consists in reading and grouping the characters of an input text into tokens. In syntactic analysis a tree-like intermediate representation (the parsing tree) is built, based on the tokens and the rules of the underlying grammar. Finally, in Semantic analysis, the actual values of symbols are computed to decorate that tree and a check for consistency between the source and the language semantics definition is carried out.

In classical parsing theory sentences either belong or not to the language; and when belonging they must be derived from the grammar, following a structured set of rules without uncertainty. This property is related with the basic membership concept found on set theory [11]. However, in the recognition of unpredictable input (incomplete sentences or handwritten expressions [9, 13, 8]), it is impossible to create a grammar that covers all possible cases. If in some cases classical parsing falls short to solve the problem at hand, in other cases it goes beyond the needs. For instance, generating high level models from source code (in the context of program comprehension) does not require its complete recognition. By limiting the recognition process only to the essential portions, smaller parsing trees will be generated and unnecessary work will be spared. These scenarios where classical parsing proved to be inadequate, led to the emergence of robust parsing theories. In this context, a sentence belongs to a language with some degree of uncertainty, differently from the *do or die* situation seen before.

To cope with the robust parsing idea, two generic approaches have stood out: Island and Fuzzy grammars. But still in these approaches, parsing tends to be time consuming as multiple attempts may be required while looking for an optimal one. Performance decays and recognition precision may be partially or completely lost, which is undesirable when the objective is to extract reliable information from the input sentences.

In this context, this short paper overviews an approach for fuzzy parsing that is targeted for reliably extracting information from partially correct or incomplete input texts. We claim that the recognition in our approach is precise while tolerant to the input's fuzziness. The underlying grammar is partial w.r.t. the original one. Therefore, our approach defines a minimum degree m of certainty (in the interval [0,1]), with which each recognized sentence belongs to the original language. The approach is based on deterministic finite automata where states define precise contexts within the sentences and edges represent potential matches of constructs of interest[1] inside each context. By using this notion of contexts, the search space becomes smaller, reducing both the time to recognize the input and the ambiguity conflicts.

Since this is a starting project, this paper only scratches the surface of the approach.

Outline. The reminder of the paper provides a running example in Section 2 to make the discussion of the paper clear. Related work on the two mentioned approaches, island and fuzzy grammars, is presented in Section 3. Then, in Section 4 it is introduced our approach for fuzzy parsing, highlighting its main aspects and features. Finally, in Section 5 we conclude the paper, with a discussion of our approach in comparison with others, and also, presenting the steps for future work.

2 Running Example

Program comprehension [17, 18] is a research field aiming at studying and providing means to allow software engineers to understand legacy code, which is intended to be evolved. Several techniques have been proposed [16, 6, 5, 4, 10] for the elaboration of high-level models that abstract aspects of the programs, known to be relevant for their comprehension. One such model is the class diagram of a system, where relations between the several entities implied in the system are shown along with their internal attributes and methods. Usually, to obtain such diagrams it is necessary to process the whole system and to construct the complete parsing tree, from where only a tiny part of it is of interest.

In this example, the intention is to extract relevant information from Java programs in order to create simplified class diagrams that only show relations of classes and ignores

[1] A construct of interest is any portion of the original language that is desired to be recognized.

methods and attributes. Clearly, only the definition of the headers of classes are relevant to analyse. Everything else is to be ignored. To add complexity, amounts the fact that there are several ways of implementing classes as can be seen a few in Listing 1.

Listing 1 Examples of class headers in the Java language.

```
public class A { . . . }
public class E extends Exception { . . . }
public interface I { . . . }
public class B extends A implements I { . . . }
```

3 Related Work

In order to fix notations for a better understanding of this section, a brief recall of the formal definition of Context-Free Grammar [7] is given. Let G be such a context free grammar. Formally it 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 function of the form $V \to (V \cup \Sigma)^*$, 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 string or any combination of symbols from V or Σ.
- $S \in V$ is the start symbol (or axiom) of G.

Informally, a context-free grammar, is a set of production rules that syntactically derive sentences of a formal language. These 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. A context-free grammar G defines a language, given by $\mathcal{L}(G)$, corresponding to all the sentences accepted by G.

3.1 Island Grammars

Island Grammars have been described in [15, 14]. They consist in grammars that are conceptually divided in two core sections: (i) productions where we specify constructs of interest – referred to as Islands – and (ii) productions designed to match the rest of the input – referred to as Water.

From a formal point of view, given a context-free grammar $G = (V, \Sigma, R, S)$, such and a set of constructs of interest $I \subset \Sigma^*$ such that $\forall_{i \in I}\ \exists_{s_1, s_2 \in \Sigma^*}\ .\ s_1\ i\ s_2 \in \mathcal{L}(G)$. An *island grammar* $G_I = (V_I, \Sigma_I, R_I, S_I)$ for $\mathcal{L}(G)$ defines a new language $\mathcal{L}(G_I)$, and has the following properties:

1. $\mathcal{L}(G) \subset \mathcal{L}(G_I)$, this is, G_I generates an extension of $\mathcal{L}(G)$;
2. $\forall_{i \in I}\ \exists_{v \in V_I}\ .\ v \to i \wedge \exists_{s_3, s_4 \in \Sigma^*}\ .\ s_3\ i\ s_4 \notin \mathcal{L}(G) \wedge s_3\ i\ s_4 \in \mathcal{L}(G_I)$, this is, G_I can recognize constructs of interest from I in at least one sentence that is not recognized by G.

Let's consider the running example introduced in Section 2. Listing 2 shows the base specification for an Island Grammar that recognizes the headers of classes in Java. First it is defined the start symbol input that derives in a (possibly empty) sequence of chunks. Then, all chunks derive in either water or island. In this particular case islands will match the constructs of interest for analysing class headers.

Listing 2 Island grammar intended to recognize classes in Java.

```
input   : chunk*

chunk   : island | water

island  : 'class' ID extend? impls?
        | 'interface' ID
extend  : 'extends' ID
impls   : 'implements' ID (',' ID)*
        ...

water   : .*

ID      : [A-Z][A-Z0-9]*
```

Although simple at first glance, this grammar is able to recognize all class headers in Java source code. More complex problems may imply the implementation of new islands, producing an almost tailored made parser.

3.2 Fuzzy Grammar

A fuzzy parser [12] 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 marks are commonly referred to as anchors.

From a more formal stand, considering a context-free grammar $G = (V, \Sigma, R, S)$, a fuzzy parser $F(G)$ is defined as $F(G) = (V', \Sigma, R', A, S)$, where V' is a set of anchor nonterminals, Σ remains the alphabet of the language, R' is the provided set of rules of type $V' \rightarrow A \times (\Sigma \cup V')*$, $A \subseteq \Sigma$ is the set of anchor symbols and S is the start symbol.

An anchor flags, thus, the beginning of a substring that is meant to be recognized by $F(G)$; this means that each $s \in \mathcal{L}(G)$ contains at least one anchor that is accepted by $F(G)$. For each anchor $a \in A$ there is a rule $r_a \in R'$ that specifies the substring of s that is meant to be recognized. In conclusion, the language $\mathcal{L}(F(G))$ that is partially accepted by $F(G)$ can be described as follows: $\mathcal{L}(F(G)) = \{s \in \mathcal{L}(G) | s = \omega_1 a \omega_2 \wedge \omega_1 \in \Sigma^* \wedge \omega_2 \in \Sigma^* \wedge a \in A\}$, where $S \rightarrow^* s$ (s derives from S by multiple applications of rules in R).

Considering again the running example, it can be easily re-engineered to conform to this notion of Fuzzy grammars. The obtained grammar is showed in 3.

The main difference to the island grammars approach is the implicit existence of a water section that consumes characters not matching anchors and associated rules.

4 Our Approach

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

Listing 3 Fuzzy grammar able to recognize classes in Java.

```
 input   : anchor*

 anchor  : 'class' ID ext? impls?
         | 'interface' ID
 ext     : 'extends' ID
 impls   : 'implements' ID (',' ID)*
         ...

ID       : [A-Z][A-Z0-9]*
```

and reduce the search space for constructs of interest. Since this is a fresh idea, by the time of writing of this paper, no formal definition was defined for this approach. Therefore, the following presentation is rather informal, and kept at a high level of abstraction following the running example of Section 2.

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 behaviour 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 behaviour 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.

To make things clear, let's build on the running example by adding a new level of complexity: recognizing methods and their arguments inside each class. In the following, notation @X and $>> X$, is considered meaning respectively, the definition of a new context X and the definition of a transition to the concrete context X. In the latter, X may be substituted by $\wedge$ to express a transition to the parent context. Optionally, $>> X[A]$ may be used to specify that after transiting to context X, the input text must be read from before the previously consumed anchor A, and not from the current position.

The grammar begins with a default context (*@default*). In this context, when an anchor `class` is recognized (along with the associated construct of interest) a transition is made to the *@cls_hdr* context:

```
@default
 class   : 'class' ID >> cls_hdr
 iface   : 'interface' ID
```

Inside the *@cls_hdr* context, two constructs of interest are defined that detail the relationships of the class (*Extends* and *Implements*) with other classes. This means that the search space in this context is limited to these two constructs. However, since the body of the class is also required to be analysed, an extra construct '{' shall be recognized that defines a transition to the new context *@cls_bdy*:

```
@cls_hdr
ext      : 'extends' ID
impl     : 'implements' ID (',' ID)*
cls_in   : '{' >> cls_bdy
```

To unburden notation and recognition complexity, the `(',' ID)*` construction on `impl` rule could be defined within a new context as follows:

```
@cls_hdr
...
impl     : 'implements' >> ids
...

@ids
id       : ID
```

Besides alleviating the notation, this version modularizes the grammar as *@ids* context may be reused from another context or construct of interest. Moreover, details like `','` symbols are automatically ignored and all the `ID` symbols are processed as a unit and not as a list, which may be beneficial when extracting and transforming data. However, extra rules would be needed to define when to transit to a different context, since *@ids* may not be desired to be final, implying more parsing time. Also, the notion of concrete *context* is lost (contradictory to the philosophy of the approach), being only possible to recover it by taking advantage of semantic actions (not in the scope of this paper).

When in the context of the body of a class (*@cls_bdy*), for this example, only the constructs for public methods and private instance attributes are of interest. Notice that methods also include class constructors and that, in Java, they have different syntax:

```
@cls_bdy
cons     : 'public' ID '(' >> args
mthd     : 'public' TYPE ID '(' >> args
mthd_in  : '{' >> mth_bdy
attr     : 'private' TYPE ID ';'
```

However, both constructors and methods define their arguments following the same syntax. Therefore, and following the approach exemplified above, a new context (*@args*) is defined to parse the arguments. This is an example of reusing a context from different constructs of interest:

```
@args
arg      : TYPE ID
arg_ext  : ')' >> ∧

@mth_bdy
mth_ext  : '}' 'public' >> ∧ ['public']
```

As explained above, an extra rule is added to the *@args* context in order to pop it from the stack and returning to the parent context (*@cls_bdy*, in this example). Such a *pop* occurs when `')'` anchor is matched.

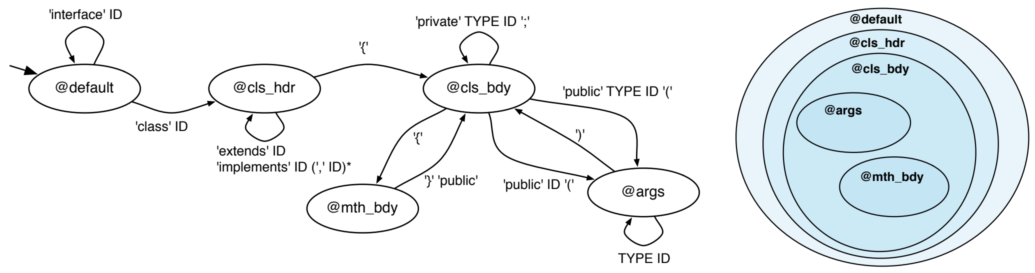

Figure 1 Automata and stack representation of the *unfuzzied* fuzzy grammar for the Java classes recognition.

Still in the *@cls_bdy*, when `'{'` occurs, the parsing transits to *@mth_bdy*, *i.e.*, the body of a method. Since for the purpose of the running example, nothing is required from the interior of a method, the grammar only pops the context when `'}'` is recognized. Everything else is ignored. The recognition of `'public'` after the `'}'` is necessary, so that other `'{'` and `'}'` (representing blocks inside the method) will be ignored. But notice that recognition of `'public'` is discarded once going to the parent context, in order to be recognized again as part of the *@cls_body* context. The careful reader may be asking what happens when there is no `'public'` anchor to leave *@mth_body* context. This occurs, for instance, on the definition of the last method of a class. In these cases the parsing will finish in the *@mth_bdy* context, as there is nothing else of interest to recognize at that point. Since every context is a final state in the automaton, then the recognition correctly accepts the input.

As expected, this last context as well as the `mthd_in` rule could be left out of the grammar. This is only included to show that from a context, it is possible to go to more than one different contexts.

The main difference of this approach to the ones presented in Section 3 resides in the usage of contexts while processing the input. This allows for a finer assessment of data, because the constructs of interest are limited to a smaller domain. This, in turn, allows for the reduction of uncertainty that is inherent to robust parsing, and to fuzzy parsing in particular (thus the title of this paper).

5 Conclusion

In this paper we informally presented an approach to fuzzy parsing that reduces the uncertainty of what is being recognized. The introduction of a notion of contexts (as states of an automaton) provide such disambiguation and goes beyond diminishing the search space for constructs of interest. The reduction of uncertainty is more valuable when the objective is to extract reliable information from the parsed input, by means of semantic actions. Semantic actions were not covered in this paper due to space limitations and lack of completeness by the time of writing. However it seems easy to understand the role of contexts to this end: at each context, it is known exactly what each syntactic construction means; for instance, `TYPE ID` can represent in a context a variable declaration whereas in another one it may be the declaration of a class attribute.

A key point of this approach is that only a part of a parsing tree will be generated. Although not covered by this paper, several approaches decided to recognize the entire source, sometimes even making multiple parses [2, 3, 9]. Again, the precise definition of the processing steps for our approach is left for future work. Nevertheless, the automata- and stack-based operational behaviours seem to provide basic intuition about it.

References

1 Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

2 Peter R. J. Asveld. A fuzzy approach to erroneous inputs in context-free language recognition. In *Proceedings of the Fourth International Workshop on Parsing Technologies IWPT'95*, pages 14–25, Prague, Czech Republic, 1995. Institute of Formal and Applied Linguistics, Charles University.

3 Peter R. J. Asveld. Fuzzy context-free languages – Part 2: Recognition and parsing algorithms. *Theoretical computer science*, 347(1):191–213, 2005.

4 Mario Berón, Pedro R. Henriques, Maria J. Pereira, and Roberto Uzal. Program inspection to incerconnect behavioral and operational view for program comprehension. In *York Doctoral Symposium, 2007.* University of York, UK, 2007.

5 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 (co-located with ETAPS'07)*, 2007.

6 Elliot J. Chikofsky and James H. Cross. Reverse engineering and design recovery: A taxonomy. *Software, IEEE*, 7(1):13–17, 1990.

7 N. Chomsky. Three models for the description of language. *Information Theory, IRE Transactions on*, 2(3):113–124, 1956.

8 John A. Fitzgerald, Franz Geiselbrechtinger, and Mohand Tahar Kechadi. Application of fuzzy logic to online recognition of handwritten symbols. In *IWFHR*, pages 395–400, 2004.

9 John A. Fitzgerald, Franz Geiselbrechtinger, and Mohand Tahar Kechadi. Structural analysis of handwritten mathematical expressions through fuzzy parsing. *ACST*, 6:151–156, 2006.

10 Yann-Gaël Guéhéneuc. A theory of program comprehension: Joining vision science and program comprehension. *International Journal of Software Science and Computational Intelligence*, 1(2):54–72, 2009.

11 Alexander S. Kechris. *Classical descriptive set theory*, volume 156. Springer-Verlag New York, 1995.

12 Rainer Koppler. A systematic approach to fuzzy parsing. *Software Practice and Experience*, 27:649, 1996.

13 Scott MacLean and George Labahn. Recognizing handwritten mathematics via fuzzy parsing. Technical report, Tech. Rep. CS-2010-13, School of Computer Science, University of Waterloo, 2010. 3, 2010.

14 Leon Moonen. Generating robust parsers using island grammars. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 13–22, 2001.

15 Leon Moonen. Lightweight impact analysis using island grammars. In *Program Comprehension, 2002. Proceedings. 10th International Workshop on*, pages 219–228. IEEE, 2002.

16 Václav Rajlich and Norman Wilde. The role of concepts in program comprehension. In *IWPC'02: Proceedings of the 10th International Workshop on Program Comprehension*, pages 271–278, Washington, DC, USA, 2002. IEEE Computer Society.

17 Scott Tilley. 15 years of program comprehension. In *Program Comprehension, 2007. ICPC'07. 15th IEEE International Conference on*, pages 279–280, 2007.

18 A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.