Rosetta ANTLR: Ultimate Grammar Extractor

Ewout van der Wal
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
e.w.vanderwal@student.utwente.nl

ABSTRACT

Parsers, and the grammars used to express them, have become geared towards solving for ambiguous parsing paths by using look-ahead and control structures such as semantic predicates to decide a language. This creates a problem when attempting to extract a grammar to a generalized structure for use in grammar comparison. While there is an existing ANTLR to BGF extractor, this implementation is outdated and generalizes away semantic information during the extraction process. We perform a replication for the tool for use with ANTLR4 and create a novel enhanced extraction methodology that improves the accuracy of the extractor. In the future, this would allow for more accurate analysis of grammars written in ANTLR.

Keywords

Grammar Extraction, Grammar Extensions, ANTLR, BGF, BGF++

1. INTRODUCTION

Over the past decades, parsing techniques have become less restrictive in the grammatical structures that can be expressed. Classic parsers such as *LL(k)* put hard limits on the look-ahead depth of the parser. Later, Ford introduced the *PackRat* parsing strategy and Parsing Expression Grammars (PEG)[7, 8]. These improved on earlier parser by implementing a backtracking and memoization strategy to work with all *LL(k)* and *LR(k)* grammars in linear time. However, PEG-based parser would still prove to have limitations [14].

To overcome the limitations of the *PackRat* parsing strategy, *LL(*)* was introduced in 2011 by Parr and Fisher [14], followed by *ALL(*)* in 2014 [15]. Unlike *PackRat*, these parsers will look ahead as far as is necessary to determine which parsing rule to apply instead of using the first matching rule. Also, *LL(*)* and *ALL(*)* allow for a great degree of freedom by way of semantic actions and other grammar extensions.

In practice, the *LL(*)* and *ALL(*)* parsers are the basis for the ANTLR3 and ANTLR4 parser generators respectively, both of which are widely used [14, 15, 18].

34th Twente Student Conference on IT Jan. 29th, 2021, Enschede, The Netherlands.

In grammar extraction, a grammar defined in a parser specific language such as ANTLR is converted into an implementation agnostic representation. Such a representation could be the BNF-like Grammar Format (BGF) used by Zaytsev [18], Lämmel and Zaytsev [12]. Accurate grammar extraction allows for grammar comparisons, regardless of implementation details, and is an active research topic. Studies such as those by Zaytsev [18], Lämmel and Zaytsev [12] into the evolution of parser grammars and by Fischer et al. [6] into the differences between parsers for the same version of a language show clear use cases for grammar extraction.

Grammar extraction is hampered by the implementation specific structures or semantic actions present in the source grammars. For example, the ANTLR3 to BGF extractor used in the Grammar Zoo project [18] disregards any kind of semantic information. While this makes extracting a general structure for a grammar much easier, it causes a loss of information such that the language accepted by the extracted BGF grammar is not the same as the original grammar. This makes comparing grammars extracted with this tool difficult and inaccurate.

In software engineering, as in many other facets of scientific research, there exists the notion of research replication as a means of verifying existing research, perhaps in light of new insights [13]. As shown by Gómez et al. [9], there is a wide variety of terms used to label exactly the type of replication a particular research is attempting. In light of this, we will be using the definitions put forward by Krein and Knutson [11] as interpreted by Gómez et al. [9]. Our goal can then be defined in two parts:

- **Goal 1:** To perform a dependent replication [9] of the ANTLR3 to BGF grammar extractor created for use in the Grammar Zoo [18]. The resulting extractor will consume grammars defined using ANTLR4 and produce a generalized structure, likely similar to the BGF structure defined in the original study.
- **Goal 2:** To create a novel methodology with which to extract the semantic actions, implicit disambiguations, and other grammar extensions specific to ANTLR4 into a generalized notation, which is agnostic to implementation. These new extraction techniques are to be implemented on top of the extractor resulting from **Goal 1**.

To achieve these goals, we will be using the following research questions (RQ) as the basis of our research:

- **RQ1:** What kinds of semantic actions, implicit disambiguations and other grammar extensions are used in real life ANTLR4 grammars? What are the defining traits of these ANTLR specific grammatical structures?
- **RQ2:** Which of the grammatical structures identified in **RQ1** can be generalized into an extracted grammar?
- **RQ3:** How do we define and implement a grammar that accurately captures these grammatical structures without loss of generality?

Our research contributes to the field of grammar extraction and analysis in two ways. First, we have created a replication of the ANTLR to BGF extractor used in the Grammar Zoo project, which is able to extract ANTLR4 grammars. Second, we have created a novel set of extraction rules for the grammar extensions specific to the ANTLR4 language. The result is a new generalized language for more accurately extracting ANTLR grammars, called BGF++.

The structure of the paper is as follows. In section 2 we will give an overview of the related works in the field of grammar extraction. Section 3 will detail the methodology and describe the dataset. In section 4 we will go into detail about what results we have been able to achieve, both in analysing the grammars in the dataset and in creating BGF++. Finally, we will conclude in section 5 and shortly discuss the research in section 6.

2. RELATED WORK

In this section we will go over some of the related work in grammar extraction.

Grammar extraction has been a topic of research at least as far back as 1996 [16]. These early efforts are geared at recovering grammars from an incomplete or unreliable documentation or parser implementation, often aimed at recovering the grammars for Domain Specific Languages (DSL).

In their 2002 paper Kort et al. [10] present the Grammar Development Kit (GDK), a toolkit providing support in grammar engineering which is optimized for COBOL. One of the functionalities of the GDK is that grammars can be extracted from a language reference such as documentation. Like in many of the later technologies, the GDK internally uses a standardized grammar format, in this case LLL.

In 2011, Lämmel and Zaytsev [12] describe a method for using grammar convergence for comparing different versions of Java grammars extracted from the Java Language Specification (JLS). As part of this comparison, the grammars are extracted from the JLS into BGF. By extracting to BGF, it is possible to compare and modify the different Java grammars using one strict, internally standardized format.

The work by Fischer et al. [6] shows how grammar extraction is used to make comparisons between grammars written to follow the same language specification. They show that even though the authors of the grammars use the same information, there can be structural differences between the resulting grammars. As an application of this type of grammar extraction and comparison the authors propose to use this technique as an automated method of comparing grammars made by students studying Computer Science to a reference solution created by their professor.

Work by Zaytsev [18] in 2015 on the Grammar Zoo uses the same BGF notation. As a result of the Grammar Zoo effort, there is an open-source ANTLR2 and ANTLR3 to BGF converter already available. As part of our efforts, we have replicated this grammar extraction tool for use with ANTLR4.

3. METHODOLOGIES

In this section we will explain the methodology for the replication of the existing grammar extractor and the analysis with as goal the extension of the BGF syntax.

As explained earlier, a tool to extract previous versions of ANTLR to the generalized BGF syntax already exists [5]. This tool was largely unusable as it required Python2 to run, a version of Python that has been largely deprecated. The replication would therefore be easier to do from the ground up.

Seeing as it was not feasible for us to write a grammar for ANTLR ourselves, we decided to use the grammar that the ANTLR project provides[1]. However, the grammar and corresponding implementation did not accept the same inputs as the ANTLR tool. This required us to fix these issues before moving on.

With the grammar now accepting inputs as expected, we could start replicating the Grammar Zoo extraction tool. The documentation for this tool provided us with a set of BGF concepts that could be encoding in an XML format. Using our grammar, we generated a parse tree visitor that would produce an XML file with the BGF representation of an ANTLR4 grammar. Like in the original tool, many features were left out, because they could not be converted to BGF.

3.1 Data Source

To get an idea of the features that are unique to ANTLR4 grammars, we would first need to acquire a dataset of, preferably real-world, example grammars. Github is an easy to use, online version control website hosting a sizable number of projects that contain an ANTLR4 grammar and are publicly available. On top of that, Github provides a REST API with which it is possible to automatically query all public repositories.

We created a Javascript application that would use the axios library to send a search request to the Github API. This request queried the "`api.github.com/search/code`" endpoint with the search parameters "`?q=grammar+extension=g4`", which returns references to all public files containing the word grammar, and with the correct ANTLR4 extension. One of the fields in the search API response to a search request is the URL from which the file corresponding to the request can be downloaded. From there, we were able to extract only these URLs to a file on disk. Next, we iterated over these URLs and downloaded the grammars.

One issue with this approach was that it did not preserve the names of the files that were downloaded, so in order for ANTLR to recognise them as proper grammars, we had to extract them from the content of the file. After this was complete, we had a dataset of 370 grammars.

3.2 Dataset

Of the 370 grammars downloaded from Github, the ANTLR tool was unable to parse six grammars. Our tool had issues with one other grammar, where the name of one of the productions was an ANTLR keyword. Renaming the production fixed this issue, so only the incorrect grammars were left out of the further analysis. This left us with a dataset of 364 correct, real-world grammars from which we could extract any features specific to ANTLR that have an effect on the language accepted by the grammars.

3.3 RQ1

From here we can go on to identify the constructs that are unique to ANTLR that are not currently extracted to BGF. By looking at the documentation, digging through the reference grammar[1], and going over the dataset we compiled a list of constructs that we would need to look at. This list contains all of the constructs that we can identify, some of which are possibly not interesting for our usecase.

With the list of all features we create a matrix that for each grammar in the dataset marks which of the ANTLR specific features are found in that grammar. For many of these, we also identify commonly occurring patterns within the feature. We can use this information later on to get a better understanding of the makeup of our dataset, and to estimate the impact of adding a feature to BGF++.

This methodology bears resemblance to the research strategy called Grounded Theory [17]. The aim of Grounded Theory, as defined by Tie et al. [17], is to "generate theory that is grounded in the data." Grounded Theory is an appropriate strategy when little is known about the subject. Instead of starting from the beginning, we are able to leverage the information from the documentation and reference grammar to speed up the identification of relevant features.

3.4 RQ2

In order for any ANTLR specific construct to be considered for BGF++ it has to change the accepted language of the grammar, and be feasible to implement. For some of these constructs, the original BGF language specification does already allow the construct to be extracted, but the previous extraction tool does not do this. In these cases, we choose to implement these extractions without extending the BGF notation.

3.5 RQ3

Next, we determine what extensions to the BGF notation are necessary in order to extract each of the ANTLR specific constructs. The goal is to use as few extensions as possible to achieve total coverage of the features we will be extracting.

Finally, the list of required extensions to BGF is implemented into BGF++.

4. RESULTS

In this section we will go over the results we were able to achieve over the course of this research.

The first task we set out to do was to replicate the previous ANTLR to BGF extractor for use with ANTLR4, and to then improve on this implementation by increasing the accuracy. We achieved this goal, and created a replication in C# with the new additions[4].

Although we use the same BGF syntax in our replication, performing replications on the same grammar in ANTLR3 and ANTLR4 using the two tools does not produce exactly the same output. No grammatical information is lost, but the new tool does away with many `<expression>` and `<sequence>` nodes. The replicated extractor guarantees that nodes on the same level within a production are a sequence, and an expression is always captured in one node. For example, where the original tool would create:

```
<expression>
    <sequence>
        <expression>
            <nonterminal>rule_name</nonterminal>
        </expression>
        <expression>
            <nonterminal>rule_name_2</nonterminal>
        </expression>
    </sequence>
</expression>
```

The replication simplifies this to:

```
<expression>
    <nonterminal>rule_name</nonterminal>
    <nonterminal>rule_name_2</nonterminal>
</expression>
```

This reduces the footprint of the output of the tool. That the two `<nonterminal>` nodes belong to the same sequence is implied by their existence on the same level within the surrounding `<expression>` node.

4.1 Enhancing the replication

The original tool did not make complete use of the existing BGF specification. Before moving on extending the BGF syntax, we can first exhaust the existing options. BGF already has the `<epsilon>` node, which allows us to specify that a production alternative accepts the empty string as input. Also, BGF supports Boolean grammars, where the `<not>` node means the input is accepted if it does not match what is inside the `<not>`. In ANTLR, the `~` construct allows for character set negation. For example, the rule `NOT_A: ~[A];` matches any character that is not the uppercase letter `A`.

There are two other constructs that we implemented, which we do not list in the results below, because they are not constructs specific to ANTLR. However, they did require additions to the BGF syntax.

The first is character sets, which are not implemented in the original tool. ANTLR allows character sets to have UTF-8 characters, Unicode escape strings, and one or more character ranges. In order to extract exactly what characters are captured by the character set, we introduce the `<charSet>`, `<charRange>`, and `<char encoding="...">` nodes. The encoding of a character is set to UTF-8 normally, and to Unicode when a Unicode escape string is detected.

Second, we added support for extracting `*?` and `+?`. Normally, the `*` and `+` operators will greedily consume as much input as they can. The question mark makes the evaluation lazy, preferring to consume as little input as possible. The `<star>` and `<plus>` will now have a `greedy` attribute.

Before implementing any enhancements to the tool that extract ANTLR specific constructs, we can already perform a more accurate extraction of ANTLR grammars than the original tool.

4.2 RQ1

There are two distinct types of constructs specific to ANTLR: code written in the parser target language injected into the grammar, and ANTLR keywords. Table 1 lists these constructs and in how many grammars in the dataset each construct can be found.

ANTLR construct	Outcome	# of grammars
@header{...}	Not extracted	90
@members{...}	Not extracted	46
@rulecatch{...}	Not extracted	2
@init{...}	Not extracted	7
production arguments	Not extracted	14
production locals	Not extracted	7
production returns	Not extracted	29
identifier = expression	Not extracted	85
{...}?	Partially extracted	17
{...}	Not extracted	55
fragment	Extracted	103
-> pushMode(...)	Extracted	20
-> mode(...)	Extracted	4
-> popMode	Extracted	21
-> skip	No effect	166
-> more	No effect	1
-> type(...)	No effect	9
-> channel(...)	No effect	48
import ...	Extracted	65
options{tokenVocab}	Extracted	76
options{language}	Extracted	18
tokens{...}	Extracted	18
<assoc=...>	Extracted	8

Table 1. All ANTLR features

Code can be injected into a grammar in three ways. The first is to use the `@header{}`, `@members{}`, and `@rulecatch{}` keywords. Code between the curly braces following `@header` is added to the top of the generated parser, before the class definition. In the dataset this is almost exclusively used to add package definitions, imports, and doc comments. The `@members` keyword works in the same way, but the code is added within the parser class. The dataset shows that this is used to add various private variables and and functions to the generated parser class. The `@rulecatch` keyword is used to define custom behaviour when the generated parser encounters an error.

Another method is to insert code into one of the grammar productions is to declare production parameters, local variables and return values. These are defined using `@init`, `locals`, `returns`, and the square brackets next to a production name, and can be used by other code in the rule. A terminal or nonterminal may also be given an identifier by which code inserted into the surrounding production may interact with it.

The third method is to insert actions and predicates into the productions using `{}` and `{}?` respectively. Actions contain code that is run once the parser has successfully entered an alternative in a production. Predicates are run before an alternative is entered by the parser and are used to conditionally allow or deny access to a certain parse path.

The remaining keywords can be split by whether they are used by the lexer or the parser. In the lexer ANTLR uses the `fragment`, `-> pushMode()`, `-> mode()`, `-> popMode`, `-> skip`, `-> more`, `-> type()`, `-> channel()`, and `~` constructs.

A production with the `fragment` modifier does not produce a token, and these productions must always be used in another lexer production.

Lexer modes allow the lexer to enter different sub-grammars within one overarching grammar. The `-> pushMode()`, `-> mode()`, and `-> popMode` keywords control the lexer moving between these grammars. Later, we will go into more detail about the difference between `-> mode()` and `-> pushMode()`.

The other arrow functions `-> skip`, `-> more`, `-> type()`, and `-> channel()` are used to control the output of the lexer after an input has been successfully matched to a token. When analysing a grammar we are interested in the language accepted by the generated parser. Since these four functions do not change the accepted input, they can be safely ignored.

The final four rows in table 1 are used in the parser. Grammar files can be imported into one another with `import`, and extra tokens are defined with `tokens{}`. ANTLR also supports the grammar options `superClass`, `language`, `tokenVocab`, `tokenLabelType`, and `contextSuperClass`. Of these, we extract the `tokenVocab` and `language` options. The former is used to import tokens from another grammar, and the latter defines the language that the parser will be generated in (the default is Java).

Finally, `<assoc=...>` is used to define the associativity of a production alternative. By default ANTLR is left-associative, and this modifier can be used to make a production right-associative.

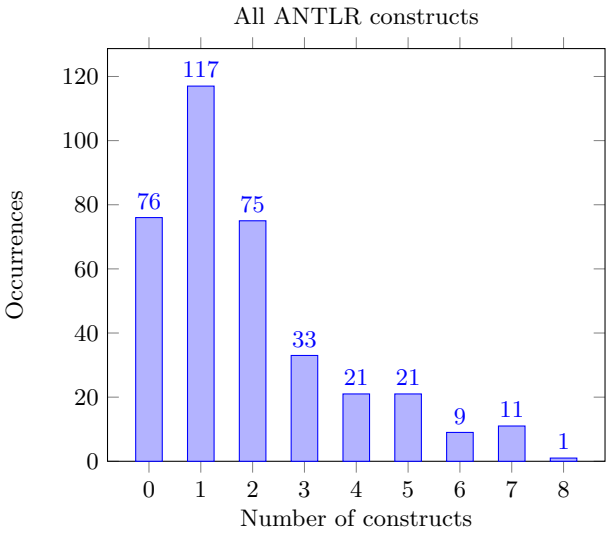

Figure 1. Constructs labeled No effect in table 1 are not included.

Figure 1 is a histogram of the amount of ANTLR constructs that are listed above, found in the grammars in the dataset.

4.3 RQ2

From this understanding of ANTLR we can determine which of the constructs listed in the previous section can be extracted to BGF++. An overview of these choices can be found in table 1. In this section we will go into further detail for some of these choices.

All of the ANTLR constructs that we label `Not extracted` are directly or indirectly only useful when using actions and predicates. Actions are used in 55 out of the 364 total grammars in the dataset. From that number, it would

seem as if we are still losing information in 15.11% of the grammars in the dataset. However, this changes when we also look at the makeup of these actions. Figure 2 shows that of these 55 grammars, we have determined that 41 (74.55%) of these grammars had actions with no effect on the language accepted by the grammars. In these cases, the actions would for example be used to generate a custom parse tree, which does not have to be done in the grammar file. ANTLR generates a parse tree visitor and listener base class specifically for this purpose. This leaves us with 14 grammars, amounting to 3.85% of the dataset, with effects that are either unclear or do change the accepted language.

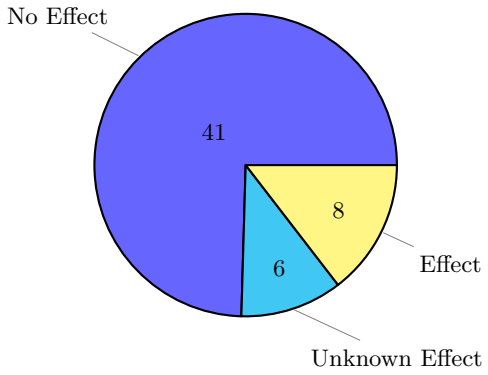

Figure 2. Makeup of grammar actions

The other notable row is the predicate (shown as `{...}?`), which we have labeled `Partially extracted`. Predicates are shorter pieces of code only meant to output a boolean value. While it is impractical for us to try and parse these predicates, it is possible to extract this information in some form. As long as this information is not lost, an analysis of the extracted structure done on a case by case basis could prove to successfully utilise this information anyway without cluttering the BGF++ syntax too much.

There are four rows with the `No effect` label. These are constructs that ANTLR uses, but that do not have any effect on the accepted language of the grammar. We list them in table 1 for completeness sake, and have not included them in any of the other graphs.

Finally, we have also identified a number of constructs that we are able to extract to BGF++.

Figure 3 is a histogram of the amount of ANTLR constructs remaining in the grammars in the dataset after removing the constructs that we are able to extract. From this, we can see that 214 of the 364 grammars (58.79%) can be extracted to BGF++ with no loss of information about the accepted language. This is a marked increase over the 58 grammars seen in figure 1. Of the remaining 150 grammars, we know that there are 14 grammars with actions that cannot be determined to have no effect and 17 grammars with predicates. Due to overlap between these groups, we have 26 grammars (7.14% of the dataset) that cannot be extracted without a loss of information. The other 124 grammars (34.07%) will lose information, but this does not impact the language that the grammar accepts. In total, we can extract 338 grammars (92.86% of the dataset) with no loss of accuracy with respect to the accepted language of the grammar.

4.4 RQ3

From the previous section we know that we can extract a portion of the constructs specific to ANTLR. In this section we will explain how we extract these constructs from ANTLR to BGF++. Table 2 is an overview of all of the changes that are made in order to improve the extraction.

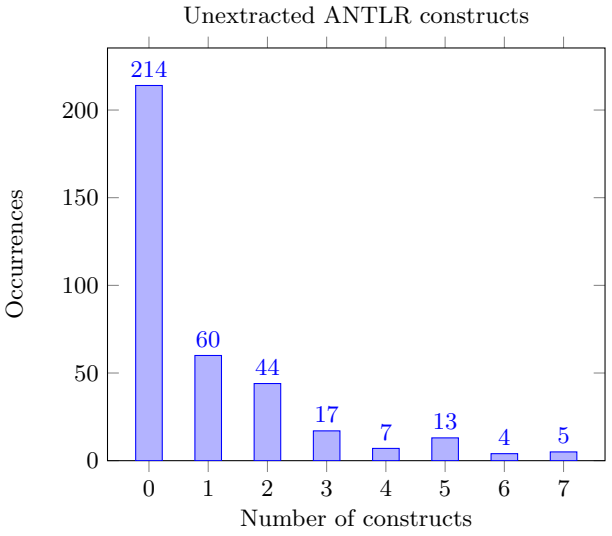

Figure 3. Constructs labeled No effect in table 1 are not included.

To extract the predicates we create a new `<predicate>` node. This node is placed in an alternative in the same place as the predicate was in the ANTLR grammar, and will contain the code that was in the predicate. An analysis that does not take predicates into account can simply skip this node.

As explained earlier, the `fragment` keyword is a modifier for a production. Therefore, we can add this information to the already existing `<production>` node as an attribute. If a production is a fragment, this would produce the XML node `<production fragment="true">`.

Modes in ANTLR are treated as separate grammars in BGF++. In order to extract the mode switching behaviour into BGF++ we need to be able to switch between these grammars. Like in ANTLR, this is done using a grammar stack. When the ANTLR lexer encounters the `pushMode()` command, the new mode is pushed onto the mode stack and the lexer moves to the corresponding mode. In BGF++, this is extracted as the `<pushGrammar>` node. The `popGrammar` command is extracted into the `<popGrammar>` node. There is also the `mode()` command in ANTLR, which does not add the new mode to the top of the stack. Instead, it replace the current head of the mode stack with the new mode. For the sake of simplicity, we first create a `<popGrammar>` node and immediately after it create a `<pushGrammar>` node. Any grammar analysis tool that uses BGF++ is expected to keep track of this stack themselves, and always make sure that the stack is non-empty whenever the analyser is in a grammar.

ANTLR grammars can be imported into other ANTLR grammars. For example, by splitting the lexer and parser and importing the lexer into the parser. To make sure this information is not lost, we create an `<importGrammar>` node for each of the `import` statements in the grammar.

Earlier, we determined there were two grammar options that had merit to extract. These are the `tokenVocab` and `language` options. The `tokenVocab` option allows for importing tokens from other grammars, which we extract as the `<importTokens>` node in order to set it apart from the

ANTLR construct	Extraction to BGF++
{...}?	Create a new <predicate> node
fragment	Create a new "fragment" attribute
-> pushMode(...)	Create a new <pushGrammar> node
-> popMode	Create a new <popGrammar> node
-> mode(...)	Combine a <popGrammar> and <pushGrammar>
import ...	Create a new <importGrammar> node
options{tokenVocab}	Create a new <importTokens> node
options{language}	Create a new <targetLanguage> node
tokens{...}	Use the existing <production> node
<assoc=...>	Create a new "associativity" attribute

Table 2. All ANTLR features

grammar import. The `language` option is extracted to the `<targetLanguage>` node. Although the target language for the parser generation does not change the accepted language of a grammar, it does make it easier to determine what language a predicate is in. It is also possible to provide this as a command line option, so not seeing this option does not guarantee that the target language is Java.

Importing other grammars and setting the grammar options is done in the preamble of an ANTLR grammar, before the first production definition. This means that we can also create a `<preamble>` node that contains all `<importGrammar>`, `<importTokens>`, and `<language>` nodes.

Aside from importing tokens from other grammars, ANTLR also supports extra token definitions in the preamble using the `tokens{}` construct. These imaginary tokens are recognised by ANTLR as correct token types, but do not contribute to parsing the input. Instead, they can be used to add information to the resulting parse tree. Tokens are extracted into `<production>` nodes, just like any other production. The resulting productions , the production will have an `<empty>` node to show that there is no associated input. These token definitions are not put in the preamble, because they can be treated as any other grammar production.

4.5 Comparing the two tools

In order to compare the two tools, we looked at two extractions done by the original tool [2, 3]. The first grammar is for Ada, written in ANTLR2. The second grammar is for Google Dart, written in ANTLR3. For both grammars, we translated them to ANTLR4 by hand to the best of our ability. These translations were then passed through the ANTLR4 to BGF++ extractor.

The largest difference between the original extractions of these grammars and the newly extracted BGF++ versions is the extraction of the content of the `tokens{}` keyword. Both grammars define a list of tokens, which the original tool throws away. The new implementation preserves this information.

Also, as explained earlier, our enhanced tool uses less `<expression>` and `<sequence>` nodes than the original tool. As a result, the BGF++ representations of the grammars are smaller than the extractions to BGF, while also preserving more of the information in the grammar.

5. CONCLUSIONS

This research had two objectives: to replicate an existing ANTLR3 to BGF grammar extractor written in Python2 to extract ANTLR4, and to investigate the various ANTLR specific constructs that prevented the previous tool from accurately extracting ANTLR grammars.

We were successful in the replication, using a grammar for ANTLR created by the ANTLR project themselves and kept up to date by the community. This grammar, especially the C# superclass implementation, ended up having some errors that we needed to fix first. These fixes have found their way back into that repository. Our replication gave us a good place to go on to achieve the second goal.

Our second goal was to create a novel methodology for extracting ANTLR specific constructs from grammars to a generalized notation. We did a survey of 364 grammars downloaded from Github, from which we identified a list of constructs that we could implement as extensions to BGF. The result of this is that we have a much better understanding of what ANTLR features are used in real world grammars.

With this knowledge we were able to implement a grammar extractor that could extract 214 grammars without needing to remove any in-grammar code, and extract up to 338 grammars from our dataset while only losing code with no effect on the accepted language.

6. DISCUSSION

There are a couple of points that need to be addressed regarding the research laid out in this paper.

For one, the extraction is far from perfect, absolutely no effort is made to preserve in-grammar code. It would have been unfeasible to, given the amount of time available, attempt to extract, parse, or analyse this code in any meaningful way. This especially, because the code in the grammars was written in multiple languages, often with no indication which language was used.

Second, the current implementation of the extractor is far from optimised. For example, currently the extractor treats imported grammars as simply as string to be displayed. A meaningful upgrade would be to actually perform the import and to combine the grammars into one, larger grammar that could be fully analysed. The same goes for the `tokenVocab` option.

Finally, there are no rigorous tests in place to ensure that the extractor and the underlying grammar accept exactly the same language as the official ANTLR tool. One difference that we have already observed, and consequently ignored for the sake of moving forward, is that a rule which has the same name as a built in keyword is accepted by the official tool, but not by the extractor.

References

[1] URL https://github.com/antlr/grammars-v4.

[2] . URL https://slebok.github.io/zoo/#ada_ada95_kellogg.

[3] . URL https://slebok.github.io/zoo/#dart_google.

[4] . URL https://github.com/EwoutWal/ANTLR-Extractor.

[5] . URL https://github.com/grammarware/slps/tree/master/topics/extraction/antlr.

[6] B. Fischer, R. Lämmel, and V. Zaytsev. Comparison of context-free grammars based on parsing generated test data. In A. Sloane and U. Aßmann, editors, *Software Language Engineering*, pages 324–343, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-28830-2.

[7] B. Ford. Packrat parsing: Simple, powerful, lazy, linear time, functional pearl. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, page 36–47, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 1581134878. doi: 10.1145/581478.581483. URL https://doi.org/10.1145/581478.581483.

[8] B. Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, page 111–122, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 158113729X. doi: 10.1145/964001.964011. URL https://doi.org/10.1145/964001.964011.

[9] O. S. Gómez, N. Juristo, and S. Vegas. Understanding replication of experiments in software engineering: A classification. *Information and Software Technology*, 56(8):1033 – 1048, 2014. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2014.04.004. URL http://www.sciencedirect.com/science/article/pii/S0950584914000858.

[10] J. Kort, R. Lämmel, and C. Verhoef. The grammar deployment kit—system demonstration—. *Electronic Notes in Theoretical Computer Science*, 65(3):117–123, 2002.

[11] J. L. Krein and C. D. Knutson. A case for replication: synthesizing research methodologies in software engineering. In *RESER2010: proceedings of the 1st international workshop on replication in empirical software engineering research*. Citeseer, 2010.

[12] R. Lämmel and V. Zaytsev. Recovering grammar relationships for the java language specification. *Software Quality Journal*, 19(2):333–378, 2011.

[13] A. A. Neto. A strategy to support replications of controlled experiments in software engineering. *SIGSOFT Softw. Eng. Notes*, 44(3):23, Nov. 2019. ISSN 0163-5948. doi: 10.1145/3356773.3356796. URL https://doi.org/10.1145/3356773.3356796.

[14] T. Parr and K. Fisher. LL(*): The foundation of the antlr parser generator. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 425–436, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306638. doi: 10.1145/1993498.1993548. URL https://doi.org/10.1145/1993498.1993548.

[15] T. Parr, S. Harwell, and K. Fisher. Adaptive LL(*) parsing: The power of dynamic analysis. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, page 579–598, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450325851. doi: 10.1145/2660193.2660202. URL https://doi.org/10.1145/2660193.2660202.

[16] A. Sellink and C. Verhoef. Development, assessment, and reengineering of language descriptions. In *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering*, pages 151–160, 2000. doi: 10.1109/CSMR.2000.827323.

[17] Y. C. Tie, M. Birks, and K. Francis. Grounded theory research: A design framework for novice researchers. *SAGE Open Medicine*, 2019. doi: 10.1177/2050312118822927. PMID: 30637106.

[18] V. Zaytsev. Grammar zoo: A corpus of experimental grammarware. *Science of Computer Programming*, 98:28 – 51, 2015. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2014.07.010. URL http://www.sciencedirect.com/science/article/pii/S0167642314003347. Fifth issue of Experimental Software and Toolkits (EST): A special issue on Academics Modelling with Eclipse (ACME2012).