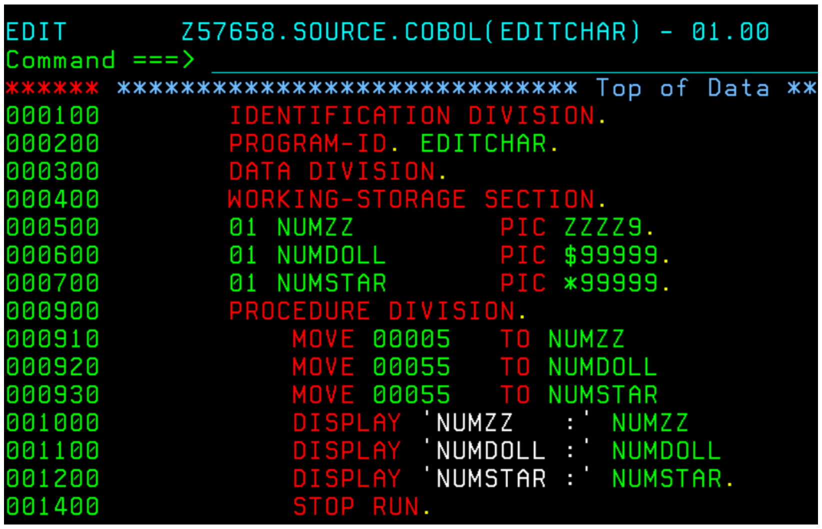
Parser Benchmarking for Legacy Languages

TOM MEULENKAMP, University of Twente, The Netherlands



```
EDIT       Z57658.SOURCE.COBOL(EDITCHAR) - 01.00
Command ===> 
****** ***************************** Top of Data **
000100        IDENTIFICATION DIVISION.
000200        PROGRAM-ID. EDITCHAR.
000300        DATA DIVISION.
000400        WORKING-STORAGE SECTION.
000500        01 NUMZZ          PIC ZZZZ9.
000600        01 NUMDOLL        PIC $99999.
000700        01 NUMSTAR        PIC *99999.
000900        PROCEDURE DIVISION.
000910            MOVE 00005    TO NUMZZ
000920            MOVE 00055    TO NUMDOLL
000930            MOVE 00055    TO NUMSTAR
001000            DISPLAY 'NUMZZ   :' NUMZZ
001100            DISPLAY 'NUMDOLL :' NUMDOLL
001200            DISPLAY 'NUMSTAR :' NUMSTAR.
001400            STOP RUN.
```

Fig. 1. A snippet from the COBOL language. [27]

Legacy languages are still being used in critical systems and are in constant need of support. Parsing frameworks play an important role in the construction of support tools. These are mainly focused on modern languages, but not on legacy languages. We do not really know how well these frameworks support legacy code features. In this paper, we evaluate one of these frameworks and its ability to support legacy programming languages.

CCS Concepts: • **Software and its engineering** → **Parsers**; **Preprocessors**.

Additional Key Words and Phrases: parsing, preprocessing, legacy, ANTLR, BabyCobol

1 INTRODUCTION

Today many programming languages are available on the market, with many of them aimed at easing the life of the programmer. However, dated languages are unfortunately still around in essential systems of many institutions [16]. Often, these languages are complex and come equipped with functionalities that can easily cause code smells. This makes them hard to maintain and analyse. A good example is COBOL [13]. It has been and is still being used in critical banking systems that are often too essential to be replaced and thus have to be maintained. Rewriting or reengering these systems is a notorious job and can sometimes not be achieved, due to risks or other constraints [22]. Examples of these constraints include the requirement to translate native language features to simulated constructs in the target language, or just pure oversimplification of the requirements for a given conversion tool. Therefore, it is important that tools can be developed to make the code more maintainable and hence more programmer friendly. These tools range from extensive debugging solutions to refactoring tools to simple syntax-highlighting.

All of these tools share one crucial requirement, the ability to efficiently and effectively scan and parse the written code. However, where most languages today have evolved to be more accessible and maintainable, older languages are usually more complicated and difficult to parse due to peculiar edge cases and possible ambiguities.

Hence, there is a need to check how compatible a given framework is with legacy languages. The contribution of this paper will be an implementation of the BabyCobol language in the ANTLR parser generator framework. And is guided by the following research question: "Can a modern parsing tool, like ANTLR, contribute to legacy code support and enhancement?" This research question will be answered with the following sub-research questions:

SRQ1 Is it possible to parse the BabyCobol language completely within the ANTLR framework?
SRQ2 Can this be done without resulting in complex structures that make use of more than the grammar language itself?
SRQ3 If complex operations have to be used, how much does this affect the parsing speed?

1.1 Parsers

The field of parsing concerns itself with "recognising grammatically formed sentences, providing error-correcting feedback, constructing graph-based representations, as well as optimising such algorithms on time, memory and lookahead" [33, p. 50]. The final goal of a parser is usually, but does not always have to be, to return a data

TScIT 37, July 8, 2022, Enschede, The Netherlands

structure, called an Abstract Syntax Tree (AST) [2, p. 227]. This tree can be used for all kinds of purposes with respect to language analysis. Mostly these are used by compilers, in order to generate machine code. Another application would be syntax highlighting in your IDE or auto-completion. Although parsing can happen in a lot of different steps, there are at least two steps present. The lexical analysis and the parsing of the tokens generated by the first process. These are shown in Figure 2.

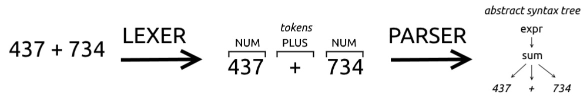


Fig. 2. The lexing and parsing phases [23]

1.1.1 Lexical analysis. Most programming languages work with keywords, numbers, and identifiers. All these values have their own structure and always come together. If one looks at the Java programming language, some keywords are: `class`, `static` or `void`. During the lexical analysis we divide the stream of characters that a parser receives into so-called tokens. Tokens are small pieces of text that will be fed to the parser. Usually each keyword has its own token, then there is a token for an identifier, integer, and other numbers. Other tokens are also available like brackets and semicolons. The collection of these tokens is done by a scanner, also known as a lexer. Regular expressions can also be used in tokens, such that they can for example match numbers of an arbitrary length [2, Chapter 2].

1.1.2 Parsing. Once we have retrieved all the tokens in the lexing phase, we can determine the structure the tokens are trying to convey. Just like natural languages, programming languages have a grammar, which the programmer should adhere to. The grammar for a programming language is defined as a list of statements that contain a name for a statement on the left-hand side and the tokens or other "grammar rules" on the right-hand side. These rules make use of regular expressions, just like the lexer does. An example can be found in Listing 1. Once the tokens and grammar rules are determined one can implement the parser [2, Chapter 3]. This can either be done completely by hand or (semi-)automatic. In our case, the parser will eventually return an AST based on the input text and the specific grammar rules and tokens. If the input text does not adhere to the specified grammar rules, the parser will return an error message and possibly a partial AST instead.

```
ifStatement: IF condition THEN body (ELSE body)?;
catchStatement: TRY body CATCH '(' expr ')' body;
```

Listing 1. An example of two grammar rules, where full uppercase words are tokens, and other words indicate other grammar rules

1.1.3 Semi-parsing. A paradigm within parsing is semi-parsing [6, 29], more specifically a version of it is called island parsing [25]. This form of parsing does not parse the entire input text that it receives, but the parser only takes out certain pieces of data and creates a structure according to that. This type of parsing is useful when one has to extract only small pieces of logic from a given input. Examples where semi-parsing is used are PHP, since it can be embedded in HTML [10]; and SQL code that is embedded within a piece of COBOL code [3, p. 20].

1.2 ANTLR

As described in the previous section, a parser can be written by hand, but it can also be (semi-)automatically generated. ANTLR [21] is a parsing framework that generates both the scanner and parser for you, based on a grammar and set of tokens that you provide to it. ANTLR makes use of the powerful parsing algorithm named Adaptive LL(*) [20]. This gives the language engineer a lot of freedom and ease when developing a grammar for a given language. Terence Parr, the creator of ANTLR, is an active member of the parsing and compiler community and understands the challenges that are faced during the development of a parser. Furthermore, the tool is very well documented [17, 18] and has an active community. It can also be used well for literature research based on grammars that have been written for ANTLR, since almost each grammar for a language can be found in the public GitHub repository [8].

1.3 BabyCobol

BabyCobol is a lab-made language that is specifically designed to challenge a software engineer to support legacy languages. The language was first introduced in a paper by Vadim Zaytsev [30]. In this paper the language is introduced and also the reasons why it has been created. In short, as discussed in the beginning of this section, it is important that new tools are still being developed for legacy languages. Since it is a lot of work to write a parser by hand, there are nowadays frameworks that can generate a lexer and parser for you, like ANTLR. However, since they are quite recent, they mainly focus on languages that are currently popular. So the BabyCobol language is created as a test or benchmark to see if a given framework is able to fully support legacy languages. More specifically BabyCobol is inspired by AppBuilder [7], CLIST [11], COBOL [13], FORTRAN [1], PL/I [14], REXX [4] and RPG [12]. An example of the BabyCobol language can be seen in Listing 2.

In the sample program there is a person defined with a first and last name. Also, an agent is like a person and inherits its structure. Once the program starts, the first and last names of the agent are collected and printed on the screen.

2 EXISTING SOLUTIONS & RELATED WORK

The idea of parsing goes back as far as 1951, when Stephen Kleene introduced the idea of regular expressions [15]. Later, the first complete compiler was introduced by the FORTRAN team in the year 1957 [1]. Refactoring tools that make use of new or existing parsers and compilers are also nothing new, as Méndez and Mariano [16] in 2011 indicate. Manually refactoring software programs that are, for example, written in FORTRAN is a big challenge and often not

```
000001 IDENTIFICATION DIVISION.
000002     AUTHOR. TOM MEULENKAMP.
000003     PROGRAM-ID. AGENT.
000004 DATA DIVISION.
000005     01 WORKING-STORAGE-AREA.
000006       02 PERSON.
000007         03 FIRST-NAME PICTURE IS 25(X).
000008         03 LAST-NAME PICTURE IS 25(X).
000009       02 AGENT LIKE PERSON.
000010 PROCEDURE DIVISION.
000011* Asks for the name of an agent and prints their
000012* full name at the end.
000013 MAIN.
000014     DISPLAY "Agent first name: ".
000015     ACCEPT FIRST-NAME OF AGENT.
000016     DISPLAY "Agent first name: " FIRST-NAME OF
000017-AGENT.
000018     DISPLAY "Agent last name: ".
000019     ACCEPT LAST-NAME OF AGENT.
000020     DISPLAY "Agent full name: " FIRST-NAME OF
000021-AGENT " " LAST-NAME OF AGENT.
000022 STOP
```

Listing 2. BabyCobol sample program

desired. In their paper, they discuss the creation of an automated refactoring tool, specifically designed for the Global Climate Model. Another example is an ANTLR-based COBOL parser implemented by Wolffgang [28]. In his implementation, he has decided to first make use of a preprocessor to normalise the file before actually parsing it. This preprocessor, for example, checks for indentation and normalises comments. Afterwards, the actual parser takes over.

More specifically to ANTLR there are already a lot of readily available grammars out there that have been written for ANTLR [8]. However, the grammars that can be found in the repository are not always accurate or miss certain features. In addition to grammars that can be found in the wild on the internet, there are also more private grammars available that have specifically been written for BabyCobol. During the master course "Software Evolution" at the University of Twente [32], students have to implement a complete compiler for the BabyCobol language. Some of the projects make use of ANTLR. However, after studying the grammars that have been written, complicated parsing problems have been skipped by all groups, like non-reserved keywords and white space ignorance. Therefore, there is room for improvement.

3 NEW ARCHITECTURE

A parser generally consists of multiple phases, where each phase has its own specific goal. The parser that has been implemented for BabyCobol consists of two phases that are dedicated to parsing, but an additional phase has been added that checks for the proper use of sufficient qualification. A visual representation of the steps taken can be found in Figure 3.

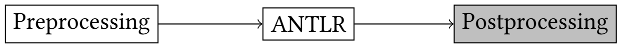


Fig. 3. Steps taken in the parser

3.1 Preprocessor

Before the actual parser can be run over the code that is given as an input, it is required to give a more consistent structure to the code. The reason why the code does not have this structure by default is due to the fact that it has been made for punch cards, which only have a limited number of columns to use [5]. The preprocessor especially takes care of the following things:

(1) Position-based syntax and semantics checks.
(2) Line indicator checks
(3) Line continuations and comments
(4) Copy statements

As discussed in "Software Language Engineers' Worst Nightmare" [30], BabyCobol makes use of position-based syntax and semantics. This feature is inspired by COBOL, and makes use of the same rules. The preprocessor splits each line into distinct sections: sequence area, line indicator, area A, area B, and ignored characters. This can also be seen in Figure 4, where the numbers above the arrows indicate the column numbers that belong to each section. Only the content area, also known as area A and area B combined, will be handed to the parser. Instead of using ANTLR for this, it was chosen to work with plain Java and regular expressions. Since ANTLR works with tokens and not necessarily with character indexes, it can be quite tricky to know for sure where a token is taken. Whereas regular expressions in Java allow for given positions to be defined. This preprocessor is greatly inspired by the one that Wolffgang [28] has written for his implementation of the COBOL parser.

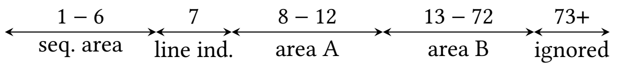


Fig. 4. Sections within a single line

BabyCobol only allows for three types of line indicators: ␣, -, *. Which respectively indicate a line of executable code; a continued line; and a line that has been commented. Any other character is invalid and raises a parsing error.

To let the parser focus purely on the text itself, all comments are ignored, and all line continuations are merged into a single line. The final lines are stored in special `Line` objects, that preserve the original line numbers, in order to give proper error messages later on in the parsing process.

BabyCobol supports copy statements. These statements allow us to insert another BabyCobol code file into the current file. Additionally these statements have the ability to replace strings in the file that is being imported. To once again let the parser focus on just parsing, all the copy statements are immediately executed in the preprocessor. To be able to identify copy statements, a small grammar was written that performs island parsing on each line. When a copy statement is executed, the line is split into three distinct sections: the code before the statement; the statement itself and the code after the statement. Each section receives its own line again. Except when a line is empty, then it is ignored. The copy statement then imports the file and replaces all occurrences indicated in the copy statement.

```
000006 ...
000007 PROCEDURE DIVISION.
000008 main.
000009     IF A = "Z" THEN COPY toCopy.bc REPLACING
000010-===A.=== BY ===B=== ELSE DISPLAY A B END.
000011 ...

      * Contents of toCopy.bc
000001     DISPLAY A.
```

Listing 3. Code file with a copy statement

In Listing 3 one can find a code section that will only execute the contents of the file `toCopy.bc` if the field `A` is equal to the string `"Z"`. It also contains the contents of the file to be copied into the code. In Listing 4 the result is found after the copy statement has been processed.

Note it was later found out that the copy statement refers to a program name, which means that a file extension is not needed. This is also discussed upon in Section 5.1.2.

```
000006 ...
000007 PROCEDURE DIVISION.
000008 main.
000009     IF A = "Z" THEN
000010     DISPLAY B
000011     ELSE DISPLAY A B END.
000012 ...
```

Listing 4. The result after processing the statement

After all the copy statements have been processed, the individual lines are aggregated and cleaned. Here, the latter means that all the sequence numbers are removed and all characters are ignored after the 72nd column. Afterwards they are passed on to the parser itself.

3.2 ANTLR

The ANTLR grammar is inspired by the grammar provided in the original paper about BabyCobol [30] and updated based on the most recent documentation as found in the language documentation [31]. However, not all details were provided, and the grammar hints that have been given by the sources are not always in the most optimal format, in order to work with ANTLR. This section will discuss the most important features included in the parser and how they have been implemented.

3.2.1 Non-reserved keywords. The most important feature that has been implemented in the ANTLR grammar is the ability to use keywords as identifiers, procedure, and paragraph names. The problem with the implementation of this feature is in how ANTLR handles ambiguity. ANTLR always returns a single parse tree. For most languages, this is a great feature and allows the programmer to focus on other more important matters. Unfortunately, this makes the ability to implement this specific feature quite difficult.

If there is any form of ambiguity when ANTLR tries to parse a given input, then the ambiguity is automatically resolved by choosing the first option that the parser can find. Which is always the first grammar rule or lexical token that is defined in the grammar specification. Unfortunately, BabyCobol does not work with this structure, since the language does not disambiguate based on natural precedence in the order of a given grammar, but based on the case in which a token is typed. If there is an ambiguity present then BabyCobol assumes a keyword if it is written in full upper case, whereas otherwise it sees it as an identifier.

The problem with this logic in combination with ANTLR is that it can only be evaluated when both options have been tried. However, once this has happened, ANTLR has already made the choice to go with the first fitting grammar rule or token, and the parser cannot be redirected on the basis of the case. One feature that allows one to partially resolve this is the use of predicates [18] [17, p. 196]. These allow one to turn of given rules *before* the parser attempts to evaluate them.

```
000004 ...
000005 DATA DIVISION.
000006     01 If PICTURE IS 9.
000007     01 ELSe PICTURE IS 9.
000008     01 then PICTURE IS 9.
000009 PROCEDURE DIVISION.
000010 MOVE.
000011     IF if = then THEN DISPLAY then ELSE
000012-display iF END.
000013 STOP
```

Listing 5. Keywords as identifiers, where bold text is a keyword and underlined text is an identifier

The final solution is a limited implementation of this feature with the help of ANTLR predicates. Every field that is declared is placed into a set. If a keyword is also used as an identifier, then it is automatically recognised as an identifier when it is not completely uppercase. If, however, it is used in full uppercase, then it is a keyword. If the keyword is not declared as identifier, then this distinction does not apply. An example of how this affects any written code can be found in Listing 5. This implementation has a direct

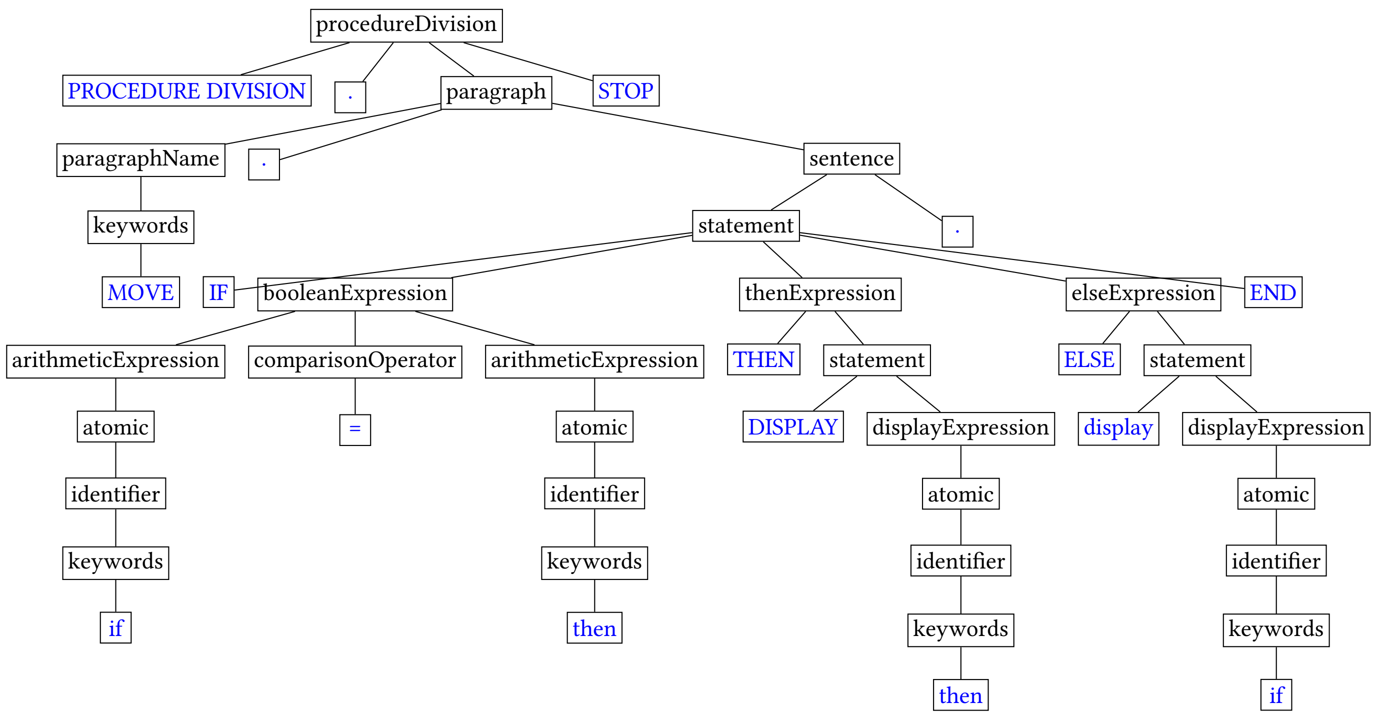


Fig. 5. The parse tree for the procedure division from Listing 5 as generated by ANTLR

influence on the case-insensitivity feature that is discussed in the next section.

This implementation only applies to references to fields and not to procedure and paragraph names. These can unambiguously be identified as just a keyword used as a name since they end directly with a period. This is also the reason that one can use the keyword `MOVE` on line 10 in Listing 5. Unfortunately, there is one edge case that is not covered properly when one uses keywords for paragraph and procedure names. This edge case is elaborated on more in Section 5.

3.2.2 Case-insensitivity. The next feature supported by the parser is case-insensitivity. Technically speaking, BabyCobol is fully case-insensitive. However, as has been hinted on in the previous section, there are cases where the parse has to be altered based on the case. This used to be an issue for older versions of ANTLR, since case-insensitivity was never properly supported right out of-the-box. An option used to be to convert the incoming stream to all lower or uppercase letters. Another grammar option is to define a fragment, which is a part of a token, for each character and let it match with both upper and lowercase versions. But, with the release of version 4.10 [19] it is finally possible to declare an entire grammar case-insensitive. The beauty of this is that the original case remains preserved, which cannot be said if one would have to modify the input stream, as has been suggested before.

3.2.3 Ignored sections. One of the first issues that have been encountered when working out the exact grammar with ANTLR was the parsing of the identification division. The parser generally does not really care about what is going on in the identification division. It is only there for humans, and it will not be used whatsoever in any application. However, it should be able to contain all characters when defined, where the only special character is the period (`.`). Since ANTLR requires that every piece of text be part of some token, there is a token named `IGNORED` that accepts any single character. This allows any value to be placed in the body of the identification division. The reason why this issue differs with for example a string is that a string has a distinct begin and end marker ("). Whereas these ignored sections do not have this.

3.2.4 White space ignorance. The last interesting feature to look at for a parser is the idea of white space ignorance. Where most languages nowadays use white space to split the tokens in the lexing phase of the parser, BabyCobol ignores all white space. This means that the statements shown in Listing 6 can possibly express identical behaviour.

```
000008    DISPLAY ABC.
000009    DISPLAY AB C.
000010    DISPLAY A B C.
```

Listing 6. A demonstration of the repercussions of white space insignificance

The problematic aspect is that these decisions must be made in some logical sense. It would, for example, make more sense to split it into the second suggestion if the field `ABC` is not defined, but the fields AB and C are. The issue is only that, in order to do this, one needs to make sure that the lexer and parser work together and are not two distinct processes. Unfortunately, this is not how ANTLR handles things. There is a clear distinction between the tokenizer and the parse tree generation. Hence, the option of providing white space ignorance in any form within a keyword or identifier is impossible with this framework. It is of course possible to instead of using a single space between two tokens to use multiple spaces, just like one could do in a language like Java.

3.3 Postprocessor

The entire parser is now finished, since ANTLR will output its own version of an AST, of which an example is shown in Figure 5. For clarity, the terminals have been printed in blue. However, we have decided to use one application of the parse tree, which is to check for sufficient qualification. BabyCobol's sufficient qualification feature allows the programmer to not have to write down the entire path to a given variable when needed. An example use where sufficient qualification is important is shown in Listing 7.

```
000007 ...
000008 DATA DIVISION.
000009     01 C1.
000010       02 C2.
000011         03 A PICTURE IS 99.
000012         03 C3.
000013           04 B PICTURE IS 99.
000014       02 X PICTURE IS XX.
000015       02 B PICTURE IS 9X.
000016     01 Y LIKE B.
000017 PROCEDURE DIVISION.
000018     ACCEPT B OF C2.
000019     ACCEPT B OF C1.
000020     DISPLAY X.
000021     DISPLAY Y A.
000022 ...
```

Listing 7. Sufficient qualification example

The implementation of sufficient qualification goes twice over the generated AST. In the first iteration, a tree visitor is used, with which you can explicitly tell ANTLR which nodes to visit and which to skip. In this first run the data division is visited and the data structure that is represented in this division is generated. We chose to make use of a tree structure where nodes have children and properties, in which each node with children should be a BabyCobol container and not a field. The resulting data structure that will be used for the sufficient qualification check, can be seen in Figure 6, which is based on Listing 7. As can be seen, each node level corresponds to the level in the tree. The root node makes sure that also the top level fields (indicated in this case with 01) have a single shared parent.

In the second iteration over the tree, both the data division and procedure division are checked. Instead of a tree visitor, a tree

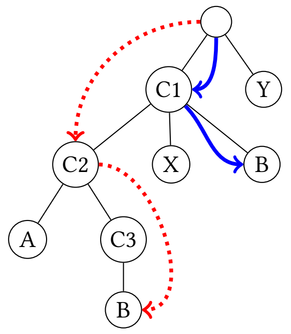


Fig. 6. Data structure after sufficient qualification

listener was used. The advantage of this is that there is no longer a need to explicitly indicate which node to visit, since all nodes are visited. This simplifies the logic a bit more, since identifiers are used on a lot of places in the procedure division. However, the disadvantage is that the entire tree has to be visited, which takes more time. Although all fields have been declared, fields of type LIKE must actually refer to a sufficiently qualified field. In Figure 6 one can see the path that the checker takes when it checks sufficient qualification for the ACCEPT statement on line 18 in Listing 7. The red dotted line first travels to C2 after which it jumps to its child B. The statement B OF C1 points to the child of C1 and not to the child of C2 although this is possible. The sufficient qualification checker first checks if any child contains the field name; if that is true, then direct children are checked; hence, the blue line corresponds to the instruction B OF C1. If no direct child contains the value, then the children are visited one by one till the field value is found. If multiple options remain, an error is generated; this will be the case for the LIKE statement on line 16, where B is not sufficiently qualified.

4 EVALUATION

The resulting parser has been tested throughout development with the help of unit testing, but also with visual checks such as observing the parse tree. In order to quickly and easily test many cases and programs after each modification, we have built a pretty printer that allows for code comparison. This pretty printer is especially important because it can show when the parser assumed that something was a keyword and when it was not. All keywords are printed in full uppercase, whereas all identifiers are printed in full lowercase. Next to the comparison of the output of the pretty printer with an expected file, all error messages are checked to see if they are generated on the correct locations, with the right file names and line numbers. An example output of the pretty printer based on the input code for Listing 5 can be found in Listing 8.

4.1 Performance

Since we have to make use of predicates in order to determine if a keyword is actually used as a keyword, or just an identifier; the parser has to constantly execute these check functions. The performance impact is quite well present when one compares it to a grammar without support for non-reserved keywords.

```
␣␣␣␣␣␣␣...
␣␣␣␣␣␣␣DATA␣DIVISION.
␣␣␣␣␣␣␣␣␣␣␣01␣if␣PICTURE␣IS␣9.
␣␣␣␣␣␣␣␣␣␣␣01␣else␣PICTURE␣IS␣9.
␣␣␣␣␣␣␣␣␣␣␣01␣then␣PICTURE␣IS␣9.
␣␣␣␣␣␣␣PROCEDURE␣DIVISION.
␣␣␣␣␣␣␣move.
␣␣␣␣␣␣␣␣␣␣␣IF␣if␣=␣then␣THEN␣DISPLAY␣then␣ELSE
␣␣␣␣␣␣-␣DISPLAY␣if␣END.
␣␣␣␣␣␣␣STOP
␣␣␣␣
```

Listing 8. Pretty print output for Listing 5

In Figure 7 one can see the percentage of time saved when using reserved keywords, instead of non-reserved keywords. The x axis has slots of 10% each. The y axis indicates the amount of programs that fall within a range of the given percentage. On average, the parser with no support for non-reserved keywords runs 45% faster. However, as it can already be seen, these performance increases are not stable and can vary a lot. There is even an outlier present where the grammar with non-reserved keywords was faster.

This test was executed with a grammar identical to the original BabyCobol grammar. The only difference is that the predicates have been disabled and that keywords have been removed from the identifier, paragraph, and procedure rules. For the sake of testing, only the preprocessor and parsing steps are used. Sufficient qualification checks have been disabled for both parsers (with and without keywords as identifiers). The time spent performing each operation has been calculated with `System.nanoTime()`. However, it must be noted that the time spent parsing is very brief because of the size of the sample programs, which causes a lot of variance in the time differences between runs. Still, Figure 7 should still be a good indication on the average distribution. The programs that have been used are the programs that run without errors or warnings from the sample programs that are used with unit tests. Additionally sample programs have been taken from other authors like groups that have worked on the BabyCobol project in the master course [32].

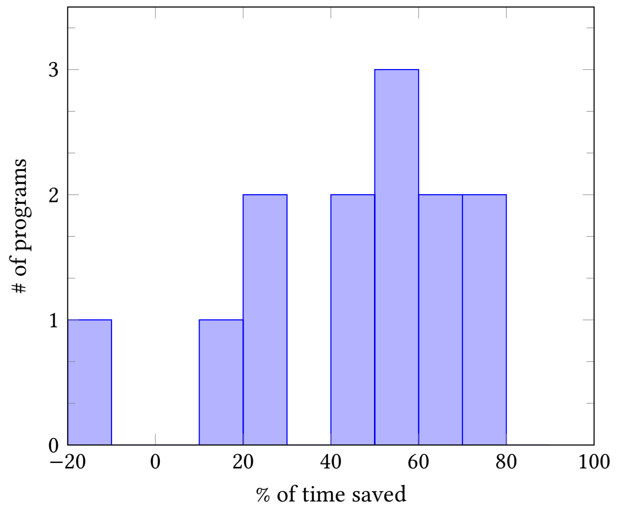


Fig. 7. Histogram of percentage of time saved when running without non-reserved keywords support

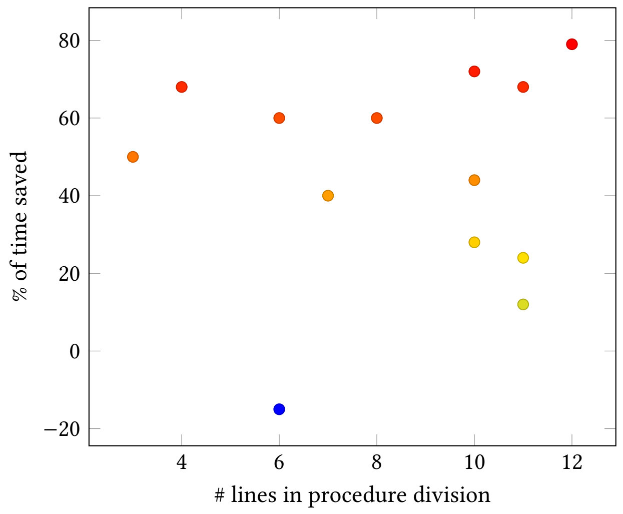


Fig. 8. The relation between the time saved while parsing and the amount of lines in the procedure division

During the testing it was also pointed out that the difference in time does have no correlation with the amount of lines in a program, especially the amount of lines in the procedure division since this is where the non-reserved keywords feature plays a big role. This relation can also be seen in Figure 8, in here it seems that the growing amount of lines causes both high and low extreme values (ignoring the outlier marked in blue). To truly understand how these times differ, one would have to get a better understanding of the inner workings of the underlying ALL(*) parsing algorithm [20], which ANTLR is using.

5 FUTURE WORK

5.1 Preprocessor

5.1.1 Implementation in ANTLR. In the current implementation, the preprocessor is fully implemented in Java. However, it should be possible to have a preprocessor with the same functionalities available in the form of an ANTLR grammar and a tree visitor or listener. This method may not be faster than the current implementation, but it will be more uniform with the parser itself.

However, possible problems during the implementation of such a grammar could be that one can end up with some form of scannerless parsing [24], which simply creates a token for each character and tries to match that with a grammar rule. Such an implementation will have big a performance impact since each possible grammar rule is tried on each attempt, before all the tokens are consumed. It would then make more sense to not use ANTLR at all and to use a SGLR-style parser [24], which stands for Scannerless Generalized

LR parser. This type of parser does not have a separate lexical phase. Another approach would be to define the sections as tokens, but one would have to account for the fact that sections do not necessarily have to be fully populated (although area B spans up to column 72, that does not mean that every line goes this far). Last but not least, sections would most likely have to be checked based on the current character positions, which can be achieved with the use of predicates. However, this makes the grammar dependent on the programming language.

5.1.2 Copy handling. Another improvement that could be made to the preprocessor is the check for self-calls to copy statements. If one would currently instruct to copy the file itself, the parsers would gladly do this. However, this would cause an infinite loop, since there is support for copy statements inside of files that are being copied. A more complex problem would then also be an indirect self-call where two or more files would import each other and cause a loop. Another smaller issue with the copy directive is that currently one has to give in a filename, including the `.bc` extension. However, it was later found that the file extension should not be needed; this should still be implemented.

5.2 Keywords for paragraph names

As discussed in Section 3.2.1 there is an edge case where keywords are not properly recognised as paragraph names when they are used as them. The issue that allows for this edge case is the way in which the paragraph names are uniquely identified. They always immediately end with a period and do not contain any spaces. This is different from a sentence that contains a statement since statements have (almost) always more than one token to be consumed before the period closes of the sentence. The only case where this is not true is for the keyword `STOP`. The most ideal way to fix this issue and to immediately also be more strict on the use of the A and B areas is to make the parser aware of these areas. Such an implementation would need to modify both the preprocessor and the parser itself. One technique that was around in the 1960s and 1970s is called stropping [9, p. 82]. This technique explicitly marks characters with some character sequence. This can easily be introduced in the preprocessor, which will indicate whether the line was started in area A or area B. Next, the ANTLR grammar could be modified in such a way to consume these markers and jump to the correct rules based on them.

5.3 Sufficient qualification

The current implementation for the sufficient qualification check runs twice over the parse tree, where it first collects and generates the data structure needed to check the references later on. And on the second run identifiers are checked against the data structure. If we were to implement this again, then it would be suggested to merge these two steps into a single step. Here a single parse tree listener should be able to do the job. This will increase the efficiency of the overall program.

6 CONCLUSIONS

To conclude this paper and to answer the research question: "Can a modern parsing tool, like ANTLR, contribute to legacy code support and enhancement?" Yes, ANTLR is able to contribute to legacy code support and enhancement, although in a limited matter. Certain features of BabyCobol are not able to be fully implemented or are simply impossible. An example of a partial implementation is the ability to use keywords as identifiers, whereas white space ignorance is simply impossible to implement due to the clear distinction between the scanner and parser phases.

To answer the first sub-research question, it is not possible to implement the BabyCobol language fully within the ANTLR framework. As suggested in Section 5.1, it might still be possible to implement a preprocessing phase with a separate ANTLR grammar. However, it is not possible to support complete white space ignorance.

Can one implement BabyCobol without any complex structures? For the most part, yes. However, in order to partially support non-reserved keywords, predicates have to be used. These add more complexity to the parser and make the grammar language dependent. Furthermore, to answer the last sub-research question, predicates have a significant performance impact, as observed in Section 4.

As a wise man once said, "information wants to be free" [26]. Hence one should also have access to the source code, which can be found at https://github.com/supertom01/BabyANTLR.

ACKNOWLEDGMENTS

I would like to thank my supervisor, Vadim Zaytsev. Especially for the nice talks about random subjects and the ability to occasionally complain about legacy languages.

REFERENCES

[1] John W. Backus, R. J. Beeber, S. Best, R. Goldberg, Lois M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, Peter B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. 1957. The FORTRAN Automatic Coding System. In *Papers Presented at the February 26-28, 1957, Western Joint Computer Conference: Techniques for Reliability* (Los Angeles, California) *(IRE-AIEE-ACM '57 (Western))*. Association for Computing Machinery, New York, NY, USA, 188–198. https://doi.org/10.1145/1455567.1455599
[2] Keith D. Cooper and Linda Torczon. 2011. *Engineering a compiler.* Morgan Kaufmann. https://doi.org/10.1016/C2009-0-27982-7
[3] James R. Cordy. 2004. TXL - A Language for Programming Language Tools and Applications. *Electronic Notes in Theoretical Computer Science* 110 (2004), 3–31. https://doi.org/10.1016/j.entcs.2004.11.006 Proceedings of the Fourth Workshop on Language Descriptions, Tools, and Applications (LDTA 2004).
[4] Michael F. Cowlishaw. 1990. *The Rexx language: A practical approach to programming.* Prentice-Hall.
[5] Frank da Cruz. 2021. IBM Punch Cards. http://www.columbia.edu/cu/computinghistory/cards.html
[6] Thomas R. Dean, James R. Cordy, Andrew J. Malton, and Kevin A. Schneider. 2003. Agile Parsing in TXL. *Automated Software Engineering* 10, 4 (Oct. 2003), 311–336. https://doi.org/10.1023/A:1025801405075
[7] Magic Software Enterprises. 1995. AppBuilder. http://www.appbuilder.com
[8] Tom Everett and Ivan Kochurkin. 2022. ANTLR 4 - Grammars. https://github.com/antlr/grammars-v4
[9] Wilfred J. Hansen and Hendrik Boom. 1977. The Report on the Standard Hardware Representation for ALGOL 68. *SIGPLAN Not.* 12, 5 (may 1977), 80–87. https://doi.org/10.1145/954652.1781178
[10] Paul Hudson. 2022. How PHP is written – Hacking with PHP - Practical PHP. http://www.hackingwithphp.com/2/6/0/how-php-is-written [Online; accessed 26. Jun. 2022].
[11] IBM. 1988. *z/OS TSO/E CLISTs Version 2 Release 1.* IBM.
[12] IBM. 1994. *Programming IBM Rational Development Studio for i ILE RPG Programmer's Guide.* IBM.
[13] IBM. 2009. *Enterprise COBOL for z/OS V4.2 Language Reference.* IBM. 700 pages.
[14] IBM. 2017. *Enterprise PI/L for z/OS Language Reference.* IBM. 756 pages.
[15] Stephen C. Kleene. 2016. *Representation of Events in Nerve Nets and Finite Automata.* Princeton University Press, 3–42. https://doi.org/10.1515/9781400882618-002

[16] Mariano Méndez. 2011. Fortran refactoring for legacy systems. National University of La Plata. https://doi.org/10.35537/10915/4201

[17] Terence Parr. 2013. *The Definitive ANTLR 4 reference* (2nd ed.). The Pragmatic Programmers.

[18] Terence Parr. 2021. ANTLR 4 Documentation. https://github.com/antlr/antlr4/blob/master/doc/index.md

[19] Terence Parr. 2022. Release 4.10 major feature, code clean up, and Bug Fix release. https://github.com/antlr/antlr4/releases/tag/4.10

[20] Terence Parr, Sam Harwell, and Kathleen Fisher. 2014. Adaptive LL(*) Parsing: The Power of Dynamic Analysis. *SIGPLAN Not.* 49, 10 (oct 2014), 579–598. https://doi.org/10.1145/2714064.2660202

[21] Terence J. Parr and Russell W. Quong. 1995. ANTLR: A predicated-LL(k) parser generator. *Software: Practice and Experience* 25, 7 (1995), 789–810. https://doi.org/10.1002/spe.4380250705

[22] Andrey A. Terekhov and Chris Verhoef. 2000. The Realities of Language Conversions. *IEEE Softw.* 17, 6 (2000), 111–124. https://doi.org/10.1109/52.895180

[23] Gabriele Tomassetti. 2022. The Antlr Mega Tutorial. https://tomassetti.me/antlr-mega-tutorial/

[24] Mark G. J. van den Brand, Jeroen Scheerder, Jurgen J. Vinju, and Eelco Visser. 2002. Disambiguation Filters for Scannerless Generalized LR Parsers. In *Compiler Construction*, R. Nigel Horspool (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 143–158.

[25] Arie van Deursen and Tobias Kuipers. 1999. Building documentation generators. In *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360).* IEEE, Oxford, UK, 40–49. https://doi.org/10.1109/ICSM.1999.792497

[26] R. Polk Wagner. 2003. Information Wants to Be Free: Intellectual Property and the Mythologies of Control. *Columbia Law Review* 103, 4 (2003), 995–1034. http://www.jstor.org/stable/1123783

[27] Sandip Walsinge. 2020. COBOL programming. https://zosmainframe.blogspot.com/2020/05/cobol-programming.html

[28] Ulrich Wolffgang. 2018. ProLeap COBOL parser. https://github.com/uwol/proleap-cobol-parser

[29] Vadim Zaytsev. 2014. Formal Foundations for Semi-parsing. In *Proceedings of the Software Evolution Week (IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering), Early Research Achievements Track (CSMR-WCRE 2014 ERA)*, Serge Demeyer, Dave Binkley, and Filippo Ricca (Eds.). IEEE, 313–317. https://doi.org/10.1109/CSMR-WCRE.2014.6747184

[30] Vadim Zaytsev. 2020. Software Language Engineers' Worst Nightmare. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering* (Virtual, USA) *(SLE 2020)*. Association for Computing Machinery, New York, NY, USA, 72–85. https://doi.org/10.1145/3426425.3426933

[31] Vadim Zaytsev. 2021. BabyCobol: The language reference. https://slebok.github.io/baby/

[32] Vadim Zaytsev. 2021. Software Evolution. https://canvas.utwente.nl/courses/7873

[33] Vadim Zaytsev and Anya H. Bagge. 2014. Parsing in a Broad Sense. In *Model-Driven Engineering Languages and Systems*, Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran (Eds.). Springer International Publishing, Cham, 50–67.