Replication and Analysis of the Berry-Sethi Parser for Ambiguous Regular Expressions

Ivo Broekhof
University of Twente
PO Box 217, 7500 AE Enschede
the Netherlands

i.broekhof@student.utwente.nl

ABSTRACT

Since regular languages, often described with Regular Expressions, are much faster to parse than context-free languages, it can prove beneficial to define regular (parts of) languages using Regular Expressions rather than the more powerful yet slower to parse Context-Free Grammars. However, ambiguity in regular expressions can make this task more difficult. We have successfully replicated a deterministic parser generator for Ambiguous Regular Expressions, based on the Berry-Sethi algorithm, in Kotlin. This replication is written in a more simple manner, with the result that it is easier to read and develop from.

Keywords

Ambiguous Regular Expression, Berry-Sethi Parser, ARE, BS algorithm

1. INTRODUCTION

1.1 Background and Motivation

Regular languages are significantly faster to parse than context-free languages, and this makes it attractive to parse regular (parts of) languages with a parser built from Regular Expressions rather than with a parser built from a Context-Free Grammar.

In the beginning of last year, a paper was published on the parsing of Ambiguous Regular Expressions (AREs) [5]. In this article, the authors describe how they implemented a deterministic parser generator for these AREs using an extended version of the Berry-Sethi algorithm [2], adding the power to parse these AREs rather than just recognize them. The parser generators produce a set of Linearized Syntax Trees (LSTs), compacted into directed acyclic graphs (DAGs). The authors have also written some code to achieve this, in Java, and have included a previously existing RE parser in C++ as well as a set of REs to test the implementations on [4].

An attempt at replicating this parser using only the paper as a source can locate potential unclarities or inconsistencies regarding its implementation. By keeping the replication simple, well-documented and concise, it can be easier to optimize this parser for a specific purpose or adapt the parser to build further on the concepts that this parser utilizes.

1.2 Preliminaries

Regular Expressions (REs) are a way to specify text patterns, and are built up of the empty word (ε) and symbols, with the operations of concatenation, union (|), zero or more (*) and one or more (+) allowing one to define any regular language with these elements. From a RE, it is possible to derive a Deterministic Finite-State Automaton (DFA) that recognizes this language.

REs can also be represented through an Abstract Syntax Tree (AST). In this representation parentheses, symbols and ε are shown as leaves and the operators mentioned before are shown as nodes. The opening and closing parentheses are the left, respectively right, sibling of the enclosed expression. An AST with all leaves marked with a number is known as a Marked Abstract Syntax Tree (MAST), and its associated RE is known as a Marked Regular Expression [5, Definition 1]. In figure 1, an example of an AST and a MAST is shown.

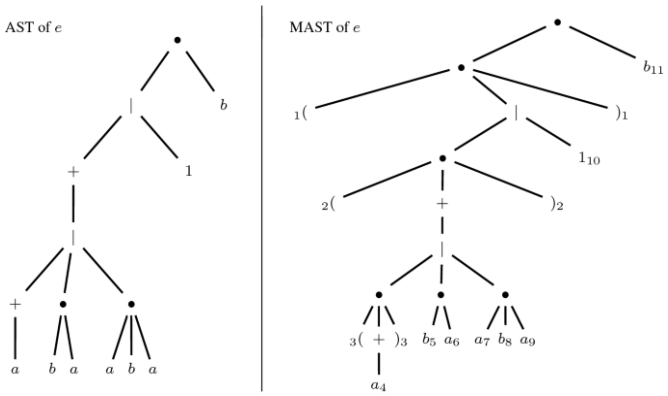

Figure 1. Example of an AST and a MAST [5]

Parsing a word with a MRE results in a Marked Syntax Tree, showing the structure of the word. Using the markings of the symbols, it is possible to trace back which part of the MRE, and thus the corresponding RE, matches a given part of the word. Concatenating the leaves of a MST from left to right results in a Linearized Syntax Tree (LST), which is the ultimate result of parsing the word.

Local languages are a strict subset of regular languages, and can be defined by its Initial set, its Digram set and its Final set, all of finite size. The Initial set is the set of all symbols that a word in this language can begin with, the Digram set is the set of all possible sequences of two symbols in a word in this language, and the Final set is the set of all symbols a word in this language can end with. In addition to this, the Follower set of a symbol is defined to contain all symbols that can come after this symbol.

A LST can be factorized into segments, consisting of zero or more marked metasymbols ((,), ε) and one marked symbol. An Acyclic Segment (AS) is a segment with all metasymbols marked distinctly, and can be treated like symbols themselves. This allows the Initial set of Acyclic Segments (Ini_{AS}) and Follower set of Acyclic Segments of a symbol a_h ($\text{Fol}_{\text{AS}}(a_h)$) to be defined similarly to before.

2. GOALS

2.1 Replication of the Algorithm

To help verify the process described in the paper and attempt to find potential flaws and/or unclarities in the method, we seek to replicate the process described in the paper without the guidance of the code written by the authors. This brought us to Research Question 1:

Is the description of the parser generator implementation clear and consistent, such that the code can be replicated, and does this replication have the same behaviour as the author's code?

2.2 Analysis of the Implementation

After the replication attempt, to properly compare the new implementation with the original, it is important to examine the original implementation and try to understand the design decisions taken by the authors. This poses Research Question 2:

How does the authors' implementation work, and for what purpose has it been defined?

2.3 Comparison of the Implementations

After the replication and the analysis of the original implementation, the two can be compared to see if they behave similarly. This poses Research Question 3:

Does the replication produce the same behaviour as the original code, or is there a difference? If there is a difference, what is it and how is it caused?

3. RELATED WORK

It has already been proven that the ambiguity, or lack thereof, in a RE can be decided [3]. If this were not the case, the parsing of AREs would also be undecidable.

The Berry-Sethi algorithm [2] has been derived in 1986 already, and forms the basis for the parsing of the REs in the similarly named BSP. If the paper is unclear about implementation details on this part, the original can provide help understanding the implementation.

The authors of the paper have previously published other papers [6] that are also referred to in this one, indicating that this research has been going on for longer. Similarly to the previously mentioned paper, insights could be gained from these previous papers in case the paper is unclear about concepts they mentioned before.

The authors have also generated a set of REs [4] to test the implementations on. Aside from this, the other sources referred to in the paper can prove useful.

4. REPLICATION OF THE ALGORITHM

4.1 Methodology

We replicated the algorithm using the Kotlin [9] programming language. Kotlin is a multi-platform language that can compile to the JVM platform, among others that is inter-operable with Java. However, Kotlin has been refined to be more concise and safer, for example with its features to easily deal with nullability. This should make it easier to implement the algorithm and will hopefully help write it quicker by requiring less debugging than with, for example, Java.

The intention is to replicate the system without looking at the code and without contact with the authors, so purely from the paper.

The expected answer to Research Question 1 is that it is possible by following the description in the paper. If this turns out to be more difficult, there is the option to look into the articles referred to by the authors, like the original Berry-Sethi algorithm as well as the authors' earlier publications. A repository has been created to house the replication [8].

4.2 Process

To start, the RE had to be parsed and the symbols had to be marked. Since there is no particular way in which it needs to be parsed, an ANTLR [1] grammar was used in combination with a ParseTreeVisitor implementation to traverse the parse tree and mark the symbols to make a MRE [5, Definition 1]. To make the MRE from a RE, the parentheses, empty words and the symbols had to be given a number marking, following an in-order traversal of the Abstract Syntax Tree. Using ANTLR, little work had to be done for parsing the RE itself, and a grammar is arguably simpler to read and adapt than custom code to parse the RE. Using the examples from the paper, the parsing of the RE and marking could be tested.

From here, the set of Initial Acyclic Segments (Ini_{AS}) [5, Definition 6] has to be constructed as well as the sets of Acyclic Segments that follow each symbol a_h ($\text{Fol}_{AS}(a_h)$). The process for constructing these sets is not explicitly described in the paper, but similar concepts exist elsewhere and the descriptions of the sets themselves together with the examples given in the article was enough to derive an algorithm. For the Ini_{AS}, a recursive algorithm can easily be defined using the properties of the nodes and leaves of the MAST, for instance the property that symbols and ε do not have children and therefore are the contents of their Ini_{AS}. A more difficult case to implement is the concatenation, since a child's Ini_{AS} is only included if the previous child is nullable, i.e. contains ε. For the Fol_{AS} of a symbol, the given symbol first has to be located in the MAST, and after that the set can be built up as the path through the MAST is taken in reverse. In the computation of these sets, an end-of-text symbol is appended to the MRE so that the Follower sets comprise both the Digrams set and the Final set of the MRE. These sets of the previous examples are also shown in the paper, making it easy to run a quick test of the replication with these. The Ini_{AS} and Fol_{AS} sets of the running example can be found below.

$$\check{e} = {}_1(\left[{}_2(\left[{}_3([a_4]^+)_3 \mid b_5\,a_6 \mid a_7\,b_8\,a_9\right]^+)_2 \mid 1_{10}\right])_1\,b_{11}$$

$$Ini_{AS}(e\,\dashv) = \{\, {}_1({}_2({}_3(a_4,\ {}_1({}_2(b_5,\ {}_1({}_2(a_7,\ {}_1(1_{10})_1\,b_{11}\,\}$$

marked symbol a_h	*set of followers of* a_h $-$ $Fol_{AS}(e\,\dashv, a_h)$
a_4	a_4 $)_3\,{}_3({}a_4$ $)_3\,b_5$ $)_3\,a_7$ $)_3\,)_2\,)_1\,b_{11}$
b_5	a_6
a_6	${}_3({}a_4$ b_5 a_7 $)_2\,)_1\,b_{11}$
a_7	b_8
b_8	a_9
a_9	${}_3({}a_4$ b_5 a_7 $)_2\,)_1\,b_{11}$
b_{11}	$\dashv$

Figure 2. Ini_{AS} and Fol_{AS} of the running example [5]

36th Twente Student Conference on IT, Febr. 4th, 2022, Enschede, The Netherlands.

Given Ini_{AS} and $\text{Fol}_{AS}(a_h)$ for each a_h, the modified Berry-Sethi algorithm [5, Algorithm 1] can be implemented from the pseudocode. This yields the Deterministic Finite-state Transducer (DFT) [5, Definition 7], from which a Directed Acyclic Graph (DAG) can be made and from which the LSTs can be derived. Since the pseudocode is more on the mathematical side, data structures in the implementation do not completely match those in the pseudocode but it would be possible to make them match perfectly if one were to follow it very closely. For example, in the pseudocode the contents of a state are denoted by the function I(q), but the reassignment combined with the requirement of the values after the construction of the DFT makes it more suitable to implement the contents of a state using a field. Also the DFT and LSTs for the running examples were given, allowing for another quick test. Below is a representation of the DFT for the example.

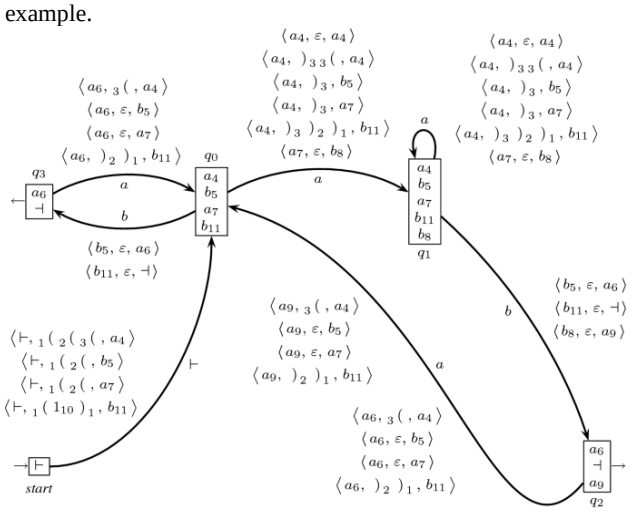

Figure 3. DFT of the running example [5]

4.3 Conclusion and Further Work

It is possible to replicate the parser generator described, following a successful attempt to do so. The source code can be found on the referenced Github repository [8]. Although some concepts were slightly more difficult to implement, the paper explains them clearly enough to conclude if the code gives the expected results.

With a little work cleaning up the current codebase and some additional documentation, it will be easy to extend. A possible extension of the implementation is the inclusion of more RE operators like option and bounded repetition, and the simplicity of the code should allow one to transcribe the code into another programming language more easily, optimizing the code for that environment as this implementation is not optimized for performance.

5. ANALYSIS AND COMPARISON OF THE IMPLEMENTATIONS

5.1 Methodology

By reading through the authors' implementation, some insights can be gained on the design choices and purposes of their implementation. It also helps to find differences in code style which can be compared against the replication.

After reading through the authors' implementation, it was attempted to link the replication with the other implementations and compare their results. As the codebase includes code to measure performance, this could be a useful metric even if this replication is not optimized for performance, but more important is the verification that the implementations produce the same results.

5.2 Analysis of the authors' implementation

The implementation by the authors has been written in one file, several thousands of lines long including comments on their deviations from the explained procedure. The implementation looks to be optimized for performance, and some other ARE parsing implementations have been included to compare against. In the same file, other adaptations have also been written to potentially increase performance even more. Names of variables and constants are often shortened strongly, making their purpose harder to derive from their name. However, after looking at the definition and corresponding documentation their purpose is made clear. Algorithms are often accompanied by long comment blocks where the approach taken is explained extensively.

5.3 Results and Further Work

Unfortunately, attempts to connect the replication to the authors' codebase have not been successful, but static analysis has allowed to draw some early conclusions.

The length of the authors' implementation is usually more than in this replication. For example, the length of their `buildBS()` method is just over 100 lines, while the similar `constructDFT()` is only 50 lines, while the amount of blank lines and comment lines is similar.

While the authors' implementation, written in Java, uses mainly imperative-style code, the replication uses more functional-style code to shorten repetitive statements and use less mutation in favour of mainly immutable variables. Because Kotlin facilitates functional expressions better than Java currently, a programmer can reap the benefits of more readable and more robust code without paying a heavy price of performance. This can also make it easier to rewrite this in a fully functional language like Haskell without having to convert everything from imperative style to functional style.

In the future, it would still be beneficial to attempt integrating this replication to the codebase again, and verify the results of the replication.

6. REFERENCES

[1] https://www.antlr.org. Retrieved January 2022.

[2] Berry, G., Sethi, R.: From regular expressions to deterministic automata. Theor. Comput. Sci. 48(1), 117–126 (1986).

[3] R. Book, S. Even, S. Greibach and G. Ott, "Ambiguity in Graphs and Expressions," in IEEE Transactions on Computers, vol. C-20, no. 2, pp. 149-153, Feb. 1971, doi: 10.1109/T-C.1971.223204.

[4] Borsotti, A., Breveglieri, L., Crespi Reghizzi, S., Morzenti, A.: A benchmark production tool for regular expressions. In: Hospodár, M., Jirásková, G. (eds.) CIAA, LNCS, vol. 11601, pp. 95–107. Springer (2019)

[5] Borsotti, A., Breveglieri, L., Crespi Reghizzi, S. *et al.* A deterministic parsing algorithm for ambiguous regular expressions. *Acta Informatica* **58,** 195–229 (2021). https://doi.org/10.1007/s00236-020-00366-7.

[6] Borsotti, A., Breveglieri, L., Crespi Reghizzi, S., Morzenti, A.: From ambiguous regular expressions to deterministic parsing automata. In: Drewes, F. (ed.) CIAA, LNCS, vol. 9223, pp. 35–48. Springer (2015)

[7] https://www.jetbrains.com/idea/features/#built-in-tools-and-integrations. Retrieved November 2021.

[8] https://github.com/brokhiv/bsp-kotlin

[9] https://kotlinlang.org. Retrieved November 2022