SLL(k) analysis in Byacc

Jorge Macedo
jorge.macedo@tecnico.ulisboa.pt
Instituto Superior Técnico, Lisbon, Portugal
October 2018

Abstract

The Byacc is a parser generator tool that generates an LALR(1) parser, that can accept a wide range of grammars. However, the LALR(1) parser requires too much memory and can be much slower when compared with an LL parser. The solution presented in this paper is about the adaptation of the Byacc in order to generate an SLL(k) parser instead of its LALR(1) parser, where the k will be the length of the lookahead to resolve the factorization problem that LL parsers has. The new parser will use an SLL¹(k) parsing table and resolve the eventually conflicts, that may appear, separately through a DFA. Thus, the rebuilt Byacc will generate a parser that requires less memory and a parser that can achieve a speed parsing time faster than the original parsing method, the LALR(1).

Keywords: Byacc, SLL(k), LALR(1), FIRSTk, FOLLOWk, DFA, SLL¹(k) Parsing table, Grammar

1. Introduction

Syntactic analysis of a grammar can be performed by several ways due the number of tools that exists and its techniques. There are some tools that generate a parser that can cover a large spectre of grammars but can not cover all of them. For instance, the ANTLR tool has as parsing algorithm the LL(1) technique; the BYACC tool was intended to be open source and compatible with YACC.

Regarding the number of grammars accepted, the LALR(1) parser is one of those who can accepts more grammars. However, an LALR parser is much more complex, less intuitive and hard to understand how it works inside. On the other hand, the LL parsers, or in specific the LL(1) parsers, are much more simple when compared with an LALR(1). The LL(1) parsers require less memory parsing a grammar, which allows them to be more compact, they are also faster, when parsing a grammar, and therefore they are more efficient.

Although the LL(1) parser seems to be better, it continues to be less used than an LALR(1), because is more difficult to find an equivalent grammar that is LL(1). Problems, such as, left-recursion on a grammar and short lookahead for parsing, prevent that an LL(1) parser can easily be able to accept any grammar.

A typical LALR(1) parser, produces a Deterministic Finite Automaton (DFA), that has a large amount of states. Then, during the parsing construction process, the parsing table generated, based on that DFA, has all of the shift’s, reduce’s and goto’s that could be possible during a parsing process, so the memory is an issue. Besides that, since the LALR(1) parser requires a large number of steps to process a simple input, like several shift or reduces, the processing parsing speed is also an issue.

However, although it seems to be easier to find an LALR(1) grammar than an LL(1), an LL(1) parser has a parsing table smaller and allows to have a parsing speed much higher. An LL(1) parser is quite easy to implement, to understand, to debug and also reports errors more accurately, unlike an LALR(1) parser.

The Byacc, generates an LALR(1) parser that, when compared with an LL(1), can accept a larger set of grammars due the constraints that the LL(1) has. So, the Byacc will always stay ahead of others LL(1) parser generators, since it can accepts more grammars, but it will always have same issues that will not make it the perfect tool.

So, it would be great if the Byacc stopped generating an LALR(1) parser and started generating an LL parser. This thought leads to the purpose of the changing of the Byacc, let it generates an LL parser. It can take advantage of the characteristics
of an LL parser, like speed and the size of the tables.

1.1. Objective
The objective of this change is to adapt the tool Byacc in order to be able to generate an Strong-LL(k) parser (SLL(k)), instead of an LALR(1), where the $k$ will be the number of the lookahead given to resolve the factorization problem, that a typical LL(1) parser has. Besides that, will be studied some algorithmic solutions to solve the problems in LL(1), like the left-recursion, which can be turn in right-recursions, or the lookahead issue, where the LL(k) approach can be the solution or even the LL(*), where the $k$ is no longer needed to be defined.

1.2. Goals
Is important produce the same type of files as before and let the Byacc with the same structure, so it still behave as before. Thus, the concrete objectives which will be focused during the solution development are the following:

1. Understand the Byacc and why is the best tool to implement the new parsing process.
2. Understand how Antlr was made and how it solved the issues that address the LL parsing method, in its LL(k) version and LL(*).
3. Understand the differences among the LL(1), LL(k), SLL(k) and LALR(1) parsing methods;
4. Include the SLL(k) parsing method in the original Byacc.
5. Try to improve the way that is generated the parser by Byacc in order to be faster and also requiring less space comparatively to the original parsing method.
6. Keep the Byacc input identical to accepts and produces the same type of files of the LALR(1) based users.
7. Be able to analyze an entire programming language, like C and have improvements of performance.

2. Background

2.1. Tools

2.1.1 Yacc
Yacc is a tool created by S. C. Johnson in the early 1970s with the objective to generate a syntactic analyzer that interprets a given grammar, that has to be described in a BackusNaur Form (BNF). This tool has its own language where the grammar should be defined in grammarExample.y file. The .y file is transformed in a C program called y.tab.c that contains the parser itself and the representation of the LALR(1) parser process written in C, with the parsing tables needed during the parsing process. The y.tab.c can be compiled, generating an executable that can be run along with a grammar case. Besides that, the Yacc has some features like the conflict resolution and the error recovery, that will allow it deal with the grammars that are not LALR(1).

During a LALR(1) parsing process some types of conflicts can appear, like a shift/reduce conflict or a reduce/reduce conflict and even with this conflicts the the Yacc is still capable of producing the parser. It does this by choosing one of the valid steps whenever it has a choice to make. A rule that describes what choice to make in a given situation is called a disambiguating rule. So, when a conflict happens the Yacc invokes two disambiguating rules by default:

1. In a shift/reduce conflict, the default is to do the shift.
2. In a reduce/reduce conflict, the default is to reduce by the earlier grammar rule.

So the Yacc prefers to choose a shift rather than a reduce. This is justified by the fact the parser LALR(1) is known as rightmost derivation, so it will try to shift as many tokens as possible and only at the end reduce them. The other rule is interpreted by the fact the rules that appears earlier in grammar are eventually the most important.

Yacc has a word error that is placed in productions where the user desires to have the recovery error behavior. Like $A \rightarrow \text{error } \alpha$, where the $A$ is the non-terminal that should have a recovery error behavior and the $\alpha$ is a string of grammar symbols, perhaps the empty string. The Yacc will generates a parser with the such specification, treating the error productions as ordinary productions.

2.1.2 Byacc
Byacc stands for Berkeley Yacc, which is a re-implementation of the Yacc (Yet another compiler-compiler). It is compatible and has the same structure as its predecessor, so the parser generated is also an LALR(1). The major difference between them is the fact that Byacc does not have copyrights, so any one can modify it, fix some bugs or even upgrade it, so it was maintained by the contributors, since the Byacc is in the public domain. Besides that, the Byacc introduced a new algorithm for the LALR construction sets more efficiently that decreased the parsing generation time[6]. Therefore, Byacc became the most popular version of Yacc.

Every data structure and algorithm used in Byacc are all either taken from documents available to the general public or are inventions of the author.
but always letting the tool identical to the original Yacc, in terms of type of files and procedures, avoiding dependencies to any particular compiler. The original author has no longer released any version after 1.9 version, once the last version was considered concluded and any other public version over the last is considered, by the author, an extension with features.

2.1.3 Bison
Bison was developed by Robert Corbett in 1985, the same author as the Byacc, a couple years earlier. This tool is also an LALR(1) parser generator but can also generate a GLR grammar. Originally, when Robert wrote the parser generator, the Bison had dependencies on gcc or c99 and was not compatible with Yacc, just later, Richard Stallman made it Yacc-compatible.

Bison is a free software and can be redistributed or even modified by any one, under the terms of the GNU General Public License as published by the Free Software Foundation. Therefore, thanks to the many contributors that help, the Bison has been grown more robust and evolved many other new additional features, like the options %pointer and %array in Flex or the UTF-8 character handler. Currently it is in 3.0.4 version and is maintained by Akim Demaille and Paul Eggert.

2.1.4 CUP
CUP stands for Construction of Useful Parsers and is an LALR(1) parser generator for Java that was developed by C. Scott Ananian, Frank Flannery, Dan Wang, Andrew W. Appel and Michael Petter around 1996. It has the syntax quite similar to the Yacc and the version 0.10 brought more changes that attempted to make it more like its predecessor, the Yacc.

One feature important in CUP is the error recovery, which is the same as the Yacc has, with the insertion of the token error.

2.1.5 Antlr
The Antlr, acronym for Another Tool for Language Recognition, is a parser generator based on LL method. It was developed by Professor Terence Parr, around 1989 and since then already was released four versions of the tool. Each version released consists of an improvement of the approach used that allows the parser be more stronger and capable against grammars that are not traditionally LL(1).

The Antlr has been evolving with its version, starting with an SLL(k) parser in its first version until its last version where the Antlr author came up with an Adaptive LL Star parser (ALL(*)), which can accept grammars with left-recursion, for instance.

2.1.6 LLama
The LLama is a parser generator based on LL(1) parsing method. It was developed by Allen I. Holub around 1990 and published on its book[9], along with the source code. It works with the LEX, in the lexical analysis and the major restriction that it has is with the grammars, which must be an LL(1) grammar. The input grammar file is very similar with the input grammar file of the Yacc tool, divided into tree sections, definitions, rules and code.

LLama was written in C along its output file, the LL(1) parser generated\(^1\), and provides a good example of a working compiler for a small programming language using a recursive-descent parsing technique as the LL(1).

2.2. Discussion
Most of the tools searched and studied here have something in common, the attempt to be more like Yacc as possible and also the parser generated, which is the LALR(1), unlike the Antlr tool. It seems that the Byacc take advantage in this dispute, base on what was explained before. It can be modified without any restriction and is a tool quite simple and solid, since there is no upgrades of it, let it as a tool very stable to use in a research project. Besides that, there are much more information and troubleshooting about it than others, since it is the most used parser generator by the users.

In relation to the Bison and CUP, the Bison despite being free software it has a policy to respect about redistributions and modifications. It also seems to be more complex and hard to understand than Byacc and it can not be said that it is stable tool, since is still being released. The Cup is basically a re-implementation of Yacc but wrote in Java along with the LALR(1) parser generated that is also in Java, which may have made it not so widely known and used by users. Regarding the Antlr, it already generates an LL parser, but it keeps releasing new versions with new features that can not help a research project, where is desired to have the better and latest version released to add more features.

2.3. Parsing Techniques
2.3.1 LL(1)
The LL(1) parsing algorithm [8] is the easiest parsing techniques to implement and also to understand. It is based on Top-Down parsing approach, which is a technique that aims to begin the parsing process with the initial and most generic rule of the grammar, in the processing stack, and then, based on the token that is seen by the parser, replace each

\(^1\)lout.c
non-terminal symbol by other rule, until there are no more non-terminal symbols and the final word matches with the input string.

The LL(1) parser method is a deterministic technique, so the choice of the correct production to use is done quickly and direct through a parsing table, that is computed in generation parsing time when the grammar is provided, which helps the process of parsing when the parser must do a derivation of a non-terminal on the stack.

The construction of the parsing table is based on the results of the FIRST1 set and FOLLOW1 set of the grammar. Assuming that the grammar to parser is the grammar below with the following set of terminal symbols \{a, b, c, d\}:

\[
S \rightarrow aAb^r_1 \mid bAb^r_2 \mid Ba^r_3 \\
A \rightarrow Bbr^4 \mid daBr^5 \mid \varepsilon^r_6 \\
B \rightarrow cAn^r_7
\]

**Figure 1:** Test case grammar1.

The resulting FIRST1 set, FOLLOW1 set and consequently parsing table obtained are:

- \(FIRST(S) = \{a, b, c\}\)
- \(FIRST(A) = \{c, d, \varepsilon\}\)
- \(FIRST(B) = \{c\}\)
- \(FOLLOW(S) = \{$\}\)
- \(FOLLOW(A) = \{b, a\}\)
- \(FOLLOW(B) = \{a, b\}\)

**Figure 2:** FIRST and FOLLOW sets of grammar in Figure 1

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>r1</td>
<td>r2</td>
<td>r3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>r6</td>
<td>r6</td>
<td>r4</td>
<td>r5</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td>r7</td>
<td></td>
</tr>
</tbody>
</table>

**Table 1:** Parse Table for the grammar in Figure 1

With the parse table generated, the process of parsing an input string by an LL(1) is straightforward. First the parser does not need to look anymore for the grammar, everything that is needed is in the parsing table. Then, for each token that appears at the input stream the parser sees in its parsing table what rule should apply into its stack.

As the process goes on, the parsing tree is construct, which represents the production chosen by the parser for the given input string. For the input adacab and the grammar of Figure 1, the parsing tree is the following.

![Parsing tree of the sentence adacab.](image)

**Figure 3:** Parsing tree of the sentence adacab.

**Limitations**

One of the major problems that an LL(1) parser has is related with the productions that begin with the same non-terminal of the left-hand side of the rule. For example, in the rule \(S \rightarrow S + \text{id} \mid \text{id} - \text{num}\), the FIRST1(S) set will be FIRST1(S + id) \(\cap\) FIRST1(id - num) which is \{FIRST1(S), id\}. So the FIRST1(S) depends on its own FIRST1, leading to a recursive dependence, which stops the computation of the sets and consequently the parser generation process.

Another problem that an LL(1) parser process faces is with grammars that need more than one symbol of lookahead to decide which rule should be chosen. The construction of the parsing table depends, as was seen, on the FIRST1 set of the non-terminal symbol, which are composed by all FIRST1 set of the productions of that non-terminal symbol and besides the fact the set has to be concluded, it also needs to be a set disjoint to each others sets that belongs to the non-terminal, in other words, the FIRST1 can not have repetitions of the same terminal symbols, or otherwise the parse table will be ambiguous.

The grammars that leads these kind of ambiguities are grammars that need more than one symbol of lookahead, therefore, are grammars that are not LL(1) and could be eventually an LL(2) grammar or upper than that. In other hand, there are some grammars that require a lookahead so big that turn unreasonable the utilization of a parser with an LL approach, which makes them a not LL grammars.

2.3.2 LL(k)

There are some grammars that require more than one symbol of lookahead to decide which is the right rule to apply. Like the following grammar:

\[
S \rightarrow aab \mid abb \mid cab
\]

**Figure 4:** LL(2) grammar
where the \( \text{FIRST}_1(S) \) will be \{a, a, c\}. This set would cause conflicts in the construction of the parsing table, but if the parsing process uses two symbols of lookahead, instead of one, during the FIRST set calculation phase, the \( \text{FIRST}_2(S) \) would be \{aa, ab, ca\}, which would behave much better during the parsing process of the input, since the parsing table would have only one production per entry.

However, if the production aac was added to the grammar, the LL(2) parser will have again the same troubles as the LL(1) had before, in this case an LL(3) would be required. Although, instead of applying an LL parser, which in its algorithm has a fixed size of lookahead symbols, would be much more smarter use an LL parser that could accept the size of the lookahead and computes the FIRST, the FOLLOW and the parsing table dynamically based on the given lookahead, at parser generation time. That kind of parser is called LL(k), where the \( k \) is the dynamically settled.

The differences between the LL(1) and the LL(k) parser begin by the way that is calculated the FIRST and the FOLLOW sets of the non-terminal symbol. The \( \text{FIRST}_k \) of a non-terminal symbol will be the \( \text{FIRST}_k \) of its productions and any prediction that could result of the productions. If the length of result prediction is smaller than the \( k \), the \( \text{FIRST}_k \) of that prediction will be just it, otherwise the \( \text{FIRST}_k \) will be the first \( k \) symbols of the prediction.

A grammar is considered an LL(k) grammar if the \( \text{FIRST}_k \) of the productions and its predictions of any non-terminal symbol \( A \) are pairwise disjoint, that is, for any prediction \( \text{A}_\omega$\, being \( A \) a non-terminal of the grammar with the productions \( \alpha_1, \alpha_2, \ldots, \alpha_n \), the sets \( \text{FIRST}_k(\alpha_1 \omega$\), \( \text{FIRST}_k(\alpha_2 \omega$\),...,\( \text{FIRST}_k(\alpha_n \omega$\) must be unique among them. The \( \text{FOLLOW}_k \) is defined the same way as the \( \text{FIRST}_1 \). For any non-terminal \( A \), the \( \text{FOLLOW}_k(A) \) is defined as the union of the sets \( \text{FIRST}_k(\omega$\), for any prediction \( \text{A}_\omega$\.

A grammar that is an LL(k) is also an LL(k+1), because if the grammar is already an LL(k), it means that the parsing table is deterministic with \( k \) symbols of lookahead, so if the number of lookahead is increased, the parse table would also be deterministic, since the parser was already able to decide which rule should be applied with help of \( k \) symbols of lookahead. The reverse is not true, a grammar the is LL(k+1) could not be LL(k), because the parse could just determine which rule should be applied with \( k + 1 \) symbols of lookahead, so with \( k \) symbols it would not able to do that.

For the Parsing Table, the procedure is the same as the one done in the LL(1) parsing table. Every grammar rule \( A \rightarrow \alpha \) is added to the \( (A,a) \) entry of the table, where \( a \) belongs to the \( \text{FIRST}_k(\text{A}_\omega) \) set. If the resulted parsing table contains at most one rule, the grammar is considered as a strong-LL(k) grammar. A strong-LL(k) grammar is a grammar that is parsed using a deterministic parsing table, that is, with at most one rule per entry, using \( \text{FIRST}_k(\text{A}_\omega) \) to do it. The difference between an LL(k) parser and a SLL(k) is the way the parser puts the production in each entry. An LL(k) only considers the \( \text{FIRST}_k(\alpha) \), where the \( \alpha \) is a production of the non-terminal symbol \( A \), whereas an SLL(k) considers the \( \text{FIRST}_k(\text{A}_\omega) \), where the \( \alpha \) is a production of the non-terminal symbol \( A \). The following grammar is an example of a grammar the is an LL(2) but is not an SLL(2):

\[
S \rightarrow aAaa | bAba
A \rightarrow a | \epsilon
\]

**Figure 5:** LL(2) grammar that is not an SLL(2).

### 2.3.3 LALR(1)

The LALR(1) parsing is a method based on bottom-up approach, which basically does the parsing tree from bottom (leafs) to the top (root). As the parser reads the input, it goes gathering and grouping the tokens, in its stack, according to the rules of the grammar, in order to replace them for the left-side non-terminal symbol in the rule applied, which is grouped with more symbols, according again by the grammar rules, until reach the initial symbol of the grammar.

The decision of applying a rule or which rule should be applied, where the set of symbols symbol are replaced by one non-terminal more complex, or if the parser should continue to grouping tokens, is made through the LALR(1) parsing table, which is a representation of the state machine, initially generated, that contains the action to be taken (shift or reduce) for a given state. To build this table, the parser must first construct the state machine with the augmented grammar and calculate the FIRST and the FOLLOW sets, which are calculated the same way as the LL(1) does.

### 2.3.4 Discussion

Both techniques have limitations and features that gives them advantage over the other. None of them is the perfect technique that can parse all grammars. In case of LL(k), as was seen, the parser does not behave well on grammar with left-recursion, since the parse is known as the leftmost derivation. Other issue is with the grammars that require a
lookahead higher then $k$, the LL(1) is a good parser technique, but only works with grammars that require, at most, one symbol to decide which is the right rule, otherwise the parsing table would contain some conflicts, like a first/first conflict or a follow/follow conflict.

LL($k$) parser has a short table parser with $N \ast (T^k + 1)$ size, where the $N$ is the number of non-terminal in the grammar, the $T$ is the number of terminals, the $k$ is the look-ahead used and the 1 is the $\$\$ symbol. So, this parser requires less space of memory, the performance is quite good, both in parser generation time and in parsing time, since the parser just needs to calculate the FIRST and FOLLOW to build the parsing table and then the selection of the rule is direct.

In relation with the LALR(1), the major disadvantage is the complexity of the algorithm. Is practically impossible to write, by hand, the full flow of the generation parser, unless for a small grammars. The parsing table is bigger with $S \ast (N + T + 1)$ size, where the $S$ is the number of the states, the $N$ is the number of non-terminal symbols, the $T$ is the number of terminal symbols in the grammar and the 1 is the $\$\$ symbol. For a small grammar, like the one in the Figure 1, the resulting parsing table has 144 entries, whereas the LL(1) parsing table has 15 entries, without forgetting the need to have the state machine to construct the LALR(1) parsing table, which is large, even for a small grammar.

Regarding the performance, the LALR(1) can be almost as fast as the LL($k$) on parsing a sentence, but it takes more steps to respond if the sentence belongs to the grammar or not, so for the same input string.

3. Architecture

The solution that is proposed comes with the idea to improve the way that an actual tool of parsing generation, does the analysis of a given grammar. This improvement is based on the utilization of a better method of parsing, that brings a speed of parsing faster compared with others parsing techniques and also is a method that requires less memory, during its parsing process. So in the solution will be implemented an SLL($k$).

With the utilization of an SLL($k$) is supposed to overcome one of the LL(1) algorithm problems. For any grammar SLL($k$), with a finite $k$, this new parser will not have any trouble during the analysis of the grammar. The end user will not have any notion of conflicts resolution that may appear, for a given $k$.

With this solution is supposed to have an improvement of time, during the parsing process of a given example input. Beyond that, is supposed to have an improvement of memory used in the parsing process.

3.1. Tool

The choice of Byacc as a tool to implement the proposed solution was based, in particular, to some factors. First, Byacc in its essence, generates an LALR(1) parser, which is more slower than LL(1) parser, for the same input and uses much more memory during the execution process of parsing than LL(1) parser. Another reason to choose the Byacc, was related to the fact of Byacc being an open-source tool, which makes, from a research point of view, quite useful to have the source code as-is and have a version of it very stable, letting it a good work base to develop the solution without fear of any update that could corrupt the work done so far.

3.2. Approach

One of the goals of the solution proposed is to keep the same structure of the Byacc, regarding both files, the input file and output file, with the difference that the $y.tab.c$ file, where the parser is placed, will have an LL parser instead of an LALR. Therefore, the Byacc was used just as is, which makes the lexical analysis being be done as before, without any modification.

One of the crucial pieces of an LL parser is its parsing table, therefore, is important to understand how the parsing table of the solution was constructed, since the parser used was an SLL($k$), instead of the traditional LL(1) parser. The choice of the SLL($k$) as the parser was due to the amount of grammars that it can analyze compared to the simple LL(1) parser, because the parsing table of an LL(1) parser contains, for the same grammar, too many conflicts, which lets the parsing process indeterministic. For instance, the grammar of Figure 4 will have an LL(1) parsing table with conflicts, as is shown in following table,

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td></td>
<td>r2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r1</td>
<td></td>
<td></td>
<td>r3</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: LL(1) Parse Table for the grammar in Figure 4

in the entry (S,a). For this reason, it is necessary have an SLL($k$), so the table can be deterministic for each entry and consequently the parsing process.

When trying to apply the traditional algorithm of LL(1) parser, over the parsing table construction, in the SLL(2) parsing table, where the 2 is the length of lookahead, the resulting table will be something like this:

<table>
<thead>
<tr>
<th></th>
<th>aa</th>
<th>ab</th>
<th>ac</th>
<th>ba</th>
<th>bb</th>
<th>bc</th>
<th>ca</th>
<th>cb</th>
<th>cc</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>r1</td>
<td>r2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 3: SLL(2) Parse Table for the grammar in Figure 4

that, besides being a deterministic parsing table, is substantially larger than the LL(1) parsing table for the same grammar.

The SLL(k) tables, has \( N \times T^k + 1 \) size, therefore, for a higher \( k \), the resulting SLL(k) parsing table, will be huge, when only some entries could needed to be disambiguated. With this issue in mind, the solution proposed will adopt an SLL(k) done in Antlr, where it applies an SLL(1)(k) and than verifies the \( k \) symbols individually to disambiguate the entries of the table that could have a conflict.

In this approach, the SLL(k) will use an LL(1) parsing table, that will be done as the original algorithm of LL(1) parser demands. Concerning to the entries that would have a conflict, the parser will analyze them, separately, for the \( k \) that was provided.

The analysis, of what rule should be applied when a conflict appears, can be made with help of a DFA. So, the entry in conflict will point to a DFA that, with the lookahead \( k \) disambiguates the conflict. The Table 2 will have a pointer, in the entry (S,a), to a DFA like the Figure 6.

While in the implementation of the second approach, the parser will have something like:

![Figure 6: DFA choice of the entry (S,a)](image)

being the \( q_0 \) the state initial, \( q_1 \) the state where is applied the \( r_1 \) after the parser has processed the symbol \( a \) and \( q_2 \) the state where is applied the \( r_2 \) after the parser has processed the symbol \( b \). In this case, the parser will drive through the states until it finds a terminal state, which contains the right rule or until there is no state to drive to, throwing an error.

With this approaches the solution proposed will manage to minimize memory used during the parsing process and increase the speed, since it will no longer use the traditional SLL(k) parsing table, where the \( k \) could increase the the parsing table exponentially.

4. Implementation

The algorithm behind the solution was based in several aspects, like the FIRST\(_k\) calculation, the FOLLOW\(_k\) calculation, the SLL(1) parsing table generation, where each entry of it that had a conflict would be automatically decided through a DFA and therefore, there will be as many DFA’s as the number of conflicts in parsing table. At the end, each DFA generated will be converted in a matrix, just like an Adjacency Matrix, that will represent the whole logic of the respective DFA, which will be included in the parser generated.

This matrix will have for one side the number of states of the respective DFA and for other side will have the existing tokens, where each entry \( E(s,t) \) of this table can have three possibilities and meanings. The number zero 0, if in the state \( s \) there is no way to go to other state, through the token \( t \); a negative number, if the state \( s \) is a terminal state and in the case the number negative represents the opposite number of the rule to be selected; and a positive number, which represents the next state to go when the parser is in state \( s \) and sees a token \( t \).

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>-2</td>
<td>-2</td>
<td>-2</td>
<td>-2</td>
</tr>
</tbody>
</table>

Table 4: DFA Matrix of conflict in the entry (S,a) of the Table 2

Algorithm 1: General Algorithm

1  begin
2    | Calculate_FIRST_SET()
3    | Calculate_FOLLOW_SET()
4    | Parse_Table_Fulfillment()
5    | DFA_Conversion()
6  end

7  Parse_Table_Fulfillment()
8  foreach entry in parse_table do
9    | if entry \( \exists \) conflict then
10       |   Build_DFA()
11  end
12 end

5. Results

In order to evaluate the solution, was take into account two measures, the time and the memory. Both were recorded during the parsing execution time. The time measured means the time the parser
took to parsing a given input. The memory measured means the memory the parser spent to process a given input.

It were executed a test battery and each test case result corresponds the average of ten unit test repeated. The battery tests used to evaluate were:

1. test a short LL(1) grammar over the LALR(1) and over the SLL(1);
2. test a short LL(k) grammar over the LALR(1), over the SLL(K) and over SLL(1);
3. test a subset of C programing language grammar over the LALR(1) and over the SLL(K);
4. test a short non-SLL(k) grammar over the LALR(1) and over the SLL(K);
5. test a short non-LALR(1) grammar over the LALR(1) and over the SLL(K);

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Parser</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLL(1)</td>
<td>tested</td>
</tr>
<tr>
<td>SLL(4)</td>
<td>tested</td>
</tr>
<tr>
<td>C Language</td>
<td>tested</td>
</tr>
<tr>
<td>Non-SLL(k)</td>
<td>tested</td>
</tr>
<tr>
<td>Non-LALR(1)</td>
<td>tested</td>
</tr>
</tbody>
</table>

Table 5: Battery tests

5.1. SLL(1) grammar
This test was intended to compare the performance of an LALR(1) parser against a normal SLL(1) parser with a simple LL(1) grammar. Was given two inputs, one with a simple and single statement and other with the same statement but repeated five hundred times, in order to give the parsers some load. The following tables describes the results obtained.

<table>
<thead>
<tr>
<th>Grammar</th>
<th>LALR(1) Parser</th>
<th>SLL(1) Parser</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>26,593</td>
<td>23,193</td>
</tr>
<tr>
<td>Time</td>
<td>0.000820</td>
<td>0.000307</td>
</tr>
</tbody>
</table>

Table 6: ex1. Memory Results in bytes and Time Results in seconds

In this test case the SLL(1) parser had a parsing time of 0.000513 seconds more faster and needed less 3,400 bytes than the LALR(1) parser, for the first input and the SLL(1) parser had a parsing time of 0.077381 seconds more faster and needed less 54,330 bytes than LALR(1) parser, for the second input.

5.2. SLL(4) grammar
In this test, was reused the previous grammar but with the introduction of some rules that turn the grammar an SLL(4) grammar. Was tested two inputs, one containing a statement that did not needed to resolve any conflict and other that needed. Both statements were repeated five hundred times again. The following tables describes the results obtained.

<table>
<thead>
<tr>
<th>Grammar</th>
<th>LALR(1) Parser</th>
<th>SLL(4) Parser</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>132,933</td>
<td>78,603</td>
</tr>
<tr>
<td>Time</td>
<td>0.212533</td>
<td>0.131364</td>
</tr>
</tbody>
</table>

Table 7: ex2. Memory Results in bytes and Time Results in seconds

With an input statement more complex or one that requires more steps, the SLL(4) parser behaved better than the LALR(1), with less memory and more faster. The input1 is identical to the input2 of the previous test, but with other grammar and the SLL(4) parser had a parsing time of 0,081169 seconds more faster and required less 54,330 bytes than the LALR(1). In the input2, which was with other statement, the SLL(4) parser had a parsing time of 0,08282 seconds more faster and required less 54,330 bytes than the LALR(1).

5.3. Programming language grammar
The Programming language grammar chosen was the grammar of the Programming language C and the baseline used was the ANSI C grammar of 1999. However, the grammar used was subset of this grammar, where there were no left-recursion problems. The lookahead used for the SLL(k) was 3.

<table>
<thead>
<tr>
<th>Grammar</th>
<th>LALR(1) Parser</th>
<th>SLL(3) Parser</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>35,836</td>
<td>33,586</td>
</tr>
<tr>
<td>Time</td>
<td>0.443219</td>
<td>0.286402</td>
</tr>
</tbody>
</table>

Table 8: ex3. Memory Results in bytes and Time Results in seconds

Although the grammar and the input used for this test are quite simple, the SLL(3) parser achieved a better metrics than the traditional LALR(1) parser, either with the memory or the speed parsing time and the test purpose was achieved. In this test, the SLL(3) had a parsing time of 0.156817 seconds more faster and required less 2,250 bytes than the LALR(1).
5.4. Non-SLL(k) grammar

The intention with this type of test was to understand how bad an SLL(k) would behave against an LALR(1) parser when the grammar is a non-SLL(k). The grammar of this test was:

\[
\begin{align*}
S & \rightarrow A^r_1 \mid B^r_2 \\
A & \rightarrow aaA^r_3 \mid aa^r_4 \\
B & \rightarrow aaB^r_5 \mid a^r_6
\end{align*}
\]

Figure 7: Non-SLL(K) Grammar

Was tested three inputs, the statement `aa`, the statement `aaa` and the statement `a2048`, the LL parser used was the SLL(3). After the test, the result was as expected. The SLL(3) parser only could run the first input, when the LALR(1) parser managed the three inputs.

Since the SLL(3) parser results were not good, it is not interesting compare neither the memory or the parsing speed time. The interesting point is the fact the SLL(3) parser could not managed a sequence of tokens `a` with a length greater than two, at the third token `a` the parser could not know if it applies the rule `r1` or the rule `r2`. For that reason only the first input was able to be parsed when the others two gave a syntax error message, since the third token, in the first input, was the token `$`.

There are two options for this kind of grammar with an SLL(k). Or the input sequence has a length equal to `k − 1`, or the lookahead `k` given is equal to `inputlength − 1` and with these options the SLL(k) can be able to parse any sequence of this kind of grammars.

5.5. Non-LALR(1) grammar

Like the previous test, the intention of this test was to is see the different behaviour among the parsers, when the grammar is a non-LALR(1). Thus, the grammar of this test was:

\[
\begin{align*}
S & \rightarrow DxB \mid Aw \\
A & \rightarrow y \mid \varepsilon \\
B & \rightarrow Sz \mid xA \\
D & \rightarrow w \mid z \mid \varepsilon
\end{align*}
\]

Figure 8: Non-LALR(1) Grammar

The input tested was the statement `(wx)333wz333`, which belong to the grammar. The lookahead given to the SLL(k) was 3.

<table>
<thead>
<tr>
<th></th>
<th>LALR(1) Parser</th>
<th>SLL(3) Parser</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>29,078</td>
<td>27,178</td>
</tr>
<tr>
<td>Time</td>
<td>0.022301</td>
<td>0.020055</td>
</tr>
</tbody>
</table>

Table 9: ex5. Memory Results in bytes and Time Results in seconds

The grammar tested was not an LALR(1) and it throws a syntax error, despite the input belonged to the grammar. The syntax error occurred when the parser read the token `z`, so the test results, of the LALR(1) parser, just have two thirds of the completed result. However and even with the uncompleted test of the LALR(1) parser, the SLL(3) parser had a parsing time, of the full input, better than the uncompleted input of the LALR(1) parser and required less memory for the whole parsing process than the two thirds that LALR(1) took.

5.6. Discussion

All these tests revealed, as was expected, that the SLL(k) could be much faster than the traditional LALR(1) and also require a lot less memory.

The memory required was affected by the grammar and the input given. The time, in average, that the SLL(k) took to parsing an input example, comparing with the LALR(1) was almost half. Therefore, it can be said that the SLL(k) can have, at most, a parsing speed approximately 0.5 faster than the LALR(1) parser. This metric was clearly influenced by the size of the input but in both parsers the time increased proportionally.

6. Conclusions

The final solution managed to achieve many important goals. One of them was the parsing technique itself, the SLL(k). This technique was thought to be better than the traditional LL(1) parsing technique where it could be left behind against grammars that has several rules of a non-terminal symbol with the same possible sequence tokens. Therefore, this rebuilt Byacc, with the SLL(k), compared to an LL(1) parsing generator tool, stays many steps ahead.

The utilization of an SLL(1) table as the default parsing table, allowed to gain an improvement of memory required compared with a theoretical SLL(k) and also compared with the LALR(1) parser.

Other important achievement was the speed parsing. Is well known that a normal LL(1) parser can be more faster than a normal LALR(1), since the LALR(1) does more steps to process the same input compared to an LL(1) parser. However, the new SLL(k) needs also to do a little more steps when a conflict appears and even that, the new SLL(k) still continues to be more faster than the LALR(1).

This goal was achieved through the utilization of the DFA.
The final achievement was regarding with the tool itself. Was intended to let the Byacc with the same behaviour so any Byacc user still continuous to use it in the same way as before and this goal was reached. The Byacc still produces the same type of files, even when is executed to generate an SLL(k), it keeps the same input flags and still has the same internal structure.

6.1. Future Work
There are some features that would be interesting to add in the solution developed. One of them would be the improvement of FIRST_{k} and FOLLOW_{k} calculations. Would be really interesting develop a new algorithms for these calculations, in order to generate an SLL(k) more faster than the generation of the LALR(1) parser.

Other interesting feature that would be nice to add, is the capability the tool accepts any grammar\(^2\), where the problem of factorization would be firstly resolved mathematically, generating a grammar more simple that can be used with a lower lookahead \(k\) if still need.

The lookahead given still is an issue when the grammar does not need a such bigger lookahead. Therefore, would be very good to add a feature that would give the ability to not be accurate any given lookahead and dynamically calculates which one \(k\) is correct for the given grammar. With this feature, the parser would not be anymore an SLL(k) but an SLL(*), just like the third version of the ANTLR tool.

Lastly, it would be great if the entire Byacc tool was rewritten with a structure design based on an LL parser, leaving only the important features. This improvement would the be good since the whole Byacc would be focused on the LL parsing process and there would be no any resource spent with something that is not needed for the LL parser generation process.

References


\(^2\)without left-recursion