#### UNIVERSITATEA “ALEXANDRU IOAN CUZA” DIN IAȘI

**FACULTATEA DE INFORMATICĂ**



LUCRARE DE LICENȚĂ

A learning tool for LR(0), SLR(1), LR(1), LALR(1) and LL(1) parsers

#### propusă de

### Tatu Georgian - Adrian

Sesiunea: iulie, 2019

#### Coordonator științific

Lect.dr. Mihai-Alex Moruz

#### UNIVERSITATEA “ALEXANDRU IOAN CUZA” DIN IAȘI

**FACULTATEA DE INFORMATICĂ**

A learning tool for LR(0) , SLR(1), LR(1), LALR(1) and LL(1) parsers

**Tatu Georgian – Adrian**

Sesiunea**:** iulie, 2019

#### Coordonator științific

**Lect.dr. Mihai-Alex Moruz**

Avizat,

Îndrumător Lucrare

Titlul, Numele şi prenumele \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Data \_\_\_\_\_\_\_\_\_\_\_\_ Semnătura \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**DECLARAŢIE**

**privind originalitatea conţinutului lucrării de licenţă/disertaţie**

Subsemntatul(a) ………………………………………………………………………………………

domiciliul în …………………………………………………………………………………………………..

născut(ă) la data de ………………..…., identificat prin CNP ………….……………..………………..., absolvent(a) al(a) Universităţii „Alexandru Ioan Cuza” din Iaşi, Facultatea de ………………………. specializarea …………………………………………………………, promoţia ………………………….,

declar pe propria răspundere, cunoscând consecinţele falsului în declaraţii în sensul art. 326 din Noul Cod Penal şi dispoziţiile Legii Educaţiei Naţionale nr. 1/2011 art.143 al. 4 si 5 referitoare la plagiat, că lucrarea de licenţă cu titlul:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_elaborată sub îndrumarea \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_, pe care urmează să o susţin în faţa comisiei este originală, îmi aparţine şi îmi asum conţinutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licenţă/disertaţie să fie verificată prin orice modalitate legală pentru confirmarea originalităţii, consimţind inclusiv la introducerea conţinutului său într-o bază de date în acest scop.

Am luat la cunoştinţă despre faptul că este interzisă comercializarea de lucrări ştiinţifice in vederea facilitării fasificării de către cumpărător a calităţii de autor al unei lucrări de licenţă, de diploma sau de disertaţie şi în acest sens, declar pe proprie răspundere că lucrarea de faţă nu a fost copiată ci reprezintă rodul cercetării pe care am întreprins-o.

Dată azi, ………………………… Semnătură student …………………………

DECLARAȚIE DE CONSIMȚĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licență cu titlul „A learning tool for LR(0), SLR(1), LR(1), LALR(1) and LL(1) parsers”, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de Informatică.

De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea „Alexandru Ioan Cuza” din Iași, să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Iași, 25-06-2019

Absolvent Tatu Georgian - Adrian

(semnătura în original)

### **Table of Contents**

[Chapter 1 Contributions 7](#_Toc8630)

[Chapter 2 Problem Description 7](#_Toc27053)

[Chapter 3 LR(0) 8](#_Toc20289)

[Chapter 4 Bottom – up syntactic analysis 13](#_Toc10632)

[Chapter 5 FIRST 16](#_Toc28510)

[Chapter 6 FOLLOW 18](#_Toc32458)

[Chapter 7 SLR(1) – Simple LR(1) 19](#_Toc20714)

[Chapter 8 LR(1) 20](#_Toc12242)

[Chapter 9 LALR(1) - Look Ahead LR(1) 24](#_Toc20469)

[Chapter 10 LL(1) 26](#_Toc17134)

[Chapter 11 Similar Solutions 30](#_Toc4617)

[Chapter 12 Statistics and Comparisons 31](#_Toc1555)

[Chapter 13 Installation 34](#_Toc14745)

# 

Introduction

Most parsers in use today rely on one of the following algorithms: LL(1) , LR(0), SLR(1), LR(1) or LALR(1) for a good reason : efficient linear time and space complexity, one good example that comes to mind being the LALR parser used in YACC. There are also other algorithms such as Earley, GLR or CYK(Cocke – Younger – Kasami) that can parse a larger set of formal languages but have worst – case complexity of O(*n*3).Backtracking can also be used but the time complexity will be exponential. That is why it is most imperative for these algorithms to be well understood by as many people as possible working in fields that are even barely related to parsing of any sort. The software tool that will be presented in the following chapters was designed with this purpose in mind.

As for why I chose this particular theme, my interest in parsing methods started with a general curiosity about formal grammars and automata and the mathematical theory behind them. Later, I discovered one of the more practical sides of this theory and that is how it can be used to generate interesting and useful parsers for formal languages and my interest in the this field grew even more , thus becoming the reason for why I chose this theme.

As for implementation details, I used the java programming language(version 11) together with JavaFx SDK and GraphViz for visual representation of parsing tables, automata and parsing trees.

Firstly, there will be one chapter describing the problem of parsing and some general terminology that will be used in the later chapters. After that, there will be one chapter dedicated for each of the above-mentioned algorithms starting with LR(0) since all the other algorithms ( except for LL(1) because it is a top-down parsing algorithm) build off of it. After the LR(0) chapter there will be a chapter explaining the implementation of the bottom – up syntactic parsing algorithm used. There will also be two chapters after this explaining the implementation of the algorithms used to determine the FIRST and FOLLOW sets since they are required in the implementation of SLR(1) and LR(1) which will be in the next two chapters. After that there will be a chapter describing the implementation of the LALR(1) automaton since it’s construction is based off of the LR(1) automaton. After that, there will be a chapter describing some of the already existing tools that serve a similar purpose. Next, there will be a chapter discussing comparisons between the algorithms by showcasing runtime statistics (such as execution times and the amount of memory used). Finally, there will be one small chapters discussing the installation of the application.

# Contributions

Similar tools for parsing table and automata visualizations already exist online but they are scarce at best and most of them are limited to a subset of the above-mentioned algorithms or they do not implement the actual syntactic analysis and they provide only the automaton and the parsing table. The tool described in this paper will provide the automata, parsing table and a visualization of the syntactic analysis algorithm under the form of yet another table describing the steps of the algorithm as well as a rightmost derivation parsing tree for better visualization of the list of derivations that will result from the syntactic parsing. There will be a visual representation of the FIRST and FOLLOW sets of the non-terminals present in the grammar. A set of 20 simple grammar examples will also be provided to the user for a simple demonstration of the different parsers.

# Problem Description

We define the following acronyms :

* CFG stands for Context Free Grammar
* CFL stands for Context Free Language

The problem can be formally described as follows:

* Input: A CFG , let us denote it by G, and an ordered sequence of tokens W (from now on referred to as a word) that may or may not belong to the CFL (denoted by L) generated by G.
* Output: True, if W belongs to L and the list of derivations that need to be applied in order to obtain W from the start symbol of G(by using leftmost or rightmost derivation) and False otherwise. Please note that if G is ambiguous there may be more than one derivation tree for W but this type of parsing does not fall in the problem scope.

As this is a very hard problem for a CFG of any type we have to limit ourselves to only a subset of those, more specifically Deterministic Context Free Grammars(DCFGs). DCFGs are a proper subset of CFGs that can be derived from Deterministic Pushdown Automata and the languages they generate are called Deterministic Context Free Languages(DCFLs). DCFGs are a subclass of unambiguous CFGs, since non- deterministic unambiguous CFGs can exist as well. DCFGs are of great importance in practice as they can be parsed in linear time. These grammar classes are referred to by the type of parser that can parse them (LR(0), SLR(1), LR(1) etc.).

# LR(0)

LR (Left-to-Right, Rightmost derivation in reverse) parsers are a type of parsers that can

very efficiently parse DCFGs in guaranteed linear time. The name LR is always followed by a numerical qualifier denoting the number of lookaheads(for example LR(1) or LR(k)).This means that the LR parser, in order to avoid guessing or backtracking(which would result in exponential time) is allowed to “peek” k lookahead input tokens before deciding how to parse earlier tokens.

Before we continue any further, we need to define what a LR(0) item is : a production rule with a dot in the right hand side of the rule denoting how much of the right hand side has been “seen” by the algorithm. A LR(0) state is a set of LR(0) items and the transitions between the states are made via a Terminal or Non Terminal.

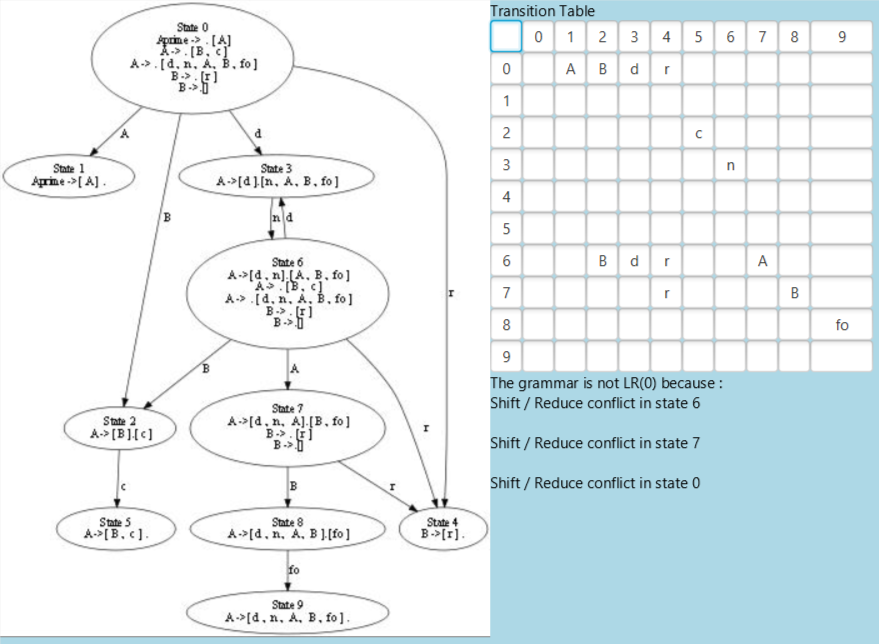
LR(0) states are created by applying the LR(0) closure procedure to LR(0) items which

will return a set of LR(0) set of items that is then added to the LR(0) state itself. The closure of a LR(0) item is obtained in the following way:

1. Add the LR(0) item given as input to the closure.
2. For all the items in the closure that have a Non-Terminal N following the dot add all LR(0) items that meet the certain criteria:
   1. The Non-Terminal in the left hand side is N
   2. The dot in the right-hand side is situated before any other token (this includes empty string rules as well)
   3. It does not belong to the closure already
3. Repeat step 2 until no more items can be added.

The LR(0) automaton can be constructed by following the next steps:

1. Augment the grammar given as input. This simply means to add a new start symbol S’ that will replace the old start symbol S and add a new production S’ -> S.
2. The initial state in our automaton is the one that will be obtained from the closure of the S’ -> .S item. We will set this state as unmarked which means that it has not been fully processed yet.
3. For each unmarked state A do the following:
   1. For each of the items in A that have a token T after the dot , obtain a new item by incrementing the dot position and add it’s closure to a new state B. If there is more than one item in A that has T after the dot add their closures to B as well.
   2. If B does not belong to the set of states add it and set it as unmarked and add a new edge that goes from A to B via the token T. An edge is an object that stores a reference to state A, state B and the token T.
   3. Set state A as marked
4. Repeat 3 until there are no more unmarked states
5. Print the automaton by using the GraphViz dot file format
6. Render the transition table of the automaton by using the JavaFx API by rendering a grid pane containing a matrix of non-editable TextArea objects. Each state will have a row and a column associated and in the cells there will be the tokens associated with the edges. The table should be interpreted in the following way: the state associated with the row is the state from which the edge leaves and the state associated with the column is the destination of that edge. If a cell is empty it means no edge exists between those two states. The width of each column will be scaled based on the amount of text in the cell with the most text in that column.



In the above image a visual representation of a LR(0) automaton can be observed alongside it’s corresponding transition table and the conflicts.

The structure of the LR(0) parsing table and all subsequent parsing tables will have one row for each state in the automaton with two sections : Action which will have a column for each terminal and Go To which will have a column for each non-terminal. The parsing table will tell our syntactic analysis algorithm what step to take in what situation. The possible steps are:

* SHIFT – this means that the algorithm will transition from the current state to a new state via the token indicated in the table column.
* REDUCE – Apply the production rule indicated in the table cell.
* ACCEPT – the word has been parsed successfully.
* ERROR – The word cannot be parsed.

Cells in the Go To section of the parsing table can have at most one SHIFT in them or they can be empty in which case it is considered an ERROR. Cells in the ACTION section of the table, however, can have more than one step in them. If the cell contains a SHIFT and a REDUCE or a SHIFT and an ACCEPT in no particular order we say that state has a SHIFT/REDUCE conflict. There can also two REDUCE steps in the same cell in which case we say that state has a REDUCE/REDUCE conflict. If the cell is empty it is the same situation as in the Go To section of the table and it is considered to be an ERROR. If the LR(0) parsing table has any conflicts then the grammar is not LR(0) and the syntactic analysis algorithm cannot be used to parse it.

Before describing the algorithm for the construction of the LR(0) parsing table we need to define the notion of a final state. A LR(0) state is final if and only if it contains exactly one complete LR(0) item, which means that the dot is not followed by any token(empty-string productions are included as well).

There is another method of telling if a state has a conflict directly from the automaton. If the state in question contains more than one complete item then the state has a REDUCE /REDUCE conflict. If the state contains one complete item and another item that has terminal after the dot then the state has SHIFT/REDUCE conflict.

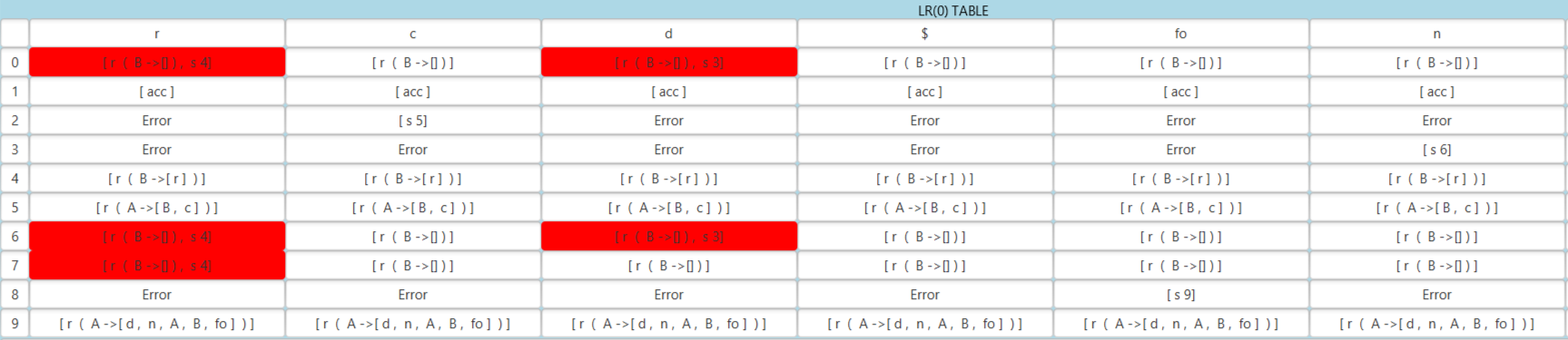
The parsing table will also contain a new terminal that does not exist in the grammar given as input usually denoted as ‘$’. This terminal will be appended to the word given as input and it signifies the end of the word. It is similar to an EOF character in the case of system files or null-terminator character in the case of strings.

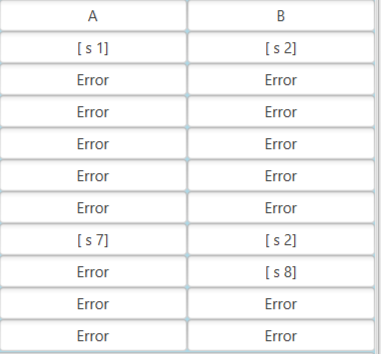
In the following algorithm description the item S’ -> S. will be referred to as the accept item. Note that this is also a complete item.

The LR(0) parsing table can be constructed in linear time as it is necessary to loop through all the states in the automaton. The steps are:

1. For each state A in the automaton:
   1. Initialize all the cells in that row as empty.
   2. Check if A is conflict free using the above-mentioned method. If it is then:
      1. Check if A is final. If it is then:
         1. If A contains the accept item add ACCEPT to all the cells in the Action section of the row. If A does not contain the accept item it means it contains some other complete item. In this case we add REDUCE using the rule from the complete LR(0) item to all the cells in the Action section of the row and we also add a SHIFT for each edge going out of state A. Each of these edges will have a token associated with it and in the cell corresponding to the column of said token we add SHIFT to the destination state of the edge.
         2. If A is not final then just add the SHIFTs as described above.
      2. If A is not conflict free then:
         1. If A contains the accept item then add ACCEPT in all the cells in the Action section of the row.
         2. If no ACCEPTs were added in the previous step then if A contains any complete items add REDUCEs as described previously.
         3. Add any SHIFTs if needed as described previously.
2. Render the Parsing Table in the graphical interface using the JavaFx API by creating a grid pane containing a matrix of non-editable TextField objects containing a string describing the list of steps in that cell(if the size of this list is greater than one that means we have a conflict and the respective TextField will be colored red).The steps are described in the following short-hand notation:
   1. s x stands for : SHIFT to state x where x is an integer
   2. r( LHS -> RHS) stands for : REDUCE using the rule that has with the corresponding left-hand and right-hand sides where LHS is a non-terminal and RHS is a sequence of alternating terminals and non-terminals or the empty string.
   3. acc stands for : ACCEPT
   4. Error stands: ERROR which will only be added if there are no other elements in that cell

The top row of the table (except for the cell in the top-left corner) will be dedicated to assigning a designation to the columns and will contain the tokens associated with each column(terminals for the Action part and non-terminals for the Go To part).The first column of the table(except for the cell in the top-left corner) will be dedicated to assigning a designation to the rows and will contain the numbers of the states associated with each row. The cell in the top-left corner has no particular meaning and is there just so that the table has perfect rectangular shape. From left to right we have the action part of the table and then the Go To part.The column widths will be scaled similarly to the automaton transition table.



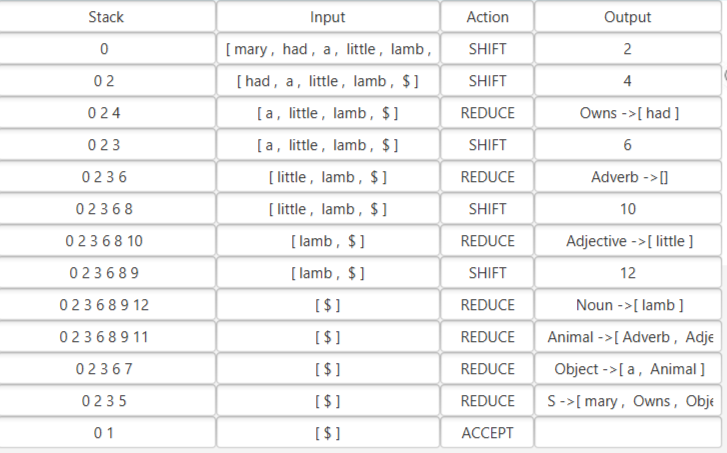


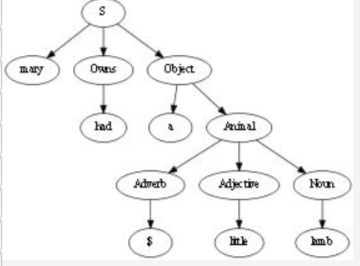
In the above two pictures a LR(0) parsing table is represented. In the first picture the Action part can be seen and the Go To part is in the second picture. In this case, the action part of the table usually is considerably larger than the Go To part, although this also depends on the number of terminals and non-terminals in the input grammar.

# Bottom – up syntactic analysis

The algorithm presented in this chapter works off of the parsing table presented in the earlier chapter. All the other bottom-up parsers ( SLR(1), LR(1), LALR(1) ) will have an associated table and automaton but the syntactic analysis algorithm will be the same for all of them. The algorithm will use a stack that can either contain states or tokens.

The algorithm is the following:

1. Append the “$” token to the end of the word to be parsed and push state 0 of the automaton into the stack. Initialize an integer I to 0 that will keep track of the current position in the word.
2. Peek into the top of the stack and identify the current state S and also identify the terminal T at position I in the word. We will call the cell in the parsing table on row S and column T (in the action section) C. C will contain a list of possible actions for the algorithm.
   1. If the list in C is empty or has more than one element the algorithm will stop with an error.
   2. Else:
      1. If C contains a SHIFT then push T then S into the stack and increment I.
      2. If C contains a REDUCE then pop the stack twice and peek into the top of the stack to identify the previous state S’. Identify cell C’ on row S’ and column N where N is the non-terminal from the left-hand side of the rule in C. Since C’ is in the go-to section of the parsing table it will always contain a SHIFT. Push N then the state in C’ to the stack.
      3. If C contains an ACCEPT then the algorithm stops.
3. Repeat 2 until the algorithm stops either with an ERROR or an ACCEPT.
4. Render the steps the algorithm has taken in the interface. For each type of step we have the current stack, the part of the input word that has not yet been processed (all the tokens after position I) , the taken action(SHIFT, REDUCE, ACCEPT or ERROR) and the output which is the associated production rule in the case of REDUCE steps and the associated state in the case of SHIFT steps. All of this information will be rendered in the form of a table similar to the parsing table or the transition table presented earlier. Here is an example:
5. Render the right-most derivation parsing tree obtained from the rules in the REDUCE steps in reverse order. The parsing tree is constructed in the following way:
   1. For each of the REDUCE steps in reverse order do:
      1. If the root of the tree is null then create a new unprocessed node, set it as root and current node and add the non-terminal from the left-hand side of the rule to it.
      2. If the root already exists then set parent as the current node. Then do:
         1. Current node becomes the right most unprocessed child of the parent node that contains the non-terminal from the left-hand side of the current rule.
         2. If the parent is not the root then parent becomes the parent of the current parent (we go up a level in the tree).
         3. Repeat the above two steps until the current node is not null.
      3. If the right-hand side of the current rule is not empty then:
         1. For each token in the right-hand side create a new unprocessed node containing it and the current node as it’s parent.
      4. If the right-hand side is the empty string then create a new unprocessed node containing “$” and set the current node as it’s parent. This is done in order to better visualize empty string productions in the tree.
      5. Set current node as processed.
      6. Render the tree using GraphViz dot file format. Here is an example:



As you can see the leaves in the tree are the terminals in the input word (except for “$” which is the empty string).

# FIRST

We define FIRSTk(X) for a token X as the set of all substrings of length at most k that

begin the strings derivable from X. More formally:

*FIRSTk(α)= { w ∈ Σ\* | α ⇒\* wx, |w|=k ∨ x = ε }*

We define k-concatenation (denoted by \*k from now on) operation on two strings as normal concatenation except the result is trimmed to have length at most k(only the first k tokens are considered).More formally:

*X \*k Y = if |X \* Y| <= k then X \* Y else*

*the first k symbols of X \* Y*

The above operation can be extended to sets of strings similarly to how normal concatenation can be extended to sets of strings. More formally:

*L \*k M = { x \*k y | x ∈ L ∧ y ∈ M }*

Please note that similar to standard set concatenation if either L or M are the empty set then the result will also be the empty set.

The algorithm to compute FIRSTk(X) is the following:

1. Create two empty maps F and F**’**
2. For each terminal T set F(T) = {T}
3. For each non-terminal N if a production of the form N -> εexists then set F(N) = {ε}, else set F(N) = ∅
4. For each non-terminal N set F’(N) = F(N)
5. For each rule of the form N → X1X2...Xn do:
   1. If n > 0 then F(N) = F(N) ∪ F**’**(X1) \*k F**’**(X2) \*k ... \*k F**’**(Xn)
6. Repeat 4 and 5 until for each non-terminal N F(N) = F’(N)
7. FIRSTk(X) = F(X)

Please note the following property of the FIRSTk:

FIRSTk(ab) = FIRSTk(a) \*k FIRSTk(b) (Property 0)

From the above mentioned property we can derive another useful property:

Let a = Y1Y2...Yn be a string, then FIRSTk(a) = FIRSTk(Y1) \*k FIRSTk(Y2) \*k ... \*k FIRSTk(Yn) or FIRSTk(a) = FIRSTk(Y1 \* FIRSTk(Y2 \* ... \* FIRSTk(Yn))) (Property 1)

This implies that we can compute the FIRSTk set for any string as long as we know the FIRSTk sets of all the tokens in the given grammar.

# FOLLOW

We define FOLLOWk(X) for a token X as the set of strings of length at most k that can

appear immediately to the right of X in some sentential form. More formally:

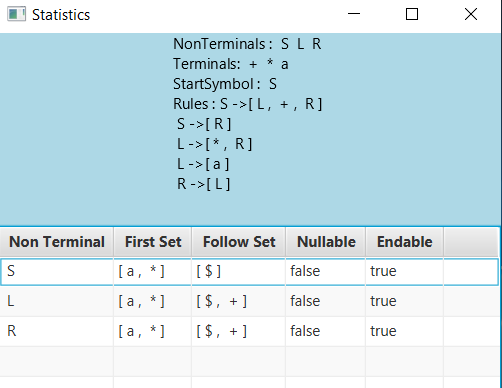
*FOLLOWk(A)= { w ∈ Σ\* | S ⇒\* xAα ⇒ xβα, w ∈ FIRSTk(α) }*

FOLLOWk can be computed in the following way:

1. Create two empty maps FL and FL’
2. Set FL(S) = {ε} where S is the start symbol
3. For each non-terminal N except for the start symbol S set FL(N) = ∅
4. For each non-terminal N set FL’(N) = FL(N)
5. For each rule of the form N → X1X2...Xn do:
   1. For i = n down to 1 do:
      1. If Xi is a non-terminal then:
         1. FL(Xi) = FL(Xi) ∪ FIRSTk(Xi+1) \*k FIRSTk(Xi+2) \*k ... \*k FIRSTk(Xn) \*k FL'(A)
6. Repeat 4 and 5 until FL(N) = FL’(N) for each non-terminal N
7. FOLLOWk(N) = FL(N)

Since all the algorithms presented in this paper will use at most k = 1 lookaheads we

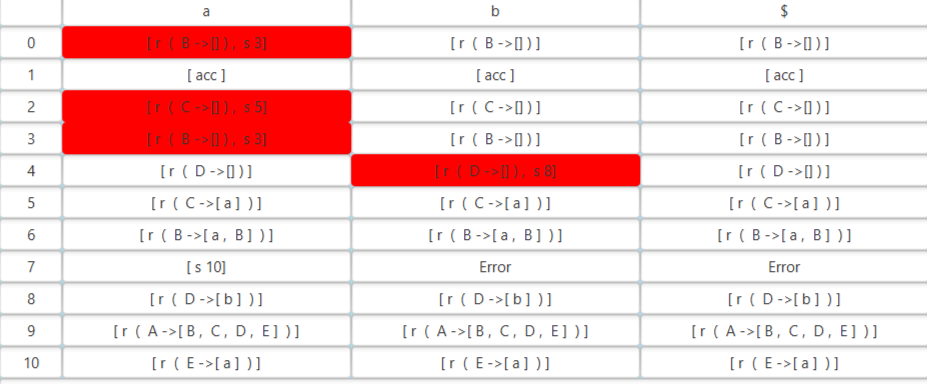
don’t actually need to compute FIRSTk or FOLLOWk for any k > 1. From now on whenever we refer to FIRST1 or FOLLOW1 we will just use FIRST and FOLLOW instead. Here is an example of FIRST and FOLLOW sets:

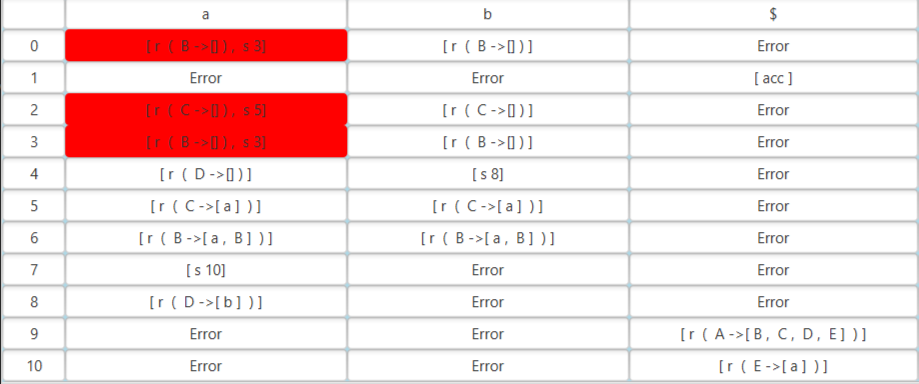


A nullable non-terminal is a non-terminal from which the empty string can be derived. This is obviously equivalent to *ε ∈ FIRST(N)* where N is the non-terminal in question. Similarly an endable non-terminal is a non-terminal from which an end of a string can be derived, which is equivalent to *ε ∈ FOLLOW(N)* where N is the non-terminal in question. In the above image “$” is used to represent the empty string.

# SLR(1) – Simple LR(1)

The only difference between LR(0) and SLR(1) is that whenever a REDUCE or

ACCEPT is added in the parsing table it is not added to all of the cells in the ACTION section of the row but only to the cells on the columns belonging to terminals that are in the FOLLOW set of left hand-side non-terminal of the production rule in the LR(0) item. This also means that ACCEPT will only be added to “$” since the left hand side of the accept item S’ -> S. will only have “$” in it’s FOLLOW set. 



In the first image we have the action part of a LR(0) parsing table and in the second image we have the action part of a SLR(1) parsing table. The same grammar was used in both cases. As it can be seen in the LR(0) table we have ACCEPT for all terminals as opposed to SLR(1) where we have ACCEPT only for “$”. The number of REDUCE moves is also greatly minimized in the SLR(1) table thus leading to less conflicts as it can be seen that the LR(0) has more conflicts than the SLR(1) table. This also good because this means the number of empty cells in the SLR(1) table is increased thus giving our algorithm better error detection allowing it to stop earlier in the case when a word that cannot be parsed is given.

# LR(1)

Like most parsers LR(1) can be automatically generated by most compiler compilers out

there such as GNU Bison, MSTA, Menhir, HYACC and LRSTAR. The special attribute of this parser is that any grammar with k > 1 can be transformed into an LR(1) grammar. However, using this method the grammar can quickly become large, repetitive and hard to understand. It is proven that LR(k) can handle all deterministic context-free languages and if any LR(k) grammar can be converted into an LR(1) grammar it is theoretically possible for this parser to handle all DCFGs which is more than what LR(0), SLR(1) , LALR(1) and LL(1) can handle. However, this parser has been avoided in the past in favor of less powerful alternatives such as LALR(1) or LL(1) due to it’s huge memory requirements.

The most basic notion this parser uses is that of a LR(1) item. It is quite similar to LR(0) item but with comes with an additional element : the lookahead. Basically, a LR(1) item is nothing more than a (LR(0) item , terminal) pair. States and edges in the LR(1) automaton are similar to their LR(0) counterparts except that they use LR(1) items instead. Same goes for the parsing table.

However, the procedure used to obtain the closure is quite different due to the existence of the lookahead. This time instead of applying the closure procedure to one item at a time we apply to an entire state, which is nothing more than a set of LR(1) items. The algorithm is the following:

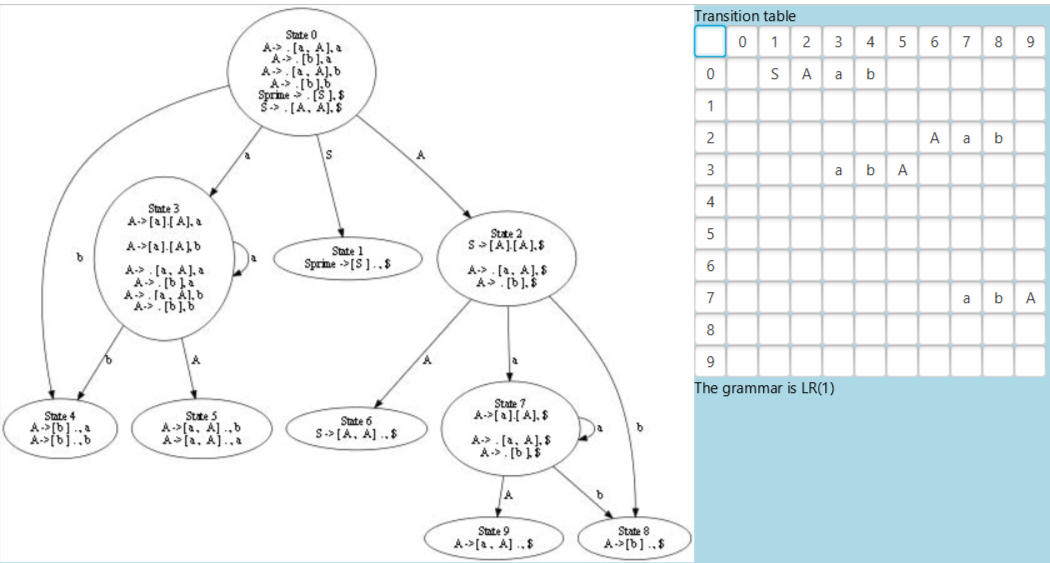
1. Initialize a Boolean variable flag to true
2. While flag is true do:
   1. Set flag to false
   2. For each LR(1) item in the state of the form LHS -> X1 X2 … Xn . A Y1 Y2 … Ym, t
      1. For each production of the form A -> RHS do :
         1. Set seq = Y1 Y2 … Ym
         2. If t is not “$” (empty string) then seq = seq \* t
         3. For each terminal f in FIRST(seq) do:
            1. Create a new LR(1) item A -> . RHS, f
            2. If the new item does not exist in the state already then :

Add the new item to the state

Set flag to true

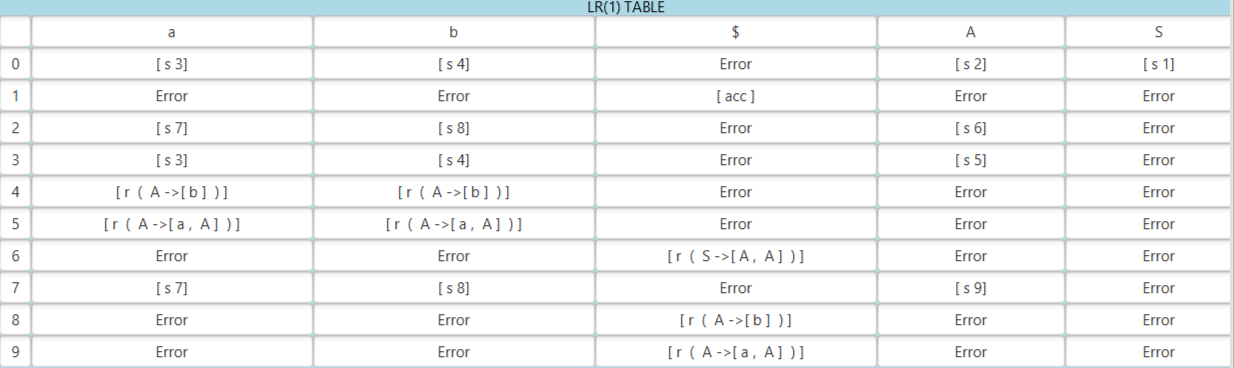
Note that in order to compute FIRST(seq) we need to use Property 1 of the FIRST set.

Although similar, the algorithm used to build the LR(1) is not quite the same as it’s LR(0) counterpart. The algorithm is the following:

1. Augment the input grammar in the same way as the LR(0) algorithm.
2. Create state 0 and add S’ -> .S , $ to it, then apply the closure procedure then set it as unmarked
3. For each unmarked state U do:
   1. For each token t in the initial grammar do:
      1. Create a new empty state U’
      2. For each item in U of the form LHS -> X1 X2 … Xn . t Y1 Y2 … Ym, o do:
         1. Add LHS -> X1 X2 … Xn t . Y1 Y2 … Yn to U’
      3. If U’ is not empty then apply the closure procedure
      4. If the automaton does not already contain U’ add it and set it to unmarked
      5. Add the edge U to U’ via token t to the automaton
   2. Set U to marked
4. Render the automaton and the transition table the same way for LR(0). Here is an example:

The LR(1) parsing table has the same structure as the LR(0) and SLR(1) parsing tables, however it’s construction method is different:

1. Add the terminal “$” to the grammar. It plays the same role as it does in the LR(0) table.
2. Sort the states in ascending order. This is necessary because the data structure used to store the states of the automaton is a set and not a list. We convert it to list and then sort it.
3. For each state in the sorted list do:
   1. Initialize all the cells in the row as empty
   2. For each item I in the state do:
      1. If I has a terminal t after the dot then add a SHIFT in the cell corresponding to t.
   3. If I is final (that means there is nothing after the dot) then:
      1. If LHS of I is S’ then add an ACCEPT to the cell corresponding to the lookahead of I. In this case, it will be “$”.
      2. If LHS of I is not S’ then add REDUCE using the production rule of I to the cell corresponding to I’s lookahead.
   4. For each non-terminal N in the grammar do:
      1. Add a SHIFT to the cell corresponding to N.
4. Render the parsing table in the same way as it was done for LR(0).Here is an example:



As it can be seen in the LR(1) automaton example that was given there are some states

for which only the lookaheads are different. This is the reason why this parser is very inefficient when it comes to space complexity. The number of states is considerably larger than LR(0) or SLR(1) since those do not have duplicate states.

# LALR(1) - Look Ahead LR(1)

The LALR(1) parser is a simplified version of LR parser.It was invented by Frank DeRemer in 1969 because of the practical difficulties at that time of implementing LR(1) parsers. He showed that the LALR(1) parser has more language recognition power than the LR(0) parser while requiring the same number of states.This makes the LALR(1) parser a memory-efficient alternative to the LR(1) parser for languages that are LALR(1).It was proven that there exist LR(1) languages that are not LALR(1).Despite this weakness, the power of the LALR(1) parser is sufficient for many mainstream computer languages , including Java, though the grammars for many languages fail to be LALR due to ambiguity.LALR(1) parsers can be automatically generated from tools such as Yacc or GNU Bison.

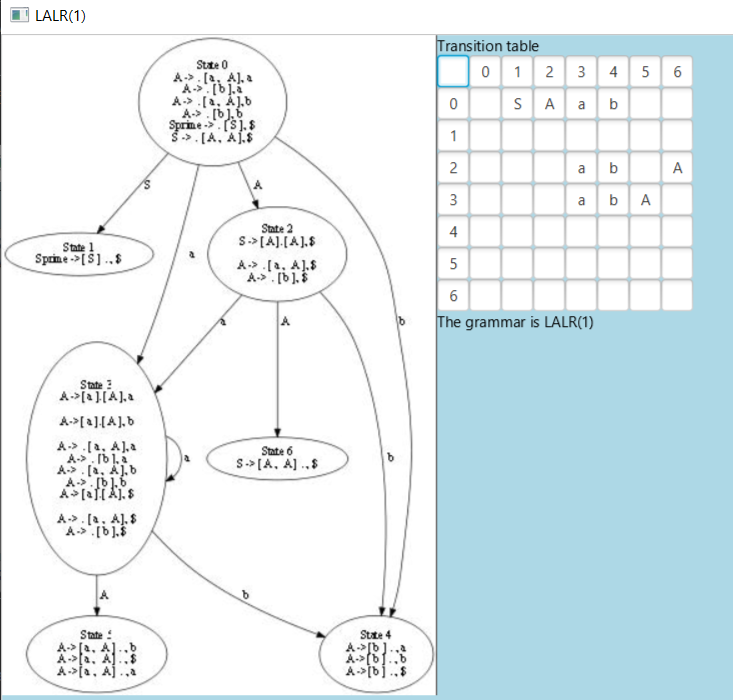
The only difference between the two parsers is the way the automaton is built.The parsing table is built using the same procedure.The method of building the LALR(1) automaton requires the LR(1) automaton to be created beforehand and then by “combining” certain states we obtain a smaller automaton.

As it was already mentioned , a LR(1) item is a LR(0) item and terminal pair.We define the nucleus of a LR(1) item as it’s LR(0) item component.I extend this notion to LR(1) states as well : the nucleus of an LR(1) state is the nuclei of all the LR(1) items contained in said state.Two LR(1) are considered to be equivalent if their nucleus is the same.Two equivalent states can be combined by adding all the LR(1) items from one state to the other, similar to set union.The algorithm to construct the LR(1) automaton is the following:

1. Create an empty map GR that will contain the combined states and the group of initial states they were created from. The combined states will act as the keys to this map.
2. Group the equivalent states from the LR(1) automaton in the following way:
   1. For each state S from the LR(1) automaton:
      1. Added = False
      2. For each group G:
         1. For each state S’ in G:
            1. If S and S’ are equivalent then add S to G
            2. Added = True
      3. If added = False:
         1. Create new empty and add S to it.
3. For each group G:
   1. Combine all the states in G in a new state CS and set GR(CS) = G
4. For each entry en in GR, let G be the value :
   1. For each state S in G:
      1. For each edge e in the LR(1) automaton that leaves S:
         1. For each entry en’ in GR, let G’ be the value:
            1. For each state S’ in G’:

If S’ is the destination of e then add a new edge in the LALR(1) automaton from the key of en to the key of en’ via the token of e.

1. Order the combined states and number them
2. Render the automaton the same way as before. Here is an example:



In the above example the same grammar was used as the one in the LR(1) automaton example.It can be seen that the number of states is considerably smaller(equal to the number of states of the LR(0) / SLR(1) automaton).The size of the parsing table will evidently be diminished as well.

# LL(1)

LL stands for Left - to - right, Leftmost derivation and as the name suggests it is a type of parser that does leftmost derivation unlike it’s LR counterpart. It is also a top-down parser that uses k=1 lookaheads. A grammar is called an LL(k) grammar if an LL(k) parser can be constructed from it.A formal language is considered to be LL(k) if it has an LL(k) grammar.An interesting property is that the set of all LL(k) languages is properly contained in the set of all LL(k+1) languages, for each k that is greater than or equal to 0.From the previous property we can deduce that not all context-free languages can be recognized by an LL(k) parser since there will a greater an LL(k+1) that can recognize more languages.LL grammars , particularly LL(1) grammars, are of great practical interest as parsers for these grammar are easy to construct and many computer languages are designed to be LL(1) for this reason.

Like the LR parsers, the LL parser uses a parsing table although this table is not constructed from an automaton, hence the need to construct an automaton for the LL parser does not exist. Because of this the structure of the LL parsing table is also different.It is not divided into two different sections ( Action and Go To) like the LR tables, instead the lines are associated with non-terminals from the input grammar and columns are associated with the terminals(including the “$” terminal, which signifies the end of the input word just like in the LR tables).In the parsing tables we will not have SHIFTS and REDUCE steps as these steps do not exist for the LL parser, instead in each cell in the parsing table there will be a list of productions from the input grammar.If a cell contains an empty list then we consider that cell contains an ERROR, similar to the LR tables.If a cell contains a list that has more than one production we consider that cell has an EXPAND / EXPAND conflict. The EXPAND step is the top-down variant of the REDUCE step from bottom-up parsers.If there is more than one production to pick from in the cell then the parser will not know which one to use in order to expand, hence the existence of a conflict in this case.

A context-free grammar is considered to be left-recursive if it contains a left-recursive production rule. A left-recursive rule has the following form : N -> N X1 X2 … Xn. Let’ s analyze an example to see exactly why this could cause the grammar to not be LL(1).Consider the following context-free grammar : A -> A a | b. As it can be easily observed the language generated by this grammar matches this regular expression : b (a)\*.At some point during the syntactic analysis the algorithm will not know if it should continue the recursion, thus expanding using the rule A -> A a or stopping the recursion thus expanding A -> b. This is caused because FIRST(A a) and FIRST(b) overlap. FIRST(A a) = {a, b} and FIRST(b) = {b}.As it can be seen their intersection is {b}. We can generalize this and deduce that a grammar is not LL(1) if the FIRST sets of two different grammar rules for the same non-terminal intersect.A left recursive rule will cause an overlap with all the other alternatives. This is also called a FIRST / FIRST conflict and it will be reflected in the parsing table as a an EXPAND / EXPAND conflict.More formally:

*Let G be :*

*1: LHS1 -> RHS1 | RHS2 | … | RHSn1*

*2: LHS2 -> RHS1 | RHS2| … | RHSn2*

*…*

*m: LHSm -> RHS1| RHS2| … | RHSnm*

*If FIRST(RHS1) FIRST(RHS2) …  FIRST(RHSn1)  ∅ or*

*FIRST(RHS1) FIRST(RHS2) …  FIRST(RHSn2)  ∅ or*

*…*

*FIRST(RHS1) FIRST(RHS2) …  FIRST(RHSnm)  ∅ then G is not LL(1).*

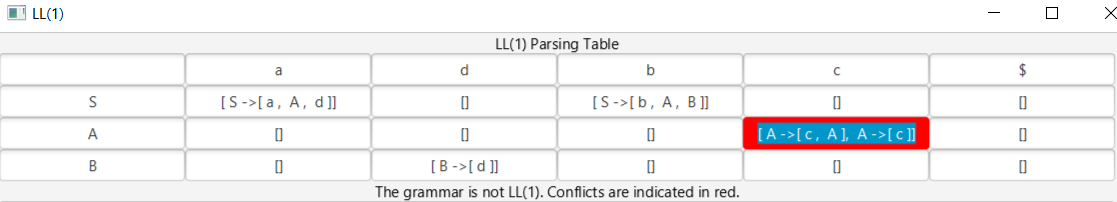
Similarly, FIRST / FOLLOW conflicts can exist as well.This is caused when the FIRST and FOLLOW sets of a nullable non-terminal intersect. A non-terminal is nullable if it can derive the empty string which means that the empty string belongs to it’s FIRST set. In this case it is impossible to know which rule to use.This will also be reflected as an EXPAND / EXPAND conflict in the LL(1) table.More formally:

*Let G be a CFG with non-terminals N1, N2, … Nm*

*If ε ∈ FIRST(Ni) and FIRST(Ni) FOLLOW(Ni)  ∅ then G is not LL(1), where 1im.*

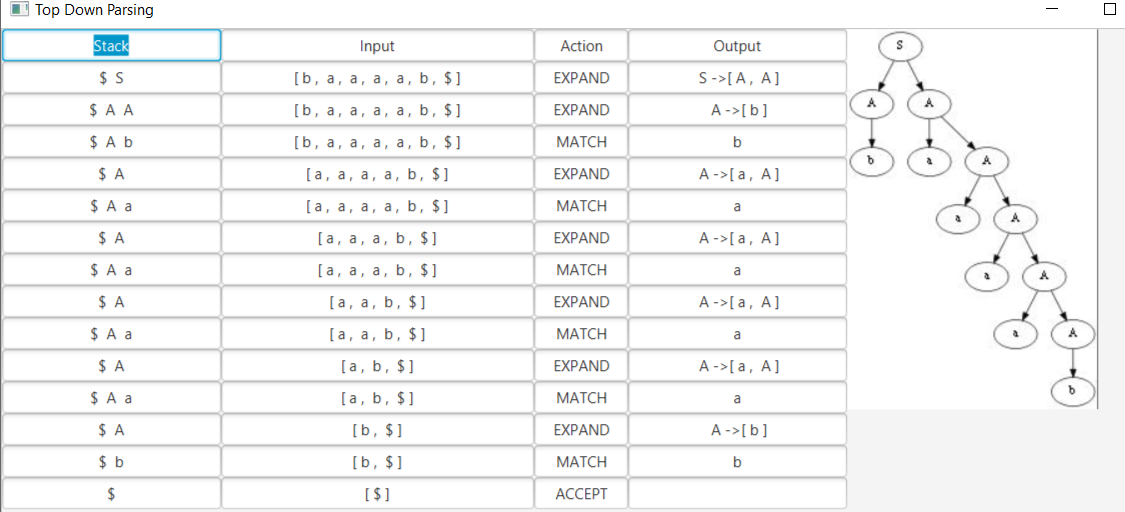
The algorithm used to construct the LL(1) table is the following:

1. Add empty cells for all the non-terminals and terminals(including “$”) in the input grammar.
2. For each non-terminal N:
   1. For each production P of the form N -> RHS:
      1. For each terminal T in FIRST(RHS):
         1. Add N -> RHS to cell (N, T)
      2. If FIRST(RHS) contains the empty string then:
         1. For each terminal T’ in FOLLOW(N):
            1. Add N -> RHS to cell (N, T’)
         2. If FOLLOW(N) contains the empty string:
            1. Add N -> RHS to cell (N, “$”)
3. Render the table in a similar manner to the LR tables. Here is an example:



Since LL(1) is a top-down parser the syntactic analysis algorithm will also be quite different from it’s LR counterpart. It will use a stack , but since there is no automaton the stack will only tokens ( terminals and non-terminals) from the input grammar. The algorithm is the following:

1. Create an empty stack
2. Set an integer variable index to 0. This variable will keep track of the current position in the input.
3. Push “$” and the start symbol S to the stack.
4. Append “$” to the end of the input.
5. Let top be the token at the top of the stack at any point in time.
6. If top is a non-terminal:
   1. If cell (top, input[index]) is empty or has a conflict then the algorithm stops with an ERROR.
   2. EXPAND step:
      1. Pop the stack once
      2. Let LHS -> X1 X2 … Xn be the rule in cell (top, input[index])
      3. Push Xn, Xn-1, … X1 to the stack.
7. If top is a terminal:
   1. If top is input[index]:
      1. If top is “$” then the algorithm stops with an ACCEPT.
      2. If top is not “$” then MATCH:
         1. Pop the stack once
         2. Increment index
   2. If top is not input[index] the algorithm ends with an ERROR.
8. Repeat steps 6 and 7 until the algorithm stops
9. Render the steps and the leftmost-derivation parsing tree. Here is an example:



The left -most derivation parsing tree was built by using the same method as for the right-most derivation one except that the steps were not iterated in reverse anymore and in the step where the current node becomes the right most unprocessed child of the parent node the current node becomes the left most unprocessed child of the parent node instead.

# Similar Solutions

Since my purpose was to create a learning tool for all of the above presented parsers in this chapter parser generators such as Yacc, Bison or ANTLR will not be considered since they fall outside the scope of this application’ s objective. There are however several similar tools that are easily that can be easily found on the internet and I shall present them in this chapter.

One of these tools is a web application that was developed by the University of Calgary that takes context-free grammar as input and provides certain statistics such as: first sets, follow sets, cyclicity and left-recursion. It also offers visualizations for parsing tables, automata and conflicts for LR(0), SLR(1), LR(1), LALR(1) and LL(1) parsers. However, it does not provide any type of visualization for bottom-up or top-down syntactic analysis and neither does it provide parsing trees.It does provide some ways for transforming a non - LL(1) grammar into a LL(1) one. These transformations are : left - recursion removal, factoring, reachability, follow set clash removal and LR(0) state annotation for LALR(1) to SLR(1) transformation.It also has an expansive set of CFG examples which are neatly categorized.

Another similar tool is Jison, an online parser generator that was written in javascript.Jison was developed by the University of South Carolina and provides an interactive web interface for LR(0), SLR(1), LR(1), LALR(1) and LL(1) parsers.It also provides a small list of examples of which the most notable is the grammar of ANSI C. The grammar must be in Backus - Naur form. It provides FIRST and FOLLOW sets visualization for all the non-terminals in the grammar.However, it only the parsing tables for the parsers listed above. Automata visualization, syntactic analysis and parsing tree visualizations are not supported.

Another application with similar features is the Compiler Construction Toolkit.It is a web application that provides FIRST, FOLLOW and PREDICT set listings and LL(1) parse tables in JSON format for easy import. It also used to provide NFA and DFA and parsing tables for LR(0) and SLR(1) parsers but as of the writing of this paper they do not seem to support these features.Only one basic example is provided.

Similar to Jison , jsmachines is a web application developed in javascript that provides closure tables ( instead of automata graphs) , parsing tables as well as visualizations for the syntactic analysis algorithm used and parse trees. These are provided only for the SLR(1), LR(1), LALR(1) and LL(1) parsers. It also provides an interactive interface for turing machines allowing the user to define the alphabet, halting and non-halting states, transitions and the initial configuration.

JFLAP is a desktop application for experimenting with formal languages topics such as: non-deterministic finite automata, non-deterministic pushdown automata, multi-tape Turing machines, several types of grammars , parsing and L-systems.In addition to constructing and testing examples for these concepts JFLAP allows it’s users to experiment with transformations from one form to another , such as converting an NFA to a DFA to a minimal state DFA to a regular expression or regular grammar. As to what functionalities it supports when it comes to parsing these are : parsing tree generation, Brute-Force parsing visualizations , CYK parsing visualization , FIRST and FOLLOW set listings and visualizations for LL(1) and SLR(1) parsing tables and syntactic analysis.

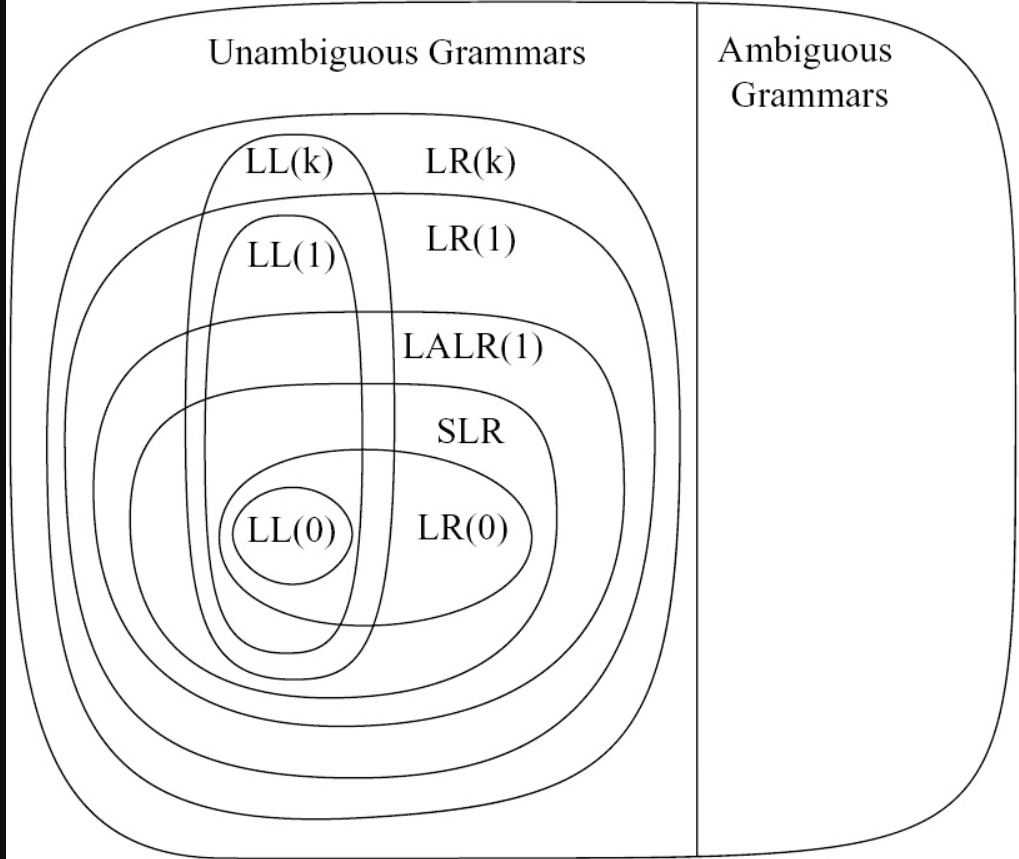
# Statistics and Comparisons

In the following chart the average time expressed in nanoseconds is shown. The rendering necessary for the interface or the interpretation of the input grammar are not included.The running times in the chart account only for the construction of the automaton(only for LR parsers) and the parsing table. Syntactic analysis is not included since almost all parsers use the same algorithm, the only different one being the LL(1) parser.

Similarly, in the following chart the average memory for each parser expressed in bytes can be seen.Like the time chart it does account for the graphical representation or for syntactic analysis.Only for the parsing tables and the automata.

When it comes to LR(0) and SLR(1) they are the fastest with SLR(1) being a bit slower than LR(0) and using a bit more memory due to FIRST and FOLLOW computations.LL(1) is not the fastest despite the fact that there is no need to build an automaton in this case but this could be because of the small sample size.However, when it comes to memory usage it is considerably better than those two which is to be expected.LR(1) and LALR(1) are by far the slowest with LALR(1) being a little bit slower due to the set reunion computations needed to build the automaton, but the memory usage is considerably smaller than LR(1) and even extremely close to LL(1).Let’s also compare the language recognition power of all the parsers.

Since it uses no lookaheads LR(0) is the weakest one. Since the set of all SLR(1) grammars properly contains the set of all LR(0) grammars then it is obvious it is a more powerful parser with similar time and memory usage, however still not good enough to be used in practice.By using the same argument as before LALR(1) is a more powerful parser than SLR(1) and it is in fact powerful enough to be used in practice.The memory usage is of LALR(1) is comparable to SLR(1) and LR(0) and it’s a far stronger alternative but much slower as well.The language recognition power of the LL(1) is similar to that of LALR(1) and their memory usage is similar as well. The only difference is in running time, which is where LL(1) excels over LALR(1).This parser is used in practice as well.LR(1) is the parser with the most power, but it’s running time is almost the same as LALR(1) and the memory requirement is huge. Similar to LR(0) and SLR(1) this parser is not used in practice.Below is a diagram of the grammar sets that can be parsed by each parser and the relationships between them.



# Installation

Requirements: JDK version 11.0.1 and JavaFx SDK version 11.0.2

Step 1: Unzip the archive.

Step 2: Open the start.bat file and change the path to the lib directory from the JavaFx SDK. You can add an environment variable or just put the whole path there.

Step 3: Run start.bat

If you are using Linux or Mac add #!/bin/bash on the first line and mark it as executable using chmod.

## Conclusion

As it was stated in the beginning these are some of the most powerful parsing algorithms that are in use even today and I have demonstrated that while speaking about each one throughout the course of this paper.Some of the most popular programming languages that are in use today , such as java, may use some of these parsers. The time complexity of the syntactic analysis is linear for each one with the parsing being computed only one time by a parser generator.I think the fact that even though all of these algorithms were discovered a long time ago and are in use even today speaks to their legacy.Not only that, but if one is ever required to ever understand or work on more complex parsers these parsing methods can offer a very good starting point.

As to my personal opinion , I think I have done an adequate job in providing a useful visualization of how these parsers work for educational and maybe even debugging purposes and I hope it will serve future experts in this field very well. I also had fun rediscovering and implementing all of them from scratch.

In the future I would like to tackle more complicated and more powerful parsers such as Earley or GLR and maybe even some of the more complicated aspects of automata and formal language theory such as turing machines, pushdown automata , combinational logic and grammar induction using neural networks or genetic algorithms.It would also be interesting to explore some more exotic computational models such as cellular automata and quantum turing machines.

## Bibliography

 DeRemer, Frank; Pennello, Thomas (October 1982). ["Efficient Computation of LALR(1) Look-Ahead Sets"](http://3e8.org/pub/scheme/doc/parsing/Efficient Computation of LALR(1) Look-Ahead Sets.pdf)

[Knuth, D. E.](https://en.wikipedia.org/wiki/Donald_Knuth" \o "Donald Knuth) (July 1965). ["On the translation of languages from left to right"](http://www.cs.dartmouth.edu/~mckeeman/cs48/mxcom/doc/knuth65.pdf)

Franklin L. DeRemer (1969). ["Practical Translators for LR(k) languages"](https://web.archive.org/web/20120405124425/http://computer-refuge.org/bitsavers/pdf/mit/lcs/tr/MIT-LCS-TR-065.pdf) . MIT, PhD Dissertation.

Pager, D. (1977), "A Practical General Method for Constructing LR(k) Parsers", *Acta Informatica 7*, pp. 249–268

Rosenkrantz, D. J.; Stearns, R. E. (1970). "Properties of Deterministic Top Down Grammars". *Information and Control*.

Dick Grune; Ceriel J.H. Jacobs (29 October 2007). *[Parsing Techniques: A Practical Guide](https://books.google.com/books?id=05xA_d5dSwAC&pg=PA585)*. Springer. pp. 585–. [ISBN](https://en.wikipedia.org/wiki/International_Standard_Book_Number" \o "International Standard Book Number) [978-0-387-68954-8](https://en.wikipedia.org/wiki/Special:BookSources/978-0-387-68954-8" \o "Special:BookSources/978-0-387-68954-8).

Aho, Sethi, Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986. [ISBN](https://en.wikipedia.org/wiki/International_Standard_Book_Number" \o "International Standard Book Number) [0-201-10088-6](https://en.wikipedia.org/wiki/Special:BookSources/0-201-10088-6" \o "Special:BookSources/0-201-10088-6)