Efficieny of Parsing Methods for Context-Free Languages

Vaastav Anand
University of British Columbia

1 Parsing and Parsing Methods

Parsing is being used extensively in a number of disciplines: in Computer Science (for Compiler Construction, Artificial Intelligence), in Linguistics (for Text Analysis, Corpora Analysis), in Computational Linguistics (for Natural Language Processing), in document preparation and coevrsion to name a few; they can be used (and perhaps are) in a far larger number of disciplines. Parsing seems like something important in the modern day and thus it leads us to ask questions as to "What is parsing?", "Why is it so important?", "Why do we want parsing to be efficient?" and "How can we make parsing efficient?".

1.1 What is Parsing?

Parsing is the process of analysing symbols represented in a linear structure, conforming to the rules of a formal grammar. This linear representation can be a sentence, a computer program, a knitting pattern, a sequence of geological strata, a piece of music, actions in ritual behavior, in short any linear seuence in which the preceding elements in some way restrict the next element [1, p. 1]. Note that if there is no restriction, then the sequence still has a grammar, but this grammar is trivial and uninformative.

1.2 Why is Parsing Important?

From a general perspective, there are a variety of reasons as to why parsing is important. The first reason is that the obtained structure from parsing helps us to process the object further. For example, in language translation, the first step would be to identify the segment in the sentence that is the subject, which aids in understanding or translating the sentence. A second reason is the fact that the grammar in a sense represents our understanding of the observations: the better a grammar we can give for the movement of the bees, the deeper our understanding is of them. A third reason is that parsers, especially error-repairing parsers, allow us to complete missing information. Given a reasonable grammar of the language, an error-repairing parser can suggest possible word classes for missing or unknowns words on ancient clay tablets [1, p. 1].

From a computer scientist's perspective, such characteristics for parsing play a major role in the fields of Compiler Construction and Computational Linguistics. In Compiler Construction, the purpose of parsing (syntax analysis) is to recombine the tokens generated by lexical analysis into a data structure called the *syntax tree* of the input text which reflects the structure of the input text. In addition to finding this structure of the input text, it is also responsible for reporting invalid texts by reporting *syntax errors* [3, p. 53]. This is important for compiled languages as it helps in finding issues in the written program before they are run and helps in preventing errors from happeing during the running of the program. Parsing in Natural Language Processing(NLP) plays a very important role as it is responsible for deriving the structure of the sentence which is then used to understand the meaning of the sentence. In modern NLP, this is done in various ways through shallow parsing, deep parsing, constituency parsing and dependency parsing.

1.3 Why do we want parsing to be efficient?

One of computer science's core foundations is to come up with soltuions for problems that are efficient in terms of its usage of resources in order to maximize the resources available. So, it is natural that solving the parsing problem becomes important. In compiler construction, we want to minimize the time and space complexity of parsing without reducing the accuracy in order to get an efficient compiler for programs. Syntax analyzers are needed in numerour situations outside complier design including programs listing formatters, programs that compute the complexity of programs and prorgrams that must analyze and react to the contents of a configuration file. Thus, having a space and time efficient syntactic analyzer

becomes very important. In NLP, parsing efficiency is crucial when building practical natural language systems. This is especially the case for interactive systems such as natural language database access, interfaces to expert systems and interactive machine translation [4, p. xvii]. This is because we want our parser methods to be quick and efficient whilst also being accurate in order to give a smooth interactive experience to the user.

1.4 How can we make parsing efficient?

Before we delve into different ways of making parsing efficient, we need to first understand what are we trying to parse. Type 0 and Type 1 grammars according to Chomsky's hierarchy, are well known to be human-unfirnedly and will never see wide application [1, p. 72-73]. Hence, the construction of a practial and reasonably efficient parsers for these grammars is not a hot research topic. Additionally, it is also a very difficult subject as all known parsing algorithms for Type 0 and Type 1 grammars have exponential time depenednecy [1, p. 73]. However, two of the other grammar classes, namely Context-Free (Type 2) and Regular (Type 3) languages have been useful for describing the syntax of programming languages. The forms of tokens of programming languages can be described by regular grammars. The syntax of whole programming languages, with minor exceptions, can be described by context-free grammars [2, p. 137]. Most of the natural languages can be represented as context-free grammars. Almost all practical parsing is done using Type 2 and Type 3 grammars [1, p. 73]. Thus to make parsing efficient, we specifically need to make parsing Context-Free and Regular Grammars efficient.

This paper presents a variety of popular parsing methods and approaches, and compares and discusses their time and space efficiecny and correctness and applicability.

2 Parsing Techniques

The basic connection between a sentence and a grammar it derives from is the parse tree (syntax tree), which describes how the grammar was used to produce the sentence. For the reconstruction of this connection we eed a parsing technique and there are two broad techniques to do this parsing - Top Down Parsing and Bottom Up Parsing.

2.1 Top Down Parsing

A top-down parser traces or builds a parse tree in preorder. A preorder traversal of a parse tree begins with the root. Each node is visited before its branches are followed. Branches from a particular node are followed in left-to-right order. This corresponds to a leftmost derivation [2, p. 196]. In essence, this method tries to imitate the original production process by rederiving the sentence from the start symbol. This method is called top-down, because the parse tree is constructed from the top downwards [1, p. 66]. Top-down parsing identifies the production rules in prefix order.

Let us define the grammar $G_1 = (V, \Sigma, R, S)$ for the language $a^n b^n c^n$ with set of variables $\{S, Q\}$ where S is the start variable; set of terminals $\Sigma = \{x, y, z\}$ and rules R: as follows:

$$S \to aSQ$$

$$S \to abc$$

$$bQc \to bbcc$$

$$cQ \to Qc$$

Suppose the input sequence is $aabbcc$ then a top-down parser would work as follows:

1. The first step would yield S as for a top-down approach, the prodcution tree must start with the start symbol.

2. Now, there are 2 rules for $S : S \to aSQ$ and $S \to abc$. The second rule would require the string to start with ab, which it does not. So, the only rule left is the first rule, which after applying yields aSQ.

3. Now again we are left with the 2 rules for S like we were in the previous step. Applying the first one would yield the string $aaSQQ$ which woul inevitabl lead to $aaa...$ in the next step which contradicts the input string. Second rule however yields $aabcQ$ which seems to contradict the input string as well.

4. Now we are left with $aabcQ$ and the only rule applicable is $cQ \rightarrow Qc$. This gives us the string $aabQc$.

5. Again, only 1 rule here is applicable which is $bQc \rightarrow bbcc$ which when applied yields the string $aabbcc$ which is our input string.

Now we will take a look some of the different top-down parsing methods.

2.1.1 Unger's Method

Unger's Method is a general non-directional top-down method. It is non-directional as it acceses the input in a seemingly arbitrary order. It is based on the fact that if the input string belongs to the language at hand then the input must be derivable from the start symbol of the grammar, S. Therefore it msut be deliverable from the RHS of this symbol. Thus, if the RHS is of the form $A_1A_2 \dots A_m$ where A_i are symbols in the CFG. This, in turn means that A_1 must derive a first part of the input, A_2 a second part, etc. Unger's method tries to find a partition of the input that fits this demand with the rules [1, p. 103]. Additionally, it also requires that the entire memory must be in memory before parsing can start. In essence, Unger's method tries to cut the input string into segmenets and impose a structure on it deriving from the start symbol; if it succeeds, it has found a parsing [1, p. 134].

Suppose we have a small grammar rule $S \rightarrow ABC|DE|F$ and we want to find out if this rule derives the string $pqrs$. To do this, for each right-hand side we must first generate all possible partitions of the input sentence. Let us assume for now that the rules don't have any ϵ-Rules or loops in the grammar. Then generating partitions is not difficult. Suppose, if we have m buckets and we want to distribute n balls into these buckets such that each bucket has at least 1 ball, then each way of distributing the balls into buckets represents a possible partition of the input string into one of the RHS of the rule.So, the number of balls corresponds to the length of the input string n and number of symbols in the RHS corresponds to m. If n is less than m then no partition is possible as for now we have assumed that there are no *epsilon*-Rules in the grammar. So, everytime the string is paritioned it is reduced to a bunch of smaller sub-problems each of which is of the form whether a particular symbol derives a given substring of the input. Thus, we end up with a search problem. Unger's method uses depth-first search to solve this search problem [1, p.105]. If we allow the grammar to have ϵ-Rules and loops then the number of possible partitions increase in number and more work is required to be done in order to find a parsing.

The algorithm in its naïve form is exponentional. It is inefficient as it can get stuck in infinite loops because of the presence of ϵ-Rules and loops in the grammar. It's also inefficient as there is an overgeneration of production rules that are useless and that it regenerates already processed rules. However, efficiency improves dramatically: from exponential to polynomial with optimizations like pre-computing which non-terminals can derive ϵ and by not solving subproblems more than once [1, p. 110].

2.1.2 SLL(1) parsers

SLL(1) stands for a *simple LL(1) grammar* or an *s-grammar*. An SLL(1) parser is a deterministic parser which parses SLL(1) grammars. In an SLL(1) grammar, the right hand sides of the production rules of every non-terminal start with a different terminal symbol [1, p.238-239]. An example of such a grammar would be as follows:

$$S \rightarrow aB$$

$$B \rightarrow b \mid aBb$$

In a non-determinsitic top-down approach (like Unger's), we need to make a prediction for the rest of the input which involves choosing one of the right hand sides of a production rule for a non-terminal. If there is a terminal symbol in front then we try to match the symbol in front with the input symbol and if there is a non-terminal symbol then we predict for the rest of the input. However, due to the nature of SLL(1) grammars, we are always guaranteed to have a terminal symbol in the front and thus our initial prediction can be quickly followed by a try at matching. So, whilst trying to pick which one of the RHS to follow or predict for

the rest of the input, all the parser needs to do is compare the input symbol to the terminal symbol of all the Right-Hand sides. As all of these terminal symbols are guaranteed to be different, there can only ever be 1 RHS that matches and we can easily make our prediction for the rest of the input to be the RHS with which the input symbol matched. We could enchance the efficiency of this method by precomputing the applicable right-hand sides for each non-terminal/terminal combination, and enter these in a table. Such a table is called a *parse table* [1, p. 236]. So, during parsing, we just need to look-up the non-terminal and terminal symbol in the parse table. Note that, because of the characteristics of an SLL(1) grammar each non-terminal/terminal combination will have only 1 entry in the table.

As during the parsing there is only 1 prediction, so no search is needed. Thus, this process is deterministic and therefore very efficient. So, SLL(1) grammars lead to very simple and very efficient parsers. However, not many practical grammars are SLL(1), although many can be transformed into SLL(1) form [1, p. 238-239].

2.1.3 LL(1) parser

LL(1) parsing method is a deterministic parsing method. It is deterministic in the sense that if a production/rule for a non-terminal has multiple options on its RHS then instead of exploring all possibilities, it looks ahead at the next symbol in the input string and makes a choice about which one of the options on the RHS it should explore."LL(1)" means that the grammar allows a determinsitic parser that scans input from Left to Right, produces a Left-most derivation, using a look-ahead of one symbol.

LL(1) is just a more general case of SLL(1). So, the idea behind parsing LL(1) grammars remains the same as it was for SLL(1). So, we want to first figure out for each RHS of a production rule for a non-terminal as to what all terminal symbols can start that particular RHS. Once, we have this information, a parse table similar to the one in the previous section can be constructed and used for parsing [1, p.239]. To do this, we first need to define the LL(1) grammar as the grammar itself paves the way for constructing the parse table to be used during parse time.

FIRST: Its a function which given a sequence of grammar symbols (eg. RHS of a production) returns the set of symbols with which strings derived from that sequence can begin.

A symbol c is in FIRST(A) iff $A \Rightarrow cB$ for some sequence B of grammar symbols

NULLABLE: Its a function which for a sequence of grammar symbols indicates whether or not that sequence can derive the empty string.

A sequence A of grammar symbols is Nullable iff $A \Rightarrow \epsilon$

Calcualtion of Nullable by case analysis is given as follows:

Nullable(ϵ) = True

Nullable(a) = False

Nullable(AB) = Nullable(A) $\wedge$ Nullable(B)

Nullable(N) = Nullable(A_1) $\vee$ Nullable(A_2) $\vee \cdots \vee$ Nullable(A_m), where the productions for N are $N \rightarrow A_1, \ldots, N \rightarrow A_m$

We can calculate FIRST in a similar fashion to NULLABLE as follows:

FIRST(ϵ) = ϕ

FIRST(a) = {a}

$$FIRST(AB) = \begin{cases} FIRST(A) \cup FIRST(B), if Nullable \\ FIRST(A), if not Nullable(A) \end{cases}$$

FIRST(N) = FIRST(A_1) $\cup \cdots \cup$ FIRST(A_m)

Here a is a terminal, A,B and A_i are sequences of grammar symbols, N is a non-terminal and ϵ is an empty sequence of grammar symbols [3, p. 70-71].

However, this is not enough to completely capture the full meaning of LL(1) grammars. We need to define the FOLLOW sets of non-terminals.

A symbol c is in FOLLOW(A) iff there is a derivation from the start symbol S of the grammar such that $S \Rightarrow ANaB$, where A and B are sequences of grammar symbols

Using these definitions, a CFG is LL(1) iff for every non-terminal N and every sequences

of symbols A, B such that $A \neq B$ and $N \rightarrow A|B$ we have

FIRST(A) $\cap$ FIRST(B) = ϕ

If $A \Rightarrow \epsilon$ then FIRST(B) $\cap$ FOLLOW(A) = ϕ

We can construct a LL(1) parsers using the FIRST and NULLABLE functions. This parser operates by starting with the start symbol S and replacing the non-terminal at hand with each of its RHS, computing the FIRST sets for each of the FIRST sets and checking whether the next input symbol is a member of any of these sets. If there is more than one prediction then the parser annoucnes that the grammar is not LL(1) and stops. Although this is a deterministic parser, it is far from ideal as it does not use a parse table and it does not check the LL(1) property of the grammar until parsing time [1, p.244-245]. But using the FOLLOW sets as well, we choose a production $N \rightarrow A$ iff $c \in FIRST(A)$ or $NULLABLE(c) and c \in FOLLOW(N)$. If we can always choose a production by these rules, this is calle LL(1) parsing [3, p. 80]. We do this by first producing the parse table and then using the same parsing approach used by SLL(1) parsers. The parse table is constructed by first computing FIRST sets of every non-terminal, then by computing FOLLOW sets of every non-terminal. This is then followed by starting with an empty parse table, we process each rule of the grammar and each RHS α is added to the entry $[A, a]$ for all terminal symbols a in FIRST(αFOLLOW(A)) [1, p.246].

The size of the table required is $O(mt)$ where t is the number of terminals and m is the number of non-terminals in the rules. Whilst parsing a string, it only does a single pass of the input string and for each symbol does a lookup in the parse table. Thus, time wise it takes $O(n)$ time where n is the length of the input string. Additionally, the choice for choosing between productions can be encoded either in a table or a recursive-descent program [3, p.87]. There are a lot of advantages of generating a recursive descent parser are numerous with one of the primary advnatge being that non-backtracking recursive descent parsers are often more efficient than table-drive ones. However, there is a big disadvantage of geenrating a recursive descent parser which is that they are quite large [1,p. 254].

2.1.4 LL(k) parser

LL(k) is another deterministic parsing method which is very similar to the LL(1) parsing method in the previous section. The only difference is that instead of juts looking at the next symbol of the input string to make a choice about the options on the RHS of a non-terminal, it instead looks ahead at most k symbols to make the choice.

LL(k) parsing is very similar to the LL(1) parsing. The only difference is that the LL(k) parse table is generated using FIRST$_k$ and FOLLOW$_k$ sets instead of FIRST and FOLLOWS sets. FIRST$_k$(A) is the set of terminal strings w such that $|w| < k$ and $A \Rightarrow w$ or $|w| = k$ and $A = wB$ where B is a non-terminal. For any non-terminal A, FOLLOW$_k$(A) is definied as the union of the sets FIRST$_k$(B) for any prediction AB [1, p.255]. So, when he parse table is generated, these sets are used while adding entries to the table instead of FIRST and FOLLOW.

LL(k) parsers are seldom used in practice, partly because the gain is marginal and partly because the parse tables can be huge [1, p. 256]. As the parse table now stores strings of length k instead of a single character, the size of the table will now be $O(mt^k)$ where t is the number of terminals and m is the number of non-terminals. So, the LL(k) algorithm grows exponentionally with k in terms of space complexity. The LL(k) algorithm examines k symbols at a time of the input string so the the input string is examined approximatley k times. However, k is not dependent on the input string and is a property of the parser and thus is a constant. So, the LL(k) parsing algoritm is still linear in the length of the input.

2.2 Bottom Up Parsing

A bottom-up parser constructs a parse tree by beginning at the leaves and progressing toward the root. This parse order corresponds to the reverse of a rightmost derivation [2, p. 197]. In essence, this method tries to roll back the production process and to reduce the symbol back to the start symbol. Thus, quite naturally this technique is called bottom-up [1, p.

66]. Bottom-down aprsing identifies the production rules in postfix order.

Let us consider the grammar G_1 defined in Section 2.1 and the same input string *aabbcc*. A bottom-up parser would thus work as follows:

1. As its first step, a bottom-up parser tries to figure out the final production rule from the input string. We recognize the right-hand side of the rule $bQc \rightarrow bbcc$ in the input string *aabbcc*. So, this gives us the final production rule.

2. Now, we find that Qc is derived by the rule $cQ \rightarrow Qc$ so now we have to figure out how $aabcQ$ was derived.

3. Again, from the current string the only RHS of a any rule that matches is of the rule $S \rightarrow abc$. So, now we need to figure out how aSQ was derived.

4. The only RHS from the rule that matches the string is $S \rightarrow aSQ$ so we end up with the start variable.

Now we will look at some of the different bottom-up parsing methods.

2.2.1 CYK Parsing Method

CYK Parsing Method, is a bottom up general non-direction parsing method named after J. Cocke, D.H. Younger, and T. Kasami, who independently discovered variations of this method. It is a non-direction method as it accesses the input in a seemingly arbitrary order. The CYK algorithm breaks parsing into two different phases. The first phase of the algorithm constructs a table telling us which non-terminal(s) derive which substrings of the sentence. This is called the recognition phase, as it tells us whether the input can be derived from the grammar or not. The second phase uses this recognition table and grammar to construct all possible derivations of the sentence [1, p.112]. Just like Unger's Parsing Method, CYK also requires the complete input to be loaded into memory before parsing can start. In essence, it tries to divide the input into recognizable segments which can then be assembled into the start symbol; if it succeeds, it has found a parsing [1, p. 134].

The CYK algorithm first concentrates on substrings of the input sentence, shortest substrings first, and then works its way up. So, the terminal symbols are recognized first and then rules of the form $A \rightarrow B$ are applied. On the next level, we calculate where the terminals are derived from using the results of the previous calculations for this as results for substrings below certain lengths are already available. On each higher level the length of the substring increases. Here a table R is used to store this information. Every entry of the form $R_{i,l}$ stores the non-terminal deriving the substring $s_{i,l}$ (substring of length l starting at position i in the input) A problem with this is that shortest substrings are the ϵ-substrings and they need to be recognized in arbitrary position. Another issue is the presence of unit rules, rules of the form $A \rightarrow B$ where A and B are nonterminals as multiple symbols can derive the same substring [1, p. 112-116].

Another problem is that the RHS of the rule could have arbitrarily many nonterminals. So, if a rule has m memebrs on the RHS then $m-1$ segment ends have to be found, each o them combining with all the previous ones. Finding a segment end costs $O(n)$ actions, since a list of proportional to the length of the input string needs to be scanned which gives us a time complexity of $O(n^{m-1})$. Since, there are $O(n^2)$ elements in R, the time complexity is increased to $O(n^{m+1})$. Thus if we could reduce the number of non-terminals on the RHS to 2, in addition to removing ϵ-Rules and loops then we could reduce the time complexity to $O(n^3)$. We know, this is done by grammars which are represented in Chmosky Normal Form (CNF). If CYK is used with a CNF grammar then it is very efficient and it takes $O(n^3)$ time. Although, the grammar must first be converted into CNF and then that conversion must be undone which could be challenging.

2.2.2 LR Methods

LR methods are parsing methods that scan the input string from left to right and construct a RightMost derivation [3, p.88]. *The RigthMost derivation is obtained by starting with the Start symbol S and expanding the rightmost nonterminal first.* LR parsing produces a RightMost

derivation but as it works from the bottom up, it reduces sequences of tokens on the left side first. LR parsers are table-driven parsers that use two kinds of actions involving the input stream and a stack:

Shift: A symbol is read from the input and pushed onto the stack.

Reduce: The top N elements of the stack hold symbols identical to the N symbols on the right-hand side of a specified production. These N symbols are by the reduce action replaced by the nonterminal at the left-hand side of the specified production [3, p. 88].

Hence, the LR parsers are also called Shift-Reduce parsers. But before we can move forward with the parsing method, we need to define a few terms in order to aid us in understanding the LR parsing method.

Sentential Forms: These are just sequences of terminals and non-terminals. Eg: aAb where a and b are terminals and A is a non-terminal.

Handle: Each step of a bottom-up parsers, working on a sentential form, identifies the latest rightmost production in it and undoes it by reducing the segment of the input to the non-terminal it derived from. The identified segment and the production rule are called the "handle" [1, p. 264].

So, to obtain an efficient parser, we need to find a n efficient method to identify handles, without considering alternative choices. So the handle search must either yield one handle, in which case it must be the proper one, or no handle, in which case an error must be generated [1, p. 264]. The LR method finds the handle by using a finite state automaton. It starts the automaton with the start rule of the grammar and only consider, in any position, RHS that could be derived from the start symbol. This is what reduces the cost of the parser to $O(n)$ [1, p.279]. The resulting automaton is started in its initial state at the left end of the sentential form and allowed to run to the right. It has the property that it stops at the right end of the handle segment and it tells us how to reduce the handle; if it ends in an error state then the sentential form was incorrect. Note that this accept state is just the accept state of the handle-finding automata and not the LR parser. The LR parser only accepts the input once the input has been completely reduced down to the start symbol [1, p.279]. Once, we have found the handle, it is reduced to the non-terminal which gives us a new sentential form which should again be scanned from left to right to find the next handle. However, since nothing has changed between the left end and the point of reduction, we can save some computation by remembering its states and storing them between the tokens on the stack. This leads to a standard setup for an LR parser. This is of the form

$$s_1 t_g s_g N_f s_f t_e s_e t_d s_d N_c s_c N_b s_b t_a s_a | t_1 t_2 t_3 \ldots \$$$

Here s_1 is the initial state and $s_e \ldots s_b$ are states from previous scans and s_a is the top deciding state. $N_f \ldots N_b$ are non-terminals whilst $t_g \ldots t_a$ are terminals that have been looked at and $t_1 \ldots$ are the terminals which correspond to the input that hasn't been looked at yet [1, p. 279]. Note that there is a dolloar sign at the right end of the input which is put there during the initialization of the parser. It is used for the normal termination of the parser [2, p. 212].

It is convenient to represent the automaton in a parse table. The LR parse table consists of 2 different tables - ACTION and GOTO table. ACTION table specifies most of what the parser does. It has state symbols as its row labels and the terminal symbols of the grammar as its column labels. Given a current parser state, which is represented by the state symbol on top of the parse stack, and the next symbol (token) of input, the parse table specifies what the parser should do, whether it should shift or reduce [2, p. 212]. The GOTO table specifies which state symbol should be pushed onto the parse stack once the reduction has been completed and the handle in the stack is replaced by the non-terminal. This table has state symbols are row labels and non-terminals as column labels [2, p. 212]. The parser finishes if the top of the stack is in accept state; in which case it accepts the input string or if it is an error state; in which case it produces an error.

Although there are many parsing algorithms based on the LR concept, they only differ in the construction of the parse table [2, p. 214]. Hence, they only actually differ

in the underlying handle-finding automaton as thats what is represented by the LR parse table. By far the most important component in an LR parser is the handle-finding automaton, and there are many methods to construct one. In the future sections, we will look at LR(0), LR(1), LR($k > 1$) and LALR(1).

One major advantage of LR parsers is that they can be built for all the programming languages. Another major advantage of LR parsers over LL parsers is that the calss of grammars parsable by LR parsers is a strict superset of the class of grammars parsable by LL parsers. But, LR parsers are very annoying and tedious to be written by hand and thus one needs to make use of parser generators to construct the parsers. A parser generator is a form of compiler-compiler that takes in as input a grammar of a programming language, and whose generated output is the source code of the parser which is often used in the compiler of that porgramming language. Although all the pasrser generator really needs to do is to geenrate the parser table [1, p. 211].

2.2.3 LR(0) parser

LR(0) parsing method which scans the input string from left to right and cosntructs a Right-Most derivation with a lookahead of 0 symbols.

LR parsing concerns itself with items. Each state in the handle-finding automaton is a set of items. LR(O) parsers use LR(0) items. 0 indictaes the number of symbols look ahead used while constructing the items. Each item is a production with a dot embedded in its RHS [1, p. 280]. So, if we had the production $A \to BaC$ then we could have the following items:

$$A \to \cdot BaC$$

$$A \to B \cdot aC$$

$$A \to Ba \cdot C$$

$$A \to BaC\cdot$$

Thus for LR(0), every rule of the form $A \to B$ has $|B| + 1$ items where B is a sentential form [1, p. 282]. Here a dotted production $A \to B_1 \cdot B_2$ effectively says we are looking for a handle B_1B_2 for the production and we have seen B_1 thus far. The closure of an item is used to see which production rules could be used to extend the current structure. It is calculated as follows:

1. Add the item itself to the closure
2. For any item in the closure $A \to \alpha \cdot \beta$ where the next symbol after the dot is a nonterminal, add the production rules of that symbol where the dot is before the first item.(Eg: for each rule $\beta \to \gamma$, add items $\beta \to \cdot\gamma$)
3. Repeat (2,3) for any new items added under (2).

So, given the grammar with the rules:

$$E' \to E$$

$$E \to T$$

$$T \to a$$

Then the closure of the production $E' \to \cdot E$ is given as follows:

$$E' \to \cdot E$$

$$E \to \cdot T$$

$$T \to \cdot a$$

Using items and closures, the DFA for finding the handle is built as follows [1, p. 280-283]:

1. Augment the grammar with a new rule $S' \to S$.
2. Build the node for State S_0. First an item for the rule created in Step 1 is introduced. Then a closure for this rule is built and the state is labelled with the closure of this rule.
3. For each unique symbol which follows a dot in the closure, a new state is created representing the recognition of the symbol. It is the linked to the original state with an arc labelled with the recognised symbol and a DFA node is added for this state.

4. Closure of each of the new states is calculated. It contains only those items from the previous state where the recognised symbol was the symbol after the dot. For each of those items, the dot is moved after the recongised symbols. Each state is then labelled with this closure. If a closure contains an item with a dot after a terminal then that is labelled as an accepting state.[5]

This automaton is then used to generate the ACTION and GOTO tables which are then used for parsing.

LR(0) immediately identifies the rule to be used for reduction and is a deterministic bottom up parser. However as gramamrs get bigger, the parse tbales contain more redundancy and empty space [1, p. 285]. Additionally, LR(0) Although this has a clever technique and is indeed an efficient algorithm, this is not as useful in practical applications as very few grammars are indeed LR(0) grammars [1, p.290].

2.2.4 LR(1) parser

The LR(1) parsing method is similar to the LR(0) parsing method but instead of a lookahead of 0 symbols it has a lookahead of 1 symbol.

The LR(1) handle-finding automaton is similar to the LR(0) automaton. In the LR(1) automaton construction, a look-ahead symbol is attached to each dotted item. So, the items in LR(1) parsers are of the form $E \rightarrow \cdot E\#$ where E is the non-terminal and # is the look-ahead symbol [1, p. 291]. The closures for each item is generated and the automaton is contructed the same way as it was for LR(0) parsers.

LR(1) parsers run in linear time [1, p.279] and can parse a bigger class of languages. It is more general than the LL parsers and as an added advantage it can detect syntactic errors as soon as it is to do so in a left-to-right scan of the input. The LR(1) parsers when developed in 1965, were considered to be impractical due to the large amount of space required by the deterministic automata. A modest grammar might already require hundreds of thousands or even millions of states, numbers that were incompatible with the coputer memories of those time [1, p. 300].

2.2.5 LR($k > 1$) parser

The LR(k) parsing method is similar to the LR(0) and LR(1) parsing method but has a lookahead of k symbols.

They are not simple extensions of the LR(1) parsing methods as we also need to compute the look-ahead sets for shift items. For each item I, two kinds of look-aheads are involved: the item look-ahead, the set of strings that can follow the end of I and the dot look-ahead, the set of strings that can follow the dot in I. The dot look-ahead of an item $A \rightarrow \alpha \cdot \beta\gamma$, where γ is the item look-ahead can be computed as $\text{FIRST}_k(\beta\gamma)$. The ACTION table is now indexed by look-ahead strings of length k instead of single tokens whilst the GOTO table is still indexed by a single token. Thus, ACTION and GOTO tables cannot be superimposed with each other. [1, p.298].

It is interesting to compare this to LR(0) and LR(1). In LR(0), the ACTION table offers no protection against impossible shifts and there must be a check upon shift; LR($k > 1$) needs the computation of dot look-ahead whilst LR(1) needs either of the two but not both [1,p. 299]. LR($k > 1$) has very limited practical value as its required tables can assume gragantuan size and it really doesn't help much. LR($k > 1$) are stronger than LR(1) but they can only handle some additional grammars. If a common-or-garden variety grammar is not LR(1), chances are minimal that it is LR(2) or higher [1, p. 299].

2.2.6 LALR(1) parser

LALR(1) stands for "Look Ahead LR(0) with a look-ahead of 1 symbol". It is a parsing technique that was developed to reduce the memory requirement of the deterministic automaton of the LR(1) parsers.

The goal of the LALR(1) automaton is to decrease the memory usage of the LR(1) automaton whilst preserving the look-ahead power. The idea is to collapse the LR(1) automaton by merging the states that have the same items regardless of their lookahead sets. When two states are merged, the lookahead

sets for matching items are merged. After this, once the parse table is completed, parsing follows the same process as the one described for LR parsing methods. One issue with this approach is that it requires to have the LR(1) automaton from the start which defeats the purpose of coming up with this approach [1, p. 302]. So, instead in practice its either done by constructing a LR(0) automaton or a SLR(1) automaton (Simple LR(1) not covered in this paper) by a maintaing a look-ahead of 1 symbol [1, p. 305,312].

This procedure preserves almsot all the original look-ahead power of the LR(1) parsers and still saves an enormous amount of memory. LALR(1) parsers are powerful, almost as powerful as the LR(1) parsers, they have fairly modest memory requirements only slightly inferior to those of LR(0) parsers and they are time-efficient like the other LR parsers as they also run in linear time. The most famous LALR(1) parser generators are yacc and its GNU version bison[1, p.301].

2.3 Left Corner Parsing

A left Corner parser is a type of chart parser which combines the top-down and bottom-up approaches of parsing. The name derives from the use of the left corner of the grammar's production rules. Left-corner parsing identifies production rules in infix order. It is a non-deterministic and a general parsing method.

Left Corner of a production rule is the 1st terminal on the RHS of that rule. So, if we have a rule of the form $S \rightarrow AB$ then A would be the left corner of this rule. This parsing method starts with the input string and tries to find the rules that geenrated it. However, when it encounters a non-terminal N, instead of following the bottom-up approach, it instead tries all the rules that have N as their left corner [1, p. 229].

It runs in $O(n^3)$ where n is the length of the input string [1, p. 232].

3 Conclusion

The linear time deterministics bottom-up parsing methods like the LR parsing methods are more powerful than the determinstic top-down parsing methods like the LL parsing methods as they accept a bigger class of languages. Moreover, the LALR(1) parser method for grammars defining programming languages is super-efficient in terms of time and space.

We have shown that there exist parsing methods that require linear time but are unable to handle all the the Context-Free grammars. In other words, they are parsing methods that are not general to the class of Context-Free Grammars. However, we have also shown that there exist parsing methods that are general to the class of Context-Free grammars but require cubic time at best. It is clear that for real speed we would like to have a linear-time parsing method. Unfortunately, no such method has been discovered to date, and although there is no proof that such a method could not exist, there are strong indications that this is the case [1, p. 81].

4 References

1. Grune, Dick, and Ceriel J.H. Jacobs. *Parsing Techniques: A Practical Guide.* 2nd Edition.

2. Sebesta, Robert W. *Concepts of Programming Languages.* 11th Edition.

3. Mogensen, Torben Ægidius. *Basics of Compiler Design.*

4. Tomita, Masaru. *Efficient parsing for natural language: a fast algorithm for practical systems.* Vol. 8. Springer Science & Business Media, 2013.

5. O'Donnell, Mick. Universidad Autnoma de Madrid. *http://arantxa.ii.uam.es/ modonnel/Compilers/LR0Summary.pdf*