# LEFT TO RIGHT-RIGHT MOST PARSING ALGORITHM WITH LOOKAHEAD

Jamil Ahmed

SemanticsWise Labs, Canada

## *ABSTRACT*

*Left to Right-Right Most (LR) parsing algorithm is a widely used algorithm of syntax analysis. It parses the input contingent to the information in its parsing table whereas the parsing tables are extracted from the grammar. The parsing table specifies the actions to be taken during parsing. It requires that the parsing table should have no action conflicts for the same input symbol. This requirement imposes a condition on the class of grammars over which the LR algorithms work. However, there are grammars for which the parsing tables hold action conflicts. In such cases, the algorithm needs a capability of scanning (looking-ahead) next input symbols ahead of the current input symbol.*

*In this paper, a 'Left to Right'-'Right Most' parsing algorithm with lookahead capability is introduced. The "look-ahead" capability in the LR parsing algorithm is the major contribution of this paper. The practicality of the proposed algorithm is substantiated by the parser implementation of the Context Free Grammar (CFG) of an already proposed programming language "State Controlled Object Oriented Programming" (SCOOP). SCOOP's Context Free Grammar has 125 productions and 192 item sets. This algorithm parses SCOOP while the grammar requires to 'look ahead' the input symbols due to action conflicts in its parsing table. Proposed LR parsing algorithm with lookahead capability can be viewed as an optimization of 'Simple Left to Right'-'Right Most' (SLR) parsing algorithm.*

## *KEYWORDS*

*Left to Right-Right Most (LR) Parsing, Syntax Analysis, Bottom-Up parsing algorithm*

## 1. INTRODUCTION

LR parsing is a well-known parsing method which employs a pushdown automaton to process the input string from left to right while constructing the corresponding syntax tree in bottom-up fashion. During this construction, non-determinism with regard to whether a sub-tree should be completed (shift/reduce conflict) and, if so, using which production (reduce/reduce conflict) can arise.

This algorithm resolves this non-determinism by giving the parsing automaton the capability of looking ahead. This capability allows inspecting the next input letter(s) without actually consuming them, which allows taking a deterministic parsing decision in many cases. This algorithm is named YALA i.e Yet another LR parsing Algorithm. This 'look ahead' capability is novel as compare to earlier work on LR parsing with 'look ahead' [4][5][6][7][8]. A standard LR parsing algorithm performs two basic operations or actions i.e. Shift and Reduce. A third action is Shift-Reduce which is a combination of the first two for the reason of optimization. This algorithm, in order to add the capability of looking ahead, introduces a new action called "LookAhead".

This algorithm is implemented in C#.NET programming language and parses the LR CFG of a newly proposed, java like, language SCOOP i.e. State Controlled Object Oriented Programming Language [2]. Its implementation is available at [3]. For its implementation, instead of relying on a parser generator tool, a parsing framework is 'hand written' to implement YALA. The auto parser generator tools use their own parsing algorithms.

The parsing framework includes a 'lexical analyzer' that breaks the input program into 'tokens' and then the 'tokens' are parsed by the parsing algorithm, YALA. Currently the framework does not compute the item set for a given LR CFG, therefore a manually designed item set of SCOOP is leveraged. The required information of manually designed item set and context free grammar are 'hard coded' in the two data structures of the parsing algorithm, TABLE 3. These two data structures are named 'parse_tab' (i.e. parsing table) and 'prod _tab' (i.e. productions table) respectively.

The item set is also available at [3], for the Context Free Grammar of SCOOP. It has 192 item sets. So far, the required information (i.e. production_table and parse_table) of the Context Free Grammar and its item set are hard coded in the parser implementation.

The rest of the paper is organized such as the parsing technique is illustrated with a Context Free Grammar in the subsequent section. Next the item set of its Context Free Grammar is presented and then the parsing algorithm is presented. Finally a sample program of SCOOP is given leading to conclude the paper.

## 2. PARSING TECHNIQUE

YALA is based on a simple technique to resolve the non-deterministic parse state. The technique is summarized in two rules as below.

1. In the non-deterministic parse state for the conflicting token, say token "t", the parser should peek at the next tokens to see if they can make up an acceptable sequence of tokens.

2. If it finds an acceptable sequence of tokens at a state, say state "s", then the pointer to the token set is back tracked to token "t", from where it started peeking. Now, at state "s", the parser will start parsing from that token "t".

## 3. AN EXAMPLE

### 3.1. Presentation

A shift-reduce/reduce conflict is illustrated in this simple example. The production number 1 in the CFG below will cause a shift-reduce/reduce conflict in the relevant state/item set. Such a conflict seems trivial which can be easily fixed by re-structuring the grammar rule. However, sometimes this kind of structure despite of its shift-reduce/reduce conflict is desirable or unavoidable in the Context Free Grammar of a practical programming language for the sake of more readable and intuitive representation of the Context Free Grammar. One such case is the production number 10 of the Context Free Grammar of SCOOP language [2]. Another similar case from the Context Free Grammar of Ada language is illustrated in [4] and it is advocated that such a structure should not be modified. Modifying the grammar takes effort, is subject to error and complicates its further processing. Therefore, grammar modifications could not be of practical convenience.

## 3.2. Sample Context Free Grammar

Table 1. Context Free Grammar

| Production number | Production Rule |
|---|---|
| 1 | <W> → w <Y> |
| 2 | <Y> → y |
| 3 | <Y> → <Z>z p |
| 4 | <Z> → y |

The item set of this CFG is given in the subsequent Section. It has 6 item sets. One possible acceptable token set is "w y z p". During parsing, at a parse state, i.e. state 2, after processing the first input symbol "w", the parser encounters a shift-reduce/reduce conflict. Now parser has two possible options as below:

Option 1: The parser would suggest a "reduce by production# 2" in order to accept the current token "y".

Option 2: The parser would also suggest a "shift_reduce by production # 3".

Therefore in such a parse state for the token "y", we will instead populate the parse table with "lookahead" action to goto another state, i.e. state 5. The purpose of this "lookahead" is to peek and check whether the next tokens are comprised of the token sequence "z p" which is a valid token sequence to be acceptable by production#3. Our "lookahead" action does not perform any modification to the parse stack.

In the state 5, if the scanned token is "z" then, in this case, we populate this state again for "lookahead" and goto another state, say state 6. The state 6 will again allow the parser to peek for the next token.

In state 6, if the scanned token is "p" then the parser knows that these peeked tokens can be accepted by production#3. Therefore, in this case, now we can safely populate state 6 with "shift_reduce by production#3" action (i.e Option 2), if the token is "p". If "p" is not found then the parser would suggest Option 1.

While peeking tokens due to "lookahead" action, as soon as an action other than "lookahead" is found then the pointer to the token set is backtracked by decrementing to the location, i.e. token index, from where "lookahead" started.

## 3.3. Item Set

Table 2 of item set, in its heading, uses abbreviations as below:

A : Action
G : Goto
P : Production number

Table 2 of item set uses following terminology:

- S represents Shift
- B represents Shift-Reduce

- L represents lookahead
- $$ is the END marker
- represents pointer to the next symbol
- Small letters e.g. w, y, z, p are terminals
- Capital letters enclosed in angle brackets are nonterminals
- α is the left hand side nonterminal of a production rule
- β is the right hand side of a production rule. It can be a combination of non-terminal and terminal

∑ = Set of all terminals and non-terminals
On Follow(x) represent a set of terminals following x
Few of the item sets are of the form as below

α->β LHS leads to α->β RHS

This kind of item sets is peculiar to this parsing technique. Only those items sets, e.g. item set i=5, have such a form if the parser has to transition to that item set i=5 due to "lookahead" in item set i=2. The $\alpha\text{->}\beta_{LHS}$ (i.e. <Z>→ • y) represents the production rule of◊ the previous item set i=2 from where parser had transited to this item set i=5. The A->BRHS represents the actual production rule of the item set i=5 on the bases of which the action for the item set i=5 is decided.

Table 2. Item Set

| **State** | **A** | **G** | **P** |
|---|---|---|---|
| **Item Set 1 or State 1** | | | |
| <W> →•w <Y> $$ | S | 2 | |
| **Item Set 2 or State 2** | | | |
| <W> → w •<Y> | B | | 1 |
| <Y>→ •y | L | 5 | |
| <Y> →•<Z> z p | S | 3 | |
| <Z> → • y | L | 5 | |
| **Item Set 3 or State 3** | | | |
| <Y> →<Z>• z p | | | |
| **Item Set 4 or State 4** | | | |
| <Y> →<Z> z •p | B | | 2 |
| **Item Set 5 or State 5** | | | |
| <Z> → • y leads to<br><Y> →<Z> •z p | L | 6 | |
| On Follow(<Z>)= ∑ - {z} | B | | 2 |
| **Item Set 6 or State 6** | | | |
| <Y> →<Z> •z p leads to<br><Y> →<Z> z•p | B | | 3 |
| On Follow(z)= ∑ - {p} | B | | 2 |

## 3.4. YALA - Yet Another LR Algorithm

YALA is intended to perform LR parsing with 'look ahead' capability. The presented algorithm, TABLE 3, is an optimization of SLR algorithm [1]. This algorithm is according to the same format as the SLR algorithm given in the book Programming Language Pragmatics [1]. The programming language independent pseudo code of the proposed algorithm in TABLE 3 uses the same terminology as used in the SLR algorithm [1]. YALA is implemented to parse the

programming language, SCOOP. As we know that an LR algorithm needs the information i.e. left hand side non-terminal of the each rule (i.e. production of CFG) and the length of its right hand side. This information is used to construct the item set in order to parse. This paper does not include the algorithm for the construction of item set. So far the item sets is manually constructed. The construction of item set for YALA involves a novel feature called “leads to”. The “leads to” feature is illustrated in the previous Section. This is also illustrated in the item set of SCOOP language [3].

In this parsing algorithm, TABLE III, the “action_record” is a data structure which keeps an action (e.g. shift, reduce, shift_reduce, lookahead) and production number in case of shift action or the new state number in case of reduce (shift-reduce) action. The algorithm given at [1] uses two separate variables in “action record”. One is used for the new state and the other is used for the production number. However, YALA combines both into one variable, since only one of those will be required at any given point in time. This gives a little optimization to YALA comparatively. YALA is given below.

Table 3. LR Parsing Algorithm with Lookahed

| |
|---|
| action_record = record<br>action: (shift, reduce, shift_reduce, lookahead, error) int : NewState_OR_ProductionNo |
| parse_tab : array[symbol,state] of action_rec |
| prod_tab: array[production] of record lhs : symbol<br>rhs_len : integer<br>--these two tables i.e. parse_tab & prod_tab are typically populated by parser generator tool |
| parse_stack : stack of record sym : symbol<br>st : state |
| parse_stack.push(<null,start_state>) |
| lookaheadcount := 0 |
| state_or_prod := 0 |
| cur_symbol : symbol := scan<br>--get new token from scanner |
| cur_state : state :=parse_stack.top.st<br>-- peek at state at top of stack |
| ar : action_rec := parse_tab[cur_state,cur_sym] |
| **Loop** |
| cur_state : state :=parse_stack.top.st<br>-- peek the state at top of stack |
| if cur_state = start_state and cur_sym = start_symbol |
| return --success |
| **case** ar.action |
| **shift:** |
| parse_stack.push(<cur_sym,ar.new_state>) |
| cur_sym := scan --get new token from scanner |
| ar : action_rec := parse_tab[cur_state,cur_sym] |
| **reduce:** |
| cur_sym := prod_tab[ar.prod].lhs |
| parse_stack.pop(prod_tab[ar.prod].rhs_len) |
| ar : action_rec := parse_tab[cur_state,cur_sym] |
| **shift_reduce:** |
| cur_sym := prod_tab[ar.prod].lhs |
| parse_stack.pop(prod_tab[ar.prod].rhs_len-1) |
| ar : action_rec := parse_tab[cur_state,cur_sym] |
| **lookahead:** |
| lookaheadcount := lookaheadcount+1 |

```
cur_sym := scan --get new token from scanner
state_or_prod := ar.prd_no_OR_nxt_state
ar:action_rec:=parse_tab[state_or_prod,cur_sym]
if (ar.action != action.lookahead)
lookaheadcount := 0
--backtrack token index to where peek begun
error:
parse_error
```

## 3.5. Parsing SCOOP by YALA

The SCOOP language definition can be found at [2]. SCOOP language allows explicitly defining the states of an object. A example source code of a program written in SCOOP language is given in TABLE 4.

Table 4. SCOOP Language Sample Program

| Line number | |
|---|---|
| 1 | [state(open,close)] |
| 2 | class File<br>{ |
| 3 | public int filepath; |
| 4 | Statebinding<br>{open:(filepath!=0)} |
| 5 | public int X() |
| 6 | { |
| 7 | read(a); |
| 8 | print a; |
| 9 | } |
| 10 | } |

This example program declares a ‘File’ object that can be in either ‘open’ state or in ‘close’ state during its lifetime. At line number 1, the term “state” is a keyword that is used to declare the possible states of the object. At line number 4, the term “statebinding” is a keyword that is used to declare the binding of a state with its variant. It means, the object can be assumed in that state as long as its variant is true. Few examples of writing programs using SCOOP can be found at [3].

The Context Free Grammar of SCOOP [3] has 125 productions and its item set [3] has 192 item sets. The production number 10 of the CFG [3] of the SCOOP language leads to item set number 13 that uses the “lookahead” action due to a conflict on the tokens “private” and “public”. Onward from item set number 13, two more item sets, i.e. item set 190 and item set 191 uses the “lookahead” to peek the next tokens.

The implementation of YALA is available at [3]. The implementation defines a “parser object” which has a “parse” function. The “parse” function implements YALA. The parse” function uses two important data structures i.e. a parse table, parse_tab, and a production table, prod_tab. Both of these data structures are created in the “parser object”. The “parser object” populates the “parse_tab” and “prod_tab” data structures. The “parse_tab” data structure will contain the item set whereas the “prod_tab” table will contain the required information about the productions or rules of the Context Free Grammar. Each element of “parse_tab” is of data type “action_record”.

## 4. CONCLUSIONS

YALA is an algorithm for LR parsing with lookahead capability. It is a modification to an SLR algorithm [1] and it is based on an intuitive technique which makes it easier to understand and implement. However, it increases the number of states/item sets as compare to the number of states/item sets of the SLR algorithm. As compare to the SLR algorithm, it will have one more state/item set for each symbol (token) to 'look ahead'. The technique to construct the item set for YALA is well defined and illustrated in this paper but the algorithm to construct its item set has not been explored. It is expected that the algorithm to construct the 'item set' for this algorithm will be investigated in the future.

## REFERENCES

[1] Michael L Scott: Programming Language Pragmatics, Chapter 2, Section 2.3.3, Third Edition, ELSEVIER.

[2] Jamil Ahmed: SCOOP, PhD Thesis, Section 5.3, Appendix A. University of Western Ontario, London, Canada (2014). https://cs.uwaterloo.ca/~smwatt/home/students/theses/JAhmed2014-phd.pdf https://ir.lib.uwo.ca/etd/2024

[3] Jamil Ahmed: The SCOOP Language, (2014). www.scooplang.com/scooplang

[4] T.P. Baker, "Extending lookahead for LR parsers" Journal of Computer and System Sciences, 22 (2) (1981), pp. 243-259

[5] F. DeRemer and T. Pennello, "Efficient computation of LALR(1) look-ahead sets". ACM Transactions on Programming Languages and Systems, 4(4):615–649, (1982)

[6] A. Aho, R. Sethi, J. Ullman, "Compilers: Principles, Techniques, and Tools" Reading, MA: AddisonWesley, (1986)

[7] K. Čulik, R. Cohen, "LR-regular grammars—an Extension of LR(k) Grammars", Journal of Computer and System Sciences 7 (1) (1973) 66–96

[8] Bermudez, M. E. and K. M. Schimpf, "Practical arbitrary lookahead LR parsing", Journal of Computer and System Sciences 41 (1990), pp. 230–250

## AUTHORS

**Jamil Ahmed** is a PhD (2014) in Computer Science from Western University, London, ON, Canada. His PhD thesis introduced an Object Oriented Programming Language with specifications of "state" of an object. It is called "State Controlled Object Oriented Programming" (SCOOP). His interests include compilers as well as software & cyber security. Currently he is Director R&D at SemanticsWise Labs, Canada.