Multithreaded Implementation of Earley Style Parsing Algorithm for F-LTAG

MULTITHREADED IMPLEMENTATION OF EARLEY STYLE PARSING ALGORITHM FOR FOR F-LTAG

Ramchandra P. Bhavsar*, AkshayDesale*, B. V. Pawar*,
Associate Professor Software Developer Professor
*{rpbhavsar,addesale,bvpawar}@nmu.ac.in
School of Computer Sciences,
North Maharashtra University,
Jalgaon(MS) 425001

Abstract

Lexicalized Tree Adjoining Grammar (LTAG) is a leading formalism in Generative Enterprise. Out of different approaches for parsing LTAG, Earley Style LTAG parsing as proposed by Joshi & Schabes is considered as favorite and popular approach for parsing LTAG. Practical implementations of this parsing algorithm have been used in LTAG based practical NL systems such as XTAG, MANTRA-MT platform and ANUVADAKSHTAG MT engine. In order to leverage the benefits of multithreaded processor architecture, multithreaded version of the algorithm along with optimizations and other issues incurred during actual implementation of algorithm for F-LTAG(Feature based LTAG) parsing have been described in the present article. Our article has been divided into four parts, Part-I describe the parsing process and FLTAGin simple words along with gist of Earley style parsing, while Part-II presents the insights on Recognizer routine along with proposed multithreaded implementation philosophy. Optimizations & their impact on time & space complexity have been discussed in Part-III followed by concluding remarks.

Keywords: FLTAG, Earley Style LTAG parser, multithreaded parser implementation.

Part-I

1. Introduction

Tree Adjoining Grammar (TAG) proposed by Prof. Arvind Joshi[5][6] is considered to be leading grammar formalism in Generative Enterprise[1]. The major strength of this formalism lies in its ability to perform constituency analysis due to its' two operations viz. *Adjunction* and *Substitution*. In LTAG grammar, lexical categories are represented using tree data structure. The elementary LTAG trees are divided in two classes viz. Initial (alpha) tree& Beta tree. Every tree is instantiated by lexicalizing it with lexical item (word) at special node in elementary called anchor. Beta tree get adjuncted on elementary trees (alpha or beta). Sentence analysis in LTAG involves combining the tree structures associated with lexical items(words) either by Adjunction or Substitution operations, in a special tree called sentential tree(generally tree anchored at verb). These operations are performed on Nodes of elementary trees which are marked with special constraints for adjunction and substitution. After combining all lexical items the resulting hierarchical structure results into complete tree for sentence, which is also called as 'parse tree' or 'derived tree' for the sentence. The yield of this tree i.e. leaf nodes(words) in left to right order represent the analyzed sentence. The parsing process actually is responsible for combining constituent tree structures into parse tree(s) using some trivial procedure. The Parsing algorithm formally defines this procedure for checking membership of given sentence to given grammar and presents the proof of this membership in the form of derivation tree & derived tree. A derivation tree records the sequence of adjunction and substitution operations amongst various participating elementary trees, in the context of sentential (main) tree. The LTAG parsing process is achieved through 'dot traversal', which defines order ofvisiting each node in the elementary tree. Dot represents the progress of parsing at node level, foran elementary tree under parsing. Considering the structure of an elementary tree, a node can have four possible dot positions viz. *la*(left above),*ra*(right above), *lb*(left below), *rb*(right below). Fig.1 below depicts the dot traversal.

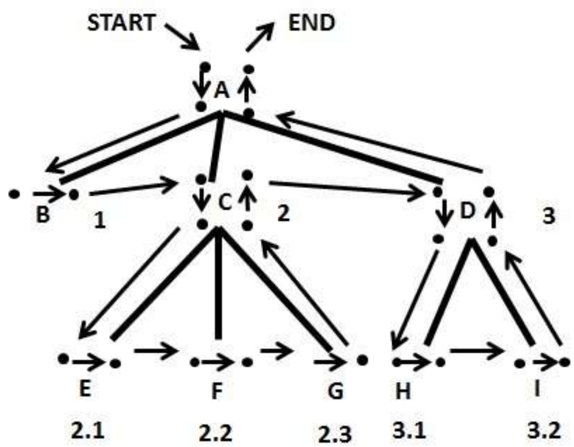
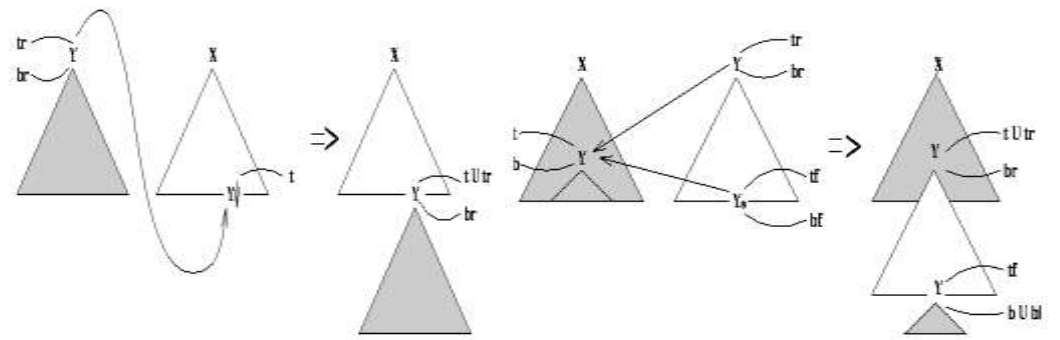


Fig. 1 dot traversal.

From fig. 1 it can be noted that the dot traverses through all four positions for intermediate nodes while for leaf node(except Foot node of beta tree), we visit only top dot positions i.e.*la* and *ra*.

Feature based LTAG(F-LTAG) is special version of LTAG in which feature structures are defined on each node of LTAG tree and feature unification takes place during completion (also referred as recognition in literature) of *adjunction* and *substitution* operations[4]. Two feature structures (top & bottom) are associated with each node. Feature unification process guarantees basic feature(syntactic and semantic) agreement between sentence constituent viz. subject-verb, object-verb, adjective-noun, aux verb-verb etc. This type of checking helps improves the correctness of output as it not checked merely on POS level. The feature unification process is encoded during parsing process while performing *adjunction* and *substitution* operations.Fig 2depicts the *adjunction* and *substitution* operations in the context of feature unification.



(*Substitution in FLTAG*) (*Adjunction in FLTAG*)

Fig. 2 Substitution & Adjunction in FLTAG

Feature unification process checks compatibility of two features structures. Compatibility is measured on equality of values for similar features in two feature structures, while non-similar features are simply included (united) in resultant feature structure.

2. *Gist of Earley style parsing*

J. Earley invented the Earley Parsing algorithm for Context Free Grammar (CFG) formalism[2]. His algorithm defines three main operations namely *Predictor*, *Scanner* and *Completer*. The algorithm uses dot symbol in the right hand side (RHS) string of CFG production for marking the parsing progress. The *Predictor* gives fair chance to all possible CFG productions, matching selection criteria(mostly syntactic) for substitution, while *Scanner* operation matches terminal symbols(after dot) in current state with current symbol in input string. On matching Scanner recognizes the input symbol and moves to next symbol in input string. *Completer* recognizes complete CFG production upon visiting last symbol in RHS of production. This parsing process involving these three operations was then popularly known as Earley style parsing.

Joshi & Schabes proposed Earley style parsing algorithm for parsing LTAG[3].However due to anatomy of LTAG in terms of tree structure representation of lexical categories and two operations viz. *Substitution* and *Adjunction*, in contrast to single operation(substitution) and linear representation of grammar rule(production) in CFG respectively, their algorithm defines three operations in Earley's using 09 operations. The detail algorithm has been described in following section.

Part-II

Earley style LTAG parsing algorithm defines 09 primitive operations along with recognizer routine. The *recognizer* routine of the algorithm drives the parsing process by invoking these operations based on dot position in the state under processing. The algorithm stores the parsing progress using *State Chart* data structure which contains state sets which in turn stores states. A state tuple represents configuration (node in tree, position of dot, return address etc.) of a tree under operation at particular time instance of parsing process. More than one state may get added to state chart as all operations defined at that dot position are tried, out of these one or more(in case of structural ambiguity) may lead to success in entire parsing while others can't proceed and get halted. The formal description of the Recognizer routine is given in below.

Following Fig. 3 summarizes primitive operations defined in the Recognizer routine above

Operation	Node Dot Position	Node Constraint/Property	Operational Details
Substitution Predictor		Substitution	*Case 1:* Predict all the possible substitutions andadd statesconfiguration to the *StateChart* *Case 2:* *Substituion node:* If no trees are available for substitution, report failure. *No beta Trees:* If on intermediate/foot node simply move dot to 'lb' of this node and add state configuration to the *StateChart*
Left Predictor	LeftAbove	Non leaf node or Foot Node	*Case 1:* predict all the possible adjunctions andadd thestates'configurationn to the *StateChart,* *Case 2:* If not beta trees available for adjunction simply move dot to the 'lb' of this node and add this configuration to the *StateChart*
Scanner		Node must be terminal symbol(word) or epsilon node(empy node)	Match the current symbol in input string symbol with the node symbol(word), if they match move dot to 'ra' of terminal node and this configuration to the *StateChart,* move the pointer to next word in input string for epsilon node simply move dot to ra' of epsilon node and add this state configuration to the *StateChart*
Left Completer	LeftBelow	FootNode of beta tree	Jumps the predicting trees node's '*lb*' position(which predicted this beta tree) and add the state denoting that configuration
Move Down		Intermediate node	Move the dot to '*la*' position of first left child and add the state for this configuration
		RootNode of Initial(alpha) tree	Complete the substitution operation on tree which predicted this tree for substitution, move dot to 'ra' of initial tree node marked with substitution constraint, on successful feature unification.
Substitution completer		State has Substitution flag ON	
		no. of tokens recognized = no. of tokens available && dot is on initial tree	Show the successful parsing output i.e. derivation tree(s) produced
	RightAbove	no. of tokens recognized != no. of tokens available	Report failure

Right Completer		Root node of beta tree	Jump back to 'ra' node of the tree which predicted this beta tree, on successful feature unification , add state configuration of predicting tree to the *StateChart,*
Move Up		Intermediate Node	Simply dot to 'rb' of parent node of this node and add this state configuration to the *StateChart*
Right Predictor	RightBelow	Intermediate Node of alpha or beta tree	*Case 1* If Adjunction was defined on 'la' position of this node and we are here beause of Left Predictor,jump to back to the 'rb' of beta tree which was predicted on this node and add this state configuration to the *StateChart,* *Case 2* If Adjunction was defined on 'la' position of this node simply move the dot to the 'ra' position of this node and add this state configuration to the *StateChart*

Fig. 3Earley Style LTAG parsing algorithm operations

The Recognizer

Let G be a TAG.

Let $a_1 \ldots a_n$ be the input string.

Program recognizer

Begin

*S_0 = **{[** α, 0, left, above, 0, -, -, -, -, -**]***

*| α is an initial tree **}***

For i := 0 to n do

Begin

Process the states of S_I , performing one of the following seven operations on each state

s = [α, dot, side, pos, l, fl, fr, star, t_l, b_l*]*

until no more states can be added

1. *Scanner*
2. *Move dot down*
3. *Move dot up*
4. *Left predictor*
5. *Left completor*
6. *Right predictor*
7. *Right completor*
8. *Substitution predictor*
9. *Substitution completor*

If S_{i+1} is empty and I < n, return rejection.

End

If there is in Sn a state

*S = **[** α, o, right, above, o, -, -, -, -, -**]***

Such that α is an initial tree

Then return acceptance.

End.

3. Insights on Recognizer:

The recognizer algorithm described above reveals that the states in the *StateChart* are processed in linear order & multiple states may be added to the *StateChart* during processing of each state in the *StateChart*. Addition of multiple states hint forking of multiple paths in the parsing out of these paths one or more may lead to success or all failure. The process of forking multiple parsing paths can go up to any depth resulting in parse forest. The recognizer halts whenever all such paths are tried.The linear state processing order of the *StatChart* defeats the parallelism achieved due to ability of applying multiple operations at the dot position. The theoretical time complexity of the algorithm has been reported as $O(|G|^2 n^9)$, while space complexity to be $O(|G| n^6)$in [3] .

From the above discussion it is apparent that the algorithm is very compute intensive and as its' implementation would make it impractical to use for web based applications, which are tightly constrained by the response time factor (time out period), in order to overcome this problem, we decided to adopt multithreading paradigm for our implementation.The details of our multithreading paradigm are explained in following section.

4. Multithreaded implementation

If the problem at hand is large and can be divided into comparatively easy subparts which can be performed in parallel fashion, it's a good idea to use multithreading. Multithreading support is available in modern programming languages as well as present day processors. Multithreading promotes parallel execution. In multithreaded applications, the execution starts with a main thread and the main thread can recursively spawn new threads. This execution model is lucrative option for the problems in which subtasks can be carried in parallel fashion. These parallel threads execute independently in their own sealed space however the main programs data space can be shared amongst executing threads. Our deep study of the recognizer algorithm hinted to apply multithreading during state processing operations i.e. whenever we are processing a state, we primarily examine the dot position on the node of processing tree and depending on the dot position recognizer suggests addition of one or more states to the State Chart, it is this point where we can spawn a new thread for each new state and leading to parallel execution. Each of the newly spawned thread needs resources, for further carrying the parsing process ahead. Considering these resource requirements, we have divided the total resources in two classes viz. mutable resources & immutable resources. Mutable resources are those data structures which drive the parsing process while immutable ones aid the parsing process. The specific mutable resources are intermediate derivation strings, *State Chart*, newly added state and Feature Vector. Immutable resources include Tree Vector and Token vector. Whenever a new thread is spawned, the mutable resources from the parent thread are cloned and passed to child thread. The immutable resources work as global resources, which are shared amongst all executing threads avoiding duplication of resources. The cloning of mutable resources may seem to be overhead at first sight but the efficiency achieved through this parallelism outweighs this duplication. Also this approach controls the size of *State Chart* as it splits at each spawning operation. Technological advances in handling of multithreading implementation further helps us to reduce the thread instances through thread pooling. The following block diagram (**Fig.4**) shows the schemata for multithreaded implementation.

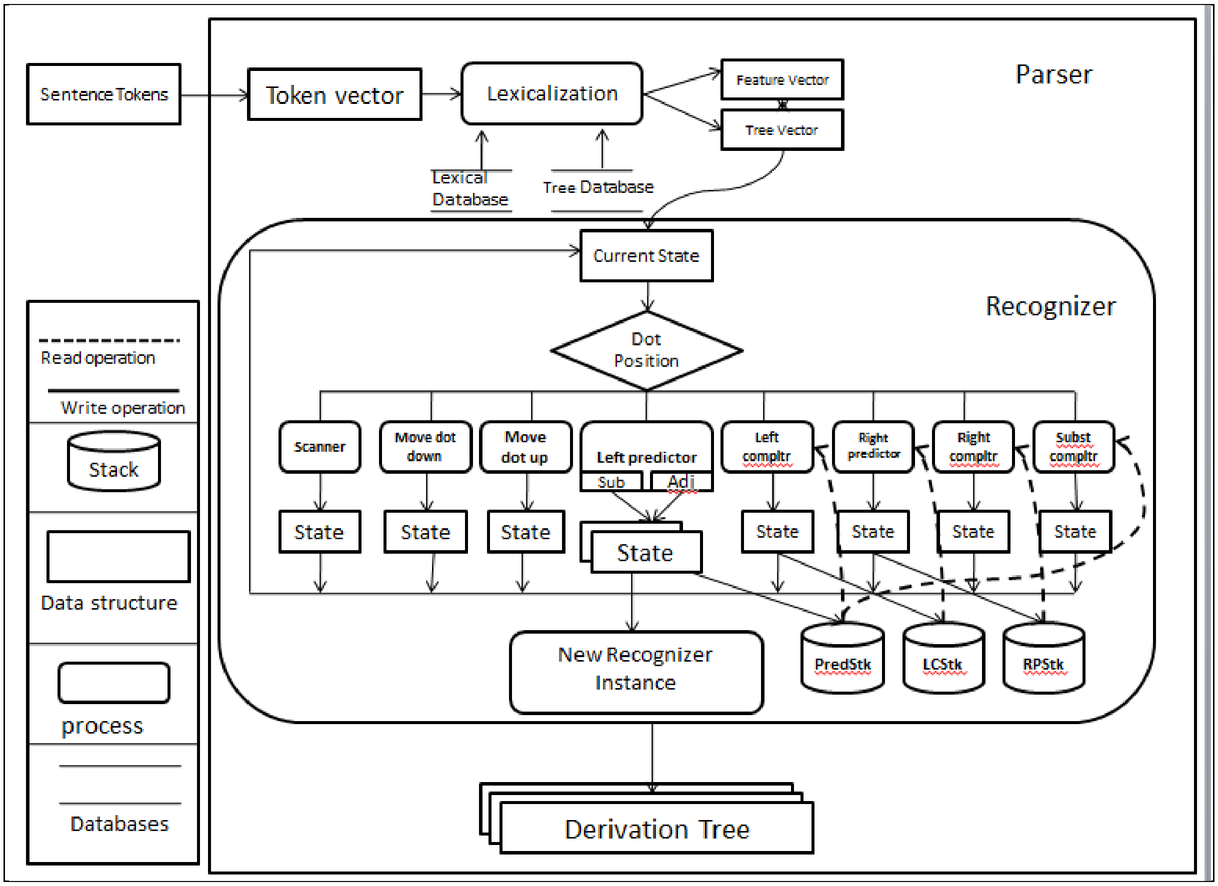


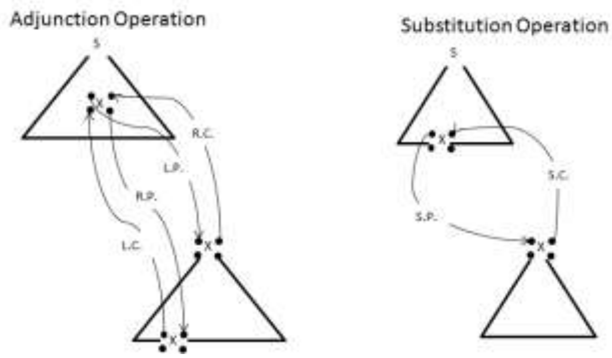
Fig.4 Multithreaded Implementation using stacks

Part-III

5. Optimization in Implementation

5.1 Removal of State Chart

The LTAG parsing process in nutshell can be described as jumping *to-and-fro* amongst the LTAG trees in Tree Vector. Although the parsing algorithm stores all states generated in the process but in the practical sense only the ones generated as consequence of tree jumps are relevant from future point of view. This observation in turn directs us in a new and innovative idea of storing only those states and that too using data structure which will give the required data ie. *state* in single lookup instead of searching linearly into the *State Chart*. This will also help to reduce the State data structure's size from 11 tuples to 5 tuples as other state variables are used for tacking the jumping addresses. The following diagram (Fig. 5) shows the jumping operations.



Operation	Tree Jump
Subst. Predictor (S.P.)	Predicting (Initial) Tree -> Predicted (Auxiliary) Tree
Subst. Completer (S.C.)	Predicted (Auxiliary) Tree -> Predicting (Initial) Tree
Left Predictor (L.P.)	Predicting (Initial) Tree -> Predicted (Auxiliary) Tree
Left Completer (L.C.)	Predicted (Auxiliary) Tree -> Predicting (Initial) Tree
Right Predictor (R.P.)	Predicting (Initial) Tree -> Predicted (Initial) Tree
Right Completer (R.C.)	Predicted (Initial) Tree -> Predicting (Initial) Tree

Fig5. Tree Jump Operations

Due to inherent non-overlapping nature of tree jumps, the states required for backward jump(from predicted tree to predicting tree) are stored in last in first out order hence, this behavior can be computationally captured using Stack data structure. Considering the tree jumping operations *Left Predictor*, *Left Completer*, *Right Predictor*, three stacks viz. predStack, lcStack and rpStack were introduced for storing return states, thus eliminating the need for *State Chart*.

5.2 Stack based implementation explanation:

1. 3 stacks each of size *n*, where *n*is no. of tokens in the input string are allocated.
2. States are pushed on respective stack, while left prediction, left completion and right prediction operations.
3. For other operations, the current state is replaced by new one for further processing.
4. The states are popped from predStack, lcStack and rpStack when the operation in progression right predictor, left completer and right completer respectively.

Part-IV

6. Benefits of new implementation

6.1 Space Complexity

c

6.2 Lookup Time

The worst case lookup time for State Chart and Hash Map is $O(n)$ in case of n token sentence, whereas for stack lookup time is precisely $\Theta(1)$, which results in faster execution.

7. Summary

In this paper, we have described the F-LTAG Parsing using Earley style parsing algorithm as proposed by Joshi & Schabes along with FLTAG and gist of Earley style parsing. After deep study of this parser, we got the idea of adopting multithreading paradigm for the proposed practical implementation of this parser using JAVA for ANUDKSH's TAG MT Engine. We have described the details of our multithreaded implementation along with its advantages. Further have presented important optimization of replacing the *StateChart* data structure which is key data structure, with stack and still guarantee the correctness of algorithm. Space requirements and time complexity have also been discussed and it is proved that use of stack reduces the space requirement and lookup time hence the parser built using this approach is more efficient and it is suitable to use for Internet based MT application like ANUVADKSH.

References:

1. Akshar Bharati, Vineet Chaitanya and Rajeev Sangal. 1995. Natural Language Processing: A Paninian Perspective, Prentice-Hall of India, New Delhi.
2. Earley, J., 1970. An Efficient Context-Free Parsing Algorithm. *Commun. ACM* 13(2):94-102.
3. Schabes, Yves and Joahi, Aravind K., 1988. *An Earley. type Parser for Tree Adjoining Grammars. In the proceedings of 26th meeting of Association of Computational Linguistics, Buffalo, June 88.*
4. XTAG report, The XTAG Research Group, Institute for Research in Cognitive Science University of Pennsylvania.
5. Joshi, A. K., L. S. Levy, and M. Takahashi. 1975. Tree Adjunct Grammars Journal of Computer and Systems Sciences, 10(1), 55-75.
6. Joshi, A. K. and Y. Schabes. 1997. Tree-adjoining grammars, InG. Rosenberg and A. Salomaa(eds.),Handbook of Formal Languages, Vol. 3, 69-123, Springer-Verlag, New York,

✡✡✡