Probe on Syntax Analyzer

Farhanaaz[1], Sanju. V[2] and **M. Vinayaka Murthy[3]**

Submitted in April, 2016; Accepted in July, 2016

Abstract – Syntax analysis is the second phase of the compiler. Checks the syntactical structure of the programming language due to the limitation of the regular expression. Grammar is used to describe the syntax or rules of the source language. Parser is a tool used to check the syntactical structure and parsers are grammar specific. Though the significant work has be done on the parallel compilation process but still the parsing area is difficult to implement parallel on multi-core machines.

Index Terms – Parallel Compilation, parallel Syntax Analysis, Context Free Grammar, Top Down Parsing and Bottom up Parsing.

NOMENCLATURE

NOC - Network On Chip, CFG − Context Free Grammars, CFL − Context Free Language

1.0 INTRODUCTION

Grammar defines the syntactic structure of a programming language. Each grammar defines a unique programming language. Change in the grammar will result in programming language. Roles and responsibilities of the syntax analyzer [1][2][3][4] are :

(i) To take token as a input from lexical analyzer
(ii) To check if tokens could be generated from the specified grammar of the programming language.
(iii) To report syntactical errors in the program if any.
(iv) To construct parse tree.

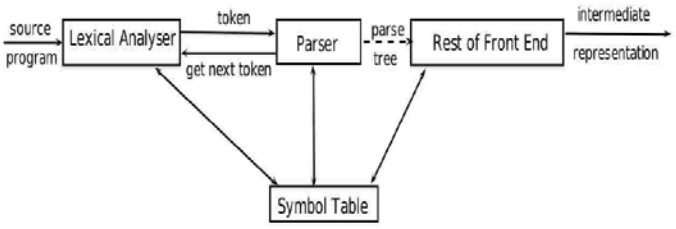

Figure 1: Position of a Parser in Compiler

[1] Assistant Professor, School of Computer Science and Applications, REVA University, Bangalore, India, Email: farhanaaz3@gmail.com
[2]Associate Professor & Head, Department of CSE, Muthoot Institute of Technology and Science, Ernakulam, India, Email: sanjuv21@gmail.com
[3]Professor & Assistant Director, Research and Innovation, REVA University, Bangalore, India, Email: vinayakamurthy@revainstitution.org

2.0 GRAMMAR

A grammar is the powerful tool for describing the language. Grammars are language generators. Noam Chomsky gave the mathematical model for the grammars in 1956.

Though it can't describe natural languages but it is very useful to describe computer languages. There are different types of grammar.

(i) Type 0: unrestricted grammar include all formal grammars. The languages generated by this grammar is known as recursively enumerable languages.
(ii) Type 1: Context Sensitive Grammar generate Context Sensitive Languages which is recognized by Non Deterministic Turing Machine.
(iii) Type 2: Context Free Grammar generate Context Free Languages which are recognized by Non Deterministic Pushdown Automaton.
(iv) Type 3: Regular Grammar generate regular languages and it is recognized by Finite State Automaton.

Out of these Context Free Grammars are used in syntax analyzer to define a structure of a language.

Class	Grammar	Languages	Automaton / Machine
Type 0	Unrestricted	Recursively Enumerable	Turing Machine
Type 1	Context Sensitive	Context Sensitive	Linear Bound
Type 2	Context Free	Context Free	Push Down Automata
Type 3	Regular	Regular	Finite

Table 1: Types of Languages and their Acceptable Machines

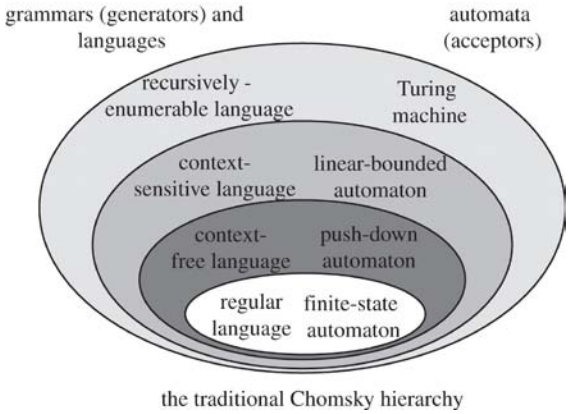

Figure 2: Chomsky Hierarchy

2.1 Context Free Grammar

Formally Context Free Grammar is define by G = (N, T, P, S)
N : Finite set of Non Terminals; generally represented by Upper case alphabets.
T : A finite set of Terminals represented by Lower case alphabets.
S : Starting Non Terminal symbol of the grammar. S ∈ N
P : Set of rules or productions in Context Free Grammar, each of the form D→ α where D ∈ N and α ∈ (N ∪T)*. First production always indicates the start symbol of the grammar.
Below is an example of CFG and the derivation of the string from the given productions.
S→0S0/1S1/0/1/ε.
Let us derive a string 1001001. Let us start with the appropriate production
$S \rightarrow 1S1 \rightarrow^{S \rightarrow 0S0} 10S01 \rightarrow^{S \rightarrow 0S0} 100S001 \rightarrow^{S \rightarrow 1} 1001001$.
Superscript production indicates the application of that production in next step. CFG generate CFL. The grammar generated by G is represented by L(G). L(G) = { w | w∈ T*, and S⇒* w}.

2.2 Parse Tree

Parse Tree is the pictorial representation of a derivation. A parse tree is an ordered tree in which nodes are labeled with the left side of the production and in which children of the nodes represents its corresponding right side. Except root and leaf nodes of the tree others are all non terminals therefore productions are applied to replace the non terminals with the RHS of the production and the leaf nodes are all terminals. Formally Parse tree is defined as, If grammar G is the CFG then G = (N, T, P, S). If G is the derivation
tree if and only if
(i) The root is the Start symbol.
(ii) Internal nodes are Non terminals from N.
(iii) Leaf nodes are Terminals from T.
(iv) If leaf node is # then it has no siblings.
Yield of the parse tree is the list of labels of all the leaf nodes from left to right. If α is the yield of derivation tree for grammar G , then S ⇒* α

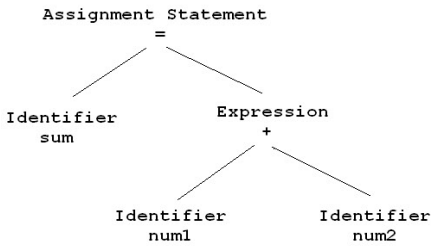

Figure 3: Parse Tree

2.3 Left and Right Linear Grammar

If all the productions in the CFG are in the form A→ Bw / w then it is known as left linear Grammar. If the productions are of the type A → wB / w then it is a right linear grammar. A and B are variable and w ∈ T*.

Left most and Right most derivation

To restrict the number of choices while deriving a string we opt for left most and right most derivation. A derivation is said to be left most iff the left most non terminal is replaced by the appropriate production till the string is formed. Likewise in the right most derivation the right most non-terminal is replaced with the appropriate production. Left most and right most derivations can be derived for the string aabbaa and the grammar is S→ aDS / a and A→ SbD / SS / ba. Left most Derivation : S→ aDS → aSbDS → aabDS → aabbaS → aabbaa. Right most Derivation : S→ aSD → aDa → aSbDa → aSbbaa → aabbaa.

2.4 Issues in writing a Context Free Grammar for programming language

1) Elimination of Ambiguous grammar: A grammar is ambiguous, if for at least one string in the language, grammar produces more than one parse tree. Derive a^3 using grammar S→ aS / Sa / a Sometimes ambiguity can be eliminated by rewriting the grammar. For the simplicity purpose we can restrict the format of the Context Free grammar without reducing the language generation power. Let L be a non-empty CFL, then CFL can be generated by a CFG G with the following properties :
(i) Elimination of Useless Symbols : variables or terminals that do not appear in any derivation of a terminal string from start symbol.
(ii)Elimination of ε productions : If production of the form D→ε for some variable D.
(iii) Elimination of unit productions : If productions of the form D→E for variables D and E.

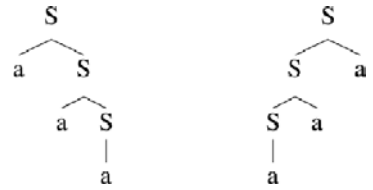

Figure 4: Different Parse Tree for a^3

2) Elimination of left Recursion: Recursive non terminals are very useful which allows grammar to describe infinite number of input but left recursive grammars couldn't be handled by top down parsing techniques. A grammar contains a production of the form A → Aα , where A is non terminal; then this production can be replaced by a non left recursive production of the form A → βB and B→ α , B→ε, without changing the strings derivable from A. This grammar is of the type Left recursive. This procedures remove left recursion from A to B generating same language as A. This procedure does not eliminate left recursion involving derivations of more than 2 steps. Above procedure can be extended to n number of variables on left hand side of the production.
3) Elimination of Left Factoring: Left factoring is the powerful tool in generating grammar which is accepted by predictive parsers or top down parser. Suppose we have the production of the form $A \rightarrow \alpha\beta_1 / \alpha\beta_2$, then on seeing the

input α in the production A it is not clear which production is of right choice, this can be rewritten in the form $A \rightarrow \alpha B$ and $B \rightarrow \beta_1 / \beta_2$.

3.0 PUSHDOWN AUTOMATON

Finite Automaton cannot store / remember anything. Therefore Finite Automaton can be extended by adding auxiliary storage to accept CFG. Push down automaton is a finite automaton with control of both an input tape and a stack to store.

Formally, PDA is defined as a Finite Automaton P = ($Q, \Sigma, \Gamma, S, F, \delta$)

Q : Non empty finite set of states

Σ : Non empty finite set of input symbols

Γ : Stack alphabet

S : Initial State, $S \in Q$

F : Non empty finite set of Final State/(s) and $F \subseteq Q$

δ : Transition Function which maps to $(Q \times \Sigma^* \times \Gamma^*) \rightarrow (Q \times \Gamma^*)$.

Moves in Push Down Automaton

(i) $\delta(q, a, z) \rightarrow (p, y)$: If PDA is in the state q, with z as the top of the stack and with a on the input tape then PDA replaces z by y on top of the stack and enters state p.

(ii) $\delta(q, a, \varepsilon) \rightarrow (p, a)$: Push a on to the stack.

(iii) $\delta(q, a, z) \rightarrow (p, \varepsilon)$: Pop element from stack.

(iv) $\delta(p, a, z) \rightarrow (p, z)$: PDA does nothing.

Push Down Automata can recognize languages for which there exist Context Free Grammar.

4.0 PARSER

Parser is the program for parsing. Parsing is the technique which it produces an output as a parse tree for the input string w. An error message will be indicated if w is not a valid for the given grammar, otherwise parse tree is generated. Parsing is classified based on the rules implemented to arrive at the solution. Following are the types

4.1 Top Down Parsing

In a top down approach, a parser starts constructing a parse tree from the top node called root node and it completes the parse tree in pre order fashion for the given input string. Top down parsing holds the technique form leftmost derivation for an input string. The types of top down parsing is depicted in Figure6.

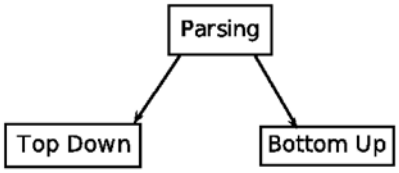

Figure 5: Types of parsing.

1) Recursive-descent parsing: This is one of simplest form of top down approach. The program consist of a set of procedures, one for each non terminal of the grammar. Execution begin with the process for the start symbol, which stops and announces hit if its procedure scans the entire input string. Following is the procedure for the non terminal A in the grammar.

```
void A(){
Choose an A production, A→ X1 ,X2 . . . Xk ;
for( i = 1 to k){
if(Xi is a non terminal)
call procedure Xi ();
else if(Xi equals the current input symbol a)
advance the input to the next symbol; else
Error occurred; }
}
```

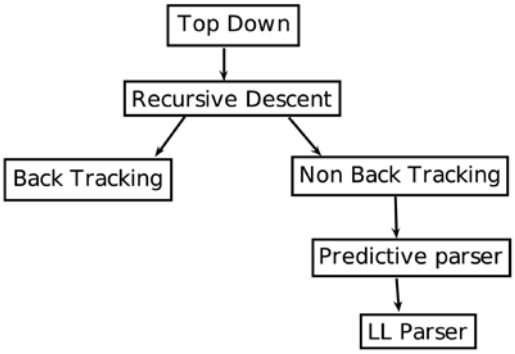

Figure 6: Types of Top down Parsers

General recursive-descent may need backtracking technique for repeated scans over input to arrive at the correct input. To allow backtracking the above code, the code needs to be modified in such a way that, it not only checks for current non terminal but also for all non terminals available in grammar to find the correct productions which matches with the input string, if it does not match then it raises an error.

Back Tracking Technique: Every string generated by applying productions on trail and error method based on the input string matched. If the prediction of the production is successful then parsing continues, otherwise in case of mismatch then at this stage previous prediction has to be rejected and pointer has to be set to the previous position and next production is predicted. This is known as Backtracking. Backtracking is one of the major drawback of top down parser. Predictive parser is the efficient non backtracking form of top down parser, where lookahead symbol unambiguously determines the procedure for each non terminal and hence no backtracking occurs.

Following example demonstrates the Backtracking technique. Consider the following grammar

$S \rightarrow hQf$

$Q \rightarrow al/a$

Using above grammar evaluate string w=haf

Step 1: S is the start symbol, therefore grammar starts from the symbol S and has only one production.

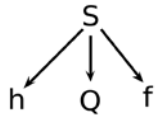

Figure 7

Step 2: Now input pointer is set to Q because h is a terminal. At this stage Q tends to 2 production that is Q → al/a. Parser predicts the production and Q is expanded.

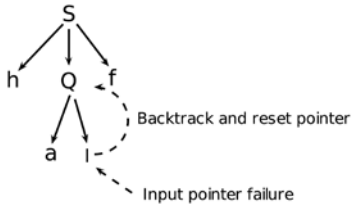

Figure 8: Backtracking

Step 3: Correct alternative is predicted and yield of the parse tree is w = haf.

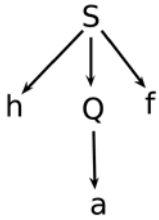

Figure 9: Alternative production

2) Predictive Parser: Recursive Descent parser which needs no back tracking is called predictive parser. Predictive parser technique can exactly decide that which production to be used based on the next input symbol. Predictive parser program maintains a stack which hold only non terminals and uses two dimensional table created from grammar.

Parser acts on the basis of two symbols that a symbol on top of stack and look ahead pointing to input buffer. Based on the various possibilities

1. If Stack top $ = lookahead symbol $ then parser halts. Successful Parsing Condition.
2. If Stack top is a terminal. Stack top t = lookahead symbol t then parser pops t and advances lookahead pointer, otherwise an error is raised.
3. If Stack top is a non terminal then parser predicts the entry. Non terminal is popped from stack and the right side of the production is pushed on to the stack from left to right. If appropriate production is not present then parser raises an error.

Predictive Parsers can be constructed for the class of grammar called LL(1). LL(1) grammar covers most of the programming constructs.

FIRST and FOLLOW Computation : FIRST and FOLLOW are the two necessary preliminary functions which is used in LL grammar. These functions allows us to select which productions to apply, based on the next input symbol.

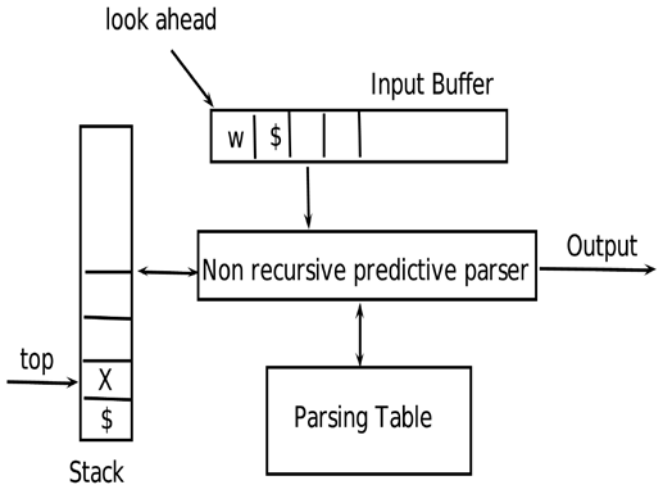

Figure 10: Model of Non recursive Parser

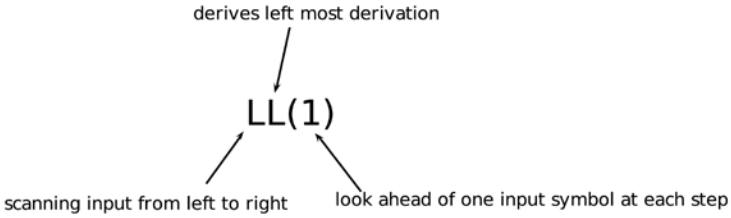

Figure 11: Meaning of LL(1)

FIRST : is function which gives the set of terminals that begins the strings derived from the production rule. Formally FIRST(α) = { t / (t is the terminal and $\alpha \Rightarrow^* t\beta$) or (t→ ε and $\alpha \Rightarrow^* \varepsilon$)}

FIRST Computation : To define FIRST(α). Let us define for a single symbol D

1. If D is a terminal : FIRST(D) = D.
2. If D is ε : FIRST(D) = ε.
3. If D is non terminal : In this case we must look at all grammar productions with D on left. If production is of the form D → $Y_1Y_2\ Y_3 \bullet \bullet \bullet Y_n$, where Y_i is single terminal or non terminal. a is in FIRST(D), if for some i, a is in FIRST(Y_i)

and ε is in all of FIRST(Y_1) • • • FIRST(Y_i) that is Y_1 • • •

$Y_{i-1} \Rightarrow^* \varepsilon$. If ε is in FIRST(Y_j) where j = 1,2,...,n then add ε to FIRST(D). Everything in FIRST(Y_1) is surely in FIRST(D). if Y_1 does not derive ε then we add nothing more to FIRST(D) but if $Y_1 \Rightarrow^* \varepsilon$ then FIRST(D) = FIRST(Y_1) -{ε} ∪ FIRST(Y_2) and same method is applied to subsequent non terminals.

FOLLOW : is function which gives the set of terminals that can appear immediately to the right side of the given symbol. It is defined for the single non terminal. Formally : FOLLOW(A) = {t / (t is the terminal and $S \Rightarrow + \alpha A t \beta$) or (t

is EOF and S⇒* α A)}. FOLLOW Computation : To define FALLOW(A). A is a single non terminal. FOLLOW(A) = EOF, if A is the start non terminal. For each production X→ α A β put FIRST(β) - {ε} in FOLLOW(A). if ε is in FIRST(β)then put FOLLOW(X) into FOLLOW(A). For each production X→ α A, put FOLLOW(X) into FOLLOW(A).

Construction of Parsing Table.

Parsing Table is constructed based on the function call select. Select function can be defined by First and Follow function. If production is of the form A→X then Select(A→X) = FIRST(FIRST(X) x FOLLOW(A)).

A Context Free grammar whose parsing table has no multiple entries is said to be LL(1). If LL(1) has same entries then the grammar is ambiguous and/or left recursive and/or not left factored. LL(1) Property : If non terminal appears on the left side of more than one production and select for those productions are disjoint. If this property hold good for a given grammar then grammar is LL(1). Following is the example to check id following grammar is LL(1).

E → PX

X → +PX / ε

P → RK

K → *RK / ε

R → (E) / id

FIRST and FOLLOW Computations : Compute FIRST and FOLLOW Functions for all the non terminals. Both are computed based on the given definitions above.

FIRST(E) = FIRST(PX) because : E →PX

= FIRST(P)

= FIRST(RK)

= FIRST(R)

= {(, id }

Likewise for the other productions FIRST function is computed

FIRST(X) = {+, ε}

FIRST(P) = FIRST(R) = {(, id }

FIRST(K) = {*, ε}

FIRST(R) = {(, id }

Now Compute FOLLOW function

FOLLOW(E) = { $,) }

FOLLOW(X) = FOLLOW(E) = { $,) }

FOLLOW(P) = FIRST(X) ∪ FOLLOW(E) = { +, ε } - {ε} ∪ { $,)} = { +,), $ }

FOLLOW(K) = FOLLOW(P) = { +,), $ }

FOLLOW(R) = FIRST(K) ∪ FOLLOW(P) ∪ FOLLOW(K) =

{*, ε} -{ε} ∪ { +,), $ } ∪ { +,), $ } = { *, +,), $}

Now Construct Parsing Table

To Construct parsing table which is also called as M table, we need Compute Select Function which guides us to fill the table. Select function is defined by FIRST and FOLLOW functions. For any production say X→W. Select function for the given production is defined as

SELECT(X→W) = FIRST(FIRST(W) x FOLLOW(X))

Let us compute Select functions for all the productions

SELECT(E→PX) = FIRST(FIRST(PX) x FOLLOW(E)) = FIRST({(, id} x {$,)}) = FIRST{((, $), ((,)), (id, $), (id,)} = {(, id}.

Similarly for the other productions SELECT function is computed.

SELECT(X→+PX) = FIRST(FIRST(+PX) x FOLLOW(X)) = FIRST({+} x { $,) }) = { + }

SELECT(X→ε) = FIRST(FIRST(ε) x FOLLOW(X)) = FIRST({ε} x { $,)}) = {$}

SELECT(P→RK) = FIRST(FIRST(RK) x FOLLOW(P) = {(, id}

SELECT(K→*RK) = { * }

SELECT(X→ ε) = { +,), $}

SELECT(R→(E)) = { (}

SELECT(R→id) = {id}

Following is the Parsing Table

The above table is filled on the following basis. For the production E →PX, SELECT(E) = { (, id } in this case the corresponding entry for M[E, (] = PX and M[E, id] = PX. Likewise other entries are made in the table based on Select function.

Non-terminals	(	id	+	*	)	$
E	PX	PX				
X			+PX		ε	ε
P	RK	RK				
K			ε	*RK	ε	ε
R	(E)	id				

Table 2: Predictive Parsing Table

LL(1) property : A grammar is an LL(1) iff the parsing table has no entries that are multiply defined. If a non terminal appears on the left side has more than one production then SELECT for those productions are disjoint, this is LL(1) property. For the same grammar above, non terminals which has more than one production are X, B and F.

For production X→+PX, X→ε SELECT function for X,

SELECT(X→+PX) ∩ SELECT(X→ε) = ϕ

For production K→*RK andK→ε SELECT(K→*RK) ∩ SELECT(K→ ε) = ϕ

For production R→(E) and R→id SELECT(R→(E)) ∩ SELECT(R→ id) = ϕ

Therefore X, K and R have LL(1) property. The given Grammar is LL(1).

4.2 Bottom Up Parsing

In bottom up approach, Parser starts constructing parse tree from the leaf node and works towards root node. Simplest form of Bottom up parsing is Shift Reduce Parser.

1) Shift Reduce: Shift reduce parser reduces the given input string into the start symbol. This parser uses 2 unique steps,

namely shift and reduce step. Data Structure used in this parser are Stack, input buffer, data structures to store and access the left and right of the production. Shift reduce parser performs various actions.

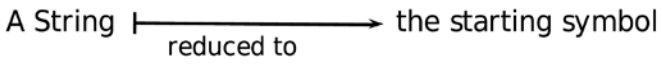

Figure 12: Shift reduce parser

(1)Shift action: Parser shifts the input symbol from the input tape on top stack one symbol at a time.
(2) Reduce action: It reduces top of the stack using appropriate production. The reduction is performed by popping the right side of the rule from the stack and pushing the left side of the production.
(3) Accept action: Parser announces the successful parse if the stack contains the start symbol and input tape is empty then input is accepted.
(4) Error action: If parser is not able to shift or reduce or accept, it announces syntax error has occurred.
Initial Configuration: $ is push on to stack to mark end of the stack and $ is concatenated at the end of the input string to indicate the end of string.
Limitations of Shift Reduce Parser :
(1) Shift Reduce Conflict : If Context Free Grammar has 2 productions of the form $A\rightarrow\beta$ and $\beta\rightarrow\beta p\gamma$. If B is on top of the stack and next token is p then parser is not able to decide whether it has to shift or reduce. This is known ans shift reduce conflict.
(2) Reduce Conflict: If Context Free Grammar has 2 productions of the form $A\rightarrow\alpha$ and $B\rightarrow\alpha$. If α is on top of the stack then in this case parser is not able to decide which production to apply to perform reduce action. This situation is Reduce Conflict.
2) LR Parser: LR parser is a non recursive, shift reduce, bottom up parser. LR grammars are a subset of CFG for which LR parsers can be constructed.

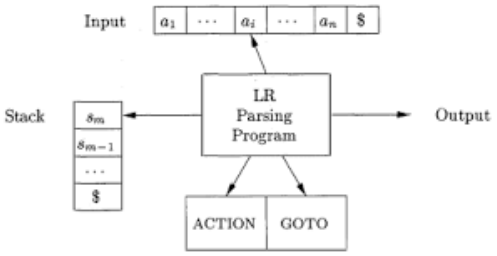

Figure 13: Model of LR Parser

LR parser require input, output, a stack, driver program and a parsing table. Parsing table consist action and goto procedure. Driver program remains same for all parsers, only parsing table changes according to the grammar. The parsing program reads characters from an input buffer one at a time. A program uses a stack to store a string of the form $S_0 X_1 S_1 X_2 \cdots S_m X_m$ where S_m is on top of the stack and X_i is grammar symbol and S_i is a State. If S_m is top of the stack and a_i is the current input symbol then driver program perform action[S_m, a_i] procedure which can one of the following actions
(1) Shift S, where S is the state action[S_m, a_i] = shift s, parser executes the shift move entering the configuration. A configuration of an LR parser is a pair whose first component is a stack content and second component is a the input.
(2) Reduce by the grammar production $A\rightarrow\beta$ action[S_m, a_i]=reduce $\rightarrow\beta$
(3) accept action[S_m,a_i] = accept, paring successfully completed.
(4) error action[S_m, a_i] = Error, parser discovers an error and calls for error recovery routine.

5.0 PARALLEL SYNTAX ANALYZER

Research was started on parallel compilation with the advent of microprocessors early in 1970 where Lincoln[12] first proposed the idea of parallel object code. Later Zosel[5] recognized the parallel loops. Mickunas and Shell[6] recognized the area in a compilation method where parallelization can be achieved and also proposed parallel lexical analysis, where lexical analysis can be broken into 2 sections called scanning and screening. They also proposed a parallel parsing method based on LR parsing. The 2 major requirement of the parallel processing is determining the ends of the reducible phrase and performing reduction parser was also extended, called piecewise LR (PLR). Many researchers attempted many other techniques to achieved parallelism during compilation process. Parallel syntax analyzer was implemented on different files[7]. This was achieved by selecting the file and scheduling to the specific processor for syntax analysis using processor affinity[8]. To estimate the speed up[9] in parallel processing 3 different modules were written.
(1) A Simulator, which emulates the behavior of the processor.
(2) A Generator, which keeps track of time as simulator works.
(3) An Estimator, computes the approximate numbers of basic parsing operation.
To compare the performance with that of parallel compilation in multi-core with respect to single core, Jacqus, Hickey and Joel[10] Computed upper Bounds for Speed up gained synchronous, multi purpose, bottom up, no back tracking parsing generated by bottom up parsing along with few assumptions made by them. Issues in implementing Parallel parsing on multi core machines was also identified[11]. Issues are division of code and Synchronization, Processor issues, Threading, Task distribution and Context Switching.

6.0 CONCLUSION

Though the work has been done but still the significant research has to carried out in this field to parallel syntax analysis. Various attempts has been made to parallelize parsing but still issues exists. Major work has to be done in identifying the area to be parallelized, splitting the code and synchronizing it. Future work is to develop syntax analyzer for NoC architecture.

REFERENCES

[1]. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffery D. Ullman Principles of Compiler Design, Addison Wesley Publication Company, USA, 1985.

[2]. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffery D. Ullman Compilers : Principles, Technique and Tools, Addison Wesley Publication Company, USA, 1986.

[3]. Jean Paul Tremblay,Paul G. Sorenson; The Theory and Practice of Compiler Writing, McGraw-Hill Book Company USA 1985.

[4]. David Gries; Compiler Construction for digital Computers, John Wiley & Sons Inc. USA, 1971.

[5]. M. Zosel; A Parallel Approach to Compilation, Conf. REc. ACM Sysposium on Principles of Programming Languages, Boston, MA, pp. 59-70, October 1973.

[6]. M. D. Mickunas, R. M. Schell; Parallel Compilation in a Multiprocessor Environment, Proceedings of the annual conference of the ACM, Washington, D.C., USA, pp. 241246, 1978.

[7]. Amit Barve and Brijendra Kumar Joshi; Parallel Syntax Analysis on Multi-Core, International Conference on Parallel, Distributed and Grid Computing, 2014.

[8]. http://www.linuxjournal.com/article/6799

[9]. Jacques Cohen and Stuart Kolodner; Estimating the Speedup in Parallel Parsing, IEEE Transaction on Software Engineering Vol SE - 11 1985.

[10]. Jacques Cohen, Tomothy Hickey and Joel Katcoff; Upper Bound for Speed up in Parallel Parsing, Journal of the Association for Computing Machinery Vol. 29 pp. 408 - 428 1982.

[11]. Amit Barve and Brijendra Kumar Joshi; Issues in Implementation of Parallel Parsing on Multi-Core Machines, International Journal of Computer Science, Engineering and Information Technology Vol 4, 2014.

[12]. N. Lincoln; Parallel Compiling Techniques for Compilers, ACM Sigplan Notices, 10(1970), pp. 18-31, 1970.