ISSN 1991- 8690 الترقيم الدولي ١٩٩١ - ٨٦٩٠
Website: http://jsci.utq.edu.iq Email:utjsci@utq.edu.iq

Enhancing the LR Parsing Strategy Using Incremental GPLR Parsing Method

Mouiad Abid Hani **Methaq Ibraheem Hashim**

Department of Computer Science - College of Education

Thi-Qar University

Abstract

Parsers in modern integrated development environments (*IDE*s) for general-purpose languages are virtually all of ad hoc, recursive descent variety. While such parsers have many disadvantages when compared with machine-generated *LALR(1)* parsers but they have two major good qualities: they are not restricted to any finite of lookahead, and in *IDE*, they can re-parse parts of a file as they change rather than re-parsing the entire file. Theoretically, both of these two capabilities can be achieved through variations of the traditional *LR* parsing techniques, but the traditional *LR* parsing methods still suffer two irresolvable problems; which are shift-reduce and reduce-reduce conflicts. In this research, we are trying to solve these two drawbacks with preserving the capabilities of the traditional *LR* parsing techniques. This has been achieved by employing the generalized piecewise *LR* parsing (*GPLR*) technique instead of the traditional *LR* techniques.

Keywords

Generalized Piecewise LR(1) GPLR (1), Chomsky Normal Form (CNF), Context-Free Grammar (CFG) Integrated Development Environment (IDE), Look-ahead LR(1) (LALR(1)), Deterministic Finite State Automata (DFSA), Cocke-Younger-Kasami (CYK).

1.Introduction

Knuth's discovery of the LR in 1964 became one of the most significant contributions of the formal language theory to software engineering. Being applicable to every context free grammar (CFG) and working in linear time, this algorithm possessed exactly the qualities in demand by the compiler industry, which ensured quick recognition and continued work in this direction [1]. Parsing is the linear structuring process achieved on sentences to know their belonging to a specific grammar. There are several reasons to perform this structuring process called parsing:

1. The first reason derives from the fact that the obtained structure helps us to process the object further. When we know that a certain segment of a sentence in German is the subject, that information helps in translating the sentence. Once the structure of a document has been brought to the surface, it can be converted more easily [2].
2. A second is related to the fact that the grammar in a sense represents our understanding of the observed sentences: the better a grammar we can give for the movements of bees, the deeper our understanding of them is [2].
3. A third lies in the completion of missing information that parsers, and especially error-repairing parsers, can provide. Given a reasonable grammar of the language, an error-repairing parser can suggest possible word classes for missing or unknown words on clay tablets [3].

There are number of ways by which this linear structuring process achieved. First an attempt to construct the parse tree can be initiated by starting at the root and proceeding downward toward the leaves. This method is called ***top-down parse***. Alternatively, the completion of the parse tree can be attempted by starting at the leaves and moving upward toward the root. This method is called ***bottom-up parse*** [4].

Example: Let the grammar is as shown below, and the string to be parsed, ***w=aabbbcc,*** see figure (1) in the appendix to get more information about the difference between top-down parsing and bottom-up parsing methods concerning the string ***w*** [5].

$$T \rightarrow R$$
$$T \rightarrow aTc \qquad \textit{grammar (1)}$$
$$R \rightarrow \lambda$$
$$R \rightarrow RbR$$

2.Related Works

Jane C. Hill and Andrew Wayne [8] outlined two different approaches for adapting the CYK algorithm to message-passing based parallel environment. Their first approach was to first distribute the grammar and string to be parsed among the processors, using a separate processor for each of the columns in the CYK table. Adrian Johnstone. Elizabeth Scott [9] has presented a new bottom-up nondeterministic parsing algorithm Generalized Reduction Modified LR Parsing (***GRMLR)*** that combines a modified notion of reduction with a Tomita-style breadth-first search of parallel parsing stack. In his thesis, Jeffery L. Overbey [10] has proved that Celenato's technique can be applied to GPLR parsers, despite their use of unbounded lookahead; furthermore, this

does not require a change to either algorithm, this result is prefaced by intuitive development of LR and GLPR parsing algorithm and Celenato's construction.

3.Foundations of Generalized piecewise LR(1) Parsing:GPLR(1)Parsing

3.1 Shift-Reduce Parsing

The shift-reduce parser for a grammar is conceptually simple, but it is not used in practice because it is ***nondeterministic***. The LR is more sophisticated variation of the shift-reduce parsing technique which eliminates this nondeterminism. Conceptually, the parser still shifts the symbols onto its stack and reduces them, but it is augmented with a state machine which controls its action. This state machine, coupled with the ability to look ahead at a finite prefix of the remaining input, guarantees that, at any point, the decision to shift or reduce by a particular production uniquely determined [5, 10].

Example: Suppose we have the balanced parentheses grammar:

$$S \rightarrow (S)$$
$$S \rightarrow ( ) \qquad \textit{grammar (2)}$$
$$S \rightarrow SS$$

And the input string is ***(( )) ( )***. The shift-reduce parser behaves as follows: symbols that are about to be popped because they match the right-hand side of a production are underlined; the nonterminal replacing them is displayed in boldface in table (1) [2]. The shift-reduce parser for a grammar is conceptually simple, but it is not used in practice because it is ***nondeterministic***. The LR is more sophisticated variation of the shift-reduce parsing technique which eliminates this nondeterminism. Conceptually, the parser still shifts the symbols onto its stack and reduces them, but it is augmented with a state machine which controls its action. This state machine, coupled with the ability to look ahead at a finite prefix of the remaining input, guarantees that, at any point, the decision to shift or reduce by a particular production uniquely determined, see table (1) [5, 6].

3.2 LR(0) Parsing

Although few useful grammar are LR(0), the construction of the LR(0) parsers can be the basis and the cornerstone of the construction for the more complicated parsers such as LALR(1), LR(1), and GPLR(1). LR(0) parser require no lookahead from the unexpected input string in order to make decision [4, 10]. The technique is called LR(0) parsing; the "L" is for left-to-right scanning of the input string, the "R" is for constructing the right-most derivation in reverse, and '0' is for no lookahead needed [7]. The most fundamental difference between the shift-reduce parser and the LR(0) parser is addition of ***Deterministic Finite State Machine(DFSA)***. While the basic shift-reduce parser decides what action to take (shift or reduce by a particular production) based solely on the contents of the parser's stack in the LR(0***)*** paring algorithm, this action is determined by the state of this machine. The state changes:

- As the inputs are read.
- When reduction is performed.

Now, let's show how to construct this ***DFSA***, and we will do that by an example as follows:

Example:

$$A \rightarrow aA$$
$$A \rightarrow B \qquad \textit{grammar (3)}$$
$$B \rightarrow bb$$

Which accepts the language $a^* bb$. When a parser starts, then, it expects to see a derivation of the input from the start symbol (in this case A) [6]. Since the two A-production in the grammar are $A \rightarrow aA$ and $A \rightarrow B$, this means the first thing it should expect to see is either an aA or B. Let us use

$$A \rightarrow .aA$$
$$A \rightarrow .B$$
$$B \rightarrow .bb$$

to indicate this, If we next need to shift the symbol a, we would use $[A \rightarrow a, A]$ to indicate this new situation, by the same way, the dot indicates where the parser is in matching the right-hand side of the production. When the symbol after the dot is terminal, like a, the meaning is obvious: The next symbol in the input should be exactly that symbol. But what does it mean for a parser to expect a nonterminal like B, since there are only terminals in the input string? Effectively, it means the parser should expect anything derivable from B this means the parser may also see bb at this point since $B \rightarrow bb$, so $B \rightarrow bb$ should also be included in the state, i.e., the set of items describing the state of the parser. Now we add a new production not already within our grammar to remedy the returning back to our initial state, so the new ***CFG*** will be [11]:

$$S' \rightarrow A\#.$$
$$A \rightarrow .aA$$
$$A \rightarrow .B$$
$$B \rightarrow .bb$$

Continuing by this manner according to algorithm, we will get the ***DFSA*** shown in figure (2) and the parse of the input string after padded it with **#** at the end, so it will be $w=aaabb\#$, knowing that the bottom of the stack also contains the symbol **#**, and this is to know when to stop parsing. See table (2) for full parsing of this string according to LR(0) algorithm [7]. The only drawback of LR(0) is its inability to insight the source input string which makes its work-limited [10]. In addition to storing shifted symbols on the stack, it is also needed to store what state the parser was in after the symbol was shifted. We will write these symbol-state pairs as $\begin{pmatrix} X \\ q \end{pmatrix}$, where X is the symbol and q is the state. At the beginning of the parse, we will place $\begin{pmatrix} \# \\ q0 \end{pmatrix}$ on the stack so that we have a record of the initial state [7, 10]

3.3 LR(1) Parsing

In ***LR(1)*** parsing method, the need to insight one symbol of the input stream make us calculate the ***FOLLOW*** sets for each nonterminal and since the ***FOLLOW*** computation needs us to know the ***FIRST*** set for each of them, we will give the full details about computing these sets [7, 10].

Follow sets computation algorithm

There is a ***Follow*** set for each variable symbol. ***Follow***(***A***) contains the set of terminals (not including λ) that could appear immediately after the variable ***A*** in some derivation.

begin
put # in ***follow***(***S***), where ***S*** is the start variable (start symbol)
repeat until no changes to any follow sets
Iterate over each rule ***R*** of the grammar
If ***R*** = $A \rightarrow \alpha B \beta$ then
add ***first*** (β) to ***follow***(***B***), excluding λ, if it is in ***First*** (β)
If (***R*** = $A \rightarrow \alpha B$) or (***R*** = $A \rightarrow \alpha B \beta$) and ***First*** ($\beta$) contains ($\lambda$) then
add ***Follow***(***A***) to ***Follow***(***B***)
end

Computing the FIRST sets algorithm

There is a ***first*** set associated with each variable symbol (some versions associate a First set with every symbol, variable or terminal, in the grammar). ***First***(***A***) contains the set of terminal symbols, plus λ, that may start strings produced by ***A***.

begin
repeat until no changes to any First sets
If ***A*** is nullable, add λ to ***First*** (***A***)
If $A \rightarrow a \gamma$, where ***a*** is a terminal, add ***a*** to ***First***(***A***)
If $A \rightarrow X_1X_2 \ldots X_n$, then
Compute the value ***i*** such that $0 \leq i \leq n$ and $X_1X_2 \ldots Xi$ are all nullable
add ***First***(X_1) $\cup$ ***First***(X_2) $\cup \ldots$ ***First*** (X_i) to ***First*** (***A***)
If $i < n$ then add ***First***(X_{i+1}) to ***First***(***A***)
end

Constructing of the set of items of LR(0) items algorithm

The procedures closure, goto and the main routine items construct the set of tems:

Function closure (***I***)
begin
J=***I***
repeat
for each item $A \rightarrow \alpha \cdot B \beta$ in J and each production
$B \rightarrow \gamma$ of ***G*** such that $B \rightarrow \cdot \gamma$ is not in J do

add $B \rightarrow \cdot \gamma$ to J
until no more items can be added to J
end

Function goto(I, X)
begin
C:= {closure {$[S' \rightarrow .S, \#]\}$;
repeat
for each set of items I in C and each grammar symbol
X such that goto (I, X) is not empty and not in C do
add goto (I, X) to C
until no more sets of items can be added to C.
end

Procedure Items (G') {G' is the augmented grammar of G}
begin
C ={closure ({[$S \rightarrow \cdot S$]})}
repeat
for each set of items I in C and each grammar symbol X
such that goto (I, X) is not empty and not in C do
add goto (I, X) to C
until no more sets of items can be added to C
end

The ***LR(1) DFSA*** is constructed as follows:

1- The initial state is closure$(\{[S' \rightarrow \cdot S, \#]\})$, where:

2- Given a set I of LR(k) items, the closure(I) is defined recursively as the smallest set satisfying:

$\text{Closure}_{LR}(I) = I \cup \{[B \rightarrow \cdot \gamma, \text{first}(\beta w)] \mid [A \rightarrow \alpha \cdot B\beta, w] \in \text{closure}_{LR}(I) \text{ and } B \rightarrow \gamma \in P\}$.

3- In a given state q, there is an item $[A \rightarrow \alpha \cdot X \beta, w]$, then there is also a state $q' = \text{goto}_{LR}(q, X)$ and there is a transition from q to q' on symbol X.

4- The function goto_{LR} is defined to be the smallest set satisfying:
$\text{goto}_{LR}(q, X) = \text{closure}_{LR}(\{ [A \rightarrow \alpha X \cdot \beta, w] \mid [A \rightarrow \alpha \cdot X \beta, w] \in q \})$.

5- The accepting state is the (unique) state containing $[S' \rightarrow S \cdot, \#]$.

Therefore, the parser is driven by two functions action_{LR} and goto_{LR}. The goto_{LR} has been described above; action_{LR} is defined as follows, given a ***DFSA's*** state q and a lookahead string $au \in V_T$, then:

$$Action_{LR}(q, au) = \begin{cases} \textbf{Shift and go to } \text{goto}_{LR}(q, a) & \text{if } \exists [A \to \alpha \cdot a\,\beta, w] \in q \\ \textbf{Reduce by } A \to \alpha & \text{if } \exists [A \to \alpha \cdot, au] \in q \\ \textbf{Accept} & \text{if } [S' \to S\cdot, \#] \in q \\ \textbf{Error} & \text{otherwise.} \end{cases}$$

General LR(1) parsing algorithm

```
begin
 set ip to the first symbol in w#
   repeat
   let q be the state on top of the stack and a is the current input symbol pointed to by ip
   if action (q, a)= shift s/ then
        begin
           push a then q/ on top of the stack;
           advance ip to the next input symbol
        end
    else
    if  (q, a)= reduce A → β then
       begin
          pop 2*|β| symbol off the stack;
          let q/ be the state now on the top of stack;
          push A then Goto[q/, A] on top of the stack;
          output the production A → β
       end
  else
     if action (q, a)= accept then
        return (success)
     else
       error (Error-Message);
end.
```

Construction LR(1) parsing table algorithm

begin

1. Construct $C = (I_1, I_2, \ldots, I_n)$, the collection set of items of ***LR(1)*** items for the augmented grammar G'.
2. State i of the parser is constructed from I_i. The parsing actions for state i are determined as follows:

a. If $[A \to \alpha.a\beta, b]$ is in I_i and goto$[I_i, a] = I_j$, then set action$[I_i, a]$ to "***shift j***". Here a required to be a terminal.

b. If $[A \rightarrow \alpha.,a]$ is in I_i, $A \neq S'$, then set action[I_i, a] to "***reduce*** $A \rightarrow \alpha$".
c. If $[S' \rightarrow S., \#]$ is in I_i, then set action to "***accept***".

If a conflict results form the above rules, the grammar is said not to be LR(1) grammar, and the algorithm is said to fail.

3. The goto transition for state ***i*** are determined as follows: If $[A_i, a] = I_j$ then the goto$[I, A] = j$.
4. All entries not defined by rules (2) and (3) are made "***error***".
5. The initial state of the parser is the one constructed from the set containing the item $[S' \rightarrow .S, \#]$.

end

The table formed from parsing action and the goto function produced by algorithm is called ***canonical*** (because the parser in this kind of parsing is one that only reduces handles, while noncanonical parser can also reduce phrase which are not handles, as it the case with GPLR parser).

Example: consider the following LR(1) grammar which generates the regular set b^*db^*d

$$S' \rightarrow \cdot S$$
$$S \rightarrow KK \qquad \textit{grammar (4)}$$
$$K \rightarrow bK / d$$

The LR(1) parsing table is shown in table (2) and the DFSA machine equivalent to the parser is shown in figure (3).

3.4 LR(1) irresolvable problems

As we have mentioned in the abstract, the traditional LR parsing methods still suffer two irresolvable problems; which are shift-reduce and reduce-reduce conflicts, these two problems made us present this study in the hope that the suggestion introduced in this research may eliminate or reduce the effect of these two conflicts. At first we will give two examples regarding these conflicts and then resent our method to manipulate them.

Example: (shift-reduce conflict) Consider the following grammar:

$$S \rightarrow .R$$
$$L \rightarrow .*R \qquad \textit{grammar (5)}$$
$$L \rightarrow .id$$
$$R \rightarrow .L$$

The set of items are as follows:

I_0: $S' \rightarrow .S$
$S \rightarrow .L=R$
$S \rightarrow .R$

I_5: $L \rightarrow id.$

I_6: $S \rightarrow .=.R$

$L \rightarrow .*R$ $R \rightarrow .L$
$L \rightarrow .id$ $L \rightarrow .*R$
$R \rightarrow .L$ $L \rightarrow .id$

$I_1: S' \rightarrow S.$ $I_7: L \rightarrow *R.$

$I_2: S \rightarrow L.=R$ $I_8: R \rightarrow L.$
$R \rightarrow .L$

$I_3: S \rightarrow R.$ $I_9: S \rightarrow L=R.$

$I_4: L \rightarrow *.R$
$R \rightarrow .L$
$L \rightarrow .*R$
$L \rightarrow .id$

Consider the set of items I_2, the first item in this set makes action (***q***, =) to be ***shift*** since ***FOLLOW(R)*** contains =(to see why; consider $S \Rightarrow L.=R \Rightarrow * R=R$), the second item set action (***q***, =) to "***reduce R → L.***" thus the entry [2, =] is multiply defined. Since there is both shift and reduce entries in the same action field [2, =], so state 2 has a shift-reduce conflict [7, 11].

Example: (reduce-reduce conflict) Consider the following grammar:

$S \rightarrow aA$
$S \rightarrow bB$
$A \rightarrow Ca$
$A \rightarrow Db$ ***grammar (6)***
$B \rightarrow Cb$
$B \rightarrow Db$
$C \rightarrow E$
$D \rightarrow E$
$E \rightarrow \lambda$

The problem arises here from the permutation of ***C, d, a,*** and ***b*** in rules ***A*** and ***B*** that makes ***FOLLOW***(***C***) and ***FOLLOW***(***D***) the same; therefore, the lookahead can not be used to guide the parser when reducing ***C*** and ***D***. This fact combined with the ***FOLLOW*** overlap, renders the grammar non-LR(1). Because a reduce-reduce conflict between rules ***C*** and ***D*** is unavoidable [5].

4.The proposed method

GPLR parser is a two-stack parsing mechanism. The first stack servers the same function as the stack in ordinary LR parsing technique; we will simply refer to it as 'the stack", which we will it the "input stack" initially contains the input string with the first

symbol on the top of the stack and the end marker "#" at the bottom of the stack. Symbols will be popped off the input stack, just as an ordinary LR parser reads its inputs from left to right. However, GPLR parser will also pushes symbols back to the next input symbols. While an ordinary LR parser has four actions: ***Shift***, ***Reduce***, ***Accept***, and ***Error,*** GPLR parser can also ***cancel*** and ***continue***. Continuation is the action which allows the parser to temporarily ignore the error resulted from the conflict; cancellation is used in returning back to the site in which the conflict happened.

4.1 The cancel action

Suppose the stack of a GPLR parser contains the following symbols (from bottom to top)

$$a \perp b\ c$$

And it has just readied a state with a single reduce item allowing it to reduce ***abc*** to some nonterminal, say ***F***. In other words, after it shifted ***a***, it was not sure whether to reduce ***a*** to something or shift the ***b***, so it ***continued***. But now that it has seen the *b* and *c* following it, it knows for sure that shifting *b* was correct. Unfortunately, since it is not allowed to look at symbols below⊥, the parser cannot reduce ***abc***, as it can only see *be.* The solution is something called an insufficient, stack depth reduction. The parser will pop as many symbols as possible (***b*** and ***c***), remove the ⊥ marker, and place a cancellation symbol at the front of the input stream, followed by the nonterminal it attempted to reduce to ***(F)***. Cancellation symbols indicate (1) which production the parser wants to reduce by, and (2) what suffix of that production's right-hand side was actually visible on the stack. In this case, we wanted to reduce by ***F —> abc,*** but only ***bc*** was visible on the stack, so the cancellation symbol will be ⟨***F —> a . bc***⟩. Now that the ⊥ marker and the stack contents above it are gone, the parser will return to the state it was in before it continued (the state on top of the stack), but it will use this cancellation symbol as the lookahead token. It will recognize that there is, in fact, an ***a*** on top of its stack, so it will cancel ***a***, i.e., pop ***a*** (and the corresponding state) off the stack and remove the cancellation symbol from the input stream. It is also possible to cancel with insufficient stack depth. In this case, if the parser needs to cancel ⟨***A —> α . βγ***⟩, but only ***β*** is visible on the stack, the cancellation symbol is popped from the input stack, the symbols of ***β*** and the ⊥ symbol below them are popped from the parser stack, and a new cancellation symbol ⟨***A —> α . βγ***⟩ is pushed into the input stack. As with an insufficient stack depth reduction, the parser's current state is set to the new state on top of the stack, and this new cancellation symbol serves as the next input symbol.

4.2 The continuation action

A GPLR parser continues when there are at least two different (shift and/or reduce) actions it could have taken. To continue, it places a special marker on the stack (denoted ⊥) and changes the current state to a continuation state. Like goto states, continuation states are constructed to follow from an existing state on a particular input symbol, so we can speak of, say, the continuation state from q_0 on ***B***. After entering a continuation state, the parser proceeds to make reductions on the remaining input, but it

only makes reductions that would have been made no matter which of the possible prior actions was taken. This is ensured by two means. First, the stack marker (⊥) is treated as the bottom of the stack; the parser is not allowed to look at any of the symbols below ⊥ until the ⊥ symbol is explicitly removed from the stack by a cancel action. This will be described further in the discussion of cancellation, but for now, suffice it to say that this will prevent the continuation site from accidentally being "absorbed" into a large reduction. Second, the continuation state is constructed to guarantee that only conservative reductions are made. Suppose one shift and two reduce actions were possible in the original state on a particular lookahead symbol. Then there are several shift items that should be carried over to the continuation state; if it made either reduction, there are several states it could have moved to, and the items from these states are carried over as well. In other words, the continuation state is constructed to include all of the items that would be valid no matter which of the original actions was taken. As symbols are shifted, the goto states from this continuation state will be followed, and if a state is ever reached where only one reduce action is possible—i.e., there is only a single reduce item, then we can be assured that, no matter which of the original actions was taken, the parser would have ended up in a state with only this single reduce item.

General GPLR(1) parsing algorithm

```
Begin
 set ip to the first symbol in w#
  repeat
    let q be the state on top of the stack and a is the current input symbol pointed to by ip
    if action (q, a)= shift s' then
        begin
           push a then q' on top of the stack;
           advance ip to the next input symbol
        end
    else
        if  (q, a)= reduce A → β then
           begin
             pop 2*|β| symbol off the stack;
             let q' be the state now on the top of stack;
             push A then goto[q', A] on top of the stack;
             output the production A → β
           end
       else
         if action (q, a)= cancel then
            begin
              if | R_{I,χ} | =0 and |Core(C_{I,χ})| =1
              where [A → α · β] ∈ Core(C_{I,χ}) then
              cancel  < A → α · β >
```

```
    end
  else
    if action (q, a)= continue then
        begin
          if any of the following hold:
          | S_{I,χ} | ≥ 1 and | R_{I,χ} | ≥ 1
          | R_{I,χ} | ≥ 1 and |Core (C_{I,χ})| ≥ 1
          | R_{I,χ} | ≥ 2
          |Core (C_{I,χ})| ≥ 2 then
          continue in Cont_GPLR (I, χ)
        end
   else
     if action (q, a)= accept then
       return (success)
     else
       error (Error-Message);
 End.
```

Construction of the sets of GPLR Items

In the following formalism, β, γ, α, ζ, δ, η and $\emptyset$ denotes strings in V^*; A, B, C, and D denote nonterminals; X and Y denote symbols in $V = V_N \cup V_T$. We will use I_{GPLR} to refer to the set of all GPLR items for a grammar, which is the set consisting of the every item appearing in every state of the GPLR *DFSA*.

Definition Given an augmented grammar $G = (V_T, V_N, V^*, P, S)$, the set of V_C of cancellation symbols for G is:

$$V_C = \{ \langle A \rightarrow \alpha \cdot \beta \rangle \mid A \rightarrow \alpha\beta \in P, |\alpha| > 0, \text{ and } |\beta| > 0\}$$

Subset Selectors

We begin by defining some notations: $S_{I,\chi}$, $R_{I,\chi}$, and $C_{I,\chi}$ *which* will be used to refer to the shift, reduce and cancellation items, respectively, in a state I on lookahead χ.

Definition Given a set $I \subseteq I_{GPLR}$ of GPLR items and the symbol $\chi \in V_C \cup V^*$, the GPLR shift items in I for lookahead χ are precisely the elements of the set:

$$S_{I,\chi} = \begin{cases} \{[A \rightarrow \alpha \cdot \chi \beta, B \rightarrow \gamma \cdot \delta] \in I\} & \text{if } \chi \in V_N \cup V_T \\ \emptyset & \text{if } \chi \in V_C. \end{cases}$$

Definition Given a set $I \subseteq I_{GPLR}$ of GPLR items and the symbol $\chi \in V_C \cup V^*$, the GPLR reduce items in I for lookahead χ are precisely the elements of the set:

$$R_{I,\chi} = \begin{cases} \{[A \to \alpha \cdot, B \to \gamma \cdot \delta] \in I \mid \delta \Rightarrow^{*} \chi w,\ \exists\ w \in V^{*}_{T}\} & \text{if } \chi \in V^{*} \cup V_T \\ \{[A \to \alpha \cdot, B \to \gamma \cdot \delta] \in I\} & \text{if } \chi = \langle B \to \gamma \cdot \delta \rangle \end{cases}$$

Definition Given a set $I \subseteq I_{GPLR}$ of GPLR items and the symbol $\chi \in V_C \cup V^{*}$, the GPLR cancellation items in I for lookahead χ are precisely the elements of the set:

$$C_{I,\chi} = \begin{cases} \emptyset & \text{if } \chi \in V_N \cup V_T \\ \{[A \to \alpha \cdot \beta, B \to \gamma \cdot \delta] \in I\} & \text{if } \chi = \langle A \to \alpha \cdot \beta \rangle \end{cases}$$

Definition Given a GPLR item, the *core* of the item is the LR(0) item

$$\textbf{Core}\ ([A \to \alpha \cdot \beta, B \to \gamma \cdot \delta]) = [A \to \alpha \cdot \beta]$$

We can extend this definition to operate on a set $I \subseteq I_{GPLR}$

Definition: given a GPLR item, the core of the item is the LR(0) item

Core(*$[A \to \alpha.\beta, B \to \gamma.\delta]$ is the item $[A \to \alpha.\beta]$*)

Definition: Given a set of $\subseteq I_{GPLR}$ of GPLR items for the grammar $G = (V_T, V_N, P, S)$, the GPLR closure of I is defined as the smallest set satisfying:

Closure$_{GPLR}(I)$ = $\{[C \to .\zeta, A \to \alpha C .\beta] / \cup [A \to \alpha.C\beta, B \to \gamma .\delta] \in$ Closure$_{GPLR}(I)$ and $\{C \to .\zeta \in P,$ where $\beta \Rightarrow^{*} w$ for some $w \in V^{+}_{T}\} \cup \{[C \to .\zeta, B \to \gamma.\delta] / [A \to \alpha.C.\beta, B \to \gamma.\delta] \in$ Closure$_{GPLR}(I)$ and $C \to \zeta \in P,$ where $\beta \Rightarrow^{*} \lambda\}$

The GPLR

Armed with a general understanding of **how** a GPLR parser should work, we will now focus on the deterministic finite state automaton (***DFSA***) which derives it. Just as the states of an LR(1) ***DFSA*** were sets of LR(1) items, the states of a GPLR items will be sets of GPLR items.

GPLR items and Cancellation Lookahead

A GPLR consists of two components items and is written, e.g.,

$$[A \to aB\cdot, C \to D.e]$$

The second component item can not have a dot at the right or left end. In some cases, there will be a second component, e.g,

$$[A \to aB.,]$$

The first component has the exact same meaning of the as in the LR(0) ***DFSA***; it is called the ***core*** of the GPLR item. We refer to the GPLR item as ***shift item*** or ***reduce item*** if its core is an LR(0) shift item or reduce item, respectively. The second component is the ***cancellation lookahead***; it serves a function similar to the lookahead component

in LR(1) items. Specifically, the parser reduces item such as [$A \rightarrow aB\cdot$, $C \rightarrow D.e$] appears in the state, then if the parser reduces aB to A (and possibly makes a few more reduction immediately after that), it may be possible for it to end up in a state where it can ***cancel*** $\langle C \rightarrow D.e \rangle$ - i.e., a state containing an item of the form $[C \rightarrow D.e,$ ***something*** $]$. As in the LR(0) ***DFSA***, if a state contains an item with the core $[C \rightarrow .De]$ and then recognized a D. So, consequently, a parser can ***cancel*** anything that appears as a core in its current state. The purpose of GPLR lookahead, then, is to determine whether a reduction may lead to a state where a particular cancellation is possible. This will be used to determine the parser's action when cancellation symbol appears in the input stream. So, for example if $[A \rightarrow aB\cdot, C \rightarrow D.e]$ appears in the parser's current state, and the cancellation symbol $\langle C \rightarrow D.e \rangle$ appears next the input, then it should ***reduce*** by $A \rightarrow aB$, since it is possible (though not guaranteed) that it will end up in a state cancellation by $\langle C \rightarrow D.e \rangle$ which is feasible.

Constructing the Machine

The Construction of GPLR(1) ***DFSA*** is very similar to the construction of the LR(1) ***DFSA***, adjusted appropriately for cancellation lookahead. The initial state of GPLR(1) ***DFSA*** is the closure of the $\{[S' \rightarrow \cdot S\#,]\}$; the final state is the (unique) state containing $[S' \rightarrow S.\ \#,]$. Before describing how closures are computed, we will explain how goto states are computed. As with LR(1) construction, to compute the goto state from a (close) state q on a terminal or nonterminal X, all of the items in q with X after the dot in the core component incorporated into the new state with the dot moved forward one position, and lookaheads are simply carried over unchanged form the original state. To see why, suppose a state contains the item $[A \rightarrow a.bc, D \rightarrow e.F]$. Then the goto state on b will contain $[A \rightarrow ab.c, D \rightarrow e.F]$. In the original state, the lookahead, $D \rightarrow e.F$ meant that, after it becomes possible to reduce abc to C, doing so (and possibly making further reductions) could put the parser in a state containing the core $D \rightarrow e.F$. This statement is just as true in the new (goto) state as it was in the original state: It does not depend on the position of the dot in core. The closure of a set of GPLR(1) is computed as follows: Each item with a nonterminal after the dot has the form:

$$[A \rightarrow \alpha.B\beta, C \rightarrow \gamma.\delta]$$

We find all the productions for B in the grammar; call them $B \rightarrow \alpha_1$, $B \rightarrow \alpha_2$,…$B \rightarrow \alpha_n$. we have two cases to consider. If β derives at least one string other than λ, we should ensure that the state contains each of the items:

$$[B \rightarrow .\alpha_1, A \rightarrow \alpha.B.\beta]$$
$$[B \rightarrow .\alpha_2, A \rightarrow \alpha.B.\beta]$$
$$\ldots,$$
$$[B \rightarrow .\alpha_n, A \rightarrow \alpha.B.\beta]$$

Intuitively, the item $[A \rightarrow \alpha.B\beta, C \rightarrow \gamma.\delta]$ indicates that the parser is in a state where it needs to recognize a B, but it will get sidetracked doing this (it will have to go through several more states and eventually ***reduce*** something to B). But after it does so, it will return to this state and jump forward to goto state on B. Then, it may be possible to

cancel $\langle A \rightarrow \alpha.B.\beta \rangle$ in that state. If $\beta \Rightarrow^{*} \lambda$ (or $\beta=\lambda$), we should ensure that the state contains each of the items:

$$[B \rightarrow .\alpha_1, C \rightarrow \gamma.\delta]$$
$$[B \rightarrow .\alpha_2, C \rightarrow \gamma.\delta]$$
$$\ldots,$$
$$[B \rightarrow .\alpha_n, C \rightarrow \gamma.\delta]$$

As before, the parser will get sidetracked recognized B but will then return to this state and jump forward to the goto state on B. the goto state on B will contain the item:

$$[A \rightarrow \alpha.B.\beta, C \rightarrow \gamma.\delta]$$

In the event that λ is reduced to β and the item

$$[A \rightarrow \alpha.B\beta., C \rightarrow \gamma.\delta]$$

Becomes valid, the parser should be prepared to ***cancel*** $\langle C \rightarrow \gamma.\delta \rangle$ without consuming any input. When it is possible to ***continue*** in a state q (we will describe how to determine this momentarily) on a particular symbol, we will also need to compute a ***continuation*** state form q on that symbol. The symbol will be either a terminal or nonterminal; call it X. the continuation state from q on X should contain the closure of all of the following items:

- All the items in q that have an X after the dot in the core should be included; i.e., if $[A \rightarrow \alpha.X\beta, B \rightarrow \gamma.\delta]$ is in q, then it is also included in the continuation state from q on X. Since there is an X after the dot, it is possible to shift X; by including these items and closing the state, we are retaining the ability to shift X after we continue.
- We should determine all of the reduce items in q that have an X after the dot in the lookahead item. Each of these items has the form $[A \rightarrow \alpha Y., \beta \rightarrow \gamma.X\delta]$ for some A, α, Y, B, γ, and δ. For of these items, we should find every production whose right-hand side includes a nonterminal which derives a string ending in B; that is, we should find every production $C \rightarrow \zeta D\eta$ (for some $C, \zeta, D,$ and η) such that $C \Rightarrow^{*} \theta B$ for some θ. Then the continuation state should include $[B \rightarrow \gamma.X\delta, C \rightarrow \zeta D.\eta]$. Effectively, this is including every possible item that could appear if we had reduced $\alpha.X\beta$ to A in q (and then moved to the goto state on A). Since $B \rightarrow \gamma.\delta$ was the lookahead in the original item, through some sequence of reductions, we could end up in a state with that core. We then need to find every lookahead that could be paired with that core, so we form all the lookaheads which have a dot immediately after B. All the lookaheads corresponding to a core that could appear immediately after B was recognized. The initial state, its goto states, the goto states from those goto states, etc. are called unprimed states. All of the other states – continuation states and their goto states that were not already in the DFSA – are called primed states.

GPLR Action Function

Definition Given a set $I \subseteq I_{GPLR}$ of GPLR items and the symbol $\chi \in V_C \cup V$, the GPLR action function is defined such that:

$$\text{Action}_{GPLR}(I,X) = \begin{cases} \text{Shift and go to Goto}_{GPLR}(I,\chi) & \text{if } |S_{I,\chi}| \geq 1 \text{ and } |R_{I,\chi}| = 0 \\ & (\text{Note that this requires } \chi \in V^*) \\ \text{Reduce by } A \rightarrow \alpha & \text{if } |R_{I,\chi}| = 1,\ |S_{I,\chi}| = 0, \text{ and } |C_{I,\chi}| = 0 \\ & \text{Where } [A \rightarrow \alpha, A \rightarrow \gamma \cdot \beta] \in R_{I,\chi} \\ \text{Cancel } < A \rightarrow \alpha \cdot \beta > & \text{if } |R_{I,\chi}| = 0 \text{ and } |Core(C_{I,\chi})| = 1 \\ & \text{Where } [A \rightarrow \alpha \cdot \beta] \in Core(C_{I,\chi}) \\ \text{Continue in Cont}_{GPLR}(I,\chi) & \text{if any of the following hold:} \\ & |S_{I,\chi}| \geq 1 \text{ and } |R_{I,\chi}| \geq 1 \\ & |R_{I,\chi}| \geq 1 \text{ and } |Core(C_{I,\chi})| \geq 1 \\ & |R_{I,\chi}| \geq 2 \\ & |Core(C_{I,\chi})| \geq 2 \\ \text{Accept} & \text{if } [S' \rightarrow S \cdot \#,] \in I \\ \text{Error} & \text{Otherwise.} \end{cases}$$

The GPLR Action function is straightforward: Shift and reduce item induce ***shift*** and ***reduce*** actions, as expected. The contexts in which a parser may ***cancel*** or ***continue*** have already been described and should not surprise.

Let q be a state in ***DFSA*** (i.e., the set of GPLR items) and χ input symbols (terminal, nonterminal, or cancellation symbol).

- If is a terminal or nonterminal contains an item of the form $[A \rightarrow \alpha \cdot \beta, B \rightarrow \gamma . \delta]$, then the parser in state q with input χ should ***shift*** χ (note that we must have $\chi \in V$].
- If q contains an item of the form $[A \rightarrow \alpha \cdot, B \rightarrow \gamma . \delta]$, then a parser in the state q input χ should ***reduce*** by $A \rightarrow \alpha$ if one of the following holds:

1. $\delta \Rightarrow^* \chi\iota$ for some $\iota \in V^*$ (note that this requires $\chi \in V$) or
2. $\chi = \langle B \rightarrow \gamma . \delta \rangle$.

- If $\chi = \langle B \rightarrow \gamma . \delta \rangle$ and q contains an item of the form $[A \rightarrow \alpha \cdot \beta, B \rightarrow \gamma . \delta]$ (So $\alpha, \beta \neq \lambda$), then a parser in state q with input $\chi = \langle A \rightarrow \alpha \cdot \beta \rangle$ should cancel $\langle A \rightarrow \alpha \cdot \beta \rangle$.

- If more than one of the above applies, the parser should ***continue***, with one exception. If both a ***cancel*** and ***reduce*** actions are indicated for an unprimed state, the conflict is unavoidable, and the parser construction fails, in this case, the grammar is said to be non-GPLR(1) grammar.

Example: to illustrate the incremental GPLR parsing, we will use the grammar:

$$S \rightarrow AB$$
$$A \rightarrow A_1x \,/A_2y$$
$$A_1 \rightarrow A_1\, a \,/a$$
$$A_2 \rightarrow A_2\, a \,/a \qquad \textit{grammar (7)}$$
$$B \rightarrow Bb \,/b$$

Which recognizes the language $a^+(x/y)b^+$. The exact parse of a^+, which may be arbitrarily long, depends on whether x or y follows. Thus, we can not reduce any of as until we have seen all of the as; since the number of as is arbitrarily long, this grammar is not LR(k) for any k. This grammar is GPLR(1) and the full parse of the string $aaaaxbbb \in a^+(x/y)\, b^+$ is shown in table (4). Note the symbol $\langle * \rangle$ is used to denote the cancellation symbol $\langle A_1 \rightarrow A_1 \cdot a \rangle$.

5 Conclusion

GPLR(1) is a two-stack parsing algorithm. The first stack serves the same function as the stack in an ordinary LR(1); the second stack, which we called either input stack, initially contains the input string, with the first symbol on the top of the stack and the end marker (#) at the bottom. Symbols will be popped off the input stack, just as an ordinary LR(1) parser reads its input from left to right. However, a GPLR(1) parser will also push symbols back to onto the input stack, at which point they will serve as the next input symbol. LR(1) parsing is often touted as one of the great successes of computer science. This problem is beautifully mathematical, the solutions are provably correct and the results are intensely practical. In this study, we have tried to give an example of trying to do something best than other, and this is done through adding two actions which were not existent in the traditional LR(1) parsing, and these actions are ***cancel*** and ***continue***. Continuation is the action which allows the parser to temporarily ignore the error resulted from the conflict; cancellation is used in returning back to the site in which the conflict happened. Of course, this not the perfect solution for the parsing problems, but at least it is a humble attempt to do something useful. Also, we thing that the actions added to traditional LR(1) will enhance its efficiency and practical use.

References

1. Alexander, Okhotin. "A PRELIMINARY REPORT ON GENERALIZED LR PARSING for BOOLEAN GRAMMARS", Technical report, Queen's University, School of Computing, Ontario, Canada, July, 2004.

2. D, G. C, J. Jacobs. "PARSING". Amsterdam University press, the Netherlands. 1998.

3. J, E. Hopcroft. J, D Ullman. "INRODUCTION TO AUTOMATA THEORY, LANGUAGES and COMPUTATION". Reading Mass. Addison-Wesley, 1979.

4. Jean-Paul Tremblay, Paul, G. Sorenson. "THE THEORY AND PRACTICE OF COMPILER WRITING". McGraw-Hill Inc. 1989.

5. T, John Parr. "OBTAINING PRACTICAL VARIANTS OF LL(K) AND LR(K) FOR K>1 BY SPLITTING the ATOMIC K-TUPLE". Doctor of Philosophy Thesis, faculty of Purdue University, 1993.

6. Torben, Egidius Mogensen. "Basics OF Compiler Design", April 10, 2007. http://www.diku.dk/˜torbenm/Basics

7. A, V. Aho. R, Sethi, J. Ullman. "COMPILERS: PRINCIPLES, TECHNIQUES AND TTOOLS", Addison-Wesley, 1986.

8. Jane C. Hill and Andrew Wayne: "A ***CYK*** APPROACH TO PARSING IN PARALLEL: A CASE STUDY". In proceedings of twenty-Second SIGCSE Technical Symposium on computer science Education: pp 240-245, ACM Press, 1991.

9. Adrian, Johnstone. Elizabeth Scott. "GENERALIZED REDUCTION MODIFIED LR PARSING for DOMAIN SPECIFIC LANGUAGE PROTOTYPING", Proceedings of the 35th Hawaii International Conference on Computer Sciences, 2002.

10. Jeffery, L. Overbey, "PRACTICAL, INCREMENTAL, AND NONCANONICAL PARSING: CELENTANO'S ALGORITHM AND THE GENERALIZED PIECEWISE LR ALGORITHM". Computer Science Master Thesis, Illinois University, 2006.

11. P, Venable. "MODELING SYNTAX FOR PARSING AND TRANSLATION". Doctor of Philosophy Thesis, School of computer science, Carnegie Mellon University, 2003.

Appendix A

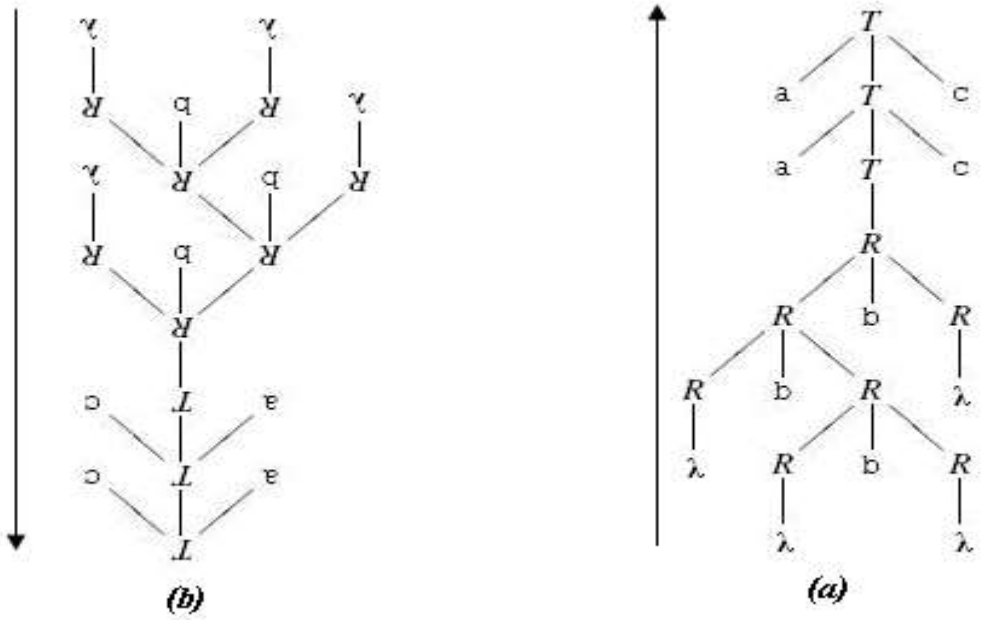

Figure (1) Illustrates Bottom-up versus Top-Down for grammar (1).

Table (1) Illustrates Shift-Reduce parsing for grammar *(2)*.

Stack contents	*Input string*	*Action*	*Production*
	(())()#	*Shift*	
#(	())()#	*Shift*	
#((	))()#	*Shift*	
#(()	)()#	*Reduce*	$S \rightarrow ()$
#(S	)()#	*Shift*	
#(S)	()#	*Reduce*	$S \rightarrow (S)$
#S	()#	*Shift*	
#S(	)#	*Shift*	
#S()		*Reduce*	$S \rightarrow ()$
#SS		*Reduce*	$S \rightarrow SS$
#S		*Accept*	

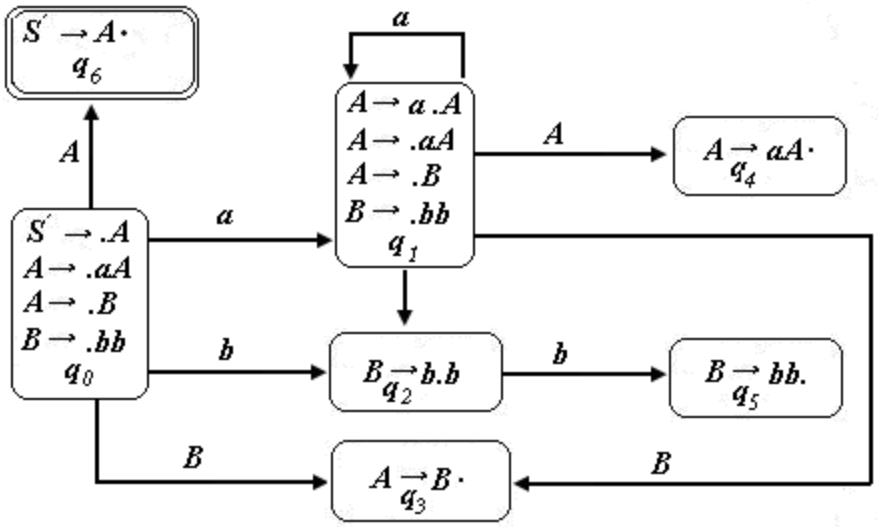

Figure (2) illustrates the *DFSA* of grammar (4)

Table (2) Canonical Parsing Table of grammar (4)

State	Action			Goto	
	b	*d*	#	*S*	*K*
0	*q3*	*q4*		1	2
1					
2	*accept*				5
3	*q6*	*q7*			8
4	*q3*	*q4*			
5	*r3*	*r3*			
6			*r1*		9
7	*q6*	*q7*			
8			*r3*		
9	*r2*	*r2*			
			r2		

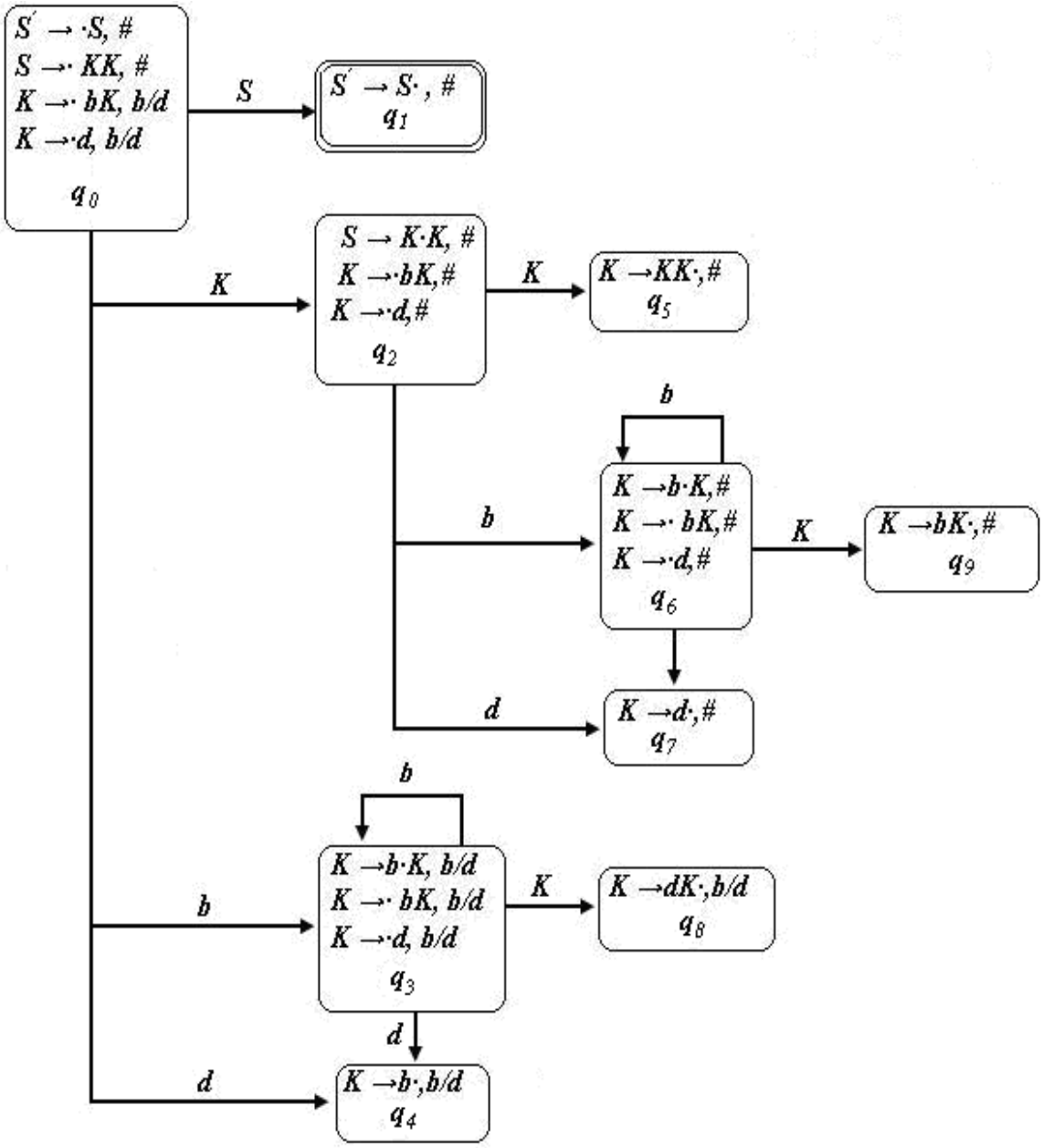

Figure (3) the goto graph for grammar (4)

Table (3) illustrates the LR(0) parsing of the string *aaabb#* of grammar (3)

Stack	Remaining input string	Action
# q_0	*aaabb#*	Shift *a*
# *a* q_0 q_2	*aabb#*	Shift *a*
# *a* *a* q_0 q_2 q_2	*abb#*	Shift *a*
# *a* *a* *a* q_0 q_2 q_2 q_2	*bb#*	Shift *b*
# *a* *a* *a* *b* q_0 q_2 q_2 q_2 q_2	*b#*	Shift *b*
# *a* *a* *a* *b* *b* q_0 q_2 q_2 q_2 $\underline{q_2\ q_5}$	#	Reduce by $B \rightarrow bb$.
# *a* *a* *a* *B* q_0 q_2 q_2 q_2 q_3	#	Reduce by $A \rightarrow B$.
# *a* *a* *a* *A* q_0 q_2 q_2 $\underline{q_2\ q_4}$	#	Reduce by $A \rightarrow aA$.
# *a* *a* *A* q_0 q_2 $\underline{q_2\ q_4}$	#	Reduce by $A \rightarrow aA$.
# *a* *A* q_0 $\underline{q_2\ q_4}$	#	Reduce by $A \rightarrow aA$.
# *A* q_0 q_4	#	Reduce by $S' \rightarrow A$.
# *A* q_0 q_4		Accept

Table (4) The Complete incremental GPLR(1) parse of *aaaaxbbb in* the example grammar (7)

Step	Parser Stack	Input Stack	Action
1.	# q_0	aaaaxbbb#	Shift a and go to q_5
2.	# a q_0 q_5	aaaxbbb#	Continue in q_{13}
3.	# a A_1 q_0 q_5 q_{13}	aaaxbbb#	Shift a and goto q_{14}
4.	# a A_1 a q_0 q_5 q_{13} q_{14}	aaxbbb#	Continue in q_{13}
5.	# a A_1 a A_1 q_0 q_5 q_{13} q_{14} q_{13}	aaxbbb#	Shift a and goto q_{14}
6.	# a A_1 a A_1 a q_0 q_5 q_{13} q_{14} q_{13} q_{14}	axbbb#	Continue in q_{13}
7.	# a A_1 a A_1 a A_1 q_0 q_5 q_{13} q_{14} q_{13} q_{14} q_{13}	axbbb#	Shift a and goto q_{14}
8.	# a A_1 a A_1 a A_1 a q_0 q_5 q_{13} q_{14} q_{13} q_{14} q_{13} q_{14}	xbbb#	Reduce by $A_1 \rightarrow A_1 a$
9.	# a A_1 a A_1 a q_0 q_5 q_{13} q_{14} q_{13} q_{14}	$(^*)A_2$xbbb#	Reduce by $A_1 \rightarrow A_1 a$
10.	# a A_1 a q_0 q_5 q_{13} q_{14}	$(^*)A_1(^*)A_2$xbbb#	Reduce by $A_1 \rightarrow A_1 a$
11.	# a q_0 q_5	$(^*)A_1(^*)A_1(^*)A_2$xbbb#	Reduce $A_1 \rightarrow a$
12.	# A_1 q_0 q_1	$(^*)A_1(^*)A_1(^*)A_2$xbbb#	Cancel $(A_1 \rightarrow A_1\ a)$
13.	# q_0	$A_1(^*)A_1(^*)A_2$xbbb#	Shift A_1
14.	# A_1 q_0 q_1	$(^*)A_1(^*)A_2$xbbb#	Cancel $(A_1 \rightarrow A_1\ a)$
15.	# q_0	$A_1(^*)A_2$xbbb#	Shift A_1
16.	# A_1 q_0 q_1	$(^*)A_2$xbbb#	Cancel $(A_1 \rightarrow A_1\ a)$
17.	# q_0	A_2xbbb#	Shift A_1

18.	# A_1 q_0 q_1	xbbb#	Shift x
19.	# A_1 x q_0 q_1 q_6	bbb#	Reduce $A \rightarrow A_1x$
20.	# A_1 q_0 q_1	bbb#	Reduce $A \rightarrow A_1$
21.	# A_1 q_0 q_2	bbb#	Shift b
22.	# A_1 b q_0 q_2 q_6	bb#	Reduce by $B \rightarrow b$
23.	# A_1 B q_0 q_2 q_8	bb#	Shift b and goto q_{12}
24.	# A_1 B b q_0 q_2 q_8 q_{12}	b#	Reduce by $B \rightarrow Bb$
25.	# A_1 B q_0 q_2 q_8	b#	Shift b and goto q_{12}
26.	# A_1 B b q_0 q_2 q_8 q_{12}	(end of input)#	Reduce by $B \rightarrow Bb$
27.	# A_1 B q_0 q_2 q_8	(end of input)#	Reduce by $S \rightarrow AB$
28.	# S q_0 q_4	(end of input)#	Accept

تحسين ستراتيجية الإعراب LR بأستخدام طريقة الإعراب التزايدية GPLR

مؤيد عبد هاني ميثاق إبراهيم هاشم

قسم علوم الحاسبات ـ كلية التربية ـ جامعة ذي قار

الخلاصة

إن المعربات في بيئات التطوير المتكاملة للغات البرمجة في اغلبها هي معربات اعتيادية تعمل بالطريقة التكرارية, وبينما لهذه المعربات العديد من المساوئ إذا ما قورنت بالمعربات التطلعية المولدة بالآلة LALR إلا أنها تملك الأفضلية عليها لأنها تملك ميزتين مهمتين رئيسيتين هما: ١- إن هذه المعربات لا تتقيد بأي عدد محدد من رموز التطلع ٢- انه لا تعيد إعراب الملف بأكمله إنما تعيد إعراب جزء منه إذا تطلب العمل ذلك, من الناحية النظرية فانه بالإمكان تحقيق هاتين الميزتين باستخدام معربات LR التقليدية وتقنياتها, إلا إن معربات LR التقليدية تعاني من مشكلتين رئيسيتين هما ١- مشكلة shift-reduce conflict ٢- مشكلة reduce-reduce conflict. في هذا البحث, حاولنا حل هاتين المشكلتين مع الحفاظ على الخصائص الجيدة لمعربات LR التقليدية وتقنياتها وقد تم ذلك باستخدام تقنية GPLR بدلا من تقنيات LR التقليدية.