A Practical GLR Parser Generator for Software
Reverse Engineering

Teng Geng, Fu Xu*, Han Mei, Wei Meng, and Zhibo Chen
School of Information Science and Technology, Beijing Forestry University, Beijing, China
*Corresponding author, Email: xufu@buaa.edu.cn
Changqing Lai
Information Center of National Defense Science and Technology Industry Bureau, Beijing, China

Abstract—Traditional parser generators use deterministic parsing methods. These methods can not meet the parsing requirements of software reverse engineering effectively. A new parser generator is presented which can generate GLR parser with automatic error recovery. The generated GLR parser has comparable parsing speed with the traditional LALR(1) parser and can be used in the parsing of software reverse engineering.

Index Terms—Parser Generator; Parsing; Error Recovery; GLR; Reverse Engineering

I. INTRODUCTION

Automated deterministic parsing has been widely used in many fields since the introduction of LALR(1) parser generator YACC [1]. However, with the development of software reverse engineering, deterministic parsing has exposed more and more limitations:

(1) Currently used parsing methods such LL(k) and LR(k) can only recognize deterministic grammar. Deterministic grammar is only a restrictive subset of context-free grammar. When using these deterministic parsing methods, grammar rewriting must often be done. Grammar rewriting obscures or destroys the original conceptual structures and will lead to potential maintenance difficulties. “Tremendous expertise is required to build tools for many popular languages such as C++ and Ada, and few programmers are able to do so” [5].

(2) Deterministic parsing can not tolerate the conflicts in parsing tables. In fact, many conflicts in programming languages are only local ambiguities which can be resolved by using non-deterministic parsing. When using deterministic parsing, developers have to use many ad hoc ways to resolve these conflicts. This brings lots of workload and difficulties.

(3) Deterministic grammar is not closed under composition [6]. In other words, combining two deterministic grammars maybe gets a non-deterministic grammar. This isn’t a severe problem in compiler construction, because one compiler usually recognizes only one language. However, in software reverse engineering field, multi-lingual parsing is a universal requirement. It is very difficult to construct multi-lingual parser when using traditional parsing methods.

Comparing with deterministic grammar, we think context-free grammar is more suitable for the parsing of software reverse engineering. Accordingly, an algorithm which can parse context-free grammar should be used. Possible candidates include: Unger [7], CYK [7], Earley [8] and GLR [3].

Unger and CYK are not suitable because they have high time and space complexity. Moreover, these methods don’t have fixed scanning direction and the entire source must be read in before parsing. This determines that they can not be used for the parsing of large legacy systems.

An ideal algorithm should satisfy the following conditions: (1) Can parse any context-free grammar; (2) Scan the input string from left to right. (3) No backtracking in parsing process. Earley and GLR algorithms fit these requirements.

Earley and GLR are mainly used in the processing of natural languages. They can parse any context-free grammars including those which are ambiguous or need infinite lookahead. Earley and GLR have the same worst-case time complexity of O(n³) and have the time complexity of O(n) for LR(k) grammar. If the grammar is highly ambiguous, the time complexity of GLR is higher than Earley. If it is not, the time complexity of GLR is lower than Earley [7]. Allowing that there are usually very limited conflicts in the grammars of programming languages, the efficiency of GLR should be better in this case. Comparing with the other context-free parsing methods, we think GLR is more suitable for the parsing of software reverse engineering.

GLR is rarely used in the parsing of software reverse engineering before. Low parsing efficiency and no effective error recovery are two important reasons. This research presented an optimized GLR algorithm with effective error recovery support. The main contributions are as follows: (1) Optimized the GLR algorithm from parsing tables and graph operations. The generated GLR parser has comparable parsing speed with the traditional LALR(1) parser. (2) Added error recovery mechanism for
GLR algorithm and can detect parsing errors and make error recovery effectively.

II. GLR ALGORITHM OPTIMIZATION

GLR is typically a factor of ten or more slow than its LR counterpart [9]. The bottlenecks come from two aspects: the split of graph stack and the complicated operations of shift/reduce actions. We optimized GLR algorithm from the following two aspects.

A. Decrease the Split of Graph Stack

Graph stack will split on meeting conflicts. This will lower parsing speed sharply. There are two methods to lower conflicts: rewriting grammar and using strong ability parsing table.

Grammar rewriting is not an appropriate way for the parsing of software reverse engineering. Just as it is pointed out before, grammar rewriting needs high expertise and it destroys the original conceptual structures which will lead to maintenance difficulties. While maintenance is an important indicator for the parsing tools in software reverse engineering.

Using strong ability parsing table can decrease conflicts too. In the commonly used parsing tables, LR(1) has the strongest parsing ability, LALR(1) is the next, SLR(1) is the third and LR(0) is the last. Obviously strong ability parsing table such as LALR(1) contains lesser conflicts than weak ability parsing table such as LR(0). Increasing the look-ahead length \( k \) can decrease the parsing conflicts too. In theory, GLR can use any LR parsing table. Which parsing table and what lookahead length should be used?

Most GLR implementations use LR(0) parsing table at present. This is because that GLR described in literature is mainly used for parsing natural languages. The grammars of natural languages have two significant features: large scale and highly ambiguous. These features determines that LR(0) table is more proper because it has the least states and the minimum memory occupation, thus it has better efficiency for large-scale automation. The strong parsing ability of SLR(1), LALR(1) and LR(1) come from their lookahead. Typically the grammars of natural languages are highly globally ambiguous which can not be resolved by using one or more lookahead. Comparing with LR(0), using strong ability parsing table such as LALR(1) usually does not gain advantages for parsing natural languages [11].

Reverse engineering mainly parses programming languages which are different from natural languages. The grammars of programming languages are usually small-scale and have very limited conflicts. A typical programming language only has several hundred LALR(1) states. Even for the notorious C++ grammar, there is only 1200 states in its LALR(1) parsing table [12]. Most conflicts in programming languages are local ambiguities which can be resolved by using one or more lookahead.

T.J. Parr investigated the lookahead length of 22 programming languages and got the following conclusions: (1) About 61% decisions did not require lookahead \( k=0 \); (2) 98.8% decisions only required one lookahead \( k=1 \); (3) If \( k=n(n+1) \) could not make the parsing decision, usually \( k=n \) could not too [13]. This explains that \( k=1 \) is the most suitable lookahead length for parsing programming languages.

Using strong ability parsing table instead of LR(0) should gain better performance for the GLR parsing of programming languages. We think LALR(1) is the most suitable parsing table because its parsing ability is close to LR(1), while its parsing table size is the same with LR(0) and far less than the size of LR(1). To accelerate the construction of LALR(1) parsing table, a fast generation algorithm was also designed and implemented in our parser generator [15].

Besides using LALR(1) parsing table, our optimized GLR algorithm also supports interactive user-defined conflict resolving routine. User usually knows which branch should be used according to the parsing context and lookahead when meeting a conflict. Thus he can make pruning from multiple decisions to choose the correct branch. Figure 1 is the pseudo code of interactive conflict resolving routine. In Figure 1, italic parts are comments and all the others parts are valid code.

```c
for current stack top and lookahead, there are more than one actions. This means that the parser meets a conflict

if (ACTION[topst, lookahead] > 1)
{
    // If user has defined a conflict resolving routine, callback this routine and pass in the stack top and lookahead as the parameters
    selected = UserSolving(topst, lookahead);
    switch (selected)
    {
        case R:
            DoStake rule R; /* reduce using rule R */
            break;
        case S:
            DoShift (state i); /* shift current symbol and break; move to state i */
            /* other actions are omitted here */
            default:
                DoAll (topst, lookahead, /* do all possible actions */
}
```

Figure 1. Pseudo code of interactive conflict resolving routine

B. Optimize Shift/Reduce Actions

The high price of standard GLR shift and reduce actions is another performance bottleneck. It was optimized in our algorithm according to the characteristics of programming languages.

1) Optimization of Shift Action

When shifting a state, GLR has to check whether the state already exists in the state set of active stack tops. If it already exists, the parsing stacks should be combined. Allowing that there are only very limited conflicts in most programming languages, graph stack does not split and is a linear stack in most of the execution time. In this case the state can be shifted directly without the need of making above checking and combining. Depending on whether the graph stack has split, the standard GLR shift can be divided into two actions: (a) **direct shift**: shift directly without checking states or combining stacks; (b)
standard shift: execute standard GLR shift action, check each shifted state and combine parsing stacks if necessary.

2) Optimization of Reduce Action

For most programming languages, the graph stack satisfies the following one or two conditions in most of the execution time: (a) The graph stack is a linear stack without split and there is only one active stack top state; (b) For an active stack top state \( n \) and input symbol \( t \), there is a unique reduce action:

\[
ACTION[n, t] = R_{\alpha \rightarrow \beta} \tag{1}
\]

The right-hand length \( |\alpha| \) fit the following condition:

\[
|\alpha| \leq j - i \tag{2}
\]

\( j \) is the synchronous position of stack top state node, and \( i \) is the synchronous position of the branch node which is nearest to node \( n \).

The synchronous position is the symbols count read by each parsing step. Figure 2 is a sample. The synchronous positions are already labeled out on the top. The branch nodes are those whose in-degree or out-degree is greater than one. For example, in Figure 2, nodes 0, 4, 5 are all branch nodes.

![Figure 2. Synchronous position](image)

The standard GLR reduce action using rule:

\[
A \rightarrow \alpha \tag{3}
\]

can be classified into three types: (1) **direct reduce**: If graph stack satisfies the above condition (a), execute direct reduce. Pop \( |\alpha| \) states from parsing stack, then push the new state into stack. (2) **simple reduce**: If graph stack does not satisfy the above condition (a) but satisfies the condition (b), execute simple reduce. Pop \( |\alpha| \) states from stack, then push the new state into stack. (3) **standard reduce**: If graph stack satisfies neither (a) nor (b), execute standard GLR reduce. In this case, checking and maintaining reduce path will be made which is a high-price operation.

After making the above optimizations, GLR will execute direct shift and direct reduce when graph stack is linear without any conflicts. Even if a few conflicts are met during parsing, graph stack still satisfies the above condition (a) or (b) in most of the time. GLR will mainly execute low-price direct shift, direct reduce or simple reduce in most of the execution time. This can improve the parsing efficiency of standard GLR shift/reduce actions significantly.

### III. GLR Error Recovery

Error recovery tries to continue parsing by adjusting parsing stack or input symbols. It is an important feature of a practical parser. As far as we know, DMS was the only parser generator which claimed supporting GLR error recovery [16]. DMS was a business property software and we had not found any literature to discover its error recovery mechanism.

Generally speaking, there are two types of error recovery methods: correcting method and non-correcting method. The former method modifies the parsing stack or input symbols. While the latter does not make such modifications. Instead, it skips some input symbols to continue parsing.

If using correcting method, it will be unavoidable to modify the complicated GLR graph stack or input symbols. Non-correcting method only skips input symbols and does not modify graph stack or input symbols. This can avoid the high price of modifying complicated GLR graph stack.

Panic mode error recovery is a commonly used non-correcting recovery method. It is easy to implement but has obvious limitations. Its recovery effect depends on the quality of synchronous tokens and requires high expertise. It is difficult to implement automatic error recovery. Richter [17] introduced another non-correcting method which had the following features: (1) Didn’t report duplicated errors; (2) Didn’t report false errors; (3) Language independent. We considered Richter’s algorithm could be used in the error recovery of GLR parser.

#### A. The Limitations of Richter’s Algorithm

Richter’s algorithm has the following limitations:

1) **Error Swallowing**

In Richter’s algorithm, a previous error maybe swallows the continuous errors. Figure 3 is a multiple line comment of C programming language. If “/\*” is omitted by mistake, all the subsequent symbols will be recognized normally until “/\*” is met. In this case, omitting “/\*” will swallow all the subsequent errors. To fix this problem, a new property disable_substring is introduced in our algorithm which will disable substring matching for the grammar rule which has this property.

```
/* This is a multi-line comment in C programming language */
```

![Figure 3. Multi-line comment of C programming language](image)

2) **Infinite Loop**

For the following grammar rule:

\[
A \rightarrow \alpha \beta \tag{4}
\]

Of this, \( \alpha \) and \( \beta \) represent any combinations of terminals and non-terminals, but they can not be nullable simultaneously. If only \( \alpha \) is recognized, reduce maybe causes Richter’s algorithm running into infinite loop. We fixed this problem by disabling reduce in this case.

3) **No Runtime Control for GLR Error Recovery**

Richter’s algorithm used LR parsing method, which lacks the necessary control for GLR error recovery. It is
necessary to add runtime control mechanism to support GLR error recovery.

B. GLR Error Recovery Algorithm

Algorithm 1 is our GLR error recovery method. The step (1)-(2) are straightforward and step (3) can be processed by algorithm 2. In algorithm 2, the beginning numbers are line numbers. Parts beginning with // are comments and all the other lines are valid code.

Algorithm 1. GLR error recovery algorithm
Input: Grammar G = (N, T, P, S) and input symbols
Output: The parsing tree or parsing forest for input symbols
Steps:
(1) Generate LALR(1) parsing table M, GLR parser P and substring parser P_i for grammar G_i;
(2) Using GLR parser P to parse the input symbols and end the parsing if no syntax error is found. If an error is found, start substring parser P_i to parse the remaining symbol string (let it be S_err);
(3) Execute algorithm 2 using substring parser P_i with S_err as the input symbols.

Algorithm 2. GLR substring parsing algorithm
Input: Grammar G = (N, T, P, S), the LALR(1) parsing table M of grammar G and the input symbols S_err
Output: The parsing tree or parsing forest for S_err

Pseudo code:
1 void SubstringRecognize()
2 {
3 Integer ShiftCount=0 // zero shift symbols counter
4 currentToken=First_Symbol(S_err) // read the first symbol
5 // look up all the states which can shift current symbol
6 Array InitialShiftStates=LookupShiftState(currentToken)
7 while (currentToken!=EOS and // not reached the end
8 InitialShiftStates==null) // can’t shift current symbol
9 InitialShiftStates=LookupShiftState(currentToken)
10 // create a new parsing stack
11 StackNode newNode=new StackNode(ERROR_STATE)
12 Add.newNode(newNode, // add a directed edge from new Node to target with
13 current error symbol as the semantic value
14 ShiftCount++ // increase shift symbols counter
15 currentToken=Next_Symbol(S_err) // look next symbol
16 if (currentToken==EOS or InitialShiftStates==null)
17 // can’t shift current symbol
18 InitialShiftStates=LookupShiftState(currentToken)
19 InitialShiftStates=LookupShiftState(currentToken)
20 // re-look up states which can shift current symbol
21 if (InitialShiftStates=LookupShiftState(currentToken)
22 Reduce ErrSymbols(target, ShiftCount)
23 // reduce for all error symbols
24 Reduce ErrSymbols(target, ShiftCount)
25 // if all symbols are error symbols or there is no state
26 if (InitialShiftStates=LookupShiftState(currentToken)
27 return directly
28 if (currentToken==EOS or InitialShiftStates==null)
29 // which can shift current symbol, return directly
30 if (currentToken==EOS or InitialShiftStates==null)
31 // fetch each state m from above state set
32 for each state m in InitialShiftStates
33 create parsing stack L with m the stack bottom
34 Use L to parse the remaining input symbols
35 }
36 }
37 Array LookupShiftState(Token t)
38 Look up states which can shift t from parsing table M
39 Return S
40 // GOTO y A z
41 }
42 }

In algorithm 2, except lines 33-34, the other lines are straightforward. The following will explain lines 33-34. Let the remaining input symbols are t_1…t_i…t_n, then the current symbol is t_i. Allowing that there may be multiple parsing stacks, to simplify the analysis, let’s take one parsing stack L and its stack top is m.

First, L shifts t_i and prepares to parse the remaining symbols t_{i+1}…t_n. For the next symbol t_i (2 ≤ i ≤ n−1), the possible actions can be:
(1) Shift S. Shift t_i and push state S into L and prepare to process the next symbol t_{i+1}.
(2) Error. Delete the stack top of L from the active stack tops and discard parsing stack L.
(3) Reduce using the following rule:

A → αβ

(5)

Of this, α and β represent any combinations of terminals and non-terminals. There are three cases:
(a) αβ is totally in L (formula (6)).

| L | αβ

(6)

There should be a state n which has a GOTO relation n_goto for A. In this case, execute standard GLR reduce. Pop up | αβ | elements from L, push n_goto into L and prepare to process symbol t_i.
(b) αβ is exactly in L (formula (7)).

| L | αβ

(7)

Reduce in this case will pop up | αβ | elements and empty parsing stack L. Looking up parsing table M can find the states set y which fit GOTO(y,A)=z. Richter’s algorithm will create a separate parsing stack for each state in this case. This maybe leads to parsing branch explosion and decreases the parsing speed significantly. We fixed this problem through the following two methods: (i) If αβ contains A, check whether the other parts of αβ except A can derive empty string. If it can, reduce in this case will cause infinite loop and should disable reduce. If it can not, execute the following steps. (ii) Check whether user defines an interactive error recovery routine. If there is, use it to prune the parsing stack. Otherwise, create a new parsing stack for each state y and use algorithm 2 to parse the remaining symbols t_{i+1}…t_n.

(c) Only part of αβ is in L (formula (8)).

| L | < αβ

(8)

This includes tree cases: (i) Only β is recognized. Reduce will recognize the prefix of the input symbols. That is a_{i-1}…a_1 (a_0 is the missing input symbols). (ii) Only α is recognized. Reduce in this case will recognize the postfix of the input symbols. That is t_{i+1}…t_n(a_0…a_i is the missing input symbols). (iii) Only the middle of αβ recognized. Reduce in this case will recognize the prefix and postfix of the input symbols. That is a_{i-1}…a_0…a_k (a_k and b_{i+1}…b_n are the missing input symbols). In this case, only part of αβ is recognized which may lead to error swallowing problem, so that disable_substring property should be checked.
Step (c) will empty parsing stack $L$ and cause the similar problem of (3)-(b). This can be resolved using the following method. Check whether $\alpha\beta$ contains $A$. If it does and only $A$ is recognized, reduce will lead to infinite loop and should disable reduce. If $\alpha\beta$ does not contain $A$, execute the similar processing of step (3)-(b).

The pseudo code of step (3) is as follows:

```c
bool SubstringReduce(Production prod, Stack stack)
|
if (stack[prod]) // prod is totally in stack
|
return Standard_GLR_Reduce // standard GLR reduce
|
else if (stack[prod] and Right (prod) contains A)
|
if (prod has disable_substring property)
|
return false // disable reduce
|
// get the left nonterminal symbol of prod
Nonterminal A = Left (prod)
// prod is exactly in stack and the right part of prod
// can deduce A
if (stack[prod] and Right (prod) contains A)
|
// If the left and right parts of A in prod can deduce
// empty string, reduce should be disabled
if (Nullable (Left A) and Nullable (Right A))
|
return false // disable reduce
|
// If prod is partly in stack and the right side of prod contains A
// and only A is recognized, reduce should be disabled
if (stack[prod] and Right (prod) contains A and only A recognized)
|
return false // disable reduce
|
// find all the states which have GOTO relation with A
StateSet allStates = LoopupGOTO(A)
// call user-defined error resolving routine if it is defined
StateSet filterStates = UserFilterCallBack(prod, stack)
empty stack
for each State m in filterStates
|
Create new stack $L$ with $m$ as the stack bottom
Use $L$ to parse the remaining input symbols
|
return true // accept and execute reduce
```

IV. EXPERIMENTS

Two groups experiments were done to test the optimized GLR algorithm and error recovery algorithm.

A. Experiment of Optimized GLR Algorithm

To verify the performance of the optimized GLR algorithm and evaluate its practicability in software reverse engineering, we generated a GLR parser using our parser generator VPGE and an LALR(1) parser using Bison which is the most widely used LALR(1) parser generator at present. Both parsers were used to parse Java 1.4 grammar. The official Java 1.4 grammar was written in EBNF form and we rewrote it to BNF form which included 216 rules. For this grammar, Bison reported 235 conflicts and 317 LALR(1) states. Because LALR(1) parser can not parse conflict grammar, we made grammar rewritings to eliminate the conflicts. The resulted grammar contained 278 rules and Bison reported 0 conflicts and 448 LALR(1) states.

Choosing Java grammar has two reasons. One reason is that the LALR(1) parser generated by Bison can not tolerate conflicts, while Java 1.4 grammar can be rewritten to LALR(1) grammar exactly. The other reason is that, Java grammar is middle-sized in programming languages which has the typical features of imperative programming languages.

Both parsers only made syntax parsing and did not execute semantic actions. Table I is the experimental result (Experimental PC settings: AMD Athlon XP 2500+, 1G RAM, Windows 2000 Advanced Server). It can be seen that, the GLR parser is 2~5 times slower than the LALR(1) parser. In a practical parser, the time ratio of syntax parsing is usually very low and most of the time is spent in semantic actions [4]. While semantic actions of both parsers can be considered the same. It can be derived that, if the grammar is LALR(1) or almost LALR(1) and there are only very limited conflicts in its parsing table, the total speed of VPGE GLR parser and Bison LALR(1) parser is comparable.

Table: The Performance Comparison of VPGE GLR Parser and Bison LALR(1) Parser

<table>
<thead>
<tr>
<th>Sample</th>
<th>Lines</th>
<th>Tokens</th>
<th>VPGE (clocks)</th>
<th>Bison (clocks)</th>
<th>VPGE/Bison</th>
</tr>
</thead>
<tbody>
<tr>
<td>iijmnik</td>
<td>53495</td>
<td>58468247</td>
<td>5.191</td>
<td>3.540</td>
<td></td>
</tr>
<tr>
<td>ekmez</td>
<td>8916</td>
<td>5410</td>
<td>2.972</td>
<td>2.864</td>
<td></td>
</tr>
<tr>
<td>iij</td>
<td>88943</td>
<td>461722165</td>
<td>5.191</td>
<td>5.381</td>
<td></td>
</tr>
<tr>
<td>ik</td>
<td>60077</td>
<td>88468247</td>
<td>5.191</td>
<td>5.381</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>1631176</td>
<td>2349846</td>
<td>2.972</td>
<td>2.864</td>
<td></td>
</tr>
<tr>
<td>iik</td>
<td>9815220</td>
<td>24398177</td>
<td>2.972</td>
<td>2.864</td>
<td></td>
</tr>
</tbody>
</table>

Tomita’s original GLR algorithm could not parse some $\epsilon$-production grammars and Nozohoor-Farshi fixed this problem [10]. Rekers’ algorithm was an important GLR implementation which added the capability of cyclic production processing, optimized the construction of parsing tree and the generation of parsing table. It was reported that Rekers’ GLR parser was 10-20 times slower than YACC’s LALR(1) parser [9]. As the back-compatible and improved version of YACC, Bison’s performance had exceeded YACC. It can be derived that, the performance of our optimized GLR implementation has exceeded Rekers’ implementation.

B. Experiment of Error Recovery Algorithm

Figure 4 gives a grammar and Table II it is its LALR(1) table. From Table II it can be seen that state 5 has reduce-reduce conflict. Grammar G[S'] can recognized symbols $i$, $j$, $k$ and $ijk$ is its valid sentence. Embedding two types of errors into $ijk$ can get the string $iijmnik$. Of this, $mn$ can not be recognized by G[S'] and $ik$ can be recognized but it is invalid. Steps (1)-(3) is the execution process.

(1) Execute regular GLR parsing and the resulted graph is Figure 5 ($iij$ is recognized). Of this, the square boxes represent states, the arrows between states represent the construction relation of states, and the attached values on arrow represent the semantic values.
(2) Execute reduce and shift actions. No reduce actions can be made and no state can shift \( m \) or \( n \). The substring parser is started to make error processing. Execute lines 8-20 of algorithm 2 two times and the unrecognized symbols \( mn \) is added into the graph stack (Figure 6, \( mn \) is processed).

![Figure 4. Grammar G[S']](image)

<table>
<thead>
<tr>
<th>Table II. LALR(1) Parsing Table of G[S']</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ACTION</strong></td>
</tr>
<tr>
<td><strong>States</strong></td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>17</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>19</td>
</tr>
<tr>
<td>20</td>
</tr>
</tbody>
</table>

![Figure 5. The graph stack of regular GLR parsing](image)

(3) In algorithm 2, currentToken = \( i \) and InitialShift-States = \( \{0, 2, 10, 12, 15\} \). Create five parsing stacks (a)-(e) for InitialShiftStates and make substring parsing for these stacks (Figure 7).

![Figure 7. The graph stacks of (a)-(e)](image)

Graph stacks (a), (b), (d) can parse all input symbols. Graph stacks (c) and (e) can not and exit intermediately.

![Figure 6. Add unrecognized symbols into graph stack](image)

The essence of error recovery is choosing the most reasonable branch from multiple decisions. If user knows which branch should be used, an interactive error recovery routine can be used to prune illegal branches. If no user-defined interactive routine, our algorithm will choose the most reasonable branch automatically. The
reasonable degree is measured by the parsed symbols count. The more count, the better degree. If two branches have the same parsed symbols count, choose the first branch met in parsing. According to the above rules, graph stack (a) should be adopted. The final parsing result is illustrated in Figure 8.

From the above analysis, it can be seen that our error recovery algorithm can detect syntax errors and make error recovery effectively.

V. CONCLUSION

The deterministic parsing methods such as LL(k) and LR(k) have dominated the parsing of programming languages for a long time. With the development of software reverse engineering, these deterministic parsing methods face some new difficulties such as complicated language parsing, multi-lingual parsing, error recovery, etc [18] [19] [20] [21]. This research adopted context-free grammar as the underlying grammar modal and used GLR as the parsing method. An optimized GLR algorithm with effective error recovery was designed and implemented. The experimental results showed that the optimized GLR algorithm had comparable parsing speed with the traditional LALR(1) parser and also it had good error recovery ability. Comparing with the traditional parser generators, the new parser generator needs less grammar expertise for developers and can fit the parsing requirements of software reverse engineering effectively.

It should be noted that, although our parser generator had already generated a few parsers for some popular programming languages such as C and Java, it did not endure the testings of complete ISO C++ and large legacy systems. We will continue to improve this parser generator and hope it can put into use in the future. Some of the future work include: (1) Design and implement a valid and universal modular definition language to support multi-lingual and multi-dialect languages processing; (2) Research and design an incremental generation algorithm to accelerate the parsing of large legacy systems; (3) Provide better support for interactive breakpoint debugging.

REFERENCES


Teng Geng was born in ShanDong Province of China in 1990. He is going to graduate and receive his master degree in computer software and theory in Beijing Forestry University in 2015. Currently he is a three-year graduate student of Beijing Forestry University.

Fu Xu was born in Shandong Province of China in 1979. He received his Ph.D degree in computer software and theory from Beihang University in 2009. Now he is a lecture in Beijing Forestry University, and his main research fields are compilers, software engineering and telecommunications.

Han Mei was born in Hebei Province of China in 1989. He is going to graduate and receive his bachelor degree in computer science major in Beijing Forestry University in 2014. Currently he is a four-year undergraduate of Beijing Forestry University.
Wei Meng was born in Shandong Province of China in 1974. Now she is an associate professor in Beijing Forestry University, and her main research fields are intelligent information processing.

Zhibo Chen was born in Shandong Province of China in 1967. Now he is a professor in Beijing Forestry University, and his main research fields are database system and internet of things.

Changqing Lai was born in Sichuan Province of China in 1981. Now he is a senior engineer in Information Center of National Defense Science and Technology Industry Bureau, and his main research fields are e-government and software engineering.