A Powerful LR(1) Error Recovery Mechanism in the Compiler Implementation System Environment

Mariza A. S. Bigonha[1]
Roberto S. Bigonha [2]

Abstract

This paper presents a scheme for error recovery in the context of LR(1) parsers based on the method proposed by Burke-Fisher [3]. The purpose of this method is the diagnosis of all syntactic errors found during the syntactic analysis without presenting misleading messages. The recovery process automatically issues error messages with the possibility of substitution, deletion or insertion of terminal or nonterminal symbols in the parser stack or in the input stream. The error recovery is conducted before and independently of any semantic analysis of the program. Nevertheless, the approach does not perclude the use of semantic information in the process of error recovery. This automatic syntactic error recovery strategy is integrated into the parser generator system developed at UFMG (SIC [12, 11, 10]). It is entirely transparent to the user and it produces good quality results without increasing significantly the size of the compiler. This method does not introduce any constraints on the use of default reductions.

Key Words

Compilers, Compiler Generators, LR Parser, Syntactic Error Recovery.

[1]DSc (PUC/RJ - 1994). Department of Computer Science, Federal University of Minas Gerais, Belo Horizonte - MG - Brazil, E-mail:mariza@dcc.ufmg.br

[2]PhD. (UCLA/USA 1981). Department of Computer Science, Federal University of Minas Gerais, Belo Horizonte - MG - Brazil, E-mail: bigonha@dcc.ufmg.br.

The Powerful LR(1) Error Recovery Mechanism in the Compiler Implementation System Environment

Abstract

This paper presents a scheme for error recovery in the context of LR(1) parsers based on the method proposed by Burke-Fisher [3]. The purpose of this method is the diagnosis of all syntactic errors found during the syntactic analysis without presenting misleading messages. The recovery process automatically issues error messages with the possibility of substitution, deletion or insertion of terminal or nonterminal symbols in the parser stack or in the input stream. The error recovery is conducted before and independently of any semantic analysis of the program. Nevertheless, the approach does not perclude the use of semantic information in the process of error recovery. This automatic syntactic error recovery strategy is integrated into the parser generator system developed at UFMG (SIC [12, 11, 10]). It is entirely transparent to the user and it produces good quality results without increasing significantly the size of the compiler. This method does not introduce any constraints on the use of default reductions.

Key Words

Compilers, Compiler Generators, LR Parser, Syntactic Error Recovery.

1 Introduction

Before the advent of interactive systems for development of programs, the availability of some syntactic error recovery method in commercial compilers was essential. This occurred because it is unacceptable for a compiler running in batch mode to abort the compilation process in the presence of the first syntactic error. With the increasing popularity of interactive systems, it becomes reasonable to have a compiler without syntactic error recovery mechanism. Compilation can then be interrupted after the detection of the first error, and the user provides the necessary corrections and then reinitiates the compilation. If the compiler is sufficiently fast, the cost of recompilation is perfectly acceptable. However, this approach presents as a drawback the fact that the compilation can not continue even when the user thus desires. In addition, the absence of mechanism of syntactic error recovery in the compiler affects negatively the quality of its error messages. In summary, even though there exist applications where it is possible to have compilers without syntactic error recovery, the availability of this mechanism is certainly very helpful, because at least it makes possible to improve the quality of the messages issued and makes the operations of these systems more flexible.

With the objective of developping an automatic syntactic error recovery mechanism which is at the same time efficient, practical and applicable within the general context of syntactic analysis based in the viable prefix property [1], we have studied several approaches [3, 14, 15, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27]. From the study of these methods, we decided, as a starting point, to use Burke & Fisher's method, which was adapted in order to fit in a LR(1) syntactic analysis based on table compression [13, 9].

2 General Description

Burke-Fisher's method supposes a context where the LR parser maintains an input buffer and two stacks, called PS and PE. The buffer may have part or all of the input stream of terminal symbols not yet processed, and possibly stack symbols. The PE stack is used as a scratch store. The PS stack is used to store parser states, such that the current state is always on its top.

The error recovery routine is activated when, given the current state and the next input terminal symbol, there is no legal syntactic action to be done. In this case, the terminal symbol or current symbol is the one that caused the error detection. The heart of this approach is to determine the nature of the error. A simple error is one which can be corrected by changing only one symbol of the input stream. This modification of the input may be an insertion, a deletion of a symbol or a substitution of a terminal symbol by another one. This type of correction is called a simple correction. If the error is not simple, its correction may involve the deletion or insertion of a small piece of program. The program piece removed may precede, follow or be around the symbol flagged as error. The symbol inserted consist of a sequence of terminal symbols that are needed to close one or more scopes. This kind of recovery is called scope recovery. Scope means nested constructions, such as procedures, blocks, control structures and parenthesized expressions. Scopes are delimited by opening and closing symbols, which are symbols that begin and finish, respectively, syntactic construct. For instance, the pairs ("(" , ")"), ("begin" , "end"), ("if" , "end if") are typical scope delimiters.

Error recovery of Burke & Fisher is composed of three phases. The first one treats simple corrections. The second one treats correction made by scope recovery. In the third phase, the

correction is made by removing a piece of program code in conjunction with insertion, deletion or substitution of a terminal symbol. Before beginning each phase, a copy of the contents of PS is saved into the scratch stack PE.

2.1 First Phase of Error Recovery

The process of trying to correct an error in a given point of the program is called a trial. The corrections or strategies tested in one trial can be insertion of a symbol, deletion of a symbol and replacement of a symbol by another. On insertion, a terminal or nonterminal symbol may be placed before the current one. On substitution, only the current terminal symbol may be removed.

Initially, the error recovery mechanism tries to insert, delete or replace only a single symbol. However, if there is no possible correction, symbols are popped of the PE stack and put back in the input buffer. For each one of these symbols the recovery mechanism tries the three strategies described above. This process continues until a correction is found or an scope opening symbol appears in the PE stack top. In this case, the scope opening symbol found marks the end of a syntactic construct that has been detected as syntactically wrong. As the text to be corrected is inside this piece of code, there is no need and nor it is desirable to pop more elements off the stack.

The set of symbols which will be used as candidates for insertion or substitution are those that can be read from the current parser configuration, i.e., the lookahead symbols of the current state. The approach used to determine whether a correction is successful is based on the distance that the parser advances in the input stream. In this implementation the maximum distance analyzed is given by the constant `MAXCHECK`.

While testing a set of candidates for insertion, all candidates that allow the parser to advance up to the maximum distance `MAXCHECK` are considered valid and reported. However, only one is selected. It is possible that no candidate makes the parser advance a `MAXCHECK` distance. In this case, the candidate chosen, if there is one, is that which makes the parser to advance over more symbols, given that the parser also advances at least the distance `MINCHECK`.

The minimum and maximum distances are defined according to the context in which the error occurs. The minimum distance can not be too small and must be such that the parser is guaranteed to advance through the input stream after the recovery. Since errors can appear near each other, the maximum distance must not be too big. The values 3 for `MINCHECK` and 24 for `MAXCHECK` have produced good results [3, 12].

2.1.1 Strategy 1: Insertion

Suppose that the programmer made a mistake by omitting a symbol. To test this hypothesis the following is done: each lookahead symbol in the current configuration of the LR(1) parser is inserted in the buffer just before the current input symbol and a trial to advance in the syntactic analysis is attempted. If the distance advanced with the insertion of this symbol is greater than or equal to the distance recorded for other lookahead symbols, then it is kept as the best candidate for correction so far. As an example, consider the toy language whose grammar is described in Section 7. The messages produced by the error recovery algorithm is shown in Figure 1.

```
------------------------------------------------------------------
5   P: procedure(i : integer)
6      a: boolean
7      begin
8         i :=  * 1
9      end;
10  Q: procedure(h: integer)
11     j : integer;
12     i : integer;
13     R : procedure(i : integer)
---------------|
***** Symbol "id" inserted before symbol  "*"  on line 8
***** Other possible corrections:
***** Insert  nonterminal symbol "cte"
***** Delete terminal symbol "*"
***** Replace "*" with terminal symbol  "-"
                              Figure 1
------------------------------------------------------------------
```

2.1.2 Strategy 1: Symbol Deletion

In this strategy, suppose that the programmer has made a mistake by writing an extra symbol. To test this hypothesis the following strategy is used: firstly, a test is done to check if the current symbol is a terminal. Only terminal symbols may be removed from the input stream. In the next step, starting with the symbol right after the current one, a trial to advance in the input stream is performed. If, when advancing at least a minimal distance `MINCHECK` is achieved, the symbol is considered as a candidate for correction. Figure 2 illustrates the error messages produced for the syntax analysis of a program piece.

```
------------------------------------------------------------------
18      begin
19         i := j;
20         if h = 0 then P(j)
21         else if h = 1 then then P(i)
22              else R(k)
23      end
24   begin
25      i := 0;
26      j := 1;
27      k := 2;
28      Q(k);
------------------------------|
***** Symbol  "then" deleted just before symbol "id" on line 21
                        Figure 2
------------------------------------------------------------------
```

2.1.3 Strategy 1: Symbol Replacement

In this strategy, suppose that the user has made a mistake by writing a symbol in the place of another. To test this hypothesis, the current symbol is replaced, in turn, by each lookahead symbols, and they are tested as in the case of insertion. For instance, the syntactic analyser

of the program in Figure 3 produces the following error messages corresponding to the error recovery for this strategy.

```
-------------------------------------------------------------------
12  Q = procedure(h: integer)
13      j : integer;
14      i : integer;
15
16  R : procedure(i : integer)
17      b : integer
------|
***** Symbol "=" replaced by symbol ":"  on line 12.
                          Figure 3
-------------------------------------------------------------------
```

2.2 Second Phase of Error Recovery

The second phase of the error recovery mechanism is activated when the first phase is not successful. This phase implements scope recovery, i.e., forces the closing one or more scope opening symbols by inserting appropriate sequences of closing symbols. The set of scope opening symbols and their corresponding scope closing symbols is language dependent and must be defered for each language.

In this strategy, suppose that the programmer made a mistake by forgetting one or more scope closing symbols, which must now be inserted. The closing of one or more scopes is done in the following way: in the first place, the set of possible scope closing symbols is determined for each scope opening symbol in the PE stack. Each one of this closing symbol is a candidate for correction. The parser rejects a candidate if it cannot advance over it. On the other hand, if the parser advances at least a minimum distance, (`MINCHECK`), the closing symbol can be used to correct the text. If the parser advances only over the candidate, then it must recursively try to close the next scope opening symbol present down in the stack. This process is repeated until there are no more scope closing symbols to the given opening symbol. For instance, for the small piece of program in Figure 4, the error recovery mechanism produces, for this strategy, the messages shown.

```
-------------------------------------------------------------------
16   R: procedure(i : integer)
17      b: integer
18      begin
19         i := i + 1;
20         b := 1
21
22   begin
-------------|
***** Symbol "end" inserted just before line 22
      to close symbol "begin" on line 18
                          Figure 4
-------------------------------------------------------------------
```

2.3 Third Phase of Error Recovery

This phase is activated only if the second one has not been successful. This phase resembles the first one described in Section 2.1. The difference between them is that now both the elements from the stack and those from the input buffer are discarded. Thus, this phase does not return poped symbols into the buffer, and it removes terminal and nonterminal symbols from the buffer in an attempt to recover from the error.

Starting with the terminal symbol that caused the error (the current symbol), it is checked whether the parser is able to recover simply by deleting the PE stack top element in conjunction with one of the strategies: insertion, deletion or substitution. If not, then more PE stack elements are popped without putting them back in the input stream. This process is repeated until find a scope opening symbol on top of PE top or a correction occurs. As in the first phase, the scope opening symbol serves as a flag, stopping the popping process from the left context. It is important to notice that, for this phase to be more efficient, the parser is allowed to advance up to the minimal distance `MINCHECK+2`. If no correction is possible with the current symbol, then it is deleted, i.e., the next symbol in the input buffer becomes the current symbol and the stack is reset to the initial configuration. The process is then repeated as described above. Figure 5 illustrates messages issued in this phase of error recovery.

```
-------------------------------------------------------------------
12  Q = procedure(h: integer)
13      j : integer;
14      i : integer;
15
16      begin (* comments are not allowed in this language *)
17          i := j;
18          if h = 0 then P(j)
19          else if h = 1 then P(i)
20               else P(k)
-----------------------------------------------------------------|
***** Text deleted from line 16 column 14 until line 16 column 60
***** Symbol ")" deleted just before symbol "id" on line 16
                          Figure 5
-------------------------------------------------------------------
```

3 The Error Analysis

The process of error recovery requires reading portions of the input stream several times in order to determine the most appropriate correction. When a syntax error is found, the parser must only report the error situation, activating the error recovery mechanism, and then resume the analysis. The semantic routines are never activated during error recovery. In order to have a more efficient syntactic analysis during recovery a new parser was created, whose objective is to determine the distance achieved in each trial of error correction. This parser is essentially a syntactic analysis LR(1) without the mechanism to defer reductions (see Section 4) and with a method for identifying transaction labels eliminated by default reductions.

3.1 Candidate Chosen for Correction

It may happen that more than one symbol or more than one strategy would make the recovery to succeed, as shown in Section 2.1. If the first encountered candidate is a candidate for insertion, deletion or substitution then this strategy is chosen. Other candidates, if they exist, are indicated as other possibilities for error correction. However, if the first candidate is obtained from the scope recovery strategy, all scope closing symbols successfully inserted are reported as possible corrections. In this case, only one candidate is chosen by the method. It has been observed that the best order to attempt to correct a syntax error is: first insertion, then deletion and then substitution [3]. When substitution has preference over insertion, the recovery achieved is in general not as good. Experience showed that changing the order of the three strategies reduces the quality of error recovery and messages generated. The possible cause for this is that the most frequent error is omission of symbols. Nevertheless, more studies are needed on this subject in order to support this claim.

3.2 Insertion and Substitution of Nonterminal Symbols

Insertion and substitution of nonterminal symbols may simplify the recovery. However, only some nonterminal symbols should be considered in order not to degrade the quality of the error messages issued by the compiler. Specifically, only substitution or insertion of nonterminals authorized by the user are permitted.

4 Effects of Error Recovery on the Parser

Since reductions in the parser stack may be done even in the presence of a syntax error in the input stream, and given that default reductions are used by table compactation methods [13], some problems appear:

(a) How to restore the stack parser to the configuration after the last "shift," considering that reductions may have been done prematurely? In other words, how to neutralize the effects of these reductions in order to have a more precise error recovery?

(b) How to recover information about transaction labels eliminated from reduction states in the compacted parse table? The labels of these transactions are necessary on attempts to recover from a syntactic error.

A solution for problem (a) is to defer reductions, as done by Burke and Fisher [3]: during parsing, reductions are deferred until the syntactic analyser is able to read the next input symbol. The technique used to defer reductions will be discussed in detail in Section 4.1.

The solution for problem (b) is to recover the eliminated labels by analyzing the finite state automaton corresponding to the canonical collection of LR(0) items [12]. This is possible because labels of transitions to reductions states [13] are also labels to states transitions that can be reached after a sequence of default reductions. In other words, these labels are symbols that must necessarily be read after the reductions have taken place. Therefore, to recover eliminated labels by the introduction of default reductions, the parser must advance by performing default reductions until it reaches a read state. The labels of transitions which leave all the states

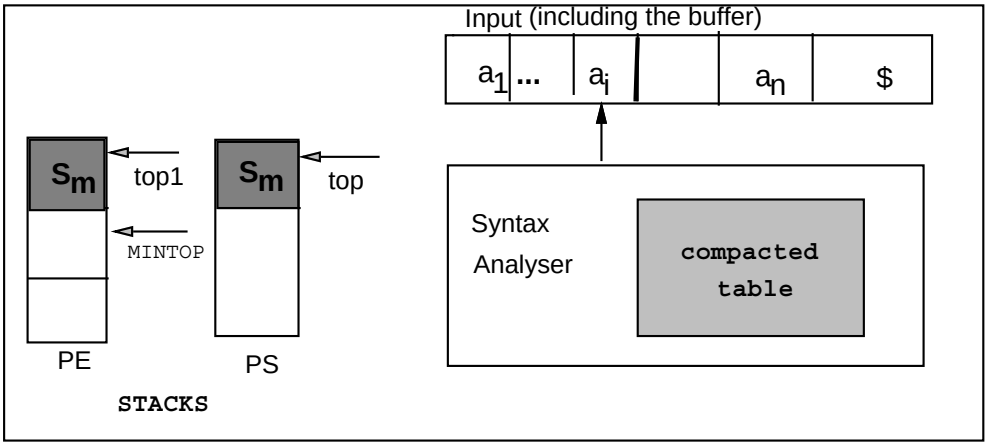

Figure 6: New LR(1) Syntax Analysis

the parser has traversed belong to the set of desired labels. To determine the first read state achieved after default reductions, it is enough to insert in the input one especial terminal symbol not belonging to the language and to activate the error analysis. When a syntax error is detected and the offending symbol is exactly that special symbol, the current state on top of the stack is the desired state and the states visited are exactly the ones desired.

4.1 New LR(1) Parser

The LR(1) parser shown in Figure 6 incorporates the effects of LR(1) table compression and implements the technique of deferring reduction until the next symbol is pushed. The PE stack is used as a scratch stack. As with the PS stack, it is used to store the parser states, so that the current state is on its top. During syntax analysis the PE stack presents exactly the situation of the analysis. On the other hand, the PS stack is updated only when a symbol is pushed into the stack. It presents the stack configuration immediately after the last symbol pushed. So, this stack does not show always the current situation of the analysis.

When a reduction action is detected, the production number which defines the reduction must be properly stored, and the PE stack shows immediately the result of the application of this reduction. However, the reduction is not applied to the parser stack, PS, and the semantic routine is not activated either, i.e., it is deferred.

When a shift action is detected, all reductions stored since the last shift and its respective semantic actions performed on the parser stack PS must be performed so as to make PS equal to PE. It is important to remember that all read symbols are pushed simultaneously on both stacks PE and PS.

When an situation of a syntaxe error is detected, the parser must go back to the configuration that existed at the moment the symbol preceding the terminal which caused the error was pushed, i.e., PE must be restored to the configuration of PS. Making PE equal to PS corresponds to undoing the reductions performed after the last push, so eliminating the effects of the default reductions included.

5 Evaluation of the Error Recovery Strategy

Recently, P. Degano and C. Prianni [29] made a comparison of syntactic error handling in LR parsers. They compared several methods and different techniques, for instance: *minimum distance techniques* [4, 5]; *phase-level recoveries* [19, 18, 14]; *local recovery* [7, 3, 8, 6]; *global recovery* [16, 22] etc.; and *interactive recovery* [17] etc. They consider a correction to be excellent if it repairs the program as a programmer would have. A correction is poor if more spurious errors are introduced.

Regarding *local recovery,* which is the technique we are interested in since we implemented one of these methods, their studies make it clear that the method of Burke-Fisher [2] is characterized by better correction than the other types of techniques. Besides that, in the evaluation of performance degradation due to error-recovery routines they found that the only performance degradation is caused by keeping its auxiliary data structures consistent with those of the parser. The method of Burke-Fisher uses two stacks, as stated before, due of the deferred action mechanism; but they claim that the degradation caused by the use of the second stack is no greater than 10% in their strategy. But, if stack PE (Section 2) is activated only after the detection of the first error, there is no performance degradation. As a consequence of performance degradation more space is needed to store information on the stacks PE and PS because more data structures should be updated. However, the better correction quality of Burke-Fisher's method repays the memory overhead.

6 Conclusion

The method proposed by Burke-Fisher was used as the basis for the implementation of a mechanism of error recovery in the System for Implementing Compilers (SIC) developed at UFMG, [12, 11, 10]. The part of this system which corresponds to error recovery contains 1400 lines of Pascal code. SIC users need only specify the grammar of the language for which they desire to implement a compiler. The user does not have to worry about syntax errors and respective messages. The error recovery and the messages are all generated automatically.

7 Grammar Definition used in the Examples

```
program   = proghead dcls cmdc;
proghead  = "program" ;
dcls      = dcl ;
dcls      = dcls ";" dcl ;
dcl       = "id" ":" "integer"
          | "id" ":" "boolean"
          | prochead "(" par ")" dcls cmdc ;
par       = "id" ":" "integer"
          | "id" ":" "boolean" ;
prochead  = "id" ":" "procedure" ;
cmdc      = "begin" cmds "end" ;
cmds      = cmd
          | cmds ";" cmd ;
cmd       = "id" ":=" exp
          | "id" "(" exp ")"
          | "if" cond "then" cmd "else" cmd
          | "while" cond "do" cmd
          | cmdc ;
cond      = exp ;
exp       = exp "*" exp | exp "+" exp | exp "/" exp  |
          | exp "**" exp | exp "-" exp | "-" exp | "(" exp ")"
          | "id" | "cte" ;
```

References

[1] Aho, Alfred V. and Sethi, R. and Ullman, J. D., Compiler Principals, Techniques and Tools, Addison Wesley Publishing Company, 1986

[2] Burke, M., Fisher Jr., G.A.,, *A Practical Method for Syntatic Error Diagnosis and Recovery*, pages: 164-197, ACM Transactions on Programming Languages and Systems, Vol. 9, No.2, April 1987.

[3] Burke, M., Fisher Jr., G.A.,, *A Practical Method for Syntatic Error Diagnosis and Recovery*, pages: 1-39, ACM 1982.

[4] Aho, A. V. and Peterson, T.J, *A minimum distance error-correction parser for context-free languages*, pages: 305-312, SIAM J. Comput., 1(4), 1972.

[5] Aho, A. V. and Johnson, S.C., *LR parsing*, ACM Computing Surveys, 6(2), 99-124, 1974.

[6] Tai, K. C.,, *Syntactic Error Correction in Programming Languages*, IEEE Trans. Software Engineering, SE-4 (5), 414-425, 1978.

[7] Boullier , P. and Jourdan, M., *A New Error Repair and Recovery Scheme for Lexical and Syntactic Analysis*, Science of Computer Programming, 9, 271-286, 1987.

[8] Charles, P., *An LR(k) Error Diagnosis and Recovery Method*, Second International Workshop on Parsing Technologies, 13-15, February 1991.

[9] Bigonha Roberto S. & Bigonha, Mariza A.S., *A Method for Efficient Compactation of LALR(1) Parsing Tables*, `to be published in 1998`.

[10] Bigonha, Mariza A,S., Bigonha, Roberto S., Russo, Valeska, Costa, Marco, *An Environment for Language Implementation called SIC*, Anais do 2nd. Congreso Argentino de Ciencias de la Computacion, 7-9 de novembro/1996, páginas: 412-424.

[11] Bigonha, Mariza A. S., Bigonha, Roberto, S., *SIC – Uma Ferramenta para Implementação de Linguagens*, Trabalho vencedor do III Prêmio Nacional de Informática 1988, Categoria Software, Anais do XXI Congresso Nacional de Informática, SUCESU, Rio de Janeiro, RJ, 426–431, 1988.

[12] Bigonha, Mariza A. S., *SIC: Sistema de Implementação de Compiladores*, Tese de Mestrado, Departamento de Ciência da Computação, UFMG, julho/1985.

[13] Bigonha Roberto S. & Bigonha, Mariza A.S., *Um Método de Compactação de Tabelas LR(1)*, Anais do III Seminário sobre Desenvolvimento Software Básico, Sociedade Brasileira de Computação, Rio de Janeiro, RJ, 141–157, 1983.

[14] Druseikis, F. C., Ripley, G.D., *Error Recovery for Simple LR(k) Parsers*, Proc. Nate. Conf. of ACM, Houston, TX, 1976.

[15] Feyock, S., Lazurus, P., *Syntax-directed Correction of Syntax Errors*, Software Practice and Experience, Vol. 6, 1976.

[16] Fisher C. N. and Mauney, J., *Determining the Extent of Lookahead in Syntactic Error Repair*, ACM TOPLAS, 10 (3), 456-469, 1988.

[17] Wilcox, T. R., *The Design and Implementation of a Table-driven Interactive Diagnostic Programming System*, CACM 19 (11), 609-616, 1976.

[18] Graham, S., Rhodes, S.P., *Practical Syntatic Error Recovery*, CACM, November 1975.

[19] Graham, S., Haley, C.B., Joy, W.N., *Practical LR Error Recovery*, Sigplan Notices, August 1979.

[20] Johnson S. C., *YACC - Yet Another Compiler Compiler*, Bell Laboratories, Murray Hill, 1977.

[21] Krol, J.S., *Simple Error Recovery Scheme for Optimized LR Parsers*, TR 81-456, March 1981.

[22] Levy, J. P., *Automatic Correction of Syntax Errors in Programming Languages*, Acta Informatica 4, 271-292, 1979,

[23] Mickunas, M.D., Modry, J.A., *Automatic Error Recovery for LR Parsers*, CACM, june 1978, Vol. 21, Number 6.

[24] Poonen, G., *Error Recovery for LR(k) Parsers*, Information Processing, August 1977.

[25] Rohrich, Johannes, *Methods for the Automatic Construction of Error Correcting Parsers*, Acta Informatica 13, 115-139, 1980.

[26] Setzer, V.W., and Melo, I.S.H, *A construção de um Compilador*, Livros Científicos e Técnicos, Editora Campus Ltda, 1983.

[27] Wirth, Nicklaus, *Algorithm + Data Structures = Programs*, 1976.

[28] Wirth, Nicklaus and Ammann *Pascal - The Language and its Implementation*, Edited by D.W. Barron, 1981.

[29] Degano, Pierpaolo and Priami, Corrado, *The Comparison of Syntactic Error Handling in LR Parsers*, Software-Practice and Experience, 25(6), 657-679, June 1995.