GLOBAL CONTEXT RECOVERY: A NEW STRATEGY FOR PARSER RECOVERY FROM SYNTAX ERRORS

Ajit B. Pai*
Programs in Mathematical Sciences
The University of Texas at Dallas

and

Richard B. Kieburtz
Department of Computer Science
State University of New York at Stony Brook

Abstract

Methods for error recovery in parsing a deterministic, context-free language need not be derived by ad hoc means. When fixed, table-driven parsing algorithms are used the error recovery procedures can also make use of knowledge about the grammar for the language, derived from the parsing tables. However, at the point of detection of certain kinds of syntax errors, the knowledge of a sentential prefix already parsed, as reflected in the current state of the parsing stack, will be incorrect. Effective error recovery under such circumstances has proven to be difficult. We suggest the use of fiducial, or trustworthy symbols of a language as a key to recovery in these difficult cases. In this paper we give a characterization of such symbols in terms of properties of a grammar and give an algorithm that is capable of global context recovery under all circumstances. It has been implemented in conjunction with an LL(1) parser. We have compared it experimentally with the LR-parser recovery algorithm of Pennello and DeRemer on a test set of 126 small Pascal programs.

*Ajit Pai was fatally injured in an automobile accident on February 16, 1979. The results presented in this paper are taken from his Ph.D. thesis [5].

The research reported here was partially supported by the National Science Foundation under grant number MCS7681087.

1. Introduction

There is an extensive literature of schemes for recovery from errors encountered while parsing a deterministic, context-free language, yet in nearly all of these schemes, it is implicitly assumed that the error is local in its extent. More precisely, the most common assumption is that an error can be corrected either by inserting, deleting, or replacing a single syntactic token in the input string at the point of detection of an error. Not surprisingly, such schemes have had reasonable success in recovery from errors that satisfy these assumptions, but much less success in attaining recovery from errors that do not.

A practical error recovery method should have the following characteristics:

a) It should not have higher space or time complexity than does the parser.
b) It should not degrade the performance of the parser on error-free text segments.
c) Following an application of the error recovery scheme parsing should be able to continue if the remainder of the input string is a valid suffix of some sentence of the language.

When attempts to recover by making local alterations of the input about the point of a detected error do not success, a more drastic strategy is warranted. By expunging a portion of the

© 1979-ACM 0-89791-002-8/79/0800-0158 $00.75 see ii

remaining input text, it may be possible to obtain a valid suffix which can be parsed. By utilizing the contextual information implicit in the parsing stack, it may be possible to "splice" a valid suffix onto a sentential prefix. We term this kind of strategy global context recovery. This paper gives the outline of a global context recovery algorithm that is assured to allow parsing to be resumed correctly, if the remainder of the input contains no more errors. It depends upon a prior analysis of a grammar for the language, and is therefore most suitable for use in conjunction with table-driven, deterministic parsers.

The algorithm has been developed in detail, and implemented in connection with LL(1) parsing. However, there is no fundamental obstacle to adapting it to LR parsers as well. The technique has proven itself to be satisfactory in the implementation of error recovery for several programming languages, and is used in the Stony Brook Pascal 360 compiler, (Release 2).

Global context recovery is performed in two stages:

i) the input is scanned until encountering one of a set of specified syntactic tokens (called the fiducial symbols of the language).

ii) the content of the parsing stack is replaced by a sequence of symbols from which the parser can accept a sentential suffix beginning with the fiducial symbol encountered in the scanning stage.

The choice of a set of fiducial symbols is obviously crucial. The term fiducial means trustworthy; choice of a symbol depends on the extent to which its appearance constrains the nature of a string that may follow it in a sentential form, according to a particular grammar for the language. There can be an infinite number of sentential suffixes starting with a given symbol, but if all of them can be derived from a single sentential suffix then we can say that the symbol fully constrains the suffices starting with that symbol. Such a symbol is called a strong fiducial symbol. We shall describe a sufficient condition for a symbol to be a strong fiducial symbol in a given grammar. For reasons that will become apparent, there are very few strong fiducial symbols in any grammar for a programming language. But one can define a large and useful subclass of sentential suffixes for which many symbols will be fiducial.

We consider the subclass of suffixes of 'non-recursively' derived sentential forms; a symbol is a weak fiducial symbol if there exists a sentential suffix from which can be derived all sentential suffixes in this subclass which also start with that symbol. Many of the reserved words in a programming language can be weak fiducial symbols if a grammar for the language is chosen appropriately. In our error recovery scheme global context recovery is attempted on weak fiducial symbols.

An important idea in error recovery is that of 'phrase-level recovery'. The intent is to isolate a grammatical phrase containing the error, then correct it by substituting a similar, but correct instance of the phrase. The implementation of such a concept in connection with a bottom-up parse rests on the observation that certain strings are always syntactically reduced in the same manner, independent of the context in which they appear. The modus operandi of phrase-level recovery is to find such uniformly reducible strings in the look-ahead strings and reduce them. For example in

....; X:=Y A[<integer exp>]:=.....

the phrase 'A[<integer exp>]' is always reduced to a <variable>, in any context. Once this reduction has been made, one-symbol look-ahead finds the ':=', and the error in the above program can be corrected by inserting ';' before the <variable>. In this sense, phrase-level recovery may be termed deferred error correction: one does not merely scan the look-ahead symbol, but also processes it before attempting to correct the error.

The identification and reduction of uniformly reducible strings (i.e. strings whose reduction is independent of the context) is facilitated if the parser itself uses as little of the context as possible. So phrase-level recovery is eminently suited to bounded-context parsers, although it has been used with the LR-family of parsers too.

Graham and Rhodest [1] have applied phrase-level recovery to simple precedence parsers. Pennello and DeRemer [2] have extended the phrase-level recovery concept to LR-family of parsers. Here the task of reduction is complicated by the fact that in LR-parsing, a correct analysis of the input scanned so far is needed in order to parse any further. To circumvent this difficulty these authors use parallel parsing from the point of error, and perform a context independent reduction in the look-ahead string: only that sequence of shift and reduce actions is considered which is common to all paths of the parallel parse. Neither of these methods allows for a case in which the suffix represented by the remaining input happens not to be a valid continuation of the left context represented by the parsing stack. In such a circumstance, recovery fails.

2. An Example: LL(1) Error Recovery

A parser for a deterministic language consists of a finite-state control and a push-down store, and can parse without backing up, using a bounded look-ahead. Its parsing actions can be interpreted in terms of a grammar for the language. Thus, in the case of a top-down parser, the push-down store contains a suffix of a left-most sentential form and the parsing action determines the next production to be applied. In a bottom-up parser the push-down store contains a prefix of a right-most sentential form, the remaining portion of the input forming the corresponding suffix, and the parsing action determines the next reduction to be made.

2.1. Some Definitions and Theorems

For a detailed description of LL(1) parsers, the reader is referred to [3].

Let $G=(V_T, V_N, S, P)$ be a deterministic, context-free grammar, where

V_T is the set of terminal symbols,

V_N is the set of non-terminal symbols,

S is the goal symbol, and

P is the set of productions.

In what follows, we use capital letters A,B,C, etc. to denote non-terminals, lower-case letters r,s,t, etc. to denote terminals and lower-case greek letters α,β,γ, etc. to denote strings containing both terminals and non-terminals.

Let $A \to \alpha$ be the ith production. We define the lookahead set (LAS) of the pair (A,i) as follows.

Case 1: from α there is no derivation of an empty string.

$LAS(A,i) = \{t \mid$ there exists β such that $\alpha \Rightarrow^* t\beta\}$

Case 2: from α an empty string can be derived.

Let $S_1 = \{t \mid$ there exists β such that $\alpha \Rightarrow^* t\beta\}$

$S_2 = \{t \mid$ there exist β,γ,δ such that $S \Rightarrow \gamma A\delta$ and and $\delta \Rightarrow^* t\beta\}$

$LAS(A,i) = S_1 \cup S_2$.

If, for all A,j,i,t ($t \in LAS(A,j)$ and $t \in LAS(A,i)$ implies that i=j) then the grammar is said to be LL(1).

Define a function $F: V_N \times V_T \to P \cup \{error\}$ as follows.

F(A,t) = i, if there exists an i such that t is in LAS(A,i).

= error, otherwise.

If the grammar is LL(1) then the function F is well-defined and is called the parsing function. A language with an LL(1) grammar can be parsed top-down without backtracking, using the LL(1) parsing function to predict the production to be applied.

In error recovery it is useful to know whether a given terminal symbol can be embedded in a string derivable from a given non-terminal. This information is furnished by a boolean-valued function, WITHIN.

WITHIN: $V_N \times V_T \to \{true, false\}$ is defined as follows.

WITHIN(A,t) = true if there exist strings α,β such that $A \Rightarrow^* \alpha t \beta$.

= false otherwise.

The WITHIN function is used in error recovery to

determine where to reconfigure the stack when local corrections have failed.

At any stage of parsing, certain symbols are permitted as valid continuation symbols. This set of acceptable symbols (AS) is defined as follows. Suppose the parsing stack contains the string $Z\omega$, where Z is a terminal or non-terminal symbol.

$AS(Z) = \{t \mid$ there exists β such that $Z \Rightarrow^* t\beta\}$ if Z is a non-terminal
$= \{Z\}$ otherwise.

$AS(Z\omega) = AS(Z)$ if Z cannot produce an empty string.
$= AS(Z) \cup AS(\omega)$ otherwise.

$AS(\omega) = \emptyset$ if ω is empty.

As mentioned previously, when local corrections fail the scanner is advanced to a fiducial symbol and the parsing stack is adjusted to accept a suffix starting with that symbol. Furthermore, if there is more than one way to reconfigure the stack, then we should choose the way most likely to succeed. These observations can be formalized.

Definition 2.1.1: For any symbols A and z in a grammar G, a z-embedding derivation from A is a left-to-right canonical derivation $A \Rightarrow^* \alpha$ such that z occurs in α. Note: In a z-embedding derivation, the string α may contain several occurrences of z. To avoid ambiguity, we will always refer to the left-most occurrence of z in α. □

Definition 2.1.2: A z-embedding derivation $A \Rightarrow^* \alpha$ is said to be minimal if no intermediate string in the derivation contains an occurrence of z. □

Definition 2.1.3: Let $\Delta = (A \Rightarrow^* W)$ be a z-embedding derivation and let T be the derivation tree of Δ. In T, let P be the path from the root node to the (left-most) leaf node with label z. If no two nodes in P have the same label then Δ is called a z-embedding left-to-right non-recursive (LRNR) derivation. □

Intuitively, a z-embedding derivation is LRNR if it uses no recursive productions. This leads us to the following theorem, which we state without proof.

Theorem 2.1.4: If WITHIN(A,t) then there exists a t-embedding minimal LRNR derivation from A. □

Definition 2.1.5: Let $A \Rightarrow^* \alpha z \beta$ be a minimal z-embedding LRNR derivation from A. Then $z\beta$ is called a minimal z-suffix of A. □

We use minimal LRNR derivations for reconfiguring the parst stack in global context recovery.

2.2 The Global Recovery Algorithm

At the beginning of parsing, the parsing stack is initialized to contain the goal symbol S, preceded by an end marker. At any stage of parsing let p be the current token scanned and let Stacktop denote the top symbol of the stack.

Context Recovery:

Step 1: Scan the rest of the input text until a fiducial symbol is encountered, say p.

Step 2: Pop the stack until either Within(Stacktop,p) is satisfied, or the stack becomes empty.

Step 3: (a) In case the stack is empty, push the goal symbol S onto the stack;

(b) If Stacktop is a non-terminal, then replace it by one of the minimal p-suffixes that can be derived from it (actually, if the fiducial symbols have been chosen correctly, Stacktop will have a unique minimal p-suffix); otherwise, Stacktop is the terminal symbol p.

We present below a non-deterministic algorithm to reconfigure the stack. In this algorithm a production-marker is stacked before expanding a non-terminal in the stack. Whenever the production-marker is popped from the stack it implies that the last production applied has failed to produce a minimal p-suffix in the stack.

Algorithm 2.2.1

```
Successful = False;
repeat
  Case 1: Stacktop = Production-marker;
     do nothing;
  Case 2: Stacktop is a terminal symbol;
     if Stacktop = p then Successful := True
     else pop the stack.
```

3. An Algorithm for Global Context Recovery

Here we give a more detailed exposition of the Context Recovery algorithm, showing in particular how serial parsing with backup is used to simulate nondeterministic parsing. The principal task of Context Recovery is to reconfigure the parsing stack so that it will predict a valid sentential prefix beginning with the terminal symbol that is found in its scanning phase. This terminal may not be the head symbol of any phrase that can be produced from a nonterminal symbol of the grammar. It must, of course, occur within some such phrase. Because the scanned terminal may not be a head symbol, Context Recovery cannot use the parsing tables directly to find an applicable production. Instead, it must try all productions from a nonterminal capable of producing a phrase that may contain the designated terminal symbol. Since the right-hand sides of these productions may themselves contain nonterminal symbols, the nondeterministic parsing algorithm is applied recursively.

To make maximum use of the contextual information available, we seek a derivation from one of the symbols already present on the parsing stack, in fact, the one closest to the top of the stack. Since any valid sentential suffix beginning with the designated terminal will suffice to initiate recovery, we want one that can be derived without expanding any nonterminal symbol recursively, that is, a suffix that can be derived by a LRNR derivation. In order to constrain the derivation to be LRNR, as each nonterminal is tried to see whether it can derive a suitable suffix, it is entered into a set of previously expanded nonterminals. In searching for a derivation, no symbol in this set is ever expanded again.

In the event that there is no symbol in the stack from which can be derived a phrase containing the designated terminal, the stack will eventually be emptied. If this happens, the goal symbol is stacked, for derivation of an appropriate sentential suffix from the goal symbol is assured. We assume that the end-of-text marker (or its surrogate) is a fiducial symbol, and that it occurs in a production from the goal symbol.

The algorithm below is written in pseudo-Pascal. We trust the reader will have no difficulty in understanding it.

The function Within determines whether there is any phrase that can be derived from its first argument and contains the terminal symbol given as its second argument. While this information can be gotten by non-recursive derivation and examination, it is actually obtained by lookup in tables produced by the LL(1) constructor. These tables are quite compact, and their use usually saves both time and space.

```
type
   Symboltype = ( list of syntactic symbols );
   Productions = ( indices of productions of the grammar );
const
   AS : array [Symboltype] of
            set of Symboltype =
        [ set of acceptable terminal symbols for each nonterminal ]
var
   T : Symboltype; (* the terminal symbol
                      returned by the lexical analyzer *)
   Parse Stack : Stack of Symboltype;
                 (* the parsing stack used by the LL(1) parser *)

function SP : Stack pointer; (* points to top of Parse Stack *)

function StackTop : Symboltype; (* value at top of Parse Stack *)

function Prod Index (NT, T : Symboltype) : Productions;
          (* the LL(1) parsing function *)

function Within (NT, T : Symboltype) : Boolean;
```

```
procedure Context Recovery;
   const
      FirstSymbol : Symboltype = least element of Symboltype;
   var
      Expanded Symbols : set of Symboltype;
      Successful : Boolean;

   procedure Try Recovery;
      var
         NT, S : Symboltype;
         LB : Stack pointer;
         Trial Symbols : set of Symboltype;
         Trial Prodn : Productions;
      begin
         if Parse Stack is empty then
            push the goal symbol into Parse Stack;
         NT := StackTop;
         pop Parse Stack;  LB := SP;
         Expanded Symbols := Expanded Symbols + [NT];
         S := First Symbol;
         Trial Symbols := AS[NT];
         repeat
            (* first, locate a symbol in AS[NT] *)
            while not (S in Trial Symbols) do
               S := succ(S);
            Trial Prodn := Prod Index (NT, S);
            if the right hand side of Trial Prodn is non-empty then
               begin
                  apply Trial Prodn; (* push the right hand side into
                                        Parse Stack *)
                  repeat
                     if T in AS[StackTop] then
                        Successful := True
                     else if StackTop is nonterminal
                        and not (StackTop in Expanded Symbols)
                        and Within (StackTop, T) then
                           Try Recovery (* recursively *)
                     else
                        pop Parse Stack
                  until Successful or (SP = LB);
               end; (* of expansion of a nonempty Trial Prodn *)
            Trial Symbols := Trial Symbols - [S]
         until Successful or (Trial Symbols = [])
      end; (* Try Recovery *)

   begin (* Context Recovery *)
      advance the lexical scanner to a fiducial symbol and assign it to
         the variable T;
      Expanded Symbols := [];  Successful := False;
      repeat
         Try Recovery
      until Successful
   end;
```

To see that the algorithm is effective, we can verify several properties that it possesses.

1) The completion flag Successful is set True in only one place, and on the condition that the terminal symbol returned by the scanner is in the set of symbols expected from the configuration of the parsing stack. Once Successful becomes true, the algorithm terminates without further modification of any global data structure. Thus if the algorithm ever terminates, LL(1) parsing can resume.
2) The innermost repeat loop of Try Recovery terminates for one of the following three reasons:
   a) Successful is explicitly set to True;
   b) Successful is set to True in a recursive call to Try Recovery;
   c) the parsing stack is popped down to the previously set stack pointer, LB. Note that LB is set (in the fourth statement of the body of Try Recovery) after initially popping a symbol from the parsing stack. Thus if Try Recovery is called when the stack is nonempty and returns with Successful = False, the stack will have been popped once.
3) The outer repeat loop of Try Recovery terminates for one of the following reasons:
   a) Successful has become True, or
   b) the set of productions applicable from the nonterminal NT has been exhausted.

Note that a production that has failed will never be tried again, as NT is entered into the set of Expanded Symbols in the fifth statement of Try Recovery. If the same nonterminal should subsequently appear at the top of Parse Stack in an expansion of some production, it will be ignored without reexamination.

4) The repeat loop of Context Recovery will terminate because either
   a) In the initial call of Try Recovery, the parsing stack contained a symbol NT such that either T belongs to AS[NT] or else NT can be expanded, producing a stack configuration from which a T-suffix can be derived, or
   b) the stack initially contained no such symbols; the first call of Try Recovery terminated with Successful = False and the parsing stack empty, and in a second call to Try Recovery the goal symbol was pushed into the parsing stack. The goal symbol has the property called for in (a).

```
    Case 3: Stacktop is a non-terminal;
      if not Within(Stacktop,p) or if Stacktop
         is a symbol that has previously been
         tried then pop the stack;
      else expand Stacktop after pushing
         Production-marker onto the stack.
until Stacktop = Production-marker or Successful;
```

If Successful = True at the conclusion of the algorithm then the original Stacktop symbol will have been replaced in the parsing stack by a minimal p-suffix. Algorithm 2.2.1 is nondeterministic in the choice of which production to apply in expanding Stacktop at Case 3. Obviously, a deterministic version can be given at the cost of a bit more bookkeeping.

In extreme cases the parsing stack may be emptied at some point during execution of Context Recovery. When this happens, the stack is reinitialized to contain the goal symbol. Since every terminal symbol satisfies the WITHIN relationship to the goal symbol, there exists an embedding LRNR derivation from the goal symbol for any fiducial symbol. Hence, Context Recovery will terminate successfully giving the desired suffix in the stack.

4. Some Experimental Results

The error recovery strategy described in the preceding section has been implemented in LL(1) parsers for several programming languages, including Pascal. In each case an LL(1) grammar was constructed for the language; the grammar was then modified, without destroying the LL(1) property, so that the maximum number of symbols in the grammar would have the WPLU property described in the next section. These symbols were selected as fiducial symbols for global context

recovery. The resulting grammar was fed to an LL(1) constructor which generated both parsing and error recovery tables. The parser, augmented by two levels of recovery routines for local recovery and global context recovery, was driven by these tables. To obtain a comparison with other methods we tested our error recovery scheme on a set of Pascal program texts provided by Ripley and Druseikis [4]. They had compiled this set from programs written by students at the University of Arizona, who were novices in Pascal. The tape contains 126 Pascal programs with about 130 single token errors, such as mispunches, missing symbols, etc., plus additional syntax errors arising from the programmer's ignorance of Pascal.

The following criteria were used in evaluating the results of our test: an error correction is termined "excellent" if it repaired the text in the way that a competent programmer might have done, "good" if it resulted in a reasonable program without any spurious errors, and "poor" otherwise. (These criteria were used by Pennello and DeRemer [2] in evaluating their error recovery scheme on the same data.) The results we obtained are listed below:

Excellent	Good	Poor	Total	Corrections by Context Recovery
99	50	40	189	57
(52%)	(26%)	(22%)	(100%)	(27%)

The results obtained by Pennello and DeRemer [2] are as follows:

Excellent	Good	Poor	Unrepaired	Total
32	21	9	14	76
(42%)	(28%)	(12%)	(18%)	(100%)

The unrepaired category in the above tabulation refers to those errors for which no repair was chosen, but the rest of the input was used up in the attempt to find a proper correction, and therefore, control was not passed back to the parser.

Spurious errors were counted in both of the above statistics. Because some errors were unrepaired by Pennello and DeRemer's strategy, which does not perform global context recovery, there were some actual errors in program texts which were overlooked by their parser, while the recovery algorithms was in a forward scanning mode. Our error recovery strategy produced 78% excellent or good corrections. All of the errors that were visually detected in the program texts were found by our parser. Most of the errors were 'corrected' by local recovery although some of these corrections induced spurious errors; only in about 20% of the cases did the correction chosen by Context Recovery lead to spurious errors. These occurred when a symbol that had been designated as fiducial could in fact occur in more than one valid context. In such cases, the resolution of the ambiguity made by Context Recovery is arbitrary and may be incorrect.

The procedure Context Recovery always returns control to the parser, but if the input should be exhausted before a fiducial symbol is found by in the forward scanning mode, then the parse terminates immediately. Since the keyword symbols chosen as fiducial were fairly dense in the test programs, such a situation did not occur in our tests. Context Recovery enabled recovery from such diverse errors as declarations and statements appearing out of order, the use of Algol syntax in a Pascal program text, and even missing or incorrect comment and string delimiters.

5. A Method for Characterizing Fiducial Symbols

The choice of a set of fiducial symbols is crucial to an effective global context recovery algorithm. What properties should we look for in making such a choice? Can these properties be formally stated, so that a set of fiducial symbols can be selected by a constructor from an analysis of the grammar? We can give some definitive answers to these questions.

5.1 Strong Fiducial Symbols

Previously we mentioned two basic assumptions underlying global context recovery. To recapitulate,

a) we assume that the portion of the input string following a fiducial symbol is a valid suffix of some sentence in the language.

b) after reconfiguration, the parser can parse any valid suffix that begins with the fiducial symbol.

The second assumption gives us an ideal, but stringent, definition for a fiducial symbol.

Definition 5.1.1: A string $t\omega$ is called a terminal t-suffix of a language L if there is a prefix α such that $\alpha t\omega$ is a sentence of L. □

Definition 5.1.2: A terminal symbol t is a strong fiducial symbol (SFS) of a grammar G for L, if there exists a string β which is a suffix of a sentential form of G and for every terminal t-suffix $t\omega$ of L, there is derivation $\beta \Rightarrow^* t\omega$. □

We use the adjective 'strong' in the above definition to distinguish it from another, more practical, definition for fiducial symbols that will be given later.
Note: All the grammars considered below will be assumed to be in reduced form. Also, we assume that the goal symbol in a grammar is nonrecursive.

Definition 5.1.3: A terminal symbol z in a grammar G has strong phrase-level uniqueness (SPLU) if either G is empty or

1) there is at most one production, say $A \rightarrow \gamma$, which contains z in its right hand side,
2) z occurs only once in the right hand side γ,
3) A does not have left or embedded recursion in G, i.e. there is no derivation in G of the form $A \Rightarrow^* \alpha A\beta$, where $\beta \neq \lambda$
(λ denotes the empty string,
4) A has SPLU in G(A), where G(A) is the grammar, in reduced form, obtained from G by deleting all productions for A. □

Theorem 5.1.4: If z has SPLU in G then z is a SFS.

Proof: From the minimal z-suffix of S we can derive any terminal z-suffix. □

However, the SPLU property is too restrictive to expect of many symbols in a typical programming language, as is seen from the following:

Lemma 5.1.5: If every terminal symbol in a grammar G has SPLU then L(G) is a regular language.

Proof: The SPLU property allows only right recursive productions in a grammar for the language. Any language generated by such a grammar is regular. □

Though SPLU is a sufficient condition it is not a necessary condition for a symbol to be SFS. However, it seems that the grammars in which every symbol is a SFS will generate a very restricted class of languages. In any grammar for an actual programming language, there are very few symbols qualified to be SFS and fewer still have SPLU. However, the choice of reserved words as fiducial symbols leads to fairly successful error recovery in most practical cases. We clearly need a less restrictive characterization of fiducial symbols in a programming language.

5.2 Weak Phrase-Level Uniqueness

Definition 5.2.1: A terminal symbol z has weak phrase-level uniqueness (WPLU) in a grammar G if either G is empty, or

1) there is at most one production, say $A \rightarrow \alpha z\beta$, which contains z in its right hand side,
2) z occurs only once in the right hand size $\alpha z\beta$, and
3) A has WPLU in G(A), where G(A) is the grammar, in reduced form, obtained from G by considering A to be a terminal symbol. □

The definition of WPLU relaxes the SPLU restriction that the nonterminal A, from which a string containing z is produced, cannot have left or embedded recursion in G. In consequence, for a symbol z having WPLU, a z-suffix may contain several repetitions of an initial phrase. This means that following error recovery on a fiducial symbol z, having WPLU but not SPLU,

there can be valid z-suffixes on which the parser will detect a spurious error. In spite of this fact, the WPLU property seems to be a good criterion on which to base the selection of fiducial symbols.

References

[1] Graham, S.L. and Rhodes, S.P. Practical syntactic error recovery, Comm. ACM 18, No. 11, pp. 639-649, (Nov. 1975).

[2] Pennello, T.M. and DeRemer, F., A forward move algorithm for LR error recovery, Proc. Fifth Annual ACM Symp. on Principles of Programming Languages, pp. 241-255, (1978).

[3] Aho, A.V. and Ullman, J.D., The Theory of Parsing, Translating and Compiling: Vol. 1, Parsing, Prentice-Hall, Englewood Cliffs, New Jersey, (1972).

[4] Ripley, D.C. and Druseikis, F.C., A statistical analysis of syntax errors, Technical Report, Department of Computer Science, University of Arizona, Tucson, (1978).

[5] Pai, Ajit B., Syntax driven error recovery in top-down parsing, Ph.D. thesis, Department of Computer Science, SUNY at Stony Brook, August 1978.