From lecomte@bond.crim.ca Tue Feb 19 07:40:18 1991
Date: Tue, 19 Feb 91 07:40:02 EST
From: lecomte@bond.crim.ca (Louis Lecomte)
To: bochmann@IRO.UMontreal.CA

TITLE: Error recovery with Yacc (revisited)

AUTHORS: Louis Lecomte, Gregor v. Bochmann

The general problem

This paper is a result of an experiment done with the parser generator Yacc. This tool has been used to build the syntax analyser for a new object oriented language named Mondel. The problem consisted in the introduction of error recovery rules into the Mondel's grammar. Such rules are mandatory in order to make the parser robust in the case of syntax errors.

The litterature

The classical litterature [**] proposes different methods or strategies to acheive a recovery. However, their application in the case of Yacc is not so immediate. The specific examples about Yacc [**] describes most of the time the same trival example about the generation of an interactive calculator. This description is far from being sufficient when one wants to apply the ideas in the case of non interactive compiler. In contrast to this elementary example, [Schr 85] proposes a complete explanation of the Yacc error recovery mecanisms. But for the usual practionner, it may appear overly technical and complex.

The qualities of a recovery

Some technical problems result from the introduction of error recovery rules. The most obvious are the introductions of parsing conflicts in the grammar or infinite loops in the parser. Actually, we have identified five criterions or nice properties the resulting parser should satisfy.

- No parsing conflicts in the grammar, (q1)
- No infinite loops in the parser, (q2)
- The input text is completely parsed, (q3)
- Minimize the false or cascaded errors, (q4)
- Makes the recovery as tight as possible. (q5)

The goal of this paper

Describe a systematic way to introduce error recovery rules into a Yacc specification. Insure that the criterions (q1) to (q5) are met.

Parser properties

Correct prefix property

The LALR(1) parsing technique supported by Yacc has the correct prefix property. That is, if we are in an input configuration xTy where the

sentence x has been read, terminal T has been found to be syntatically illegal, and y is the remaining input, then we may infer that x is the prefix of some valid program, but xT is not. Such a terminal occurence T is called a symptom.

A symptom is not shifted onto the parse stack when it is detected. (p1)

Two working modes (p2)

The parser works in normal mode while it has not detected a symptom.
Only the normal gramatical rules are considered.
When it finds out a symptom (the lookahead symbol) in the input text, it switches in error mode. In that mode, the recovery rules are considered.

Specification style resulting from (p2) (tout au plus un footnote)

It is convenient to group the recovery rules together into a recovery grammar and separate them from the normal grammar.

- Reflects the way the parser works with its two working modes,
- Emphasizes the distinction between the language definition and the recovery rules,
- Increase the readability of the semantical actions.

General format of a recovery rule

Assumption: (A)

All the recovery rules used in a grammar specification have exactely the format defined by the rules (g2) to (g6).

```
scope        : context normal_definition                          (g1)
             ;
scope        : context error anchors {yyerrok;} continuation      (g2)
             ;
continuation : ...                                                (g3)
             | OUT
             ;
anchors      : resynch                                            (g4)
             | get_out       {insert(OUT);}
             ;
resynch      : ...                                                (g5)
             ;
get_out      : ...                                                (g6)
             ;
```

... : to be filled out by the designer.

OUT, VOID : two special terminals which do not belong to the language. In addition VOID is never returned by the lexical scanner. (VOID does not appear above but it will be used in the application examples).

A recovery rule is an extension to the normal language aimed to consider the case of syntax error. The rule (g1) is part of the language definition. It is called a normal rule. Observe here,

the symbol scope is not necessarily the start symbol. However, it defines a sub-language extended by the rule (g2). In the following, when we talk about a recovery rule, we refer to a rule with format (g2). Subsequent rule (g3) to (g6) refine the (g2) definition and provide the recovery algorithmic (known as the panic mode recovery or something like).

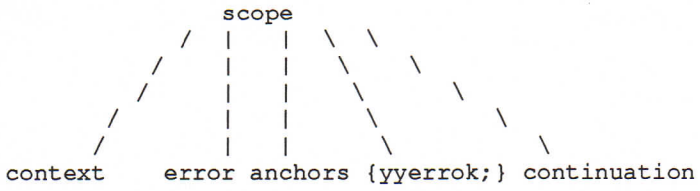


Figure Tree representation of a recovery rule

The recovery rule components

Footnote: To distinguish
symbol : Terminal or Nonterminal
Token : occurence of a terminal

The recovery rule components can be defined as follows:

The scope component is a nonterminal which belongs to the normal language. It delimits the contruct on which the resynchronisation makes sense.

The context component is a sequence of symbols (might be empty) which begins the normal construct definition (g1). So a context is shared by both, the normal and the recovery rules. It is a 'significative' sequence of symbols which identifies uniquely the construct. It consists generally in one terminal like BEGIN, IF, UNIT or any keyword.
The aims of a context are: (1) Avoid introducing parsing conflicts in the grammar, (2) increase the confidence in our assumptions, that is, if such a symbol legally appears on the parse stack, we may infer with enough accuracy, that the construct being analyzed is the one defined by normal scope definition.

The 'error'component is a Yacc predefined terminal. It represents syntax errors occuring after the context recognition and before the construct has been completely recognized. Its semantics will be discussed later.

The 'anchors' component defines a language, usually a set of 'safe' terminals, on which the recovery is is based. Anchors are splitted into two exclusive categories, the 'resynch' and 'get_out' anchors (g4). This will be discussed latter.

The synthetic symbol {yyerrok;} is an action which resumes the parser to the normal working mode.

The continuation symbol defines the normal parsing after resynchronisation.

How the parser behaves in the case of a syntax error

When a syntax error is detected, the parser switches in error mode.

It pops its parse stack until a state that can shift the error symbol is found. If it cannot find such a state (for instance if no recovery rules are specified) then it stops. Otherwise the parsing continue according to a rule of type (g2) (see our assumption (A)).
Starting from the symptom (see (p1)), the parser skips the remaining input text until it finds out an anchor (in accordance with the error symbol semantics). The parser then resumes to the normal mode (in accordance with the yyerrok action semantics) and the parsing restarts according to the continuation definition.

The non looping properties

In most cases, the recognition of an anchor is followed by an action which replaces the anchor to the beginning of the input text. Observe here that the symptom and the anchor can be equal (in accordance with (p1)). If we do not take care, the same error will be detected once more. An infinite loop will result. To avoid this problem, the designer should insure that one of the following properties holds.

The anchors are necesarily shifted when the parser resumes. (c1)
The anchors are shifted or the recovery scope is changed (c2)
to a broader one.

Furthermore, to insure the input text is parsed completely, the designer should also insure the following property.

In the case of a syntax error, the parser always finds out a state (c3)
to shift the error symbol.

The basic result

If a recovery grammar is designed in such a way that condition (c1) holds in the higher recovery scope and condition (c2) holds in the others,then it does not introduce infinite loops in the parser.

Moreover, if the empty context is used to recover in the scope of the start symbol(the higher recovery scope), then the condition (c3) holds and therefore the input text is completely scanned.

The methodology

In accordance with the results above, the suggested methodology consists first in providing the start symbol with a recovery rule and an empty context. Then the recovery is made tighter, by considering other rules with finer scopes. This is repeated until a satisfactory result (rel. (q5)) is reached.

The lexical requirements

The lexical scanner constitutes the interface between the parser and the input text. It models the text as a sequence of tokens. In addition to the usual function, 'get' say, which returns the next token available, we need two operations for the error recovery. They are named 'unget' and 'insert'.
The unget operation undoes the effects of 'get'. Invocation of 'unget(n)' replaces at the beginning of the text the last sequence of n tokens returned by 'get'. In order to make some repairs to the

input text, token insertions are allowed by invocation of 'insert(T)'.
The use of these operations are shown below.

Application examples

Example 1 The start symbol

```
specification   : UNIT_definition                                          (s1)
                ;
UNIT_definition : UNIT unit_id                                             (s1')
                    opt_TYPE_definitions
                    opt_BEHAVIOR_definition
                    opt_WHERE_clause
                  ENDUNIT unit_id
                ;
specification   : error spec_anchors { yyerrok; } spec_cont                (s2)
                ;
spec_cont       : UNIT_definition                                          (s3)
                | OUT
                ;
spec_anchors    : spec_resynch                                             (s4)
                | spec_out  { insert(OUT); }
                ;
spec_resynch    : UNIT      { unget(1); }                                  (s5)
                | TYPE      { unget(1); insert(ID); insert(UNIT); }
                | BEHAVIOR  { unget(1); insert(ID); insert(UNIT); }
                | WHERE     { unget(1); insert(ID); insert(UNIT); }
                | ENDUNIT   { unget(1); insert(ID); insert(UNIT); }
                ;
spec_out        : VOID                                                     (s6)
                ;
```

Syntax:

Rules (s1) to (s6) but (s1') are special cases of the general rules (g1) to (g6) where the blanks have been filled out. The normal rule (s1') is given only to provide a better understanding of the anchor choices. THe special terminal VOID is the only alternative of (s6). Since, by definition, VOID is never returned by the lexical scanner, the spec_out's language is empty. It is a trick to preserve the general format.

Semantics:

Assume (s2) is the only recovery rule. When a syntax error is detected, the parser switches in error mode. It empties its parse stack. It shifts the terminal error and skips the text until it finds out one of the terminal enumerates in (s5). The recognition of such a symbol activates an 'unget' action which replaces the anchor at the beginning of the input text. But for the first case, an alteration is performed through the 'insert' actions. This completes the anchor to a legal prefix of the continuation 'spec_cont'. So, condition (c1) holds. No looping.

Choices:

The start symbol with an empty context has been chosen to meet the condition (c3). The anchors (s5) have been choosen as 'safe' terminals in the scope's vocabulary. The context being empty the only way to continue

the parsing is to restart from the beginning (s3). Alterations has been defined in order to satisfy the condition (c1).

Recovery refinements

Example 2 A compound statement

```
BEGIN_stmt    :  BEGIN  statements  END                                  (b1)
              ;
BEGIN_stmt    :  BEGIN  error  begin_anchors  {yyerrok;} begin_cont      (b2)
              ;
begin_cont    :  opt_statements  END                                     (b3)
              |  OUT
              ;
begin_anchors :  begin_resynch                                           (b4)
              |  begin_out        { insert(OUT); }
              ;
begin_resynch :  first_stmts      { unget(1); }                          (b5)
              |  END              { unget(1); }
              ;
begin_out     :  ENDUNIT          { unget(1); }                          (b6)
              |  ...
              ;
```

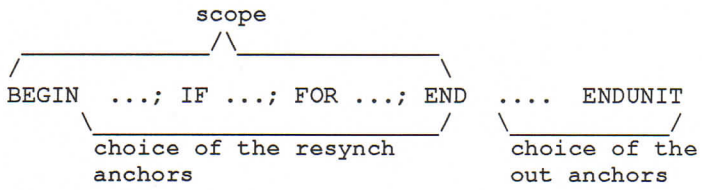
Because, the rule (s2) does not provide an enough tight recovery, we extend finer symbols with the same recovery mecanism; here the begin_stmt is considered. Rules (b1) to (b6) above are also special cases of the general rules (g1) to (g6).

Two exclusive anchor categories

In contrast to the first example, the current one offers a non empty context BEGIN and a non empty set of "get_out" anchors. In the following, we explain the distinctions with a "resynch" anchor.

The resynch anchors

Consider the rule (b4). The first anchor category, named 'begin_resynch' is used to resynchronize the parser in the scope of a BEGIN statement.



The designer concentrates on the sub-language defined by the scope symbol. He/she defines the 'resynch anchors' by selecting 'safe' symbols inside this sub-language. In the current example,
we use the reserved identifiers beginning a statement. The terminal END is also used. Reaching such an anchor, the parser behaves like in the

previous example. Indeed, the first continuation alternative (b3) has been defined not "too far" from the normal definition (b1) but in such a way that the condtion (c1) is met.

The out anchors

The second category of anchors, named 'begin_out', is introduced to detect when we are out of scope (second alternative of (b4))/ Consider the case where the
mistake is on the closing END; a misspelt ENND for instance. The erroneous statement might be the last one of a program. So, the next 'resynch anchor' the parser needs for resynchronization might be far in the text. A large portion of text can be skipped before the resynchronization.

A 'begin out' anchor has no sense in the scope of a BEGIN statement; for instance, the keyword ENDUNIT. But it can make sense in a broader scope. Reaching such an anchor (b6), we infer that the BEGIN statement is completed. Here is a trick. A special token OUT, which does not belong to the language, is inserted (b4) at the begining of the remaining input text. Since OUT appears as an alternative of the continuation (b3), it is shifted onto the parse stack, and rules (b3) and (b2) are reduced.

However, the ungeted 'out anchor' is not necessarily shifted onto the parse stack. The condition (c1) does not hold. If the ungeted anchor is found syntactically illegal, a broader recovery scope is considered. Either condition (c1) holds in that new scope or a scenario as above happens. Since (c1) holds in the higher recovery scope, by construction, the anchor will eventually be shifted. No looping. This justifies the (c2) condition.

The methodology once more

With the previous examples as justifications we can define a kind of 'algorithm' to direct the introduction of recovery rules into a Yacc specification.

(1) Extend the start symbol with a recovery rule. In that rule use a context that insures (c3).An empty context generally constitutes a good choice.

(2) While the recovery is not tight enough, select a finer scope symbol and extend it with a recovery rule.

The recovery rule definitions should conform to the general format (g1) to (g6). Each rule should be defined according to the following steps.

1. Identify a context
2. Select a set of 'resynch' anchors
3. Define the continuation in such a way that condition (c1), restricted to the 'resynch' anchors, holds.
4. Define a set of 'get out' anchors. For the special case where the scope is the start symbol, use VOID only.

Quality of the recovery: How to meet the criterions (q1) to (q5)?

(q1) by selecting 'significative' contexts.
(q2) by insuring conditions (c1) and (c2). (see the result (r1)).
(q3) by insuring condition (c3).
(q4) by selecting 'safe' anchors.
(q5) by providing symbols with 'finer' scope with an error recovery rule.
by increasing the list of anchors.

Let us make some precisions on what we mean by 'fine', 'safe' and 'significative'.

A 'finer' scope

Grammatical symbols are naturally ordered (reflexive and transitive relationship) into a grammar specification. The start symbol is said having the broader scope and the terminals the finer ones. Our methodology consists of providing the broader scope with a recovery rule. Then the recovery is made tighter by considering symbols with finer scopes. But not too fine.

Observe that some symbols are not really suitable for recovery, for instance an expression. The main reasons are (1) an expression occurence can have a quite complex structure which is ideal for cascading errors, (2) an expression occurence consists generally of a short sentence that can be written on one line. So detecting only one error into an expression and recovers according to a broader recovery scope seems sufficient, at least it is labour saving.

A 'safe' anchor

For resynchronisation, an anchor is said safe if

- it is long enough not to be the result of a misspelling
- it makes sense in the scope considered,
- it provides a non ambiguous information on where we are in the text.

For example, reaching an anchor such as (b5) one can assert the next construct is a statement. This eases the definition of the continuation. To justify the 'long enough', let us consider the case where the statement terminator ';' is used for resynchronisation. It is often suggested in the litterature. But we think it is a wrong choice because many statements use the ';' in multiple ways. For instances we have the following constructs

```
forall a:A;b:B  DO ... END;   in Mondel
for(;;);                      in C
p(a;b);                       in Pascal
```

Actually, the latter example is syntactically incorrect. The right version is p(a,b);. But the mistake is likely to occur. Look at what happens if the ';' is used to resynchronize. The symptom becomes the anchor (p1). Parsing restarts on b and a wrong error is detected on ')'. Such scenario are likely in C and Mondel since the ';' does not necessarily identify the end of a statement.

'get out' anchors

- It is long enough not to be the result of a misspelling,

- It does not make sense in the scope considered,.

A 'significative' context

- It should have the 'long enough' property.
- It should identify the scope in a unique way

Influences on the language design

*** To fill out

The fault model

The danger when one specifies the recovery rules is to consider the cases of extraordinary errors. For instance a ENDUNIT occurence as a statement. The designer should restrict the fault model to a set of simple mistakes. For instances, misspelt keywords, terminator missing or any sample of usual faults. Don't worry about the parser behavior in the case of exceptionnal faults. We must be realistic, error recovery is not perfect yet.

Automation of the process

But for the part to fill out, the general rule format above as well as the systematic methodology we have presented suggest to look at some automation in the process.

For further studies

What about the designer intuition.

Experiments with Mondel

For the Mondel grammar, recovery rules having the general format have been used. It works surprisingly well. The criterions (q1) to (q5) have been met. But it is a long job. A kind of automation would be interesting. The designer should wait until the syntax definition adoption before defining many many recovery rules. A simple rule as (s2) should be sufficient until the syntax stability.

Bibliography