Robust Parsing in TXL

TXL Working Paper 9 (Also published as Legasys WP98-161-JRC)
James R. Cordy
July 1998

This paper describes how an existing TXL grammar can be easily extended to provide robust identification and correction of minor syntax errors without causing the parse to fail. The strategy is a TXL implementation of the general strategy described by David Barnard in his MSc thesis (Toronto, 1975).

1. Barnard syntax error recovery and repair

The general strategy proposed by Barnard is based on the observation that the syntax of programming languages tends to be structured into "statements", each of which ends in an explicit marker, for example, the semicolon (;) in Pascal, PL/I, C, etc. and the period (.) in Cobol. David makes the further observation that even when there is no explicit marker in the language syntax, programmers tend to use line boundaries in the same way. For example, Turing programmers tend to hit carriage return at the end of a statement (although they need not since the Turing language is not line-boundary sensitive).

This observation leads to a very simple and proven effective ("ask our millions of satisfied clients") syntax error handling and repair strategy described in Barnard's MSc thesis, refined in his PhD, and used in virtually every modern compiler. A simple version of the strategy can be described as follows:

- When a syntax error is detected, enter "recovery state".

- While in recovery state, continue parsing as if each expected input token were matched, but do not accept any actual input.

  (This has the effect of continuing to build a valid parse, but of some program slightly different from the actual input program.)

- When the expected input token is a statement end marker (e.g., semicolon, period, line boundary, etc. depending on the language), flush the actual input to the next statement end marker, exit recovery state, and continue to parse normally.

  (Flushing the input to the expected statement end marker has the effect of resynchronizing the input with the parse, most often minimizing collateral syntax errors due to the first one, and preventing syntax error "cascades".)

2. Barnard syntax error recovery and repair in TXL

Barnard's strategy is based on the assumption that the parsing algorithm knows more or less locally when a syntax error is detected and can choose a preferred lookahead from the expected lookahead tokens in any parse state.

Both these properties are true of LL and LR parsers, but unfortunately, neither is true of TXL's full backtracking ambiguous grammar parser, which only knows for sure that there is a syntax error when it has fully backtracked out of the entire parse. Thus it is not practical for TXL itself to use Barnard's syntax error recovery strategy.

On the other hand, this exact same generality and handling of ambiguity allows the TXL grammar programmer to implement the Barnard strategy him/herself, by coding a "fall through" case for language "statements" that captures statements that fail to parse and explicitly parsing them as arbitrary tokens followed by a statement end marker. This effectively flushes the statement containing the syntax error, capturing the unparseable stuff in the fall through case, and allows the parse to continue.

The programming of the strategy is on the face of it very simple, although tricky to get right in detail. For example, here are the changes to a Cobol grammar to make it robust with respect to syntax errors. The changes for PL/I can be very similar.

```
% Fall through to catch malformed records
define record_description
        ...
    |   [garbage_record]
end define

% There must be at least one token we failed to parse ...
define garbage_record
    [token] [flush_to_dot]
end define

% ... followed by any amount of other stuff, ending in a dot.
% Note the recursion paradigm to accept anything up to a dot.
% The built-in nonterminals [token] and [key] partition all the
% possible inputs - [token] is defined to accept anything other
% than a keyword, and [key] accepts all of those.
% Because it is the first choice, the recursion will terminate
% at the first dot (even though it could also be parsed as a
% [token]).
define flush_to_dot
        '.
    |   [token] [flush_to_dot]
    |   [key]   [flush_to_dot]
end define
```

The tricky part is the fact that [garbage_record] begins with [token], and cannot begin with [key]. This is extremely important, since the only way a parser using a Cobol grammar recognizes the end of the DATA DIVISION is by the keyword PROCEDURE following it. If [garbage_record] were to allow [key] as its first input, the entire rest of the program following the first syntax error would be parsed as [garbage_record]'s !

It is even trickier in the case of making the [sentence] grammar robust. Firstly, since Cobol [statement]'s don't have an end marker, we cannot apply Barnard's method at that level, so we must move up a level and apply the strategy to [sentence], which ends with a dot, instead.

```
% Fall through to catch malformed sentences
define sentence
        ...
    |   [garbage_sentence]
end define

% A garbage sentence begins with anything but an identifier
% or number, followed by anything at all up to the next dot
define garbage_sentence
        [key]       [flush_to_dot]
    |   [special]   [flush_to_dot]
    |   [stringlit] [flush_to_dot]
    |   [charlit]   [flush_to_dot]
end define

% Every possible special character and compound token except dot
define special
    % All the special characters on the keyboard, except dot
        '` | '~ | '! | '@ | '# | '$ | '% | '^ | '& | '*
    |   '( | ') | '_ | '+ | '- | '= | '{ | '} | '[ | '] | '|
    |   '\ | ': | '; | '< | '> | ', | '/ | '?
    % All the Cobol compound tokens
    |   '<= | '>=
end define
```

The tricky part in this case is the fact that a sequence of Cobol [sentence]'s usually ends at the next paragraph header, which is recognized by the parser by the fact that it begins with an [id] or [number]. If [garbage_sentence] were allowed to begin with an [id] or [number], we would parse all the paragraph headers in the program as [garbage_sentence]'s.

Finally, we must make the parsing of malformed [paragraph_header]'s robust in order to handle syntax errors where malformed statements happen to begin with an [id] or [number]. In this case, we know that the malformed item begins with one of these since otherwise it would have been caught by [garbage_sentence].

```
% Fall through to catch malformed paragraph headers
define paragraph_header
        ...
    |   [garbage_paragraph_header]
end define

define garbage_paragraph_header
        [id_or_integer] [flush_to_dot]
end define
```

3. Processing repaired syntax

It is important that when garbage is recognized in the parse, the user is informed of it. Thus the robust parser must run rules to check for items identified as garbage, and emit appropriate warning messages. For example, for a Cobol robust parser, the rule to check for garbage sentences might look like this:

```
rule warnAboutGarbageStatements
    replace $ [sentence]
        G [garbage_sentence]
    construct _ [garbage_sentence]
        G [putp "***WARNING: Malformed sentence '%' ignored"]
    construct QuotedGarbage [charlit]
        _ [quote G]
    by
        'DISPLAY ''GARBAGE-SENTENCE' QuotedGarbage '.
end rule
```

It is also important that the malformed garbage be changed by the rule into something legal, in order to save the rest of the process from having to handle the syntax errors (i.e., the other phases shouldn't need the robustness overrides). For example, the rule above turns garbage sentences into Cobol DISPLAY statements containing the garbage text.