LL Error Recovery

Panic-mode error recovery is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens appears. LL error recovery refers to the stack of a table-driven parser since it makes explicit the terminals and nonterminals that the parser hopes to match with the remainder of the input. An error is detected in LL parsing when

a. the terminal symbol on top of the stack does not match the next input symbol or

b. the entry in the parsing table for the nonterminal symbol on top of the stack and the next input symbol is error or the parsing table is empty.

Def: Let S be a set of tokens called a synchronization set.

Def: Let s ∈ S. s is called a synchronization token.

The panic-mode error recovery can be implemented using the synchronization set(s) as follows:

1. Pop items off the parse stack until a synchronization token is at the top of the stack. Note: error recovery is not possible if no such item is on the stack.

2. Read the input symbols until the current lookahead symbol matches the symbol at the top of the stack or until you reach the end of the input.

3. If you are at the end of the input, error recovery failed, otherwise you have recovered.

The effectiveness of the panic-mode error recovery depends on the choice of synchronizing set(s). The sets should be chosen so that the parser recovers quickly from errors that are likely to occur in practice and so that a minimal amount of input is ignored. Some heuristics for constructing synchronization set(s) are as follows:

1. As a starting point, place all symbols in FOLLOW(A) into the synchronization set for nonterminal A. If we skip tokens until an element of FOLLOW(A) is seen in the input and pop A from the stack, we have effectively removed anything that could be generated from A, and it is likely that parsing can continue successfully.

2. It is often times not enough to use FOLLOW(A) as the synchronization set for A. For example, if semicolons terminate statements, as in C, then keywords that begin statements may not appear in the FOLLOW set of the nonterminal representing expressions. A missing semicolon after an assignment statement may therefore result in the keyword beginning the next statement being skipped. Often in programming languages, there is a hierarchical structure on constructs in a language. e.g. expressions appear within statements, which appear within blocks, etc. We can add symbols that begin the higher-level constructs to the synchronization set of a lower-level construct. For example, we might add keywords that begin statements to the synchronization sets for the nonterminals that generate expressions.

3. If we add the symbols in FIRST(A) to the synchronization set for nonterminal A, then it may be possible to resume parsing of A if a symbol in FIRST(A) appears in the input. This situation might occur if terminal symbols were injected into the input.

4. If a nonterminal can generate the empty string, then the production(s) deriving λ can be used as a default. Using this technique may postpone some error detection, but it cannot cause an error to be missed. This approach reduces the number of nonterminals that have to be considered during error recovery.
5. If a terminal on top of the stack cannot be matched to the input symbol, a simple idea is to pop the terminal, issue an informative message saying that the terminal was inserted into the input stream, and continue parsing. In effect, this approach takes the synchronization set of a token to consist of all other tokens.

**Phrase-Level Recovery**

Phrase-level error recovery can be implemented by filling in the blank entries in the LL parsing table with error routines that are called when that entry in the table is accessed. These routines could change, insert, or delete symbols on the input and issue appropriate warning messages. The routines could also manipulate the stack by popping or inserting symbols from/to the stack. Alteration of stack symbols is very questionable for many reasons. In general, such stack manipulation should ensure that there is no possibility of an infinite loop and that the corresponding derivation really represents a string in the language of the grammar.