Next: Handling Context Dependencies, Previous: The Bison Parser Algorithm, Up: Bison [Contents][Index]

# 6 Error Recovery

It is not usually acceptable to have a program terminate on a syntax error. For example, a compiler should recover sufficiently to parse the rest of the input file and check it for errors; a calculator should accept another expression.

In a simple interactive command parser where each input is one line, it may be sufficient to allow `yyparse` to return 1 on error and have the caller ignore the rest of the input line when that happens (and then call `yyparse` again). But this is inadequate for a compiler, because it forgets all the syntactic context leading up to the error. A syntax error deep within a function in the compiler input should not cause the compiler to treat the following line like the beginning of a source file.

You can define how to recover from a syntax error by writing rules to recognize the special token `error`. This is a terminal symbol that is always defined (you need not declare it) and reserved for error handling. The Bison parser generates an `error` token whenever a syntax error happens; if you have provided a rule to recognize this token in the current context, the parse can continue.

For example:

```
stmts:
  %empty
| stmts '\n'
| stmts exp '\n'
| stmts error '\n'
```

The fourth rule in this example says that an error followed by a newline makes a valid addition to any `stmts`.

What happens if a syntax error occurs in the middle of an `exp`? The error recovery rule, interpreted strictly, applies to the precise sequence of a `stmts`, an `error` and a newline. If an error occurs in the middle of an `exp`, there will probably be some additional tokens and subexpressions on the stack after the last `stmts`, and there will be tokens to read before the next newline. So the rule is not applicable in the ordinary way.

But Bison can force the situation to fit the rule, by discarding part of the semantic context and part of the input. First it discards states and objects from the stack until it gets back to a state in which the `error` token is acceptable. (This means that the subexpressions already parsed are discarded, back to the last complete `stmts`.) At this point the `error` token can be shifted. Then, if the old lookahead token is not acceptable to be shifted next, the parser reads tokens and discards them until it finds a token which is acceptable. In this example, Bison reads and discards input until the next newline so that the fourth rule can apply. Note that discarded symbols are possible sources of memory leaks, see Freeing Discarded Symbols, for a means to reclaim this memory.

The choice of error rules in the grammar is a choice of strategies for error recovery. A simple and useful strategy is simply to skip the rest of the current input line or current statement if an error is detected:

```
stmt: error ';'  /* On error, skip until ';' is read. */
```

It is also useful to recover to the matching close-delimiter of an opening-delimiter that has already been parsed. Otherwise the close-delimiter will probably appear to be unmatched, and generate another, spurious error message:

```
primary:
  '(' expr ')'
| '(' error ')'
…
;
```

Error recovery strategies are necessarily guesses. When they guess wrong, one syntax error often leads to another. In the above example, the error recovery rule guesses that an error is due to bad input within one `stmt`. Suppose that instead a spurious semicolon is inserted in the middle of a valid `stmt`. After the error recovery rule recovers from the first error, another syntax error will be found straight away, since the text following the spurious semicolon is also an invalid `stmt`.

To prevent an outpouring of error messages, the parser will output no error message for another syntax error that happens shortly after the first; only after three consecutive input tokens have been successfully shifted will error messages resume.

Note that rules which accept the `error` token may have actions, just as any other rules can.

You can make error messages resume immediately by using the macro `yyerrok` in an action. If you do this in the error rule's action, no error messages will be suppressed. This macro requires no arguments; '`yyerrok;`' is a valid C statement.

The previous lookahead token is reanalyzed immediately after an error. If this is unacceptable, then the macro `yyclearin` may be used to clear this token. Write the statement '`yyclearin;`' in the error rule's action. See Special Features for Use in Actions.

For example, suppose that on a syntax error, an error handling routine is called that advances the input stream to some point where parsing should once again commence. The next symbol returned by the lexical scanner is probably correct. The previous lookahead token ought to be discarded with '`yyclearin;`'.

The expression `YYRECOVERING ()` yields 1 when the parser is recovering from a syntax error, and 0 otherwise. Syntax error diagnostics are suppressed while recovering from a syntax error.

---

Next: Handling Context Dependencies, Previous: The Bison Parser Algorithm, Up: Bison [Contents][Index]