Chapter 8. Error Reporting and Recovery

The previous chapters discussed techniques for finding errors within bison grammars. In this chapter, we turn our attention to the other side of error detection—how the parser and lexical analyzer detect errors. This chapter presents some techniques to incorporate error detection and reporting into a parser. We'll make a modified version of the SQL parser from Chapter 4 that demonstrates them.

Bison provides the `error` token and the `yyerror()` routine, which are typically sufficient for early versions of a tool. However, as any program begins to mature, especially a programming tool, it becomes important to provide better error recovery, which allows for detection of errors in later portions of the file, and to provide better error reporting.

Error Reporting

Error reporting should give as much detail about the error as possible. The default bison error declares only that it found a syntax error and stops parsing. In our examples, we used `yylineno` to report the line number. This provides the location of the error but does not report any other errors within the file or where in the specified line the error occurs. The bison locations feature, described later in this chapter, is an easy way to pinpoint the location of an error, down to the exact line and character numbers. In our example, we print out the locations, but precise location information would also allow a visual interface to highlight the relevant text.

It is often useful to categorize the possible errors, perhaps building an array of error types and defining symbolic constants to identify the errors. For example, in many languages a common error is to fail to terminate a string. Another error might be using the wrong type of string (a quoted string instead of an identifier, or vice versa). A parser might detect the following:

- General syntactic errors (e.g., a line that makes no sense)
- A nonterminated string
- The wrong type of string (quoted instead of unquoted, or vice versa)
- A premature end-of-file within a comment or other item that should have a terminator
- Names with multiple definitions, or names used but not defined

The duty for error detection does not lie with bison alone, however. Many fundamental errors are better detected by the lexer. For instance, the normal quoted string matching pattern is as follows:

```
\"[^\"\n]*\"
```

We would like to detect an unterminated quoted string. One potential solution is to add a new rule to catch unterminated strings as we did in the SQL parser in Chapter 4. If a quoted string runs all the way to the end of the line without a closing quote, we print an error:

```
\"[^\"\n]*\"  {
                yylval.string = yytext;
                return QSTRING;
              }
\"[^\"\n]*$ {
                warning("Unterminated string");
                yylval.string = yytext;
                return QSTRING;
            }
```

This technique of accepting not quite valid input and then reporting it with an error or warning is a powerful one that can be used to improve the error reporting of the compiler. If we had not added this rule, the compiler would have reported the generic "syntax error" message; by reporting the specific error, we can tell the user precisely what to fix. Later in this chapter, we will describe ways to resynchronize and attempt to continue operation after such errors.

The bison equivalent of accepting erroneous input is demonstrated by testing for the improper use of a quoted string for an identifier. For example, in MySQL it can be easy to confuse a quoted string in single forward quotes, `'string'`, with a quoted name in back quotes, `` `name` ``. In contexts where only one is valid, you can add a rule for the other and diagnose it in detail. Here's a version of the `column_list` rule that is used as the target of `SELECT ... INTO`:

```
column_list: NAME { emit("COLUMN %s", $1); free($1); $$ = 1; }
  | STRING        { yyerror("Column name %s cannot be a string", $1);
                    emit("COLUMN %s", $1); free($1); $$ = 1; }
  | column_list ',' NAME   { emit("COLUMN %s", $3); free($3); $$ = $1 + 1; }
  | column_list ',' STRING { yyerror("Column name %s cannot be a string", $3);
                             emit("COLUMN %s", $3); free($3); $$ = $1 + 1; }
  ;
```

If the user types a string rather than a name, it calls `yyerror()` to report it, and then it goes ahead pretending the string was a name.

Some simple flex hackery can let you produce better error reports than the rather dull defaults. A very simple technique that we used in the SQL parser reports the line number and current token. The `yylineno` option automatically increments the line number on each `\n` character, and the current token is always available in `yytext`, so a simple but useful error routine would be the following:

```
void yyerror(char *s)
{
```

```
            printf("%d: %s at %s\n", yylineno, s, yytext);
      }
```

A slightly more complex trick saves the input a line at a time:

```
      %code {
      char linebuf[500];
      %}
      %%
      \n.*  { strncpy(linebuf, yytext+1, sizeof(linebuf)); /* save the next line */
              yyless(1);      /* give back all but the \n to rescan */
            }
      %%

      void yyerror(char *s)
      {
           printf("%d: %s at %s in this line:\n%s\n",
                 lineno, s, yytext, linebuf);
      }
```

The pattern `\n.*` matches a newline character and the entire next line. The action code saves the line, and then it gives it back to the scanner with `yyless()`.

To pinpoint the exact position of an erroneous token in the input line, we need to use locations.

Locations

The bison *locations* feature associates a section of the input file identified by line and column numbers with every symbol in the parser. Locations are stored in `YYLTYPE` structures, which by default are declared as follows:

```
typedef struct YYLTYPE
{
  int first_line;
  int first_column;
  int last_line;
  int last_column;
} YYLTYPE;
```

Later we'll see how to override this if, for example, you want to add a filename or other extra information to each location.

The lexer sets the location for each token it returns, and every time it reduces a rule, the parser automatically sets the location of the newly created LHS symbol to run from the beginning of the first RHS symbol to the end of the last RHS symbol. Within action code in the parser, you can refer to the location of the LHS symbol as `@$` and the RHS symbols as `@1`, `@2`, and so forth. The lexer has to put the location information for each token into `yylloc`, which the parser defines each time it returns a token.

Fortunately, we can do this without having to add code to each lexer action.

Adding Locations to the Parser

Bison automatically adds the location code to the parser if it sees a reference to an `@N` location in the action code, or you can put the `%locations` declaration in the declaration part of the program.

We also need to change the error routines to use the location information. In our SQL example, we have both `yyerror()`, which uses the current location in `yylloc`, and a new routine `lyyerror()`, which takes an extra argument, which is the location of the error. In both cases, it prints out the location information (if any) before the error report.

```
 /* in code section at the end of the parser */
void
yyerror(char *s, ...)
{
  va_list ap;
  va_start(ap, s);

  if(yylloc.first_line)
    fprintf(stderr, "%d.%d-%d.%d: error: ", yylloc.first_line, yylloc.first_column
            yylloc.last_line, yylloc.last_column);
  vfprintf(stderr, s, ap);
  fprintf(stderr, "\n");

}

void
lyyerror(YYLTYPE t, char *s, ...)
{
  va_list ap;
  va_start(ap, s);

  if(t.first_line)
    fprintf(stderr, "%d.%d-%d.%d: error: ", t.first_line, t.first_column,
            t.last_line, t.last_column);
  vfprintf(stderr, s, ap);
  fprintf(stderr, "\n");
}
```

Note the defensive check for a nonzero `first_line` value; a rule with an empty RHS uses the location information of the previous item in the parse stack.

Within the parser proper, we change all the `yyerror` calls to `lyyerror` to report the appropriate token. For example:

```
column_list: NAME          { emit("COLUMN %s", $1); free($1); $$ = 1; }
  | STRING                 { lyyerror(@1, "string %s found where name required", $
                             emit("COLUMN %s", $1); free($1); $$ = 1; }
  ...
select_opts:               { $$ = 0; }
```

```
| select_opts ALL          { if($$ & 01) lyyerror(@2,"duplicate ALL option"); $$ =
  ...
insert_asgn_list:
     NAME COMPARISON expr  { if ($2 != 4) {
                             lyyerror(@2,"bad insert assignment to %s", $1); YYERR
                             }
                             emit("ASSIGN %s", $1); free($1); $$ = 1;
                           }
```

That's all we need to do to the parser, since the default location update code does the right thing for us.

Adding Locations to the Lexer

Since locations need to report the line and column range for errors, the lexer needs to track the current line and column each time it scans a token and return that information to the parser in `yylloc`. Fortunately, a little-known feature called *`YY_USER_ACTION`* makes that very simple. If you define the macro `YY_USER_ACTION` in the first part of your lexer, it will be invoked for each token recognized by `yylex`, before calling the action code. We define a new variable, `yycolumn`, to remember the current column number, and we define `YY_USER_ACTION` as follows in the definition section of the lexer:

```
%code {
/* handle locations */
int yycolumn = 1;

#define YY_USER_ACTION yylloc.first_line = yyloc.last_line = yylineno; \
    yylloc.first_column = yycolumn; yylloc.last_column = yycolumn+yyleng-1; \
    yycolumn += yyleng;
%}
```

Since `yyleng`, the length of the token, is already set, we can use that to fill in `yylloc` and update `yycolumn`. In a few cases (comments and whitespace), the token isn't returned to the parser and the lexer keeps going, but it doesn't hurt to fill in `yylloc` anyway. This takes care of the vast majority of location bookkeeping.

The last thing we have to do is to reset `yycolumn` to 1 whenever there's a newline. (Flex already handles `yylineno` for us.)

```
NOT[ \t]+EXISTS { yylval.subtok = 1; return EXISTS; }

ON[ \t]+DUPLICATE { return ONDUPLICATE; } /* hack due to limited lookahead */

<COMMENT>\n     { yycolumn = 1; }
<COMMENT><<EOF>> { yyerror("unclosed comment"); }

[ \t]           /* whitespace */
\n              { yycolumn = 1; }
```

In each pattern that can match a newline, we've separated out the `\n` into a separate pattern and set `yycolumn` to 1. We've also simplified the patterns for `NOT EXISTS` and `ON DUPLICATE` so they don't allow newlines. The alternative would be to manually rescan the tokens to check for newlines and set `yycolumn` to the number of characters after the newline.

That's enough to report errors with the exact line and column numbers. Since it's so easy to do, there's little reason not to use locations in your bison parsers even if you don't need the exact column numbers of each token and rule.

More Sophisticated Locations with Filenames

Most of the parsers we've written can handle more than one input file. How hard would it be to include the filename in the location data? Not very, it turns out. We have to define our own `YYLTYPE` that includes a pointer to the filename. We redefine the parser macro `YYLLOC_DEFAULT` that combines the location information when the parser reduces a rule, change the code in `YY_USER_ACTION` in the lexer to put the filename into `yylloc` for each token, and make a few other small changes to remember the name of the file the parser is reading. We add this section to the definition section of the parser:

```
%code requires {

char *filename; /* current filename here for the lexer */

typedef struct YYLTYPE {
  int first_line;
  int first_column;
  int last_line;
  int last_column;
  char *filename;
} YYLTYPE;
# define YYLTYPE_IS_DECLARED 1 /* alert the parser that we have our own definition

# define YYLLOC_DEFAULT(Current, Rhs, N)                                \
    do                                                                  \
      if (N)                                                            \
        {                                                               \
          (Current).first_line   = YYRHSLOC (Rhs, 1).first_line;        \
          (Current).first_column = YYRHSLOC (Rhs, 1).first_column;      \
          (Current).last_line    = YYRHSLOC (Rhs, N).last_line;         \
          (Current).last_column  = YYRHSLOC (Rhs, N).last_column;       \
          (Current).filename     = YYRHSLOC (Rhs, 1).filename;          \
        }                                                               \
      else                                                              \
        { /* empty RHS */                                               \
          (Current).first_line   = (Current).last_line   =              \
            YYRHSLOC (Rhs, 0).last_line;                                \
          (Current).first_column = (Current).last_column =              \
            YYRHSLOC (Rhs, 0).last_column;                              \
          (Current).filename  = NULL;                        /* new */ \
        }                                                               \
```

```
    while (0)
}
```

Rather than try to write the structure and macro from scratch, I looked at the generated C code for the first version of the parser with locations, copied the definitions of `YYLTYPE` and `YYLLOC_DEFAULT`, and modified them a little. The default declaration of `YYLTYPE` is enclosed in `#if !YYLTYPE_IS_DECLARED`, and the default declaration of `YYLLOC_DEFAULT` is enclosed in `#ifndef YYLLOC_DEFAULT`, so our new versions have to define them to turn off the default versions.

This code is enclosed in a `%code requires { }` block. The normal `%code { %}` block puts the code after the default definition of `YYLTYPE`, which is too late in the generated C program, and doesn't put a copy on the generated header file. The `requires` tells bison to copy the code ahead of the default versions and also into the header file.

This version of `YYLTYPE` includes the four standard fields, as well as a pointer to the filename. Then comes a definition of `YYLTYPE_IS_DECLARED` to prevent the standard version of `YYLTYPE`.

The long `YYLLOC_DEFAULT` macro copies location information from the RHS of a rule to the new LHS symbol. The three arguments to the macro are `Current`, the location information for the LHS; `Rhs`, the address of the first RHS location structure; and `N`, the number of symbols on the RHS. The internal macro `YYRHSLOC` returns the location structure for a particular RHS symbol. If `N` is nonzero, that is, there's at least one RHS symbol, it copies the relevant information from the first and Nth symbols. The `do ... while(0)` is a C idiom to make the macro expansion a statement that will parse correctly when the macro is followed by a semicolon. (Remember that there's no semicolon after the `}` at the end of a block such as the `else` clause here.) Only the two lines marked `new` are new; the rest is copied from the default `YYLLOC_DEFAULT`.

Having added the filename to the `YYLTYPE` structure, we add small amounts of code to `yyerror` and `lyyerror` to report the filename and to `main()` to set `filename` to the filename or the string `(stdin)` before starting the parser.

```
void
yyerror(char *s, ...)
{
  va_list ap;
  va_start(ap, s);

  if(yylloc.first_line)
    fprintf(stderr, "%s:%d.%d-%d.%d: error: ", yylloc.filename, yylloc.first_line,
     yylloc.first_column, yylloc.last_line, yylloc.last_column);
  vfprintf(stderr, s, ap);
  fprintf(stderr, "\n");

}
```

```
void
lyyerror(YYLTYPE t, char *s, ...)
{
  va_list ap;
  va_start(ap, s);

  if(t.first_line)
    fprintf(stderr, "%s:%d.%d-%d.%d: error: ", t.filename, t.first_line,
     t.first_column, t.last_line, t.last_column);
  vfprintf(stderr, s, ap);
  fprintf(stderr, "\n");
}

main(int ac, char **av)
{
  extern FILE *yyin;

  if(ac > 1 && !strcmp(av[1], "-d")) {
    yydebug = 1; ac--; av++;
  }

  if(ac > 1) {
    if((yyin = fopen(av[1], "r")) == NULL) {
      perror(av[1]);
      exit(1);
    }
    filename = av[1];
  } else
    filename = "(stdin)";

  if(!yyparse())
    printf("SQL parse worked\n");
  else
    printf("SQL parse failed\n");
} /* main */
```

The change to the lexer adds just one line—just copy `filename` into `yylloc.filename` in the `YY_USER_ACTION` macro:

```
#define YY_USER_ACTION yylloc.filename = filename; \
    yylloc.first_line = yylloc.last_line = yylineno; \
    yylloc.first_column = yycolumn; yylloc.last_column = yycolumn+yyleng-1; \
    yycolumn += yyleng;
```

Now our compiler reports the filename and the line and column. In a compiler with `include` statements that switch files within a single parse, the reports with this technique wouldn't be completely accurate, since they would report the first filename only if an error spanned input from two files, but the additional code to remember two filenames in `YYLTYPE` should be obvious.

Error Recovery

We concentrated on error reporting in the previous section; in this section, we discuss the problem of error recovery. When an error is detected,

the bison parser is left in an ambiguous position. It is unlikely that meaningful processing can continue without some adjustment to the existing parser stack.

Depending on the environment in which you'll be using your parser, error recovery may not always be necessary if the environment makes it easy to correct the error and rerun the parser. In other environments such as a compiler, it may be possible to recover from the error enough to continue parsing and look for additional errors, stopping the compiler at the end of the parse stage. This technique can improve the productivity of the programmer by shortening the edit-compile-test cycle, since several errors can be repaired in each iteration of the cycle.

Bison Error Recovery

Bison has some provisions for error recovery, which are available by using the special-purpose `error` token. Essentially, the `error` token is used to find a *synchronization point* in the grammar from which it is likely that processing can continue. That's *likely*, not certain. Sometimes attempts at recovery will not remove enough of the erroneous state to continue, and the error messages will cascade. Either the parser will reach a point from which processing *can* continue or the entire parser will abort.

After reporting a syntax error, a bison parser discards symbols from the parse stack until it finds a state in which it can shift an `error` token. It then reads and discards input tokens until it finds one that can follow the `error` token in the grammar. This latter process is called *resynchronizing*. It then resumes parsing in a *recovering* state, which doesn't report subsequent parse errors. Once it has shifted three tokens successfully, it presumes that recovery is complete, leaves the recovering state, and resumes normal parsing.

This is the basic "trick" to bison error recovery—attempting to move forward in the input stream far enough that the new input is not adversely affected by the older input.

Error recovery is easier with proper language design. Modern programming languages use statement terminators, which serve as convenient synchronization points. For instance, when parsing a C grammar, a logical synchronizing character is the semicolon. Error recovery can introduce other problems, such as missed declarations if the parser skips over a declaration looking for a semicolon, but these can also be included in the overall error recovery scheme.

In the SQL parser, the simplest place to resynchronize is at the semicolon at the end of each SQL statement. These rules added to the parser resynchronize at the semicolon that terminates each statement:

```
stmt_list: error ';'       error in the first statement
   | stmt_list error ';'   error in a subsequent statement
```

```
    ;
```

The potential for cascading errors caused by lost state (e.g., discarded variable declarations) can make a strategy that throws away large portions of the input stream ineffective. One mechanism for counteracting the problem of cascading errors is to count the number of error messages reported and abort the compilation process when the count exceeds some arbitrary number. For example, some C compilers abort after reporting 10 errors within a file.

Like any other bison rule, one that contains `error` can be followed with action code. One could clean up after the error, reinitialize data state, or otherwise recover to a point where processing can continue. For example, the previous error recovery fragment might say the following:

```
stmt_list: error ';'     { yyerror("First statement discarded, try again"); }
   | stmt_list error ';' { yyerror("Current statement discarded, try again"); }
   ;
```

Freeing Discarded Symbols

Bison's error recovery involves popping symbols off the internal parse stack. Each symbol can have a semantic value, and if those semantic values contain pointers to allocated storage or data structures, storage leaks and data corruption can occur. The `%destructor` declaration tells bison what to do when it pops a symbol with a semantic value. Its syntax is as follows:

```
%destructor { ... code ... } symbols or <types>
```

This tells the parser to execute the code each time it pops one of the named symbols or a symbol whose value is of the given type. There can be as many `%destructor` declarations as there are different treatments of discarded symbols. The type `<*>` is a catchall for any type of symbol with a defined type but no other destructor.

In our SQL parser, the only symbols that need special treatment are the ones with `<strval>` values, which are just strings that need to be freed:

```
  /* free discarded tokens */
%destructor { printf ("free at %d %s\n",@$.first_line, $$); free($$); } <strval>
```

This code reports what it's doing, including the location reference, which is useful for debugging but perhaps overkill for a production parser.

Error Recovery in Interactive Parsers

When a bison parser is designed to read directly from the console, a few tricks can smooth the error recovery. A typical parser reads a sequence of commands:

```
commands:      /* empty */
   |        commands command
   ;

command:        . . .
   |        error {
                  yyclearin /* discard lookahead */
                  yyerrok;
                  printf("Enter another command\n");
                  }
   ;
```

The macro `yyclearin` discards any lookahead token, and `yyerrok` tells the parser to resume normal parsing, so it will start anew with the next command the user types.

If your code reports its own errors, your error routines can use the bison macro `YYRECOVERING()` to test whether the parser is trying to resynchronize, in which case you shouldn't print any more errors, for example:

```
warning(char *err1, char *err2)
{
       if (YYRECOVERING() )
              return;      /* no report at this time */
        . . .
}
```

Where to Put Error Tokens

The proper placement of error tokens in a grammar is a black art with two conflicting goals. You want make it likely that the resynchronization will succeed, so you want error tokens in the highest-level rules in the grammar, maybe even the start rule, so there will always be a rule to which the parser can recover. On the other hand, you want to discard as little input as possible before recovering, so you want the error tokens in the lowest-level rules to minimize the number of partially matched rules the parser has to discard during recovery. The most practical recovery points are places where punctuation delimits elements of a list.

If your top-level rule matches a list (e.g., the list of statements in the SQL parser) or a list of declarations and definitions in a C compiler, you can make one of the alternatives for a list entry contain `error`, as in the previous command and SQL examples. This applies equally for relatively high-level lists such as the list of statements in a C function.

For example, since C statements are punctuated by semicolons and braces, in a C compiler you might write this:

```
     stmt:         . . .
           |       RETURN expr ';'
           |       '{' opt_decls stmt_list '}'
           |       error ';'
```

```
            |       error '}'
            ;
```

The two `error` rules tell the parser that it should start looking for the next statement after a `;` or `}`.

You can also put error rules at lower levels, for example, as a rule for an expression, but unless the language provides punctuation or keywords that make it easy to tell where the expression ends, the parser can rarely recover at such a low level.

Compiler Error Recovery

In the previous section we described the mechanisms bison provides for error recovery. In this section we discuss external recovery mechanisms provided by the programmer.

Error recovery depends upon semantic knowledge of the grammar rather than just syntactic knowledge. This greatly complicates complex recovery within the grammar.

It may be desirable for the recovery routine to scan the input and, using a heuristic, perform appropriate error recovery. For instance, a C compiler writer might decide that errors encountered during the declaration section of a code block are best recovered from by skipping the entire block rather than continuing to report additional errors. She might also decide that an error encountered during the code section of the code block need only skip to the next semicolon. A truly ambitious writer of compilers or interpreters might want to report the error and attempt to describe potential correct solutions. Some recovery schemes have tried to insert new tokens into the input stream, based on what the parser would have been able to accept at the point where the error was detected. They might do some trial parses to see whether the proposed correction does indeed allow the parser to keep reading from the input.

There is a great deal of literature on error correction and recovery, most dating from the era of batch computation when the time between program runs might be measured in hours or days, and compiler developers tried to guess what sorts of errors programmers might make and how to fix them. I can report from experience that they didn’t guess very well, and errors other than the most trivial invariably baffled the correction schemes. On today’s computers, the interval is more likely to be seconds, so rather than trying to guess the programmer’s intentions and continue after severe errors, it makes more sense to recover as quickly as possible to a state where the programmer can revise the input and rerun the compiler.

Exercises

1. Add error recovery to the calculator in Chapter 3. The most likely place to recover is at the `EOL` token at the end of each statement. Don't forget destructors to free up ASTs, symbols, and symbol lists.
2. (Term project.) Bison's error recovery works by discarding input tokens until it comes up with something that is syntactically correct. Another approach inserts rather than discards tokens, because in many cases it is easy to predict what token must come next. For example, in a C program, every `break` and `continue` must be followed by a semicolon, and every `case` must be preceded by a semicolon or a close brace. How hard would it be to augment a bison parser so that in the case of an input error it can suggest appropriate tokens to insert? You'll need to know more about the insides of bison for this exercise. The bison parser skeleton has some undocumented code that tries to suggest valid tokens you can start with.