Generating LR Syntax Error Messages from Examples

CLINTON L. JEFFERY
New Mexico State University

LR parser generators are powerful and well-understood, but the parsers they generate are not suited to provide good error messages. Many compilers incur extensive modifications to the source grammar to produce useful syntax error messages. Interpreting the parse state (and input token) at the time of error is a nonintrusive alternative that does not entangle the error recovery mechanism in error message production. Unfortunately, every change to the grammar may significantly alter the mapping from parse states to diagnostic messages, creating a maintenance problem.

Merr is a tool that allows a compiler writer to associate diagnostic messages with syntax errors by example, avoiding the need to add error productions to the grammar or interpret integer parse states. From a specification of errors and messages, Merr runs the compiler on each example error to obtain the relevant parse state and input token, and generates a `yyerror()` function that maps parse states and input tokens to diagnostic messages. Merr enables useful syntax error messages in LR-based compilers in a manner that is robust in the presence of grammar changes.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Translator writing systems and compiler generators*

General Terms: Languages

Additional Key Words and Phrases: Syntax error messages, LR parsers

1. INTRODUCTION

LR parser generators have stood the test of time because they solve the parsing problem well. LR parsing is efficient, the underlying theory is sound, and many implementations are available. Unfortunately, mainstream LR parser generators do not provide much support for the production of syntactic error messages. This paper describes methods for incorporating error messages in LR parsers, advocates a particular method which does not modify the base grammar specification, and then presents a tool that generates an error message function from a declarative specification of syntax errors and corresponding messages.

Merr (pronounced *mare*, from *meta error generator*) is a tool that generates an error message function `yyerror()` usable with Berkeley YACC,

This work was supported in part by the National Library of Medicine, Specialized Information Services Division and NSF Grant EIA-0220590.
Author's address: Department of Computer Science, New Mexico State University, Box 30001 MSC CS, Las Cruces, NM 88011; email: jeffery@cs.nmsu.edu.

AT&T YACC, or Bison. This paper uses the term *YACC* generically to refer to any compatible implementation. Merr writes the error message function in either ANSI C/C++ or the Icon programming language [Griswold and Griswold 1997]. The Merr tools and user's guide can be downloaded from `http://unicon.sourceforge.net/merr/`.

1.1 Producing Syntax Error Messages using YACC

By default, YACC prints the message *syntax error* (or *parse error*) when parsing fails. The function `yyerror(s)` overrides this default behavior. The `yyerror(s)` function is called with the default message as its parameter except under rare circumstances such as a stack overflow, but `yyerror(s)` can easily replace the message or augment it to produce error messages that include the filename, line number, and token at which the error was discovered. The GNU C compiler [GNU-Project 2001] is a widely used compiler that emits messages of this minimalist flavor, such as

```
goof.c : 1 : parse error before '}' token
```

Expert programmers seldom need any more than this, but novice and intermediate programmers benefit from better error messages. A strong case has been made that compiler error messages should be targeted at nonexperts, and that improved error messages correlate to better programmer performance [Brown 1983], [Shneiderman 1982]. A good summary of compiler error message design can be found in Horning [1974].

1.2 Using Error Recovery to Produce Error Messages

One way to obtain better error messages in LR parsers is to utilize error recovery productions, adding grammar productions with the symbol `error` where errors are expected. The following rule encodes a diagnostic using a semantic action in an error production:

```
Lbrace : '{' | {error_code=MISSING_LBRACE;} error ;
```

In this case, if a left curly brace (`'{'`) is missing, the error production includes an assignment to a global variable `error_code` which can be used by `yyerror()` to produce an appropriate message. The global variable can be avoided by calling `yyerror()` explicitly, as illustrated by the following rule adapted from the GCC 3.0 Java grammar. An error after the token `package` (PACKAGE_TK) is a missing package name:

```
package_declaration : PACKAGE_TK error
     { yyerror ("Missing name"); yyerrok; } ;
```

The `yyerrok` macro prevents the usual implicit call to `yyerror("syntax error")`, since an explicit call with a better error message (*Missing name*, instead of *syntax error*) is made.

Annotating an entire grammar with such error productions and corresponding error messages has disadvantages. Adding one or two hundred error productions to a grammar clutters the specification and makes it less maintainable.

In the case of GCC's Java grammar, approximately 150 error productions were added to approximately 350 nonerror grammar rules. Since most programming languages change gradually during their normal lifespan, maintainability is an issue, even if it is not the only issue.

Inserting error messages directly into the grammar also ties the error recovery strategy unnecessarily to the error message system, possibly causing cascading error messages (and additional mechanisms to control them). Also, while a YACC specification of a context free grammar might be generic enough to be used for other purposes such as a pretty printer, or a syntax-directed editor, inserting hundreds of error production rules to support compiler error messages may make the YACC specification a less attractive prospect for code reuse.

In addition, adding many error productions without introducing reduce-reduce conflicts requires great care, and reduce-reduce conflicts are generally unacceptable in LR parsers. The widely quoted conventional wisdom is that error tokens should be used sparingly, limited to higher-level units such as statements, and common special cases such as a missing semicolon [Levine et al. 1992]. This conventional wisdom might explain why GCC's C compiler has ultra-simple error messages: error tokens occur on only 18 of some 380+ production rules in the GCC C grammar. Using error recovery to produce descriptive error messages runs directly contrary to this advice on limiting error recovery tokens.

2. SEPARATING ERROR MESSAGES FROM ERROR RECOVERY

To separate error message handling from error recovery, the `yyerror()` function can utilize the point in parsing at which it was called. LR parsing uses a finite automaton plus a stack during parsing as it reads one token at a time from the input. At any given point during parsing, the information that might be considered in selecting an error message consists of all the terminal and nonterminal symbols that are on the parse stack, plus the current token. For example, if the entire program consisted of six tokens *abcxyz*, at a given point in parsing, some of the tokens will have been shifted from the input onto the stack, some of the tokens on the stack may have been reduced to nonterminals, and the remainder, beyond the current token, have not yet been considered. After a sequence of shift and reduce operations that process the first three symbols of *abcxyz*, the parser might be looking at a stack with *aBc* and a remaining input of *xyz*, of which x is the current token and y and z have not yet been scanned.

In practice, the symbols are not actually placed on the stack; instead the stack consists of integer *parse states*, each of which can be thought of as either a set of items denoting possible successful forward parses from a given point in parsing or an abstraction of the symbols that have been seen previously. It is traditional to depict a point in parsing showing the stack on the left, and the current input on the right, in such a way that the two integers used for parsing (and in our case, error diagnosis) are in the middle. Figure 1 shows an LR parse example, in which the relevant information on which to base an error message are the integer parse state (119) and the current token (the integer labeled x).

Parse Stack (integer states)	Input String (tokens)
$0\{a\}7\{B\}32\{c\}\underline{119}$	$\underline{x}yz$

Fig. 1. An LR parse, showing a <state,token> of $<119, x>$.

The symbols, which are implicit in the parse state integers and therefore not actually present on the parse stack, are shown in curly braces. In the figure, the stack values [0, 7, 32, 119] implicitly encode the information that "aBc is on the stack".

The LR parse algorithm uses the pair of integers <state,token> to determine there is an error; the same pair of integers constitutes suitable information from which `yyerror()` can select an appropriate error message. The current token is defined in YACC's public interface and easily accessed via global variables `yychar`, `yytext`, and `yylval`, but the parse state is *not* part of YACC's public interface. For each implementation, the correct variable (with a name similar to `yystate`) is usually easy to identify by studying the generated code. Berkeley YACC has global variables that can be read from `yyerror()`, but AT&T YACC and Bison store the parse stack and state in local variables. A small trick is required in order for `yyerror()` to read the parse state, for example by adding the parse state as an extra parameter to `yyerror()`.

Given the parse state, the `yyerror()` function must map each <state,token> pair to an appropriate diagnostic message. Creating this mapping by hand involves studying the `y.output` file generated with YACC's `-v` option, to understand what symbols have been seen, and what is expected next, in each parse state.

2.1 Mapping Parse States to Error Messages

Parsers for most real programming languages have hundreds of states. For example, approximately 360 states occur in the parser for the Icon programming language, produced using AT&T YACC [Griswold and Griswold 1986]. Out of these hundreds of states, a substantial portion may not have any error message. If the action table says in a given parse state to reduce by some production rule for every possible input token, there is no need for a diagnostic message; this may occur somewhat more frequently in LALR tables than in canonical LR(1) tables due to table compression.

In any case, for most languages there remain hundreds of parse states that warrant error messages, and a suitable data structure is needed to represent this mapping. In the case of Icon, a source file `src/h/parserr.h` has a (sparse) table containing messages for 170+ of the some 360 states. Figure 2 shows some lines from `parserr.h`. The mapping ignores the current input token and considers only the parse state in deciding what error message to emit. Many states are not listed in the table and get a generic syntax error message.

Each time the Icon grammar is changed, the state numbers change, and `parserr.h` needs to be updated by a painful manual process of trial and error. In practice, this contributes to a culture in which changing the grammar is strongly discouraged and new language features are constrained. This approach

```
static struct errmsg {
   int e_state; /* parser state number */
   char *e_mesg; /* message text */
   } errtab[] = {
     0, "invalid declaration",
     1, "end of file expected",
     2, "invalid declaration",
    12, "missing semicolon",
    14, "link list expected",
    ...
```

Fig. 2. Icon's mapping from states to error messages.

to providing error messages is less than ideal for compiler maintenance and experimental work.

2.2 Looking at the Input Token Can Improve Error Messages

In many parse states, the same error message should be issued regardless of the current input token. For example, when a required token `foo` is missing, a *foo expected* message may be reasonable, independent of the input token. This is common enough that reasonable compilers (such as the Icon compiler) have based their error messaging solely on parse states.

However, utilizing the input token in producing error messages adds precision analogous to the increased power of LR(1) parsers over LR(0) parsers. In some circumstances, looking at the input token allows a compiler to report syntax errors with a better message, or a suggestion of how to fix the error. Where token `x` was expected, certain of the unexpected tokens may indicate common errors that can be explained. Consider the C code fragment

```
if (x = = y) ···
```

The parser detects an error when the first "=" has been shifted onto the parse stack, and the second "=" is the input token. A typical bad error message for this case might read `= unexpected` or `expression expected`. A better error message describing the parse state (`missing or bad right operand in assignment`) can be further improved for this particular token with a more specific error message like: `spaces are not allowed inside == operator`. Of course, the error message should not jump to conclusions. Maybe two assignments were intended and an operand is missing. A more conservative message might read `illegal space inside == operator or missing assignment operand`.

3. THE MERR TOOL

The virtual machine compiler for Unicon (`http://unicon.sourceforge.net`), a successor to the Icon programming language, uses a version of Berkeley YACC that was modified to generate Icon code [Pereda 2000]. Although Unicon's grammar is based on Icon, Berkeley YACC's parse states have no connection with AT&T YACC's parse states, so a new `parserr.h` equivalent was needed for Berkeley YACC and the Unicon grammar. Worse yet, during development the Unicon grammar was changing relatively frequently, creating a maintenance

problem for the task of mapping parse states to error messages. One or two wildly incorrect error messages demonstrated that the compiler's diagnostic messages would either be perpetually out of date, or they would drag down the rate of development. A manually maintained error message facility would defeat the main purpose of writing the compiler in Unicon, which was to reduce development time and cost of maintenance.

The Merr tool eliminates manual decoding of parse state integers. Instead, the compiler writer presents example errors as code fragments, and supplies corresponding diagnostic messages. Merr invokes the compiler separately on each example error to extract the relevant <state,token> pair of integer values. The declarative specification of example errors is independent of which LR parser implementation is in use; the only requirement is that the parse state must be ascertainable in order to pass it the `yyerror()` function. The Merr User's guide at `http://unicon.sourceforge.net/merr/merrguid.pdf` contains instructions for using Merr with AT&T YACC, Bison, and Berkeley YACC. To port to another LR parser generator, examine the generated parser to identify the variable holding the parse state and modify the `yyerror()` macro. In the case of Unicon, the use of Merr eliminated the initial, complete interpretation of some 360 states that was needed, as well as the subsequent reexamination of those states every time the grammar changes.

3.1 Building the Error Message Table

Merr first compiles and links a version of the compiler executable with a special `yyerror()` function that prints the <`yy_state`,`yychar`> tuple in error messages. Merr then writes out each error fragment to a source file one at a time, and for each fragment it invokes the compiler, reads the error message to obtain the parse state and input token at the time of the error, and inserts that <state,token> tuple into a table analogous to the action table used in LR parsing, with the corresponding diagnostic. Merr must be rerun each time the grammar changes, since states can change dramatically even for relatively modest grammar changes.

The Merr program writes the table as code, along with a `yyerror()` function that produces an error message similar to the GCC message in Section 1.1, but with the improved diagnostic messages obtained via table lookup on the <state,token>. The `yy_state` subscript indexes into an array of unions; for each parse state the union may contain (a) no diagnostic, (b) a single error message to be used for all input tokens, or (c) a default message and a sparse array (with lookups based on the current input token, `yychar`) of custom messages for specific input tokens. The latter case is used when multiple fragments generate errors with the same parse state, but different input tokens producing different diagnostic messages.

The defaulting behavior makes it very easy to "grow" an error message set from generic messages to reasonable parse state-based messages, and finally to good messages that consider the current token. Nothing prevents the Merr user from producing incorrect messages, or failing to produce a message that will describe a particular error if no fragment for that parse state is supplied;

the Merr tool just simplifies the job of creating and maintaining the mapping of errors to messages.

3.2 Error and Message Specification

Merr reads error fragments and their corresponding messages as a sequence of *code*:::*diagnostic* pairs, where *code* may span several lines. The code fragment to generate an error is usually small, averaging less than 20 tokens thus far. The fragment must include all context (previous declarations or include directives, control structures, and so forth) that is necessary to put the parser into the state for which the diagnostic message is to be produced. The fragment may contain extra tokens beyond the point of error, which affect human readability but are not considered by the tool. The following are some example error fragments and associated messages:

```
int main{} ::: parenthesis or semi-colon expected
int x y; ::: missing comma in variable list
char () { } ::: function name expected
int a[] = {1,2; ::: unclosed initializer
struct foo
  int x;
::: missing { after struct label
```

The number of such error fragments may grow quite large. A lazy compiler writer can create fragments on a demand basis, starting from a generic GCC-style error message and adding more specific diagnostics as new parse states are identified by errant programs to achieve a "working set" of diagnostic messages. Alternatively, an initial set of error fragments can be created by studying the grammar and writing as many errant fragments for each production rule as possible. The important thing is that once a set of error fragments is written, changes in the grammar that change the parse state integers no longer require manual reexamination in order to avoid incorrect error messages.

Recursion in the grammar may allow the same parse state to be reached from many different nested levels in the code; a single error fragment suffices to produce the diagnostic for that parse state from any level of nesting in actual programs. For example, the following error fragment from the Unicon compiler:

```
procedure p() {
end
::: invalid compound expression
```

detects a parse error on the `end` token in a parse state attempting to produce a block enclosed by left and right braces. This same parse state occurs in more deeply nested code, such as

```
procedure main()
if y then {
  if x then { := 3
end
```

This example also illustrates the value of using the lookahead token. In this parse state, the illegal assignment (`:=`) operator can be reported using a better message by appending the following to the specification:

```
procedure main()
  { := 3
end
::: assignment missing its left operand
```

As before, this less nested fragment is sufficient to generate diagnostics for arbitrarily more nested errors that occur in the same parse state in actual programs.

4. RELATED WORK

Syntactic error *recovery* mechanisms have been studied extensively; see for example Graham et al. [1979]. The work presented in the current paper is is not about error recovery per se; as Section 2 points out, it is desirable to separate error message reporting from error recovery in order to allow maximum flexibility to the error recovery strategy. Although Merr generates error messages for conventional LR parsers with minimal error recovery mechanisms, more powerful error recovery mechanisms are related in that they can present the repaired program in the error message, as a suggestion of how the programmer may correct their program [Conway and Wilcox 1973].

AI techniques have been used to produce far superior error message facilities than those proposed in this work [Johnson 1986]. These facilities have been used in instructional settings, but rarely in production compilers. Similarly, computer-aided instruction (CAI) and programming environment tools have been sources of rich diagnostic facilities; this paper does not subsume such work—Merr is more of an attempt to raise the lowest common denominator.

Less work has been done to enable production of quality syntax error messages from parsers produced by LR parser generators. Sippo and Soisalon-Soininen [1983], Kantorowitz and Laor [1986], and Heirich [1989] built parsers that list what tokens would have been valid at an error site; Sippo and Soisalon-Soininen [1983] decides between a *token expected* message and a *nonterminal A can't start like this* message, depending on the parse state. These techniques are more fully automated than Merr, but the error messages are much less user-friendly than the messages a human can write in the Merr specification file, and do not allow for diagnostic distinction based on the current input token. Messages oriented toward listing what tokens would *not* have produced a syntax error at the point one was encountered can be unhelpful (say, if thirty different tokens would have been OK), and the extra reductions YACC may take before detecting the error make construction of such lists problematic in any case.

Recent sources of work on error messages can be found in advanced compiler construction systems such as ANTLR [Parr 2000] and Eli [Gray et al. 1992] which integrate multiple phases of compilers. Such tools may provide

good alternatives for those willing to give up YACC and learn them. ANTLR is notable for its use of exception handling to allow customization of the default parse error messages, and its ability to associate such messages with either individual production rules, or with groups of production rules that reduce to the same nonterminal symbol.

5. CONCLUSION

Merr is a concise solution to a problem in compilers that use LR parsers. It significantly eases the difficulty of providing diagnostic messages, and the difficulty of keeping those messages synchronized with parse states when the grammar changes. Compiler writers simply collect code fragments for each unique kind of syntax error, and write the most explanatory error message that they can invent for each one. Advocates of recursive descent parsers cite error messages as an advantage of their approach; a tool such as Merr can help raise LR parsers a notch closer to recursive descent parsers in this regard.

It is trivial to extend Merr to work with other C-based LR parser generators, and straightforward to port it to other languages for which LR parser generators are available. In the Merr source code there are about 40 calls to `write()` that contain the syntax of the error tables and the error function. These 40 output calls must be translated to use Merr with an LR parser for another language.

Further improvements to Merr are possible. Error fragments might be automatically generated from an analysis of the grammar for subsequent human annotation. A simple macro facility could reduce the size of the diagnostic messages. Merr might incorporate an analysis similar to Kantorowitz and Laor [1986] to provide a smarter default message when no custom message is applicable. Merr might be adapted for use by more novel LR-based parsers such as generalized LR parsers [Tomita 1986], by associating diagnostics with <set of parse states,token> tuples.

ACKNOWLEDGMENTS

The modified version of Berkeley YACC, called iyacc, was implemented by Ray Pereda. It is part of the Unicon source distribution, and is also usable with Icon. It is also available direct from its author; his e-mail is `raypereda@hotmail.com`. The example of supplying an error message in an error production via a global variable was due to Saumya Debray by personal correspondence. Alexandre Petit-Bianco provided a helpful clarification of the GCC Java example. Mikhail Auguston, Kay Robbins, and Saumya Debray, as well as the anonymous referees, provided helpful comments on this paper.

REFERENCES

BROWN, P. 1983. Error messages: the neglected area of the man/machine interface? *Commun. ACM 26*, 4, 246–249.

CONWAY, R. AND WILCOX, T. 1973. Design and implementation of a diagnostic compiler for PL/1. *Commun. ACM 16*, 3, 169–179.

GNU-PROJECT. 2001. *GNU C Compiler, Source Distribution, Version 3.0*. Available online at `www.gnu.org`. Source files `c-parse.y` and `java/parse.y` were used in this paper.

GRAHAM, S. L., HALEY, C. B., AND JOY., W. N. 1979. Practical LR error recovery. *ACM SIGPLAN Not. 14*, 8, 168–175.

GRAY, R. W., HEURING, V. P., LEVI, S. P., SLOANE, A. M., AND WAITE, W. M. 1992. Eli: A complete, flexible compiler construction system. *Commun. ACM 35*, 2, 121–130.

GRISWOLD, R. E. AND GRISWOLD, M. T. 1986. *The Implementation of the Icon Programming Language*. Princeton University Press, Princeton, NJ.

GRISWOLD, R. E. AND GRISWOLD, M. T. 1997. *The Icon Programming Language*, 3rd ed. Peer-to-Peer Communications, San Jose, CA.

HEIRICH, A. 1989. *Error Reporting in LR Parsers*. A comp.compilers posting, available online at `http://compilers.iecc.com/comparch/article/89-08-011`.

HORNING, J. 1974. What the compiler should tell the user. In *Compiler Construction: an Advanced Course*. Springer-Verlag, Berlin, Germany, 525–548.

JOHNSON, W. L. 1986. *Intention-Based Diagnosis of Novice Programming Errors*. Pitman, London.

KANTOROWITZ, E. AND LAOR, H. 1986. Automatic generation of useful syntax error messages. *Softw.: Pract. Exp. 16*, 7, 627–640.

LEVINE, J. R., MASON, T., AND BROWN, D. 1992. *lex & yacc*, 2nd ed. O'Reilly & Associates, Sepastopol, CA.

PARR, T. 2000. *ANTLR Reference Manual, Version 2.7.1*. jGuru, available online at `http://www.antlr.org/doc/antlr_2.7.1_ref_man.pdf`.

PEREDA, R. 2000. *iyacc: A Parser Generator for Icon*. Unicon Technical Report #3. Available online at `http://unicon.sourceforge.net/utr/utr3.pdf`.

SHNEIDERMAN, B. 1982. Designing computer system messages. *Commun. ACM 25*, 9, 610–611.

SIPPO, S. AND SOISALON-SOININEN, E. 1983. A syntax error handling technique and its experimental analysis. *ACM Trans. Program. Lang. Syst. 5*, 4, 656–679.

TOMITA, M. 1986. *Efficient Parsing for Natural Language : A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, Dordrecht, The Netherlands.

Received February 2002; revised August 2002; accepted March 2003