Merr User's Guide

CLINTON L. JEFFERY
New Mexico State University
jeffery@cs.nmsu.edu

July 25, 2002

Introduction

Merr (pronounced "mare", from "meta error generator") is a tool that generates an error message function yyerror() usable with Berkeley YACC, AT&T YACC, or Bison. Merr can generate the error message function for either ANSI C/C++ or the Icon programming language [Gris97]. The Merr tool can be downloaded from http://unicon.sourceforge.net/merr/. This document is the primary user's guide and reference for Merr. A separate document [Jeff02] describes Merr's underlying principles and motivation. You should read that document if you want to port Merr to work with other parser generators, implement features similar to Merr in your own tools, or better understand why Merr is superior to the old way of generating syntax error messages.

This document assumes full familiarity with the YACC family of tools. The term YACC will henceforth be used to denote the union of: AT&T YACC, Berkeley YACC, Bison, and compatible tools. If you are not familiar with YACC, you should go read [Levi92] or any of several other books or papers that describe it in detail.

Reasonable YACC applications supply a yyerror() function to override the default generic syntax error message and provide a useful message. Minimally, most YACC applications use yyerror(s) to augment the default message with the filename, line number, and token at which the error was discovered.

Merr is for YACC compiler writers who want to do better than this. 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 [Brow83], [Shne82]. A good summary of compiler error message design can be found in [Horn74].

With the Merr tool, you present example errors as code fragments, and supply corresponding diagnostic messages. Merr invokes your compiler separately on each example error to extract the parse state and input symbol at the time

the syntax error occurs. This declarative specification of example errors is independent of which LR parser implementation is in use.

The Merr Command Line

Merr is invoked from the directory in which the compiler is built, with the following command line arguments and options.

merr [-yYB] [-s make] [-o msgfile] compiler [target]

where *compiler* is the name of the compiler for whom error messages will be generated, and *target* is the source filename that the errant programs will be written to and compiled from, defaulting to m_err.c (or m_err.icn)

Merr executes a system command to rebuild the compiler in order to create its error message table. The default command is "make *compiler*". The -s option overrides this default, as in the line:

merr -s "make all" mycc bug.c

Merr writes the error message table and yyerror() function to a file named yyerror.c (or .icn) by default. A "-o" option directs Merr to write the file with a different name.

Command line options -y, -Y, and -B direct Merr to generate compatible C yyerror() functions for three popular C/C++ YACC implementations (the default is to generate a yyerror() function for Icon/Unicon). For AT&T YACC and Bison, Merr writes a header file yyerror.h that defines a macro to add the parse state information to the yyerror() function, appropriate to each version of YACC. This header file must be included in the YACC specification (.y) file and its generated y.tab.c file. Figure 3 shows the command line options and corresponding yyerror.h contents.

-B	Bison. #define yyerror(s) _yyerror(s,yystate)
-Y	AT&T YACC. #define yyerror(s) _yyerror(s,yy_state)
-y	Berkeley YACC. No yyerror.h required.

Figure 1: Options handle differences among C YACC implementations.

To port to another LR parser generator, examine the generated parser to identify the variable holding the parse state and modify the yyerror() macro.

Error and Message Specification

Rather than introduce a myriad of tiny files, Merr uses a single file named meta.err that contains all error fragments and their corresponding messages.

The file format is a sequence of code:::diagnostic pairs, where code can be one or multiple lines. The code fragment to generate an error is usually small, but must include as much context (previous declarations, control structures, and so forth) as is necessary to put the parser into the state for which the diagnostic message is to be produced. The diagnostic error message is normally a single line but is extended when a line ends with a backslash. 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. 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.

Merr's version of yyerror()

The Merr program generates a yyerror() function with the following pseudo-C template:

```
void yyerror(char *s, int parse_state)
{
    if (yyerrors++ > yymaxerrors)
        stop("too many errors, aborting");
    if (!strcmp(s, "syntax error"))
        s = yyerrmsg[parse_state, yychar];
    fprintf(stderr, "%s:%d: # \"%s \":%s\n",
            yyfilename, yylineno, yytext, s);
}
```

The table yyerrmsg[] is the key component that maps the parse_state to the most specific diagnostic message available. The table is sparse, and is not

really represented using a two-dimensional array, so the above pseudocode is a simplification. In reality the parse_state subscript looks up a union which may contain (a) no diagnostic, (b) a single error message to be used irrespective of the input token, or (c) a default message and a sparse array (with lookups based on yychar) of custom messages for specific input tokens.

Error Message Defaulting

If a parse state is not found in the message table, the default error message (usually "syntax error") is printed out with current line number and token information. If the parse state is present in the table, there may be a shared message (independent of the input token) or individual messages for specific input tokens.

If only one error fragment produced a given parse state, the yyerrmsg[] table returns that diagnostic in that parse state for all possible input tokens. If multiple fragments fail in the same parse state, the first one in the specification is used as the default, and the second and subsequent messages for that parse state override that default only for the input token appearing in their error fragment. If more than one fragment fails on identical parse states and input tokens, a warning message is produced and only the first one is used.

The default behavior makes it very easy to "grow" an error message set from free minimal GCC-style messages to reasonable parse state-based messages to good messages that consider the current token. Nothing prevents the Merr user from producing incorrect messages, or failing to produce an error fragment and custom message that will describe a particular error; the Merr tool just makes the job easier.

Conclusion

Merr is a simple, elegant solution to a common problem encountered in writing compilers using YACC. Using Merr allows you to improve your syntax error messages by supplying examples. Merr is robust in the presence of changes to the grammar; just rerun it whenever the grammar is changed.

Acknowledgements

This work was supported in part by the National Library of Medicine, Specialized Information Services Division. 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 provided helpful comments on this paper.

Appendix: Example Merr Specification File

This is an example Merr specification for some syntax errors in the Icon programming language. It is a subset of the Merr specification used by the Unicon programming language compiler.

```
procedure main()
every x do { }
}
end
::: too many closing curly braces

global::: unexpected end of file
global x y::: invalid global declaration
global x, , y::: missing identifier
procedure p(x) end::: missing semicolon
link procedure p(x)
end
::: link list expected
invocable procedure p(x)
end
::: invocable list expected
local x
::: invalid declaration
procedure main()
    a +
end
::: missing or invalid second argument to +
procedure main()
    a *
end
::: missing or invalid second argument to *
procedure main()
    a !
end
::: missing or invalid second argument to !
```

```
procedure ()
end
::: procedure name expected
procedure p(1)
end
::: parameter name expected
procedure p(x,)
end
::: parameter name expected
procedure p(x)
global x
end
::: semicolon expected
procedure p(x);
global x
end
::: invalid procedure body
procedure p()
!
end
::: invalid argument to unary !
procedure p()
create
end
::: invalid create expression
procedure p()
{
end
::: invalid compound expression
procedure p()
if
end
::: invalid if control expression
procedure p()
case
end
::: invalid case control expression
procedure p()
while
end
::: invalid while control expression
```

```
procedure p()
until
end
::: invalid until control expression
procedure p()
every
end
::: invalid every control expression
procedure p()
repeat
end
::: invalid repeat control expression
link x+
procedure p()
end
::: invalid link declaration
procedure p
write()
end
::: missing parameter list in procedure declaration
procedure p()
local ”hello”
end
::: invalid local declaration
procedure p()
initial ]
end
::: invalid initial expression
procedure p()
if (1) {
hello
}
end
::: missing then
procedure p()
write(p()
hello
end
::: unclosed parenthesis
procedure p()
local l := []
```

```
end
::: illegal assignment in declaration
procedure p()
if \a.b \— (\a.b & c(e) == "\t" then {
end
::: unclosed parenthesis
procedure main()
    case x of {
        y:
            f(); g()
    }
end
::: malformed case expression
procedure main()
    case x of {
        case y:
            f()
        }
end
::: missing "of" in case expression
procedure main()
    while x do { x
end
::: a while loop is missing } somewhere before "end"
procedure p()
initial {
    if foo then {
    }
end
::: an "initial" clause is missing } somewhere before "end"
procedure dbdelete(db, filter)
sql(db, "DELETE FROM " —— blah blah —— filter)
end
::: unclosed literal or missing operator
procedure p()
s —— := k
end
::: missing operand after —— or illegal space inside ——:=
```

References

[Brow83] Brown, P. Error messages: the neglected area of the man/machine interface? *Communications of the ACM*, 26(4):246–249, 1983.

[Gris97] Griswold, R. E. and Griswold, M. T. *The Icon Programming Language, 3rd edition*. Peer-to-Peer Communications, San Jose, CA, 1997.

[Horn74] Horning, J. What the compiler should tell the user. In *Compiler Construction: an Advanced Course*, pages 525–548, Berlin, 1974. Springer Verlag.

[Jeff02] Jeffery, C. Generating LR syntax error messages from examples. *submitted for publication*, 2002.

[Levi92] Levine, J. R., Mason, T., and Brown, D. *lex & yacc, 2nd ed.* O'Reilly & Associates, Sepastopol, CA, 1992.

[Shne82] Shneiderman, B. Designing computer system messages. *Communications of the ACM*, 25(9):610–611, 1982.