

Ytracc: a Parse Browser for Yacc Grammars

RICHARD FURUTA, P. DAVID STOTTS AND JEFFERSON OGATA

*Department of Computer Science, University of Maryland, College Park, MD 20742,
U.S.A.*

SUMMARY

We describe a program for the display and exploration of complex, domain-specific information: ytracc, an interactive grammar debugging tool for compiler writers. The ytracc system provides the designer of a yacc grammar a method of tracing a parser as it uses the grammar. ytracc captures the states of the parse as it is carried out. The captured parse can then be replayed forwards or backwards, step-by-step, or subtree-by-subtree, as defined by the non-terminals of the grammar. The tool has been successfully used by students as an assistant in an advanced undergraduate compiler construction class, and we use the tool in our everyday work.

KEY WORDS Grammatical specification Parser generators Specialized debuggers Visualization Yacc

1. INTRODUCTION

Visual representations are generally acknowledged to be valuable for aiding comprehension of complex topics. In this report we will describe a system, called ytracc, that produces an interactive graphical representation of the complex, domain-specific workings of a grammatically-defined parser; in other words, ytracc is a software system for visualization of the compilation process.

The ytracc system captures a representation of the data structures required for compiling as they are created and modified. This captured state can then be replayed to show the parse tree abstractly as tokens are recognized from the source string. The associated parse stack is seen growing and shrinking as the translation progresses. The ytracc system is based on the same context-free grammar description used by the well-known yacc parser generator;¹ indeed the state information is captured by automatically instrumenting a yacc-produced parser. Section 2 of this paper further describes the ytracc system in terms of its user interface and of its system design.

The visual representations produced by ytracc are intended to assist a compiler writer in development of a correct grammatical specification of a programming language. As the application is within the specialized area of language definition and compiler construction, ytracc is intended to help the experienced programmer, not the novice user. Nevertheless we find that ytracc is useful for helping advanced undergraduate students learn the techniques of compiler construction; we report on these experiences in Section 3.

Our original interface for ytracc was textual. Based on our experiences with this version, we reimplemented the interface to be more graphical, although still displayable on a generic computer terminal. The resulting graphical interface is the one

that will be described in this report. Our experiences with ytracc have suggested additional modifications to this graphical interface. These modifications are discussed in [Section 4](#), which concludes the paper.

2. Ytracc: ILLUSTRATED yacc- BASED PARSING

This section will present the functionality of the ytracc system, first in terms of its user interface, and then in terms of its implementation.

2.1. Ytracc user interface

The primary element in the ytracc system is a module (also called ytracc) that instruments the code produced by the well-known Unix* parser-generator yacc.[†] † yacc generates a LALR(1) parser from context-free grammars specified in the appropriate form. A parser produced by ytracc is generated from a yacc source file (grammar), requiring no changes in its format. From the point of view of the compiler writer, one simply substitutes use of ytracc for yacc. Indeed in practice, we have replaced *all* use of yacc with ytracc, as ytracc reports some errors in specification accepted by yacc. In terms of data structures and behaviour, the ytracc -produced parser is functionally identical to the yacc- produced version. As parsing proceeds, the activity of the parsing automaton is recorded with the instrumentation added by ytracc. After parsing a sample string, the recorded parse history can be displayed and manipulated as an illustration of the behaviour of the grammar that generated the parser. Details of this two-phase procedure are presented below.

During parsing of an input string with a ytracc- generated parser an intermediate file is produced that records the history of actions and states encountered. A second tool, called yshow, is then used to navigate and display the history information. The functions performed by the display tool are summarized in [Table I](#).

After parsing an input string (program) the display tool will show the parsing activity as requested by the programmer. Here we will consider the application of a parser generated from the grammar shown in [Figure 1](#) to the small input string

$$\left(\begin{array}{l} (a)b \\ (c) \end{array} \right)$$

The grammar is a right-recursive formulation of Lisp-like lists. Grammar non-terminals are shown in capital letters and grammar terminals in lower-case letters.

```

1  LIST ::= lparen ELTS rpren
2  ELTS ::= atom ELTS
3  ELTS ::= LIST ELTS
4  ELTS ::=

```

Figure 1. Example grammar used in the text

* Unix is a registered trademark of AT&T Bell Laboratories.

† Throughout this paper, we mean 'Berkeley Unix, 4.3 bsd', when we refer to 'Unix'.

Table I. yshow commands. Some commands permit an optional argument, shown in square brackets in the table's entries. The default action is carried out when the ytracc user omits the argument

<cr	(carriage return or newline) forward one action backward one action
m[n]	forward n actions (default: one)
-[n]	backward n actions (default: one)
:[n]	go to absolute action number n (default: the current action)
s[t]	find next shift by the token named t (default: by any token)
S[t]	find previous shift by the token named t (default: by any token)
r[n]	find next reduce by the rule numbered n (default: by any rule)
R[n]	find previous reduce by the rule numbered n (default: by any rule)
e	find next error
E	find previous error
n	repeat previous 'find' command
N	reverse previous 'find' command
^R	redraw display
q	quit

state	ACTION				GOTO	
	lparen	rparen	atom	\$	LIST	ELTS
0	s2				1	
1				acc		
2	s2	r4	s4	r4	5	3
3		s6				
4	s2	r4	s4	r4	5	7
5	s2	r4	s4	r4	5	8
6	r1	r1	r1	r1		
7	r2	r2	r2	r2		
8	r3	r3	r3	r3		

Figure 2. yacc/ytracc parse tables for the example

Note the ϵ -production (rule 4). Figure 2 shows the parse tables generated by yacc. The format-of the tables is adopted from Reference 2.*

Assume that the compiler writer has compiled his ytracc -produced parser and executed it on the sample input. Invoking the yshow command will produce the display of Figure 3. The display contains five snapshots of the parse stack.† The centre snapshot represents the parse stack at the 'current' time; in this Figure, the time is the first of the 21 actions in the parse, as shown on the bottom line of the display. Each action in the parse represents a shift, a reduce, an accept or an error. Stacks displayed to the left of the current display show the preceding actions, and those to the right the subsequent actions. The look-ahead token is shown below the

* We initially considered providing a table of this format in ytracc but came to realize that the process of formatting the display would be difficult because of the large number of grammar terminals and non-terminals necessary to implement more realistic languages. An additional complication is that a grammar writer may prefer to use longer names for the grammar elements, thereby making the task of fitting a tabular display onto a relatively limited sized window even more difficult. In a multi-windowed workstation environment, a grammar writer can refer to the yacc-produced y.output, a tabular listing of the parse table, while running ytracc.

†The actual size of the display and the number of snapshots shown depends on the size of the display area and on the length of the token and non-terminal names used by the grammar writer.

```

13 | 13 | 13 | 13 | 13
12 | 12 | 12 | 12 | 12
11 | 11 | 11 | 11 | 11
10 | 10 | 10 | 10 | 10
 9 |  9 |  9 |  9 |  9
 8 |  8 |  8 |  8 |  8
 7 |  7 |  7 |  7 |  7
 6 |  6 |  6 |  6 |  6
 5 |  5 |  5 |  5 |  5
 4 |  4 |  4 |  4 |  4
 3 |  3 |  3 |  3 |  3
 2 |  2 |  2 |  2 |  2
 1 |  1 |  1 |  1 |  1 lparen
 0 |  0 |  0 |  0 lparen | 0 lparen
see | see | see | see | see
   |   | shift | shift | shift
-----
? for help >
line 1 ( a ) b
rule
step 1 of 21

```

Figure 3. yshow display, action 1

stack. Here the lparen token is highlighted, since it is about to be shifted, as shown in the final line of the stack's display.

The bottom three lines of the display show the current input line (with the lexeme corresponding to the look-ahead token highlighted), the current grammar rule in the case of a reduce action, and an indication of how far along we have progressed in the parse. A command prompt may be seen in the figure between the parse stacks and the status information. The default command, given by entering a carriage return ($\langle cr \rangle$), is to advance the display by one action. Entering the $\langle cr \rangle$ produces the display of Figure 4, reflecting the effect of performing the shift.

At this point, let us skip ahead three actions to the display of Figure 5 (action 5). (The stack for action 3, to the right of the current stack in Figure 4, is now at the left-hand edge of the display.) Action 5 is a reduce by grammar rule 2, and we note that in the case of a reduce, the handle is highlighted on the stack. As shown in the history, nothing was highlighted for action 4, however, because the reduction was by the grammar's ϵ rule.

As indicated by the command list of Table I, yshow allows forward and reverse browsing of the parse in steps of one action or of multiple actions. One can also go directly to a specific action, for example entering the command $':21 \langle cr \rangle$ shows the final action of the parse (see Figure 6). yshow also permits searching forward or backward for shift, reduce and error actions. For example, giving the command $'r \langle cr \rangle$ will move ahead to the next reduce action of any kind, whereas the command $'r2 \langle cr \rangle$ finds the next reduction by rule 2. Similarly $'Sparen \langle cr \rangle$ finds the preceding shift of an rparen. *

* When a command takes a separate argument, yshow prompts for the argument after the command character has been typed. The examples here show the sequence of keys that would be typed by the yshow user.

```

13      13      13      13      13
12      12      12      12      12
11      11      11      11      11
10      10      10      10      10
9       9       9       9       9
8       8       8       8       8
7       7       7       7       7
6       6       6       6       6
5       5       5       5       5
4       4       4       4       4
3       3       3       3       3
2       2       2       2       2 atom
1       1       1       1 lparen 1 lparen
0       0       0 lparen 0 lparen 0 lparen

see     see lparen  see lparen  see atom  see rparen
shift  shift      shift      shift      reduce 4
-----
? for help >
line 1 ( ( a ) b
rule
step 2 of 21
    
```

Figure 4. yshow display, action 2

```

13      13      13      13      13
12      12      12      12      12
11      11      11      11      11
10      10      10      10      10
9       9       9       9       9
8       8       8       8       8
7       7       7       7       7
6       6       6       6       6
5       5       5       5       5
4       4       4       4       4
3       3       3 ELTS  3 rparen
2       2 atom  2 atom  2 ELTS  2 ELTS
1 lparen 1 lparen 1 lparen 1 lparen 1 lparen
0 lparen 0 lparen 0 lparen 0 lparen 0 lparen

see     see rparen  see rparen  see rparen  see
shift  reduce 4   reduce 2   shift      reduce 1
-----
? for help >
line 1 ( ( a ) b
rule 2 ELTS : atom ELTS
step 5 of 21
    
```

Figure 5. yshow display, action 5

One final characteristic of yacc is worth mentioning in the context of ytracc and yshow. yacc permits actions to be specified at any point in the rule; for example in our sample grammar we might wish to include an action such as

```

LIST: lparen
      { /* action here */ }
ELTS rparen
    
```

```

13      13      13      13      13
12      12      12      12      12
11      11      11      11      11
10      10      10      10      10
9       9       9       9       9
8       8       8       8       8
7       7       7       7       7
6       6       6       6       6
5       5       5       5       5
4       4       4       4       4
3       3       3       3       3
2       2       2       2       2
1  ELTS  1  ELTS  1  ELTS  1  ELTS  1  ELTS
0  lparen 0  lparen 0  LIST 0  lparen 0  lparen
see rparen | see reduce 1 | see <EOF> | see | see
accept

-----
? for help >
line 4
rule
step 21 of 21

```

Figure 6. yshow display, final action

In yacc, such interior actions cause anew non-terminal to be generated, along with a new rule producing the empty string:

```

1 {1} ::=
2 LIST ::= lparen{1} ELTS rparen

```

These generated non-terminals are also tracked by ytracc and displayed by yshow (see Figure 7), which shows this modified specification near the end of a parse of '(a)'.

```

13      13      13      13      13
12      12      12      12      12
11      11      11      11      11
10      10      10      10      10
9       9       9       9       9
8       8       8       8       8
7       7       7       7       7
6       6       6       6       6
5       5       5       5       5
4       4       4       4       4
3  ELTS  3  rparen  3  rparen  3  rparen  3  rparen
2  atom  2  ELTS  2  ELTS  2  ELTS  2  ELTS
1  (1)  1  {1}  1  {1}  1  {1}  1  {1}
0  lparen 0  lparen 0  lparen 0  lparen 0  lparen
see rparen | see rparen | see | see <EOF> | see
reduce 3 | shift | reduce 2 | accept |
-----
? for help >
line 1 (a)
rule 2 LIST : lparen (1) ELTS rparen
step 7 of 8

```

Figure 7. yshow display, final action

2.2. Ytracc design and implementation

Since the ytracc system is an extension to yacc, we decided to implement it as a yacc post-processor, rather than as an entirely re-implemented parser generator. This approach ensures the greatest compatibility between the source formats of the two systems and the behaviours of the parsers produced by each. The ytracc program first creates a temporary directory to work in. It then invokes the standard Unix yacc program on the input grammar. After yacc completes, ytracc post-processes the C code in `y.tab.c` prior to compilation of a parser. The modified `y.tab.c` file is then moved to the original directory and the temporary space is cleaned up. As with yacc, the routines `yylex()`, `main()` and `yyerror()` must be provided by the author of a ytracc parser, and the yacc-standard versions of `main()` and `yyerror()` can be used if desired. This system structure is shown by the top half of the block diagram in [Figure 8](#).

Naturally, ytracc handles all the options that yacc can have. It has, in addition, a `-g` option to generate a condensed grammar report called `y.tab.g`, which can then be consulted during parse visualization. This report is a modified form of the input grammar, and is in essentially the same form as the grammar examples shown earlier (e.g. the grammar of [Figure 1](#)). To obtain the modified form of the input grammar, semantics actions are stripped out, and productions are formatted and numbered to correspond to the reduction numbers shown by the visualization filter. In addition, the ‘phantom’ productions that yacc creates to trigger semantic actions in the middle of user-supplied productions are explicitly represented in the grammar report, with unique non-terminal names (e.g. ‘{ 12 }’) generated from their reference numbers. Care must therefore be exercised by a ytracc user to avoid explicit productions with non-terminal names of this form if the grammar report is to be useful in visualizing a parse trace.

Once compiled, a ytracc parser will produce a parse history each time it is used, capturing a trace of the events (reductions, token shifts, errors, accept) that occur while parsing a particular input string. This trace file, called `ytrace.out`, contains information that allows the visualization filter to reconstruct the original lexeme string, the token stream constructed from it and the resulting parse tree. The bottom half of [Figure 8](#) illustrates the structure and use of `yshow`, and shows its relationship to ytracc. In the parse trace, each parsing action is numbered, a code recorded for its kind (shift, reduce, error, accept), and action-specific data such as production number and non-terminal, or token name shifted, is saved. The filter `yshow` uses this information to construct the displays shown earlier. For general portability, cursor control and highlighting is performed using the standard Unix `curses` package.³ `yshow` scans the trace file to determine the longest token or non-terminal name, and then divides the display window accordingly into the largest odd number of columns that will properly fit. Thus a larger historical context will be shown in a wider window.

The inner workings of ytracc can now be explained in more detail. First, when yacc is invoked the `-d` option is always activated in order to generate both of the files `y.tab.h` and `y.tab.c`. The original input grammar is then rescanned to build internal representations of the productions and the non-terminal names (the file `y.tab.h` already contains the token names). After this, the source file `y.tab.c` is scanned, and several alterations are made to the C code in it. First, two new procedures are inserted: `yytname()` and `yytrname()`, to print terminal and non-terminal names respect-

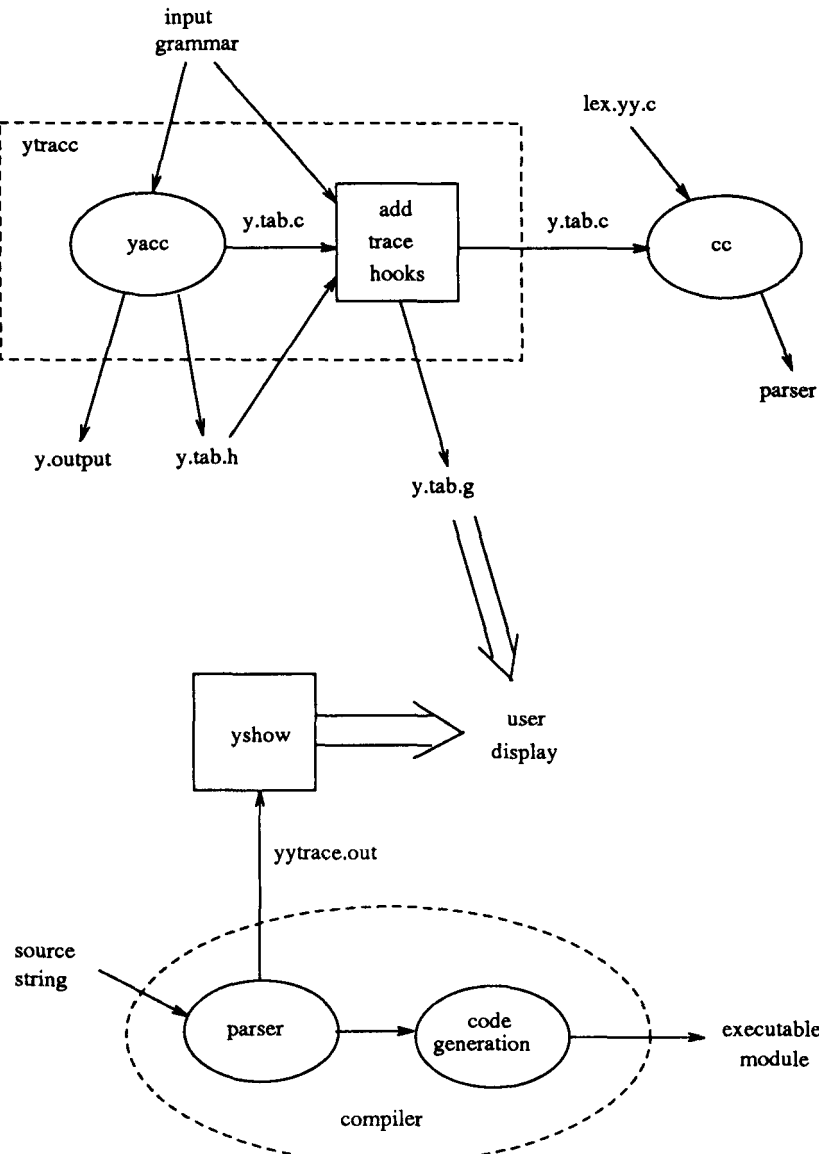


Figure 8. Block diagram of the ytracc system architecture

ively. These procedures are generated from the information in `y.tab.h` and the internal representation of non-terminal names. In addition, an instrumented lexical analysis routine called `ytrlex()` is inserted (which calls the expected routine `yylex()` and all other calls to `yylex()` in the code are renamed to invoke `ytrlex()`). Finally, the yacc-built routine `yyparse()` is renamed `ytrparse()` and a new version of `yyparse()` is inserted that opens the trace file `yytrace.out` and then calls `ytrparse()`. In the body of the yacc-built parsing routine, an addition is made after every parse action to record the event in the trace file. The modified `y.tab.c` file is then ready for compilation.

Several aspects of `ytracc` make it sensitive to changes in the standard Unix `yacc` program, and to a lesser extent, `lex` as well.* The modifications that `ytracc` makes to the output of `yacc` are obviously dependent on the format of the C files `y.tab.c` and `y.tab.h`, so future releases of `yacc` could cause `ytracc` to generate syntactically or semantically invalid code. `ytracc` will report an error, however, if it encounters a `y.tab.c` format that differs from what it expects. The parser produced by `ytracc` also requires three symbols to be defined by the lexical analyser. These symbols are `char yytext[]`, containing the lexeme for the current token, `int yyleng`, containing the length of the lexeme, and `int yylinen()`, holding the current line number of the file containing the source string. The values of these symbols need not be maintained; as long as the symbols are defined and contain initialized values, the parser will operate correctly. However, the performance of `yshow` will reflect the level of maintenance. An advantage of correct maintenance of these symbols is that `yshow` will illustrate the accuracy of the lexical analyser in addition to the accuracy of the parser. This is because `yshow` prints for each source line the tokens derived from it. Note that lexical analysers generated by `lex` automatically define and use all three of these symbols.

To summarize, `ytracc` modifies `yacc` products in several ways:

- (a) text elements of the grammar are saved for presentation during parsing visualization
- (b) phantom productions (created by `yacc` for triggering hooks in the middle of explicit productions) are named and made visible during visualization
- (c) the parsing routine is rewritten to create a history file and to record in it every action the parser makes while parsing a particular string
- (d) the lexical analysis routine is rewritten to store the program source as lexemes while parsing proceeds, so it can be regenerated line-by-line as a parse trace is browsed.

3. EXPERIENCES WITH `ytracc`

We made `ytracc` and `yshow` available for voluntary use in two consecutive semester-long undergraduate compiler classes in the Department of Computer Science at the University of Maryland. The students in this class are seniors or first year graduate students, and are relative accomplished programmers, although they have not had prior exposure to compiler construction tools. Each student was given information describing `ytracc` and `yshow` and encouraged to use them instead of, or in addition to, `yacc` in the construction of a compiler for a small but complete C-like language (for instance, the language contains recursive functions). Forty-eight students chose to use one or both of the programs; about half of the combined class sizes. We instrumented the versions of `ytracc` and `yshow` used by the students to capture information about the use of the system, and we had all students respond to a questionnaire at the end of the semester. As might be expected, the captured data

* `lex`⁴ is the Unix system's lexical analyser generator.

and the survey showed a great variation in responses, but our analysis suggests some general trends, which we will describe here.

Figure 9 summarizes the use of the two tools by each of the participants. Four pieces of information are shown for each participant: (1) the number of times they invoked `ytracc`; (2) the number of those invocations that completed successfully (i.e. in which the `yacc` specification was free of syntax errors); (3) the number of times the resulting parser was executed; and (4) the number of times that `yshow` was invoked. The Figure is sorted on the number of `ytracc` invocations.

The total number of `ytracc` invocations represented by the Figure is 3756. The Figure also represents 2994 successful completions, 5336 parser runs, and 403 invocations of `yshow`. Reassuringly, the collected information indicates that the number of error-free `ytracc` runs is less than the number of `ytracc` invocations, and the two numbers roughly parallel each other. Parsers tended to be run one or more times, probably on different test files.

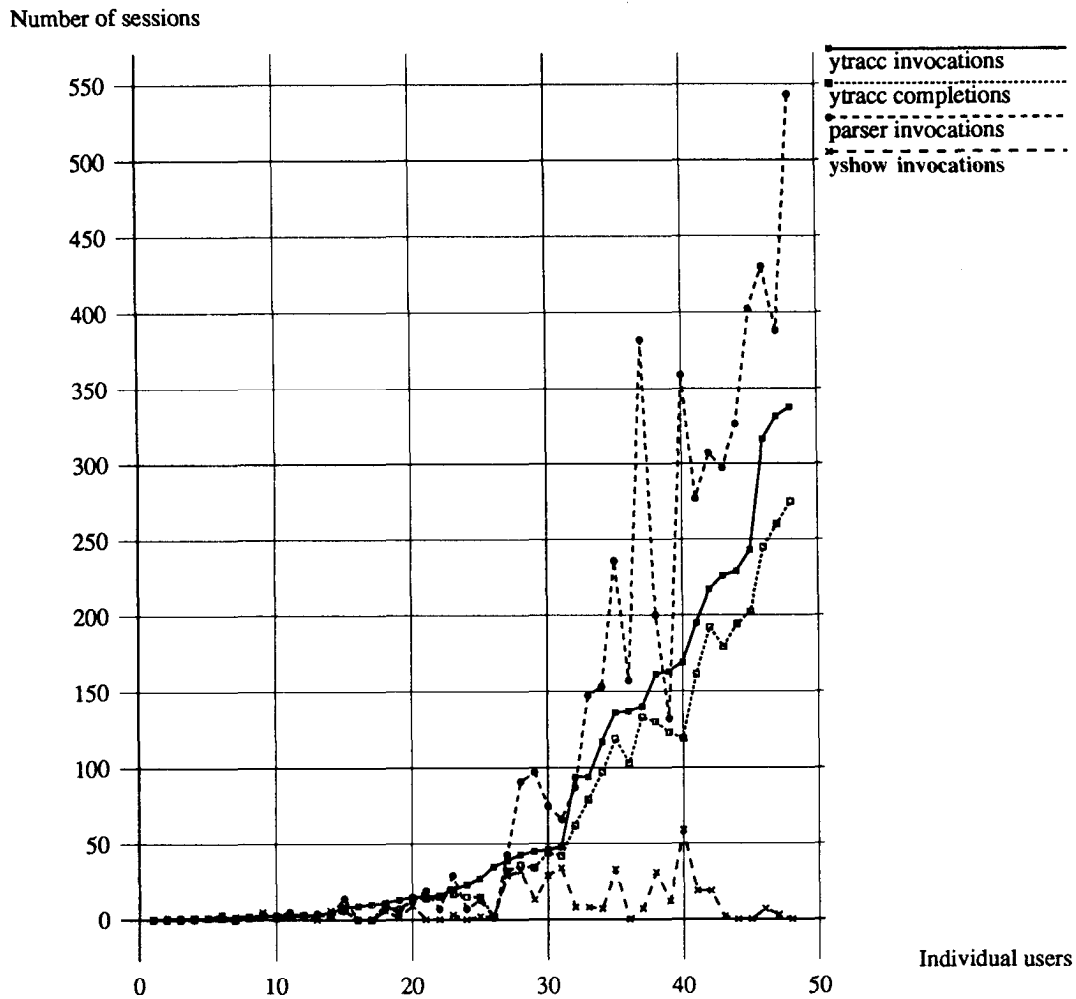


Figure 9. User program invocations (see text for explanation)

Analysis of the survey, supported by the captured data, suggests that the participants can be divided into several general groups, distinguished by the level of use of each of the two tools. We saw a large group of respondents who reported little or no use of either tool. As noted, about half of the students did not try `ytracc` or `yshow` at all, deciding to use `yacc` alone. Additionally, the left-hand tail of the graph in the Figure shows a number of participants who tried one or both of the tools a couple of times but chose not to use them intensively. In the survey, almost all of these students indicated an unwillingness to participate in something that might involve extra time over that already required by the course material. This was usually phrased as not having time to 'learn something new'.

A small group of respondents indicated no use of the tools because they already knew `yacc` and did not feel that they needed help in generating good grammars and parsers.

An equally small group of respondents reported using `yacc` most of the time, resorting to `ytracc` and `yshow` use only when parsing problems could not be easily deduced from inspection of the grammar (in the Figure, these are the participants for whom the number of `ytracc` completions is close to the number of `yshow` invocations). This behaviour was probably due to the fact that the students were working with limited CPU and disk resources. The execution time of `ytracc` is slightly longer than for `yacc` (it effectively scans the input grammar twice), and `ytracc` parsers run slower than `yacc` parsers (they do output to a file on each parse action); the trace files also can consume large amounts of disk space. These differences are exaggerated in a multi-user, minicomputer environment, and we suspect that the respondents in this group did not want to incur the extra overhead.

A few respondents reported using `ytracc` alone, without `yshow`. They indicated that several of `ytracc`'s features, not provided by `yacc`, gave some help in grammar construction. These include its more thorough syntax checking and error reporting for the input grammar file (such as finding missing semicolons), and the grammar summary (`y.tab.g`) that it extracts from the input. As can be seen in the Figure, many of these participants appear to have completely replaced use of `yacc` with `ytracc` and the number of `ytracc` invocations is considerable.

Finally, we saw a moderately-sized group that reported using the tools substantially. These respondents indicated that both components were very helpful in constructing their parsers. Respondents in this group also replaced all use of `yacc` with `ytracc` so that parsing trace data was available whenever needed. They indicated a very high level of satisfaction with the tools, and most reported that the parsing visualization provided by `yshow` was instrumental in locating grammar faults. One respondent described the help given by the `ytracc` system as being equivalent to the advantage that the Sun-based `dbxtool` program debugger provides over `in-line printf()` statements in source-code debugging. No one who reported non-trivial use of the system felt that they had been unaided by it, and all respondents recommended continued use of the tools in teaching parser construction.

Let us now focus briefly on the use of `yshow` commands. An average `yshow` session was just under 40 commands long and lasted 168 seconds. However, there was wide variability in session lengths—the number of commands executed ranged from a single command to a 347-command session. The elapsed time of sessions ranged from 0 to one that was 25 minutes and 36 seconds long.

Table II shows the distribution of yshow command use in the 403 recorded sessions.* (The yshow commands were previously enumerated in Table I.) The value reported for 'redraw display' is artificially high, as an instance of this command is automatically generated at the beginning of a session and as a bug, since corrected, existed for a time in yshow causing its display to be improperly updated on some network-based virtual terminal implementations. We also believe that our implementation's prompts for commands that permitted an optional argument may have given some students the expectation that those arguments were required, hence the relatively low use of optionless versions of those commands. Several respondents did report these effects in their surveys, noting that prompt messages were somewhat incomplete, in that they did not adequately convey default conditions for commands that interactively request arguments.

Taking these factors into consideration, it is immediately apparent that the default 'forward one action' command was by far the most commonly used and that the 'backward one action' was a distant second. The implication is that the students preferred to use yshow primarily as a means of browsing through the display of successive parser states, perhaps reflecting the fact that they were trying to develop an internal model of how yacc-based parsing proceeds at the same time that they were generating their grammars.

4. DISCUSSION AND CONCLUSIONS

The student participants in this study were developing a single, medium-sized grammar (the average grammar size encountered by successful ytracc runs was about 82 productions long; the maximum was 125 productions). In addition, they were all developing grammars for the same programming language. Although no detailed studies have been done of more varied or extensive applications of ytracc, our own use of these tools has shown them to be helpful in developing a wide range of parsers (including simple ones like that used in compiling the instrumentation data gathered during this project).

Our survey of participants revealed several 'missing' features of ytracc and yshow. One respondent expressed a need for a yshow command to advance the parse to a particular source line (yshow currently contains commands to advance by token-shift, production-reduction and action number). It was also noted that the on-line help for commands were terse and could be more useful if expanded and given some detail. This certainly was a factor in the earlier-discussed confusion over command default values.

In considering the earlier table of yshow command use (Table II), it is interesting to note the relative lack of use of the parser-action-based 'find' commands, and the general preference for the 'forward' version of commands rather than the 'reverse' version. Unlike the students, we have noticed that our own use of the tool tends to rely heavily on these commands, in particular on the error-locating commands. We speculate that as a compiler writer becomes more comfortable with the tools, yshow will primarily be used for quickly determining the state of the parse when unexpected errors occur. Such use will focus on quickly jumping to the area of interest in the

* The values do not sum to 100 per cent because of roundoff error.

Table II. yshow command use

Command class	Command character	Percentage of all commands		comments
		total	subtotal	
Forward one action		77.56		
	Carriage return		77.20	
	Newline		0.36	
Redraw display		7.70		
	^R		7.48	
	^L		0.22	
Backward one action		6.76		
Non-command characters		2.54		(User errors)
Quit		2.31		
	q		2.29	
	Q		0.02	
Go to absolute action		1.57		
			1.26	With optional argument
			0.31	Without optional argument
Forward n actions	m	0.51		
			0.47	With optional argument
			0.04	Without optional argument
Backward n actions	-	0.40		
			0.30	With optional argument
			0.10	Without optional argument
Repeat find command		0.19		
	n		0.17	Forward
	N		0.02	Reverse
Find next reduce		0.18		
	r		0.09	Forward with argument
			0.05	Forward without argument
	R		0.04	Reverse with/without argument
Find next shift		0.17		
	s		0.12	Forward with argument
			0.04	Forward without argument
	S		0.01	Reverse with/without argument
Find next error		0.09		
	e		0.08	Forward
	E		0.01	Reverse

parse with commands such as ‘find next error’, ‘find next reduce by non-terminal’, or ‘find next shift of symbol’, followed by finer-grained forward and reverse examination of the state of the parse.

It is interesting to note that although the student participants did not seem to use the more powerful features of yshow, the survey shows that they were equally split in describing the system as a ‘debugger only’ and ‘both a debugger and a learning

tool'. Our original goal in building the tool was primarily to provide a learning tool—an aid to teaching and understanding the process of bottom-up, deterministic parsing. The survey results suggest that the tool also is of use as a debugger, and that this usefulness continues on even after the basics of the parsing process are understood.

Much of the previous work in developing software environments for showing a visual representation of the inner workings of a program (see Reference 5 for example) has concentrated on the 'novice' student—the student at the freshman or sophomore level who is beginning to take computer science classes. Courses for such students teach the basics of computer science theory and of programming. The programs that these students write are small—perhaps 100 to 500 lines of code—and the algorithms are not very intricate. Tools designed for the use of such novices reflect the need to compensate for the relative inexperience of the programmers, and become unwieldy and overly restrictive as the student gains experience.

ytracc was not designed to be used by novice programmers, or indeed to be used in general programming, but rather to be used by a specialized group performing a specific task, namely compiler writers designing language grammars for parsing. By targeting a specific group with the tool, we feel that the displayed information can be made more detailed and presented in a more useful manner because of context-specific knowledge in the tools themselves.

It is this tailoring of the displayed information to the task domain that is the ytracc system's small contribution to programming. It is trivially easy to obtain voluminous printouts showing the states passed through by a yacc -based parser; the programmer can insert printf() statements in actions or more directly simply set the yacc variable yydebug to true. Locating the desired part of the transcript and determining the state of the parser once the appropriate segment has been found is not as easy. Easing such searches and presenting the information necessary for easy interpretation is the value of a tool such as yshow to a compiler writer; and indeed this is the point for development of such domain-specific visual tools.

ACKNOWLEDGEMENTS

This work was supported in part by FULCRUM grant #M300 from IBM to the University of Maryland as part of a larger project called the *Visible Tool Shop*.⁶

REFERENCES

1. Stephen C. Johnson, 'Yacc: yet another compiler compiler', *Computing Science Technical Report 32*, Bell Laboratories, Murray Hill, NJ, July 1975.
2. Alfred V. Aho, Ravi Sethi and Jeffrey D. Unman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.
3. Kenneth C. R. C. Arnold, 'Screen updating and cursor movement optimization: a library package', in *UNIX Programmer's Manual, Supplementary Documents*, University of California, Berkeley, March 1984.
4. M. E. Lesk, 'Lex—a lexical analyzer generator', *Computing Science Technical Report 39*, Bell Laboratories, Murray Hill, NJ, October 1975.
5. Tim Teitelbaum and Thomas Reps, 'The Cornell program synthesizer: a syntax-directed programming environment', *Communication of the ACM*, 563–573 (1981).
6. P. David Stotts, Richard Furuta and Jefferson Ogata, 'The visible tool shop: increasing programmer productivity through visual displays', in *Productivity: Progress, Prospects, and Payoff, Twenty-Seventh Annual Technical Symposium of the Washington, D. C. Chapter of the Association for Computing Machinery*, ACM Washington, D.C. Chapter, June 1988, pp. 77-83.