Tuning UNIX Lex
or
It's NOT True What They Say About Lex

Van Jacobson

Real Time Systems Group, Lawrence Berkeley Laboratory,
University of California, Berkeley, CA 94720
van@lbl-rtsg.ARPA

ABSTRACT

Unix Lex performance has been getting dismal reviews lately. At the Atlanta USENIX, Peter Honeyman [1] noted that Pathalias sped up a factor of two when its Lex scanner was replaced by a hand-coded scanner. Several USENET authors have written of similar experiences and the phrase "it's true what they say about Lex" has become rooted in Unix folklore. This folklore is now spreading to the Formal Language community: In recent articles on scanning, W. M. Waite [2] and V. P. Heuring [3] claim that poor performance is inherent in table-driven scanners and alternative scanner generators should be developed.

The Real Time Systems Group at LBL has used table-driven finite automata to construct some of the world's fastest data acquisition and control systems. We had doubts about the "inherently slow" claim and were curious why Lex went so slow. On investigation, we found a number of implementation problems but nothing that looked fundamental. To demonstrate there were no inherent problems, we attempted to write a "tuned" version of Lex. About one week's effort was invested in recoding the Lex runtime recognizer loop and making small changes in the table output routine. The result was a factor of ten performance improvement. When tested against the hand-coded scanners described in [1] and [2], the new Lex was always faster. There was also a serendipitous result: a new table compression scheme, intended to speed up the recognizer inner loop, resulted in a factor of two table size reduction. The new Lex scanners were not only faster than their hand-coded counterparts, both the original source and final machine code were smaller.

This paper describes the tuning work. In the next section we introduce a fragment of C that will be used as an ongoing example. We give an ad-hoc scanner for this fragment and two forms of table driven scanner, one using full tables and the other using S. C. Johnson's elegant compression algorithm [4,5] (this style of compressed table is used by both Lex and Yacc).

The performance of these scanners is compared in section three. The time to execute each scanner was computed using Knuth's time honored method of counting MIX instructions. Since instruction counting gives only execution time per type of token, it was also necessary to know the token type and length distributions for typical inputs. We give these distributions and show how they were produced by running small Lex and Awk programs over C source files.

In section four we investigate why the original Lex is slow compared to the "ideal" table driven scanner of section one. Extensive table compression accounts for about a third of the slow down. While compression was necessary in the original PDP-11 Lex, changing memory prices and cpu architecture have made it more costly than beneficial today. The remainder of the slow down is due to the cumulative cost of several infrequently used Lex features (e.g., YYREJECT, trailing context). We describe the impact of each of these features on the recognizer loop.

Section five describes the tables and runtime of the new Lex and the changes made to the original Lex source. We also describe how adding a little intelligence to the Lex compiler saves the user from paying the cost of unused features: While compiling, the new Lex keeps track of what features are used by the current input file and produces a scanner loop tailored for only those features. Finally we discuss the breakdown of the tuning work: In particular, why it took only three hours to modify the 600 lines of code involved in Lex table output but five days to write the six line subroutine that replaced Lex's 200 line recognizer.

[1] Peter Honeyman, 'PATHALIAS or The Care and Feeding of Relative Addresses', *The 1986 Summer USENIX Proceedings*, June, 1986, p.126.

[2] W.M. Waite, 'The Cost of Lexical Analysis', *Software--Practice and Experience*, 16(5), May, 1986, p.473.

[3] V.P. Heuring, 'The Automatic Generation of Fast Lexical Analysers', *Software--Practice and Experience*, 16(9), September, 1986, p.801.

[4] A.V. Aho and J.D. Ullman, *Principles of Compiler Design*, Addison-Wesley, 1977, Sect. 3.8.

[5] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers, Principles, Techniques and Tools*, Addison-Wesley, 1986, Sect.3.9.