Myths and Facts about the Efficient Implementation of Finite Automata and Lexical Analysis

Klaus Brouwer, Wolfgang Gellerich and Erhard Ploedereder

Department of Computer Science
University of Stuttgart
D-70565 Stuttgart
Germany
telephone: +49 / 711 / 7816 213; fax: +49 / 711 / 7816 380
gellerich@informatik.uni-stuttgart.de

keywords: Scanner, Lexical Analysis, Finite Automata, Run-time Efficiency

Abstract. Finite automata and their application in lexical analysis play an important role in many parts of computer science and particularly in compiler constructions. We measured 12 scanners using different implementation strategies and found that the execution time differed by a factor of 74. Our analysis of the algorithms as well as run-time statistics on cache misses and instruction frequency reveals substantive differences in code locality and certain kinds of overhead typical for specific implementation strategies. Some of the traditional statements on writing "fast" scanners could not be confirmed. Finally, we suggest an improved scanner generator.

1 Introduction and background

Finite automata (FA) and regular languages in general are well understood and have found many applications in computer science, e.g., lexical analysis in compilers [23,3]. A number of different implementation strategies were developed [3,24] and there are many scanner generators translating regular specifications (RS) which consist of regular expressions (RE) with additional semantic actions into executable recognizers. Examples are Lex [3,25], flex [25,28], Alex [27], Aflex [31], REX [18,19], RE2C [5], and LexAgen [32]. There seem, however, to be only a few studies comparing the efficiency of different implementation strategies on today's computer architectures.

Our original interest in this problem came from research in the field of programming language design for parallel architectures. Obviously, the assumption of a single flow of control is not the most natural approach when programming parallel machines. This is, however, the only possible model behind the infamous `GOTO` statement. An examination of `GOTO` usage in Ada, Fortran, and C [13] showed that `GOTO` is used rarely today, but one final stronghold of `GOTO`s are programs produced by generators. We were interested to see whether `GOTO` is

really needed to efficiently encode FAs or more structured implementations are equally or possibly even more efficient.

This paper presents the result of a run-time comparison of lexical analyzers produced by two scanner generators and four hand-coded scanners for lexical analysis of the programming language Ada 95 [1,4]. All scanners were written in the same language, Ada, and compiled using the same compiler, GNAT [14]. We also examined the impact of compiler optimization levels. As GNAT and the widely used GNU C compiler gcc share the same code generator, similar run-time data can be expected for scanners written in C. Execution behavior was evaluated on a Sun workstation using a RISC processor, on a PC with an Intel i486 CISC processor under OS/2, and on a PC with an Intel Pentium processor running Windows NT.

Section 2 provides further details on how the measurements were obtained. Section 3 describes the evaluated implementation approaches. Section 4 presents the execution statistics and discusses probable causes for run-time differences. Somewhat surprisingly, some well-known statements about scanner optimization could not be confirmed. In section 5 we compare our results to data gathered by other authors. Section 6 summarizes our conclusions and outlines our design of an improved scanner generator.

2 Evaluation strategy and setup

Lexical analysis in compilers is probably the most important application of FAs and most scanner generators are designed to utilize FAs. We therefore decided to compare *scanners* instead of pure FAs. This has the additional advantage that the interaction of the analysis of REs with other parts of a program is included in the measurements.

Typical tasks of a scanner are keyword recognition, conversion of numeric literals into internal representation, recognizing operator symbols, and maintaining a table for strings and identifiers. The implementation strategies differ significantly in how these further tasks are integrated into the pure FA realization. We used common packages for identifier handling etc. in all scanners to avoid biasing the run-time data by these peripheral activities. We were careful to ensure that all scanners had the same functionality.

2.1 The test problem

As problem to be solved and measured, we chose lexical analysis of the Ada 95 programming language. This can be considered a non-trivial problem, addressing most features occurring in lexical analysis.

Keywords and identifiers in Ada are not case-sensitive, so the scanner must map these lexical elements to either upper- or lower-case form. None of our scanners supports Ada's pre-defined type `Wide_Character` whose values correspond to the 16-bit representation of ISO 10646, because the scanner generators we intended to test were not able to handle such characters. All scanners do, however, support the Latin-1 8-bit character set.

Numeric literals can occur in several different notations which do not impose scanning problems, but complicate the conversion into an internal representation.

Correct handling of the apostrophe is somewhat difficult in Ada as this symbol can be a token of its own, and is also used to delimit character literals. For example, in `t'('a','b')`, `t` is a `type_mark` qualifying a character aggregate consisting of two literals `'a'` and `'b'`. Since an apostrophe token can only appear after the keyword `all`, closing parentheses, or an identifier, possible implementations are setting a flag, storing the previously recognized token, or switching into a special recognition mode after one of the three tokens is recognized. The need to consider limited context information in lexical analysis contradicts the principles of REs but is necessary in many programming languages [17,25] and can be handled by most scanner generators.

2.2 Hardware, run-time, and instruction measurement

Run-time measurements were made on a Sun SPARCstation 10 RISC workstation with Solaris 2.5 and OpenWindows 3.5, on a PC with an Intel 486 CPU, running IBM OS/2 version 3, and on a PC with an Intel Pentium, running Windows NT 4.0.

Initially, we measured consumed CPU time under Solaris using the `times`(2) function. This, however, turned out to be rather unprecise. Therefore, we used the workstation in real-time mode and started the scanners as processes of highest priority with a potentially infinite time slice, thus suppressing multitasking [22,26]. Time measurements were obtained via the function `gethrtime`(3C) reading the internal hardware timer. The relative error of the measurements is about 0.2%. On the PCs, run-time was measured using the function `times` provided by the `emx` library [14], offering a precision of only about 5%.

On the Sun, there was no way to measure the CPU's internal cache hit rate and to gather statistics about instruction frequency. However, the Intel Pentium CPU provides special instructions for this purpose which can can be accessed under Windows NT using a special driver [29,38,21].

2.3 Compilation and code optimization

All scanners were compiled using the same compiler, the GNU Ada 95 Translator GNAT [14]. GNAT uses the back-end of the GNU C compiler gcc [12] which provides four levels of optimization. Level `O0` does no optimization at all. Level `O1` enables a few code optimizations including improved register allocation, and in particular performs RISC-specific improvements like filling delay slots. Level `O2` enables many common code optimizations such as strength reduction, constant optimizations, and common subexpression elimination, while `O3` additionally performs heuristic inline expansion of subprogram calls.

2.4 Input to the scanners

Run-time tests on the SUN workstation applied the scanners to large input files created by simply concatenating existing Ada source code. We used 14 files

Abbreviation	Remarks on scanner implementation
AFLEX	with table compression
REXSK	Keyword recognition via REs, with table compression
REXS	Keyword recognition via REs, no table compression
REXK	Keyword recognition via hash table, table compression turned on
REX	Keyword recognition via hash table, table compression turned off
GNAT01	Keyword recognition using many `GOTO`s (from GNAT 3.01)
GNAT03	Keyword recognition via hash table (from GNAT 3.03)
WAITPI	with packing, with inlining
WAITI	without packing, with inlining
WAIT	without packing, without inlining
OPT1	Number conversion by cascaded `IF`s
OPT2	Number conversion via array

Fig. 1. List of examined scanners

ranging from 1.5 MB to 11.4 MB in size and consisting of over 58.1 MB in total. The run-time data given in this paper is the total time a scanner needed to scan all 14 files. On the PCs we only tested the 11.4 MB source file.

3 Examined scanners

Basically, there are two different strategies to implement a FA:

Table-driven scanners encode state transitions in one or more tables which control execution of a FA-algorithm which itself is independent of the RS to be analyzed [34,3]. We tested two scanner generators, Aflex and REX, both generating table driven scanners.

Directly encoded scanners consist of code reflecting the structure of the RS. An overview of methods for directly programming a FA is given in [24]. We tested two hand-written scanners carved from different versions of GNAT and wrote a scanner according to a strategy suggested by W.M. Waite [35]. Based on our experience gained during the project, we finally wrote a hand-optimized scanner intended to optimally solve the problem. Figure 1 gives an overview of all scanners tested. The following subsections describe them briefly.

Common packages providing number conversion, a hash table for identifier handling, and storing string constants were used in all scanners. Details are left out due to space limitations.

3.1 Table-driven scanners

Aflex. Aflex [31] is the Ada version of flex [25,28], a lex-compatible scanner generator which, according to [28], works as described in [3] by first constructing a non-deterministic FA which is then converted to a deterministic FA (DFA). By default, table compression is turned on. As turning it off caused program failure, we did not test that case. The original version only supported 7-bit characters and was extended to handle 8-bit characters.

Aflex provides an option to automatically map input characters to upper case when matching against REs. We used this to implement case-insensitive keyword and identifier recognition. The apostrophe problem is solved by entering a start state for special handling.

REX. The scanner generator REX [17–19] is part of a set of compiler generation tools [16]. Originally implemented in Modula-2, we had ported REX to Ada for a different project [30] and used this version for our measurements. REX handles 8-bit characters. The generated scanner uses so-called "tunnel automata" which are based on a pattern matching algorithm described in [2] and extend the DFA by a "tunnel function" [18]. In short, the automaton begins matching input using one of several alternative paths. When the match fails, the tunnel function causes a transition to a different state that represents a different continuation of the prefix read so far, thus implementing overlapping REs. Table compression is optionally available and applies the "comb-vector" row displacement technique [3]. We measured REX-generated scanners with and without table compression.

Matching input characters is always case-sensitive. We tested two versions: the first one only has a rule to match general character strings, with identifiers and keywords handled via a hash table. A second version has REs for all keywords completely written in upper or lower case, respectively. According to style guides and tradition, one will almost never see mixed case keyword spelling in Ada. The apostrophe problem is solved by entering a start state.

3.2 Directly coded scanners

GNAT. We measured two different scanners carved from GNU Ada Compiler GNAT version 3.01 (GNAT01) and version 3.03 (GNAT03). The earlier version recognizes keywords by a hand-coded algorithm using a total of 282 `GOTO`s. The typical structure of the source code looks like

```
when 'c' => Lower_Case_C : begin     --  CASE
  if Source (Scan_Ptr + 1) = 'a' then
    if Source (Scan_Ptr + 2) = 's' then
      if Source (Scan_Ptr + 3) = 'e' then
        Token := Tok_Case;
        goto Scan_Keyword_4;
      else goto Scan_Identifier_3; end if;
    else goto Scan_Identifier_2; end if;
  elsif Source (Scan_Ptr + 1) = 'o' then    --  CONSTANT
```

The program logic is based on the assumption that keywords are usually written in lower-case letters. Keywords with upper-case letters and all identifiers are recognized via a hash table. Version 3.03 no longer uses this approach and always recognizes keywords via a hash table, thus reducing code size [9]. The apostrophe problem is solved by storing the previously recognized token.

In order to get comparable results, we wanted all scanners to have nearly identical functionality. In the case of the GNAT scanner, we removed code for GNAT's excellent error handling. We also omitted some code to implement an Ada 83 mode, wide character support, style checking, and the generation of syntax tree nodes which GNAT does, in some cases, in its scanner.

The Waite method. W.M. Waite suggested a technique to efficiently implement hand-coded scanners [35]. The set of all tokens is divided into a few classes like "identifiers", "numbers" or "operators". Tokens from different classes must start with different characters. An array maps characters to the token class starting this way. This array is used in a `CASE` statement which then branches to code recognizing the particular token class.

Identifiers, string literals and numbers are recognized by loops. The decision whether the next input character belongs to the currently scanned token is implemented by arrays mapping characters to boolean values. The apostrophe problem is solved using a flag.

As the size of the array might conflict with cache management, we tested two versions of the scanner, one packing the array using `pragma(Pack)`, which, however, might cause expensive unpacking. We also tested a version making heavy use of `pragma(Inline)` for expanding subprogram calls.

Our hand-optimized scanner. Based on our experiences reported here, we designed an optimized hand-coded scanner. It works similar to the Waite method but avoids *double* arrays mapping: the first character of any token is used in a `CASE` statement to branch to code dealing with the rest of the token's characters. This, as well as the lack of any `GOTO` statements, can yield a high degree of locality in the code. Depending on the strategy chosen to implement `CASE` statements, this approach might introduce extra tests when the cases are not compact. We did not examine this in detail, because it is likely that the cases *are* compact in scanners for common programming languages.

We also were careful to never discard information needed later. For example, the main `CASE` statement already distinguishes between digits. In order not to loose that information, we provided individual branches for every digit, initialized a variable with that number and then continued recognizing the rest of that number using common code.

We tested two versions of number conversion. The first one tests characters using `IF` statements and then assigns the appropriate value while the second one maps characters to the appropriate value using an array. The apostrophe problem is solved using a flag.

4 Results and conclusions

Figure 2 shows the run-time of all scanners on the SUN. There are four entries given for each scanner, referring to optimization level `O0` to `O3`, respectively.

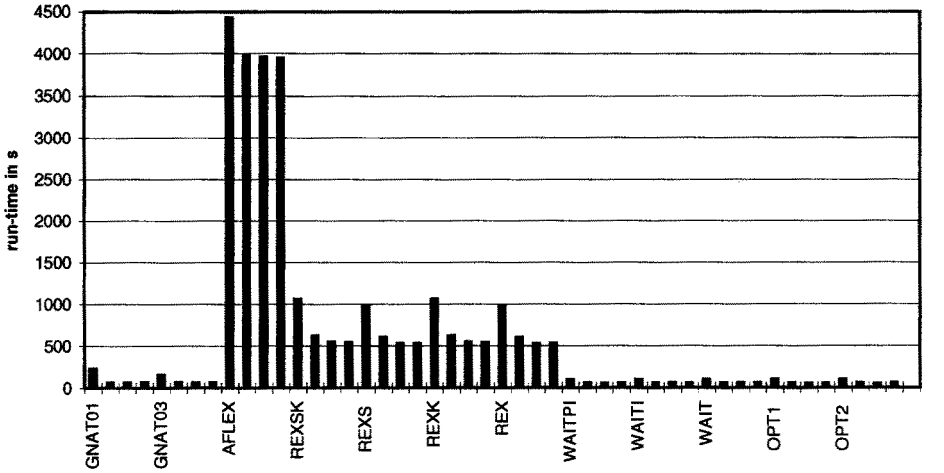

Fig. 2. Run-time of all scanners (workstation)

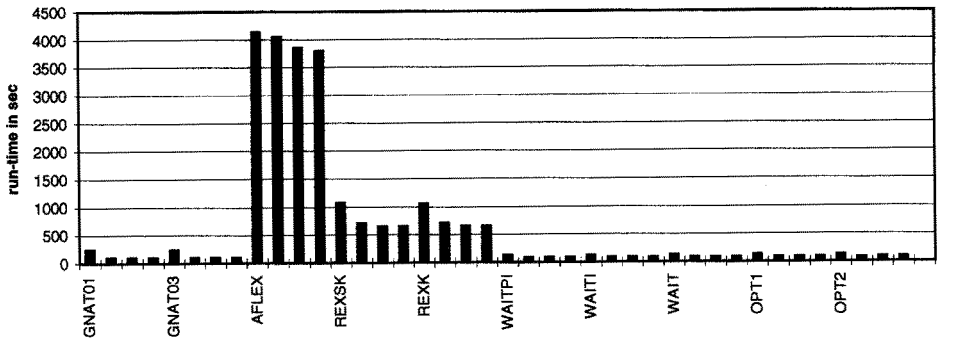

Fig. 3. Run-time of all scanners (PC)

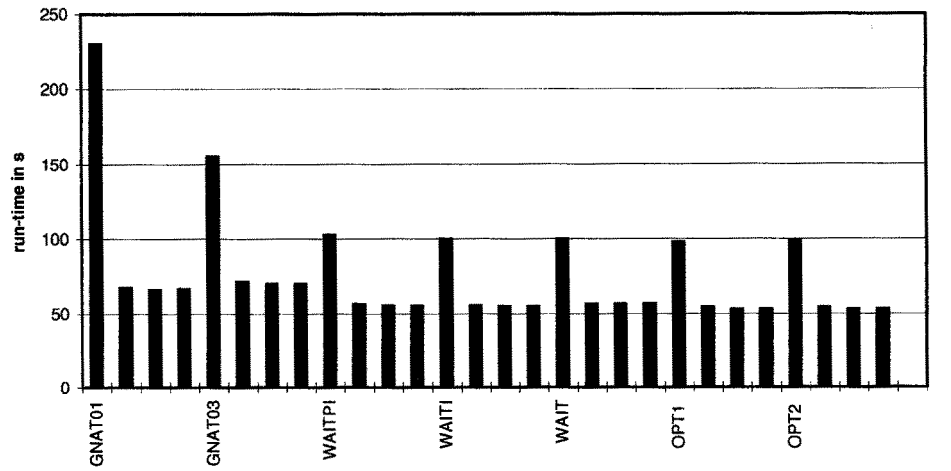

Fig. 4. Run-time of hand-written scanners only (workstation)

Scanner	Time/ sec	Data Read Inst/10^6	Data Cache Miss/10^6	Code Read Inst/10^6	Code Cache Miss/10^6	Locality Data	Locality Code	Locality Total
AFLEX	66.37	2188.95	87.33	2592.65	9.01	25.07	287.75	49.63
REXSK	51.17	1439.63	49.18	1411.20	8.67	29.27	162.77	49.28
REXS	46.83	1397.69	36.57	1372.29	9.00	38.22	152.48	60.79
REXK	50.48	1350.93	43.63	1440.17	9.17	30.96	157.05	52.86
REX	46.79	1321.45	30.74	1374.34	9.28	43.37	148.10	67.36
GNAT01	3.52	82.98	1.22	176.72	5.16	68.02	34.25	40.70
GNAT03	3.68	92.38	1.56	198.93	4.21	59.22	47.25	50.49
WAITPI	3.81	93.34	3.83	210.09	6.65	24.37	31.59	28.95
WAITI	3.76	92.68	3.66	199.31	3.87	25.32	51,50	38.78
WAIT	4.15	95.28	3.96	210.55	8.63	24.06	24.40	24.30
OPT1	3.37	88.04	1.85	204.61	2.11	47.59	96.97	73.76
OPT2	3.39	88.04	1.95	204.86	2.06	45.15	99.45	73.04

Fig. 5. Scanning 11 MB Ada sources (Pentium PC), min. and max. values underlined

All table-driven scanners were slow in general, with REX being about 7.5 times faster than Aflex when full optimization is enabled. OPT1 is the fastest directly encoded scanner and outperforms Aflex by a factor of about 74. Turning on the first level of optimization yields significant speedup while higher optimizations show little or no effect.

When repeated on a faster SUN, the difference was only a factor of 27.This can be explained by the fact that IO is relatively insensitive to CPU speed. There are fixed costs increasingly dominating total time when *scanning* becomes faster.

Figures 3 and 5 give the results measured on the PCs with 486 and Pentium CPU, respectively, which are, in general, quite similar to the workstation results. Figure 4 shows the run-time of directly encoded scanners in more detail. With optimization turned on, there is not much difference in execution time. Figure 5 shows run-time and the number of read instructions and cache misses executed by the scanners compiled with optimization `O2` while scanning 11 MB Ada sources on a Pentium PC. There is a quite obvious correlation between run-time and the number of read instructions and cache misses.

The rest of this section discusses possible reasons for the differences in performance. Some reasons are what we had expected, but we also found some surprising results. We tried to verify each hypothesis with differential measurements which was, however, not possible in all cases as the effect of certain implementation properties are very difficult to isolate.

4.1 The representation of states

In order to handle the next input character correctly, the scanning algorithm needs to know a "history" of characters processed before. FAs encode this information in the current *state*.

Our directly coded scanners represent the state *implicitly* by the current position of control flow. Another approach to represent states by control flow is

to associate a label with every FA state and then use GOTOs for state transition, a technique used in RE2C [5].

Other approaches *explicitly* store the state using a variable which is then used to index an array in the case of a table-driven scanner or to dispatch using a CASE statement [3]. Explicit encoding of state information is, however, an unnecessary level of indirection, causing run-time overhead – a difference similar to implementing a programming language using an interpreter or a compiler.

The indirection multiplies the number of instructions to be executed per input character. Figure 5 shows that those scanners using an indirect representation of state need to execute up to 26 times more data reads and fetch up to 14 times more instructions than those using implicit state representation. However, there are further effects increasing the number of instructions, and we could not isolate these: indirection always comes along with less locality, and the algorithms used in Aflex and REX have certain weaknesses, sometimes causing unnecessary operations to be performed.

4.2 Locality and cache usage

Most of today's computers have cache memory which is usually as fast as it is small. Program run-time therefore strongly depends on achieving a good cache hit rate and this is determined by the degree of referential locality. There is locality both of data and of program code, affecting data and instruction caches, resp. Locality in code means that flow of control stays in the same region of code for a relatively long time while, e.g., far-reaching jumps violate code locality.

We measured the data and instruction cache misses for all scanners on a Intel Pentium CPU. Both caches have a size of 8 kB and are organized as 32bit words with LRU replacement strategy [21]. Relative miss rates reflect the algorithm's working set size which can be considered a measure for locality.

In figure 5, *locality* is defined as the ratio of read instructions and cache misses, thus yielding the average number of read instructions between two cache misses. Total locality is the total number of read instructions divided by the total number of cache misses.

The fastest scanners of all, OPT1 and OPT2, are those constructed to have a high degree of both kinds of locality. They achieved good values for code and data locality and the best values for total locality. However, total locality varied only by a factor of about 3, and the table-driven scanners exhibited rather good values, too. Total locality can not be a major factor for speed differences.

Code and data locality show contrary behavior in general. GNAT01, encoding most information in its control structure and thereby using several hundred GOTOs and EXITs achieves a very low code locality but has the highest data locality as it uses almost no large data structures. On the other side, Aflex shows the best code locality of all scanners but has low data locality. Not surprisingly, all table-driven scanners yielded similar results, caused by the use of large tables which do not completely fit into the data cache, while the interpreting algorithm is rather small.

Token class	Frequency	Token class	Frequency
Single (non-letter) character tokens	41.8%	Integer literals	4.7%
Identifier	32.9%	String literals	0.94%
Keywords	14.1%	Character literals	0.37%
Two (non-letter) character tokens	4.9%	Real literals	0.1%

Fig. 6. Frequency of tokens in Ada

The WAIT scanner, although rather fast, got low values in locality. Possible causes might be five arrays frequently used for mapping characters, and many subprogram calls. Code locality is improved by a factor of two when subprogram calls are expanded, which also reduced run-time by a few percent. Packing arrays did not significantly improve cache behavior but decreased code locality, presumably caused by the extra code needed for unpacking. The weak cache behavior might also be explained by working set theory: perhaps WAIT would produce *much* less cache misses if the cache were only a *little* bit larger.

4.3 Algorithms used in REX and Aflex

To some degree, the inefficiency of the two table-driven scanners is caused by weaknesses of the algorithms used. For example, the Aflex-generated scanner does not use the input character directly but first maps it into an integer number.

For REX, we traced the generated scanner with a debugger and found that after accepting keyword `IF` and the following space character, the scanner performed another nine tunnel transitions before it finally recognized that token. This suggests that the information available from a language definition is not yet fully exploited: For our RS, spaces act as separators thus terminating any token and making any further tunneling unnecessary. Additional inefficiencies exist but are omitted here due to space limitations.

In the case of Aflex we found that some of its inefficiency is caused by the translation from C to Ada. For example, using subtypes for array ranges and index variables instead of integer ranges and integer variables will enable good compilers to omit many range checks.

4.4 Keyword and identifier recognition

Identifiers occur twice as often as keywords in Pascal which suggests that one should avoid spending additional work for the less probable case of character strings being keywords [35]. Figure 6 shows that Ada programs have a similar relation between keywords and identifiers.

For table-driven scanner generators, one suggested method [17,25,19] is implementing keyword recognition as part of the automaton which then skips to general identifier recognition only if input does not match against any keyword. The identifier is not yet stored in a name table by that process.

More efficient seems the "sifter" approach [37] where the automaton recognizes strings of characters. These are then passed to a name table which detects

	length of .s file in Byte	number of `loads`	number of `stores`	number of `nops`
GNAT01 `-O0`	666552	5263	2928	5029
GNAT01 `-O1`	217170	1358	328	905
GNAT03 `-O0`	175401	1629	991	1823
GNAT03 `-O1`	103467	447	206	297
OPT1 `-O0`	104792	689	454	1059
OPT1 `-O1`	106793	396	307	268

Fig. 7. Frequency for some machine instructions

keywords via hashing or a specialized search tree [32] and also does general identifier handling which needs to be done anyway.

Although these arguments sound quite convincing, a comparison of both methods with REX (see figure 1) revealed that run-times differ by less then 0.5%. Similar statements hold for the two GNAT scanners.

But, keywords and identifiers occur rather frequently. Weaknesses in their recognition will considerably slow down execution time. Although identifiers occur more frequently then keywords, it is the case that "important" keywords are used more often than any identifier, and in larger programs the number of different identifiers will considerably exceed the number of keywords. A hash table used to access the name table should therefore avoid collisions between keywords by choosing a *perfect* hash function [6,7,39,8,20].

4.5 Facts about `GOTO`

A comparison of all directly encoded scanners contradicts the contention that structured programming might be inefficient and the usage of `GOTO` is necessary to get fast code.

Another myth concerning `GOTO` is that unstructured code necessarily interferes with code optimization. Our results are different: enabling the first level of optimization yielded a significant speedup for GNAT01 and GNAT03. To explain this result, we took a closer look at the generated assembly code (see figure 7) and counted the total number of `load`, `store`, and `nop` instructions [36]. We ignored any files common to all scanners.

The large number of jumps in GNAT01 seems to split the code into many small basic blocks. With register optimization disabled, this results into a large number of `load` and `store` instructions and, as no instruction scheduling is done, many `nops` are needed to fill RISC-specific delay slots. Enabling first level optimization when compiling GNAT01 reduces the number of `loads` by 74.2%, the number of `stores` by 88.8% and the number of `nops` by 82%. These improvements are lower for GNAT03 and OPT1 which are more well-structured programs. Higher levels of optimization did not significantly improve run-time any further, but this was observed for *all* scanners.

4.6 Never touch characters twice

A key design goal for scanners should be to "minimize the number of times each character is touched by the program" [35]. Although this sounds quite obvious it is often violated when using a scanner generator. The problem occurs when REs do not describe a single lexical token but a token class. Consider how numbers are recognized: first, the input characters are compared against the corresponding RE and stored in a buffer usually provided by the scanner generator. This phase stops when reaching a terminator symbol. Then, the whole number is read again, this time from buffer, in order to be converted to internal representation.

Directly encoded scanners, however, can do matching and immediate conversion within the same loop. This eliminates the buffer and the second loop and also avoids re-scanning the particular character sequence. Scanner specification languages thus should allow semantic actions inside of REs instead of invoking them only after an expression is completely matched, basically by supporting an attribution syntax already used by many parser generators.

To check the actual effect, we modified OPT1 by inserting code to first store the digits in an array and then convert the number after the RE is fully recognized. This increased overall run-time by about 1.5%. Considering that real and integer literals make up only about 5% of all tokens in Ada (see figure 6), this difference would be significant for regular languages with a high frequency of numeric literals.

4.7 Input buffering

We ran experiments to find out whether execution time depends on the way input is buffered. We found that buffer size has only a very small influence on the scanner's execution time. The scanner generated by REX has a default buffer size of 8 kB[1]. Increasing the buffer size to 128 kB led to an acceleration of only 5%. This might be surprising at first glance, but can be explained by the fact that the operating system (and perhaps the run-time system, too) already buffers input from disk. In a different experiment, we measured the percentage of execution time of OPT1 that was attributable to IO. The result of about 10% suggests that the cost of IO is sometimes over-estimated.

5 Related Work

P. Bumbulis and D.D. Cowan [5] measured run-time performance of four different scanners for C on four machines. Figure 8 cites the results for SPARC and i486. The compiler was gcc with optimization `O1` enabled. The generators gla [15], and RE2C [5] produce directly encoded scanners which are faster than the table-driven Flex.

Lcc is a hand-written scanner from a public C compiler [10,11]. It was implemented following the Waite method and we were surprised to see that it did

[1] plus additional 256 Bytes which remain unused

Scanner Generator	Time/s on SPARC	Time/s on i486	Scanner Generator	Time/s on SPARC	Time/s on i486
flex -Cem	18.81	23.12	gla	5.64	6.29
flex -Cf	10.53	10.30	re2c	5.43	5.86
lcc	6.47	6.67	re2c -s	5.18	6.02

Fig. 8. Run-time evaluation of scanners for C [5]

Scanner Generator	Time/s (with hashing of identifiers and number conversion)	Time/s (without hashing of identifiers and number conversion)
lex	7.21	6.88
flex -Cem	3.99	3.69
flex -Cf	2.12	1.80
Rex -c	1.77	1.37

Fig. 9. Run-time evaluation of scanners for Modula-2 [18]

not yield really good results but then took a look at its source code. Lcc is likely not to achieve good referential locality as it uses many `GOTO`s for jumping between different branches of the outer `switch` statement. Including `break` and `continue`, lcc has a total of 33 jumps. An additional weakness might be keyword and identifier recognition. The code is similar to the mechanism used in GNAT 3.01 and is claimed to recognize individual keywords faster than even a perfect hash table could [10], but this is at the expense of handling identifiers which occur much more frequently. Also, lcc unnecessarily tests for keywords having a common prefix longer than one character.

J. Grosch [18] compared some generated scanners for lexical analysis of Modula-2. Figure 9 shows that the original lex generated the slowest scanners. The relative speed of Flex and REX differs less than in our test.

6 Summary and future work

We compared twelve differently implemented scanners and found that their execution time solving the same problem differs by more then a factor of 70 with table-driven scanners being slow in general. There is no single explanation for this difference. As a general result we can state that today's computer architectures require algorithms with a high degree of locality in both the executable code and the data. Locality of code means the absence of frequently executed and far-reaching transfers of control. Locality of data of course is particularly endangered by the existence large, frequently and randomly accessed tables. Another factor worth noting is that generated scanners suffer from executing unnecessary actions, using unnecessary indirection in the implementation of states, or doing the same thing more than once.

Some other causes which at first seemed quite plausible had only little effect in practice. Several claims sometimes made about implementation strategies for fast scanning could not be confirmed. Some might have been valid decades ago

but fail to consider the advances in code generation and hardware architectures since then. In particular, the effect of input buffering and inline-expansion is over-estimated and the use of GOTOs did not prove to be necessary for achieving good performance – structured programming yields better results.

Experimental evaluations are published rarely which has been considered a problem of computer science in general [33]. However, we found some data published in the context of newly developed scanner generators and these confirm our result about the speed of table-driven scanners.

Measuring the effect of different code optimizations leads to the conclusion that register allocation and, for RISC CPUs, instruction scheduling are critical issues but most other code optimizations have no or very little effect on a scanner's run-time.

It is important to note that the efficiency does not depend on whether scanners are hand-written or automatically generated. It is, however, the case that table-driven scanners are slow, and many scanner generators produce such scanners. The fastest scanners in our test were hand-written, but there is no reason why the construction principles we used should not be incorporated into a scanner generator.

Currently, we design a new scanner generator based on the principles we found: the input language will allow semantic actions to be written within REs instead of executing them after a complete RE is matched to avoid unnecessary buffering and re-scanning of, e.g., numbers. The generated scanning algorithms will be directly encoded and achieve good locality by avoiding jump statements as well as the use of large arrays.

References

1. *Ada 95 Reference Manual.* Intermetrics, Inc., 1995. ANSI/ISO/IEC-8652:1995.
2. A.V. Aho and M.J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM*, 18(6):333–340, June 1975.
3. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers.* Addison-Wesley, 1986.
4. J. Barnes. *Programming in Ada 95.* Addison Wesley, 1995.
5. P. Bumbulis and D.D. Cowan. RE2C: A More Versatile Scanner Generator. *ACM Letters on Programming Languages and Systems*, 2(1-4):70–84, 1993.
6. R.J. Cichelli. Minimal Perfect Hash Functions Made Simple. *Communications of the ACM*, 23:17–19, 1980.
7. C.R. Cook and R.R: Oldehoeft. A Letter Oriented Minimal Perfect Hashing Function. *ACM SIGPLAN Notices*, 17(9):18–27, 1982.
8. Z.J. Czech and G. Havas. An optimal algorithm for generating minimal perfect hash functions. *Information Processing Letters*, 43(5):257–264, October 1992.
9. R. Dewar. (private communication).
10. C.W. Fraser and D.R. Hanson. A Retargetable C Compiler. *ACM SIGPLAN Notices*, 26(10):29–43, October 1991.
11. C.W. Fraser and D.R. Hanson. *A Retargetable C Compiler.* Addison-Wesley, 1995.
12. Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA 02111-1307 USA. *Using and Porting GNU CC*, 1995. (for GCC Version 2.7.2).

13. W. Gellerich, M. Kosiol, and E. Ploedereder. Where does goto go to? In *Reliable Software Technologies – Ada-Europe 1996*, volume 1088 of *LNCS*, pages 385–395. Springer, 1996. (`http://www.informatik.uni-stuttgart.de/ifi/ps/Gellerich/adagotowww.ps`).
14. Gnu ada translator (gnat) documentation, 1995. (`ftp cs.nyu.edu:/pub/gnat`).
15. R.W. Gray, V. Heurig, S.P. Levi, A.M. Sloane, and W.M. Waite. Eli: A complete, flexible compiler construction system. *CACM*, 35(2):121–131, February 1992.
16. J. Grosch. Generators for High-Speed Front-Ends. In *Compiler-Compilers and High-Speed Compilation*, volume 371 of *LNCS*, pages 81–92. Springer, 1988.
17. J. Grosch. Selected Examples of Scanner Specifications. Technical Report 7, Gesellschaft fuer Mathematik und Datenverarbeitung mbH, 1988.
18. J. Grosch. Efficient Generation of Lexical Analysers. *Software Practice and Experience*, 19(11):1089–1103, November 1989.
19. J. Grosch. Rex – A Scanner Generator. Technical Report 5, Gesellschaft fuer Mathematik und Datenverarbeitung mbH, 1992.
20. G. Havas and B.S. Majewski. Graph Theoretic Obstacles to Perfect Hashing. *Congressus Numerantium*, 98:81–93, 1993.
21. Intel Corporation. *Pentium Processor Family Developer's Manual*, 1997.
22. SPARC International. *SPARC Architecture Manual, Vers. 8.* Prentice Hall, 1992.
23. W.L. Johnson, J.H. Porter, S.I. Ackley, and D.T. Ross. Automatic Generation of Efficient Lexical Processors Using Finite State Techniques. *ACM SIGPLAN Notices*, 11(8):805–813, December 1968.
24. D.W. Jones. How (Not) to Code a Finite State Machine. *ACM SIGPLAN Notices*, 23(8):19–22, 1988.
25. J.R. Levine, T. Mason, and D. Brown. *lex & yacc.* O'Reilly & Associates, Inc., Sebastopol, 2. edition, 1990.
26. SUN Microsystems. *Solaris 2.3 Software Developer Answerbook*, November 1993.
27. H. Moessenboeck. Alex – a simple and efficient scanner generator. *ACM SIGPLAN Notices*, 21(5):69–78, May 1986.
28. Vern Paxon. *Flex, Version 2.5.* University of California, Berkeley, March 1995.
29. U. Post. Gleitzeit – Performance Monitoring deckt Gleitkommanutzung auf. *c't*, pages 256–259, Sep 1997.
30. Ada version of REX. (`www.informatik.uni-stuttgart.de/ifi/ps/cocktail`).
31. J. Self. Aflex – An Ada Lexical Analyzer Generator. Technical Report UCI-90-18, University of California, Irvine, May 1990.
32. D. Szafron and R. Ng. LexAGen: An Interactive Incremental Scanner Generator. *Software – Practice and Experience*, 20(5):459–483, 1990.
33. W.F. Tichy, P. Lukowicz, Lutz Prechelt, and E.A. Heinz. Experimental Evaluation in Computer Science: A Quantitative Study. *Journal of Systems and Software*, 28(1):9–18, Januar 1995.
34. J.P. Tremblay and P.G. Sorenson. *The Theory and Praxis of Compiler Writing.* McGraw-Hill, 1985.
35. W.M. Waite. The Cost of Lexical Analysis. *Software – Practice and Experience*, 16(5):473–488, 1986.
36. D.L. Weaver and T. Germond. *SPARC Architecture Manual, Version 9.* Prentice Hall, 1994.
37. R. Wilhelm and D. Maurer. *Compiler Design.* Addison-WesleySpringer, 1995.
38. M. Withopf and A. Stiller. Durchgriff – Direkte Zugriffe unter Windows NT 4.0 und ein entfesselter Cyrix 6x86. *c't*, pages 312–315, Jan 1997.
39. D.A. Wolverton. A Perfect Hash Function for Ada Reserved Words. *ACM Ada Letters*, VI(1):40–44, July/August 1984.