RE2C – A More Versatile Scanner Generator

Peter Bumbulis Donald D. Cowan
Computer Science Department and Computer Systems Group
University of Waterloo*

April 15, 1994

Abstract

It is usually claimed that lexical analysis routines are still coded by hand, despite the widespread availability of scanner generators, for efficiency reasons. While efficiency is a consideration, there exist freely available scanner generators such as GLA [7] that can generate scanners that are faster than most hand-coded ones. However, most generated scanners are tailored for a particular environment, and retargetting these scanners to other environments, if possible, is usually complex enough to make a hand-coded scanner more appealing. In this paper we describe RE2C, a scanner generator that not only generates scanners which are faster (and usually smaller) than those produced by any other scanner generator known to the authors, including GLA, but also adapt easily to any environment.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications – *specialized application languages*; D.3.4 [**Programming Languages**]: Processors

General Terms: Algorithms, Languages, Performance

Additional Key Words and Phrases: Lexical analysis, scanner generator

1 Introduction

Lexical analysis routines are still often coded by hand despite the widespread availability of scanner generators. For example, while most Unix systems have a scanner generator installed (typically LEX [15] or flex [16]), few Unix applications use a mechanically generated scanner. One commonly cited reason for not using LEX-generated scanners is performance: they can be 10 times slower than equivalent hand-coded scanners [13]. As a result, there has been considerable research into improving the performance of mechanically generated scanners [16, 7, 9]. GLA [7], one such scanner generator, can produce scanners that are faster than most hand-coded scanners. However, the use of hand-coded scanners is still prevalent. One possibility is that this is due to the difficulty of adapting the generated scanners to specific applications.

Most scanner generators are tailored to a particular environment. In fact, the trend in recent years has been to integrate scanner generators with compiler toolkits. For example, GLA is part of the Eli compiler construction system [8], and Rex [9] is part of the GMD Toolbox for Compiler Construction[1]. Scanners

[1] Also known as Cocktail (Compiler-Compiler-Toolbox Karlsruhe).

generated by these tools assume the existence of a library of support modules for error handling, input buffering, symbol table management, and similar functions. While these support modules simplify the task of implementing a compiler or interpreter, they make adaptation to other purposes more difficult. Adaptation to other environments is also made more difficult because often assumptions are made about the input and restrictions are placed on tokens in order to achieve better performance. RE2C goes to the other extreme: it concentrates solely on generating code for matching regular expressions.

RE2C is successful at its task: not only does it produce scanners which are faster than those created by other scanner generators but, surprisingly, they are usually smaller as well. Further, RE2C does not make any assumptions about the input or place any restrictions on tokens. To a large degree, the performance and flexibility of RE2C-generated scanners is due to a novel method for determining when to refill a buffer which avoids the complications introduced by the sentinel method [1].

The following sections of this paper describe RE2C scanner specifications, discuss how these specifications are converted into scanners, and give performance results achieved by our implementation (including a comparison with GLA).

2 Scanner Specifications

An RE2C source file consists of C[14] or C++[4][2] code interleaved with comments of the form `/*!re2c ... */` containing scanner specifications. These specifications are replaced with generated code that is invoked simply by "falling into" the comments as illustrated in Figure 1 and in Appendix A[3].

```
#define YYCURSOR  p
unsigned char *scan_uint(unsigned char *p){
/*!re2c
    [0-9]+          {return p;}
    [\000-\377]     {return NULL;}
*/
}
```

Figure 1: A simple scanner.

A scanner specification takes the form of a list of rules, each rule consisting of a regular expression [10] and an action expressed in executable code. Figure 2 illustrates a trivial RE2C scanner specification that will be used as an example throughout this paper. Each call to the code generated from a specification will

```
"print"         { return PRINT;  /* rule 5 */ }
[a-z]+          { return ID;     /* rule 4 */ }
[0-9]+          { return DEC;    /* rule 3 */ }
"0x" [0-9a-f]+  { return HEX;    /* rule 2 */ }
[\000-\377]     { return ERR;    /* rule 1 */ }
```

Figure 2: Sample specification. [a-b] matches any character between a and b, inclusively. The last rule, for example, will match any eight bit character. Rules are listed in order of precedence.

first determine the longest possible prefix of the remaining input that matches one of the regular expressions and will then execute the action in the first applicable rule.

[2]Retargetting RE2C to a different language is straightforward.
[3]RE2C-generated scanners require no additional support code.

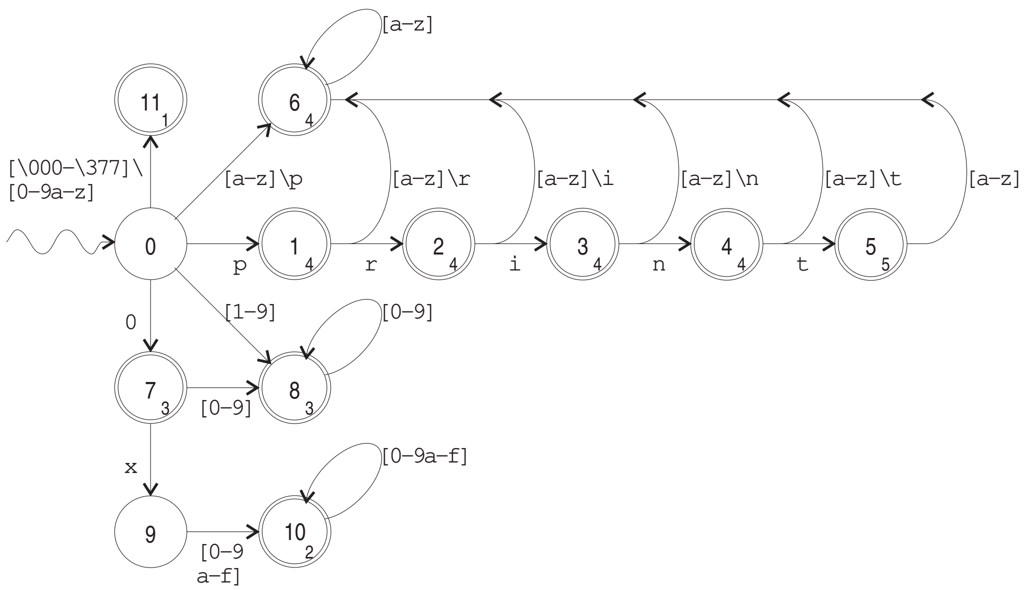

Figure 3: A DFA for the sample specification in Figure 2. State 0 is the start state. Accepting states are labeled with the number of the rule that they match. For example, state 10 accepts rule 2. Transitions differing only by label are represented with the same arc. For example, state 0 has transitions to state 6 on all of the following characters: `a`, ..., `o`, `q`, ... `z`.

RE2C is different from most other scanner generators in that the user must provide the input buffering mechanism for the scanner; the generated code simply assumes that the user has defined three pointers: `YYCURSOR`, `YYLIMIT` and `YYMARKER`, and a routine `YYFILL(`n`)`. Before executing the generated code, `YYCURSOR` and `YYLIMIT` must be set to point to the first and one past the last character in the buffer, respectively. After a token is recognized, and before any action is executed, `YYCURSOR` is set to point to just past the token. `YYFILL` will be called as the buffer needs filling; at least n additional input characters should be provided. When `YYFILL` is called, `YYCURSOR` will point to the next character to be scanned and `YYMARKER`, if set, will point to a possible backtracking point in the buffer. `YYFILL` must update `YYLIMIT`, and possibly `YYCURSOR` and `YYMARKER` before returning. Typically `YYCURSOR`, `YYLIMIT`, `YYMARKER`, and `YYFILL(`n`)` will be defined as macros.

2.1 Things That RE2C Doesn't Provide

RE2C doesn't provide many things available in more conventional scanner generators including default rules, end-of-input pseudo-tokens, and buffer management routines. All of these must be supplied by the user. Rather than being a handicap, this allows RE2C-generated scanners to be tailored to almost any environment. For example, the scanner defined in Figure 1 compiles into 32 bytes of i486 code (using Watcom C 9.5); the same size as an equivalent hand-coded routine. Most other scanner generators cannot produce scanners that are competitive with hand-coded analyzers in this case. Further, it is not overly difficult to implement a more traditional scanner using RE2C. For example, Appendix A contains the support code for the C scanner benchmarked in Table 1. Note that this code allows for arbitrarily long contiguous tokens and provides line and column number information.

3 Generating Directly Executable Scanners

As demonstrated by GLA [7] generating directly executable code instead of tables can result in much faster scanners. However, to achieve this speed, GLA-generated scanners make some assumptions about the input and place certain restrictions on tokens[4]. In this section we will show how to generate directly executable scanners which not only avoid such restrictions, but are also faster and usually smaller. The approach presented here has the added benefit that even faster scanners can be easily be created, at the expense of increased code size, by using a technique akin to loop unrolling.

3.1 Constructing a DFA

The first step in generating a directly executable scanner is to construct a DFA that recognizes the regular expressions in the specification. Figure 3 presents a DFA that recognizes the regular expressions in Figure 2. One possible algorithm for constructing such a DFA can be found in [1]. Given such a DFA, the task of

[4] These assumptions and restrictions are discussed in more detail in Sections 3.3.1 and 5.1.

scanning the input can be expressed as follows:

> Starting from the start state, move from state to state along transitions labeled with consecutive characters from the input. When no further transitions can be made, backtrack to the last accepting state, say q. The path to q spells the next token and the rule associated with q determines the code to be executed.

As a result, the problem of generating scanners essentially reduces to the problem of generating an executable representation for a DFA.

3.2 Generating Code

If we assume that the input is entirely contained in a single buffer then generating code for the DFA is relatively straightforward, as is illustrated by the code templates in Figure 4. Note that the only difference

Prologue

```
        int yyaccept;
        goto Mstart;
        fin: YYCURSOR = YYMARKER;
        switch(yyaccept){
            ...
An:       case n:   action(n);
            ...
        }
        code for states
```

Code for accepting state

```
Lq:   ++YYCURSOR;
      yyaccept = rule(q);
      YYMARKER = YYCURSOR;
Mq:   switch(*YYCURSOR){
          ...
        case c:  goto Lgoto(q,c);
          ...
        default:  goto fin;
      }
```

Code for non-accepting state

```
Lq:   ++YYCURSOR;


Mq:   switch(*YYCURSOR){
          ...
        case c:  goto Lgoto(q,c);
          ...
        default:  goto fin;
      }
```

Figure 4: Directly executable scanner. The code generated for a scanner consists of a prologue followed by code for each state. *start* is the start state. *action*(n) denotes the code associated with rule n, *goto*(q, c) denotes the state reached from state q along the transition labeled with c and *rule*(q) denotes the rule associated with state q. `yyaccept` is used to save backtracking information. The `M`-labels will be used in section 3.4.2.

between the templates for accepting and non-accepting states is that the accepting states have additional code to save backtracking information. Figure 5 shows code that might be generated for state 1 in Figure 3.

3.3 Buffering

Complications arise when the input is not contained in a single buffer: additional code is needed for filling the buffer as necessary.

```
L1:   ++YYCURSOR;
      yyaccept = 4;
      YYMARKER = YYCURSOR;
M1:   switch(*YYCURSOR){
        case 'a':  goto L6;
          ...
        case 'q':  goto L6;
        case 'r':  goto L2;
        case 's':  goto L6;
          ...
        case 'z':  goto L6;
        default:  goto fin;
      }
```

Figure 5: Code for state 1.

```
L6:   ++YYCURSOR;
      if(YYLIMIT == YYCURSOR) YYFILL(1);
      yyaccept = 4;
      YYMARKER = YYCURSOR;
M6:   switch(*YYCURSOR){
          ...
      }
```

Figure 6: Code for state 6.

3.3.1 The Sentinel Method

Most scanner generators use the sentinel method [1] to determine when the buffer needs filling. In the simplest case, a symbol that does not appear in valid input is chosen as the sentinel character. An extra state is added to the DFA and transitions to this state on the sentinel symbol are added to the original states. When the DFA arrives in this new state it is time to refill the buffer. After the buffer is refilled, scanning must be restarted in the previous state. Unfortunately, this is not possible with the approach outlined in Figure 4: the necessary information is simply not available. Code could be added to each state to save the necessary information but this would result in slower and larger scanners. GLA solves this problem by ensuring that the sentinel only gets inserted between tokens: if this is the case, the scanner can always be restarted from the start state. To ensure that the sentinel only gets inserted between tokens, GLA allows newline (ASCII LF) characters to appear only at the end of a token and disallows the buffering of partial lines[5].

3.3.2 Buffering

RE2C-generated scanners check if the buffer needs filling simply by comparing `YYCURSOR` and `YYLIMIT`. A method inspired by the mechanism used to guard against stack overflow in [17][6] is used to reduce the amount of checking.

Checks are only inserted in certain key states. These checks simply ensure that there is enough input in the buffer for the scan to proceed until the next key state. For example, in the DFA of Figure 3 it is sufficient to check that there are at least 6 characters in the buffer when it starts, and that there is at least one character in the buffer when the DFA is in states 6, 8, or 10. No other checks are required. The checks inserted in key states are of the form

```
if((YYLIMIT - YYCURSOR) < n) YYFILL(n);
```

where n is the maximum number of characters that can be consumed before another key state is reached. For example, Figure 6 shows the code generated for state 6 in Figure 3.

A set of key states can be determined by discovering the strongly-connected components (SCCs) of the DFA. An SCC is a maximal subset of states such that there exists a path from any state in the subset to any other. The set of key states consists of all of the states in non-trivial SCCs, together with the start state. Note that for each SCC S, we actually only have to include a subset of states of S such that when the subset is removed, S becomes acyclic. Indeed, [17] describes a simple heuristic for choosing such a subset. However, since in practice most of the (non-trivial) SCCs encountered will consist of a single state the current version of RE2C simply includes all states in non-trivial SCCs[7]. An algorithm given in [3] was used to compute the SCCs.

3.4 Optimizations

Even good optimizing C compilers can be coaxed into generating much smaller and slightly faster code if some transformations are first applied to the generated code.

3.4.1 Eliminating Backtracking

Consider state 1 in the DFA in Figure 3. Note that since all of the transitions from state 1 reach only accepting states, backtracking information does not need to be saved if the code for the `default` case is changed to go directly to the code associated with state 1. The result of this optimization is shown in Figure 7. More generally, this optimization can be applied to all accepting states which have transitions

```
L1:  ++YYCURSOR;
M1:  switch(*YYCURSOR){
       case 'a':  goto L6;
         ...
       case 'q':  goto L6;
       case 'r':  goto L2;
       case 's':  goto L6;
         ...
       case 'z':  goto L6;
       default:  goto A4;
     }
```

Figure 7: Code for state 1 with backtracking eliminated.

only to accepting states.

3.4.2 Optimizing `switch`es

Most C compilers will generate either a jump table or a set of `if` statements for a `switch` statement depending on the distribution of the `case` labels. In many compilers the decision as to which method to use is biased towards generating jump tables since in most cases this results in faster albeit larger code. However, experience with directly executable scanners has shown, that replacing many of these jump tables

[5] If the input contains no newlines, a GLA scanner will attempt to buffer the entire input stream.

[6] The problem of detecting stack overflow in LR parsers is probably best left to hardware mechanisms [12].

[7] It should be noted that finding the minimal set of states to remove from an SCC in order to render it acyclic is equivalent to the FEEDBACK VERTEX SET problem which is NP-complete [6].

with `if` statements results in scanners which are much smaller, and surprisingly, in some cases slightly faster as well[8]. As a result, the capability of replacing a `switch` statement with `if` statements was added to RE2C.

RE2C bases its decision on whether to generate a `switch` statement or to replace it with `if`s solely on the density[9] of the `switch` statement. It is surprising that such a simple heuristic works well. For more esoteric applications in which the input alphabet is not a simple interval RE2C has the advantage in that there is no provision for don't care entries in a `switch` statement: if no `case` matches none of the statements in the `switch` must be executed. However, for the examples in Table 1 this is not so: RE2C simply does a better job of generating code for `switch` statements than the compiler. [18], [11], and [2] also address the problem of generating good code for `switch` statements.

Replacing `switch`es with `if`s When replacing a `switch` statement with `if` statements, it is useful to sort the `case`s by label and then group them according to rule into subranges, as illustrated in Figure 8. RE2C replaces a `switch` with either a linear or binary search, depending on the number of subranges in the

```
switch(*YYCURSOR){
  case '\000':  ... case '/':     goto L11;
     case '0':                    goto L7;
     case '1':  ... case '9':     goto L8;
     case ':':  ... case '`':     goto L11;
     case 'a':  ... case 'o':     goto L6;
     case 'p':                    goto L1;
     case 'r':  ... case 'z':     goto L6;
     case '{':  ... case '\377':  goto L11;
}
```

Figure 8: `switch` for state 0.

`switch`. If there are only a few subranges a linear search is generated; otherwise, a binary search is used.

Figure 9 and Figure 10 show linear and binary searches, respectively, that could be used to replace the

```
if(*YYCURSOR <= '/') goto L11;
if(*YYCURSOR <= '0') goto L7;
if(*YYCURSOR <= '9') goto L8;
if(*YYCURSOR <= '`') goto L11;
if(*YYCURSOR == 'p') goto L1;
if(*YYCURSOR <= 'z') goto L6;
goto L11;
```

Figure 9: Linear lookup code sequence for state 0.

[8] See Table 1 for examples.

[9] The number of distinct subranges divided by the total number of cases.

```
if(*YYCURSOR <= '`'){
    if(*YYCURSOR <= '/') goto L11;
    if(*YYCURSOR <= '0') goto L7;
    if(*YYCURSOR <= '9') goto L8;
    goto L11;
} else {
    if(*YYCURSOR == 'p') goto L1;
    if(*YYCURSOR <= 'z') goto L6;
    goto L11;
}
```

Figure 10: Binary lookup code sequence for state 0.

`switch` in Figure 8. Note in particular the comparison for the "`p`" in Figure 9. This optimization eliminates a comparison each time it is applied. Also note that no comparisons are required at the top and bottom of the range.

Simplifying `switch`es As a general rule, better replacement code can be generated for a `switch` if it contains fewer subranges. One way of reducing the number of subranges in a `switch`, at the expense of some speed, is to locate a *base* `switch` which is very similar and then replace the code for all cases which appear identically in the base `switch` with a `goto` to (the code generated for) the base `switch`. RE2C uses this optimization to good advantage when generating code in the transitions of states used for matching keywords. For example, note that the `switch`es for states 1 through 4 differ from the `switch` of state 6 only on "`r`", "`i`", "`n`", and "`t`", respectively. Figure 11 shows the code generated for these states. Another way

```
L1:  ++YYCURSOR;
M1:  if(*YYCURSOR != 'r') goto M6;
L2:  ++YYCURSOR;
M2:  if(*YYCURSOR != 'i') goto M6;
L3:  ++YYCURSOR;
M3:  if(*YYCURSOR != 'n') goto M6;
L4:  ++YYCURSOR;
M4:  if(*YYCURSOR != 't') goto M6;
     goto L5;
```

Figure 11: Code for states 1–4 after all optimizations.

of implementing this optimization is to construct a tunnel automaton [9] from the DFA, and then generate code from the tunnel automaton.

Common Subexpression Elimination Many compilers will miss the fact that `*YYCURSOR` in Figures 9 and 10 should be loaded into a register. Most can be coaxed to do so by first assigning `*YYCURSOR` to a local variable.

4 Experimental Results

Table 1 compares two RE2C-generated C scanners with the (hand-coded) lcc scanner [5] and comparable GLA- and flex-generated scanners on a variety of platforms. It reports the times in seconds required by the various scanners to scan about 170,000 lines of C source. The 5,607,820 byte source file used essentially

program	time			space			
	user	sys	total	text	data	bss	total
R4000 / gcc2.3.3 -O							
flex -Cem	10.36	0.87	11.23	5200	4192	48	9440
flex -Cf	5.44	0.72	6.16	4688	64384	48	69120
lcc	3.19	0.67	3.86	7328	1216	8256	16800
gla	2.89	0.63	3.52	11552	3056	144	14752
re2c	2.54	0.68	3.22	13264	512	0	13776
re2c -s	2.38	0.67	3.05	11056	4528	0	15584
R4000 / cc2.11.2 -O -Olimit 5000							
flex -Cem	9.97	0.89	10.86	4704	4240	32	8976
flex -Cf	6.19	0.72	6.91	4256	64432	32	68720
lcc	2.74	0.72	3.46	9664	864	8256	18784
gla	2.46	0.69	3.15	19232	2992	128	22352
re2c	2.97	0.63	3.60	15088	528	0	15616
re2c -s	2.94	0.61	3.55	16080	11808	0	27888
SPARC / gcc2.3.3 -O							
flex -Cem	16.03	2.78	18.81	8992	24	48	9064
flex -Cf	7.84	2.69	10.53	6560	62232	48	68840
lcc	4.46	2.01	6.47	7800	384	8256	16440
gla	4.08	1.56	5.64	10864	2168	136	13168
re2c	3.67	1.76	5.43	13552	0	0	13552
re2c -s	3.48	1.70	5.18	15464	0	0	15464
i486 / gcc2.4.5 -O							
flex -Cem	21.86	1.26	23.12	8536	20	24	8580
flex -Cf	9.12	1.18	10.30	6200	62228	24	68452
lcc	5.45	1.22	6.67	5924	384	8240	14548
gla	5.11	1.18	6.29	15496	2144	108	17748
re2c	4.73	1.13	5.86	9800	0	0	9800
re2c -s	4.85	1.17	6.02	12968	0	0	12968
68020 / gcc1.40 -O							
flex -Cem	117.37	5.89	123.26	7700	20	22	7742
flex -Cf	50.93	5.27	56.20	5388	62228	22	67638
lcc	33.28	6.28	39.56	4956	384	8236	13576
gla	33.80	4.20	38.00	13904	2144	106	16154
re2c	28.92	2.91	31.83	8556	0	0	8556
re2c -s	30.72	3.19	33.91	9856	0	0	9856

Table 1: Comparison of generated C scanners.

consists of 10 copies of the source to James Clark's SGML parser, sgmls[10]. The times reported are averages for 10 trials; the sizes reported include everything but C library code[11]. flex provides a number of table compression options including `-Cem` for tables optimized for space, and `-Cf` for tables optimized for speed. By default, RE2C will use a heuristic to decide if a `switch` should be replaced with `if`s: the `-s` option forces RE2C to always generate `switch`es.

To make comparisons more meaningful, all semantic processing code was removed from the GLA-generated and lcc scanners, and code to provide line and column number information was added to the RE2C specification. The remaining differences of note between the scanners include:

- The flex-generated scanners do not provide line or column number information.
- The GLA-generated scanner assumes 7-bit input.

As a general rule, the RE2C-generated scanners were the fastest, followed by the GLA-generated scanner and then the lcc scanner. The flex-generated scanners were significantly slower. Only the space-optimized flex scanner was smaller than the default RE2C scanner, and only by a narrow margin. There are some architectures, notably the IBM 370, on which table driven scanners will probably produce better results: IBM 370 compilers typically generate poor code for large routines.

The various scanners and input files used for the tests are available for anonymous ftp from csg.uwaterloo.ca in /pub/peter/re2c/sampler.tar.Z. flex is available for anonymous ftp from ftp.uu.net as /packages/gnu/flex-2.3.7.tar.Z, GLA is available for anonymous ftp from ftp.cs.colorado.edu as part of the Eli package /pub/cs/distribs/eli/Eli3.4.2.tar.Z, and the lcc front end is available for anonymous ftp from princeton.edu as /pub/lcc/lccfe-1.9.tar.Z. An alpha version of RE2C will soon be made available for anonymous ftp from csg.uwaterloo.ca as /pub/peter/re2c/re2c-0.5.tar.Z.

5 Related Work

The key to the performance and flexibility of an RE2C-generated scanner is the approach used to determine when the buffer needs filling. Interestingly, the lcc scanner [5] uses a similar approach (with certain concessions to keep the bookkeeping manageable.)

5.1 Comparison With GLA

It is natural to compare RE2C to GLA [7] as it also generates directly executable scanners. RE2C and GLA have many differences simply because they are targeted for different types of users: GLA is intended for people who simply wish to leverage their efforts with existing tools and libraries; RE2C is intended for people that have more specialized needs and are willing to provide their own support routines. For example, GLA provides a good buffering mechanism, RE2C users must supply their own. These differences, however, are not unique to GLA and have been addressed for the most part in previous sections.

Of more interest is the differences in the code that RE2C and GLA generate. Scanners generated by RE2C and GLA differ primarily in two aspects: how they determine when the buffer needs filling, and how they generate code for `switch`es.

GLA uses the ASCII NUL character as the sentinel to determine when the buffer needs filling. To improve the speed and reduce the size of the generated scanners GLA buffers only complete lines and restricts tokens to those that do not contain newline (ASCII LF) characters[12]. If a token with an embedded newline character (such as a comment) is required it must be recognized with an auxiliary scanner written in C. This code has to perform the buffering-related bookkeeping that is done automatically by GLA-generated code.

[10] Available for anonymous ftp from ftp.uu.net as /pub/text-processing/sgml/sgmls-1.1.tar.Z.
[11] The GLA-generated scanner sizes also do not include the size of an error reporting module `err.o`.
[12] This is discussed in more detail in Section 3.3.1.

The mechanism RE2C uses to refill the buffer eliminates these restrictions and yet allows RE2C to generate faster and smaller scanners. RE2C also allows both auxiliary and primary scanners to be specified using regular expressions. For example, Appendix A contains an auxiliary scanner for comments.

Like RE2C, GLA usually replaces `switch`es with `if`s. Unlike RE2C, GLA does not use a `case`-based heuristic to decide which `switch`es to replace: rather, it always generates a `switch` for the start state and uses `if`s for the rest. GLA replaces `switch`es with code sequences of the form:

```
if(*YYCURSOR in S1) goto L1;
    ⋮
if(*YYCURSOR in Sn) goto Ln;
```

Bit vectors are used for all membership tests involving sets with more than one element. As an optimization, if a state has a transition to itself the test as to whether to remain in the same state or not is performed first. For example, Figure 12 shows the GLA-generated code for state 8 in Figure 2[13]. Note the use of

```
static unsigned char yytable[] = {
0x00, 0x00, 0x00, 0x00, /*   0.   1.   2.   3. */
...
0x00, 0x00, 0x00, 0x00, /*  ,  -  .  / */
0x01, 0x01, 0x01, 0x01, /*  0  1  2  3 */
0x01, 0x01, 0x01, 0x01, /*  4  5  6  7 */
0x01, 0x01, 0x00, 0x00, /*  8  9  :  ; */
0x00, 0x00, 0x00, 0x00, /*  <  =  >  ? */
...
0x00, 0x00, 0x00, 0x00 }; /*  |  }     127. */
     ⋮
L8: if(yytable[(*YYCURSOR++)+0] & 1<<0) goto L8;--YYCURSOR;
goto A3;
```

Figure 12: GLA code for state 8 in Figure 2.

128 element entries for the bit vectors to reduce the scanner size: A GLA-generated scanner will crash or otherwise behave unpredictably if a non-ASCII character appears in the source[14].

In some sense the results of Section 4 are a bit misleading: the GLA specification that was used to obtain the figures in Table 1 is not a typical GLA specification. Usually scanners implemented using GLA will handle keywords as identifiers as GLA has been optimized for this [7]. Table 2 presents a more fair comparison: the keyword matching rules were removed from both the GLA and RE2C specifications. The RE2C-generated scanners were still faster and smaller except on the MIPS R4000, where the cc-compiled GLA scanner was slightly faster.

Note however, that the RE2C specification can be substantially sped up by using a technique akin to loop unrolling. Replacing the original keyword matching rule in the RE2C specification[15]

```
L I*                    { RET(ID); }
```

with the following rules

[13] Actually, GLA would generate a `while` statement. Most compilers will generate the same object code for both.
[14] No checks are made to ensure that only 7-bit characters appear in the input.
[15] `L = [a-zA-Z_]` and `I = [a-zA-Z_0-9]`.

	time			space			
program	user	sys	total	text	data	bss	total
R4000 / gcc2.3.3 -O							
gla	2.63	0.58	3.21	5040	2496	144	7680
re2c	2.50	0.65	3.15	6448	512	0	6960
re2c -s	2.49	0.67	3.16	4976	4224	0	9200
re2c -s †	2.08	0.59	2.67	5792	4224	0	10016
R4000 / cc2.11.2 -O -Olimit 5000							
gla	2.43	0.64	3.07	6512	2416	128	9056
re2c	2.93	0.67	3.60	8048	528	0	8576
re2c -s	3.04	0.64	3.68	9952	2208	0	12160
SPARC / gcc2.3.3 -O							
gla	4.08	1.65	5.73	5472	1656	136	7264
re2c	3.77	1.67	5.44	7008	0	0	7008
re2c -s	3.66	2.37	6.03	9112	0	0	9112
i486 / gcc2.4.5 -O							
gla	5.04	1.15	6.19	5368	1632	108	7108
re2c	4.75	1.17	5.92	5448	0	0	5448
re2c -s	5.06	1.13	6.19	8248	0	0	8248
68020 / gcc1.40 -O							
gla	32.69	3.37	36.06	4772	1632	106	6510
re2c	29.86	3.74	33.60	4468	0	0	4468
re2c -s	28.77	3.55	32.32	5616	0	0	5616

Table 2: Scanner performance with keywords treated as identifiers. † uses an "unrolled" specification.

```
L                       { RET(ID); }
L I                     { RET(ID); }
L I I                   { RET(ID); }
L I I I                 { RET(ID); }
L I I I I               { RET(ID); }
L I I I I I             { RET(ID); }
L I I I I I I           { RET(ID); }
L I I I I I I I         { RET(ID); }
L I*                    { RET(ID); }
```

reduces the number of end-of-buffer checks and results in a significant speed improvement over the GLA-generated scanner.

6 Summary and Further Work

This paper has described RE2C, a tool for creating lexical analyzers. Unlike other such tools, RE2C concentrates solely on generating efficient code for matching regular expressions. Not only does this singleness of purpose make RE2C more suitable for a wider variety of applications, it allows it to generate scanners which approach hand-crafted scanners in terms of size and speed. Compared to scanners generated by flex, and GLA, RE2C-generated scanners are faster and in many cases smaller as well.

While RE2C-generated scanners perform well, there is still room for improvement. Near term improvements include using GLA's bit vectors to simplify some `switch`es and adding a state unrolling operator.

In the longer term, inline actions will be added to RE2C. For example, a specification like

```
D {c = $} (D {c = 10*c + $})*
```

might be used to obtain the value of a previously scanned integer. Typically, these sorts of specifications would be used as an action in some other specification.

7 Acknowledgments

The authors thank the referees for their many valuable comments and suggestions.

A C Scanner

```
#define BSIZE           8192
#define RET(i)          {s->cur = cursor; return i;}

#define YYCTYPE         uchar
#define YYCURSOR        cursor
#define YYLIMIT         s->lim
#define YYMARKER        s->ptr
#define YYFILL(n)       {cursor = fill(s, cursor);}

typedef struct Scanner {
    int                 fd;
    uint                line;
    uchar               *bot, *tok, *ptr, *cur, *pos, *lim, *top, *eof;
} Scanner;

uchar *fill(Scanner *s, uchar *cursor){
    if(!s->eof){
        uint cnt = s->tok - s->bot;
        if(cnt){ /* move partial token to bottom */
            memcpy(s->bot, s->tok, s->lim - s->tok); s->tok = s->bot;
            s->ptr -= cnt; cursor -= cnt; s->pos -= cnt; s->lim -= cnt;
        }
        if((s->top - s->lim) < BSIZE){ /* buffer needs to be expanded */
            uchar *buf = (uchar*) malloc(((s->lim - s->bot) + BSIZE)*sizeof(uchar));
            memcpy(buf, s->tok, s->lim - s->tok); s->tok = buf;
            s->ptr = &buf[s->ptr - s->bot]; cursor = &buf[cursor - s->bot];
            s->pos = &buf[s->pos - s->bot]; s->lim = &buf[s->lim - s->bot];
            s->top = &s->lim[BSIZE];
            free(s->bot); s->bot = buf;
        }
        if((cnt = read(s->fd, (char*) s->lim, BSIZE)) != BSIZE){ /* EOF */
            s->eof = &s->lim[cnt]; *(s->eof)++ = '\n';
        }
        s->lim += cnt;
    }
    return cursor;
}

int scan(Scanner *s){
        uchar *cursor = s->cur;
std:    s->tok = cursor;
/*!re2c
        "/*"         { goto comment; }
```

... more rules ...

```
        [ \t\v\f]+   { goto std; }
        "\n"         { if(cursor == s->eof) RET(EOI); s->pos = cursor; s->line++;
                       goto std; }
        [\000-\377]  { printf("unexpected character: '%c'\n", *s->tok);
                       goto std; }
*/
comment:
/*!re2c
        "*/"         { goto std; }
        "\n"         { if(cursor == s->eof) RET(EOI); s->tok = s->pos = cursor; s->line++;
                       goto comment; }
        [\000-\377]  { goto comment; }
*/
}
```

References

[1] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: principles, techniques, and tools.* Addison-Wesley, 1988. Reprinted with corrections.

[2] Bernstein, R. L. Producing good code for the case statement. *Software–Practice and Experience 15*, 10 (October 1985), 1021–1024.

[3] DeRemer, F., and Pennello, T. Efficient computation of $LALR(1)$ look-ahead sets. *ACM Transactions on Programming Languages and Systems 4*, 4 (October 1982), 615–649.

[4] Ellis, M., and Stroustrup, B. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[5] Fraser, C. W., and Hanson, D. R. A retargetable compiler for ANSI C. *SIGPLAN Notices 26*, 10 (October 1991), 29–43.

[6] Garey, M. R., and Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1991.

[7] Gray, R. W. γ-GLA - A generator for lexical analyzers that programmers can use. *USENIX Conference Proceedings* (June 1988), 147–160.

[8] Gray, R. W., Heuring, V. P., Levi, S. P., Sloane, A. M., and Waite, W. M. Eli: A complete, flexible compiler construction system. *Communications of the ACM 35*, 2 (February 1992), 121–131.

[9] Grosch, J. Efficient generation of lexical analysers. *Software–Practice and Experience 19*, 11 (1989), 1089–1103.

[10] Harrison, M. A. *Introduction to Formal Language Theory.* Addison-Wesley, 1978.

[11] Hennessy, J. L., and Mendelsohn, N. Compilation of the Pascal case statement. *Software–Practice and Experience 12*, 9 (September 1982), 879–882.

[12] Horspool, R. N., and Whitney, M. Even faster LR parsing. *Software–Practice and Experience 20*, 6 (1990), 515–535.

[13] Jacobson, V. Tuning UNIX Lex or it's NOT true what they say about Lex. In *USENIX Conference Proceedings* (Washington, DC, Winter 1987), pp. 163–164. Abstract only.

[14] Kernighan, B. W., and Ritchie, D. M. *The C Programming Language, 2nd Ed.* Prentice-Hall, Inc., 1988.

[15] Lesk, M. E. LEX – a lexical analyzer generator. Computing Science Technical Report 39, Bell Telephone Laboratories, Murray Hill, NJ, 1975.

[16] Paxson, V. flex – man pages, 1988. In flex-2.3.7.tar.Z. Available for anonymous ftp from ftp.uu.net in /packages/gnu.

[17] Pennello, T. J. Very fast LR parsing. In *Proceedings of the ACM SIGPLAN'86 Symposium on Compiler Construction* (July 1986), ACM.

[18] Sale, A. The implementation of case statements in Pascal. *Software–Practice and Experience 11*, 9 (September 1981), 929–942.