“Maximal-Munch” Tokenization in Linear Time

THOMAS REPS
University of Wisconsin

The lexical-analysis (or scanning) phase of a compiler attempts to partition the input stream into a sequence of tokens. The convention in most languages is that the input is scanned left to right, and each token identified is a “maximal munch” of the remaining input—the *longest* prefix of the remaining input that is a token of the language. Most textbooks on compiling have extensive discussions of lexical analysis in terms of finite-state automata and regular expressions: Token classes are defined by a set of regular expressions R_i, $1 \le i \le k$, and the lexical analyzer is based on some form of finite-state automaton for recognizing the language $L(R_1 + R_2 + \cdots + R_k)$. However, the treatment is unsatisfactory in one respect: The theory of finite-state automata assumes that the end of the input string—*i.e.*, the right-hand-side boundary of the candidate for recognition—is known *a priori*, whereas a scanner must identify the next token *without* knowing a definite bound on the extent of the token.

Although most of the standard compiler textbooks discuss this issue, the solution they sketch out is one that—for certain sets of token definitions—can cause the scanner to exhibit quadratic behavior in the worst case. This property is not only dissatisfying, it blemishes an otherwise elegant treatment of lexical analysis.

In this paper, we rectify this defect: We show that, given a deterministic finite-state automaton that recognizes the tokens of a language, maximal-munch tokenization can always be performed in time linear in the size of the input.

CR Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory – *syntax*; D.3.4 [**Programming Languages**]: Processors – *compilers*; F.1.1 [**Computation by Abstract Devices**]: Models of Computation – *automata*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems – *pattern matching*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search – *backtracking, dynamic programming*; I.5.4 [**Pattern Recognition**]: Applications – *text processing*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: memoization, tabulation, tokenization

1. INTRODUCTION

The lexical-analysis (or scanning) phase of a compiler attempts to partition the input stream into a sequence of tokens. The convention in most languages is that the input is scanned left to right, and each token identified is a “maximal munch” of the remaining input—the *longest* prefix of the remaining input that is a token of the language. For example, the string “12” should be tokenized as a single integer token “12”, rather than as the juxtaposition of two integer tokens “1” and “2”. Similarly, “123.456e–10” should be recognized as one floating-point numeral rather than, say, the juxtaposition of the four tokens “123”,

This work was supported in part by the National Science Foundation under grant CCR-9625667, and by the Defense Advanced Research Projects Agency (monitored by the Office of Naval Research under contracts N00014-92-J-1937 and N00014-97-1-0114).

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes, notwithstanding any copyright notices affixed thereon. The views and conclusions contained herein are those of the authors, and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the above government agencies or the U.S. Government.

Author’s address: Computer Sciences Department, University of Wisconsin, 1210 W. Dayton St., Madison, WI 53706.
E-mail: reps@cs.wisc.edu.

".456", "e", and "−10". (Usually, there is also a rule that when two token definitions match the same string, the earliest token definition takes precedence. However, this rule is invoked only if there is a tie over the longest match.)

Most textbooks on compiling have extensive discussions of lexical analysis in terms of finite-state automata and regular expressions: Token classes are defined by a set of regular expressions R_i, $1 \le i \le k$, and the lexical analyzer is based on some form of finite-state automaton for recognizing the language $L(R_1 + R_2 + \cdots + R_k)$. However, the treatment is unsatisfactory in one respect: The theory of finite-state automata assumes that the end of the input string—*i.e.*, the right-hand-side boundary of the candidate for recognition—is known *a priori*, whereas a scanner must identify the next token *without* knowing a definite bound on the extent of the token.

Most of the standard compiler textbooks, including [2,15,3,8,16], discuss this issue briefly. For example, Aho and Ullman's 1977 book discusses the issue in the context of a lexical analyzer based on DFAs:

> There are several nuances in this procedure of which the reader should be aware. First, there are in the combined NFA several different "accepting states". That is, the accepting state of each N_i indicates that its own token, P_i, has been found. When we convert to a DFA, the subsets we construct may include several different final states. Moreover, the final states lose some of their significance, since we are looking for the longest prefix of the input which matches some pattern. After reaching a final state, the lexical analyzer must continue to simulate the DFA until it reaches a state with no next state for the current input symbol. Let us say we reach *termination* when we meet an input symbol from which the DFA cannot proceed. We must presume that the programming language is designed so that a valid program cannot entirely fill the input buffer . . . without reaching termination . . .
>
> Upon reaching termination, it is necessary to review the states of the DFA which we have entered while processing the input. Each such state represents a subset of the NFA's states, and we look for the last DFA state which includes a final state for one of the pattern-recognizing NFA's N_i. That final state indicates which token we have found. If none of the states which the DFA has entered includes any final states of the NFA, then we have an error condition. If the last DFA state to include a final NFA state in fact includes more than one final state, then the final state for the pattern listed first has priority [2, pp. 109–110].

The 1986 book by Aho, Sethi, and Ullman discusses the issue in the context of the design of the lexical-analyzer generator Lex [12], under the assumption that the lexical analyzer generated will simulate an NFA, rather than first convert the NFA to a DFA:

> To simulate this NFA we can use [an algorithm that] ensures that the combined NFA recognizes the longest prefix of the input that is matched by a pattern. In the combined NFA, there is an accepting state for each pattern p_i. When we simulate the NFA . . . we construct the sequence of sets of states that the combined NFA can be in after seeing each input character. Even if we find a set of states that contains an accepting state, to find the longest match we must continue to simulate the NFA until it reaches *termination*, that is, a set of states from which there are no transitions on the current input symbol.
>
> We presume that the Lex specification is designed so that a valid source program cannot entirely fill the input buffer without having the NFA reach termination . . .
>
> To find the correct match, we make two modifications [to the NFA-simulation algorithm]. First, whenever we add an accepting state to the current set of states, we record the current input position and the pattern p_i corresponding to this accepting state. If the current set of states already contains an accepting state, then only the pattern that appears first in the Lex specification is recorded. Second, we continue making transitions until we reach termination. Upon termination, we retract the forward pointer to the position at which the last match occurred. The pattern making this match identifies the token found, and the lexeme matched is the string between the lexeme-beginning and forward pointers [3, pp. 104].

The discussions of the problem given in Waite and Goos [15], Fischer and LeBlanc [8], and Wilhelm and

Maurer [16] are similar.

However, regardless of whether the tokenization process is based on DFAs or NFAs, the recommended technique of backtracking to the most recent final state and restarting is not entirely satisfactory: It has the drawback that—for certain sets of token definitions—it can cause the scanner to exhibit quadratic behavior in the worst case.[1] For example, suppose that our language has just two classes of tokens, defined by the regular expressions "*abc*" and "$(abc)^*d$", and suppose further that the input string is a string of m repetitions of the string *abc* (*i.e.*, $(abc)^m$). To divide this string into tokens, the scanner will advance to the end of the input, looking for—and failing to find—an instance of the token "$(abc)^*d$". It will then back up $3(m-1)+1$ characters to the end of the first instance of *abc*, which is reported as the first token. A similar pattern of action is repeated to identify the second instance of *abc* as the second token: The scanner will advance to the end of the input, looking for—and failing to find—an instance of the token "$(abc)^*d$"; it will then back up $3(m-2)+1$ characters to the end of the second instance of *abc*. Essentially the same pattern of action is repeated for the remaining $m-2$ tokens, and thus this method performs $\Theta(m^2)$ steps to tokenize inputs of the form $(abc)^m$.

This drawback serves to blemish the otherwise elegant treatment of lexical analysis in terms of finite-state automata and regular expressions.

The possibility of quadratic behavior is particularly unsettling because the separation of syntax analysis into separate phases of lexical analysis and parsing is typically justified on the grounds of simplicity.[2] Because a program's syntax is typically defined with an LL, LALR, or LR grammar, the parsing phase can always be carried out in linear time. It is a peculiar state of affairs when the recommended technique for the supposedly simpler phase of lexical analysis could use more than linear time.

Remark. It is not the division of syntax analysis into separate phases of lexical analysis and parsing *per se* that is source of the trouble. Even if the token classes were specified with an LR grammar, say, we would still have the problem of designing an efficient automaton that identifies the *longest* prefix of the remaining input that is a token of the language. □

In this paper, we show that, given a deterministic finite-state automaton that recognizes the tokens of a language, maximal-munch tokenization can always be performed in time linear in the size of the input. We present two linear-time algorithms for the tokenization problem. Both techniques rely on tabulation (or "memoization") to avoid repeating work that is known not to lead to the identification of a new token:

(i) In Section 2, we obtain a linear-time algorithm by a somewhat indirect method. We make use of a result due to Mogensen [14], extending an earlier result by Cook [6], that a certain variant of a two-way deterministic pushdown automaton (a so-called "WORM-2DPDA") can be simulated in linear time on a RAM computer (even though the WORM-2DPDA itself may perform many more than a linear number of steps). Given a deterministic finite-state automaton that recognizes the tokens of a language, we describe how to construct a WORM-2DPDA that performs tokenization; hence, we

---

[1] None of abovementioned books explicitly point out that quadratic behavior is possible.

[2] For instance, Aho, Sethi, and Ullman say

> The separation of lexical analysis from [parsing] often allows us to simplify one or the other of these phases . . . Compiler efficiency is improved . . . [and] compiler portability is enhanced [3, pp. 84–85].

know that using Mogensen's construction, this can be converted into a linear-time RAM program for tokenization.

(ii) In Section 2, we do not actually give the details of the program that results from Mogensen's construction. However, it serves as a proof of the existence of a linear-time algorithm, and motivates the development of a simpler solution, which is discussed in Section 3. The latter algorithm tabulates pairs of states and index positions that have been previously encountered in order to avoid repeating fruitless searches of the input text.

Section 4 presents a few concluding remarks.

In the remainder of the paper, we make the following assumptions:

- We assume that none of the regular expressions R_i that define the language's tokens admit λ, the empty string. That is, for all i, $\lambda \notin L(R_i)$.
- We assume that whitespace is treated as just another lexeme to be identified in the input stream (using a maximal munch), and that the next higher level of the compiler filters out the whitespace tokens. The latter phase is sometimes called *screening* [7,16].
- We ignore all issues related to buffering the input.
- We use n to denote the length of the input string.
- Suppose that the language's tokens are defined by the regular expressions $R_1, R_2, \cdots, R_k$. We assume that we are given a deterministic finite-state automaton (DFA) M such that $L(M) = L(R_1 + R_2 + \cdots + R_k)$. We make use of standard notation for DFAs (*e.g.*, see [9]). That is, a DFA M is a five-tuple $\langle Q, \Sigma, \delta, q_0, F \rangle$, where
  - Q is a finite nonempty set of *states*.
  - Σ is a finite nonempty set of *input symbols*.
  - δ, the *transition function*, is a partial function from $Q \times \Sigma$ to Q.
  - $q_0 \in Q$ is the *initial state*.
  - $F \subseteq Q$ is the set of *final states*.

2. An Indirect Solution Based on Simulation of WORM-2DPDAs

This section presents a somewhat indirect method for obtaining a linear-time solution to the maximal-munch tokenization problem. It makes use of a result due to Mogensen [14], extending an earlier result by Cook [6], that a certain variant of a two-way deterministic pushdown automaton (2DPDA) can be simulated in linear time on a RAM computer (even though the 2DPDA itself may perform many more than a linear number of steps).

A (one-head) two-way deterministic pushdown automaton (2DPDA) is a (one-head) deterministic pushdown automaton in which the head reading the input tape is permitted to move in both directions (*i.e.*, in the course of a single transition of the machine, the input head may remain where it was, move one square to the right, or move one square to the left). Cook showed that, through the use of dynamic programming, it is always possible to simulate a 2DPDA in linear time on a RAM [6,1]. That is, given the description of a 2DPDA and an input string x of size n, using a RAM it is always possible to determine in time $O(n)$ whether the 2DPDA accepts x. The technique exploits the fact that even though the 2DPDA might use as many as 2^n steps, there are only a linear number of possible "stack configurations" that the 2DPDA can be in. The RAM program uses dynamic programming to tabulate information about sequences of transitions

between these configurations. Whereas the 2DPDA might repeat certain computation sequences many times, the information obtained via dynamic programming allows the RAM program to take "shortcuts" that, in essence, allow it to skip the second and successive repetitions of these computation sequences and proceed directly to a configuration further along in the computation.)

Cook's result was extended by Jones, who gave an alternative 2DPDA simulation method that tabulates configuration transitions on-line, at run-time [10, 13]. This technique is similar to the technique of memoization of function calls in functional programs. Because the Cook construction tabulates configuration transitions off-line, it considers transitions for stack configurations that are not reachable from the initial configuration (and thus are ones that the 2DPDA would never enter). In contrast, Jones's technique only considers (and tabulates) transitions for stack configurations that the 2DPDA being simulated would also reach.

Jones's approach was further extended to a language of stack-manipulation programs by Andersen and Jones [5]. In their work, they showed how each program in the language could be compiled to a program that runs in linear-time (where the program contains code to tabulate transition information for the configurations that it encounters).

Using the compilation approach, Mogensen showed how to obtain linear-time programs for a variant of 2DPDAs that are equipped with one or more auxiliary "*w*rite-*o*nce, *r*ead-*m*any" tapes, whence the name "WORM-2DPDAs" [14]. The WORM tapes provide WORM-2DPDAs with a limited form of auxiliary storage. They represent a *limited* form of auxiliary storage because the following restrictions apply:

(i) The WORM tapes can be thought of as being a kind of writable store that is parallel to the input tape. However, a WORM tape can be accessed *only* at the same index position as the read head of the input tape: In addition to the current state, the symbol on the top of the stack, and the symbol at the current position of the input-tape head, the transition function for a WORM-2DPDA is allowed to depend on the symbol(s) on the WORM tape(s) at the same index position as the input-tape head. (We say that a transition that depends on a symbol from one or more of the WORM tapes "reads" those tapes.)

(ii) The transition function is also allowed to specify that a specific symbol is written on a square of the WORM tape (at the index position of the input-tape head). However, once a square on a WORM tape is written, it cannot be overwritten. That is, each (successful) write to a WORM tape is indelible. (A write to a square that has previously been written to is ignored, or, alternatively, can be deemed to be illegal.)

(iii) A read of a square of a WORM tape returns the value previously written on the square. If there has been no previous write to the square, then a default value (uniform for all the squares of the tape) is returned. (Alternatively, a read on a square that has never previously been written to can be deemed to be illegal.)

For our purposes, the significance of all of this is that, given a DFA M that recognizes the tokens of a language, it is easy to construct a WORM-2DPDA M' for the maximal-munch tokenization problem. M' has a single WORM tape, and works similarly to the standard (quadratic-time) technique of backtracking to the most recent final state and restarting [2,15,3,8,16]. The machine has two "modes": "forward mode" and "backtrack mode".

- In forward mode, the input-tape head moves exclusively to the right. M' simulates each transition of machine M (*i.e.*, the part of the transition function of M' that controls forward-mode computation incor-

porates the transition function of M). In addition, during the course of each transition of M', the state of M is pushed onto the stack. (The stack alphabet of M' consists of $Q \cup \{ \mathit{Bottom} \}$, where *Bottom* is a special bottom-of-stack symbol.) Thus, the stack records a trace of the progress of M.

- The trace of M recorded on the stack is used to implement the phase of "review[ing] the states of the DFA which we have entered while processing the input" [2], which is also carried out by the standard backtracking algorithm for tokenization. (Note, however, that the standard algorithm need not use a stack to carry out this review: As the input is scanned, it merely has to maintain a pair of variables, say *last_final_state_position* and *last_final_state*, to record the position and state, respectively, for the maximum index position at which M was in a final state).

  If M is unable to make any transition, or is not in a final state when the end of the input is reached, M' switches to backtrack mode. In backtrack mode, the input-tape head moves exclusively to the left; in each transition, M' pops an element from the stack until either a final state of M is encountered or the symbol *Bottom* is encountered. The latter condition implies that no left-to-right maximal-munch tokenization is possible, and M' halts with failure. On the other hand, when a state that is final state of M is popped off the stack, M' records this fact on the WORM tape by marking the current head position on the WORM tape, using a symbol distinct from the WORM tape's default value. The input-tape head is then moved one position to the right, a new instance of *Bottom* is pushed onto the stack, and M' re-enters forward mode.

M' halts with success when M is in a final state and the input-tape head of M' is at the end of the input tape. (Before halting, a mark is also placed on the final square of the WORM tape.) On successful termination, the WORM tape contains an indelible mark at the final character of each maximal-munch token.

Given the transition function for DFA M, it is easy to create WORM-2DPDA M': The transition function for M is augmented with appropriate additional components to control the stacking of the states of M during forward mode, and a bounded number of new states and transitions are added to implement backtrack mode.

WORM-2DPDA M' is quadratic in the worst case: To tokenize an input string of size n, it may perform as many as $O(n^2)$ moves. However, by applying Mogensen's construction for the linear-time simulation of WORM-2DPDAs to machine M', we (automatically) obtain a linear-time (*i.e.*, $O(n)$) RAM algorithm for the maximal-munch tokenization problem.

3. A Direct Solution

This section presents a direct solution to the problem of tokenization in linear time. The easiest way to understand the linear-time algorithm is to first consider an algorithm that does not run in linear time, namely the program shown in Figure 1(a). The algorithm shown in Figure 1(a) prints out the positions in the input string that are the final characters of each token. This algorithm can be viewed as either a description in an Algol-like language of the WORM-2DPDA from Section 2 (except that it emits an index position each time a maximal-munch token is recognized, rather than making an indelible mark on a WORM tape), or else as a variant of the standard (quadratic-time) backtracking algorithm [2,15,3,8,16] that uses a stack—and explicit pushes, pops, and a test for whether the popped state is a final state—to implement an explicit search for the most recent final state.

```
[1]  procedure Tokenize(M : DFA, input : string)
[2]  let ⟨Q, Σ, δ, q0, F⟩ = M in
[3]  begin
[4]
[5]
[6]
[7]    i := 1
[8]    loop
[9]      q := q0
[10]     push(Bottom)
         /* Scan for tokens */
[11]     while      i ≤ length(input)
[12]           and  δ(q, input[i]) is defined
[13]
[14]     do
[15]       push(q)
[16]       q := δ(q, input[i])
[17]       i := i + 1
[18]     od
         /* Backtrack to the most recent final state */
[19]     while q ∉ F do
[20]
[21]       q := pop()
[22]       i := i − 1
[23]       if q = Bottom then
[24]         return "Failure: tokenization not possible"
[25]       fi
[26]     od
[27]     print(i − 1)
[28]     if i > length(input) then
[29]       return "Success"
[30]     fi
[31]     reset the stack to empty
[32]   pool
[33] end
```

(a)

```
[1]  procedure Tokenize(M : DFA, input : string)
[2]  let ⟨Q, Σ, δ, q0, F⟩ = M in
[3]  begin
[4]    for each q ∈ Q and i ∈ [1..length(input)] do
[5]      failed_previously[q,i] := false
[6]    od
[7]    i := 1
[8]    loop
[9]      q := q0
[10]     push(Bottom)
         /* Scan for tokens */
[11]     while      i ≤ length(input)
[12]           and  δ(q, input[i]) is defined
[13]           and  ¬ failed_previously[q,i]
[14]     do
[15]       push(q)
[16]       q := δ(q, input[i])
[17]       i := i + 1
[18]     od
         /* Backtrack to the most recent final state */
[19]     while q ∉ F do
[20]       failed_previously[q,i] := true
[21]       q := pop()
[22]       i := i − 1
[23]       if q = Bottom then
[24]         return "Failure: tokenization not possible"
[25]       fi
[26]     od
[27]     print(i − 1)
[28]     if i > length(input) then
[29]       return "Success"
[30]     fi
[31]     reset the stack to empty
[32]   pool
[33] end
```

(b)

Figure 1. (a) A tokenization algorithm that employs a stack of encountered states to implement backtracking when either DFA *M* is unable to make any transition, or is not in a final state when the end of the input is reached, (b) A modified tokenization algorithm that tabulates which pairs of states and index positions that were previously encountered failed to lead to the identification of a longer token. This information is used to avoid repeating fruitless searches of the input text.

Like the tokenizing WORM-2DPDA, Figure 1(a) implements backtracking by stacking states and, if DFA *M* is unable to make any transition, or if *M* is not in a final state when the end of the input is reached, it pops elements from the stack (into variable *q*) and backs up (by decrementing variable *i*) until either a final state is found or the symbol *Bottom* is found. The latter condition implies that no left-to-right maximal-munch tokenization is possible.

On first consideration, the backtracking loop that searches for the most recent final state in Figure 1(a) (lines [19]–[26]) seems like extra work. That is, compared with the approach of just maintaining a record of the last final state and the last final-state position during the simulation of *M*, the use of a stack and an explicit search for the most recent final state seems to be overkill.[3]

[3]Although there is no impact from the standpoint of asymptotic worst-case complexity: The overall worst-case cost of tokenization by backtracking to the most recent final state and restarting is quadratic no matter which of these two backtracking methods is used.

However, there is method to our madness: The linear-time solution to the tokenization problem, which is shown in Figure 1(b), is obtained merely by adding five lines to Figure 1(a). The added code (lines [4]–[6], [13], and [20], which are indicated in Helvetica-Bold typeface in Figure 1(b)) tabulates which pairs of states and index positions that were previously encountered failed to lead to the identification of a longer token. This information is gathered at the time states are popped off the stack (line [20]), and used during the scanning loop to determine whether the current configuration is known to be unproductive (line [13]).

More precisely, the algorithm shown in Figure 1(b) carries out the same process as Figure 1(a), except that it uses a two-dimensional table, *failed_previously*$[q,i]$, to tabulate previously encountered pairs of states and index positions that failed to lead to the identification of a longer token. By consulting this table on line [13] of the scanning loop (lines [11]–[18]) the algorithm keeps track of—and avoids repeating—fruitless searches of the input text. Because the algorithm repeatedly identifies the last character of the *longest* prefix of the remaining input that is a token, a pair $\langle q,i\rangle$ for which *failed_previously*$[q,i]$ is true on line [13] must represent a *failed* previous search that started from position i in state q. Hence, it would be unproductive to make another search from position i in state q. For this reason, the algorithm exits the scanning loop and switches to backtrack mode.

Because scanning proceeds left to right, processing one character at a time, the scanning loop (lines [11]–[18]) can encounter a given pair $\langle q,i\rangle$ at most $|Q|$ times: In the worst case, it can reach $\langle q,i\rangle$ exactly once from each of the configurations $\langle q', i-1\rangle$, $q' \in Q$. Because for an input of length n there are only $O(n)$ pairs of the form $\langle q,i\rangle$, the algorithm's running time is $O(n)$ (where the constant of proportionality depends on $|Q|$).

The effect produced by the five lines added in Figure 1(b) is very similar to the effect produced by Mogensen's WORM-2DPDA simulation technique on the tokenizing WORM-2DPDA of Section 2. The overall structure of both Figure 1(a) and Figure 1(b) is very similar to the tokenizing WORM-2DPDA; the loops on lines [11]–[18] and [19]–[26] implement separate phases that are of the form "move the input head right, simulating DFA *M* " (forward mode) and "move the input head left, searching for a final state of *M* " (backtrack mode), respectively. The test on line [13] of Figure 1(b) represents a new condition under which the machine shifts from forward mode to backtrack mode. The information tabulated in table *failed_previously* allows the algorithm shown in Figure 1(b) to take "shortcuts" that proceed directly to a configuration further along in the computation, thereby skipping the execution of certain computation sequences that would otherwise be just repetitions of unproductive computation sequences performed previously.

Example. It is instructive to consider the behavior of Figure 1(b) on the example given in the Introduction, which caused the standard backtracking algorithm to exhibit quadratic behavior. In particular, let us assume that we are working with the DFA whose state-transition diagram is shown in Figure 2. This DFA recognizes the language $abc + (abc)^* d$.

Suppose the algorithm is invoked on the input string $(abc)^4 = abcabcabcabc$. The first invocation of the loop on lines [11]–[18] will advance to the end of the input, looking for—and failing to find—an instance of the token "$(abc)^* d$". However, during this process, the states of the DFA are stacked so that as the loop on lines [19]–[26] repeatedly decrements variable i, entries are made in table *failed_previously* for the following pairs of states and index positions:

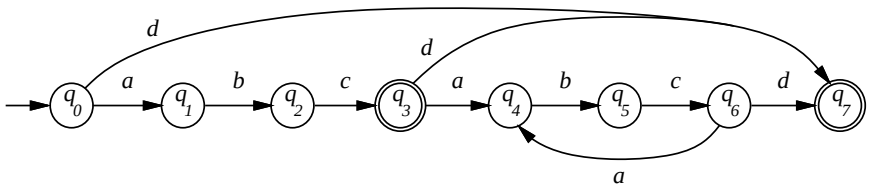

Figure 2. State-transition diagram for a finite-state automaton that recognizes the language $abc + (abc)^* d$.

$\langle q_6, 13\rangle$	$\langle q_5, 12\rangle$	$\langle q_4, 11\rangle$
$\langle q_6, 10\rangle$	$\langle q_5, 9\rangle$	$\langle q_4, 8\rangle$
$\langle q_6, 7\rangle$	$\langle q_5, 6\rangle$	$\langle q_4, 5\rangle$

The loop continues until the symbol q_3 is popped off the stack and i receives the value 4. At this point, because q_3 is a final state of the DFA, the program exits the loop on lines [19]–[26], leaving i set to 4; the first instance of *abc* is reported as the first token; the stack is cleared; q is set to the initial state q_0; *Bottom* is pushed on the stack; and scanning is resumed (at position 4 in state q_0).

In the second invocation of the scanning loop, after the characters *abca* in positions 4–7 have been processed, variable i attains the value 8—two positions beyond the end of the second *abc* token—whereupon the program encounters a configuration that previously failed to lead to the identification of a longer token, namely $\langle q_4, 8\rangle$ and the program exits the scanning loop. This time, just one iteration of the loop on lines [19]–[26] is performed. This pops the symbol q_3 off the stack, which causes the loop to terminate because q_3 is a final state of the DFA, and leaves i set to 7. The second instance of *abc* is reported as the second token.

The same pattern of action is repeated for the third instance of *abc*: Scanning is resumed in state q_0 at position 7. Variable i attains the value 11—two positions beyond the end of the third *abc* token—whereupon the program reaches a configuration that previously failed to lead to the identification of a longer token, namely $\langle q_4, 11\rangle$. Again, the algorithm exits the scanning loop, performs one iteration of the loop on lines [19]–[26]—popping off an instance of final state q_3—and reports the third instance of *abc* as the third token.

The fourth instance of *abc* is identified as a token because q is in a final state (*i.e.*, q_3) when the end of the input is reached.

In general, the same processing pattern carries over to all inputs of the form $(abc)^m$: After the first scanning/backtracking pass over the input string, which results in the recognition of the first instance of *abc* as a token, all subsequent passes except the last one will attain a configuration that previously failed to lead to the identification of a longer token, where variable i is positioned two characters beyond the end of the next instance of *abc*. (On pass j, for $2 \le j < m$, the configuration is of the form $\langle q_4, 3j+2\rangle$.) At this point, the algorithm exits the loop on lines [11]–[18], performs one iteration of the loop on lines [19]–[26]—popping off an instance of final state q_3—and reports the instance of *abc* that ends at position $3j$ as the j^{th} token.

Because a linear amount of work is done to recognize the first token, and thereafter only a constant amount of work is done per token recognized, the algorithm performs $\Theta(m)$ $(= \Theta(n))$ steps to tokenize inputs of the form $(abc)^m$. □

Remark. For inputs of the form $(abc)^m$, after the first scanning/backtracking pass over the input string, table *failed_previously* contains entries for configurations of the form $\langle q_4, 3j+2\rangle$, $\langle q_5, 3j+3\rangle$, and $\langle q_6, 3j+4\rangle$, for $1 \le j < m$. The state-transition diagram shown in Figure 2 has the property that the state entered after reading a string of the form "$abc(abc)^p a$", for $p > 0$—namely state q_4—is the same as that entered after reading "*abca*", which allows the algorithm in Figure 1(b) to switch from forward mode to backtrack mode after variable *i* reaches a position just two characters past the next instance of token *abc*. With a different DFA for the language $abc + (abc)^* d$, a similar effect still occurs: The program switches modes whenever it enters a previously encountered configuration that failed to lead to the identification of a longer token; however, the point at which a previously encountered configuration is reached takes place some number of characters further to the right in the input string. (This number is a constant whose value depends on which DFA for the language $abc + (abc)^* d$ is being used. The total cost of tokenization is still linear in the length of the input string.) □

4. CONCLUDING REMARKS

Although most programming languages do have the property that there is a token class for which some of the tokens are prefixes of tokens in another token class (*e.g.*, integer and floating-point constants), the potential quadratic behavior of lexical analyzers is probably not a problem in practice. However, lexical-analysis tools such as Lex are often used for tasks outside the domain of compilation. For example, Aho and Ullman mention the use of Lex to recognize imperfections in printed circuits [2]. Some of these non-standard applications may represent situations in which the algorithms presented in this paper could be of importance. The lexical analysis of spoken natural language might be another such application.

Some readers may find it unsettling that the solution to the maximal-munch tokenization problem depends on—or at least was motivated by—properties of 2DPDAs, which, after all, are not only more powerful than DFAs, but also more powerful than the DPDAs used in the parsing phase of syntax analysis.[4] As noted in the Introduction, the problem would not necessarily go away merely by folding lexical analysis back into parsing. Although this would allow the tokenization problem to be tackled by the more powerful formalisms used to implement parsing, such as LR grammars and DPDAs, this by itself does not address the maximal-munch requirement of the tokenization problem. Even if the token classes were specified with an LR grammar, say, we would still have the problem of designing an efficient automaton that identifies the *longest* prefix of the input that represents a token. Thinking about WORM-2DPDAs provided the insight that led to the solutions described in Sections 2 and 3 not only because WORM-2DPDAs permit backtrack searching to be carried out efficiently (when the automaton is simulated on a RAM), but also because they provide a mechanism for recording the endpoints of the tokens that are identified, and hence have a better "fit" for a problem that involves repeated identification of longest prefixes.

After presenting the quadratic-time backtracking algorithm for the tokenization problem, Waite and Goos advance the following conjecture:

> We have tacitly assumed that the initial state of the automaton is independent of the final state reached by the previous invocation of *next_token* [*i.e.*, the tokenizer]. If this assumption is re-

[4] It is easy to construct a 2DPDA that recognizes the language $a^n b^n c^n$, which is not context-free.

> laxed, permitting the state to be retained from the last invocation, then it is sometimes possible to avoid even the limited backtracking discussed above . . . Whether this technique solves all problems is still an open question [15, pp. 138–139].

The solutions given in the present paper are based on a different principle; rather than "permitting the state to be retained from the last invocation", they rely on tabulation to avoid repeating work. However, there is a sense in which Waite and Goos are nearly on the mark: If the notion of "state" is broadened to include the memoization table (*i.e.*, the table *failed_previously*) in addition to the initial state of the automaton, it is correct to say that the solution to the problem does involve the retention of state from the last invocation of the tokenizer.

While the algorithm presented in Section 2 is probably only of academic interest (at least as far as the design of compilers is concerned), it does serve as another example of an interesting algorithm-design methodology, namely:

> It may be easier to write a 2DPDA program (which can either be simulated in linear time or compiled to run in linear time) than directly writing the program that runs in linear time. Therefore, design a (WORM)-2DPDA to solve the problem of interest, and then apply the Cook, Jones, Andersen and Jones, or Mogensen constructions to obtain a linear-time algorithm for the problem.

For the maximal-munch tokenization problem, once the proper insights had been obtained, it was relatively easy to write the linear-time algorithm directly (see Figure 1(b)). However, the thought process of the author was influenced by the way in which dynamic programming is used in the Cook, Jones, Andersen and Jones, and Mogensen constructions. Thus, Section 3 illustrates an alternative algorithm-design methodology, namely:

> Design a (WORM)-2DPDA for a language-recognition problem that is a *near relative* of the problem of interest; study where the Cook, Jones, Andersen and Jones, or Mogensen constructions are able to introduce shortcuts; and use these insights to obtain a linear-time algorithm for the problem of interest.

As discussed in [11,1,4], the Knuth-Morris-Pratt linear-time algorithm for string matching was also developed in this fashion.

ACKNOWLEDGEMENTS

Mooly Sagiv introduced me to the problem, and showed me the example that a student of his, Ram Nathaniel, had come up with to illustrate the potential quadratic behavior of the standard backtracking algorithm. (The example is essentially the one discussed in the Introduction and in Section 3.) I am grateful for further discussions that I had about the problem with Sagiv, as well as with Charles Fischer and Reinhard Wilhelm.

REFERENCES

1. Aho, A.V., Hopcroft, J.E., and Ullman, J.D., *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, MA (1974).
2. Aho, A.V. and Ullman, J.D., *Principles of Compiler Design,* Addison-Wesley, Reading, MA (1977).
3. Aho, A.V., Sethi, R., and Ullman, J.D., *Compilers: Principles, Techniques, and Tools,* Addison-Wesley, Reading, MA (1986).
4. Aho, A.V., "Algorithms for finding patterns in strings," pp. 255-300 in *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity,* ed. J. van Leeuwen,The M.I.T. Press, Cambridge, MA (1990).
5. Andersen, N. and Jones, N.D, "Generalizing Cook's transformation to imperative stack programs," pp. 1-18 in *Results and Trends in Theoretical Computer Science, Lecture Notes in Computer Science,* Vol. 812, ed. J. Karhumäki, H. Maurer, and G. Rozenberg,Springer-Verlag, New York, NY (1994).
6. Cook, S.A., "Linear time simulation of deterministic two-way pushdown automata," pp. 172-179 in *Information Processing 71: Proceedings of the IFIP Congress 71,* ed. C.V. Freiman,North-Holland, Amsterdam (1972).

7. DeRemer, F.L., “Lexical analysis,” pp. 109-120 in *Compiler Construction: An Advanced Course*, ed. F.L. Bauer and J. Eickel,Springer-Verlag, New York, NY (1974).
8. Fischer, C.N. and LeBlanc, R.J., *Crafting a Compiler*, Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA (1988).
9. Hopcroft, J.E. and Ullman, J.D., *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA (1979).
10. Jones, N.D., “A note on linear time simulation of deterministic two-way pushdown automata,” *Information Processing Letters* **6**(4) pp. 110-112 (August 1977).
11. Knuth, D.E., Morris, J.H., and Pratt, V.R., “Fast pattern matching in strings,” *SIAM J. Computing* **6**(2) pp. 323-350 (1977).
12. Lesk, M.E., “Lex – A lexical analyzer generator,” Comp. Sci. Tech. Rep. 39, AT&T Bell Laboratories, Murray Hill, NJ (October 1975).
13. Mehlhorn, K., *Data Structures and Algorithms 1: Sorting and Searching*, Springer-Verlag, Berlin (1984).
14. Mogensen, T., “WORM-2DPDAs: An extension to 2DPDAs that can be simulated in linear time,” *Information Processing Letters* **52** pp. 15-22 (1994).
15. Waite, W.M. and Goos, G., *Compiler Construction*, Springer-Verlag, New York, NY (1983).
16. Wilhelm, R. and Maurer, D., *Compiler Design (English Edition)*, Addison-Wesley, Reading, MA (1995).