“Maximal-Munch” Tokenization in Linear Time

THOMAS REPS
University of Wisconsin

The lexical-analysis (or scanning) phase of a compiler attempts to partition an input string 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. Although most of the standard compiler textbooks present a way to perform maximal-munch tokenization, the algorithm they describe is one that, for certain sets of token definitions, can cause the scanner to exhibit quadratic behavior in the worst case. In this paper, we show that maximal-munch tokenization can always be performed in time linear in the size of the input.

CR Categories and Subject Descriptors: 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.5.4 [**Pattern Recognition**]: Applications – *text processing*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: backtracking, dynamic programming, memoization, tabulation, tokenization

1. INTRODUCTION

The lexical-analysis (or scanning) phase of a compiler attempts to partition an input string 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”, “.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

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.

$L(R_1 + R_2 + \cdots + R_k)$. 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 [2,12,3,7,13], discuss this issue briefly. For example, Aho and Ullman's 1977 book discusses the issue in the context of a lexical analyzer based on deterministic finite-state automata (DFAs) [2, pp. 109–110]:

> 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.

The discussion of the problem in the 1986 book by Aho, Sethi, and Ullman is similar, except that they assume that lexical analysis is performed by simulating an NFA, rather than first converting an NFA to a DFA [3, pp. 104]. Other similar discussions are given by Waite and Goos [12], Fischer and LeBlanc [7], and Wilhelm and Maurer [13].

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.*, $(abc)^m$).[2] 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

---

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

[2] The simplest example that exhibits quadratic behavior involves the token classes a and a^*b, with an input string of the form a^m. However, in this case each individual character of the input string represents a separate token. Therefore, we use an example that does not exhibit this degeneracy.

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. 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].

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.

Note that the division of syntax analysis into separate phases of lexical analysis and parsing is not the source of our difficulties: Even if the token classes were specified with, say, an LR grammar, 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 DFA 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:

(i) In Section 2, we give a program that uses tabulation (or "memoization") to avoid repeating fruitless searches of the input string. The principle underlying the program is quite simple:

> Whenever a scanner processes a character at some position in the input string, it is in some state of the DFA. In the course of tokenizing the input, a conventional scanner may pass over the same character position several times (because of backtracking), moving to the right in each case; all passes except the last lead to failure. Because processing is deterministic, there is never a need to repeat a transition that is doomed to failure. Therefore, to avoid repeating a search that cannot possibly bear fruit, the tabulating scanner keeps track of the pairs of states and index positions encountered that have failed to lead to the identification of a longer token.

Because the number of states is a constant, there is a constant upper bound on the number of times the tabulating scanner makes a transition at each character position, and thus it performs maximal-munch tokenization in linear time.

(ii) Section 3 discusses ways to reduce the amount of storage used by the scanner presented in Section 2. The technique described in Section 3 also provides a way to detect many of the cases in which an unbounded amount of lookahead beyond final states is never necessary.

Section 4 presents a few concluding remarks.

The remainder of the paper relies on the following assumptions and notational conventions:

- 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). (The next higher level of the compiler is assumed to filter out the whitespace tokens. The latter phase is sometimes called *screening* [6,13].)

- The symbol n is used to denote the length of the input string.
- All issues related to buffering the input are ignored.
- 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 $L(M) = L(R_1 + R_2 + \cdots + R_k)$.
- Standard notation for DFAs is used (*e.g.*, see [8]). 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. A TABULATING SCANNER

This section presents an algorithm that performs maximal-munch tokenization in linear time. The easiest way to understand the linear-time algorithm is to first consider a version of the algorithm that does not run in linear time, namely the program presented in Figure 1(a). This program prints out the positions in the input string that are the final characters of each token. The algorithm can be viewed as a variant of the standard (quadratic-time) backtracking algorithm [2,12,3,7,13] 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.

The version of procedure *Tokenize* given in Figure 1(a) implements backtracking by stacking pairs of the form ⟨state, position⟩, and, if DFA M is unable to make a transition, or if M is not in a final state when the end of the input is reached, *Tokenize* repeatedly pops elements from the stack into variables q and i. By this means, it backs up 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 [20]–[26]) seems like extra work. A maximal-munch tokenization algorithm need not use a stack to "review the states of the DFA which we have entered while processing the input" [2]: 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. Compared with the latter approach, the use of a stack and an explicit search for the stack entry for most recent final state seems to be overkill.[3]

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], [14], and [21], which are indicated in a contrasting 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 pairs are popped off the stack (line [21]), and used during the scan-

[3]However, there is no impact from the standpoint of asymptotic worst-case complexity: The worst-case running time is quadratic no matter which of these two backtracking methods is used.

```
[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, i⟩)
         /* Scan for tokens */
[11]     while      i ≤ length(input)
[12]           and  δ(q, input[i]) is defined
[13]     do
[14]
[15]       if q ∈ F then reset the stack to empty fi
[16]       push(⟨q, i⟩)
[17]       q := δ(q, input[i])
[18]       i := i + 1
[19]     od
         /* Backtrack to the most recent final state */
[20]     while q ∉ F do
[21]
[22]       ⟨q, i⟩ := pop()
[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]   pool
[32] 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)+1] do
[5]      failed_previously[q,i] := false
[6]    od
[7]    i := 1
[8]    loop
[9]      q := q0
[10]     push(⟨Bottom, i⟩)
         /* Scan for tokens */
[11]     while      i ≤ length(input)
[12]           and  δ(q, input[i]) is defined
[13]     do
[14]       if failed_previously[q,i] then break fi
[15]       if q ∈ F then reset the stack to empty fi
[16]       push(⟨q, i⟩)
[17]       q := δ(q, input[i])
[18]       i := i + 1
[19]     od
         /* Backtrack to the most recent final state */
[20]     while q ∉ F do
[21]       failed_previously[q,i] := true
[22]       ⟨q, i⟩ := pop()
[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]   pool
[32] end
```

(b)

Figure 1. (a) A tokenization algorithm that employs a stack of ⟨state, position⟩ pairs to implement backtracking. Backtracking occurs when either (i) DFA M is unable to make a transition, or (ii) M 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. (Algorithm (b) is obtained by adding five lines to algorithm (a). These additions are indicated in Helvetica-Bold typeface.)

ning loop to determine whether the current configuration is known to be unproductive (line [14]).

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 [14] of the scanning loop (lines [11]–[19]) 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 [14] 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 backtracking mode.

Because scanning proceeds left to right, processing one character at a time, the scanning loop (lines [11]–[19]) 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 $|Q|$).

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]–[19] 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, pairs of the form ⟨state, position⟩ are stacked so that as the loop on lines [20]–[26] pops the stack, entries are made in table *failed_previously* for the following pairs of states and index positions:

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

The loop continues until the pair $\langle q_3, 4\rangle$ is popped off of the stack. At this point, because q_3 is a final state of the DFA, the program exits the loop on lines [20]–[26], leaving i set to 4; the first instance of *abc* is reported as the first token; q is set to the initial state q_0; *Bottom* is pushed on the now-empty stack; and scanning is resumed (at position 4 in state q_0).

In the second invocation of the scanning loop, after the characters *abca* at 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 [20]–[26] is performed. This pops the pair $\langle q_3, 7\rangle$ 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 [20]–[26]—popping off the pair $\langle q_3, 10\rangle$—and reports the third instance of *abc* as the third token.

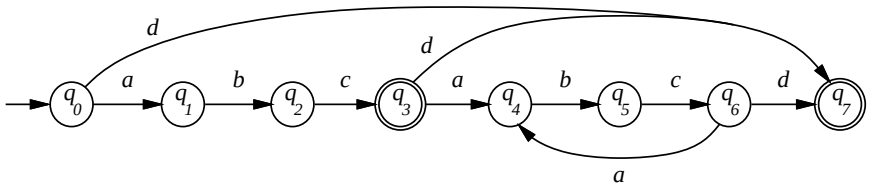

Figure 2. State-transition diagram for a finite-state automaton that recognizes the language $abc + (abc)^* d$.

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, 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 $p > 0$ is q_4, the same as that entered after reading "*abca*". Thus, all subsequent passes, except the very last one, will attain a configuration that previously failed to lead to the identification of a longer token: Variable i will be positioned two characters beyond the end of the next instance of *abc*, and the state will be q_4. That is, 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]–[19], performs one iteration of the loop on lines [20]–[26]—popping off the pair $\langle q_3, 3j+1\rangle$—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 $(abc)^m$.

With a different DFA for the language $abc + (abc)^* d$, a similar effect would still occur: The program would switch from scanning mode to backtracking mode 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 could take 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.

3. OPTIMIZING THE TABULATING SCANNER

A straightforward implementation of the tabulating scanner from Figure 1(b) uses a table of $|Q| \times (n+1)$ bits, and $\Omega(|Q| \times (n+1))$ time for initialization.[4] Therefore, in this section we describe several optimization techniques aimed at reducing storage utilization.

A second concern is that the languages used for lexical analysis in most compilers are designed so that an unbounded amount of lookahead beyond final states is never necessary.[5] Consequently, we would like to have a mechanism that entirely avoids the overheads of the tabulating scanner in cases in which tabulation is not warranted. To address this concern, we present an implementation of the tabulating scanner that imposes no overhead in many cases that cannot produce superlinear behavior.

One strategy for reducing storage utilization is based on the idea that tabulation does not have to be carried out at every index position. Instead, the scanner can tabulate only at every k^{th} position, where k is some constant. Initialization time and space fall by a factor of k. Running time could increase by a factor of at most k^2, but the overall running time is still linear in n. In particular, if k is $|Q|$, the space used is $n+1$ bits—*i.e.*, a quantity independent of the number of states—and the running time is

[4]By using a technique described by Aho, Hopcroft, and Ullman (see [1, Problem 2.12]), we can sidestep the need to initialize *failed_previously*; however, this increases the space used from $|Q| \times (n+1)$ bits to $\Omega(|Q| \times (n+1))$ pointers.

[5]The DO statement in Fortran requires unbounded lookahead, but Fortran does not use the maximal-munch convention.

$O(|Q|^2 \times n) = O(n)$).

Another strategy for optimization (which can be used in conjunction with the previous one) is based on the idea of not tabulating certain states in Q (at *any* index position of the input string). That is, *Tokenize* will only maintain tabulation information for states that are members of some set $Tab \subseteq Q$. This allows table *failed_previously* to be of size $|Tab| \times (n+1)$ bits, and allows a non-tabulating scanner to be substituted when $Tab = \emptyset$.

Let us postpone for the moment the question of how to identify a suitable set *Tab*. The version of the tabulating scanner given in Figure 3(b) makes concrete the idea of what it means to maintain tabulation information only for the states in *Tab*. In Figure 3(b), four lines are changed from Figure 1(a) (which is repeated as Figure 3(a) for the reader's convenience). As before, changes in the code that appears on the right-hand side are indicated in a contrasting typeface.

```
[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)+1] do
[5]      failed_previously[q,i] := false
[6]    od
[7]    i := 1
[8]    loop
[9]      q := q0
[10]     push(⟨Bottom, i⟩)
         /* Scan for tokens */
[11]     while      i ≤ length(input)
[12]           and  δ(q, input[i]) is defined
[13]     do
[14]       if failed_previously[q,i] then break fi
[15]       if q ∈ F then reset the stack to empty fi
[16]       push(⟨q, i⟩)
[17]       q := δ(q, input[i])
[18]       i := i + 1
[19]     od
         /* Backtrack to the most recent final state */
[20]     while q ∉ F do
[21]       failed_previously[q,i] := true
[22]       ⟨q, i⟩ := pop()
[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]   pool
[32] end
```

(a)

```
[1]  procedure Tokenize(M : DFA, input : string)
[2]  let ⟨Q, Σ, δ, q0, F⟩ = M in
[3]  begin
[4]    for each q ∈ Tab and i ∈ [1..length(input)+1] do
[5]      failed_previously[q,i] := false
[6]    od
[7]    i := 1
[8]    loop
[9]      q := q0
[10]     push(⟨Bottom, i⟩)
         /* Scan for tokens */
[11]     while      i ≤ length(input)
[12]           and  δ(q, input[i]) is defined
[13]     do
[14]       if q ∈ Tab cand failed_previously[q,i] then break fi
[15]       if q ∈ F then reset the stack to empty fi
[16]       if q ∈ Tab ∪ F then push(⟨q, i⟩) fi
[17]       q := δ(q, input[i])
[18]       i := i + 1
[19]     od
         /* Backtrack to the most recent final state */
[20]     while q ∉ F do
[21]       if q ∈ Tab then failed_previously[q,i] := true fi
[22]       ⟨q, i⟩ := pop()
[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]   pool
[32] end
```

(b)

Figure 3. (a) The tabulating tokenization algorithm from Figure 1(b). (b) A tabulating tokenization algorithm that only tabulates states in *Tab*. (Algorithm (b) is obtained by changing four lines of algorithm (a). These modifications are indicated in Helvetica-Bold typeface.)

The ideas behind the changes are as follows:

(i) *failed_previously* needs to contain entries only for the members of *Tab*. These are initialized in the loop on lines [4]–[6].
(ii) *failed_previously* is only accessed for states that are members of *Tab* (see lines [14] and [21]).
(iii) The stack only ever contains entries for *Bottom* (line [10]), final states (line [16]), and states that are members of *Tab* (line [16]).

Note that if $Tab = \emptyset$, only a single pair ever appears on the stack—*i.e.*, one that is either of the form ⟨*Bottom*, position⟩ or ⟨final-state, position⟩. In essence, Figure 3(b) degenerates in this case to a variant of the algorithm that just maintains a record of the last final state and the last final-state position during the simulation of *M*. When $Tab = \emptyset$, the code can be cleaned up by (i) replacing the stack with an explicit pair of variables, (ii) eliminating array *failed_previously* entirely, (iii) removing the dead code that is guarded by the tests $q \in Tab$ in lines [14] and [16], and (iv) simplifying the condition in line [16] from $q \in Tab \cup F$ to $q \in F$.

Our goal is now to find suitable *Tab* sets, in particular, ones for which the version of *Tokenize* shown in Figure 3(b) will always have linear-time behavior. Our first cut at defining *Tab* is $Tab = Q - F$: Because *Tokenize* never backtracks over a ⟨final-state, position⟩ configuration, there is no need to reserve space in *failed_previously* to tabulate final states. However, when *Tab* is $Q - F$, there are no essential differences in behavior between Figures 3(a) and 3(b).

To identify a nontrivial *Tab* set, it is necessary to analyze the finite-state control of DFA *M*. For this purpose, it is convenient to think of *M*'s transition function δ as defining a labeled directed graph (or state-transition diagram) in the usual way: The nodes are the states Q; each transition $\delta(q, a) = q'$ yields an edge $q \rightarrow^a q'$, labeled with a. In addition, however, we assume that the graph is augmented with an explicit failure node, q_{fail}, which represents a new non-final state, and that the graph is normalized as follows:

(i) Nodes (states) from which there is no path to a final-state node are said to be *useless*. All useless nodes are condensed to q_{fail}. That is, if node m is useless, edges of the form $m \rightarrow^a q'$ and $q \rightarrow^b m$ are replaced by edges of the form $q_{fail} \rightarrow^a q'$ and $q \rightarrow^b q_{fail}$, respectively.
(ii) The graph is made into a "total representation" of δ: An edge of the form $q \rightarrow^c q_{fail}$ is added to the graph for each undefined transition $\delta(q, c)$.
(iii) q_{fail} is made into a sink node: All edges of the form $q_{fail} \rightarrow^a q_{fail}$ are removed from the graph.

Figure 4 shows the augmented version of the state-transition diagram from Figure 2.

Because *Tokenize* never backtracks over a ⟨final-state, position⟩ configuration, *Tokenize* should only tabulate states that are reachable from final states. We say that *ReachableFromFinal*(q) holds iff there is a (possibly empty) path from a final state to q. This is a simple reachability question: We can determine the set of q's for which *ReachableFromFinal*(q) holds either using depth-first search starting from the members of *F*, or, equivalently, by finding the least solution to the following set of equations over the nodes of *M*'s graph (under the ordering $false \sqsubset true$):

$$ReachableFromFinal(q) = \begin{cases} true & \text{if } q \in F \\ \bigvee_{p \rightarrow q} ReachableFromFinal(p) & \text{otherwise} \end{cases}$$

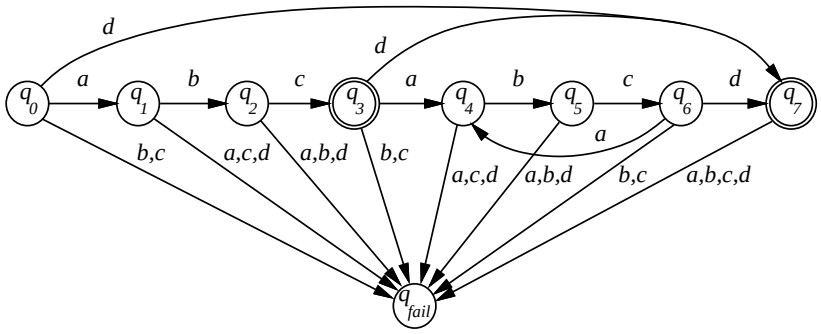

Figure 4. State-transition diagram from Figure 2, augmented with node q_{fail}. (Arrows labeled with several letters are a shorthand for multiple edges.)

The least solution to these equations can be obtained by iteration, starting from an initial approximation to the solution in which, for each node q, $ReachableFromFinal(q) = false$. Define the set *ReachableFromFinal* as $ReachableFromFinal = \{ q \mid ReachableFromFinal(q) \}$. Thus, our second cut at defining *Tab* is $Tab = ReachableFromFinal - F$.

We are not done yet, for we can further narrow the *Tab* set and still have *Tokenize* retain linear-time behavior. Let *Bounded* be the set of states such that either $q \in F$, or q has the property that when *Tokenize* continues from q it always either

(i) reaches a final state within a bounded number of steps, or
(ii) fails within a bounded number of steps.

As observed earlier, *Tokenize* does not need to maintain tabulation information for final states. The reason *Tokenize* does not need to maintain tabulation information for the non-final states in *Bounded* is that the total amount of work performed by *Tokenize* in processing such states is bounded by $O(n)$. To see this, consider the scanning loop (lines [11]–[19]) of Figure 3(b). Once *Tokenize* reaches one of the non-final states in *Bounded*, charge all work performed by the scanning loop, until either a final state is reached or failure occurs, to position $i - 1$, where i is the value that appears in the pair $\langle q, i \rangle$ at the *bottom* of the stack. Note that q is either *Bottom*, if $\langle q, i \rangle$ came from line [10], or a final state if $\langle q, i \rangle$ came from line [16] (see also line [15]). By properties (i) and (ii) above, *Tokenize* can only consume a bounded amount of input once it reaches one of the non-final states in *Bounded*. Because *Tokenize* never backtracks over a ⟨final-state, position⟩ configuration, the above accounting scheme charges at most a constant amount of work to each index position, and thus the total cost of processing the non-final states in *Bounded* is $O(n)$.

Consequently, by choosing *Tab* to be *ReachableFromFinal* − *Bounded*, we are assured that the version of *Tokenize* shown in Figure 3(b) will always have linear-time behavior.

The presence of cyclic, accepting-state-free paths in *M*'s graph are the only possible cause of unbounded behavior. Thus, we can recast the definition of *Bounded* as follows:

$q \in Bounded$
iff
($q \in (F \cup \{ q_{fail} \})$ or all accepting-state-free paths from q to nodes in $F \cup \{ q_{fail} \}$ are acyclic)

One way to identify the members of *Bounded* is to find the least solution to the following set of equations over the nodes of *M*'s graph (under the ordering $false \sqsubset true$) and let $Bounded = \{ q \mid Bounded(q) \}$:

$$Bounded(q) = \begin{cases} true & \text{if } q \in (F \cup \{ q_{fail} \}) \\ \bigwedge_{q \to r} Bounded(r) & \text{otherwise} \end{cases}$$

The least solution to these equations can be obtained by iteration, starting from an initial approximation to the solution in which, for each node q, $Bounded(q) = false$. (By finding the *least* solution, we pessimistically assume, until proven otherwise, that there exists a cyclic, accepting-state-free path from q to a member of $F \cup \{ q_{fail} \}$.)

For instance, for the graph shown in Figure 4 we have

state	*ReachableFromFinal*	*Bounded*
q_0	*false*	*true*
q_1	*false*	*true*
q_2	*false*	*true*
q_3	*true*	*true*
q_4	*true*	*false*
q_5	*true*	*false*
q_6	*true*	*false*
q_7	*true*	*true*
q_{fail}	*true*	*true*

Thus, in this example, we need to tabulate only the states $Tab = ReachableFromFinal - Bounded = \{ q_4, q_5, q_6 \}$.

Because the languages used for lexical analysis in most compilers are designed so that an unbounded amount of lookahead beyond final states is never necessary, it is evident from the discussion above that, in these cases, the algorithm described above will set *Tab* to ∅. If a (tabulating-)scanner generator were extended with this method, it could test each input lexical specification to see whether $Tab = \emptyset$; when this occurs, it could supply a scanner that just maintains a record of the last final state and the last final-state position during the simulation of *M*. Thus, the common case would be that tabulation would never come into play, and would therefore impose no overhead (except at scanner-generation time to verify that $Tab = \emptyset$). However, if the scanner generator were ever applied in a non-standard situation involving a set of highly ambiguous token definitions, a tabulating scanner along the lines of Figure 3(b) could be supplied to prevent superlinear behavior from occurring. In this case, the analysis algorithm described above serves to reduce the size of tabulation table *failed_previously* from $|Q| \times (n+1)$ bits to $|Tab| \times (n+1)$ bits (with a consequent reduction in initialization time, as well).

4. CONCLUDING REMARKS

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 relaxed, 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 [12, 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.

The author was motivated to develop the algorithms presented in the paper after observing that a certain result in automata theory, due to Mogensen [10] (which extends an earlier result by Cook [5,1]) implied that maximal-munch tokenization could be performed in linear time. Mogensen showed 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 an exponential number of steps). Mogensen's technique exploits the fact that no matter how many steps the WORM-2DPDA performs, there are only a linear number of possible "stack configurations" that the WORM-2DPDA can be in. The RAM program uses dynamic programming to tabulate information about sequences of transitions between these configurations: Whereas the WORM-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.

Given a DFA M that recognizes the tokens of a language, it is easy to construct a WORM-2DPDA M' that identifies maximal-munch tokens (see [11]). Once the proper insights had been obtained (*i.e.*, by considering the way in which tabulation is used in the Cook and Mogensen constructions) it was relatively easy to write a linear-time algorithm for the maximal-munch tokenization problem directly: The effect produced by the five lines added in Figure 1(b) is very similar to the effect produced by Mogensen's simulation technique on the WORM-2DPDA that identifies maximal-munch tokens. This process illustrates an interesting 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 Mogensen construction is able to introduce shortcuts by using previously tabulated information; and use these insights to obtain a linear-time algorithm for the problem of interest.

The well-known algorithm of Knuth, Morris, and Pratt for linear-time pattern matching in strings is another example of an algorithm that was developed in a similar fashion [9,1,4]. (A lengthier discussion of these ideas can be found in [11].)

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 almost certainly 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 nonstandard applications may represent situations in which the algorithms presented in this paper could be of importance.

ACKNOWLEDGEMENTS

M. Sagiv introduced me to the problem, and showed me the example that a student of his, R. 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 2.) I am grateful for

further discussions that I had about the problem with Sagiv, as well as with C. Fischer, S. Horwitz, and R. Wilhelm. In particular, Wilhelm offered several valuable suggestions that initiated the development of the material in Section 3.

The comments and suggestions of the referees contributed greatly to the present form of the paper.

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 Theor. Comp. Sci., Vol. A: Algorithms and Complexity*, ed. J. van Leeuwen,The M.I.T. Press, Cambridge, MA (1990).
5. Cook, S.A., "Linear time simulation of deterministic two-way pushdown automata," pp. 172-179 in *Information Processing 71: Proc. of the IFIP Congress 71*, ed. C.V. Freiman,North-Holland, Amsterdam (1972).
6. 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).
7. Fischer, C.N. and LeBlanc, R.J., *Crafting a Compiler*, Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA (1988).
8. Hopcroft, J.E. and Ullman, J.D., *Introduction to Automata Theory, Languages, and Computation,* Addison-Wesley, Reading, MA (1979).
9. Knuth, D.E., Morris, J.H., and Pratt, V.R., "Fast pattern matching in strings," *SIAM J. Computing* **6**(2) pp. 323-350 (1977).
10. Mogensen, T., "WORM-2DPDAs: An extension to 2DPDAs that can be simulated in linear time," *Inf. Proc. Let.* **52** pp. 15-22 (1994).
11. Reps, T., "'Maximal-munch' tokenization in linear time," TR-1347, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI (May 1997; revised August 1997).
12. Waite, W.M. and Goos, G., *Compiler Construction,* Springer-Verlag, New York, NY (1983).
13. Wilhelm, R. and Maurer, D., *Compiler Design (English Edition),* Addison-Wesley, Reading, MA (1995).