On the look-ahead problem in lexical analysis

WUU YANG

Computer and Information Science Department, National Chiao-Tung University, HsinChu, Taiwan, R.O.C.

Abstract. Modern programming languages use regular expressions to define valid tokens. Traditional lexical analyzers based on minimum deterministic finite automata for regular expressions cannot handle the look-ahead problem. The scanner writer needs to explicitly identify the look-ahead states and code the buffering and re-scanning operations by hand. We identify the class of finite look-ahead finite automata, which is general enough to include all finite automata of practical lexical analyzers. Finite look-ahead finite automata are then transformed into suffix finite automata. A new lexical analyzer makes use of the suffix finite automata to identify tokens. The new lexical analyzer solves the look-ahead problem in a table-driven approach and it can detect lexical errors at an earlier time than traditional lexical analyzers. The extra cost of the new lexical analyzers is the larger state transition table and three additional 1-dimensional tables. Incremental lexical analysis is also discussed.

1. Introduction

Modern programming languages use regular expressions to define valid tokens. A scanner (or lexical analyzer) is built from the finite automaton (FA) that corresponds to the set of regular expressions. In determining the end of a token, a scanner usually needs to examine an extra symbol from the input. This is called 1-symbol lookahead. In some cases, the scanner needs to look ahead multiple symbols. For instance, to scan 10..20 in Pascal and Ada, the scanner needs to look ahead two symbols (*i.e.*, the two ".") after the 10 [6]. A similar example in Fortran is "10.EQ.IX".

There are two difficulties when the scanner needs to look ahead symbols. First, the previewed symbols are usually handled by a buffering mechanism. The size of the buffer must be no less than the maximum number of look-ahead symbols required by the scanner. It is not straightforward to determine the max-

---

This work was supported in part by National Science Council, Taiwan, R.O.C. under grants NSC 83-0111-S-009-001-CL and NSC 84-2213-E-009-043.

The running title should read as "The look-ahead problem".

imum number of look-ahead symbols from the regular-expression specification of tokens. Second, the previewed symbols must be buffered and re-scanned when the next token is requested. For scanner generators that generate transition tables, rather than code, the buffering and re-scanning operations must be explicitly identified and hand-coded by the scanner writer. The minimum finite automaton underlying a scanner for a language similar to Ada contains 20 states and 86 non-error transitions [6]. It is a time-consuming task to analyze such an automaton by hand. Though some scanner generators may automatically generate code to solve the look-ahead problem, these generators work only in an *ad hoc* way. According to Paxson, the look-ahead problem (called backtracking in *flex*) is "messy and often may be an enormous amount of work for a complicated scanner" [16].

The look-ahead problem originates from the longest-match convention, which states that the scanner should find, starting from the current location of the input, a longest string satisfying a token definition. This convention can be translated into the state-transition behavior of the finite automata of the regular expressions. The first result of this paper is to propose a technique that determines the maximum number of look-ahead symbols by analyzing the finite automata. An application of the technique is to determine whether an unbounded buffer is required and, if not, the size of the buffer.

A traditional scanner makes state transitions according to a finite automaton. When the scanner fails to transit to a next state, it determines the end of the current token and puts the previewed symbols into a buffer. The finite automaton that underlies the traditional scanner is usually deterministic and minimum. The reason for choosing the minimum deterministic FA is to reduce the size of the transition table of the scanner. However, the minimum deterministic FA does not help the scanner to decide the end of current token and to process the previewed symbols. By using a different finite automaton, we can build new scanners that solve the look-ahead problem in a table-driven approach. The scanner writer is completely free from concerns of the look-ahead problem.

To solve the lookahead problem, we define the class of *finite-lookahead* finite automata (FFAs), which consists of the deterministic FAs (DFAs) that look ahead at most a finite number of symbols. Some languages may require infinite lookahead. For example, a hexadecimal number in Modula-2 [19] must end with the character "H" and a prefix of a hexadecimal number may look like an integer. This causes the problem of infinite lookahead. It is argued that real-world scanners should utilize only finite-lookahead FAs since it is difficult to handle buffers of potentially infinite size. Furthermore, an FA that may look ahead an infinite number of symbols needs $O(n^2)$ time on input of length n. By contrast, an FFA takes at most $O(n)$ time.

The minimum FFA is transformed into an equivalent *suffix FA* (suffix FAs will be defined in Section 3). New scanners are built from the suffix FAs. Due to the properties of the suffix FAs, the look-ahead problem is solved in a table-driven approach. The scanner writer no longer needs to write code to perform the buffering and re-scanning operations.

A traditional scanner needs to re-scan the previewed symbols. By contrast, the new scanner "remembers" the previewed symbols with the help of three additional tables. It is no longer necessary to re-scan the previewed symbols. With slight modification to the three tables, the new scanner can also handle the right-context problem at no extra cost.

The new scanner can detect lexical errors at an earlier time than a traditional scanner. The time when errors are detected might be even earlier than some preceding tokens are recognized. The extra cost of the scanners is the larger state-transition table and three additional 1-dimensional tables.

The purpose of incremental lexical analysis is to identify tokens in a slightly modified string, using as much information as possible from the tokens in the original string. The look-ahead problem also needs to be carefully considered in incremental lexical analysis. Incremental lexical analysis can be applied in incremental integrated programming environments [4].

In the next section, we identify the class of finite-lookahead finite automata. An algorithm is proposed that determines whether an FA belongs to that class. The class of suffix FAs is defined in the third section. A technique is presented that transforms a finite-lookahead FA into an equivalent suffix FA. Section 4 describes the the three tables used by the new scanner. Section 5 presents the new table-driven scanner. A weaker form of suffix FAs is defined in Section 6, which is as useful as the suffix FAs, but contains fewer states. Section 7 discusses incremental lexical analysis when the scanner may look ahead symbols. The last section summaries the paper and discusses related work.

2. The compute-lookahead algorithm

Given a set of regular expressions defining tokens for a programming language, an algorithm is presented in this section that determines the maximum number of look-ahead symbols required by the scanner. We assume that the scanner employs the longest-match convention to resolve conflicts.

We assume that the input is a string of symbols and that there is a finite vocabulary of distinct symbols. The vocabulary is augmented by a special end-of-file symbol that does not occur in the regular expressions. We use α, β, and γ to denote strings of symbols. The length of a string α, denoted by $|\alpha|$, is

the number of symbols in it. The notation $\alpha \subset \beta$ means α is a proper prefix of β. A cycle in a directed graph is a path that starts and ends at the same node. A *state* of a finite automaton and a *node* in the graph of the finite automaton will be used as synonyms in the following discussion. In a DFA, there is an initial state and one or more accepting states. The initial state may or may not be an accepting state. The initial state is pointed to by an arrow whose tail is labeled *start*; the accepting states are indicated by double circles. Without loss of generality, we may assume that every state of the finite automaton is reachable from the initial state and that every state can reach an accepting state. An example FA is shown in Figure 2. An *accepting-to-accepting path* is a path from an accepting state to an accepting state that does not pass through any accepting states, except the first and the last accepting states on the path. The first and the last accepting states on an accepting-to-accepting path need not be distinct.

Definition. A *finite-lookahead finite automaton* (FFA) is a deterministic finite automaton that looks ahead at most a finite number of input symbols when determining the end of a token.

Consider the example of 10..20 in Pascal or Ada again. A close look reveals that the reason why the scanner needs to look ahead two symbols after the 10 is because both the integer 10 and a decimal number such as 10.5 are valid tokens but 10. is not. This observation leads to the following theorem.

Theorem 1. *The maximum number of lookahead symbols required by the scanner = max (*$|\gamma| - |\alpha|$*), where* α *and* γ *are valid tokens,* $\alpha \subset \gamma$*, and for all* β *such that* $\alpha \subset \beta \subset \gamma$*,* β *is not a valid token.*

The algorithm in Figure 1 makes use of Theorem 1. The set of regular expressions may be transformed into a deterministic finite automaton [3, 6]. A valid token corresponds to a unique accepting state in the deterministic finite automaton, which is the terminating state of the finite automaton when the token is used as input. When $\alpha \subset \gamma$, the length difference $|\gamma| - |\alpha|$ is the length of one of the accepting-to-accepting paths from the accepting state corresponding to α to the accepting state corresponding to γ.

By Theorem 1, a DFA is an FFA if and only if no accepting-to-accepting paths contain cycles of non-accepting states. If an accepting-to-accepting path contains a cycle of non-accepting states, it is possible to find two tokens α and γ, one for the first accepting state and the other for the last accepting state on the accepting-to-accepting path, such that $\alpha \subset \gamma$ and for all β, $\alpha \subset \beta \subset \gamma$, β corresponds to a non-accepting state on the accepting-to-accepting path. The difference of α's and γ's lengths, which is the length of the accepting-to-accepting path, could be arbitrarily large. On the other hand, if all accepting-to-accepting paths are acyclic, for any two tokens α and γ satisfying the conditions in Theorem 1, the difference of their lengths is bounded by a constant.

Algorithm: Compute-Lookahead
/* Given a set of regular expressions, this algorithm determines the maximum number */
/* of lookahead symbols required by the scanner. */
Transform the set of regular expressions into a deterministic finite automaton.
Number the non-accepting states $1, 2, \ldots, k$; number the accepting states $k+1, k+2, \ldots, n$.
Delete all states that are not reachable from an accepting state and all accepting states.
Check whether there is any cycle in the resulting graph.
if the resulting graph contains cycles **then** return(∞) /* the FA is not an FFA */
Put all the states and edges deleted in the above step back into the graph.
for $a := 1$ **to** n **do**
  for $b := 1$ **to** n **do**
    if $a = b$ **then** $A^0[a, a] := 0$
    else if there an edge $a \longrightarrow b$ **then** $A^0[a, b] := 1$
    else $A^0[a, b] := -\infty$
for $i := 1$ **to** k **do**
  for $a := 1$ **to** n **do**
    for $b := 1$ **to** n **do**
      $A^i[a, b] := \max (A^{i-1}[a, b], A^{i-1}[a, i] + A^{i-1}[i, b])$
The maximum number of lookahead symbols required by the scanner is the maximum of
  $A^k[a, b]$, where $a = k+1, \ldots, n$ and $b = k+1, \ldots, n$.

Figure 1. The Compute-Lookahead Algorithm

To detect cycles of non-accepting states on any accepting-to-accepting paths, we may temporarily delete all the states that are not reachable from any accepting state and all the accepting states and then check whether there are cycles in the resulting graph.

After it is determined that all accepting-to-accepting paths are acyclic, the maximum number of lookahead symbols is the length of the longest accepting-to-accepting paths. This length is computed by a modified dynamic programming method. The original dynamic programming method is used to compute the length of the shortest paths between any two nodes in a directed graph [1].

In order to perform the dynamic programming steps, the non-accepting states are numbered 1 through k; the accepting states are numbered $k+1$ through n, where n is the number of states in the finite automaton. We use $A^i[a, b]$ to denote the the length of the longest paths from node a to node b that pass through only nodes $1, 2, \ldots, i$. Initially, $A^0[a, a] = 0$; if there is an edge $a \longrightarrow b$ $(a \neq b)$, $A^0[a, b] = 1$; $A^0[a, b] = -\infty$ otherwise. $A^i[a, b]$ is computed with the following formula:

$$A^i[a, b] = \max (A^{i-1}[a, b], A^{i-1}[a, i] + A^{i-1}[i, b])$$

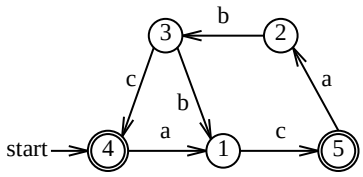
The dynamic programming steps are performed only for $i = 1, 2, \ldots, k$ since the accepting-to-accepting paths pass through only non-accepting states (that is, nodes $1, 2, \ldots, k$). The maximum number of looka-

head symbols required by the scanner is the maximum of $A^k[a, b]$, where $a = k+1, \ldots, n$ and $b = k+1, \ldots, n$. Figure 1 summaries the algorithm.

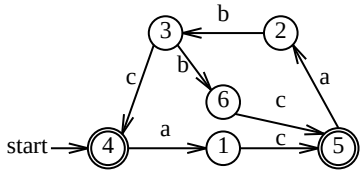
Example. Consider the two regular expressions: $\{ ac \{ abbc \}^* abc \}^*$ and $ac \{ abbc \}^* \{ abcac \{ abbc \}^* \}^*$. A deterministic finite automaton corresponding to the two regular expressions is shown in Figure 2(a). The states are numbered 1 through 5. State 4 is the initial state; states 4 and 5 are accepting states. When the two accepting states are deleted temporarily, no cycles exist in the resulting graph. Therefore, the DFA is an FFA. The dynamic programming steps are performed for $i = 1, 2,$ and 3. The A^i arrays are shown in Figure 2(c). The maximum number of lookahead symbols is the maximum of $A^3[4, 4]$, $A^3[4, 5]$, $A^3[5, 4]$, and $A^3[5, 5]$, which is 4. In this example, 4 is the length of the accepting-to-accepting path $5 \longrightarrow 2 \longrightarrow 3 \longrightarrow 1 \longrightarrow 5$. This means the scanner needs to look ahead at most 4 symbols in order to detect the end of a token. For the string *acabbb*, the scanner needs to look ahead four symbols after the token *ac*. Figure 2(b) is explained in the next section. □

The following theorem demonstrates that the maximum number of look-ahead symbols is determined from the regular expressions, rather than the particular DFA chosen for the regular expressions.

Theorem 2. *Let M_1 and M_2 be equivalent DFAs. The maximum numbers of look-ahead symbols required by M_1 and M_2 are the same.*



(a) DFA



(b) suffix FA

(c) steps in the dynamic programming

A^0	1	2	3	4	5
1	0	-∞	-∞	-∞	1
2	-∞	0	1	-∞	-∞
3	1	-∞	0	1	-∞
4	1	-∞	-∞	0	-∞
5	-∞	1	-∞	-∞	0

A^1	1	2	3	4	5
1	0	-∞	-∞	-∞	1
2	-∞	0	1	-∞	-∞
3	1	-∞	0	1	2
4	1	-∞	-∞	0	2
5	-∞	1	-∞	-∞	0

A^2	1	2	3	4	5
1	0	-∞	-∞	-∞	1
2	-∞	0	1	-∞	-∞
3	1	-∞	0	1	2
4	1	-∞	-∞	0	2
5	-∞	1	2	-∞	0

A^3	1	2	3	4	5
1	0	-∞	-∞	-∞	1
2	2	0	1	2	3
3	1	-∞	0	1	2
4	1	-∞	-∞	0	2
5	3	1	2	3	4

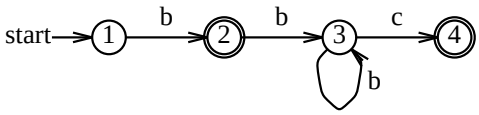
Figure 2. An example to illustrate the Compute-Lookahead algorithm.

Proof. Suppose that M_1 needs to look ahead at least one symbol. Therefore, we can find valid tokens α and γ such that (1) $\alpha \subset \gamma$, (2) for all β, $\alpha \subset \beta \subset \gamma$, β is not a valid token, and (3) $|\gamma| - |\alpha|$ is the length of the longest accepting-to-accepting paths in M_1 ($|\gamma| - |\alpha|$ can be arbitrarily large if M_1 is not an FFA). Since M_1 and M_2 are equivalent, α and γ are valid tokens for M_2 and for all β, $\alpha \subset \beta \subset \gamma$, β is not a valid token for M_2. Therefore, by Theorem 1, the maximum number of look-ahead symbols required by M_2 is at least $|\gamma| - |\alpha|$. That is, the maximum number of look-ahead symbols required by M_2 is at least as large as that required by M_1. By symmetry of the argument, The maximum number of look-ahead symbols required by M_1 is at least as large as that required by M_2.

On the other hand, suppose that M_1 does not need to look ahead any symbol. Then no valid token is a prefix of any other valid token in M_1. Since M_1 and M_2 are equivalent, no valid token is a prefix of any other valid token in M_2. Therefore, M_2 does not need to look ahead any symbol. □

Corollary. *Let M_1 and M_2 be equivalent DFAs. M_1 is an FFA if and only M_2 is an FFA.*

It is our opinion that all real-world scanners should use only finite-lookahead FAs since it is difficult to handle a potentially unbounded buffer. Furthermore, an FA that may look ahead an infinite number of symbols needs $O(n^2)$ time on input of length n. Consider the following FA, where states 2 and 4 are accepting states.



The input $bb \ldots b$ (there are n occurrences of b's) is broken into n tokens, each consisting of a single b. However, the scanner has to examine all the remaining symbols in the input in determining each token. Therefore, the scanner takes $O(n^2)$ time. By contrast, a finite-lookahead FA needs only $O(n)$ time on any input of length n.

3. Suffix finite automata

When the DFAs corresponding to the regular expressions are FFAs, table-driven look-ahead scanners can be built for the regular expressions. Rather than the minimum FFA, an equivalent finite automaton is used. This equivalent finite automaton is a member of the suffix FAs, which are defined in this section. A path on a DFA can be divided into segments by the accepting states on the path. The last segment on the path is called the *suffix*.

Definition. A *suffix* of a non-accepting state s is a path from an accepting state to s that does not go through any accepting states. If a state is not reachable from any accepting states, its suffix is the empty path. The suffix of an accepting state is always the empty path.

The length of any suffix of any state in an FFA is at most the maximum number of look-ahead symbols computed in the previous section. The *label* of a path is the concatenation of the labels of all the edges on the path. For simplicity, we assume that each edge is labeled with a symbol. There might be more than one edge from one node to another. These edges will be labeled with different symbols.

Definition. A *suffix FA* (SFA) is a deterministic finite automaton in which, for each state s, all the suffixes of state s carry the same labels.

In a suffix FA, each state s has a unique suffix label. This suffix label represents the look-ahead symbols when the finite automaton halts at state s. Note that all SFAs are also FFAs and only FFAs have equivalent SFAs.

To transform an FFA into an equivalent SFA, we split all the non-accepting states that have two or more different suffixes. For instance, consider state 1 in Figure 2(a). State 1 has two different suffixes, that is, the two paths $5 \longrightarrow 2 \longrightarrow 3 \longrightarrow 1$ and $4 \longrightarrow 1$, which are labeled *abb* and a, respectively. After splitting state 1, we get an equivalent suffix FA shown in Figure 2(b).

The algorithm in Figure 3 transforms an FFA into an equivalent SFA. All the states that are not reachable from any accepting states are removed at the very beginning. Since the traversals should avoid passing through any accepting states, all edges that enter accepting states are removed at the very beginning of the algorithm. The traversals visit the edges, rather than the nodes, in a topological order. That is, an edge $x \longrightarrow n$ is visited after all the edges entering state x have been visited. The topological order is well defined since there are no cycles after all the states unreachable from any accepting states and all the edges entering accepting states are removed from an FFA. All suffixes of a state s are represented by paths from accepting states to s. By repeatedly visiting the edges, we can find all suffix labels of all states.

Initially, it is known that the suffix labels of all accepting states are the empty strings. When an edge $x \xrightarrow{a} n$ is visited, a suffix label of state n, which is the suffix label of state x appended with the symbol a, is discovered. Because the edges are visited in a topological order, when an edge $x \xrightarrow{a} n$ is visited, a suffix label of x must be available.

```
Algorithm: Transform-FFA-to-SFA
/* Given an FFA, this algorithm produces an equivalent suffix FA. */
Initially, no states have suffix labels or buddies.
Delete all states that are not reachable from any accepting states.
Delete all edges that enter an accepting state.
for each accepting state f do
    f's suffix label := the empty string
repeat
    for each edge x --a--> n visited in a topological order do begin
        pathlabel := the suffix label of x appended with the symbol a
        if state n does not have a suffix label then n's suffix label := pathlabel
        else if n's suffix label ≠ pathlabel then begin
            for each of n's buddies m do
                if m's suffix label = pathlabel then begin
                    delete the edge x --a--> n
                    add a new edge x --a--> m
                    end
            if the suffix labels of all of n's buddies are different from pathlabel then begin
                create a new state m
                m has the same set of outgoing edges as n
                delete the edge x --a--> n
                add a new edge x --a--> m
                m's suffix label := pathlabel
                mark m as n's buddy
                end
            end
        end
until no more states need splitting
```

Figure 3. The algorithm to transform an FFA into an equivalent SFA.

Whenever it is discovered that a non-accepting state n has two different suffix labels, n is split into two states m and n. States m and n are called *buddies* (all states that originate from the same state, directly or indirectly, are called *buddies*). The newly created state m has one incoming edge $x \xrightarrow{a} m$; the original state n keeps all the incoming edges except $x \xrightarrow{a} n$. Both m and n have the same set of outgoing edges. The split operation transforms an FFA into an equivalent one. In order to avoid creating too many new states, before a state n is split, all the buddies of n are examined. If one of n's buddies m has the same suffix label as the newly created state will have, state n will not be split; instead, the edge $x \xrightarrow{a} n$ is replaced with a new edge $x \xrightarrow{a} m$.

In order to demonstrate the correctness of the transformation in Figure 3, it is necessary to give a new semantics—the scan semantics—to finite automata. The scan semantics differs from the acceptance semantics of finite automata in the formal language theory. The differences originate from three issues.

First, the scanner follows the longest-macth convention. Second, the scanner differentiates classes of input tokens. Third, the scanner cuts the input string into tokens. By contrast, in the traditional acceptance semantics a finite automaton only determines whether a string satisfies a regular expression. The scan semantics is stated in terms of equivalence of finite automata.

Definition. Two finite automata are *equivalent* if and only if they output the same sequence of tokens before they report any lexical errors for the same input string.

The correctness of the transformation is stated in Theorem 3, which is based on the following lemma.

Lemma. *The split operation in Figure 3 transforms an FFA into an equivalent one.*

Theorem 3. *The transformation algorithm in Figure 3 transforms an FFA into an equivalent SFA.*

Proof. The transformation consists of a sequence of split operations. By the above lemma, each split operation transforms an FFA into an equivalent FFA. Therefore, the final FFA is equivalent to the original one. Since the *repeat* loop iterates until, for any state s, all suffixes of s carry the same label, the final FFA is an SFA.

Next we demonstrate that the transformation terminates in a finite amount of time. Because the input is an FFA, in which there are no cycles of non-accepting states reachable from an accepting state, the number and lengths of accepting-to-accepting paths are finite. Note that a split does not increase the number and lengths of accepting-to-accepting paths (it only changes the states on the accepting-to-accepting paths). Because the number and lengths of accepting-to-accepting paths are finite and fixed, the number of non-accepting states remains finite even if every non-accepting state is on at most one accepting-to-accepting path after successive splits. Since each iteration splits at least one non-accepting state (otherwise the transformation terminates), the *repeat* loop cannot iterate an infinite number of times. Furthermore, since each iteration visits an edge at most once, each iteration terminates in a finite amount of time. We conclude that the transformation terminates. □

Theorem 4. *All and only FFAs have equivalent SFAs.*

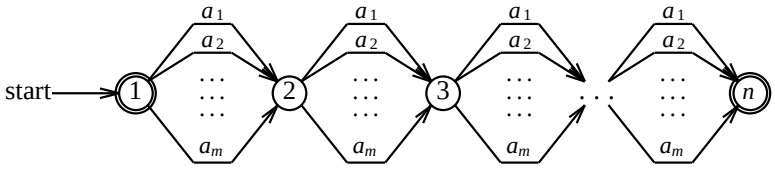
Proof. It follows from Theorem 3 that all FFAs have equivalent SFAs. On the other hand, if a DFA looks ahead a potentially infinite number of symbols, by Theorem 2, all other equivalent DFAs also look ahead a potentially infinite number of symbols. By Theorem 1, in all DFAs that look ahead a potentially infinite number of symbols, there are cycles of non-accepting states and all states on the cycles are reachable from an accepting state. Because a non-accepting state on such cycles has an infinite number of different suffix labels, any DFA that contains such cycles is not an SFA. Therefore, a DFA that looks

ahead a potentially infinite number of symbols has no equivalent SFAs. □

A topological traversal terminates after each edge is visited once. Therefore, each iteration of the *repeat* loop terminates in $O(e)$ time, where e is the number of edges in the final SFA. The *repeat* loop iterates at most k times, where k is the length of the longest accepting-to-accepting paths (this length is computed by the algorithm in the previous section). Therefore, the algorithm terminates in $O(ek)$ time.

Now consider the number of states in the final SFA. In a topological traversal, an edge can induce at most one split. There are at most lm (directed) edges, where l is the number of states in the FFA just before the traversal begins and m is the size of vocabulary of the FFA. Therefore, after one topological traversal, the number of states can become at most $lm + l$. That is, the number of states can increase at most by a factor of $m + 1$ after each traversal. There are k traversals. The number of states in the final SFA is at most $(m + 1)^k n$, where n is the number states in the original FFA. (By Theorem 2, k remains constant no matter how many new states are created.) Note that k can be as large as n in the worst case. Therefore, the number of states in the final SFA is $O((m + 1)^n n)$.

The number of states can actually increase exponentially after transformation. Consider the following FFA.



The number of states of the SFA produced by the transformation algorithm is $1 + (m^{n-1} - 1) / (m - 1)$. Though in the worst case k can be as large as n, we expect k to be a small constant in practice. For example, Pascal and Ada scanners need to look ahead at most two symbols.

Next consider the number of edges in the final SFA. Because the SFA is deterministic and the number of states in the final SFA is at most $(m + 1)^k n$, there can be at most $m(m + 1)^k n$ edges.

We have grossly over-estimated the size of the final SFA. Note that only non-accepting states that are reachable from accepting states can be split. The FFAs of the scanners for practical programming languages are quite "flat"; that is, the fanout of the initial state is large. Furthermore, the maximum number of look-ahead symbols is quite small. We do not expect that the SFA will be significantly larger than the minimum FFA. In particular, for languages that require at most 1-symbol look-ahead, no splitting is ever needed. All FFAs that look ahead at most one symbol are SFAs.

4. Lexical tables

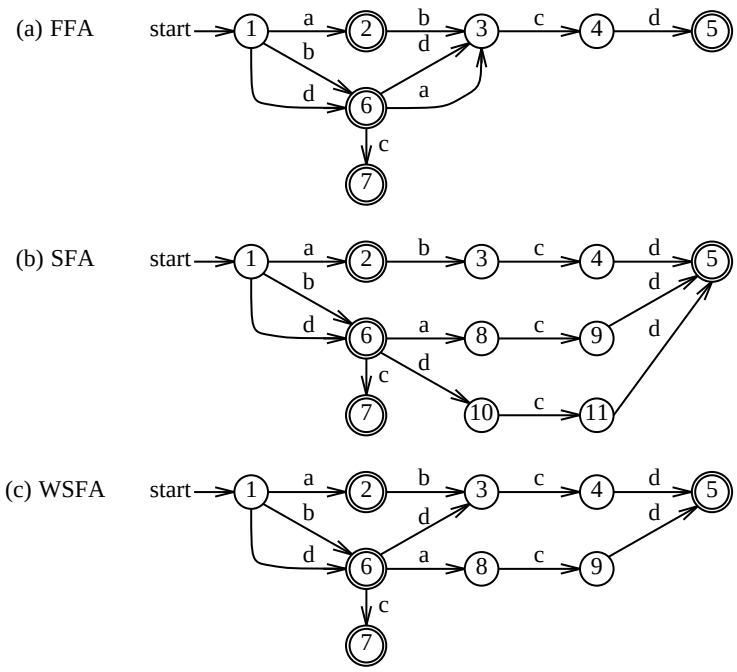
The suffix FA is the basis for the new scanner, which is discussed in the next section. In addition to the traditional state transition table, three new tables are generated: a continuation table, an action table, and an output table. Remember that the suffix labels of a state s represent the previewed symbols from input when the scanner halts at state s. When the next token is requested, this suffix label should be scanned again. Since in an SFA, each state has a unique suffix label, it is possible to pre-compute the path which the SFA passes through when the suffix label of a state is fed into the SFA.

We first define a function $final$ that takes a string as argument and returns a pair, of which the first is a sequence of tokens and the second is a string. Roughly speaking, $final(\alpha)$ returns the output of the SFA and the remaining suffix of α when α is used as input. Given a string α, if the SFA can scan α completely, then $final(\alpha) = (nil, \alpha)$. On the other hand, if the SFA cannot scan α completely and some non-null prefix of α can move the SFA to an accepting state, let β be the longest prefix of α that moves the SFA from the initial state to an accepting state f. We may write α as $\beta\alpha'$. Then $final(\alpha) = (concat(\mathrm{T}f, first(final(\alpha'))), second(final(\alpha')))$, where $\mathrm{T}f$ is the token associated with state f, $first$ and $second$ return the first and second elements of a pair, respectively, and $concat$ concatenate a token to a sequence. If the SFA cannot scan α completely and no prefix of α can move the SFA to an accepting state, $final(\alpha) = (nil, error)$.

There is one entry for each state of the SFA in the continuation, action, and output tables. These entries are determined from the suffix label of the state. Let α be the suffix label of state s. $Output[s]$ is $first(final(\alpha))$. If $second(final(\alpha))$ is not *error*, let p be the path which the SFA passes through when $second(final(\alpha))$ is used as input. $Continuation[s]$ is the last state on the path p. $Action[s]$ is the token associated with the last accepting state on the path p, except the very first initial state (note that the initial state might well be an accepting state). If there is no accepting state on the path p, except the very first initial state, $action[s]$ is *NULL*.

Suppose $second(final(\alpha))$ is an *error*. Then $continuation[s]$ is *error*, which means that the SFA eventually detects the lexical error when the suffix label α of state s is used as input. $Action[s]$ is *NULL* in this case.

Example. Figure 4(a) shows an FFA. The depth-first traversals starting from states 2, 6, 7, and 5 (*i.e.* the accepting states) will visit the edges $2 \xrightarrow{b} 3$, $3 \xrightarrow{c} 4$, $4 \xrightarrow{d} 5$, $6 \xrightarrow{d} 3$, $3 \xrightarrow{c} 4$, $4 \xrightarrow{d} 5$, and $6 \xrightarrow{a} 3$ in that order. State 3 has three different suffix labels: b, c, and d. It is split into three states 3, 8, and 10.



(d) CON, ACT, and OUT tables

states	1	2	3	4	5	6	7	8	9	10	11
CON	1	1	6	7	1	1	1	2	error	6	7
ACT	NULL	NULL	T6	T7	NULL	NULL	NULL	T2	NULL	T6	T7
OUT	*nil*	*nil*	*nil*	*nil*	*nil*	*nil*	*nil*	*nil*	T2	*nil*	*nil*

Figure 4. An FFA, its equivalent SFA and WSFA, the continuation, action, and output tables of the SFA.

After state 3 is split, state 4 will be split as well, creating new states 9 and 11. The resulting SFA is shown in Figure 4(b). We will explain the WSFA of Figure 4(c) later. Note that four new states are added to the minimum DFA of Figure 4(a).

Next the three tables are computed. The suffix labels of accepting states are the empty strings. The entries of the accepting states in the continuation table are always the initial state (which is state 1 in this example). The entries of the accepting states in the action table are always *NULL*. The entries of the accepting states in the output table are always the empty sequence. So only states 1, 3, 4, 8, 9, 10, and 11 need to be considered.

The suffix of state 1 is the empty string. So the entries of state 1 in the three tables are the same as those of the accepting states, respectively. The suffix associated with state 4 is bc. When this suffix is

fed into the SFA, the SFA goes through the path $1 \longrightarrow 6 \longrightarrow 7$. Therefore, *continuation*[4] is the last state on the path, which is state 7. *Action*[4] is the token associated with the last accepting state, which is state 7. This entry is marked as T7 in the action table. The entries of states 3, 8, 10, and 11 are computed in the same way.

Let's consider state 9. The suffix of this state is ac. The SFA cannot scan ac completely. Thus, $\beta = a$ and $\alpha' = c$. The SFA ends in state 2 when β is used as input. Since state 2 is an accepting state, the token associated with state 2 (written as T2) is appended to *output*[9]. Since the SFA cannot scan α', *continuation*[9] is marked as *error*. □

There are several ways to reduce the sizes of the three tables. First, all entries of the accepting states in the three tables are not necessary. Second, all entries of the non-accepting states that are not reachable from accepting states will not be needed by the scanner. Finally, the continuation table may be combined with the state transition table. This is because the continuation table is consulted only when the corresponding entries of the state-transition table is blank. This point will become clear after we study the new scanner driver in the next section.

5. The new scanner driver

The new scanner driver depends on the state transition table to change states. The state transition table determines the next state of the SFA from the current state and the current input symbol. If no transition is possible, the entry is marked as *no-transition*. When the scanner encounters a *no-transition* entry, it has to decide the end of the current token from the symbols already examined. The decision is made by examining the last accepting state on the path that the scanner goes through. Note that each accepting state is associated with a class of tokens defined by the set of regular expressions.

Figure 5 is the scanner driver. It makes use of the state transition table for changing the states of the SFA. A variable *token* is used to record the token associated with the last accepting state that the SFA passes through during the transitions. Let s be the state when the SFA cannot make further transition. The token in the variable *token* is recognized and printed. The suffix label of state s represents the previewed symbols. When the suffix label is fed into the SFA, we know, from the definitions of the continuation, action, and output tables, that the tokens in *output*[s] will be recognized. Hence, they are printed one by one. Finally, the SFA will reach the state specified by *continuation*[s]. *Action*[s] is the token associated with the last accepting state when the suffix label of s is fed into the SFA, after all the tokens in *output*[s] are recognized.

```
Algorithm: New-Scanner-Driver
/* Given the state transition table ST, the continuation table CON, the action table*/
/* ACT, and the output table OUT of an SFA, the scanner driver groups symbols*/
/* from input into tokens. We assume that there is no transition from any state*/
/* when the next input symbol is the end-of-file symbol. */
current_state := the initial state of the SFA
token := NULL
next_symbol := next symbol from input
repeat
    if ST[current_state, next_symbol] ≠ no-transition then begin
        current_state := ST[current_state, next_symbol]
        if current_state is an accepting state
        then token := the token associated with the state current_state
        next_symbol := next symbol from input
        end
    else /* no transition is possible at this point. */
        if token = NULL then lexical_error()
        else begin
            print token
            print the tokens in OUT[current_state]
            token := ACT[current_state]
            if CON[current_state] is error then lexical_error()
            else current_state := CON[current_state]
            end
until next_symbol = end-of-file and current_state = the initial state of the SFA
```

Figure 5. The new scanner-driver.

Example. Consider the SFA of Figure 4(b) on input *abca*. The prefix *abc* moves the SFA to state 4. At state 4 and on input *a*, the SFA cannot make further state transition. Therefore, it outputs the token of the last accepting state that the SFA goes through, in this case, T2. Then the SFA jumps directly to *continuation*[4], which is state 7. Since *output*[4] is *nil*, no tokens will be printed. The variable *token* is assigned the value *action*[4], which is T7.

Then the SFA attempts to make state transition from state 7 on input *a*. However, the SFA still cannot proceed. Therefore, the token in the variable *token*—T7—is printed. The SFA jumps to *continuation*[7], which is state 1. The variable *token* is assigned the value *action*[7], which is *NULL*. Then the SFA attempts to make state transition from state 1 on input *a*. This time the SFA can change to state 2. The next input symbol is $end-of-file$; the SFA will print the token associated with state 2 and jump to *continuation*[2], which is the initial state. Finally, the SFA exits the *repeat* loop and stops. □

Scanners based on suffix FAs can detect lexical errors at an earlier time than traditional scanners. Early error detection can be accomplished by checking the continuation table before printing out a token

when the scanner encounters a *no-transition* entry. A lexical error may be reported before some preceding tokens are recognized. For instance, in Figure 4(b), given the input *bacc*, the scanner halts at state 9, after *bac* is examined. By looking at the continuation table of state 9, the scanner can declare that a lexical error will occur eventually. The new scanner is capable of detecting the lexical error at the first *c*, before the tokens *b* and *a* are produced. By contrast, a traditional scanner can detect the lexical error only after it has identified that *b* is a token associated with state 6 and *a* is a token associated with state 2.

6. The weak SFA

In the suffix FA, all suffixes of a state carry the same label. This requirement is too strong for the purpose of lexical analysis. Many states are split unnecessarily during the transformation of an FFA into an SFA. For instance, state 3 in the FFA of Figure 4(a) is split into three states 3, 8, and 10 due to different suffixes. However, states 3 and 10 may be combined into a single state since their entries in the continuation, action, and output tables are the same, respectively. Similarly, states 4 and 11 may be combined. The resulting FA is shown in Figure 4(c). Based on the above observation, we introduce the notion of *weak* suffix FA.

Remember that each valid token corresponds to an accepting state of a DFA. A DFA can identify distinct classes[1] of tokens based on the accepting states. A token class consists of a set of accepting states of the DFA. All tokens that correspond to a state in (the set of accepting states of) a token class belong to that token class. Token classes are disjoint.

Definition. Two strings α_1 and α_2 are *compatible* with respect to a DFA if and only if one of the following four conditions is satisfied.

(1) The DFA cannot completely scan α_1 and α_2 and all non-null prefixes of α_1 and α_2 cannot move the DFA to accepting states.

(2) Suppose that the DFA cannot completely scan α_1 and α_2 but some non-null prefixes of α_1 and α_2 can move the DFA to accepting states. Let β_1 and β_2 be the longest prefixes of α_1 and α_2, respectively, that move the DFA to accepting states. We may write α_1 and α_2 as $\beta_1\alpha'_1$ and $\beta_2\alpha'_2$, respectively. Then β_1 and β_2 belong to the same token class and α'_1 and α'_2 are compatible with respect to the DFA.

---

[1]Remember that, in the scan semantics, FAs differentiate classes of tokens.

(3) Suppose that the DFA can completely scan α_1 and α_2 but all non-null prefixes of α_1 and α_2 cannot move the DFA to accepting states. Then α_1 and α_2 move the DFA to the same state.

(4) Suppose that the DFA can completely scan α_1 and α_2 and some non-null prefixes of α_1 and α_2 can move the DFA to accepting states. Let β_1 and β_2 be the longest prefixes of α_1 and α_2, respectively, that move the DFA to accepting states. We may write α_1 and α_2 as $\beta_1\alpha'_1$ and $\beta_2\alpha'_2$, respectively. Then β_1 and β_2 belong to the same token class and α_1 and α_2 move the DFA to the same state.

Intuitively, a DFA processes two compatible strings in a similar way: either the DFA halts with lexical errors (case 1 above) or the DFA produces the same sequences of tokens (case 2 above) and ends in the same terminating states (cases 2, 3, and 4).

Compatibility is an equivalence relation over the set of all strings. The next theorem can be proved by analysis of the four cases in the definition of compatibility.

Theorem 5. *Compatibility relations with respect to equivalent DFAs are identical.*

Proof. Suppose M_1 and M_2 are equivalent DFAs (under the scan semantics). It suffices to show that the following four conditions are always true for any string α.

(1) If M_1 cannot completely scan α and all non-null prefixes of α cannot move M_1 to an accepting state, then M_2 cannot completely scan α and all non-null prefixes of α cannot move M_2 to an accepting state.

(2) If M_1 cannot completely scan α and some non-null prefixes of α can move M_1 to an accepting state, then M_2 cannot completely scan α and some non-null prefixes of α can move M_2 to an accepting state. Furthermore, the longest prefix of α that moves M_1 to an accepting state is the same as the longest prefix of α that moves M_2 to an accepting state.

(3) If M_1 can completely scan α and all non-null prefixes of α cannot move M_1 to an accepting state, then M_2 can completely scan α and all non-null prefixes of α cannot move M_2 to an accepting state.

(4) If M_1 can completely scan α and some non-null prefixes of α can move M_1 to an accepting state, then M_2 can completely scan α and some non-null prefixes of α can move M_2 to an accepting state. Furthermore, the longest prefix of α that moves M_1 to an accepting state is the same as the longest prefix of α that moves M_2 to an accepting state.

These four conditions follow directly from the definition of equality of two DFAs given in the previous section. Based on the above four conditions, α_1 and α_2 are compatible with respect to M_1 if and only if they are compatible with respect to M_2. Therefore, the compatibility relations with respect to equivalent

DFAs are identical. □

Definition. A *weak suffix FA* (WSFA) is an FFA in which, for any state s, all suffix labels of s are compatible with respect to the FFA.

An SFA is a WSFA because all suffix labels of a state are identical in an SFA and compatibility is a reflexive relation. A WSFA is not necessarily a SFA.

We may modify the algorithm of Figure 3 to transform an FFA to a WSFA. When an edge $x \xrightarrow{a} n$ is visited during the topological traversal, a suffix label of n, which is a suffix label of x appended with the symbol a, is discovered. This newly discovered suffix label of n is tested for compatibility (rather than equality as is done in Figure 3) with other suffix labels of n that have already been discovered. If the newly discovered suffix label of n is not compatible, the state n is split into two states m and n.

The WSFA contains fewer states than the equivalent SFA since fewer states are split during the transformation from the minimum FFA. The WSFA can be used instead of the SFA as the the basis for the new scanner. The continuation, action, and output tables are computed in the same way.

7. Incremental lexical analysis

Suppose the string $\alpha\beta\gamma$ is scanned and broken into tokens. Then β is replaced by another string δ. Incremental lexical analysis is to identify tokens in the string $\alpha\delta\gamma$, using as much information as possible from the tokens in $\alpha\beta\gamma$. Incremental lexical analysis can be applied in incremental integrated programming environments [4].

Due to the look-ahead behavior of the scanner, incremental lexical analysis needs to decide the positions where re-scanning should start and stop. Specifically, we may write $\alpha\beta\gamma$ as $\tau_1\tau_2 \ldots \tau_i \ldots \tau_j \ldots \tau_m$, where each τ_k is a token recognized by the scanner. Further, assume that β starts in the middle of τ_i and ends in the middle of τ_j. If the scanner does not look ahead symbols in identifying the tokens, re-scanning may start from the beginning of τ_i when β is replaced by δ. The problem becomes more involved when the scanner may look ahead symbols. For instance, suppose the string *bdabcabcd* is fed into the scanner of Figure 4(b). The string will be broken into five tokens b, d, a, bc, and $abcd$. When the sixth symbol, a, is changed to d, the last three tokens, rather than just the last token, must be re-scanned. The resulting tokens are b, d, $abcd$, bc, and d. Note that the first two tokens are not changed.

To perform incremental lexical analysis, the scanner needs to record the *look-ahead positions* of tokens and the accepting states of the DFA when the tokens are recognized, in addition to the starting and ending

positions of tokens. (It is obvious that the ending positions of tokens are immediately before the starting positions of the following tokens; for the sake of presentation, the ending positions are also recorded explicitly.) The look-ahead position of a token τ is the position of the last symbol that is previewed by scanner when τ is identified. The accepting state of the DFA when the token τ is recognized is useful for re-scanning. Therefore, each token is represented by a quadruple: the starting position, the ending position, the look-ahead position, and the accepting state of the DFA when the token is recognized. The quadruples are sorted in the ascending order of the starting positions.

In the above example, the first token *b* of the string *bdabcabcd* starts at position 1 and ends at position 1. The scanner has to look ahead one symbol—*d*—when the token *b* is identified. Hence, the look-ahead position is position 2. The accepting state of the token *b* is state 6. Therefore, the first token *b* is represented by the quadruple (1, 1, 2, 6). Similarly, the remaining four tokens of *bdabcabcd* are represented by the quadruples (2, 2, 2, 7), (3, 3, 5, 2), (4, 5, 5, 8) and (6, 9, 9, 5). The five quadruples are ordered by the their starting positions.

When a string $\alpha\beta\gamma$ is changed to $\alpha\delta\gamma$, we need to determine the position and the state of the FA when re-scanning starts. The tokens in α whose look-ahead positions are before the end of α will not be affected by the change. The tokens whose look-ahead positions are immediately preceding β, within β, or after β may potentially be affected. Therefore, we find the first token τ whose look-ahead position is immediately preceding β, within β, or after β.

If τ does not overlap with β, re-scanning can start at the position that is 1 plus τ's ending position. In this case, the state of the DFA is set to the accepting state of the SFA when τ is recognized. The situation is shown in Figure 6(a), in which p and q are the starting and ending positions of τ and r is its look-ahead position. Re-scanning starts from position $q + 1$. On the other hand, if τ overlaps with β, re-scanning should start at the starting position of τ; the state of the DFA is set to its initial state. The situation is shown in Figure 6(b). Re-scanning starts from the beginning of τ. Re-scanning stops as soon as δ is completely scanned and the scanner starts identifying a token at a position that is the first component of certain quadruple.

Consider the above example where the symbol at position 6 is changed from *a* to *d*. The first token whose look-ahead position is at least position 5 (the starting position of β minus 1) is the third token *a*. The quadruple of token *a* is (3, 3, 5, 2). Since this token does not overlap with β, re-scanning starts at position 4 and the state of the DFA is set to state 2.

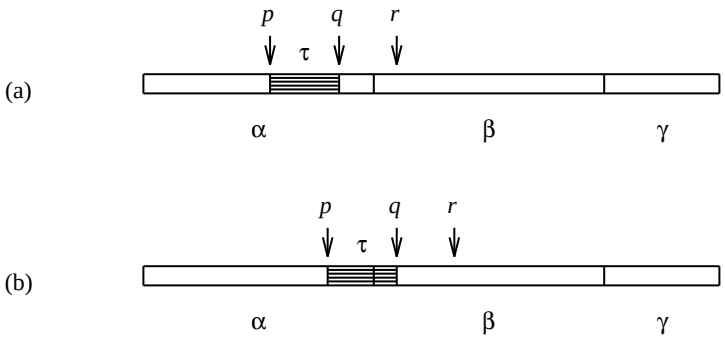


Figure 6. Re-scanning should start at appropriate positions.

An incremental scanner can be constructed from the new scanner driver of Section 4. Additional information for the look-ahead positions is needed for the tokens in the *output* table. The quadruples can be constructed when tokens are identified; the search for the appropriate quadruple can be implemented with some simple search mechanism, such as binary search.

When the chosen token τ does not overlap with β, at most n symbols of α need to be re-scanned, where n is the maximum number of look-ahead symbols of the DFA. This is because τ's look-ahead position must be at least immediately preceding β. When τ overlaps with β, τ must be re-scanned completely. The length of token τ can potentially be as large as $|\alpha|$.

8. Conclusion and related work

The look-ahead problem is examined in detail in this paper. It is found that traditional scanners based on minimum deterministic finite automata solve the look-ahead problem in an *ad hoc* way. We propose a new table-driven scanner that solves the look-ahead problem automatically.

It is interesting to compare our approach with others. Scanners generated by *lex* [13] always reserve a buffer of 1024 characters to accommodate the previewed symbols. *Lex* also generates code to handle the previewed symbols. *Scangen* [6] generates only the transition table of the minimum DFA. The look-ahead problem is left to the scanner writer. In *flex* [16], the recommended way to get rid of look-ahead (called backtracking) is to introduce "error" token definitions. By contrast, our approach solves the look-ahead problem in a clean and automatic way. In TOOLS [12], backtracking is avoided by adding a new kind of token; it is similar to the error-token approach. A user still needs to write code to handle the new

kind of token. Alex [14] introduces the *if-followed-by* operator to define the right context of a token. Instead of attacking the look-ahead problem, this operator actually changes the longest-match convention. Alex does not allow look-ahead in the general sense. LexAGen [17] solves the look-ahead problem in essentially the same way as Alex does. It allows two-character look-ahead and employs special *LookAheadStates* to solve the problem. With appropriate modifications to the continuation, action, and output tables, the new scanner discussed in this paper may solve the right context problem without re-scanning. Nawrocki [15] solves a problem similar to that of Alex and LexAGen by checking the left context derived from the LALR grammars of the programming languages. GLA [8] does not address the look-ahead problem; backtracking is not allowed. Rex [7] uses a tunnel automaton for efficient scanner generation. It does not address the look-ahead problem.

Traditionally, scanners generated from regular expressions [10] are slower than one carefully coded by hand. However, Waite [18] reports that new scanner generators can produce scanners that are as fast as hand-coded scanners. GLA [8] and Mkscan [9] do not support the full set of regular expressions. They aim at handling only those tokens that are used in common programming languages. LexAGen [17] provides a graphic user interface for constructing scanners incrementally. ALADIN [5] ignores the look-ahead problem by adopting a multiple-match rule, instead of the longest-match rule.

Incremental lexical analysis in Galaxy [4] did not take the look-ahead problem into consideration. Since the incremental scanning algorithm in Galaxy checks the tokens that abut the changed symbols, their algorithm is useful for languages that require at most 1-symbol look-ahead.

Our approach to the look-ahead problem is similar in spirit to the string pattern matching algorithm of Knuth, Morris, and Pratt [11]. The continuation table can be considered as a generalized *failure* function in [11]. The efficient string matching algorithm of Aho and Corasick [2] can locate all occurrences of any of a finite number of keywords in a text. These occurrences of keywords may overlap. By contrast, the new scanner discussed in this paper can only identify non-overlapping tokens; however, it is able to identify all the infinite sets of tokens that can be defined by FFAs.

In formulating the lookahead problem we follow the longest-match convention. A different problem may be proposed as follows: The scanner is asked to cut the input string into pieces so that each piece is a valid token. This problem is more difficult in that the scanner may need to examine the whole input file before it can identify tokens. For instance, consider the Fortan statement "*DO10I=1,10*". Finite-automata-based scanners will return the five tokens "*DO10I*", "=", "*1*", ",", and "*10*". However, a correct

scanner *should* return the seven tokens "*DO*", "*10*", "*I*", "*=*", "*1*", ",", and "*10*". Traditional scanners built from regular expressions cannot solve this more difficult problem.

ACKNOWLDEGEMENTS

The author would like to thank the referees for their valuable comments on an earlier version of this paper.

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Reading, MA: Addison-Wesley 1974
2. Aho, A.V., Corasick, M.J.: Efficient string matching: An aid to bibliographic search. Commun. ACM **18**(6), 333-340 (1975)
3. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Reading, MA: Addison-Wesley 1986
4. Beetem, J.F., Beetem, A.F.: Incremental scanning and parsing with Galaxy. IEEE Trans. Software Engineering **17**(7), 641-651 (1991)
5. Fischer, B., Hammer, C., Struckmann, W.: ALADIN: A scanner generator for incremental programming environments. Software—Practice and Experience **22**(11), 1011-1025 (1992)
6. Fischer, C.N., LeBlanc, R.J., Jr.: Crafting a Compiler with C. Reading, MA: Benjamin/Cummings 1991
7. Grosch, J.: Efficient generation of lexical analysers. Software—Practice and Experience **19**(11), 1089-1103 (1989)
8. Heuring, V.P.: The automatic generation of fast lexical analysers. Software—Practice and Experience **16**(9), 801-808 (1986)
9. Horspool, R.N., Levy, M.R.: Mkscan—An interactive scanner generator. Software—Practice and Experience **17**(6), 369-378 (1987)
10. Johnson, W.L., Porter, J.H., Ackley, S.I., Ross, D.T.: Automatic generation of efficient lexical processors using finite state techniques. Commun. ACM **11**(12), 305-313 (1968)
11. Knuth, D.E., Morris, J.H., Jr., Pratt, V.R.: Fast pattern matching in strings. SIAM J. on Computing **6**(2), (1977)
12. Koskimies, K., Paakki, J.: Automating Language Implementation. New York: Ellis Horwood 1990
13. Lesk, M.E., Schmidt, E.: LEX — A lexical analyzer generator. Computer Science Technical Report 39, Bell Labs., Murray Hill, N.J., 1975

14. Mössenböck, H.: Alex — A simple and efficient scanner generator. ACM SIGPLAN Notices **21**(5), 69-78 (1986)

15. Nawrocki, J.R.: Conflict detection and resolution in a lexical analyzer generator. Information Processing Letters **38**, 323-328 (1991)

16. Paxson, V.: The Flex User Document, Version 2.3. Computer Science Department, Cornell Univ., Ithaca, NY, 1990

17. Szafron, D., Ng, R.: LexAGen: An interactive incremental scanner generator. Software—Practice and Experience **20**(5), 459-483 (1990)

18. Waite, W.M.: The cost of lexical analysis. Software—Practice and Experience **16**(5), 473-488 (1986)

19. Wirth, N.: Programming with Modula-2 (3rd corrected ed.). New York: Springer-Verlag 1985

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 Corasick, M.J., Efficient string matching: An aid to bibliographic search, *Comm. ACM* **18**(6) pp. 333-340 (June 1975).

3. Aho, A.V., Sethi, R., and Ullman, J.D., *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA (1986).

4. Beetem, J.F. and Beetem, A.F., Incremental scanning and parsing with Galaxy, *IEEE Trans. Software Engineering* **17**(7) pp. 641-651 (July 1991).

5. Fischer, B., Hammer, C., and Struckmann, W., ALADIN: A scanner generator for incremental programming environments, *Software—Practice and Experience* **22**(11) pp. 1011-1025 (November 1992).

6. Fischer, C.N. and LeBlanc, R.J. Jr., *Crafting a Compiler with C*, Benjamin/Cummings, Reading, MA (1991).

7. Grosch, J., Efficient generation of lexical analysers, *Software—Practice and Experience* **19**(11) pp. 1089-1103 (November 1989).

8. Heuring, V.P., The automatic generation of fast lexical analysers, *Software—Practice and Experience* **16**(9) pp. 801-808 (September 1986).

9. Horspool, R.N. and Levy, M.R., Mkscan—An interactive scanner generator, *Software—Practice and Experience* **17**(6) pp. 369-378 (June 1987).

10. Johnson, W.L., Porter, J.H., Ackley, S.I., and Ross, D.T., Automatic generation of efficient lexical processors using finite state techniques, *Comm. ACM* **11**(12) pp. 305-313 (November 1968).

11. Knuth, D.E., Morris, Jr., J.H., and Pratt, V.R., Fast pattern matching in strings, *SIAM J. on Computing* **6**(2)(1977).

12. Koskimies, K. and Paakki, J., *Automating Language Implementation*, Ellis Horwood, New York (1990).

13. Lesk, M.E. and Schmidt, E., LEX — A lexical analyzer generator, Computer Science Technical Report 39, Bell Labs., Murray Hill, N.J. (1975).

14. Mossenbock, H., Alex — A simple and efficient scanner generator, *ACM SIGPLAN Notices* **21**(5) pp. 69-78 (May 1986).

15. Nawrocki, J.R., Conflict detection and resolution in a lexical analyzer generator, *Information Processing Letters* **38** pp. 323-328 (1991).

16. Paxson, V., *The Flex User Document, Version 2.3*, Computer Science Department, Cornell Univ., Ithaca, NY (May 1990).

17. Szafron, D. and Ng, R., LexAGen: An interactive incremental scanner generator, *Software—Practice and Experience* **20**(5) pp. 459-483 (May 1990).

18. Waite, W.M., The cost of lexical analysis, *Software—Practice and Experience* **16**(5) pp. 473-488 (May 1986).

19. Wirth, N., *Programming with Modula-2 (3rd corrected ed.)*, Springer-Verlag, New York (1985).