MEALY MACHINES ARE A BETTER MODEL OF LEXICAL ANALYZERS

WUU YANG

Computer and Information Science Department,

National Chiao-Tung University, HsinChu, Taiwan, R.O.C.

Abstract—Lexical analyzers partition input characters into tokens. When ambiguities arise during lexical analysis, the longest-match rule is generally adopted to resolve the ambiguities. The longest-match rule causes the look-ahead problem in traditional lexical analyzers, which are based on Moore machines. In Moore machines, output tokens are associated with states of the automata. By contrast, because Mealy machines associate output tokens with state transitions, the look-ahead behaviors can be encoded in their state transition tables. Therefore, we believe that lexical analyzers should be based on Mealy machines, rather than Moore machines, in order to solve the look-ahead problem. We propose techniques to construct Mealy machines from regular expressions and to perform sequential and data-parallel lexical analysis with these Mealy machines.

Key Words: automata, finite-lookahead automata, lexical analysis, Mealy machines, Moore machines, parallel algorithms, regular expressions, suffix automata

1. INTRODUCTION

Lexical analyzers of compilers partition input characters into tokens. Automata, in particular, the Moore machines, are a traditional model of the lexical analyzers [1]. In Moore machines, the output tokens are associated with the accepting states. However, the Moore machines suffer from the look-ahead problem caused by the longest-match rule.

When input characters are partitioned into tokens, ambiguities may arise. In lexical analysis, the *longest-match* rule is generally adopted to resolve ambiguities. The longest-match rule dictates that the scanner should find the longest string that satisfies a token definition. For instance, the input string 123456 is considered to be an integer of six digits, rather than six integers of one digit each.

This work was supported in part by National Science Council, Taiwan, R.O.C. under grants NSC 84-2213-E-009-043 and NSC 85-2213-E-009-051.

The longest-match rule causes the look-ahead problem. A lexical analyzer adopting the longest-match rule usually needs to look beyond the end of the current token in order to decide the current token. Consider the example of 10..20 in Pascal or Ada. A scanner needs to look ahead the two dots .. after the 10. This look-ahead problem is usually solved by buffering the input characters and later backtracking to re-scan the previewed characters. The buffering and re-scanning operations are beyond the capabilities of Moore machines. By contrast, these operations can be encoded naturally in the Mealy machines.

The look-ahead problem can be divided into two subclasses: the *finite*-lookahead problem and the *infinite*-lookahead problem. Previously, we solved the finite-lookahead problem with suffix automata [2]. Backtracking is avoided if the suffix automata, rather than the traditional minimum deterministic automata, are used in the lexical analyzers. In this paper, we propose two techniques to solve the infinite-lookahead problem. The first technique is a new notation, called the *cut* operator, in writing regular expressions. The cut operators are used to designate cutoff states. The second technique is to automatically identify cutoff states by examining the context-free syntax of the programming languages. The cutoff states cut off backtracking. Hence, infinite-lookaheads are truncated into finite ones by these cutoff states. The *generalized suffix automata* combine suffix automata and cutoff states. Lexical analyzers based on generalized suffix automata can solve the (finite and infinite) look-ahead problem without buffering and re-scanning overhead. A practical application of generalized suffix automata is the lexical analyzers of Modula-2 [3], which may look ahead potentially infinite number of characters.

Although generalized suffix automata, which are Moore machines, avoid the buffering and re-scanning overhead, they still depend on auxiliary data structures to solve the look-ahead problem. Generalized suffix automata can be further transformed to Mealy machines. Mealy machines may encode lexical analyzers' longest-match behaviors in the state transition and output functions. We believe that Mealy machines are a better model of lexical analyzers than Moore machines because they solve the look-ahead problem without any extra-automata devices.

Our belief is strengthed by examining the data-parallel lexical analysis algorithm of [4]. The messy work of buffering and re-scanning the previewed characters in Moore machines makes the data-parallel lexical analysis algorithm useless. By contrast, Mealy machines derived from generalized suffix automata can serve as the underlying data structure of the data-parallel lexical analysis algorithm because there is no need of buffering and re-scanning in Mealy machines.

The remainder of this paper is organized as follows. The next section reviews important definitions and results concerning suffix automata [2]. We present the cut operator and define the generalized suffix automata in the third section. In the fourth section, we derive the Mealy machine from a generalized suffix automaton. We discuss the lexical analysis with genralized suffix automata and with the corresponding Mealy machine in the fifth section. The last section concludes this paper and discusses related work.

2. REVIEW OF SUFFIX AUTOMATA

Previously, we solved the finite-lookahead problem with suffix automata [2]. In this section, we will review the definitions and important results. We assume throughout this paper that, in an automaton, there is a path from the starting state to every state and that there is a path from every state to an accepting state. We use the word "automaton" to mean "finite automaton".

DEFINITION. A *finite-lookahead automaton* (FA) is a deterministic automaton that looks ahead at most a finite number of input characters when determining the end of a token. An automaton that is not a finite-lookahead automaton is called an infinite-lookahead automaton.

Consider the example of 10..20 again. A closer look reveals that the reason why the scanner needs to look ahead two characters 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. *The maximum number of look-ahead characters 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 notation $\alpha \subset \beta$ means that α is a proper prefix of β. In terms of the deterministic automaton corresponding to the regular expressions that define the tokens, the numbers of look-ahead characters are equal to the lengths of the paths between two accepting states that does not pass through intermediate accepting states. We may compute the maximum number of look-ahead characters required by an automaton with dynamic programming [2]. Furthermore, we proved that all equivalent automata look ahead the same number of characters.

Finite-lookahead automata cannot be used directly in lexical analysis to solve the look-ahead problem. It is necessary to transform finite-lookahead automata into equivalent *suffix automata*.

DEFINITION. A *suffix* of a non-accepting state *s* is a path from an accepting state to *s* that does not go through any intermediate accepting states. If a state is not reachable from any accepting states, it has no suffix. The suffix of an accepting state is always the empty path.

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

We also proved that all and only finite-lookahead automata have equivalent suffix automata. FAs are converted to SAs by repeatedly splitting the states that have more than one distinct suffixes.

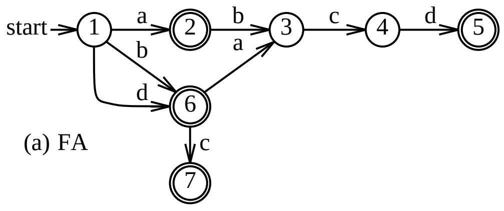
A lexical analyzer based on suffix automata is presented in [2]. That lexical analyzer does not need to buffer input, backtrack, and re-scan the previewed characters. Rather, it consults a continuation table whenever the suffix automaton cannot make a normal state-transition. From the continuation table, the automaton knows the ultimate state when the suffix of the current state is re-scanned (possibly after some tokens are produced) and hence can jump to that ultimate state directly. This is possible because, for each state s, all suffixes of s carry the same label in a suffix automaton. A few other auxiliary data structures are used to produce output tokens.

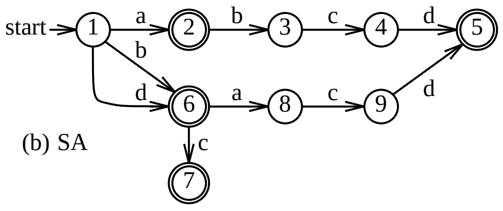
Consider the example in Figure 1(a), which is an FA. The double circles are accepting states. Since state 3 has two distinct suffixes, the path $2 \longrightarrow 3$ and the path $6 \longrightarrow 3$. State 3 is split into states 3 and 8. Similarly, state 4 is split into states 4 and 9. The resulting automaton in Figure 1(b) is an SA. Figure 1(c) is the continuation table of the SA. When the SA is in state 4 and the input character is a during scanning, the SA cannot make a state transition. Instead, it should back up two characters, output a token corresponding to the accepting state 2, and re-scan the suffix bc, and finally rest in state 7. Since all suffixes of any state in an SA are identical, back-up and re-scanning can be pre-computed and stored in the continuation table. Therefore, when the SA cannot proceed from state 4, it simply jumps directly to $continuation[state\ 4]$, which is state 7. It should be clear that the continuation table may constructed from an SA.

3. THE CUT OPERATOR AND GENERALIZED SUFFIX AUTOMATA

3.1. The cut operator

The suffix automata technique proposed in [2] is constrained to process only the finite-lookahead automata. When the regular expressions defining tokens correspond to an infinite-lookahead automaton, the suffix automata technique becomes handicapped. There are a few practical cases that do require the infinite lookahead capability. For instance, consider the regular expression for comments in Ada $--A*(EOL\ |EOF)$, where EOL and EOF are the end-of-line and end-of-file characters, respectively, and





(c) the continuation table

states	1	2	3	4	5	6	7	8	9
continuation	1	1	6	7	1	1	1	2	error

Figure 1. (a) A finite-lookahead automaton, (b) its equivalent suffix automaton, and (c) the continuation table.

A is any character other than *EOL* and *EOF*. The resulting automaton is an infinite-lookahead automaton because a single – (minus) sign is also a token in Ada. Hence, the regular expression for comments is beyond the capabilities of the suffix automata technique. But a closer look reveals that, after seeing two consecutive – signs, the scanner must discover a comment eventually since there is always an *EOF* character at the very end of the input by convention.

A similar situation arises for the regular expression for strings in Ada "(*NOT*(")|"")*". A string is defined as several characters enclosed in a pair of quotation marks. If a quotation mark must be included in the string, it must repeat itself twice. This regular expression also defines an infinite-lookahead automaton. However, from Ada's grammar, we know that there cannot be two consecutive strings in an Ada program. Therefore, after seeing two consecutive quotation marks inside a string, the scanner knows that it must eventually see at least one more quotation mark.

To extend the suffix automata technique to process infinite-lookahead automata, we introduce a new operator ! (cut). A traditional scanner needs to backtrack to the last accepting state when it cannot make further transitions from the current state and the current input character. The ! operator, which is a suffix operator, is used to cut off backtracking. This operator is similar to the cut operator in Prolog. With the ! operator, comments in Ada may be defined as $--!A*(EOL\,|EOF)$. This means that, after seeing the second – sign in two consecutive – signs, the scanner is determined to look for a comment; it will *never* backtrack across this second – sign to produce a token of a single – sign. Similarly, strings in Ada may be defined as "(*NOT*(")|""!)*", which means that, after seeing two consecutive quotation marks inside a string, the scanner is determined to look for yet another quotation mark; it will *never* backtrack across the second quotation mark in the two consecutive quotation marks.

Lexical analyzers for practical programming languages usually look ahead 2 characters. An exception is that of Modula-2 [3], which looks ahead potentially infinite number of characters. In Modula-2, a hexa-decimal number consists of digits and the characters "A" through "F" and ends with an "H". If there is no "H" at the end, the scanner must back up to the first occurrence of one of "A" through "F". For instance, 123ABCH is scanned as a hexa-decimal number whereas 123ABC is scanned as an integer 123 and an identifier ABC. But notice that an integer can never be followed by an identifier in a valid Modula-2 program. We may well treat 123ABC as a lexical error. Therefore, the regular expression for hexa-decimal numbers should be $D(D \mid X!)^*H$, where D is a digit and X is any of "A" through "F".

To process the ! operator, we notice the following three basic properties: (we assume that the ! operator has higher precedence than all other operators in a regular expression.) Let α and β be regular expressions.

(1) $(\alpha\beta)! = \alpha\beta!$

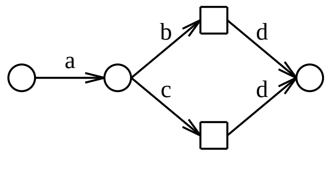
(2) $(\alpha^*)! = \lambda!(\alpha!)^*$

(3) $(\alpha \mid \beta)! = \alpha! \mid \beta!$

Based on the above three properties, we can easily verify the following lemma:

LEMMA. $\alpha^+! = \alpha!^+$.

A ! operator in a regular expression corresponds to one or more states in the automaton for the regular expression. These states are called *cutoff* states. To find the cutoff states, we may use the above three properties to transform a regular expression so that a ! operator always follows a character, say a, or a λ immediately. When the regular expression is converted into a nondeterministic automaton [5], the character a or the λ is represented by an edge, say $s \longrightarrow t$, in the graph of the nondeterministic automaton, where s and t are states. Then state t is the cutoff state corresponding to the ! operator. For instance, the regular expression $a(b \mid c)!d$ can be transformed into $a(b! \mid c!)d$. The states at the heads of arrows (transitions) representing the characters b and c are cutoff states, which are denoted by squares in the following figure.



When the nondeterministic automaton is transformed into an equivalent deterministic one by subset construction [6], all states of the deterministic automaton (which are represented as subsets of states of the nondeterministic automaton) that contain a cutoff state of the nondeterministic automaton are also marked as cutoff states. When the minimization operation is performed on the deterministic automaton [7], all states of the minimum deterministic automaton (which are represented as subsets of states of the original deterministic automaton) that contain a cutoff state of the original automaton are also marked as cutoff states. The initial partitions in the minimization algorithm are slightly different from that stated in [7]. There are three partitions initially: all the accepting states are in a partion; all cutoff states that are not accepting states and are reachable from an accepting state are in the second partition; the rest are in the third partition.

When cutoff states are marked on automata, the semantics of automata is slightly changed. We need a new definition.

DEFINITION. Two deterministics automata with cutoff states are *equivalent* if and only if they produce the same sequences of tokens for the same input string before they report any lexical errors.

Note that cutoff states do not affect the language accepted by an automaton with cutoff states in the traditional automata-theoretic semantics. Therefore, equivalent automata with cutoff states are also equivalent in the traditional sense: they accept the same language.

It can be shown that the minimization process mentioned above transforms a deterministic automaton with cutoff states into an equivalent one. In addition, equivalence defined above is a reflexive, symmetric, and transitive relation. We can prove the following theorem.

THEOREM. Two deterministic automata with cutoff states are *equivalent* if and only if they can be transformed to identical deterministic automata with cutoff states by the minimization process.

3.2. The generalized suffix automata

A cutoff state is useful only if it is not an accepting state and it is reachable from an accepting state in the automaton. Firstly, an accepting state automatically cuts off backtracking; it does not matter whether the accepting state is a cutoff state or not. Secondly, if a cutoff state is not reachable from an accepting state, there is no need to backtrack across this cutoff state. A useful cutoff state will be treated as an accepting state since it cuts off backtracking in the same way as an accepting state does. From now on, we will assume that all cutoff states are not accepting states and that all cutoff states are reachable from some accepting states.

DEFINITION. A state is an *ending* state if it is an accepting state or a cutoff state.

Example. Consider the regular expression for comments in Ada $--!A*(EOL \mid EOF)$. Its minimum deterministic automaton is shown in Figure 2. The cutoff states are indicated by squares. Accepting states are by double circles.

DEFINITION. A deterministic automaton is a *generalized finite-lookahead automaton* (GFA) if and only if all paths from an accepting state to a (not necessarily distinct) ending state, without going through intermediate ending states, are of finite lengths.

THEOREM. Suppose that M and N are equivalent deterministic automata with cutoff states. M is a GFA if and only if N is a GFA.

PROOF. Suppose that M is a GFA but N is not. Since N is not a GFA, there must exist an accepting state s and an ending state t such that there is a path P from s to t on which there are no intermediate ending states. Furthermore, the path P can be arbitrarily long. Let α be a string accepted by state s and β be the label of path P.

Let γ be any non-λ proper prefix of β. First, we claim that M can completely scan $\alpha\gamma$ without reporting a lexical error. The reason is as follows. Since N can completely scan $\alpha\gamma$, there must exist a string δ such that $\alpha\gamma\delta$ is a token accepted by N. Since M and N are equivalent, M must also accept $\alpha\gamma\delta$ as a token. Therefore, M must be able to completely scan $\alpha\gamma$. Let u be the state of the automaton M after $\alpha\gamma$ is fed into M. Second, we claim that u cannot be an accepting state for otherwise $\alpha\gamma$ is accepted by M but not by N, which contradicts the assumption that M and N are equivalent.

Since the path P could be arbitrarily long, we could choose appropriate P and γ so that the state u is a cutoff state of M. This is due to the fact that, in M, there are only a finite number of states that are not ending states. Without loss of generality, assume that state u is a cutoff state. Consider the input string

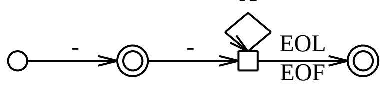


Figure 2. A finite automaton with cut states.

$\alpha\gamma\$$, where $\$$ is the end-of-file character, which does not appear in the transition tables of M and N. The automaton N will return a token representing α before reporting a lexical error at the $\$$ sign. However, the automaton M will not return any token. Therefore, M and N are not equivalent. This contradicts a previous assumption.

We conclude that N must also be a GFA. By symmetry of the arguments, we proved the theorem. $\square$

By similar arguments, we can prove that M and N look ahead the same number of characters. Since this result is not required, the proof is omitted.

In the definition of suffix automata, a suffix of state s, which represents the previewed characters, is a path from an accepting state to s. In the generalized suffix automata (to be defined later), only those states that are reachable from an accepting state without passing through intermediate ending states have suffixes.

DEFINITION. A *suffix* of a non-accepting state s is a path from an accepting state to s that does not go through any intermediate ending states. If such a path does not exist, the state s has no suffix. The only suffix of an accepting state is always the empty path.

DEFINITION. A *generalized suffix automaton* (GSA) is a deterministic automaton (with or without cutoff states) in which, for each state s, all suffixes of s carry the same labels.

A GFA can be transformed into a GSA by splitting states. To split a state s of a deterministic automaton, we create a new state s'. One or more of the original incoming edges of state s are re-directed to the new state s'. State s' has the same set of outgoing edges as s. When a cutoff state is split, the newly generated states are also cutoff states. Figure 3(a) is a recast of the splitting procedure. For example, consider Figure 1(a). State 3 has two incoming edges and one outgoing edge. When state 3 is split, the new state 8 is generated. State 8 has the same outgoing edge as state 3. One of the incoming edge of state 3, $6 \longrightarrow 3$, is redirected to state 8, $6 \longrightarrow 8$. Similarly, when state 4 is split, the new state 9 is generated. The result of splitting states 3 and 4 is shown in Figure 1(b).

Suppose that a deterministic automaton M' is obtained from another deterministic automaton M by splitting a state. It can be shown that M' and M can be transformed to the same deterministic automaton with cutoff states by the minimization process. Therefore, by a previous theorem, M and M' are equivalent. We conclude that the state-splitting process transforms a deterministic automaton with cutoff states into an equivalent one. Figure 3(b) presents the transformation procedure. In a GFA, if any state has more than one distinct suffix, we may split that state. By repeatedly splitting states, we can transform

```
(a) the splitting procedure
   procedure split ($n$, $s \longrightarrow n$)
   /* $n$ is a state; $s \longrightarrow n$ is an incoming edge of state $n$.  */
   create a new state $m$
   for each outgoing edge $n \longrightarrow t$ do
      create a new edge $m \longrightarrow t$
   od
   delete the edge $s \longrightarrow n$
   create a new edge $s \longrightarrow m$


(b) Transformation procedure
   procedure transform ($M$)
   /* M is GFA; the result of transformation is an equivalent GSA.  */
   repeat
      choose a state $n$ that satisfies the following conditions:
         (1) there is a path from an accepting state to $n$ without passing through ending states
         (2) $n$ has two incoming edges $s_1 \longrightarrow n$ and $s_2 \longrightarrow n$
      split ($n$, $s_1 \longrightarrow n$)
   until no more change
```

Figure 3. (a) The splitting procedure and (b) the transformation procedure.

a GFA to an equivalent GSA. This result is similar to a result in [2]: A finite-lookahead automaton may be transformed to a suffix automaton by a similar state-splitting process.

Similar to suffix automata, generalized suffix automata can be used to implement lexical analyzers that avoids backtracking. We have implemented the suffix automata technique and the ! operator in an improved lexical analyzer generator, called *scangen2*. The implementation is an enhancement of the *scangen* program [6].

3.3. Automatically identifying cutoff states

The cut operators are essentially used to identify the cutoff states of infinite-lookahead automata. The designer of token definitions needs to insert the cut operators in appropriate places. An alternative is for the scanner generators to identify cutoff states automatically. We can do this by analyzing the context-free grammar of the programming language.

In a conventional compiler, token definitions and the context-free syntax are usually independent. However, there is a subtle relationship between the two that we can make use of to identify the cutoff states. First notice that, from the context-free grammar, we know whether a sequence of two or more tokens can appear in a program. For instance, an integer can never be followed by an identifier in a

Modula-2 program. (This justifies the use of a cut operator in the earlier Modula-2 example in Section 3.1.)

DEFINITION. Given a context-free grammar G, a sequence of tokens is *allowable with respect to* G if and only if that sequence of tokens can appear in a sentence generated by G.

In an infinite-lookahead automaton, there must be a path P from an accepting state s to an ending state t that does not pass through intermediate ending states. (Note that, initially, all ending states are accepting states. we will identify cutoff states one by one.) Furthermore, the path P can be arbitrarily long. Let T_0 denote the token accepted by state s. On P, we can choose a non-accepting state u. Let P' be the prefix of P from s to u. Let $T_1, T_2, \ldots, T_k$ be the tokens produced by the automaton when the string P' is scanned by the automaton and P'' be the suffix of P' that is not part of any of $T_1, T_2, \ldots, T_k$. Then state u is marked as a cutoff state if (1) the automaton reports a lexical error when scanning P'' or (2) the sequence of tokens $T_0, T_1, \ldots, T_k$ is not allowable with respect to the context-free grammar.

Condition (1) above indicates that a lexical error eventually will occur if the automaton backtracks across state u. Condition (2) means that a syntactic error eventually will occur if the automaton backtracks across state u. In either case, the automaton can simply cut off backtracking at state u. The result is that (lexical or syntactic) errors are detected earlier.

We may mark sufficient cutoff states so that the resulting automaton is a generalized suffix automaton. Unfortunately, finding sufficient cutoff states seems a difficult problem. Furthermore, there is no guarantee that there will always be sufficient cutoff states in an infinite-lookahead automaton. We still need to rely on the cut operators to constrain the look-ahead behaviors in some cases.

4. COMPLETED GENERALIZED SUFFIX AUTOMATA

A lexical analyzer usually employs a single deterministic automaton to recognize various classes of tokens, such as identifiers, numerals, delimiters, etc. Different token classes correspond to different accepting states. That is, accepting states are distinguished by the token classes they recognize. In a generalized suffix automaton, the suffixes of a non-accepting state s are identical. But note that token classes of the last accepting states on all paths from the starting state to s are *not* necessarily the same. Therefore, when a generalized suffix automaton scans input characters, it has to remember (the token class of) the last accepting state it has passed with an auxiliary data structure. This information is necessary in producing correct tokens.

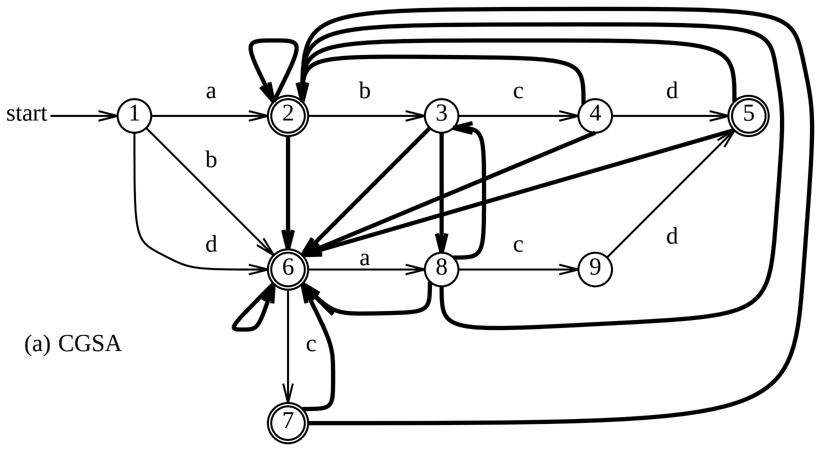
DEFINITION. A *suffix origin* of a non-accepting state s is an accepting state t such that there is a path from t to s that does not go through any intermediate ending states. If such a path does not exist, s has no suffix origins. The only suffix origin of an accepting state is the state itself.

If, for each state s, all suffix origins of s recognize the same token class, the automaton need not remember the last accepting state it has passed. The information is encoded in the states of the automaton. In other words, the auxiliary data structure is integrated into the automaton. A GSA M may be transformed by applying the state-splitting procedure repeatly so that, for any state s in the resulting automaton, all suffix origins of s recognize the same token class. Once the GSA has been transformed to this form, we are ready to derive the Mealy machine.

DEFINITION. A *completed GSA* (CGSA) M corresponding to a generalized suffix automaton N is obtained as follows: (We assume that, for each state s of N, all suffix origins of s recognize the same token for otherwise N may be transformed by the state-splitting procedure before the completion operation is applied.) Let the state transition function of N be denoted by n. (1) M has the same set of states as N. (2) The state transition function of M, denoted by m, is defined as follows: For a state s and an input character c, if $n(s, c) \neq error$ then $m(s, c) = n(s, c)$. If $n(s, c) = error$ and state s does not have a suffix origin, then $m(s, c) = error$. If $n(s, c) = error$ but state s has a suffix origin, then $m(s, c)$ is the state of N when the string αc is fed into N, possibly after some tokens are produced, where α is the suffix of state s; $m(s, c) = error$ if N cannot completely scan αc. (3) The output function of M, denoted by out, is defined as follows: For a state s and an input character c, if $n(s, c) \neq error$ then $out(s, c) = empty\_sequence$. If $n(s, c) = error$ and state s has no suffix origin, then $out(s, c) = empty\_sequence$. If $n(s, c) = error$ but state s has a suffix origin, then $out(s, c)$ is the token recognized by a suffix origin of state s concatenated with the sequence of tokens produced by N when αc is fed into N, where α is the suffix of state s.

The CGSA is identical to the original GSA except that certain *error* transitions are replaced with *completion* transitions. The completed generalized suffix automata belong to the class of Mealy machines in that the the completion transitions and some *error* transitions carry output tokens.

Example. Figure 4(a) is the completed (generalized) suffix automaton for the SA of Figure 1(b). The bold arrows, of which the labels are omitted for the sake of simplicity, are the *completion* transitions. Figure 4(b) is the state transition table. Each entry of the transition table consists of a pair: the next state and the tokens that are produced whenever the transition represented by the entry is taken. The notation



(b) the transition table

states	1	2	3	4	5	6	7	8	9
a	2	2/T2	8/T2	2/T2T7	2/T5	8	2/T7	2/T6T2	error/T6T2
b	6	3	6/T2T6	6/T2T7	6/T5	6/T6	6/T7	3/T6	error/T6T2
c	error	error/T2	4	error/T2T7	error/T5	7	error/T7	9	error/T6T2
d	6	6/T2	6/T2T6	5	6/T5	6/T6	6/T7	6/T6T2	5
$a \circ b$	8	8/T2	8/T2T6	8/T2T7	8/T5	8/T6	8/T7	8/T6T2	error/T6T2

Figure 4. (a) The completed (generalized) suffix automaton, and (b) the transition table.

T2 denotes the token class recognized by state 2. The last row of the transition table is the composition of the first and the second rows. Row composition is discussed in the next section.

The transition table of a CGSA differs from a traditional transition table. In a traditional transition table, there are many *error* entries, which indicate that the automaton cannot proceed. Some of these *error* transitions are related to the look-ahead behavior of the scanner. For instance, consider the (generalized) suffix automaton in Figure 1(b). In state 4 and on input b, the generalized suffix automaton would back up two input characters, produce a token T2 for the prefix a and a token T7 for the look-ahead bc, and rest in state 6 due to the current input character b. Therefore, that entry is marked 6/T2T7 in the CGSA.

To compute the transition table, we consider the *error* transitions. Suppose that there is an *error* transition in state s and on input c. If state s does not have a suffix origin, the generalized suffix automaton encounters a real *error*. On the other hand, if s has a suffix origin, it should output the token recognized by the suffix origins of s (note that all suffix origins of a state recognize the same token). Then the automaton starts from the initial state and attempts to scan the suffix of s (note that all suffixes of s are identical) and the current input c. Several tokens may be produced before before a correct next state or an *error* is determined. Due to the definition of generalized suffix automata, the transition table can be computed independently of the input. Therefore, a transition table like Figure 4(b) is constructed. An algorithm similar to [8] can be used to efficiently compute the transition table.

5. LEXICAL ANALYSIS WITH GSAS AND CGSAS

5.1. Sequential lexical analysis

Either the GSAs or the CGSAs may be used in sequential lexical analysis. They both avoid the buffering and re-scanning overhead in processing look-aheads. Figure 5 is a GSA-based scanner, which is derived from the SA-based scanner of [2]. A GSA uses a continuation table, which was discussed in Section 2,

```
Algorithm: GSA-Scanner
/* Given the state transition table ST, the continuation table CON, the action table*/
/* ACT, and the output table OUT of an GSA, the scanner driver groups symbols*/
/* from input into tokens. */
current_state := the initial state of the GSA
token := NULL
next_symbol := next symbol from input
repeat
    if ST[current_state, next_symbol] ≠ error 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
        else if current_state is a cutoff state then
            token := NULL
        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 GSA
```

Figure 5. Scanning with a GSA.

to skip look-aheads. It also needs to remember the last accepting state it has passed. The OUT, CON, and ACT tables are defined as follows: For a state s, $OUT[s]$ = the sequence of tokens that are produced when the suffix of state s is used as input to the automaton; $CON[s]$ = the state of the automaton when the suffix of s is used as input to the automaton, after the sequence of tokens $OUT[s]$ is produced; $ACT[s]$ = the last accepting state that the automaton passes when the suffix of s is used as input to the automaton, after the sequence of tokens $OUT[s]$ is produced. On the other hand, scanning with a CGSA simply follows the state transition table. Tokens are produced when completion transitions or certain *error* transitions are followed.

Since GSAs and CGSAs are obtained by splitting states of the minimum deterministic automata, it is reasonable to be concerned with the sizes of GSAs and CGSAs. Note that only non-accepting states that are reachable from accepting states might be split. Lexical analyzers of most practical programming languages look ahead 1 or 2 characters. This means that only a few states are candidates for splitting. For a realistic example, consider the lexical analyzer of Modula-2 published in [9]. That lexical analyzer is an infinite-lookahead automaton. After designating one state as a cutoff state, it becomes a generalized finite-lookahead automaton. The numbers of states in the GFA, GSA, and CGSA are 48, 50, 52, respectively. (A simple technique has been used to reduce the numbers of states.) Only four states are added due to state splitting. The advantage of the technique proposed in this paper is to completely relieve the scanner writers of the look-ahead problem. We believe this advantage outweighs the slightly increased table size.

5.2. Data-parallel lexical analysis

Hillis and Steele proposed a data-parallel lexical analysis algorithm [4]. Their algorithm is based on automata. However, not all automata may be used in their algorithm due to the look-ahead problem. We will show that the CGSA defined in the previous section, in fact, the Mealy machines in general, could be used in their algorithm. This discussion supports our belief that the Mealy machines are a better model of lexical analyzers.

The data-parallel lexical analysis algorithm of [4] depends on the ability of the automaton that it can detect the beginning of a token immediately after reading the first character of the token. All 1-character look-ahead automata possess such a capability; hence the 1-character look-ahead automata can be used in Hillis and Steele's algorithm. For automata that look ahead more than one character, we need to transform the automata into the completed, generalized suffix automata before applying that algorithm.

In the algorithm of [4], each input character is considered as a transition function that maps a current state to a next state or *error*. The transition function representing a string of characters is the composition of the transition functions representing individual characters. The composition of two transition functions f and g is defined as follows:

$$f \circ g(s) = \begin{cases} f(g(s)) & \text{if } g(s) \neq \mathit{error} \\ \mathit{error} & \text{otherwise} \end{cases}$$

In completed, generalized suffix automata, the output functions are also needed. Let h and k be the output functions associated with the transition functions f and g, respectively. The composition of two output functions is defined as follows. (The *append* function concatenates two sequences of tokens.)

$$h \circ k(s) = \begin{cases} \mathit{append}(k(s), h(g(s))) & \text{if } g(s) \neq \mathit{error} \\ k(s) & \text{otherwise} \end{cases}$$

It is easy to verify that function composition defined above is associative. Associativity is required in the parallel prefix-composition operations in the data-parallel algorithm.

Example. Each row of Figure 3(b) is considered as a transition function and an output function representing the character in that row. Let a and b denote the state-transition functions representing the characters a and b, respectively. Then $a \circ b(\mathit{state}\ 1) = \mathit{state}\ 8$ since $b(\mathit{state}\ 1) = \mathit{state}\ 6$ and $a(\mathit{state}\ 6) = \mathit{state}\ 8$. On the other hand, $a \circ b(\mathit{state}\ 2) = \mathit{state}\ 8$ with output T2. Similarly, $a \circ b(\mathit{state}\ 9) = \mathit{error}$ with output T6 and T2, since $b(\mathit{state}\ 9) = \mathit{error}$.

The data-parallel lexical analysis algorithm of [4] is summarized as follows. One processor is allocated for each input character. Each character is viewed as a function that maps one state to another. Since function composition is associative, a parallel prefix-composition operation is performed. Finally, the initial state is used as an argument to the output function in the last processor to extract the tokens. Detailed account of the algorithm can be found in [4].

We wish to emphasize that the data-parallel algorithm cannot be used with arbitrary automata. One that can process look-ahead appropriately and allow associative function composition is essential to the correctness of that algorithm. We found that completed, generalized suffix automata fit this purpose just right.

6. CONCLUSION AND RELATED WORK

This paper extends the capabilities of suffix automata technique [2] with the cutoff states to process infinite-lookahead automata. The cutoff states may be designated by the cut operators or be identified automatically. The resulting Mealy machines can solve the look-ahead problem without any extra-automata devices. That Mealy machines are a better model of lexical analyzers is supported by an application of CGSAs in a data-parallel lexical analysis algorithm.

It is interesting to compare our approach with others. *Lex* [10] generates code to buffer and handle the previewed symbols. Though the output of the scanner generator *Scangen* [6] is similar to a Mealy machine, the look-ahead problem is left to the scanner writer. In *flex* [11], a user introduces "error" token definitions to get rid of look-ahead (called backtracking). A user still needs to write code to handle the new kind of token. In TOOLS [12], backtracking is avoided by adding a new kind of token; it is similar to the error-token approach. Alex [9] introduces the *if-followed-by* operator. This operator actually changes the longest-match convention. Alex does not allow look-ahead in the general sense. LexAGen [13] , which is similar to Alex, allows two-character look-ahead and employs special *LookAheadStates* to solve the problem. Nawrocki [14] 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 [15] does not address the look-ahead problem; backtracking is not allowed. Rex [16] which uses a tunnel automaton for efficient scanner generation, does not address the look-ahead problem. GLA [15] and Mkscan [17] do not support the full set of regular expressions. They aim at handling only those tokens that are used in common programming languages. LexAGen [13] provides a graphic user interface for constructing scanners incrementally. Incremental generation of lexical analyzers is also discussed in [18], which did not address the lookahead problem. ALADIN [19] ignores the look-ahead problem by adopting a multiple-match rule, instead of the longest-match rule. Incremental lexical analysis in Galaxy [20] did not take the look-ahead problem into consideration; that 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 [21]. Continuation from *error* transitions can be considered as a generalized *failure* function in [21]. The efficient string matching algorithm of Aho and Corasick [8] may be adapted to compute the transition table during the completion operation.

ACKNOWLEDGEMENT The author would like to thank Jien-Tsai Chan at the National Chiao-Tung University, Taiwan, R.O.C., for implementing *scangen2*.

References

1. Hopcroft, J.E. and Ullman, J.D. *Introduction to Automata Theory, Languages, and Computation*. Reading, MA.: Addison-Wesley; 1979.
2. Yang, W. On the look-ahead problem in lexical analysis. *ACTA Informatica* **32**: 459-476; 1995.
3. Wirth, N. *Programming with Modula-2 (3rd corrected ed.)*. New York: Springer-Verlag; 1985.
4. Hillis, W.D. and Steele, Jr., G.L. Data parallel algorithms. *Comm. ACM* **29**: 1170-1183; 1986.
5. Aho, A.V., Sethi, R., and Ullman, J.D. *Compilers: Principles, Techniques, and Tools*. Reading, MA.: Addison-Wesley; 1986.
6. Fischer, C.N. and LeBlanc, Jr., R.J. *Crafting a Compiler with C*. Reading, MA.: Benjamin/Cummings; 1991.
7. Hopcroft, J.E. An *n log n* algorithm for minimizing states in a finite automaton. In *The Theory of Machines and Computations* (Edited by Z. Kohavi and A. Paz), pp. 189-196. New York: Academic Press; 1971.
8. Aho, A.V. and Corasick, M.J. Efficient string matching: An aid to bibliographic search. *Comm. ACM* **18**: 333-340; 1975.
9. Mössenböck, H. Alex — A simple and efficient scanner generator. *ACM SIGPLAN Notices* **21**: 69-78; 1986.
10. Lesk, M.E. and Schmidt, E. LEX — A lexical analyzer generator. Murray Hill, NJ: Bell Laboratories, CSTR 39; 1975.
11. Paxson, V. *The Flex User Document, Version 2.3*. Ithaca, NY: Computer Science Department, Cornell Univ.; 1990.
12. Koskimies, K. and Paakki, J. *Automating Language Implementation*. New York: Ellis Horwood; 1990.
13. Szafron, D. and Ng, R. LexAGen: An interactive incremental scanner generator. *Software—Practice and Experience* **20**: 459-483; 1990.
14. Nawrocki, J.R. Conflict detection and resolution in a lexical analyzer generator. *Information Processing Letters* **38**: 323-328; 1991.
15. Heuring, V.P. The automatic generation of fast lexical analysers. *Software—Practice and Experience* **16**: 801-808; 1986.
16. Grosch, J. Efficient generation of lexical analysers. *Software—Practice and Experience* **19**: 1089-1103; 1989.

17. Horspool, R.N. and Levy, M.R. Mkscan—An interactive scanner generator. *Software—Practice and Experience* **17**: 369-378; 1987.

18. Heering, J., Klint, P., and Rekers, J. Incremental generation of lexical scanners. *ACM Trans. Programming Languages and Systems* **14**: 490-520; 1992.

19. Fischer, B., Hammer, C., and Struckmann, W. ALADIN: A scanner generator for incremental programming environments. *Software—Practice and Experience* **22**: 1011-1025; 1992.

20. Beetem, J.F. and Beetem, A.F. Incremental scanning and parsing with Galaxy. *IEEE Trans. Software Engineering* **17**: 641-651; 1991.

21. Knuth, D.E., Morris, Jr., J.H., and Pratt, V.R. Fast pattern matching in strings. *SIAM J. on Computing* **6**: 323-350; 1977.

REFERENCES

1. J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation,* Addison-Wesley, Reading, MA (1979).
2. W. Yang, On the look-ahead problem in lexical analysis, *ACTA Informatica* **32** pp. 459-476 (1995).
3. N. Wirth, *Programming with Modula-2 (3rd corrected ed.),* Springer-Verlag, New York (1985).
4. W.D. Hillis and G.L. Steele, Jr., Data parallel algorithms, *Comm. ACM* **29**(12) pp. 1170-1183 (December 1986).
5. A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques, and Tools,* Addison-Wesley, Reading, MA (1986).
6. C.N. Fischer and R.J. LeBlanc, Jr., *Crafting a Compiler with C,* Benjamin/Cummings, Reading, MA (1991).
7. J.E. Hopcroft, An *n log n* algorithm for minimizing states in a finite automaton, pp. 189-196 in *The Theory of Machines and Computations,* ed. Z. Kohavi and A. Paz,Academic Press, New York (1971).
8. A.V. Aho and M.J. Corasick, Efficient string matching: An aid to bibliographic search, *Comm. ACM* **18**(6) pp. 333-340 (June 1975).
9. H. Mossenbock, Alex — A simple and efficient scanner generator, *ACM SIGPLAN Notices* **21**(5) pp. 69-78 (May 1986).
10. M.E. Lesk and E. Schmidt, LEX — A lexical analyzer generator, Computer Science Technical Report 39, Bell Labs., Murray Hill, N.J. (1975).
11. V. Paxson, *The Flex User Document, Version 2.3,* Computer Science Department, Cornell Univ., Ithaca, NY (May 1990).
12. K. Koskimies and J. Paakki, *Automating Language Implementation,* Ellis Horwood, New York (1990).
13. D. Szafron and R. Ng, LexAGen: An interactive incremental scanner generator, *Software—Practice and Experience* **20**(5) pp. 459-483 (May 1990).
14. J.R. Nawrocki, Conflict detection and resolution in a lexical analyzer generator, *Information Processing Letters* **38** pp. 323-328 (1991).
15. V.P. Heuring, The automatic generation of fast lexical analysers, *Software—Practice and Experience* **16**(9) pp. 801-808 (September 1986).
16. J. Grosch, Efficient generation of lexical analysers, *Software—Practice and Experience* **19**(11) pp. 1089-1103 (November 1989).
17. R.N. Horspool and M.R. Levy, Mkscan—An interactive scanner generator, *Software—Practice and Experience* **17**(6) pp. 369-378 (June 1987).

18. J. Heering, P. Klint, and J. Rekers, Incremental generation of lexical scanners, *ACM Trans. Programming Languages and Systems* **14**(4) pp. 490-520 (October 1992).

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

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

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