PERGAMON

Computer Languages, Systems & Structures 28 (2002) 273–288

COMPUTER LANGUAGES, SYSTEMS & STRUCTURES

www.elsevier.com/locate/cl

On the applicability of the longest-match rule in lexical analysis☆

Wuu Yang[a,*], Chey-Woei Tsay[b], Jien-Tsai Chan[a]

[a] *Computer and Information Science Department, National Chiao-Tung University, HsinChu, Taiwan, ROC*
[b] *Department of Computer Science and Information Management, Providence University, Taichung County, Taiwan, ROC*

Received 25 April 2002; accepted 28 June 2002

Abstract

The lexical analyzer of a compiler usually adopts the *longest-match* rule to resolve ambiguities when deciding the next token in the input stream. However, that rule may not be applicable in all situations. Because the longest-match rule is widely used, a language designer or a compiler implementor frequently overlooks the subtle implications of the rule. The consequence is either a flawed language design or a deficient implementation. We propose a method that automatically checks the applicability of the longest-match rule and identifies precisely the situations in which that rule is not applicable. The method is useful to both language designers and compiler implementors. In particular, the method is indispensable to automatic generators of language translation systems since, without the method, the generated lexical analyzers can only blindly apply the longest-match rule and this results in erroneous behaviors. The crux of the method consists of two algorithms: one is to compute the regular set of the sequences of tokens produced by a nondeterministic Mealy automaton when the automaton processes elements of an input regular set. The other is to determine whether a regular set and a context-free language have nontrivial intersection with a set of equations.

Keywords: Compiler; Context-free grammar; Finite-state automaton; Lexical analyzer; Mealy automaton; Moore automaton; Parser; Regular expression; Scanner

☆ This work was supported in part by National Science Council, Taiwan, ROC, under grants NSC 86-2213-E-009-021 and NSC 86-2213-E-009-079.

* Corresponding author. Tel.: +886-3-5712121x56614; fax: +886-3-5721490.
E-mail address: wuuyang@cis.nctu.edu.tw (W. Yang).

1477-8424/02/$ - see front matter
PII: S0096-0551(02)00014-0

1. Introduction

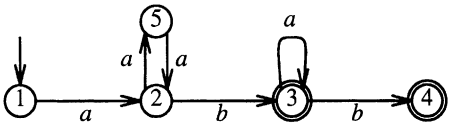
In modern compilers, the lexical analyzers are implemented according to regular-expression specifications. The lexical analyzer partitions a stream of characters into groups, called tokens.[1] Lexical ambiguities arise when a sequence of characters may be partitioned in more than one way. For instance, the six-character string "123456" may be considered as an integer of six digits or six integers of 1 digit each, according to common regular-expression specifications. Intuitively, the formal view, finding a longest match, is more natural and more reasonable.

The traditional model of a lexical analyzer is a Moore automaton, which can be made deterministic with the subset-construction technique [1]. However, due to the look-ahead behavior of a lexical analyzer, a Mealy machine is a better model of a lexical analyzer [2]. Lexical ambiguities arise because the Mealy automaton underlying the lexical analyzer is, in general, *nondeterministic*. Furthermore, there is no way to make it deterministic in general. For example, in Fig. 1(a), two classes of tokens, σ and τ, are defined. A (deterministic) Moore automaton [1] corresponding to the

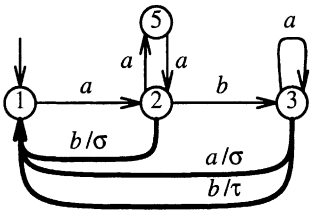
(a) A scanner specification

TOKEN $\sigma = a(aa)^*ba^*$
TOKEN $\tau = a(aa)^*ba^*b$

(b) the (deterministic) Moore automaton M for the scanner specification



(c) the (nondeterministic) Mealy automaton for the scanner specification



(d) the automaton $M_{1,a}$

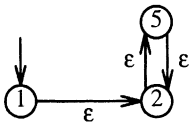


Fig. 1. A scanner specification and its Moore and Mealy automata.

[1] In our view, white spaces and comments also constitute tokens; only these whitespace tokens are discarded rather than transmitted to the parser.

two tokens is shown in Fig. 1(b).[2] This Moore automaton is constructed with standard techniques discussed in most compiler textbooks [3,4]. An equivalent (nondeterministic) Mealy automaton is shown in Fig. 1(c). Notice the three bold arrows that carry output tokens. The Mealy automaton is nondeterministic in that there are two outgoing edges from state 2 labeled *b*. Similar situations hold for state 3. (Fig. 1(d) will be explained in a later section.) When the stream of characters "*abab*" is fed into the automaton in Fig. 1(c), either the two tokens $\sigma\sigma$ or a single token τ may be produced.

The longest-match rule is generally adopted to enforce determinism on a nondeterministic Mealy automaton [2]. The longest-match rule dictates that the next token is the one that contains the most number of input characters. Though the rule is applicable in many situations, it may not always yield the desired results. For instance, consider the string "≫" in C++ programs [5]. In the program fragment "$out \gg f\,f$", the string "≫" should be considered as a single *redirection* token whereas in the program fragment "$foo < bar < buzz \gg$", the string "≫" should be interpreted as two separate, consecutive *greater-than* tokens. Upon encountering the string "≫", the lexical analyzer needs to consult the parser to check the context in which the string "≫" occurs. For a second example, note that, in Modula-2 [6], integers, such as "10", real numbers, such as "10.", and the range symbol ".." are all allowed tokens. For the string "10..20", the partitioning yields the three tokens "10.", ".", and "20" if the longest-match rule is observed strictly. But a correct partitioning in this case should be the three tokens "10", "..", and "20". For a third example, consider the string "$a'(' b',' c')$" in an Ada program. Partitioning according to the longest-match rule yields the tokens: "a", "$'($", "*b*", "$',$", "*c*", "$'$", and "$'$", and ")". A correct partitioning should be "*a*", "$'$", "(", "$'b'$", ",", "$'c'$", and ")". Similar parser errors will occur in the following C++ program fragments: "i := j +++++ k" and "i := j ++k" if the longest-match rule is strictly obeyed. We conclude that misinterpretations by the longest-match rule occur very frequently in practice. Since there are many such misinterpretations in lexical analysis, it is necessary to have a technique that determines when the longest-match rule can be safely applied. In this paper, we propose such a technique.

It is tempting to conjecture that the longest-match rule is not applicable *whenever* a token is a prefix of another token. This is not necessarily so. For example, both "<" and "≪" are tokens in the C++ language. Since no two consecutive "<" tokens can appear in any C++ programs, two consecutive "<" characters are always interpreted as a single "≪" token. In this case, the longest-match rule is applicable.

One naturally will ask when the longest-match rule can be safely applied. In the past, it depends on the experience of the language designer or the compiler implementor to answer this question. However, this is an unreliable approach when a new language is designed or implemented. The longest-match rule is so common that a language designer or a compiler implementor frequently overlooks the subtle implications caused by the longest-match rule in the design or implementation. The consequence is either a flawed language design or a deficient implementation. Thus, it is important to have a method that automatically checks the applicability of the longest-match rule and warns the compiler implementor about the potential problems in the use of the longest-match rule. Such a technique is particularly indispensable in automatic generators of language translation systems, such as lex [7], Eli [8], and PCCTS [9], because, without the technique, the generated lexical analyzers

[2] Our model of Moore automata is slightly different from the one defined in [1]: our Moore automaton always transits to the initial state immediately after a token associated with an accepting state is emitted.

can only blindly apply the longest-match rule. However, blind application of the longest-match rule only results in erroneous lexical analyzers.

A necessary condition for ambiguities is that a token is a prefix of another. A naive approach is to post a warning message whenever such a condition arises. This amounts to giving up the longest-match rule. However, past experience shows that the longest-match rule is applicable in most cases. It would be better if warning messages are issued only when ambiguity will actually occur. We propose such an approach to detecting the ambiguity occurrences more precisely.

Our approach makes use of the context-free grammars underlying the parsers. Ambiguities are not declared prematurely whenever there is more than one partitioning of a sequence of tokens. Rather, ambiguities are declared only if there is more than one partitioning, among the many plausible partitionings, that may be part of sentences derived from the context-free grammar. To be more specific, suppose that an input stream of characters may be partitioned into two different sequences of tokens $\sigma_1, \sigma_2, \ldots, \sigma_j$ and $\tau_1, \tau_2, \ldots, \tau_k$ according to a lexical specification. This could be an ambiguity for the lexical analyzer. However, if only one of the two sequences can form a valid sentence, the lexical analyzer is forced to choose that sequence—there is no ambiguity in this case.

Because it is impractical for the lexical analyzer to examine the whole input stream in deciding the next token, the lexical analyzer is constrained to read only as many characters that may constitute a token as possible. The input stream beyond what was already read is assumed to contain any characters. Under this assumption, it suffices to consider alternative partitionings of a sequence of characters that constitute a token τ, rather than alternative partitionings of the whole input stream. The single token τ, of course, could be part of a valid sentence. Therefore, we examine whether any alternative partitioning of the sequence of characters for τ could also be part of a valid sentence. If so, the language designer or the compiler implementor is warned of the potential ambiguity. Otherwise, the longest-match rule can be safely adopted.

The remainder of this paper is organized as follows. Section 2 presents the overview of our method. In Section 3, we propose a set of axioms to compute the possible sequences of output tokens of a finite automaton when elements of a regular set are used as input to the finite automaton. The difficulty in such computation lies in the Kleene star operator in the regular expressions. To solve this difficulty, Section 4 defines a "macro" automaton derived from the original automaton and solves a set of equations related to the macro automaton. In Section 5, we discuss how to decide whether a regular set and a context-free language intersect with an iterative algorithm. The complete detection algorithm is summarized in Section 6. The last section concludes this paper and discusses related work.

2. Overview of the detection method

In order to detect lexical ambiguities, a lexical specification, written in regular expressions, is transformed into a deterministic Moore automaton M [3,4]. There is a unique initial state and one or more accepting states in the Moore automaton. In order to simplify the following presentation, we will assume that different accepting states accept different classes of tokens.

Lexical ambiguities may arise when one accepting state can reach another (not necessarily distinct) accepting state via a nonnull path. An example is a path from states 3 to 4 in Fig. 1(b). Given two accepting states s and t of a Moore automaton M, consider a path P that starts from the initial state,

passes state s and reaches state t. On the sequence of input characters corresponding to the path P, M moves from the initial state to state t, emits the token τ accepted by state t, and returns to the initial state (if the longest-match rule is adopted). Alternatively, M may move from the initial state to state s, emit the token σ accepted by state s, and scan the remaining part of the input from the initial state. When scanning the remaining input, M may produce other tokens. If the sequence of alternative output tokens may not be part of any sentence accepted by the parser, the longest-match rule does produce the desired output—the single token τ. On the other hand, when the sequence of alternative output tokens may become part of a sentence accepted by the parser, ambiguity will arise. Thus, we need to examine the accepting-to-accepting paths and determine the sequences of output tokens when M scans input corresponding to these paths, and tests whether the sequences of output tokens of M can be embedded in sentences accepted by the parser.

Define the *look-ahead set* as the set of all accepting-to-accepting paths. Given two accepting states s and t of a Moore automaton M, the set of all paths from s to t is a regular set, which may be described by a regular expression. For the sake of computing alternative output tokens, each such regular expression is annotated with the token class accepted by the state s. Thus, elements of the look-ahead set are pairs (σ, la), where σ is a token class and la is a regular expression of the input characters, called the *look-ahead expression*.

To compute the look-ahead expression from state s to state t, we identify the subgraph of the state-transition graph of M induced by all the states that are reachable from state s. This subgraph is also a finite automaton, in which state s is the initial state and t is the (only) accepting state. Then this new finite automaton is converted back to a regular expression, which is the required look-ahead expression. The conversion of a finite automaton to a regular expression is by a technique very similar to the one discussed in Section 4 [10].

The second step of the detection method is, for each pair (σ, la) of the look-ahead set, to determine the sequences of tokens produced by the automaton M when M scans strings of input characters satisfying the regular expression la. Since the look-ahead expression la may represent an infinite number of strings, it is not feasible to run M on the strings one by one. We propose a set of axioms to compute this set of sequences of output tokens in Section 3. It turns out that this set of sequences of tokens produced by M is also a regular set. Call this set the *alternative output sequences* corresponding to the pair (σ, la), denoted by $AOS(\sigma, la)$.

Elements of $AOS(\sigma, la)$ are potential sequences of output tokens if the longest-match rule is *not* applied. If any element of $AOS(\sigma, la)$ may be part of a sentence derivable from the context-free grammar underlying the parser, there are potential lexical ambiguities. Only in such case will the language designer or the compiler implementor be warned about this potential ambiguity.

Our next task is to determine whether any element of a regular set $AOS(\sigma, la)$ can be part of a sentence of a context-free language. Let Λ be the regular expression denoting the set $AOS(\sigma, la)$. Let Γ be the regular set defined by the expression $V^*\Lambda V^*$, where V is the set of all possible tokens. Let L denote the context-free language specified by the context-free grammar underlying the parser. We solve the above problem by determining whether the intersection of the regular set Γ and the context-free language L is an empty set or not. If Γ and L do not intersect, we may safely adopt the longest-match rule. Otherwise, there is a string of input characters that can be partitioned into several distinct sequences of tokens two or more of which may be part of sentences derivable from a context-free grammar. We use an iterative algorithm to solve the intersection problem. The details are in Section 5.

3. Computing sequences of output tokens

In this section, we will show how to compute the sequence of tokens produced by a (deterministic) Moore automaton when the automaton scans a string of input characters that is an element of a regular set. When the regular set is finite, the Moore automaton can scan strings of the regular set one by one and collects the results. The difficulty lies in that the regular set may be infinite. We use eight axioms and an algorithm to compute the set of all possible sequences of output tokens. The following result shows that this set of all possible sequences of output tokens is also a regular set, whose vocabulary is the set of all tokens.

In order to simplify the following presentation, we assume that the Moore automaton is deterministic. Let M be a deterministic Moore automaton (M will also denote the state-transition function of the automaton). Let a be an input character and A and B be regular expressions of input characters. Let q_0 denote the initial state of M and $Token(M(q,a))$ denote the output token associated with state $M(q,a)$. Let X and Y be sets of pairs of the form $[q,\alpha]$,[3] where q is a state of M and α is a regular expression of output tokens (not input characters). Intuitively, q denotes the state of M at a certain point of time and α denotes the set of accumulated sequences of output tokens at that time.

First, we define the *combined transition and output function* $\oplus$ of the automaton M on a regular expression by the following eight axioms:

(1) $\{[q,\alpha]\} \oplus a = \{[M(q,a),\alpha]\} \cup \{[q_0, \alpha \cdot Token(M(q,a))]\}$ if $M(q,a)$ is an accepting state
$= \{[M(q,a),\alpha]\}$ if $M(q,a)$ is not an accepting state
$= \varnothing$ if $M(q,a)$ is *error*,
(2) $X \oplus \varepsilon = X$,
(3) $X \oplus (A \cdot B) = (X \oplus A) \oplus B$,
(4) $X \oplus (A \mid B) = (X \oplus A) \cup (X \oplus B)$,
(5) $X \oplus A^* = limit\ \{Y_1, Y_2, \ldots\}$, where $Y_1 = X, Y_{n+1} = Y_n \cup Y_n \oplus A$,
(6) $\varnothing \oplus A = \varnothing$,
(7) $(X \cup Y) \oplus A = (X \oplus A) \cup (Y \oplus A)$,
(8) $\{[q,\alpha],[q,\beta]\} = \{[q,(\alpha \mid \beta)]\}$,

Axioms 1, 2, 3, 4, 6, and 7 are quite straightforward. Axiom 1 describes the state-transition and output behavior of M. Note that M transits to the initial state immediately after a token is emitted. Note also that M also reserves the right *not* to emit a token even if it enters an accepting state. Axiom 8 combines pairs with the same state into a single pair. Axiom 5 reflects the fact that the Kleene star operation denotes zero or more repetitions of a regular expression. The *limit* operation in the fifth axiom is defined as follows: given two regular expressions α_1 and α_2, we say $\alpha_1 \leqslant \alpha_2$ if the regular set defined by α_1 is a subset of that defined by α_2. Given two pairs $[q_1,\alpha_1]$ and $[q_2,\alpha_2]$, we say $[q_1,\alpha_1] \leqslant [q_2,\alpha_2]$ if $q_1 = q_2$ and $\alpha_1 \leqslant \alpha_2$. Given two sets of pairs $U = \{[p_i,\alpha_i] \mid i = 1,2,\ldots\}$ and $V = \{[q_j,\beta_j] \mid j = 1,2,\ldots\}$, we say $U \leqslant V$ if (1) for every pair $[p_i,\alpha_i]$ in U, there is a pair $[q_j,\beta_j]$ in V such that $[p_i,\alpha_i] \leqslant [q_j,\beta_j]$, or (2) there is a set W such that $U \leqslant W$ and $W = V$ (based on Axiom 8). Given a sequence of sets of pairs $V_1, V_2, \ldots$, we say $V = limit\ \{V_1, V_2, \ldots\}$ if (1)

[3] We use the square brackets $[\ldots]$ for pairs concerning the sequence of output tokens. We reserve the round brackets $(\ldots)$ for pairs in the look-ahead set.

$V_i \leqslant V$ for all i, and (2) for any pair $[q, \beta]$ in V, and any element β' of the regular set defined by β, there exists a set V_k in the sequence of the sets of pairs such that $\{[q, \beta']\} \leqslant V_k$. We can easily prove the following lemma according to the above definition.

Lemma. *Given two sets of pairs U and V, if $U \leqslant V$ then $U \cup V = V$.*

Intuitively, *limit* $\{Y_1, Y_2, \ldots\}$ is equal to $Y_1 \cup Y_2 \cup \ldots$. But this definition is useless since, in Axiom 5 above, it is already known that $Y_n \leqslant Y_{n+1}$ for all n. Note that, in general, for any m, there exists $n > m$ such that $Y_m \neq Y_n$. Therefore, *limit* $\{Y_1, Y_2, \ldots\} \neq Y_n$, for any n. Direct computation of $Y_1, Y_2, \ldots$ effectively enumerates the elements of a regular set. This implies that direct computation of the sets $Y_1, Y_2, \ldots$ one by one may not always result in the desired *limit*. We will present a method to compute the *limit* in the next section.

$X \oplus A^*$ is a fixed point of a monotone function on a lattice, defined as follows: consider a deterministic Moore automaton M. Define $\Omega =$ the set of all output tokens. Define $E = \{[q, \alpha] \mid q$ is a state of M and α is regular expression over $\Omega\}$. Let 2^E denote the powerset of E. Then $(2^E, \leqslant)$ is a partially ordered set. Define, for $U, V \in 2^E$

$$U \sqcup V = \text{lub}(U, V) = U \cup V,$$

$$U \sqcap V = \text{glb}(U, V).$$

We can show that $(2^E, \sqcup, \sqcap)$ is a lattice. Unfortunately, it is not a complete lattice. On this lattice, we may define a (monotone) function: $f_A = \lambda U.\ U \cup (U \oplus A)$. Note that $f_A^n(X) = Y_{n+1}$ defined in Axiom 5. The limit *limit* $\{Y_1, Y_2, \ldots\}$ is a fixed point of f_A. The fixed point corresponds to the equation $(X \oplus A^*) \oplus A = X \oplus A^*$.

Based on the algorithm in the next section, we assert that the *limit* always exists for any X and is unique up to equivalent regular expressions and the application of Axiom 8. Furthermore, the computation of the *limit* in Axiom 5 will halt in finite amount of time and the *limit* may be represented as a *finite* set of pairs (that is, the number of pairs in the *limit* is finite though each pair may denote an infinite regular set).

A pair $[q, \beta]$ contains both the information regarding the final state and the information regarding the sequences of output tokens. Since we are only interested in the sequences of output tokens, we use the operation *collect* to collect all the potential output, which is defined as follows: consider a finite set of pairs $X = \{[p_i, \alpha_i] \mid i = 1, 2, \ldots, k\}$. For each pair $[p_i, \alpha_i]$ in X, let $\{\beta_{i,j} \mid j = 1, 2, \ldots, mi\}$ be the set of tokens associated with the accepting states that are reachable from state p_i in M's state-transition graph. Let δ_i denote the regular expression $\alpha_i(\beta_{i,1} \mid \beta_{i,2} \mid \ldots \mid \beta_{i,mi})$. Then $collect(X) = (\delta_1 \mid \delta_2 \mid \ldots \mid \delta_k)$.

Our purpose in this section is, given a pair (σ, la) of the look-ahead set, to compute the set of the alternative output sequences corresponding to the pair (σ, la), that is, $AOS(\sigma, la)$. Note that $AOS(\sigma, la) = collect(\{[q_0, \sigma]\} \oplus la)$, where q_0 is the initial state of M. An example of computing AOS is included in the next section.

4. Computing $X \oplus A^*$ in Axiom 5

In this section, we discuss in detail how to compute the *limit* operation used in Axiom 5 listed in the previous section. Our purpose is to compute $X \oplus A^*$, where X is a finite set of pairs of the form

```
Algorithm: Construction of M_{q,A}
create a state q in M_{q,A}
WL := { q }
while WL is not empty do
   remove a state from WL, call it state p
   X := { [p, ε] } ⊕ A
   for each pair [p_i, α_i] in X do
      if p_i is not already in M_{q,A} then
         create a new state p_i in M_{q,A}
         WL := WL ∪ { p_i }
      end if
      create an edge p ⟶ p_i in M_{q,A} and label it α_i
   end for
end while
```

Fig. 2. Construction of M_q, A.

$[q, \beta]$ and A is a regular expression. As was discussed in the previous section, it is not feasible to compute the sequence $X, X \oplus A, (X \oplus A) \oplus A, \ldots$, etc. Our solution is to construct a new automaton and then solve equations of regular expressions on this new automaton.

Though X is a set of pairs, due to Axiom 7, we may consider one pair at a time. In order to compute $\{[q, \beta]\} \oplus A^*$ on a given finite automaton M, we first construct a new macro automaton $M_{q,A}$. $M_{q,A}$ is a nondeterministic Mealy automaton in which every transition corresponds to an aggregate of transitions of the original automaton M on an element of the regular set defined by A. The macro automaton $M_{q,A}$ is constructed with the work-list algorithm shown in Fig. 2. Initially, the work list contains a single state, that is, state q. Then a state p is picked up from the work list and $\{[p, \varepsilon]\} \oplus A$ (ε is the empty string) is computed inductively by the axioms in the previous section. Let $\{[p, \varepsilon]\} \oplus A = \{[p_i, \alpha_i] \mid i = 1, 2, \ldots, k\}$. For each pair $[p_i, \alpha_i]$, create a new state p_i in $M_{q,A}$ (if one does not already exist) and create a new transition from state p to state p_i, which is labeled with the output sequence α_i. If p_i is a newly created state, then state p_i is added to the work list. The above step is repeated for each state added to the work list. This work-list algorithm terminates when the work list becomes empty. The initial state of $M_{q,A}$ is the state q. Note that the edges of $M_{q,A}$ are labeled with the output token sequences, rather than the input characters (since input is always the regular expression A). Note that the work-list algorithm must terminate because there are only a finite number of states in M and each state is processed at most once. Fig. 2 is a recast of the work-list algorithm.

After constructing the macro automaton $M_{q,A}$, we will calculate the output regular expression for each state of $M_{q,A}$. The calculation is performed by solving a set of equations. For each state p of $M_{q,A}$, there is a variable P representing the regular expression associated with state p. Let $\{r_i \rightarrow {}^{\alpha_i} p \mid i = 1, 2, \ldots, m\}$ be the set of edges entering state p (the notation $r_i \rightarrow {}^{\alpha_i} p$ denotes an edge from r_i to p labeled α_i). If state p is not the initial state of $M_{q,A}$, the variable P is defined by the equation $P = R_1\alpha_1 \mid R_2\alpha_2 \mid \ldots \mid R_m\alpha_m$. For the initial state q of $M_{q,A}$, let $\{r_i \rightarrow {}^{\alpha_i} q \mid i = 1, 2, \ldots, n\}$ be the set of edges entering state q. The equation defining q's variable is $Q = \beta \mid R_1\alpha_1 \mid R_2\alpha_2 \mid \ldots, \mid R_n\alpha_n$. This set of mutually recursive equations can be solved with a backward substitution strategy.

After obtaining the regular expression P for each state p of automaton $M_{q,A}$, the set $\{[p,P] \mid p$ is a state of $M_{q,A}$, and P is the regular expression associated with state $p\}$ is $\{[q,\beta]\} \oplus A^*$.

Example. Consider the scanner specification in Fig. 1(a). Two classes of tokens are specified: σ and τ. The scanner specification is converted to the deterministic Moore automaton M shown in Fig. 1(b), where state 3 recognizes the token class σ and state 4 recognizes the token class τ. There is an accepting-to-accepting path from states 3 to 4. Hence, the look-ahead set is $\{(\sigma, a^*b)\}$. Then we need to compute $AOS(\sigma, a^*b) = collect(\{[1,\sigma]\} \oplus a^*b)$. First we compute $\{[1,\sigma]\} \oplus a^*$.

To compute $\{[1,\sigma]\} \oplus a^*$, we need to construct the automaton $M_{1,a}$. Initially, there is a state 1 in $M_{1,a}$. Then the following computation is performed, based on axioms in Section 3:

$$\{[1,\varepsilon]\} \oplus a = \{[2,\varepsilon]\},$$

$$\{[2,\varepsilon]\} \oplus a = \{[5,\varepsilon]\},$$

$$\{[5,\varepsilon]\} \oplus a = \{[2,\varepsilon]\},$$

Thus, two more states 2 and 5 are created in $M_{1,a}$, together with the transitions. Fig. 1(d) shows the automaton M_1, a. Let P, Q, and R be the regular-expression variables for states 1, 2, and 5, respectively. Then we may set up the following equations:

$$P = \sigma,$$

$$Q = P\varepsilon \mid R\varepsilon,$$

$$R = Q\varepsilon.$$

Solving these equations, we obtain the "least" solution $P = Q = R = \sigma$. (The least solution is obtained by considering the equal sign $=$ in the equations as the derivation sign $\rightarrow$ in production systems.) Therefore, $\{[1,\sigma]\} \oplus a^* = \{[1,\sigma],[2,\sigma],[5,\sigma]\}$. Call this set X. Next we may compute $X \oplus b = \{[3,\sigma],[1,\sigma\sigma]\}$. Therefore, $\{[1,\sigma]\} \oplus a^*b = \{[3,\sigma],[1,\sigma\sigma]\}$. This means that, if elements of the regular set defined by the look-ahead expression a^*b are fed into the automaton M, either M will end up in state 3 with no output (the token σ is accounted for previous input characters) or in state 1 with a single output token σ. For example, suppose that the input is *abaab*, of which the suffix *aab* is the look-ahead string. The automaton in Fig. 1(b) may exhibit three kinds of possible behaviors: (1) it produces a token τ and returns to the initial state 1; (2) it emits a token σ and halts at state 3; or (3) it emits two tokens $\sigma\sigma$ and returns to state 1. Case (1) above occurs when the longest-match rule is adopted. Cases (2) and (3), corresponding to the above computation, indicate the possible states of the automaton if the longest-match rule is *not* adopted.

Finally, the *collect* operation is performed. In the automaton in Fig. 1(b), both state 1 and state 3 can reach the two accepting states 3 and 4. Each of the two tokens σ and τ is appended to each of the two expressions σ and $\sigma\sigma$. Hence, we reach the result $AOS(\sigma, a^*b) = \{\sigma\sigma, \sigma\tau, \sigma\sigma\sigma, \sigma\sigma\tau\}$, which corresponds to the regular expression $\Lambda = (\sigma \mid \sigma\sigma)(\sigma \mid \tau)$. The regular set defined by Λ is tested for containment in sentences derived by the context-free grammar underlying the parser. Though the regular set contains only four elements and can be tested easily, we will propose a general solution for arbitrary regular sets in the next section.

Before we prove that the above is correct, we first claim, without a proof, that the set of equations of regular expressions is solved correctly. Specifically, we claim the following two lemmas.

Lemma. *For any pair* $\{[r,R]\} \leqslant \{[q,\beta]\} \oplus A^n$, *for some* n, *the state* r *is included in* $M_{q,A}$. *Furthermore, let* R' *be the regular expression associated with state* r *obtained by solving the set of equations. Then* R *is an element of the regular set defined by* R'.

Lemma. *Let* r *be a state of* $M_{q,A}$ *and* R' *be the regular expression associated with state* r. *Let* R *be an element of the regular set defined by* R'. *Then* R *corresponds to a* finite *path from the initial state to state* r *in* $M_{q,A}$.

Below we show that the computation of $\{[q,\beta]\} \oplus A^*$ is correct. Specifically, we need to prove the following theorem.

Theorem. *The set* $\{[p,P] \mid p$ *is a state of* $M_{q,A}$, *and* P *is the regular expression associated with state* $p\}$ *computed by the method in this section is equal to* $\{[q,\beta]\} \oplus A^*$ *defined in the previous section.*

Proof. Let X denote the set $\{[q,\beta]\}$. Let Y denote the set $\{[p,P] \mid p$ is a state of $M_{q,A}$, and P is the regular expression associated with state $p\}$ computed by the method in this section. Let $Y_1 = X$ and $Y_{n+1} = Y_n \cup Y_n \oplus A$. We need to show that $Y = limit\ \{Y_1, Y_2, \ldots\}$.

First we show that $Y_n \leqslant Y$, for all n. Since $Y_n \leqslant Y_{n+1}$ for all n, consider any set of a pair $\{[r,R]\}$ such that $\{[r,R]\} \leqslant Y_{n+1}$ but $\{[r,R]\} \not\leqslant Y_n$. Intuitively, $\{[r,R]\} \leqslant \{[q,\beta]\} \oplus A^n$. Based on the construction of $M_{q,A}$, the state r must be a state in the macro machine $M_{q,A}$. Let R' be the regular expression associated with state r obtained by solving the set of equations. Since we assume that the set of equations of regular expressions are solved correctly, R is an element of the regular set defined by R'. Hence, $\{[r,R]\} \leqslant Y$. This implies that $Y_n \leqslant Y$, for all n.

Next we need to show that for any pair $[r,R']$ in Y, and any element R of the regular set defined by R', there exists a set Y_k such that $\{[r,R]\} \leqslant Y_k$. Since we assume that the set of equations is solved correctly, R must correspond to a finite path from the initial state to state r in $M_{q,A}$. Let k be the length of the path. Then $\{[r,R]\} \leqslant Y_k$. □

Based on the correctness of solving the set of equations induced by the macro automaton $M_{q,A}$, we may prove the correctness of the axioms of the previous section.

Theorem. *The set of the* 8 *axioms of Section* 3 *correctly computes* $[q,\alpha] \oplus A$ *for any regular expression* A *in finite amount of time.*

Proof. By structural induction [11] on the syntax of the regular expressions. Details are omitted. □

5. Determining the intersection of a regular language and a context-free language

Our next task is to determine whether any element of a regular set $AOS(\sigma, la)$ can be part of a sentence of the context-free language accepted by the parser. Let Λ be the regular expression denoting the set $AOS(\sigma, la)$. Let Γ be the regular set defined by the expression $V^*\Lambda V^*$, where V is the set of all possible tokens. Let L denote the context-free language accepted by the parser. We solve the above problem by determining whether the intersection of the regular set Γ and the

context-free language L is an empty set or not. Note that both Γ and L are based on the same vocabulary, the set of all tokens.

Let M be a finite automaton corresponding to the regular set Γ and G be the context-free grammar underlying the parser. We define an operator $\otimes$ that takes two arguments: a set of states (of M) and a string of terminals and nonterminals (of G). The result of $\otimes$ is a set of states (of M). The notation $Q \otimes \alpha = Q'$ means that, starting from a state of Q, M will reach a state of Q' on an input string that is derivable from the string α (according to the production rules of G). With the $\otimes$ operator, the regular set and the context-free language intersect if and only if the set $\{q_0\} \otimes S$ contains an accepting state of M, where q_0 is the initial state of M and S is the start symbol of the grammar of G.

The $\otimes$ operator may be viewed as an extension of the transition function of the automaton to strings of terminals and nonterminals of the context-free grammar. The operator $\otimes$ is defined by the following five axioms:

(1) $\{q\} \otimes a = \{q' \mid M$ moves from state q to state q' on input a, where a is a terminal of G$\}$.
(2) $\{q\} \otimes A = \{q' \mid M$ moves from state q to state q' on a string of terminals α, where A is a nonterminal of G and $A \rightarrow {}^*\alpha\}$.
(3) $Q \otimes \alpha\beta = (Q \otimes \alpha) \otimes \beta$, where α and β are strings of terminals and nonterminals.
(4) $(Q_1 \cup Q_2) \otimes \alpha = (Q_1 \otimes \alpha) \cup (Q_2 \otimes \alpha)$, where α is a string of terminals and nonterminals.
(5) $\varnothing \otimes \alpha = \varnothing$.

The operator $\otimes$ is similar to the $\oplus$ operator defined in the previous section. They differ in two aspects: (1) The second argument of the $\otimes$ operator is a context-free language whereas that of the $\oplus$ operator is a regular set; and (2) The $\otimes$ operator does not compute output components.

To find $\{q\} \otimes a$ for a state q of M and a terminal a of G, we may simply examine the transition table of M. To find $\{q\} \otimes A$, where A is a nonterminal of G, we establish a set of equations and solve the equations iteratively. Let $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_m$ be the set of all the A-productions in G. Then $\{q\} \otimes A = (\{q\} \otimes \alpha_1) \cup (\{q\} \otimes \alpha_2) \cup \ldots \cup (\{q\} \otimes \alpha_m)$.

There is one such equation for each state q of M and each nonterminal A of G. The above set of equations can be solved by an iteration algorithm. Initially, assume $\{q\} \otimes A = \varnothing$ for each q and each A. Then we repeatedly evaluate the set of equations until a stable solution is reached. The iteration algorithm is shown in Fig. 3, where an expression, such as $\{q\} \otimes A$, is treated as a variable.

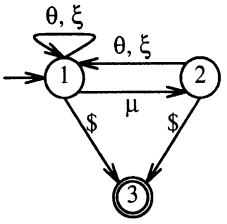
Example. Fig. 4(a) is a deterministic automaton corresponding to the regular set Γ. State 1 is the initial state. Fig. 4(b) is a grammar defining the context-free language L. Fig. 4(b) is the $\otimes$ operator applied to the automaton in Fig. 4(a) and the context-free grammar in Fig. 4(b). For the sake of brevity, we have omitted the set symbols $\{\ldots\}$. From the definition of the $\otimes$ operator, we obtain the following six equations:

$$\begin{aligned}\{1\} \otimes T &= (\{1\} \otimes \mu T \mu) \cup (\{1\} \otimes \xi T \xi) \cup (\{1\} \otimes \theta) \\ &= (\{1\} \otimes \mu \otimes T \otimes \mu) \cup (\{1\} \otimes \xi \otimes T \otimes \xi) \cup (\{1\} \otimes \theta) \\ &= (\{2\} \otimes T \otimes \mu) \cup (\{1\} \otimes T \otimes \xi) \cup \{1\},\end{aligned}$$

```
Algorithm: Iteration
Given a set of k equations x_i = f_i(...), for i = 1, 2, ..., k
for i := 1 to k do x_i := ∅ end for
repeat
   for i := 1 to k do
      x_i' := f_i(...)
   end for
   stable := true
   for i := 1 to k do
      if x_i ≠ x_i' then
         x_i := x_i'
         stable := false
      end if
   end for
until stable
```

Fig. 3. The iteration algorithm.

(a) An automaton Γ



(b) A context-free grammar L

P1: $S \longrightarrow T\ \$$
P2: $T \longrightarrow \mu T \mu$
P3: $T \longrightarrow \xi T \xi$
P4: $T \longrightarrow \theta$

(c) the $\otimes$ operator

⊗	μ	ξ	θ	$	T	S
1	2	1	1	3	1,2	3
2	∅	1	1	3	1	3
3	∅	∅	∅	∅	∅	∅

Fig. 4. An automaton, a context-free grammar, and the $\otimes$ operator.

$$\{2\} \otimes T = (\{2\} \otimes \mu T \mu) \cup (\{2\} \otimes \xi T \xi) \cup (\{2\} \otimes \theta)$$
$$= (\{2\} \otimes \mu \otimes T \otimes \mu) \cup (\{2\} \otimes \xi \otimes T \otimes \xi) \cup (\{2\} \otimes \theta)$$
$$= (\varnothing \otimes T \otimes \mu) \cup (\{1\} \otimes T \otimes \xi) \cup \{1\},$$

$$\{3\} \otimes T = (\{3\} \otimes \mu T \mu) \cup (\{3\} \otimes \xi T \xi) \cup (\{3\} \otimes \theta).$$
$$= (\{3\} \otimes \mu \otimes T \otimes \mu) \cup (\{3\} \otimes \xi \otimes T \otimes \xi) \cup (\{3\} \otimes \theta)$$
$$= (\varnothing \otimes T \otimes \mu) \cup (\varnothing \otimes T \otimes \xi) \cup \varnothing,$$

$$\{1\} \otimes S = \{1\} \otimes T\$$
$$= \{1\} \otimes T \otimes \$,$$

$$\{2\} \otimes S = \{2\} \otimes T\$$
$$= \{2\} \otimes T \otimes \$,$$

$$\{3\} \otimes S = \{3\} \otimes T\$$
$$= \{3\} \otimes T \otimes \$.$$

Let x_1, x_2, x_3, x_4, x_5, and x_6 denote the six terms: $\{1\} \otimes T, \{2\} \otimes T, \{3\} \otimes T, \{1\} \otimes S, \{2\} \otimes S, \{3\} \otimes S$, respectively. After some simplification, we get the following six equations:

$$x_1 = (x_2 \otimes \mu) \cup (x_1 \otimes \xi) \cup \{1\},$$
$$x_2 = (x_1 \otimes \xi) \cup \{1\},$$
$$x_3 = \varnothing,$$
$$x_4 = x_1 \otimes \$,$$
$$x_5 = x_2 \otimes \$,$$
$$x_6 = x_3 \otimes \$.$$

Initially, assume that $x_1 = x_2 = x_3 = x_4 = x_5 = x_6 = \varnothing$. We repeatedly evaluate the six equations. After three iterations, we reach a stable solution: $\{1\} \otimes S = \{3\}$, $\{2\} \otimes S = \{3\}$, $\{3\} \otimes S = \varnothing$, $\{1\} \otimes T = \{1, 2\}$, $\{2\} \otimes T = \{1\}$, and $\{3\} \otimes T = \varnothing$.

Because $\{1\} \otimes S = \{3\}$, which means that the automaton moves from state 1 (the initial state) to 3 (an accepting state) on a sentence derived from the start symbol S, the regular set and the context-free language do have common elements. An example is the string $\mu\xi\theta\xi\mu\$$.

The iteration algorithm in Fig. 3 always halts due to its accumulative nature. That the solution is correct can be proved by an inductive reasoning, as follows: the addition of a state q' into the solution of $\{q\} \otimes A$ can be traced backward eventually to a transition $q_1 \rightarrow^a q_2$ in the finite automaton, where a is a token. Thus, we can construct a string of terminals that is derivable from A and the finite automaton moves from state q to state q' on that string. Conversely, if the finite automaton moves from state q to state q' on a string derivable from A, there must be a path (of finite length) from q to q' on the state-transition graph of the finite automaton that is labeled with the string. Thus, q' must be eventually added to $\{q\} \otimes A$. Specifically, the following theorem is asserted:

Theorem (Correctness of the iteration algorithm). *$q' \in \{q\} \otimes A$ if and only there is a string derivable from the (terminal or nonterminal) symbol A according to the context-free grammar on which the finite automaton moves from state q to state q'.*

Yet another problem that we need to address is the independence of the answer on the particular finite automaton and the particular context-free grammar used in the computation. We investigate whether a regular set and a context-free language have common elements. In the iteration algorithm, an arbitrary finite automaton for the regular set and an arbitrary context-free grammar for the context-free language are chosen for setting up the set of equations. It is natural to ask whether we will reach a different answer if different finite automata or different context-free grammars are chosen. Based on the correctness of the iteration algorithm, we may claim that the same result is always reached no matter which finite automaton or context-free grammar is chosen.

```
Algorithm: Detection of lexical conflicts
Let G be the context-free grammar of the programming language under consideration.
R = the set of regular expressions defining tokens (including white spaces and comments) of the language
M = the deterministic Moore automaton that accepts the tokens defined by regular expressions of R
for each pair of accepting states s and t in M do
   if state t is reachable from state s in M's transition diagram then
      σ = the token associated with state s
      la = the regular expression representing all paths from state s to state t
      Compute AOS(σ, la) by the algorithms in Sections 3 and 4.
      Λ = the regular expression for the regular set AOS(σ, la)
      V = the set of all possible tokens
      Γ = the regular set defined by the equation V*ΛV*
      /* Determine whether the regular set Γ and the context-free language
         defined by G have common elements by the algorithm in Section 5. */
      warning = intersection(Γ, G)
      if warning then
         issue a warning message together with the tokens associated with states s and t
      end if
   else if state s is reachable from state t then
      /* This case is handled in the identical way as the above case. */
      /* The details are omitted. */
   end if
end for
```

Fig. 5. The complete detection algorithm.

A classical method for the intersection problem is to integrate the finite automaton of the regular set and the pushdown automaton of the context-free language into a new pushdown automaton. A new context-free grammar can then be derived from the integrated pushdown automaton. Though the $\otimes$ operator did not compute the exact intersection, it provides additional information relating the states of a finite automaton and the nonterminals of a context-free grammar. This information is useful in simplifying the LR parser of the context-free grammar as well as its parser [12].

6. The complete detection algorithm

Fig. 5 is the complete conflict detection algorithm. It first computes the determinisitic Moore automaton for the set of regular expressions for tokens. Then each pair of accepting states of the automaton is examined for alternative partitionings when the longest-match rule is *not* observed. The regular set $AOS(\sigma, la)$ is computed by the algorithms in Sections 3 and 4. Then the algorithm in Section 5 is applied to determine whether any element of $AOS(\sigma, la)$ could be part of a sentence of the programming language. If so, a misinterpretation may potentially arise and hence a suitable warning message is issued. It is this warning message that indicates the precise situation in which the longest-match rule is not applicable.

7. Conclusion and related work

We have identified the applicability problem of the longest-match rule and proposed a solution. The crux of the solution consists of two algorithms: one is to compute the regular set of the sequences of tokens produced by a nondeterministic Mealy automaton when the automaton processes elements of an input regular set. The other is to determine whether a regular set and a context-free language have nontrivial intersection with a set of equations.

The work reported here is a systematic method to detect potential ambiguities in lexical analysis. It aids a language designer to check his/her design of a new language and it helps a compiler writer in applying the longest-match rule. As far as we know, this kind of integrated check (of both the lexical specification and the syntactic specification) has not been studied in details previously. The work reported here may be viewed as a refined longest-match rule. The longest-match rule is widely used in compiler implementation [3,4] and has been studied in details in [13], where the author proposed a new lexical analysis method to solve the look-ahead problem. Most compiler textbooks treat a lexical analyzer as a Moore machine. The *scangen* scanner generator [4] exhibits some flavor of a Mealy machine. A deterministic Mealy-machine model is proposed in [2] for lexical analysis due to the longest-match rule. The work reported in [2] is concerned only with a subclass of automata—the class of finite look-ahead automata; by contrast, the work reported in this paper is applicable to infinite look-ahead automata as well as finite look-ahead ones.

References

[1] Hopcroft JE, Ullman JD. Introduction to automata theory, languages, and computation. Reading, MA: Addison-Wesley, 1979.
[2] Yang W. Mealy machines are a better model of lexical analyzers. Computer Languages 1996;22(1):27–38.
[3] Aho AV, Sethi R, Ullman JD. Compilers: Principles, Techniques, and Tools. Reading, MA: Addison-Wesley, 1986.
[4] Fischer CN, Leblanc Jr RJ. Crafting a compiler with *C*. Reading, MA: Benjamin/Cummings, 1991.
[5] Ellis MA, Stroustrup B. The annotated *C++* reference manual. Reading, MA: Addison-Wesley, 1990.
[6] Wirth N. Programming with modula-2, 2nd corrected ed. New York: Springer, 1983.
[7] Lesk ME, Schmidt E. LEX—a lexical analyzer generator. Computer Science Technical Report 39, Bell Labs., Murray Hill, NJ, 1975.
[8] Gray RW, Heuring VP, Levi SP, Sloane AM, Waite WM. Eli: a complete, flexible compiler construction system. Communications of the ACM 1992;35(2):121–31.
[9] Parr T, Language translation using PCCTS and *C++*: a reference guide. San Jose, CA: Automata Publishing, 1997. A pre-release version is available from ftp://ftp.parr-research.com/pub/pccts/Book/reference.ps.
[10] Aho AV, Ullman JD. The theory of parsing, translation, and compiling: parsing. Englewood Cliffs, NJ: Prentice-Hall, 1972.
[11] Burstall RM. Proving properties of programs by structural induction. The Computer Journal 1969;12(1):41–8.
[12] Yang W. A lattice framework for analyzing context-free languages with applications in parser simplication and data-flow analysis. Journal of Information Science and Engineering 1999;15(2):287–306.
[13] Yang W. On the look-ahead problem in lexical analysis. ACTA Informatica 1995;32:459–76.

Wuu Yang received his B.S. degree in computer science from National Taiwan University in 1982 and the M.S. and Ph.D. degrees in computer science from University of Wisconsin at Madison in 1987 and 1990, respectively. Currently he is a professor in the National Chiao-Tung University, Taiwan, Republic of China. Dr. Yang's current research interests include

Java and network security, programming languages and compilers, and attribute grammars. He is also very interested in the study of human languages and human intelligence.

Chey-Woei Tsay received his B.S. degree in computer science from National Taiwan University in 1982. After receiving a Ph.D. degree in computer science from University of Utah, he joined the Department of Computer Science and Information Management, Providence University, Taiwan, Republic of China. Dr. Tsay's current research interests include network computation, computer graphics, programming languages and compilers, and user interface design.

Lien-Tsai Chan received his M.S. degree in Computer and Information Science from the National Chiao Tung University in 1996. Currently he is a Ph.D. candidate in the same university. His research interests include programming languages, attribute grammars, compilers, programming systems.