Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

J. Heering, P. Klint, J. Rekers

Incremental generation of lexical scanners

Computer Science/Department of Software Technology Report CS-R8761 December

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

Incremental Generation of Lexical Scanners

J. Heering
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

P. Klint
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
Department of Computer Science, University of Amsterdam,
Kruislaan 409, 1098 SJ Amsterdam, The Netherlands

J. Rekers
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

It is common practice to specify textual patterns by means of a set of regular expressions and to transform this set into a finite automaton to be used for the scanning of input strings. In many applications, the costs of this preprocessing phase can be amortized over many uses of the constructed automaton. In this paper new techniques for lazy and incremental scanner generation are presented. The lazy technique postpones the construction of parts of the automaton until they are really needed during the scanning of input. The incremental technique allows modifications to the original set of regular expressions to be made and reuses as many parts of the previous automaton as possible. This is interesting in situations where modifications to the definition of lexical syntax and the use of the generated scanners alternate frequently, for instance, in environments for the interactive development of language definitions.

Key Words & Phrases: program generator, lazy and incremental generation of lexical scanners, finite automata, subset construction.

1987 CR Categories: D.1.2 [**Programming Techniques**]: Automatic programming; D.3.4 [**Programming Languages**]: Processors.

1985 Mathematics Subject Classification: 68N20 [**Software**]: Compilers and generators.

Note: Partial support received from the European Communities under ESPRIT project 348 (Generation of Interactive Programming Environments - GIPE).

Note: This paper will be submitted for publication elsewhere.

Report CS-R8761
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

1. INTRODUCTION

Searching for textual patterns occurs in many areas of computerized information processing such as text editing, query processing, bibliographic searches, linguistic analysis, and lexical analysis. The following approach to the problem of specifying and matching a textual pattern in an input string is frequently used:
(1) specify the pattern by means of a set of regular expressions;
(2) transform the regular expression into a (deterministic or non-deterministic) finite automaton;
(3) minimize the automaton (optional);
(4) use the automaton to perform the actual (and fast) matching of the input string.
This approach has become particularly popular due to the widespread availability of scanner generators such as, e.g., LEX [L75]. In many applications, the preprocessing time needed for the construction and optimization of the automaton can be amortized over many uses of the constructed automaton. In this paper we are interested in applications for which this assumption does not hold.

The applications we are interested in have the following characteristics:
(1) definition and use of regular expressions alternate frequently;
(2) matching has to be fast;
(3) regular expressions change gradually, i.e., they are the result of a series of editing operations.
Characteristic (1) excludes extensive preprocessing for each regular expression since the cost of preprocessing will outweigh the advantage gained (fast matching). It may even happen that all alternatives of the regular expression are considered during preprocessing, but that only a few of them are actually used during matching. Characteristic (2) makes it mandatory to use *some* preprocessing in order to achieve acceptable performance, while characteristic (3) suggests the possibility of *reusing* the result of preprocessing previous regular expressions. This direction will be pursued in this paper.

Applications with the above characteristics occurred in the context of the GIPE project which aims at generating programming environments from formal language definitions [HKKL86]. Language definitions include (among other things) a lexical and a context-free syntax of the language being defined. On the basis of this definition a lexical scanner, a parser, and a syntax-directed editor are generated. If the definition for some language L is being developed interactively, one wants to observe the result of a modification to the definition in the generated L-environment without waiting for a time consuming regeneration of the total environment. This can be achieved if the generation phase is *incremental*, i.e., if only those parts of the L-environment are regenerated that are affected by the change in the L-definition. To achieve this one first of all needs techniques for the incremental generation of lexical scanners and context-free parsers.

A similar problem occurs in languages that allow general, user-defined, syntactic extensions. Texts in such languages cannot be parsed by a fixed scanner and parser. Here too, one would like to have the possibility of incrementally changing the scanner and parser. A typical example is a specification language which allows arbitrary user-defined syntactic notation for the functions in the specification. One such specification language is currently being developed in the GIPE project [HK87].

Another area of application deals with textual search operations in text editors or query systems in which the set of search patterns is gradually changing This may occur, for instance, in queries to a bibliograpic system, where the user refines its search criteria as information becomes available from earlier searches.

In this paper we discuss algorithms for lazy and incremental lexical scanner generation, describe an implementation of these algorithms and present some measurements of their performance. The algorithms to be presented are summarized in Table I.

Construct DFA for given set of regular expressions:	Scan inputstring by means of constructed DFA:	Described in section:
CONSTRUCT	*SCAN*	**2**
L-CONSTRUCT	*L-SCAN*	**3**
MODIFY	,,	**4**
S-MODIFY	,,	**5**

Table I. Overview of algorithms

In section 2, we present the pair *CONSTRUCT/SCAN*. These are existing algorithms for the direct construction of a deterministic finite automaton (DFA) for a set of regular expressions and for the simulation of the constructed automaton.

In section 3, it is shown how a DFA can be constructed in a lazy fashion. *L-CONSTRUCT* only constructs the start state of a DFA (leading to a *partial* DFA, or PDFA for short). Next, *L-SCAN* starts simulating this automaton and gradually expands the PDFA by adding those parts that are needed to scan the input string.

In section 4, the lazy construction and incremental modification of sets of regular expressions is addressed. Given a PDFA (constructed by *L-CONSTRUCT/L-SCAN* or *MODIFY*), major parts of this PDFA are reused when constructing a new PDFA for a modified set of regular expressions.

The technique of incremental construction and modification is refined in section 5: before changing the old PDFA into a new one, an attempt is made to recognize (by means of *L-SCAN*) all literal regular expressions using the old automaton. This avoids adding new states to the new automaton for regular expressions that were already recognized by the old one.

In section 6, extensions to the above algorithms are given for the case that regular expressions may contain character classes. An overview of the implementation of an incremental scanner generator is given in section 7. Some performance measurements of this scanner generator are given in section 8. Section 9 concludes the paper and discusses the results achieved.

2. AN EXISTING ALGORITHM FOR COMPILING REGULAR EXPRESSIONS

In this section we present one of the existing algorithms for the compilation of regular expressions into deterministic finite automa. The method is based on a classical algorithm described by McNaugton and Yamada [MY60]; the presentation is based on algorithm 3.5 in [ASU86], but has been slightly adapted in anticipation of the lazy and incremental case which are discussed later on.

2.1. Preliminaries

First, we introduce the notions of regular expression and labelled regular expression.

Definition 1.
Regular expressions over a finite alphabet Σ and the sets of strings over Σ denoted by them are defined as follows:
(1) The empty string ε is a regular expression denoting the set $\{\varepsilon\}$.
(2) $a \in \Sigma$ is a regular expression denoting the set $\{a\}$.

(3) If r and s are regular expressions denoting the sets R and S respectively, then
(a) $(r)(s)$ is a regular expression denoting RS,
(b) $(r)|(s)$ is a regular expression denoting $R \cup S$,
(c) $(r)^*$ is a regular expression denoting R^*,
(d) (r) is a regular expression denoting R.

RS, $R \cup S$, and R^* are operations for constructing sets by means of, respectively, concatenation of pairs of elements in R and S, union of R and S, and repeated concatenation of elements in R. We will adopt the convention that parentheses may be omitted under the assumption that the operators in regular expressions are left associative and that * has the highest priority, concatenation has the second highest priority and | has the lowest priority.

Definition 2.
A *labelled regular expression* is a regular expression in which a unique natural number p is associated with each occurence of a symbol $a \in \Sigma$ in e. We say that a occurs at position p and that the symbol at position p is a, notation: a_p. Also define $symbol(p)=a$ for each a_p.

In labelled regular expressions, the occurrences of symbols of the alphabet are explicitly labelled. This allows the definition of functions on labelled regular expressions which describe properties of the strings recognized by them. First, three auxiliary functions are introduced. The predicate *nullable* determines whether a regular expression can recognize the empty string. The function *firstpos* maps a labelled regular expression to the set of positions that can match the first symbol of an input string. The function *lastpos* maps a labelled regular expression to the set of positions than can match the last symbol of an input string.

Definition 3.
The auxiliary functions *nullable*, *firstpos* and *lastpos* on a labelled regular expression e are defined as follows:

(1) if $e = \varepsilon$: $nullable(\varepsilon) = \text{true}$
$firstpos(\varepsilon) = \varnothing$
$lastpos(\varepsilon) = \varnothing$

(2) if $e=a_p$, $a \in \Sigma$, p a position:
$nullable(a_p) = \text{false}$
$firstpos(a_p) = \{p\}$
$lastpos(a_p) = \{p\}$

(3) (a) if $e=rs$ is a labelled regular expression, r and s are labelled regular expressions as well and:
$nullable(rs) = nullable(r) \wedge nullable(s)$
$firstpos(rs) =$ **if** $nullable(r)$ **then** $firstpos(r) \cup firstpos(s)$ **else** $firstpos(r)$
$lastpos(rs) =$ **if** $nullable(s)$ **then** $lastpos(r) \cup lastpos(s)$ **else** $lastpos(s)$

(b) if $e=r|s$ is a labelled regular expression, r and s are labelled regular expressions as well and:
$nullable(r|s) = nullable(r) \vee nullable(s)$
$firstpos(r|s) = firstpos(r) \cup firstpos(s)$
$lastpos(r|s) = lastpos(r) \cup lastpos(s)$

(c) if $e=r^*$ is a labelled regular expression, r is a labelled regular expressions as well and:
$nullable(r^*) = \text{true}$
$firstpos(r^*) = firstpos(r)$
$lastpos(r^*) = lastpos(r)$.

Now we can define the function *followpos* which maps a position in a labelled, regular expression e to the set of positions that can follow it, i.e., if p is a position with $symbol(p) = a$ and p matches the symbol a in some legal input string $...ab...$, then b will be matched by some position in $followpos(p, e)$.

In the sequel, we will adopt the convention that a unique symbol $\$ \in \Sigma$ is used to terminate both regular expressions and input strings. A *terminated, labelled,* regular expression e over an alphabet Σ, has the form $e'\$$, where e' is a labelled regular expression over $\Sigma \setminus \{\$\}$.

Definition 4.
The function *followpos* on positions in a labelled regular expression e is defined as follows. First, introduce the sets *cat* and *star*:

(a) $cat(p, e) = \{ rs \mid p \in lastpos(r) \wedge rs$ a subexpression of $e\}$
(b) $star(p, e) = \{ r^* \mid p \in lastpos(r) \wedge r^*$ a subexpression of $e\}$.

Followpos can then be defined by:

$$followpos(p, e) = \bigcup_{rs \in cat(p, e)} firstpos(s) \cup \bigcup_{r^* \in star(p, e)} firstpos(r).$$

In the sequel, we will use the abbreviation $followpos(p, E) = \bigcup_{e \in E} followpos(p, e)$, where E is a set of labelled regular expressions.

Definition 5.
An *accepting sequence of positions* for a labelled regular expression e is a sequence of positions $p_1,...,p_n$ such that $p_1 \in firstpos(e)$, $p_n \in lastpos(e)$, and $p_{i+1} \in followpos(p_i, e)$, $i=1,...,n-1$.

Theorem 1.
For all strings $s \in \Sigma^*$ and for all terminated, labelled, regular expresions e over Σ the following holds: $s=a_1...a_n$ with $a_n = \$$ belongs to the set of strings denoted by e if and only if there exists an accepting sequence of positions $p_1,...,p_n$ for e such that $a_i = symbol(p_i)$, $i=1,...,n$.

Proof
By induction on the structure of the labelled regular expression e (see [YM60], Theorem 3.1).

2.2. *CONSTRUCT:* an algorithm for the construction of a DFA

Using the notions introduced in the previous section we now formulate an algorithm for the construction of a deterministic finite automaton for a given set of regular expressions. The basic idea is to construct a deterministic automaton in which each state corresponds to a *set* of positions in the set of regular expressions. In this way, each state may represent *several* ways of recognizing an input string.The initial state of the automaton consists of the first positions of all the regular expressions. Transitions from the start state, as well as from any other state, are computed as follows: consider for each symbol a in the alphabet (or the end marker) the positions that can be reached when recognizing a in the input; the set of positions that can be reached in this way form the (perhaps already existing) state to which a transition should be made from the original state on input a. This process is repeated until all transitions for all states have been computed. The set of positions that corresponds to a state thus characterizes the progress of all possible accepting sequences for input strings with a common head.

The standard formulation of the following DFA construction algorithm takes *one* labelled regular expression as input and constructs the corresponding automaton. Here, *CONSTRUCT* takes a *set* of labelled regular expressions as input. This is motivated by the desire to define a version of *CONSTRUCT* later on, which allows adding or deleting regular expressions from a given set of regular expressions and constructs a new DFA by updating the DFA constructed for the orginal set. However, in a set of labelled regular expressions equal position numbers may have been assigned

to symbols in different labelled regular expressions in the set. We assume in the sequel, that such conflicting position numbers are avoided by a suitable renumbering.

In principle, the powerset of all positions in the set of regular expressions should be considered during the construction of a DFA. The following algorithms only consider the sets of positions that are really used during this construction. These sets are collected in the set *States*. To determine efficiently whether there are still states for which the transitions have to be computed, a notion of *marking* will be used. When a state S is added to *States*, it is unmarked and *marked*(S) = **false** holds. A state $S \in$ *States* can be marked by the operation *mark*(S), after which *marked*(S) = **true** holds.

The DFA that is being constructed is represented by an initial state *start* $\in$ *States* and a transition function *Trans* : *States* $\times \Sigma \rightarrow$ *States*.

Algorithm *CONSTRUCT*
Construction of a DFA that accepts the language described by a set of regular expressions.
Input. A set E of terminated, labelled, regular expressions over alphabet Σ.
Output. A DFA A described by *States* (set of states), *start* (start state) and *Trans* (transition function).
Method.

$A.start := \bigcup_{e \in E} firstpos(e)$

$A.States := \{A.start\}$
$A.Trans := \emptyset$
while $\exists\, S \in A.States$ [$\neg$ *marked*(S)]
do *mark*(S)
  $A := EXPAND(E, A, S)$
od
return(A)

where *EXPAND* is defined by the following algorithm:

Algorithm *EXPAND*
Expansion of a DFA state.
Input. A set of terminated, labelled, regular expressions E, a corresponding PDFA A, and a state S.
Output. The original PDFA expanded with all states to which S has transitions, and a definition of these transitions.
Method.

for $\forall\, a \in \Sigma \backslash \{\$\}$
do $U := \bigcup_{\{p \in S \mid symbol(p) = a\}} followpos(p, E)$

  if $U \neq \emptyset \wedge U \notin A.States$ **then** $A.States := A.States \cup U$ **fi**
  $A.Trans(S,a) := U$
od
return(A)

From this definition it follows that a state can never correspond to an empty set of positions. For convenience, we will assume in the sequel that all automata contain an *error state* with the following properties:
1. The error state corresponds to the empty set of positions.
2. The error state is not an accepting state.
3. The transition function is augmented as follows:
   (a) for each state, transitions to the error state are added for all characters in Σ for which that state

has no legal transition.
(b) for all characters in Σ, the transition function contains a transition from the error state to itself.

These additions to the generated automata are implicit and will not be shown in the diagrams.

2.3. *SCAN:* the scanning algorithm associated with *CONSTRUCT*

Application of *CONSTRUCT* for a given set of regular expressions results in a DFA described by a start state and a transition function. The simulation of this DFA for a given input string is straightforward.

Algorithm *SCAN*
Simulate a given DFA on a given input string.
Input. A DFA A and an input sentence $s = a_1 \ldots a_n$, with a_n=\$.
Output. **true** or **false**.
Method.

$S := A.start$
$i := 1$
while $a_i \neq \$$ **do** $S := A.Trans(S, a_i)$; $i := i + 1$ **od**
return $FINAL(S)$

where *FINAL* determines whether a state can accept an input string. Due to our conventions concerning error states, this algorithm also works correctly on erroneous input strings.

FINAL is defined by the following algorithm:

Algorithm *FINAL*
Determine whether a given state is an accepting state.
Input. A state S.
Output. **true** or **false**
Method.

return $\exists\, p \in S \;\; [symbol(p) = \$]$

2.4. An example

Consider the following set E of terminated, labelled, regular expressions:

$$E = \{\ a_0(b_1 \mid c_2)^*\$_3,\ b_4{}^*d_5\$_6,\ a_7b_8c_9\$_{10},\ b_{11}c_{12}d_{13}\$_{14}\ \}$$

or, written in the form of ordinary regular expressions:

$$E = \{\ a(b \mid c)^*,\ b^*d,\ abc,\ bcd\ \}$$

The DFA resulting from application of *CONSTRUCT*(E) is shown in figure 1. For each state in the figure, its set of labelled symbols (instead of positions only) is given.

Note that some of the states contain more than one \$-position; this corresponds to an ambiguity in the automaton. The string `abc`, for instance, is recognized by both $a_0(b_1 \mid c_2)^*\$_3$ and $a_7b_8c_9\$_{10}$. We will return to this phenomenon in section 5.

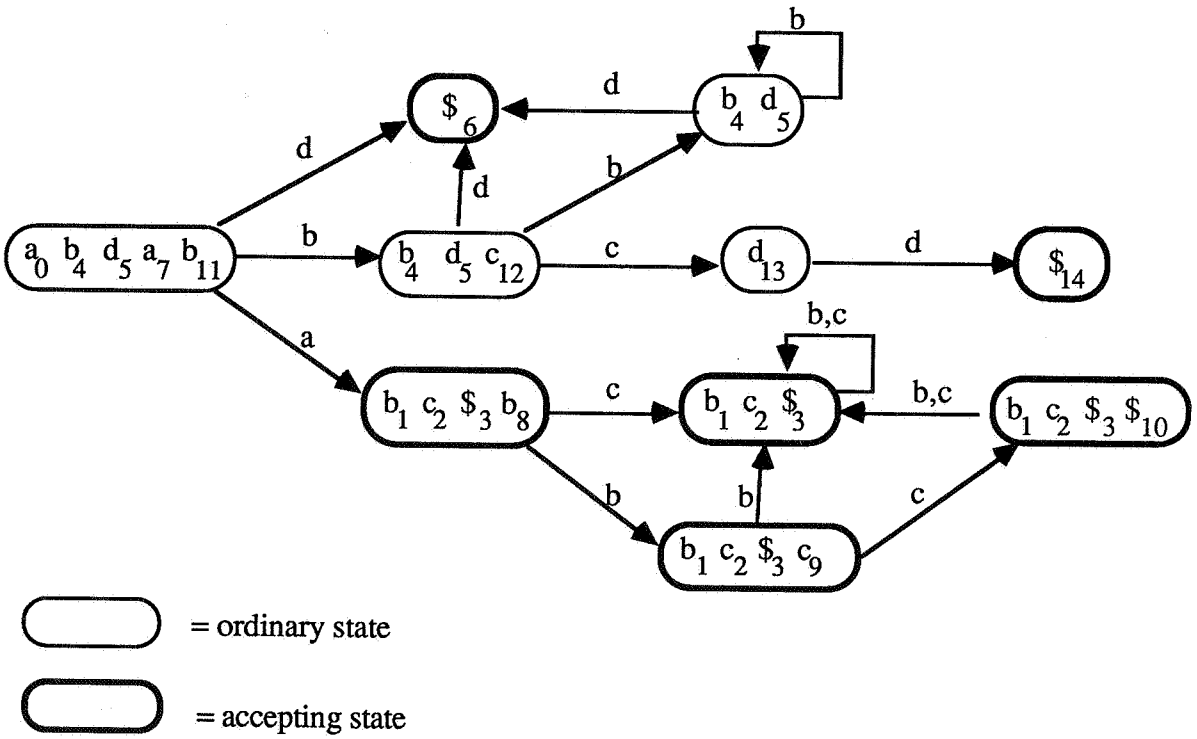

Figure 1. Example of a DFA produced by *CONSTRUCT*

2.5. Correctness and complexity of *CONSTRUCT* and *SCAN*

Theorem 2.
Given a set of terminated, labelled, regular expressions E, and the corresponding DFA $A=CONSTRUCT(E)$, then for all $s\in\Sigma^*$ the following holds:

$$s \text{ is denoted by some } e\in E \Leftrightarrow SCAN(A,s) = \textbf{true}$$

Proof
Assume $s=a_1...a_n$, with a_n=\$.
($\Rightarrow$) If s is denoted by some $e\in E$, there exists (according to theorem 1 above) an accepting sequence $p_1,...,p_n$ of positions for e, such that $a_i = symbol(p_i)$, i=1,...,n. From the way A is constructed by *EXPAND* it follows that there exist states $S_1,...,S_n$ such that $p_i\in S_i$ and $A.Trans(S_i,a_i) = S_{i+1}$, i=1,...,n–1. This sequence of states will be followed by $SCAN(A,s)$, which will return **true** because $FINAL(S_n)$ holds.

($\Leftarrow$) There exist states $S_1,...,S_n$ such that $S_1=A.start$, $S_{i+1} = A.Trans(S_i, a_i)$ and $FINAL(S_n)$. From the construction in *EXPAND* it follows that there exists a sequence of positions $p_1,...,p_n$ such that $p_i\in S_i$ and $p_{i+1}\in followpos(p_i, E)$. Therefore, $p_1,...,p_n$ is an accepting sequence for some e in E. In addition to this, it follows from the definition of *EXPAND* that $a_i = symbol(p_i)$. Using theorem 1, one may conclude that s is denoted by e.

The above proof is based on the proof of Theorem 3.2 given in [MY60]. See [BS87] for a proof of a similar result which is based on derivatives of regular expressions.

Without proof, we state the complexity of the above algorithms. Let the number of positions in the

set of regular expressions E be $|E|$, and let the size of an input string s for *SCAN* be $|s|$, then we have the following complexity results:

CONSTRUCT: time: $O(2^{|E|})$, space: $O(2^{|E|})$.
SCAN: time: $O(|s|)$.

The following well-known example (see [A80] or [ASU86]) exhibits the worst case behavior. Consider the regular expression consisting of `a*b` followed by $m-1$ times `(a|b)`. This expression denotes strings in which the mth symbol from the right is a `b`. Unfortunately, the smallest DFA that recognizes this expression has 2^m states.

3. LAZY CONSTRUCTION OF A DFA

3.1. *L-CONSTRUCT:* lazy construction of a DFA

The first step in the direction of an incremental lexical scanner generator is made by observing that *CONSTRUCT*, as given above, constructs *all* states of the DFA, while not all of them are always needed for the matching of an input string. This can be prevented if we shift the responsibility for expanding the DFA from *CONSTRUCT* to *SCAN*, i.e., initially we only construct the start state and when later on during scanning a state is used which has not yet been expanded, it is expanded by need.

Algorithm *L-CONSTRUCT*
Construction of the initial part of the DFA that accepts the language described by a set of regular expressions.
Input. A set E of terminated, labelled, regular expressions over alphabet Σ.
Output. A PDFA in which only the start state has been expanded.
Method.

$A.start := \bigcup_{e \in E} firstpos(e)$

$A.States := \{ A.start \}$
$A.Trans := \emptyset$
return(*EXPAND*(E, A, $A.start$))

Strictly speaking, the above expansion step of the start state is unnecessary, since it could also be carried out by *L-SCAN* (see below). The reason for including it here is that this gives the possibility of optimizing the representation of the transition function for the start state (as is done in the actual implementation of the algorithm).

3.2. *L-SCAN:* the scanning algorithm associated with *L-CONSTRUCT*

L-SCAN is the scanning algorithm associated with *L-CONSTRUCT*. It is similar to *SCAN*, but performs expansions of needed, unmarked, states.

Algorithm *L-SCAN*
Simulate a given PDFA on a given input string, incrementally expanding the PDFA when necessary.
Input. A set E of terminated, labelled, regular expressions, a corresponding PDFA A, and an input sentence $s = a_1 ... a_n$, with a_n=\$.
Output. **true** or **false** (indicating acceptance or rejection of the input string) and a possibly extended version of A.

Method.
S:= A.*start*
i := 1
while $a_i \neq \$$
do **if** $\neg$ *marked*(S) **then** *mark*(S); A := *EXPAND*(E, A, S) **fi**
S:= A.*Trans*(S, a_i)
i := i + 1
od
return (*FINAL*(S), A)

Note that at most n applications of *EXPAND* are necessary when a string of length n is scanned (see section 3.4).

3.3. Example

Consider the same set E of terminated, labelled, regular expressions as in section 2.4:

$$E = \{\ a_0(b_1 \mid c_2)^*\$_3,\ b_4{}^*d_5\$_6,\ a_7b_8c_9\$_{10},\ b_{11}c_{12}d_{13}\$_{14}\ \}$$

The PDFA resulting from application of *L-CONSTRUCT*(E) is shown in figure 2a. Unexpanded states are shaded. Next, several extensions of this PDFA are shown, in figures 2b and 2c, resulting from reading the input strings `bcd` and `bbd`. Compare these partial automata with the complete automaton in figure 1.

3.4. Correctness and complexity of *L-CONSTRUCT* and *L-SCAN*

Theorem 3
Given a set E of regular expressions, and a sequence $\{s_i\}_{i\geq 1}$ of strings over Σ.
Let
A_0 = *L-CONSTRUCT*(E)
(u_i, A_i) = *L-SCAN*(E, A_{i-1}, s_i) ($i\geq 1$)
A = *CONSTRUCT*(E),
then
(*a*) $A_0 \subseteq ... \subseteq A_{i-1} \subseteq A_i \subseteq ... \subseteq A$,
(*b*) s_i is denoted by some $e \in E \Leftrightarrow u_i$ = **true**.

Proof
(*a*) First observe that A_i.*start* = A.*start* ($i\geq 1$) and that *L-SCAN* only adds new states but never removes them. From the construction in, on the one hand *CONSTRUCT* and on the other hand *L-CONSTRUCT*/ *L-SCAN*, it is clear that both methods will ultimately construct the same automata: as long as A_i.*States* contains unmarked states, new states may be added to it by *L-SCAN* while scanning S_{i+1}, but all these states already occur in A.*States*.
(*b*) Similar to the proof of theorem 2. In the ($\Rightarrow$) part, it may now not be assumed that a sequence of states S_1,...,S_n already exists, but it can be shown that this same sequence is constructed during the execution of *L-SCAN*. The ($\Leftarrow$) part of the proof is identical.

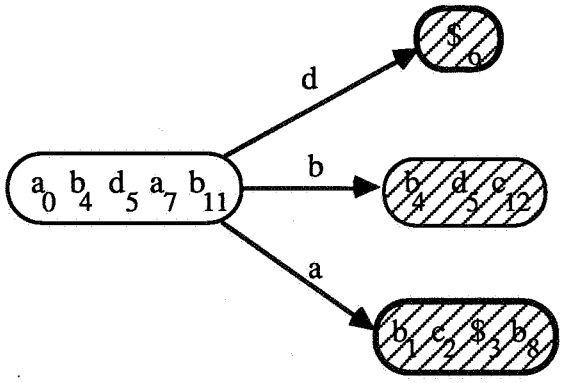

Figure 2a. PDFA produced by *L-CONSTRUCT* (no input read yet).

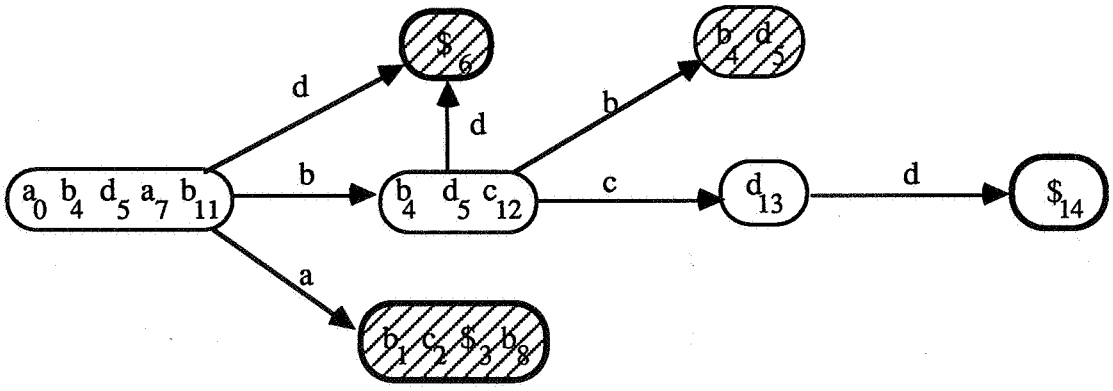

Figure 2b. PDFA after reading the input `bcd`.

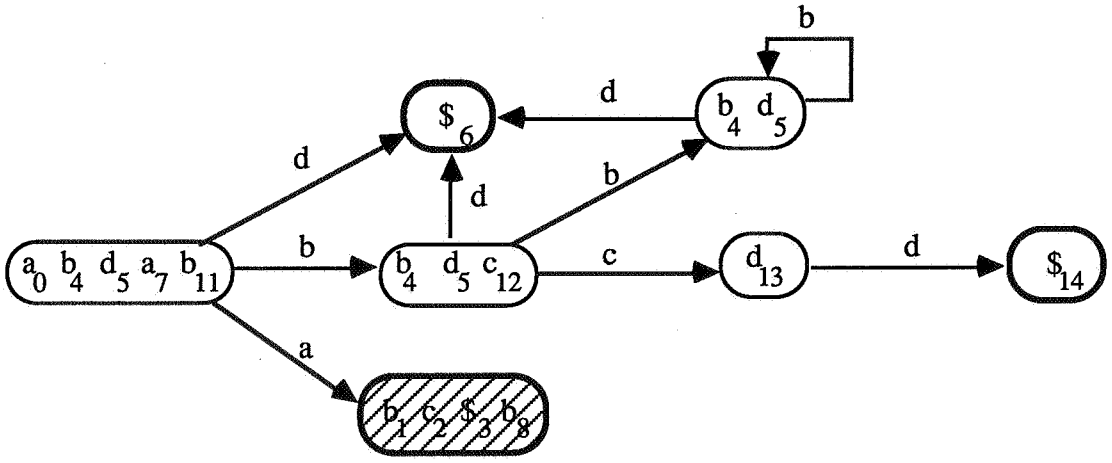

Figure 2c. PDFA after reading the inputs `bcd` and `bbd`.

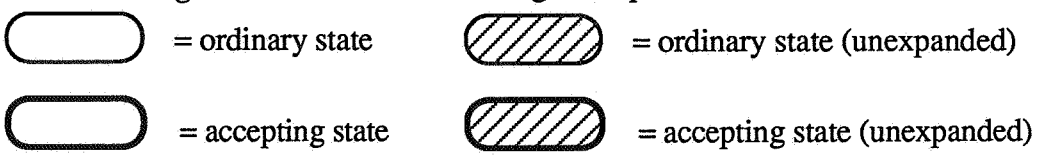

Figure 2. Stages in the lazy construction of a DFA.

The complexity of the above algorithms is comparable to, but more difficult to express than that of *CONSTRUCT* and *SCAN*. Obviously, *L-CONSTRUCT* is now a simple operation of time O(|*E*|) since most of the work has been shifted to *L-SCAN*. It is also clear that *L-SCAN* is still of time O(|*s*|) for input string *s*, when all states of the automaton have been expanded, but before that moment each application of *L-SCAN* has to be charged with some time for the construction of parts of the automaton. The amount of construction time, however, depends on the particular input sentences presented to *L-SCAN*. In the worst case, the total amount of construction time and the total space requirements over a sequence of applications of *L-SCAN* are both of the order $O(2^{|E|})$, just as before.

It is interesting to compare the behaviour of *L-CONSTRUCT* and *L-SCAN* with the results described in section 2.4. for the regular expression consisting of `a*b` followed by $m-1$ times `(a|b)`. In the lazy case, when parsing a given input string, only $2m$ states will be constructed from the 2^m states of the fully constructed automaton.

4. INCREMENTAL MODIFICATION OF A PDFA

4.1. *MODIFY:* incremental modification of a PDFA

The next algorithm in the series addresses the problem of modifying a PDFA after a modification has been made to the original set of regular expressions for which the PDFA was generated by *L-CONSTRUCT/L-SCAN*. The overall strategy will be to reuse the old PDFA as much as possible. This is achieved in the following, simple, way. First, a new start state is computed for the modified set of regular expressions. Next, a garbage collection is performed on the old PDFA using the new start state, i.e., all states in the old PDFA which cannot be reached from the new start state are removed. The new start state together with the cleaned-up old PDFA constitute the new PDFA.

Algorithm *MODIFY*
Incremental modification of a PDFA.
Input. A set of terminated, labelled, regular expressions *Eold*, a terminated, labelled, regular expression e_0, an edit operation *Operation* with values *add* or *delete*, and a PDFA *Aold*.
Output. The modified set of regular expressions $Enew = E \cup \{e_0\}$ (if *Operation* = *add*) or $Enew = E \setminus \{e_0\}$ (if *Operation* = *delete*), and the corresponding PDFA *Anew*.
Method.

if *Operation* = *add* **then** $Enew := Eold \cup \{e_0\}$ **else** $Enew := E\,old \setminus \{e_0\}$ **fi**

$start' := \bigcup_{e \in Enew} firstpos(e)$

Atmp.start := *start'*
Atmp.States := *Aold.States* ∪ { *start'* }
Atmp.Trans := *Aold.Trans*
(create intermediate PDFA *Atmp*)

Atmp := *EXPAND*(*Enew*, *Atmp*, *start'*)
(expand new start state)

Anew.start := *start'*
Anew.States := *REACHABLE*(*start'*, *Atmp*)
Anew.Trans := $Atmp.Trans\,|_{Anew.States}$
return(*Enew*, *Anew*)
(create resulting PDFA *Anew*)
(restrict to reachable states)

The above formulation of *MODIFY*, is greedy, i.e., the modification is applied to the automaton at the moment that the modification is defined, as opposed to postponing this work until the automaton is used. We have actually implemented *MODIFY* in a lazy fashion: in this way, a series

of modifications, as occurs, for instance, when a new grammar is defined by adding regular expressions to an initially empty set, can be combined and much superfluous processing can be eliminated.

Reachable is defined by the following algorithm:

Algorithm *REACHABLE*
Determine the reachable states of a PDFA.
Input. A start state *start*, and a PDFA *A*, with *start*$\in$*A.States*.
Output. A set $R \subseteq A.States$, such that all states in *R* can be reached from *start* using the transition function *A.Trans*.
Method.

Define the derivation relation $\Rightarrow$ between two states *S* and *S'* for a given transition function *A.Trans* as follows: $S \Rightarrow S'$ if and only if $\exists a \in \Sigma$ such that $A.Trans(S, a) = S'$.

return $\{ S \in A.States \mid start \Rightarrow^* S \}$

After a modification of a PDFA by *MODIFY*, *L-SCAN* can be used for scanning input strings and for lazy expansion of the PDFA in the modified context.

4.2. An example

Consider adding the regular expression $d_{15}e_{16}\$_{17}$ to the set *E* given in sections 2.4 and 3.3. Assume that the PDFA has been expanded as shown in figure 2c. The result of applying *MODIFY* is shown in figure 3. Note that most parts of the old automaton can be reused.

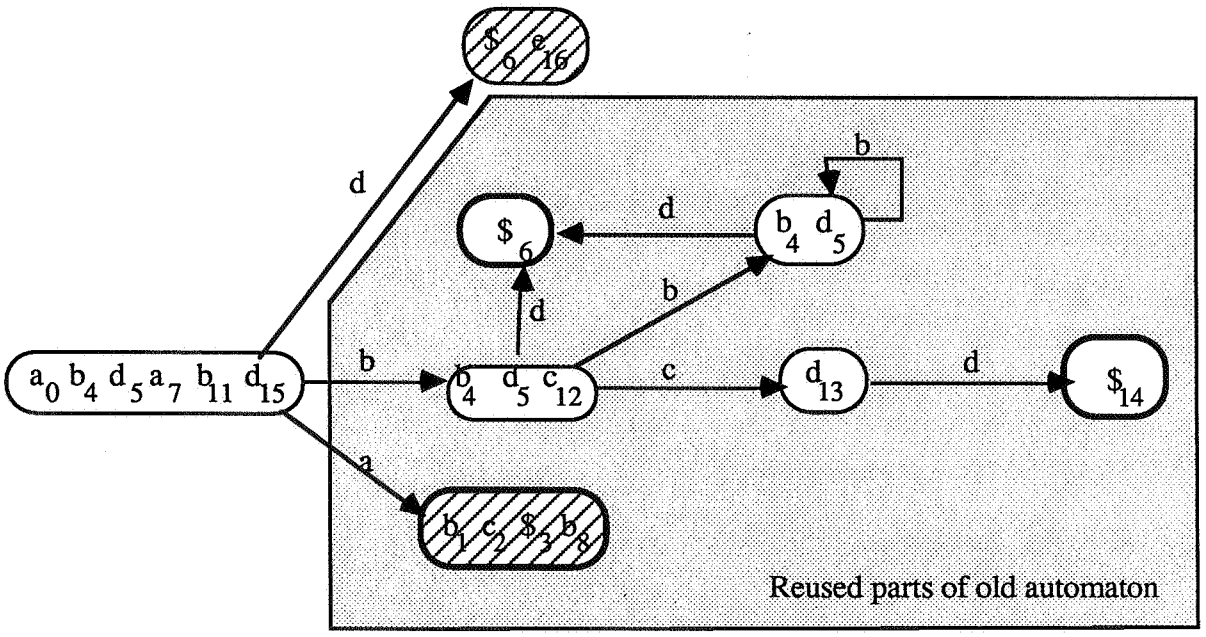

Figure 3. New states constructed for the expression $d_{15}e_{16}\$_{17}$ by *MODIFY*.

4.3. Correctness and complexity of *MODIFY*

The correctness of *MODIFY* can be verified by comparing *L-CONSTRUCT* and *MODIFY*. We will not give a detailed proof here. Note that theorem 3 also holds for automata constructed by means of *MODIFY*, since the garbage collection of states (as achieved by *REACHABLE*) guarantees the ordering $A_0 \subseteq ... \subseteq A_{i-1} \subseteq A_i \subseteq ... \subseteq A$ of successive approximations to A.

Let |*Enew*| be the number of positions in the new set of regular expressions, and let |*Aold.States*| be the number of states of the old automaton. The time needed for *MODIFY* is then of the order O(|*Enew*|+ |*Aold.States*|). The time needed for modifying a PDFA is thus directly related to its level of expansion.

Unfortunately, *MODIFY* does not optimally reuse states of the old automaton in the new one. This is due to the "eagerness" of *REACHABLE* to remove unreachable states. Some of the removed states could have been reused, but only became reachable after one or more expansions of the new automaton. Consider, for instance, the case that we have a fully expanded automaton for $a_1b_2\$_3$ and that the expression $a_4c_5\$_6$ is added. In the new automaton, the state corresponding to $\{\$_3\}$ is not reachable from the new start state and will be removed. Later on, however, it may occur that this same state is created once again. This situation is sketched in figure 4.

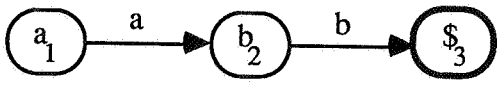

Figure 4a. Complete automaton for $a_1b_2\$_3$.

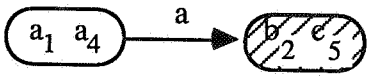

Figure 4b. New automaton after addition of $a_4c_5\$_6$.

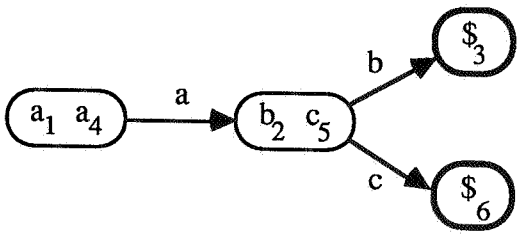

Figure 4c. Completely expanded new automaton.

5. INCREMENTAL MODIFICATION USING THE SCANNER

5.1. *S-MODIFY:* Incremental modification of a PDFA using the scanner

The last algorithm to be presented is an improved version of *MODIFY*. It is based on the observation that frequently occurring sets of regular expressions (such as, for instance, the lexical syntax of a programming language) are *ambiguous*. Such ambiguities occur when an expression defining a general case (e.g., identifiers) and an expression describing a special case (e.g., the keyword `begin`) match the same string. The following algorithm exploits this phenomenon by trying to match *literal* regular expressions (i.e. regular expression not containing | or *) by means of the old automaton, before new states are added for matching the literal. If the literal is already recognized by the old automaton, no new states have to be added. We only present an algorithm for

accepting or rejecting strings and not for determining which regular expressions were responsible for the recognition. The latter requires adding additional information to the generated automaton. Note that this technique may give improvement in the classical (non-incremental) case as well.

Algorithm *S-MODIFY*
Incremental modification of a PDFA, scan literal strings using old PDFA before modification.
Input. A set of terminated, labelled, regular expressions *Eold*, a terminated, labelled, regular expression e_0, an edit operation *Operation* with values *add* or *delete*, and a PDFA *Aold*.
Output. The modified set of regular expressions $Enew = E \cup \{e_0\}$ (if *Operation* = *add*) or $Enew = E \setminus \{e_0\}$ (if *Operation* = *delete*), and the corresponding PDFA *Anew*.
Method.

if *Operation* = *add* **then** $Enew := E\,new \cup \{e_0\}$ **else** $Enew := E \setminus \{e_0\}$ **fi**

(create an intermediate PDFA *Atmp*)
$Atmp.start := \bigcup_{e \in Enew \setminus LITERALS(Enew)} firstpos(e)$
$Atmp.States := Aold.States$
$Atmp.Trans := Aold.Trans$

(collect all literal expressions in *Enew* which are matched by *Atmp*)
$MATCHED := \emptyset$
for $\forall\ e \in LITERALS(Enew)$
do $(b, Atmp') := L\text{-}SCAN(Enew \setminus LITERALS(Enew), Atmp, e)$
$Atmp := Atmp'$
if b = **true then** $MATCHED := MATCHED \cup \{e\}$ **fi**
od

(create start state for all but the already matching expressions in *Enew*)
$start := \bigcup_{e \in Enew \setminus MATCHED} firstpos(e)$

$Atmp := EXPAND(Enew \setminus MATCHED, Atmp, start)$

(construct the resulting PDFA *Anew*)
$Anew.start := Atmp.start$
$Anew.States := REACHABLE(start, Atmp)$
$Anew.Trans := Atmp.Trans\,|_{Anew.States}$
return (*Enew*, *Anew*)

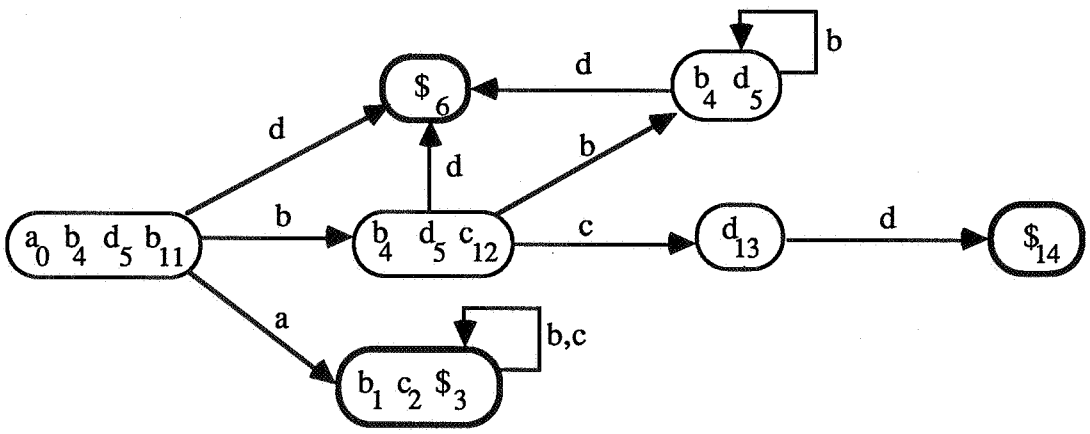

Figure 5. DFA resulting from *S-MODIFY*.

5.2. Example

Continuing our running example, the fully expanded automaton resulting from *S-MODIFY* is shown in figure 5. Note that the number of states was 10 in figure 1 and that the new automaton has only 7. The explanation is that the literal expression `abc` is subsumed by `a(b|c)*`. No additional states are needed for recognizing it.

5.3. Correctness and complexity of *S-MODIFY*

The correctness of *S-MODIFY* can be verified by comparing the PDFAs built by *MODIFY* and *S-MODIFY*. There are three possibilities for rejecting/accepting an input string: (1) the input string is not accepted; (2) it is accepted by a literal regular expression only; (3) it is accepted by both a literal and a non-literal regular expression. In cases (1) and (2) both automata behave in an identical manner. In case (3), the automata behave differently. However, the application of *L-SCAN* in *S-MODIFY* ensures that both automata recognize the same strings.

In addition to *MODIFY*, *S-MODIFY* has to perform a number of scan operations which is linearly related to the number of positions in the new set of regular expressions *Enew*. The time complexity of *S-MODIFY* thus remains O(|*Enew*| + |*Aold.States*|). Note that the *L-SCAN* operations performed may lead to expansion of the old automaton and that, in the worst case that no match occurs using the old automaton, these new states may become unreachable when we construct the new automaton.

6. SOME EXTENSIONS

The algorithms presented in the previous sections can be extended to more general forms of regular expressions. Simple extensions include operators such as + (one or more repetitions) and ? (an optional construct) which can be translated into the already available operators using identities such as e+ ≡ e e* and e? ≡ e | ε. In this section we consider two more involved extensions: *named regular expressions* and *character classes*.

6.1. Named regular expressions

Named regular expressions extend regular expressions with the facility to give a name to a regular expression and to use that name in another expression. The intention is that the use of the name in the second expression is equivalent to a textual substitution of the first regular expression in the second one. The additional complexity introduced by named regular expressions is that, after a modification of some named regular expression, all expressions using it should also be considered to be modified.

6.2. Character classes

Character classes are sets of characters that may match at a certain position in the regular expression. The usual ways of denoting character classes are:
(1) by explicit enumeration of the characters in the class: $[a_1 a_2 ... a_n]$;
(2) by indicating the range of characters in the class: $[a_f - a_l]$, where a_f is the first character in the range and a_l is the last one. Note that this notation assumes an ordering on the alphabet Σ.
Character classes do not add descriptive power to regular expressions, they are just a very convenient abbreviation for a sequence of alternative operators, i.e.

$$[a_1 a_2 ... a_n] \equiv a_1 \mid a_2 \mid ... \mid a_n .$$

It turns out that an implementation of character classes based on their expansion into a list of alternatives is very inefficient. The reason for this is that, in principle, the same set of follow positions will be constructed for each character in the class. Obviously, it is more efficient to construct the set of follow positions once for the whole class. However, one has to be careful when character classes overlap with each other or with single characters in the regular expression. Consider, for instance, the following set of labelled regular expressions:

$\{ [a\text{-}z]_1, [e\text{-}l]_2, [x\text{-}z]_3, i_4, p_5 \}$.

When representing character classes by means of alternatives, 39 positions are occupied and 26 sets of follow positions have to be constructed (assuming that the alphabet only consists of letters). If a character class is allowed to occupy only one position, only 5 positions are needed and a case analysis shows that only 5 different combinations of these positions may occur during the expansion step of the start state for the above set of expressions (see figure 6). Clearly, the best strategy is to construct these combinations only once.

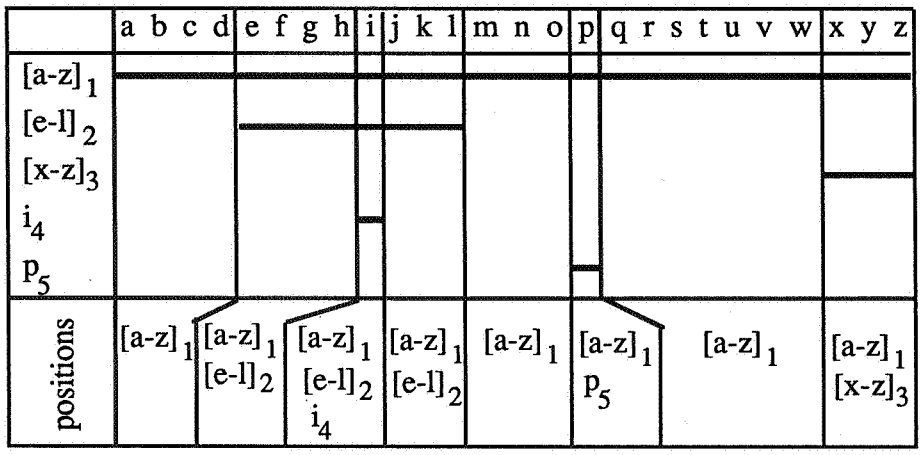

Figure 6. Case analysis of sets of positions in example

We now discuss a variant of *EXPAND* which achieves this. First, some additional notions for manipulating character classes in regular expressions have to be introduced :
(1) Assume that the alphabet is linearly ordered, i.e. $\Sigma=\{a_1,...,a_n\}$.
(2) The predicates *is-char*(p) and *is-charclass*(p) are true if a character or a character class occurs at position p, respectively.
(3) The function *symbols*(p) is defined as follows: if position p corresponds to a single character then the singleton {*symbol*(p)} is returned, otherwise position p corresponds to a character class and the set of characters in that class is returned.
(4) The function *diff*(p, i) returns the index j of the first element of the alphabet for which $j > i$ and $a_j \in$ *symbols*(p) differs from $a_i \in$ *symbols*(p), i.e., the smallest element of the alphabet greater than a_i that has a different member relationship to *symbols*(p) than a_i has. By means of the function *diff* all membership tests for characters between a_i and a_j can be eliminated.

In *EXPAND*, as given in section 2.2, a set of follow positions is constructed for each character in the alphabet. Such a set of follow positions represents the positions that can be reached from positions in the original set S on input of each particular character. The new version of *EXPAND*, postpones the construction of sets of follow positions until the complete alphabet has been processed. This is achieved by introducing a mapping *CCpositions* from subsets of S to sets of ordered pairs of indices in the alphabet. These pairs describe the ranges of characters in the alphabet

for which a transition from the original state S to the state corresponding to the follow positions of this subset of S should be constructed. The positions in each subset correspond to character classes only.

The algorithm constructs, for each character a in the alphabet, a subset P of S wich contains positions matching a. Next, two cases are distinguished:
(1) Some position in P corresponds to a single character: this makes the subset P unique and the set of follow positions can be constructed immediately.
(2) All positions in P correspond to character classes: the subset P may not be unique and may also occur for other characters in the alphabet. The existence of P is recorded in *CCpositions* and the construction of the set of follow positions of P is postponed until the whole alphabet has been treated.

Algorithm *CC-EXPAND*
Expansion of a DFA state in the presence of character classes.
Input. A state S, a PDFA A, and a set of regular expressions E.
Output. The original PDFA expanded with all states to which S has transitions and a definition of these transitions.
Method.

```
CCpositions := ∅
i := 1                                   (index in the alphabet)
while i ≤ n
do P := { p∈S | a_i ∈ symbols(p) }
   if P = ∅ then
      i := i +1
   elseif ∃p ∈ P [ is-char(p) ] then
      (P is unique, it contains at least one character position)
      U := ∪_{p∈P} followpos(p, E)

      if U ∉ A.States then A.States := A.States ∪ U fi
      A.Trans(S,a_i) := U
      i := i +1
   else
      (P may not be unique, it contains character classes only)
      nxt := min( { diff(p, i) | p ∈ P } )
      CCpositions(P) := CCpositions(P) ∪ (i, nxt −1)
      i := nxt
   fi
od
for ∀P ∈ CCpositions
do U := ∪_{p∈P} followpos(p, E)

   if U ∉ A.States then A.States := A.States ∪ U fi
   for ∀(l,h) ∈ CCpositions(P)
   do
      for k := l,...,h do A.Trans(S,a_k) := U od
   od
od
return (A)
```

It can be shown that the automata constructed using *EXPAND* and *CC-EXPAND* recognize the same language. Assume that A and A' are the automata constructed by means of respectively *CONSTRUCT/EXPAND* and *CONSTRUCT/CC-EXPAND* for the same set of regular

expressions. Then there exists a homorphism h: $A \rightarrow A'$ such that
$h(A.States) = A'.States$
$h(A.start) = A'.start$
$h(A.Trans(S,a)) = A'.Trans(h(S),a)$
S is an accepting state of $A \Leftrightarrow h(S)$ is an accepting state of A'.

7. IMPLEMENTATION

7.1. Overview of the implementation

The fully lazy/incremental scanner generator ISG is based on the algorithms given in the prevous sections. It consists of 1200 lines of LeLisp[CH86] code and supports the incremental generation of lexical scanners defined by means of named regular expressions using the following primitives and constructors:

`a`	`a` is some character from the ASCII alphabet.
`.`	an arbitrary character.
`\a`	escape for meta characters, such as + and *.
`[`$a_1 \ldots a_n$`]`	character class: each a_i is either a ASCII character, or has the form a_1`-`a_h which indicates the range of characters from a_1 to a_h.
`[~`$a_1 \ldots a_n$`]`	complemented character class: same as character classes, but now all characters or character ranges *not* in the class are given.
$e_1 e_2$	concatenation
e_1`|`e_2	alternation
`e*`	zero or more repetitions
`e+`	one or more repetitions
`e?`	optional expressions
`(e)`	parentheses for grouping
`{def}`	use a previously defined regular expression with name `def`.

The scanners generated by ISG allow repeated reading of lexical tokens from an input string until the end of the string is reached. They return the recognized part of the input string together with all its possible interpretations. The generated scanner attempts to read the longest possible token from the input, but if the longest possible match is still ambiguous, a list with all possible interpretations is returned. This list is ordered according to increasing generality of the matching regular expressions, i.e., a match of a literal rule describing a keyword preceedes a match by the more general rule for identifiers.

7.2. Some remarks on implementation techniques

It turns out that the overall efficiency of the implementation is determined by the efficiency of *set operations*, the *expand function* and the *transition functions* of the generated scanners.

The set operations that occur most frequently are set union (combining sets of positions, for instance) and test for membership in sets of sets (for instance, to determine whether a given set of positions already exists). The former operation is efficiently implemented by imposing an implicit ordering on the elements in each set, the latter by representing sets of sets by means of hash tables.

The expand operation is an efficient implementation of *CC-EXPAND* as given in section 6.2. The major optimizations applied are (1) avoiding some of the set constructions occurring in the case that P is not unique (the computation of *nxt* , for instance, is distributed over previous loops); (2)

avoiding repeated assignments to *Trans* for all elements in some range of the alphabet by extending the representation of the *Trans* function with range information (see below).

A state and its transitions are usually represented by means of a vector, containing a state value for each element in the alphabet. Several compactification techniques exist to reduce the memory requirements of this scheme. In our implementation, transitions are represented by lambda expressions. In principle, each lambda expression contains a conditional expression that distinguishes the various ranges of characters for which transitions exist. This is a compact but slow way of representing the transition function. One optimization is to represent the transition function for states with transitions to themselves by means of a while-expression rather than an if-expression.

Finally, literals that are also matched by some non-literal regular expression are implemented by attaching a hash table to all accepting states of the generated automaton. When reaching an accepting state, a table lookup is performed to determine all possible interpretations.

8. MEASUREMENTS

8.1. Method

Experimental performance analysis of computer programs is a delicate job. This is particularly true when measurements are very sensitive to the particular input data being used. Unfortunately, this is precisely the situation that occurs during incremental scanner generation: the sequence of partial scanners generated depends on the particular sequence of inputs presented to the scanner generator. In principle, the following parameters have to be taken into account:

- the scanner construction/modification algorithm (*MODIFY* or *S-MODIFY*);
- the level of compilation of the generated scanners;
- the input data presented to the generated scanners;
- the modification operations on the original lexical syntax.

We will simplify this situation by considering the following, fixed, scenario:

(1) read lexical syntax for the programming language C;
(2) read C program text;
(3) read same text again;
(4) add a new keyword to the lexical syntax;
(5) read same text;
(6) read same text;
(7) modify the definition of identifiers;
(8) read same text;
(9) read same text.

The expectation is that in steps (2), (5) and (8) time will be spent in (re)constructing the automaton while reading input text. In steps (3), (6) and (9), however, no preprocessing is necessary and we may expect the best possible execution time of the generated scanner. Modification (4) is a simple modification of the lexical syntax which has only a very local effect on the generated automaton. Modification (7), however, may have more global effects since the recognition of keywords and the recognition of identifiers interfere.

In order to measure the total generation time needed by the incremental algorithms the above scenario has been applied to two text files:

File1: 8000 characters of arbitrary C text,
File2: 8000 characters of C text which includes all possible forms of lexical tokens.

Clearly, after reading File2, the complete automaton will have been generated.

Finally, we have measured the influence of the level of compilation of the generated scanners:
C.0: no compilation of the generated transition functions;
C.1: only transition functions for states with transitions to themselves are compiled;
C.2: all transition functions are compiled.
One may expect, that increasing the level of compilation will increase the generation time but decrease the execution time of the generated scanners.

8.2. Measurements

All measurements were performed on a VAX 780 running under Unix version BSD 4.3.
As a yardstick we use the performance of LEX [L75]. The time needed for the generation of a scanner for C, the number of states of that scanner and its execution times on File1 and File2 are given in Table II.

The results of applying the scenario sketched in the previous section are given in tables III and IV. We used a version of the scanner generator compiled with the LeLisp compiler Complice. In Table III, no time is given for the modification operations of the grammar (steps (4) and (7)), since all such operations are queued in the implementation and the actual work is postponed until the scanner is used, which happens in steps (5) and (7) respectively. All times given are the result of repeated measurements and the maximal relative error observed was 10%. Care has been taken to minimize the influence of garbage collections on the measurements.

In Table IV, the total number of states for each generated scanner is given. Between parentheses, the number of *new* (i.e. regenerated) states is given. The number of new states is small when the old automaton is reused to a high degree.

scanner generation	77.0	sec. (= 41 (LEX) + 36 (C compilation))
number of states	211	
read File1	1.4	sec.
read File2	1.4	sec.

Table II. Measurements for LEX

	File1 (MOD)			File1 (S-MOD)			File2 (MOD)			File2 (S-MOD)		
	C.0	C.1	C.2	C.0	C.1	C.2	C.0	C.1	C.2	C.0	C.1	C.2
(1) read syntax	2.4	2.4	2.4	2.0	2.0	2.2	2.2	2.2	2.5	2.0	2.1	2.2
(2) read text	7.6	6.4	11.2	5.0	2.8	3.5	10.1	9.8	23.5	5.6	4.3	5.8
(3) read text	5.2	3.5	1.5	4.5	2.0	1.9	5.2	3.6	1.4	4.2	2.0	1.7
(4) ---------- add new literal ----------												
(5) read text	6.4	4.5	2.6	5.3	2.8	2.6	6.2	4.6	2.6	5.2	2.9	2.5
(6) read text	5.6	3.5	1.5	4.7	2.0	1.8	5.2	3.6	1.4	4.3	2.0	1.7
(7) ---------- change identifiers ----------												
(8) read text	8.5	6.4	10.1	5.4	2.9	2.8	13.6	10.8	20.8	5.2	2.9	2.5
(9) read text	5.2	3.5	1.5	4.7	2.0	1.8	6.0	4.0	1.5	4.3	1.9	1.7

Table III. Summary of execution times (in seconds)

	File1 (MOD)	File1 (S-MOD)	File2 (MOD)	File2 (S-MOD)
(1) read syntax	43 (+43)	34 (+34)	43 (+43)	34 (+34)
(2) read text	129 (+86)	61 (+27)	204 (+161)	80 (+46)
(3) read text	129 (+0)	61 (+0)	204 (+0)	80 (+0)
(4)	----------	add new literal	----------	
(5) read text	131 (+2)	61 (+0)	206 (+2)	80 (+0)
(6) read text	131 (+0)	61 (+0)	206 (+0)	80 (+0)
(7)	----------	change identifiers	----------	
(8) read text	131 (+73)	61 (+1)	206 (+129)	80 (+1)
(9) read text	131 (+0)	61 (+1)	206 (+0)	80 (+0)

Table IV. Summary of generated states

The memory organization of C and Lelisp are difficult to compare, therefore we have not attempted to compare the sizes of the scanners produced by ISG and LEX.

8.3. Interpretation of the measurements

From the above measurements several conclusions can be drawn:
(1) the longest generation time measured was about 23.5 seconds (File2 (MOD): C.2, step (2)); this is more than three times as fast as the total time needed by LEX.
(2) the smallest measured execution time of the generated scanner was about 1.5 seconds (C.2, step (3)) and this is equal (within the relative error) to that of the scanner generated by LEX.
(3) *S-MODIFY* always (re)constructs scanners much faster than *MODIFY* does.Table IV shows that this is caused by the smaller number of states of the generated automaton, and the correspondingly smaller number of states that has to be regenerated.
(4) The combination of S-MOD/C.1 (i.e., *S-MODIFY* with compilation of states with self-transitions) leads to a total scanner generation time that is 17.5 times faster than LEX, at the expense of a generated scanner that is 40% percent slower than the one generated by LEX. The graphical representation of this case in figure 7 shows that we have obtained the desired behavior described in the introduction.

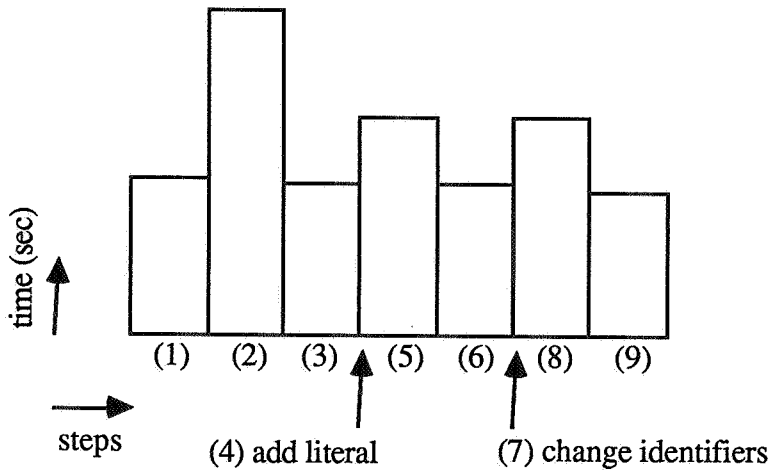

Figure 7. Summary of S-MOD/C.1 on File2.

9. DISCUSSION

As soon as the benefits of lazy implementation techniques become apparent, it is tempting to speculate on possibilities to make the algorithms even lazier than they already are. We see the following possibilities for this:
(1) As already mentioned in section 3.1, *L-CONSTRUCT* is too greedy in the sense that the expansion step of the start state could be postponed until the automaton is used by *L-SCAN*.
(2) In the algorithms as presented, we have assumed the availability of the values of *followpos*. In the implementation of ISG, this information is computed as soon as a regular expression is added to the set of regular expressions. This could be replaced by a lazy computation of *followpos*.
(3) As already mentioned in section 4.1, *MODIFY* is too greedy.
(4) In *EXPAND* and *L-SCAN*, *all* transitions and states to which these transitions lead are constructed when an unmarked state is used during scanning. It might be advantageous to construct only the needed transition and leave the other ones unexpanded.
(5) Another potential area of improvement is the garbage collection of states after each modification of an automaton. *REACHABLE* removes all states which are not reachable from the start state of a partially generated automaton. A more conservative method would only remove those states which can never become part of a future expansion of the partial automaton. Such states are characterized by a set of positions in which one or more of the positions are obsolete, i.e., the labelled, regular expression in which they occurred has been deleted from the set of regular expressions.

The compilation of regular expressions to finite automata is an old and well-understood subject. However, in the literature not much attention has been paid to lazy and incremental techniques. In [ASU86], section 3.7, a technique for "lazy transition evaluation" is mentioned which is very similar to ours. The purpose of that technique is to reduce the size of a generated DFA. The transition function is stored in a cache and transitions are only computed when needed; when the cache overflows, previously computed transitions are removed. It seems that this technique has been applied in the implementation of the UNIX tool *egrep*, but, as far we know, no detailed description of it has been published. In the implementation of ISG, we have not yet taken advantage of recent insights in the efficiency of implementation techniques for lexical scanners [W86].

Our results on the incremental modification of sets of regular expressions and the reuse of the corresponding, partially generated, automata seem to be new. The general principles of lazy and incremental program generation techniques are discussed in [HKR87]. We have implemented a lazy/incremental parser generator IPG accepting finitely ambiguous context-free grammars using techniques similar to the ones described here [HKR88].

10. REFERENCES

[A80] A.V.Aho, "Pattern matching in strings", **in** R.V.Book (*ed.*), *Formal Language Theory*, Academic Press, New York, 1980, pp. 325-347.

[ASU86] A.V.Aho, R.Sethi, & J.D.Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Reading, Mass., 1986.

[BS87] G. Berry & R. Sethi, "From regular expressions to deterministic automata", INRIA report 649, 1987.

[CH86] J. Chailloux, M. Devin, F. Dupont, J.M. Hullot, B. Serpette & J. Vuillemin, "Le_Lisp Version 15.2 -- Le manuel de référence", INRIA report, 1986.

[HK87] J. Heering & P. Klint, "A syntax definition formalism", **in** *ESPRIT'86: Results and Achievements*, North-Holland, 1987, pp. 619-630.

[HKKL86] J.Heering, G.Kahn, P.Klint & B.Lang, "Generation of interactive programming environments", **in** *ESPRIT'85, Status Report of Continuing Work, Part I*, 1986, North-Holland, pp. 467-477.

[HKR87] J.Heering, P.Klint & J.Rekers, "Principles of lazy and incremental program generation", Centrum voor Wiskunde en Informatica, Report CS-R8749, 1987.

[HKR88] J.Heering, P.Klint & J.Rekers, "Incremental generation of parsers", Centrum voor Wiskunde en Informatica, in preparation.

[L75] M.E.Lesk, "LEX - a lexical scanner generator", CSTR 39, Bell Laboratories, Murray Hill, N.J., 1975.

[MY60] R. McNaughton & H. Yamada, "Regular expressions and state graphs for automata", *IRE Transactions on Electronic Computers,* **EC-9** (1960) pp. 38-47.

[W86] W.M.Waite, "The cost of lexical analysis", Software–Practice and Experience, **16** (1986) 5, pp. 473-488.