1991

P. Klint

Lazy scanner generation for modular regular grammars

Computer Science/Department of Software Technology Report CS-R9158 December

CWI, nationaal instituut voor onderzoek op het gebied van wiskunde en informatica

CWI is the research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a non-profit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands organization for scientific research (NWO).

Lazy Scanner Generation for Modular Regular Grammars

P. Klint

CWI
Department of Software Technology
P.O. Box 4079, 1009 AB Amsterdam,
The Netherlands
and
Programming Research Group, University of Amsterdam

When formal language definitions become large it may be advantageous to divide them into separate modules. Such modules can then be combined in various ways, but this requires that implementations derived from individual modules can be combined as well. This occurs, for instance, in interactive development environments for modular language definitions. In this paper we address the problem of combining regular grammars appearing in separate modules and of combining the lexical scanners generated for them.

Key Words & Phrases: Program generator, lazy and incremental generation of lexical scanners, modular regular grammars, finite automata, subset construction.

1991 CR Categories: D.1.2 [**Programming Techniques**]: Automatic programming; D.3.4 [**Programming Languages**]: Processors - *Compilers, Parsing, Translator writing systems and compiler generators.*

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

Note: Partial support received from the European Communities under ESPRIT project 2177 (Generation of Interactive Programming Environments - GIPE II), and from the Netherlands Organization for Scientific Research (NWO) under the *Incremental Program Generators* project.
Note: This is a completely revised version of a paper that appeared earlier as "Scanner generation for modular regular grammars" in *J.W. de Bakker, 25 jaar semantiek, Liber Amicorum*, Centrum voor Wiskunde en Informatica, Amsterdam 1989.

1. INTRODUCTION

1.1. Background

The benefits of dividing complex systems into several, smaller, modules are well-known. Apart from a reduction in complexity that can be achieved for individual modules, one also introduces the possibility of re-using a module several times. In this paper we will apply the idea of modular decomposition to the definition of formal languages (such as, e.g., programming languages and specification languages), and concentrate on *lexical syntax*,

one of the syntactic aspects that have to be defined for a language. Typically, the lexical syntax defines comment conventions, layout symbols, and the form of identifiers, keywords, delimiters, and constants (e.g., numbers, strings) in a language. The standard method is to use a regular grammar to specify the precise form of the various elements in the lexical syntax and to compile this regular grammar into a deterministic finite state automaton (DFA) to be used for the actual reading of program texts.

Here, we are interested in the problem of how the lexical syntax can be subdivided in separate modules and how DFAs can be obtained from various combinations of these modules. The motivation for this problem comes from two different sources:

- In a setting where definitions for more than one language are being developed, it is natural to construct a set of standard modules defining frequently used notions (e.g., identifiers, floating point numbers, or string constants). Of course, these standard notions may sometimes need adaptation depending on their use (e.g., the letters appearing in identifiers may or may not contain both lower case letters and upper case letters, integer constants may or may not contain hexa-decimal digits).
- In modular specification languages that allow user-definable syntax to be introduced in each module, the composition of modules requires, among others, the composition of lexical syntax.

We came across this problem during the implementation of the ASF+SDF meta-environment: an interactive system for developing and testing modular language definitions [Kli91].

1.2. The Problem

The flexibility we want to achieve can best be illustrated by an example. Consider the grammar shown in Figure 1. It consists of two parts: abbreviations and rules. The abbreviations part defines named regular expression to be used in the rules part. When several regular expressions e_i are associated with one name, we associate with that name a regular expression containing all expressions e_i as alternatives. The actual regular grammar is defined in the rules part. Names appearing in rules can be completely eliminated by textual substitution. The names of rules define the token-name to be associated with a string recognized by that particular rule. Note that more than one rule may recognize the same string; in that case we associate more than one token-name with it.

Abbreviations:

```
M1:  <DIGIT>   = 0 | 1 | ... | 7
M2:  <DIGIT>   = 8 | 9
M3:  <LETTER>  = a | b | ... | z
```

Rules:

```
M4:  <INT>     = <DIGIT>+
M5:  <REAL>    = <INT> "." <INT>
M6:  <ID>      = <LETTER> (<LETTER> | <DIGIT>)*
M7:  <KW>      = if
M8:  <KW>      = end
```

Figure 1. A modular regular grammar.

modules	selection (1)	selection (2)	selection (3)	selection (4)
M1	x	x	x	x
M2	x	o	x	x
M3	x	x	x	x
M4	x	x	o	x
M5	x	x	x	x
M6	x	x	x	o
M7	x	x	x	x
M8	x	o	x	x
sentences	**recognized as**	**recognized as**	**recognized as**	**recognized as**
123	<INT>	<INT>	-	<INT>
678	<INT>	-	-	<INT>
2.8	<REAL>	-	-	<REAL>
abc	<ID>	<ID>	<ID>	-
end	<ID>,<KW>	<ID>	<ID>,<KW>	<KW>
xy9	<ID>	-	<ID>	-

Figure 2. Examples of module selections.

In the example, we define a lexical syntax containing integer constants, real constants, identifiers and the keywords `if` and `end`. Each regular expression in the grammar is labelled with a module name. In general, several expressions may be labelled with the same name, but in this example we have the extreme case that every expression is labelled with a different name. The use of this modular regular grammar is shown in Figure 2. Given a list of selected module names, only those rules are to be used whose module name appears in the selection. For each selection of modules, the modular regular grammar thus corresponds to a (probably) different ordinary regular grammar.

An implementation of modular regular grammars should, clearly, have the following two properties:

- The time needed to construct a DFA for a given selection of modules (in the modular case) should be significantly less than the time needed to construct the automaton from scratch (in the non-modular case) using only the rules from the selected modules.
- The efficiency of the DFA generated in the modular and in the non-modular case should be comparable.

How can modular regular grammars be compiled into DFAs? There are two, fundamentally different, solutions to this problem:

- Compile all rules that are labelled with the same module name into a single DFA and define a composition operation on DFAs. The DFA constructed for a certain selection of modules then consists of the composition of the DFAs constructed for each individual module in the selection.
- Compile all rules into a single DFA and define a selection operation that, given a list of selected modules, extracts the sub-automaton that corresponds to that selection.

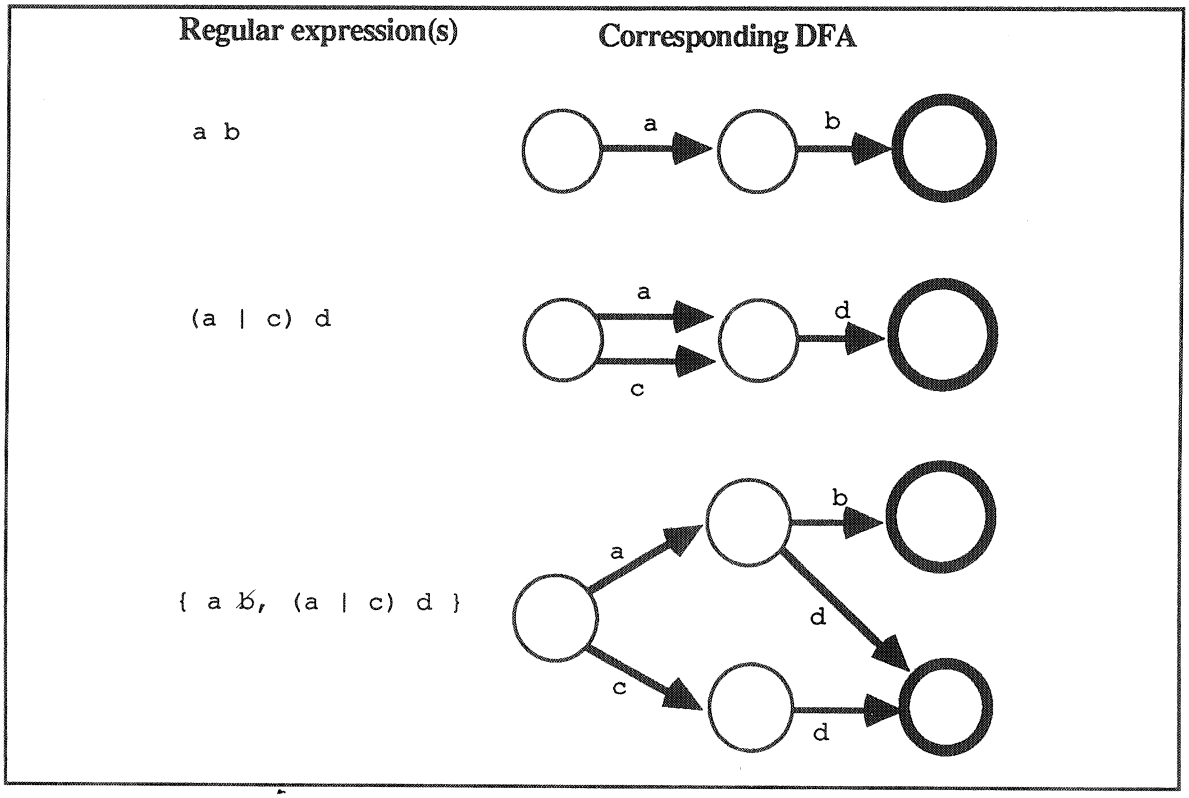

Figure 3. Two DFAs and their composition as DFA.

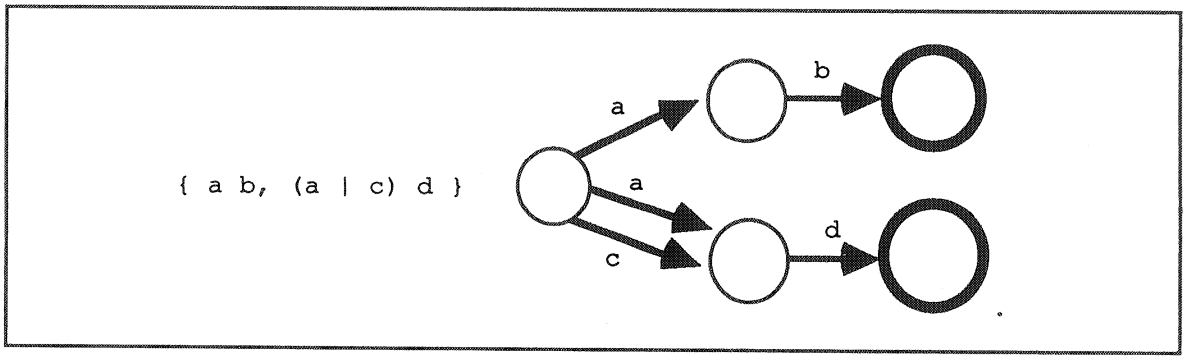

Figure 4. Same DFAs and their composition as NFA.

Obviously, the first solution is the most elegant one since it leads to a truly modular implementation of lexical scanners. Unfortunately, the composition operation on deterministic automata is expensive: given two DFAs A and B and their composition $A \cup B$, in many cases the states of A and B do not appear in $A \cup B$. Instead, they are combined into new states thus reflecting the interactions between the languages recognized by A and B. As a result, the computation of $A \cup B$ requires roughly the same amount of work as the construction of a completely new automaton. This is illustrated in Figure 3, where the DFAs for the two regular expressions `a b` and `(a | c) d` are shown together with the resulting DFA for the combined language `{ a b, (a | c) d }` or, equivalently, `(a b) | ((a | c) d)`. It should be emphasized that the complexity of the composition operation is caused by our (efficiency) requirement that the result of the composition is again a *deter-*

ministic automaton. Allowing a non-deterministic automaton (NFA) as result, would significantly simplify the composition operation as is shown in Figure 4.

1.3. Approach

In this paper, we now concentrate on the second solution mentioned above and investigate how a selection operation can be defined that extracts from a DFA a sub-automaton corresponding to a selection of modules.

Point of departure is the method for *lazy* generation of lexical scanners described in [HKR87]. Unlike the conventional approach, where for a given set of regular expressions a complete DFA is generated before any scanning is done, we only construct the start state of the DFA and then begin scanning input sentences. When necessary for the scanning of a particular sentence, the DFA is extended with new states and transitions. In [HKR87] it was shown that this method of lazy scanner generation can easily be extended to *incremental* scanner generation: modifications of the original set of regular expressions are allowed and result in a modification of the generated DFA.

Here we show that the lazy scanner generation method can also be extended to implement modular regular grammars. Note that the two issues are unrelated—the implementation method for modular regular grammars can also be presented in a more conventional, that is "greedy", form—but we find the lazy presentation more attractive since it is closer to the intended use of modular regular grammars where switching between selections of modules occurs frequently.

2. LAZY COMPILATION OF REGULAR EXPRESSIONS

We sketch an algorithm for the lazy compilation of regular expressions into deterministic finite automata. A full description can be found in [HKR87], a brief description can be found in [HKR91].

2.1. Preliminaries

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

Regular expressions over a finite alphabet Σ are composed of the symbols from that alphabet, the empty string (ε), the operators concatenation (denoted by juxtaposition), alternation (|), repetition (*), and parentheses. 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. We will also use the undefined regular expression ($\perp$) denoting the empty set of strings, i.e. $\perp$ does not recognize anything. The following identities characterize the interactions between concatenation, |, * and $\perp$:

(a) $r \perp = \perp r = \perp$

(b) $\perp^* = \varepsilon$

(c) $r \mid \perp = \perp \mid r = r$

A *labelled regular expression* is a regular expression in which a unique natural number p is associated with each occurrence of a symbol $a \in \Sigma$. 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.

We will use some auxiliary functions on labelled regular expressions which describe properties of the strings recognized by them:

(1) The predicate *nullable* determines whether a regular expression can recognize the empty string.

(2) The function *firstpos* maps a labelled regular expression to the set of positions that can match the first symbol of an input string.
(3) The function *lastpos* maps a labelled regular expression to the set of positions that can match the last symbol of an input string.
(4) The function *followpos* maps a position in a labelled, regular expressions 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 $followpos(p, e)$.

For precise definitions of these functions we refer the reader to [HKR87] or [BS87].

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 $\Sigma \setminus \{\$\}$.

An *accepting sequence of positions* for a labelled regular expression e can now be defined as a sequence of positions $p_1,...,p_n$ such that $p_1 \in firstpos(e)$, $p_n \in lastpos(e)$, and $p_{i+1} \in followpos(p_i, e)$, $i=1,...,n-1$. For all strings $s \in \Sigma^*$ and for all terminated, labelled, regular expressions 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 $a_i = symbol(p_i)$, $i=1,...,n$. (see [YM60], Theorem 3.1).

2.2. Algorithms for the lazy construction of a DFA

Using the notions introduced in the previous section we now formulate an algorithm for the lazy 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. 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.

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*. When a state S is added to *States*, it is unexpanded and $expanded(S)$ = **false** holds. A state $S \in States$ can be marked as expanded by setting $expanded(S)$:= **true**.

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

In standard DFA construction algorithms, a complete DFA is computed for a given regular expression. In the following lazy algorithm only a Partial DFA (PDFA) is constructed which is further extended when needed during scanning of given input strings.

First, we give the algorithms for the lazy construction of the start state and for the expansion of a state.

Algorithm *L-CONSTRUCT*
Construction of the initial part of the PDFA 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, $A.start$))

Algorithm *EXPAND*
Expansion of a PDFA 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 $U \neq \emptyset$ **then**
    if $U \notin A.States$ **then** $A.States := A.States \cup \{U\}$; $expanded(U) :=$ **false fi**
    $A.Trans(S,a) := U$
  fi
od
$expanded(S) :=$ **true**
return(A)

For later reference it is useful to emphasize that the existence of a transition between two states S and T on alphabet symbol a may be caused by *several* positions in S that correspond to the symbol a. State T will contain as subsets the, possibly overlapping, sets of follow positions for each of these positions in S. This situation is illustrated in Figure 5.

From the definition of *EXPAND* 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.

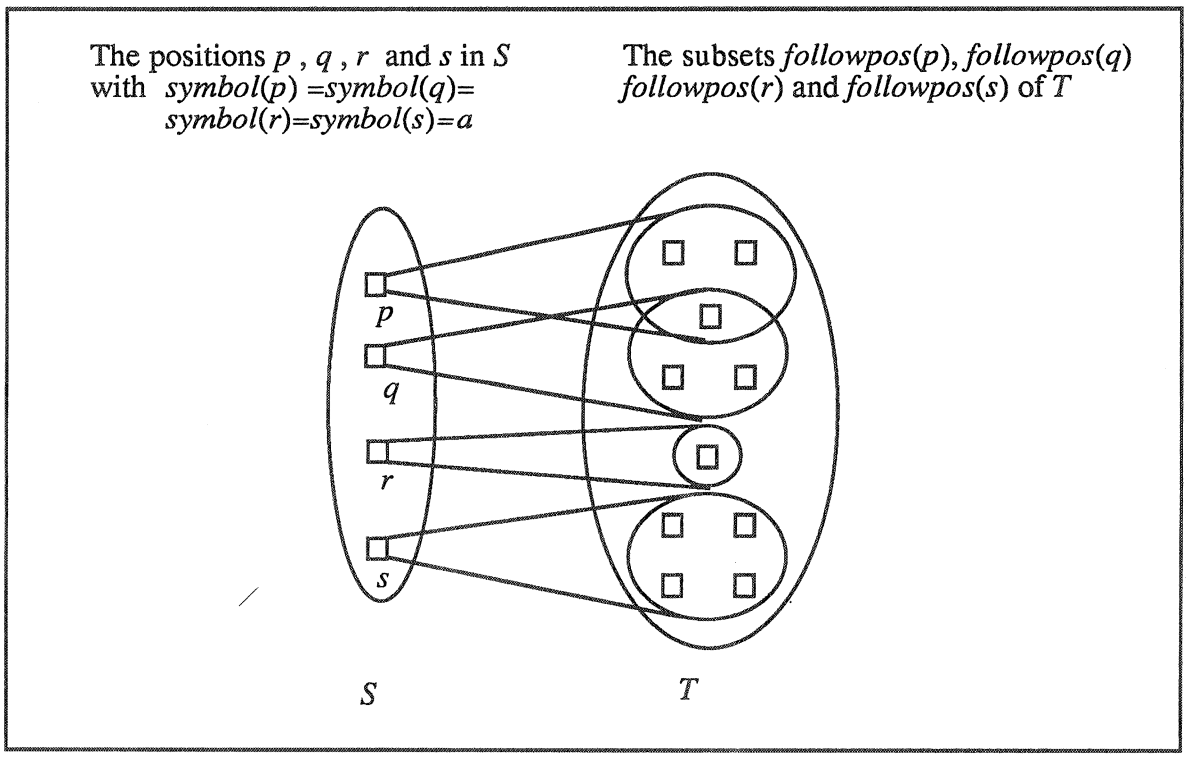

Figure 5. Positions causing a transition between states S and T on symbol a.

Finally, we give the scanning algorithm associated with *L-CONSTRUCT*. It performs expansions of needed, unexpanded, 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 $\neg$ *expanded*(S) **then** $A :=$ *EXPAND*(E, A, S) **fi**
  S:= *A.Trans*(S, a_i)
  $i := i + 1$
od
return (*FINAL*(S), A)

The last state reached during the scanning of an input string determines whether the input string should be accepted or rejected. A state is accepting if one of its positions corresponds to the end marker \$. This 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) = \$]

Note that a state may contain several positions with symbol \$. This may happen when a string is recognized by more than one rule in the regular grammar.

3. AN ALGORITHM FOR COMPILING MODULAR REGULAR GRAMMARS

3.1. Modular regular expressions versus modular regular grammars

Before generalizing these lazy scanner generation techniques to the case of modular regular grammars, we first need a definition of modular regular grammars. It would be natural to describe them as (module-name, regular expression) pairs. However, it turns out that not all sub-expressions of a regular expression need to originate from the same module. This will become clear when discussing named regular expressions in Section 3.3. Therefore, we choose a method that allows more refined control over the module information and associate module names with the *positions* in a (terminated, labelled) regular expression and not with the regular expression as a whole. We will write *module*(p) to denote the module name associated with position p and we will write ${}_m a_p$ to denote a position p such that *symbol*(p)=a and *module*(p)=m. We will call these regular expressions with associated module information *modular regular expressions*. In this section, we will only use sets of modular regular expressions. In Section 3.3, *modular regular grammars* will be introduced and we will show how they can be reduced to sets of modular regular expressions.

Given a modular regular expression e and a list of module names M we can now *restrict* expression e to M (notation: e/M) by replacing all ${}_m a_p$ in e with $m \notin M$ by the undefined expression $\perp$. We extend the restriction operator / to sets of regular expressions.

The problem we want to solve can now be formulated as follows: given a set of modular regular expressions E, a partially constructed automaton A for these regular expressions, and a list of module names M, can we select a part of A that precisely recognizes the language defined by E/M?

The simplest method one can imagine to restrict the language accepted by a given DFA is to use the DFA as it is, but impose restrictions on accepting states according to the current selection of modules. This method would only require some recomputations on the accepting states of the DFA.

Consider, for instance, the set of expressions E= { ${}_m a_1\ {}_m b_2\ {}_m \$_3$, ${}_n c_4\ {}_n d_5\ {}_n \$_6$ } and the corresponding DFA A shown in Figure 6 (the border lines of accepting states are shown in bold face). Choosing the set of modules {m}, E/{m} is then equal to { ${}_m a_1\ {}_m b_2\ {}_m \$_3$, $\perp$ } = { ${}_m a_1\ {}_m b_2\ {}_m \$_3$ } and the automaton obtained from A by only retaining accepting states that are labelled with a position in the selection {m} correctly recognizes the language defined by E/{m} (see Figure 7).

However, on closer inspection it turns out that this simple method may be incorrect when the positions in a single modular regular expression are labelled with different module names. This is illustrated by the following counter example. Consider

E = { (${}_m a_1$ | ${}_n b_2$) ${}_n c_3\ {}_n \$_4$ }

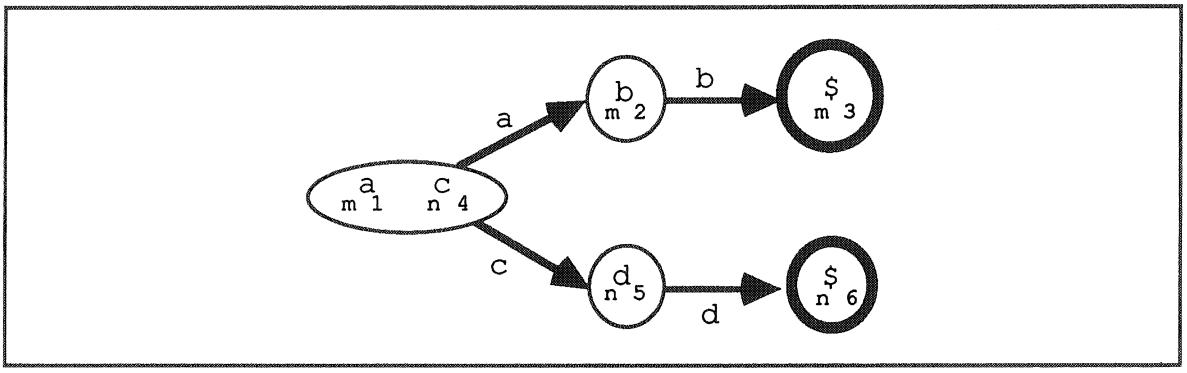

Figure 6.

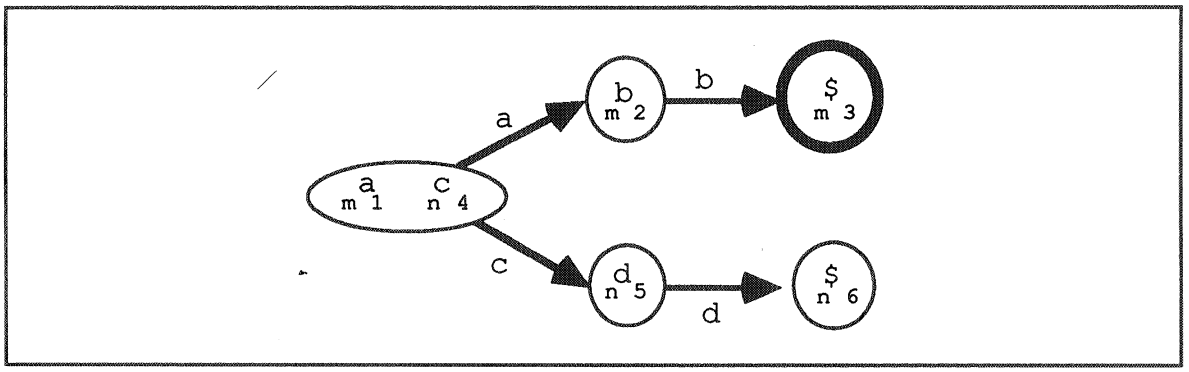

Figure 7.

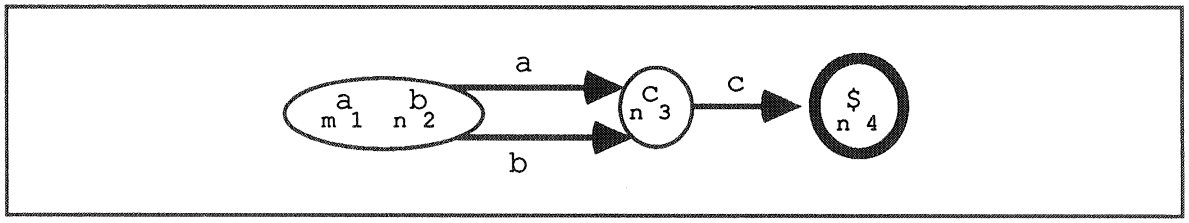

Figure 8.

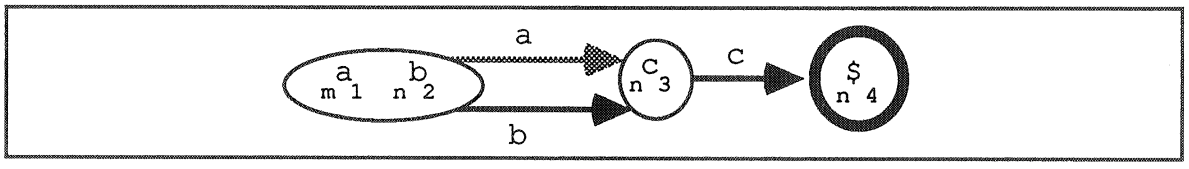

Figure 9.

with corresponding DFA shown in Figure 8. When we restrict E to the single module {n} we obtain

$$E/\{n\} = \{ (\perp \mid {}_n b_2) \; {}_n c_3 \;\; {}_n \$_4 \} = \{ {}_n b_2 \;\; {}_n c_3 \;\; {}_n \$_4 \}$$

but the string ac will (erroneously) be accepted using the simple method of restricting the accepting states of the DFA corresponding to E. The reason is, of course, that it is not sufficient to require that the accepting position is in the current selection of modules, as long as it can be reached using transitions that do *not* belong to that selection, e.g., the transition from the start state on symbol a. Therefore, we should remove all such invalid transitions obtaining the DFA shown in Figure 9 (invalid transitions are represented by shaded arrows).

Following this second method, we restrict the language accepted by a given DFA by eliminating all transitions in the DFA that do not correspond to the regular expressions in the current selection. This method requires the calculation of modifications to the transition table of the DFA for each new selection of modules.

3.2. Restricting a PDFA to a selection of modules

Remains the problem of formulating criteria to decide when a transition between two states S and T on alphabet symbol a is still valid in the current selection of modules. Our goal is to restrict the automaton A in such a way that it is equivalent to an automaton A' that would have been constructed when using the restricted set of regular expressions right from the start. In other words a transition should be valid in the restricted automaton A when it would have been constructed in A' as well. Looking at the way expansion of states is defined (see Section 2.2, algorithm *EXPAND*) we observe that the existence of a transition between S and T on symbol a implies that

(1) S contains a position p whose symbol a;
(2) position p is reachable from the start state of the automaton,
(3) T contains some position q in the set of follow positions of p.

Implication (2) is trivially true in an ordinary automaton, but plays an important role in the restricted one.

In the restricted automaton one should impose the additional restriction that positions p and q are labelled with module names appearing in the current selection.

Referring to Figure 5 given in Section 2.2, we illustrate the situation in the modular case in Figure 10. Positions that are labelled with a module in the current selection are indicated by a black square. Assuming that q is reachable from the start state, a transition between S and T on symbol a is valid in this particular selection, since q is selected and *followpos*(q) contains a selected position as well. In this case, q is the only position that supports this transition! Note that states S and T correspond to states S' and T' in automaton A' that contain *only* these selected positions.

Given a PDFA A and a list of selected modules M, we have to compute those parts of the transition table that are still valid in this new selection. We introduce the following notions to achieve this goal:

(1) The table containing the transitions that are valid in the current selection will be called *SelTrans*, it is always a subset of the complete (but perhaps only partially computed) transition table *Trans* of A.

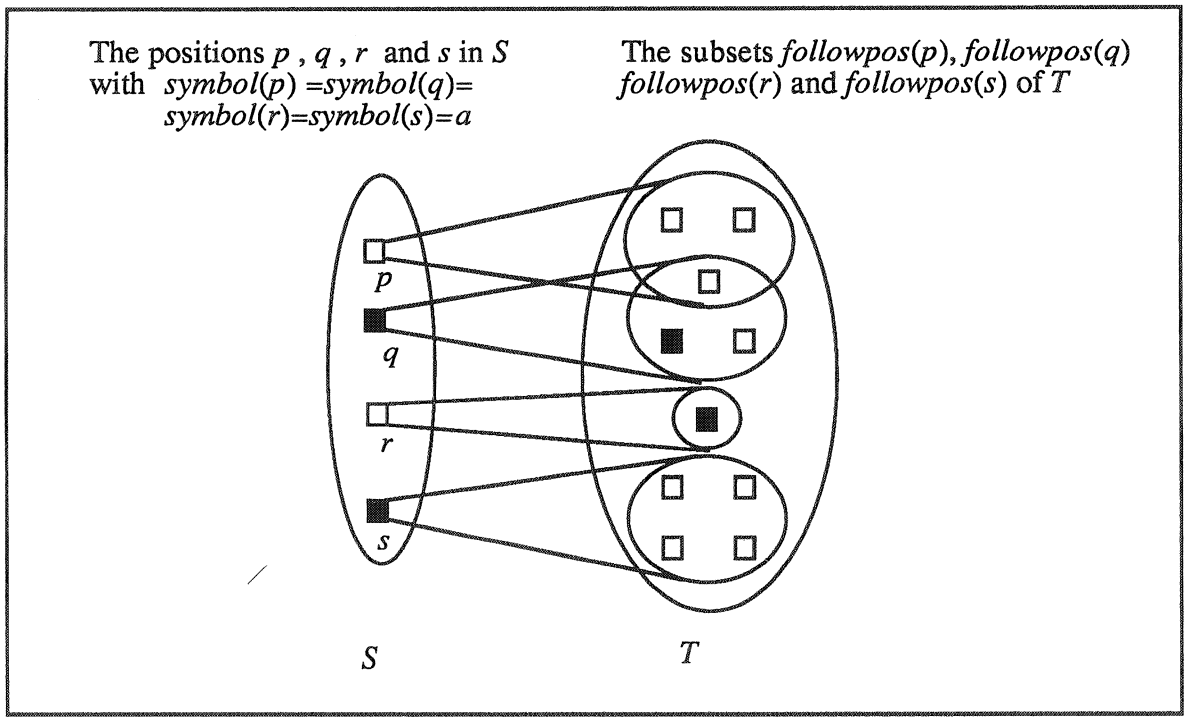

Figure 10. Positions causing a transition between states S and T on symbol a in the modular case.

(2) With each occurrence of a position in a set of positions we associate a color: *green* positions are reachable from the start state, while for *red* positions this can not (yet) be determined.

(3) With each state S in A we associate a 3-valued function *specialized:*

specialized(S) = **false**

state S has not yet been encountered during this specialization, its outgoing transitions are undetermined as well the colors of its positions.

specialized(S) = **partial**

state S has already been encountered during this specialization, its own outgoing transitions are still undetermined but the colors of its positions have been initialized to red (and perhaps they have already been partially redefined during the expansion of other states with outgoing transitions to S).

specialized(S) = **true**

state S has been visited during this specialization, both its outgoing transitions and the colors of its positions have been determined completely.

Before actual scanning starts, all states have to be set to unspecialized (*BEGIN-SPECIALIZATION*). Next, the specialization of a single state has to be defined (*SPECIALIZE*). Finally, the actual scanning is defined (*M-SCAN*).

Algorithm *BEGIN-SPECIALIZATION*
Input. A set E of modular regular expressions, a PDFA A, and a list of modules M.
Output. A modified version of A in which the start state is specialized and for all other states the specialization is set to uninitialized.
Method.

```
for ∀ S∈ A.States do specialized(S) := false od
for ∀ p∈ A.start do color(p, A.start ) := if module(p)∈ M then green else red fi od
specialized(A.start) := partial
return(A)
```

Algorithm *SPECIALIZE*
Input. A set E of modular regular expressions, a PDFA A, a state S, and a list of modules M.
Output. A modified version of A in which all valid transitions from state S in the modules in M have been recorded in *A.SelTrans*.
Method.

```
for ∀ a∈ Σ\{$}
do let T = A.Trans(S,a)
   in  if T ≠∅ then
          if specialized(T) = false ∧ T ≠ S then
            for ∀ q∈ T do color(q, T) := red od
            specialized(T) := partial
          fi
          support := false
          for ∀ p∈ S
          do if [symbol(p) = a ∧ color(p, S) = green]
             then for ∀ q∈ followpos(p, E)
                  do if module(q) ∈ M then color(q, T) := green; support := true fi
                  od
             fi
          od
          A.SelTrans(S,a) := if support then T else ∅ fi;
          specialized(S) := true
       fi
   ni
od
return(A)
```

Remains to be described how specialization and expansion of states interact. During scanning states are encountered that are either not yet expanded (and should be both expanded and specialized) or expanded but not yet specialized (and should be specialized). In addition, the specialization information of new states created during expansion should be initialized properly. This is important since the positions in an unexpanded state may change color before the state itself is expanded. This is achieved by simply expanding a state and immediately specializing it.

Here is, finally, the scanning algorithm:

Algorithm *M-SCAN*
Simulate a given PDFA for a given selection of modules on a given input string, incrementally expanding and specializing the PDFA when necessary.
Input. A set E of modular regular expressions, a corresponding PDFA A, a list of modules M, 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 $\neg$ *expanded*(S) **then** A := *EXPAND*(E, A, S) **fi**;
  if *specialized*(S) $\neq$ **true then** A := *SPECIALIZE*(E, A, S, M) **fi**;
  S:= A.*SelTrans*(S, a_i)
  i := i + 1
od
return (*M-FINAL*(S), A)

In the modular case, a state is accepting if one of its green positions corresponds to the end marker \$. This is defined by the following algorithm.

Algorithm *M-FINAL*
Determine whether a given state is an accepting state in a given selection of modules.
Input. A state S.
Output. **true** or **false**
Method.

return $\exists\, p \in S$ [*symbol*(p) = \$ $\wedge$ *color*(p, S) = *green*]

3.3. An example

Consider the regular expression: $(_m a_1 |_n b_2)^*{}_m a_3\ {}_m b_4\ {}_m b_5\ {}_m \$_6$. Its fully expanded automaton specialized for {m} is shown in Figure 11a. Green positions are displayed in an outline font and, as before, invalid transitions are represented by shaded arrows.
It is interesting to compare this specialized automaton with the automaton in Figure 11b that would be obtained for the equivalent expression $_m a_1{}^*{}_m a_3\ {}_m b_4\ {}_m b_5\ {}_m \$_6$ using *L-CONSTRUCT* alone.

3.4. Correctness and complexity of SPECIALIZE

The specialization of PDFA A for selection M (denoted by A/M) described by *SelTrans* has three interesting properties:

- For each string in the language defined by the restricted set of regular expressions E/M, the accepting sequence of positions in A/M is identical to the accepting sequence as it would occur in the new automaton A' that is constructed independently for the restricted set of regular expressions E/M.

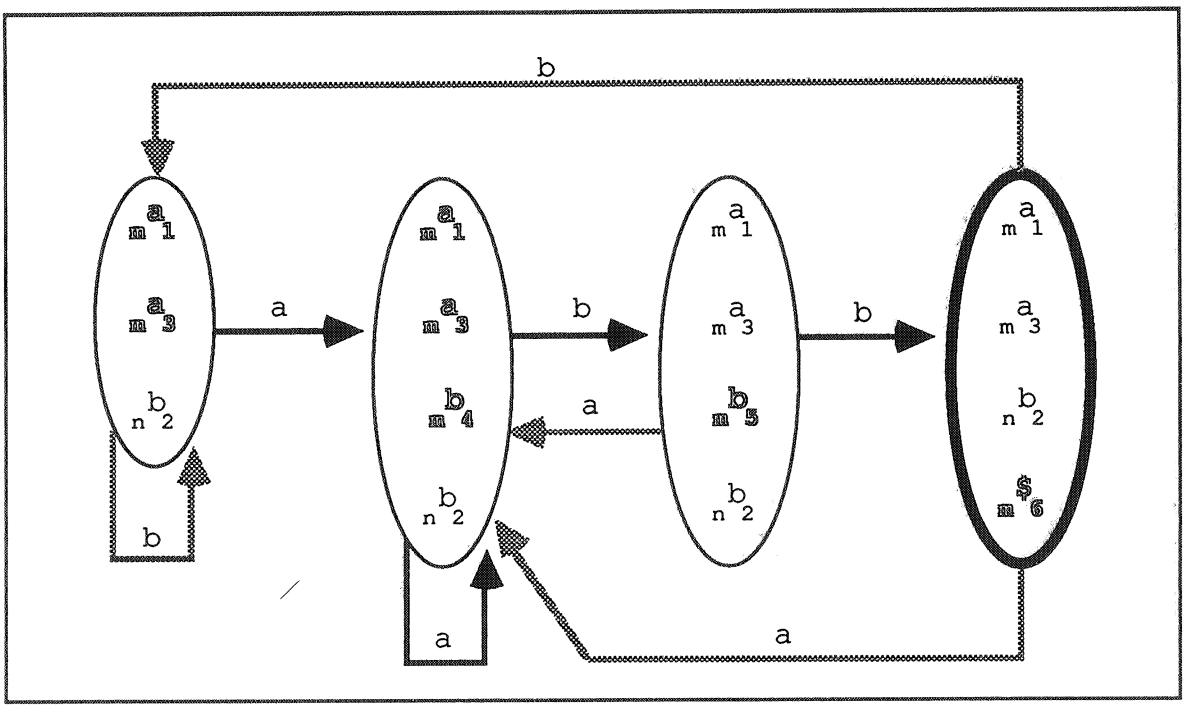

Figure 11a. Automaton for $(_{m}a_1 \mid _{n}b_2)^*{}_{m}a_3\ {}_{m}b_4\ {}_{m}b_5\ {}_{m}\$_6$ specialized for {m}.

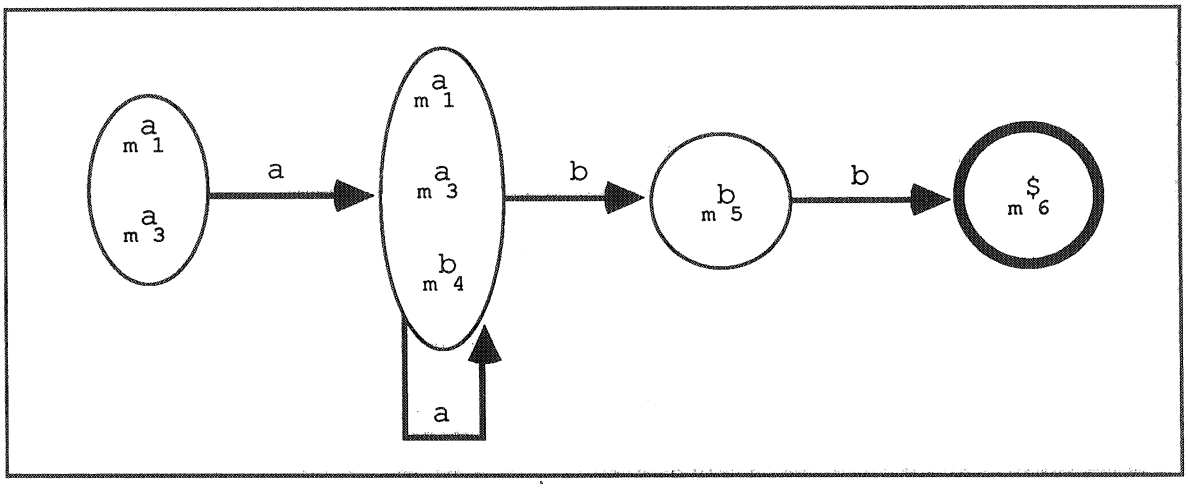

Figure 11b. Automaton for the simplified expression $_{m}a_1{}^*{}_{m}a_3\ {}_{m}b_4\ {}_{m}b_5\ {}_{m}\$_6$.

- The above mentioned automaton *A'* does *not* need to occur as sub-automaton of *A/M*, since two distinct states *S* and *T* in *A/M* may become effectively equivalent due to specialization (i.e., after specialization *S* and *T* contain the same subset of green positions and will thus behave identically; they correspond to a single state in automaton *A'*), but they will remain distinct states in *A/M*.
- We have not made a detailed comparison between the complexities of *EXPAND* and *SPECIALIZE* . The former constructs unions of sets (of positions) and has to perform a complex membership test to determine whether a newly constructed state already exists, while the latter does no set construction at all but only performs pair-wise comparisons of set elements. Assuming that set construction and the

membership test for sets of sets are the most expensive operations, it is to be expected (and confirmed by our implementation, see Section 4) that *SPECIALIZE* is cheaper than *EXPAND*.

3.3. Modular regular grammars

In the previous section we have discussed lexical definitions that have the form of a list of modular regular expressions. Each position occurring in these expressions is labelled with both a position name and a name of a module. Now we turn our attention to the complete modular regular grammars as sketched in Section 1. Such a grammar consists of two parts: abbreviations and rules. Both abbreviations and rules contain triples of the form

module-name: *token-name* = *regular-expression*.

Names appearing in a regular expression should always have been defined by a previous abbreviation or rule and can always be eliminated by textual substitution. When several regular expressions e_i are associated with one name, we associate with that name a regular expression containing all expressions e_i as alternatives. We will now show how a modular regular grammar of this form can be reduced to a set of modular regular expressions as defined in Section 3.1. We proceed in four stages:

1. Associate module names and positions with all alphabet symbols in the modular regular grammar.
2. Replace all uses of names in regular expressions by their definition (after inserting parentheses where necessary).
3. Terminate all resulting expressions in the rules section with a $ symbol and associate both the module name preceding the expression and a new position with that $ symbol. When this terminator appears in a state (= set of positions) it will uniquely identify this rule. This fact can be used to determine the token-name to be associated with the recognized input string.
4. The set of modular regular expressions obtained in step 3 is the reduced form of the original modular regular grammar.

Only step 2 is non-trivial and requires some further comments, since what will happen when a named expression that is *not* selected is used in a expression that *is* selected? Intuitively, one would like to replace the use of the named expression by $\perp$. It turns out that this can be achieved by an appropriate definition of textual substitution.

Let e be the regular expression associated with some name in a modular, regular grammar E. Define $e' = copy(e)$ as the labelled regular expression obtained by taking a literal copy of e, with the exception that each symbol a_p appearing in e is replaced by $a_{p'}$, where p' is a new, unique, position label and we define $module(p')=module(p)$. It is important to note that the module name associated with each position remains the same.

Using this definition of taking a copy of a regular expression, all named regular expressions can be removed from a grammar by replacing each occurrence of a name by a copy of its associated expression. The positions occurring in the resulting, expanded, regular expression may be labelled with different module names (this possibility was already mentioned in Section 3.1).

We conclude this section by applying all the techniques described so far to the modular regular grammar given as example in Figure 1 (see Section 1). In Figure 11, the complete

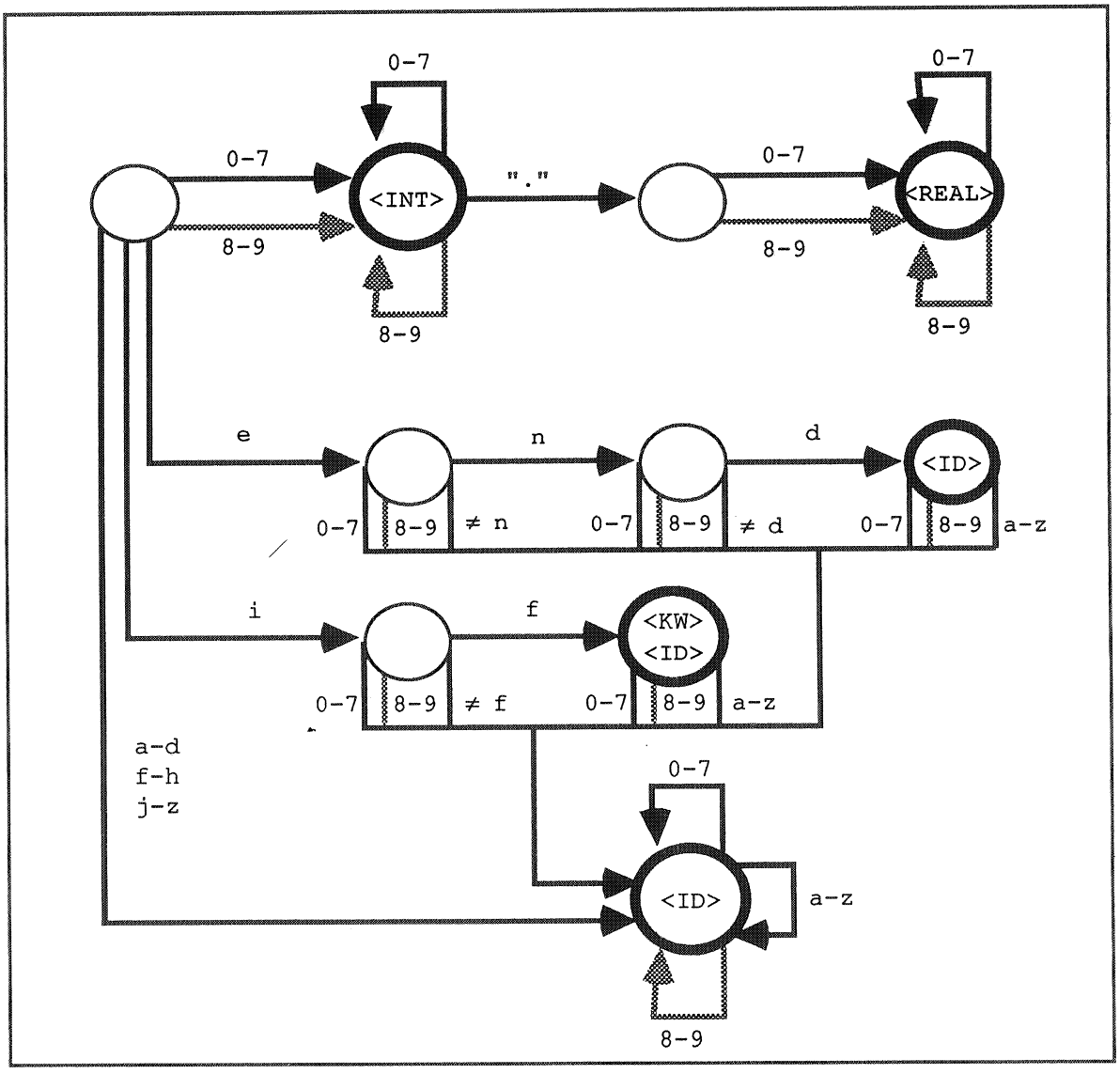

Figure 11. DFA corresponding to the selection { M1, M3, M4, M5, M6, M7 }.

DFA corresponding to this grammar is shown for the selection of modules { M1, M3, M4, M5, M6, M7 }, in others words the modules M2 (that includes the digits 8 and 9 in the definition of <DIGIT> and M8 (the keyword end) are not selected. The following conventions have been used in this figure:

- Invalid transitions are (as before) indicated by shaded arrows.
- Potentially accepting states are labelled with the name of the accepting token.
- The abbreviation $\neq c$ stands for all letters a-z, except the letter *c*.

Note that all transitions labelled with 8-9 are invalid and that the state that could potentially recognize end both as a keyword and as an identifier can only recognize it as an identifier in this particular selection.

4. CONCLUDING REMARKS

The algorithms presented in this paper have been implemented as extension of the Incremental Scanner Generator (ISG) described in [HKR87]. The resulting Modular Scanner Generator (MSG) shows that the method of selecting a sub-automaton from the large automaton corresponding to *all* regular expressions in *all* modules is by far superior over the method of constructing a new automaton for each selection of modules. In typical cases, the construction of a new automaton would require several seconds, while our method based on selection only requires several *tenth* of seconds.

In our particular setting where the interactive development of formal language definitions is the major goal, the quasi simultaneous editing of many specification modules is required. Switching between the editing of modules implies, among many other things, switching between lexical scanners. This is precisely the functionality provided by MSG.

In addition, MSG supports the incremental modification of modular regular grammars. As a result, one can—in arbitrary order—switch between modules, change them or use them.

As indicated in Section 1, the modularization of language definitions implies that all parts of such a definition have to be processed in a modular fashion. In [Rek89], a technique is sketched for the generation of parsers for modular context-free grammars. It turns out that the general techniques for lazy and incremental program generation as described in [HKR91], form a good foundation for program generation techniques for modular specification formalisms.

ACKNOWLEDGEMENTS

Jan Rekers made several comments on a draft of this paper and Marcel Wijkstra found a serious error in an earlier version of one of the algorithms. The technique presented here was inspired by discussions with Jan Heering and Jan Rekers on lazy/incremental program generation techniques.

REFERENCES

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

[HKR87] J. Heering, P. Klint & J. Rekers, "Incremental generation of lexical scanners", Centre for Mathematics and Computer Science, Report CS-R8761.

[HKR91] J. Heering, P. Klint & J. Rekers, "Principles of lazy and incremental program generation" (revised version), Centre for Mathematics and Computer Science, Report CS-R9124, 1991.

[Kli91] Klint, P., A meta-environment for generating programming environments, in J.A. Bergstra & L.M.G. Feijs (eds), *Algebraic Methods II: Theory, Tools, and Applications*, Lecture Notes in Computer Science, Vol. 490, Springer-Verlag, 1991, 105-124.

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

[Rek89] Rekers, J. Modular parser generation. Report CS-R8933, Centre for Mathematics and Computer Science, 1989.