
Efficient Generation of
Table-Driven Scanners

J. Grosch

DR. JOSEF GROSCH

COCOLAB - DATENVERARBEITUNG

GERMANY

Cocktail

Toolbox for Compiler Construction

Efficient Generation of Table-Driven Scanners

Josef Grosch

May 24, 1987

Document No. 2

Copyright © 1994 Dr. Josef Grosch

Dr. Josef Grosch
CoCoLab - Datenverarbeitung
Breslauer Str. 64c
76139 Karlsruhe
Germany

Phone: +49-721-91537544
Fax: +49-721-91537543
Email: grosch@cocolab.com

SUMMARY

General tools for the automatic generation of lexical analysers like LEX [Les75] convert a specification consisting of a set of regular expressions into a deterministic finite automaton. The main algorithms involved are subset construction, state minimization, and table compression. Even if these algorithms do not show their worst case time behaviour they are still quite expensive. This paper shows how to solve the problem in linear time for practical cases, thus resulting in a significant speed-up. The idea is to combine the automaton introduced by Aho and Corasick [AhC75] with the direct computation of an efficient table representation. Besides the algorithm we present experimental results of the scanner generator Rex [Groa] which uses this technique.

KEY WORDS lexical analysis scanner generator

INTRODUCTION

Today, there exist several tools to generate lexical analysers like e.g. LEX [Les75], flex [Pax88], GLA [Heu86, Wai86], ALEX [Mös86], or Mkscan [HoL87]. This indicates that the automatic implementation of scanners becomes accepted by today's compiler writers. In the past, the low performance of generated scanners in comparison to hand-written ones may have restricted the generation approach to prototyping applications. However, since newer results [Grob, Heu86, Wai86] show that generated scanners have reached the speed of hand-written ones there is no argument against using automatically generated lexical analysers in production compilers. In the following we present an efficient algorithm to convert a scanner specification based on regular expressions into a deterministic finite automaton.

A specification of a scanner consists of a set of regular expressions (REs). Each RE usually describes a deterministic finite automaton (DFA) being able to recognize a token. The whole set of REs, however, usually specifies a non-deterministic finite automaton (NFA). To allow scanning to be done in linear time the NFA has to be converted into a DFA. This problem can be solved with well known algorithms for subset construction and state minimization [ASU86, WaG84]. In the worst case subset construction takes time $O(2^n)$ and state minimization $O(n^2)$ or $O(n \log n)$ if n is the number of states. If the DFA is implemented as a table-driven automaton an additional algorithm for table-compression is needed in practice, usually taking time $O(n^2)$.

Running example:

```
letter ( letter | digit ) *   → IdentSymbol
digit +                      → DecimalSymbol
octdigit + Q                 → OctalSymbol
BEGIN                        → BeginSymbol
END                          → EndSymbol
:=                           → AssignSymbol
```

The above specification describes six tokens each by a RE using an abstract notation. The automaton for these tokens is a NFA whose graphical representation is shown in Figure 1. The conversion of this NFA to a DFA yields the automaton shown in Figure 2.

Definition 1: constant RE

A RE consisting only of the concatenation of characters is called a *constant* RE. A constant RE contains no operators like $+$ $*$ $|$ $?$

The language of a constant RE contains exactly one word. In the above example the constant REs are:

```
BEGIN  END  :=
```

During the construction of tables for a DFA by hand we observed that the task was solvable easily and in linear time for constant REs. Common prefixes have simply to be factored out thus arriving at a DFA having a decision tree as its graphical representation. Only the few non-constant REs required real work.

Scanner specifications for languages like Modula or C usually contain 90% constant REs: 60% keywords and 30% delimiters. Only 10% are non-constant REs needed to specify identifiers, various constants, and comments (see Appendix 2 for an example). The languages Ada and Pascal are exceptions from this observation because keywords can be written in upper or lower case letters or in any mixture.

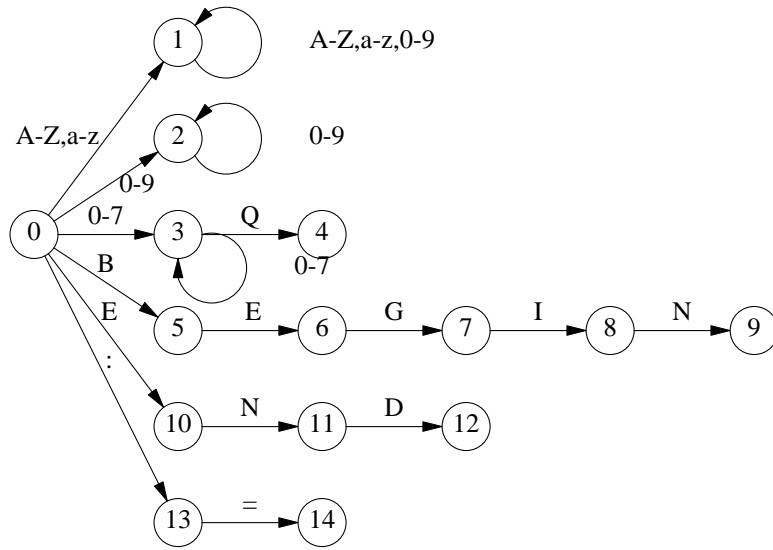


Fig. 1: NFA for the running example

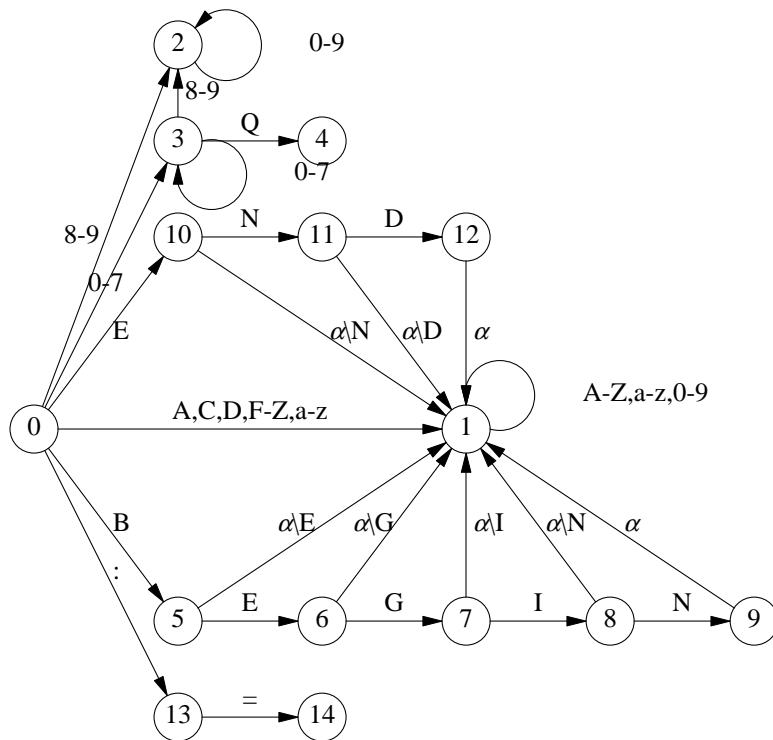


Fig. 2: DFA for the running example ($\alpha \setminus \beta$ stands for A-Z,a-z,0-9 except β)

It would be nice to retain the linear time behaviour for constant REs and perform subset construction and state minimization only for the few non-constant REs. The following chapter describes how this can be achieved by first converting the NFA of the non-constant REs to a DFA and incrementally adding the constant REs afterwards. The results of this method are shown for the running example. Then we generalize the method to automata with several start states. We conclude with a comparison of some scanner generators and present experimental results.

THE METHOD

The basic idea is to combine the special automaton of Aho and Corasick [AhC75] with the so-called "comb-vector" or row displacement technique [ASU86] for the table representation. The automaton of Aho and Corasick, called a pattern matching machine, extends a usual DFA by a so-called failure function and was originally used to search for overlapping occurrences of several strings in a given text. This automaton can also be used to cope with a certain amount of nondeterminism.

To introduce the terminology needed we present a definition of our version of the automaton. We call it a *tunnel automaton* and the tunnel function used corresponds to the failure function of Aho and Corasick. The name tunnel automaton is motivated by the analogy to the tunnel effect from nuclear physics: Electrons can switch over to another orbit without supply of energy and the tunnel automaton can change its state without consumption of an input symbol.

Definition 2: tunnel automaton

A *tunnel automaton* is an extension of a DFA and consists of:

StateSet	a finite set of states	
FinalStates	a set of final states	$\text{FinalStates} \subseteq \text{StateSet}$
StartState	a distinguished start state	$\text{StartState} \in \text{StateSet}$
Vocabulary	a finite set of input symbols	
Control	the transition function	$\text{Control}: \text{StateSet} \times \text{Vocabulary} \rightarrow \text{StateSet}$
NoState	a special state to denote undefined	$\text{NoState} \in \text{StateSet}$
Semantics	a function mapping the states to subsets of a set of procedures	$\text{Semantics}: \text{StateSet} \rightarrow 2^{\text{Proc}}$
Tunnel	a function mapping states to states	$\text{Tunnel}: \text{StateSet} \rightarrow \text{StateSet}$

A tunnel automaton usually works like a DFA: in each step it consumes a symbol and performs a state transition. Additionally the tunnel automaton is allowed to change from state s to state $t = \text{Tunnel}(s)$ without consuming a symbol, if no transition is possible in state s . In state t the same rules apply, so the automaton may change the state several times without consuming any symbols. After recognizing a token a semantic procedure associated with the final state is executed. Algorithm 1 formalizes the behaviour of a tunnel automaton. To guarantee the termination of the WHILE loop the StateStack is initialized with a special final state called DefaultState.

Algorithm 1: tunnel automaton

```

BEGIN
  Push (StateStack , DefaultState)
  Push (SymbolStack, Dummy      )
  State  := StartState
  Symbol := NextSymbol ( )
  LOOP
    IF Control (State, Symbol)  $\neq$  NoState THEN
      State := Control (State, Symbol)
      Push (StateStack , State )
      Push (SymbolStack, Symbol)
      Symbol := NextSymbol ( )
    ELSE
      State := Tunnel (State)
      IF State = NoState THEN EXIT END
    END
  END
  WHILE NOT Top (StateStack)  $\in$  FinalStates DO
    State := Pop (StateStack )
    UnRead (Pop (SymbolStack))
  END
  Semantics (Top (StateStack)) ( )
END

```

Before we present the algorithm to compute the control function we need some more definitions:

Definition 3: trace

The *trace* of a string is the sequence of states that a tunnel automaton passes through given the string as input. States at which the automaton does not consume a symbol are omitted. This includes the start state. If necessary we pad the trace with the value NoState (denoted by the character -) to adjust its length to the length of the string.

Examples (computed by using the DFA of Fig. 2 as tunnel automaton):

The trace of	IF	is	1 1
The trace of	ENDIF	is	10 11 12 1 1
The trace of	1789	is	3 3 2 2
The trace of	777Q	is	3 3 3 4
The trace of	::=	is	13 - -

Definition 4: ambiguous state


We call a state s *ambiguous* (or ambiguously reachable) if there exist more than one string such that for each string the repeated execution of the control function (first loop in Algorithm 1) ends in state s .

Example: The states 1, 2, 3, and 4 of the DFA of Fig. 2 are ambiguous states.

Method: Construction of a tunnel automaton from a given set of regular expressions

- Step 1: Convert the NFA for non-constant REs to a DFA using the usual algorithms for subset construction and state minimization [ASU86, WaG84].
- Step 2: Extend the DFA to a tunnel automaton by setting Tunnel (s) to NoState for every state s .
- Step 3: Compute the set of ambiguous states of the tunnel automaton. For convenience Appendix 1 contains an algorithm to compute the ambiguous states.
- Step 4: For every constant RE execute Step 5 which incrementally extends the tunnel automaton. Continue with Step 6.
- Step 5: Compute the trace of the string specified by the constant RE using the current tunnel automaton. We have to distinguish 4 cases:
- Case 1: The trace contains neither NoState nor ambiguous states:

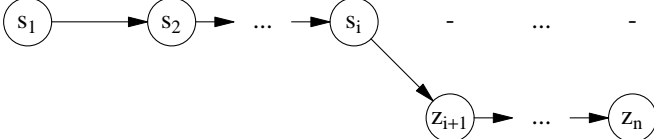
Let the RE be	a_1	a_2	...	a_n
Let the trace be	s_1	s_2	...	s_n



As the path for the RE already exists include (add) the semantics of RE to the semantics of s_n and include s_n into the set of final states.

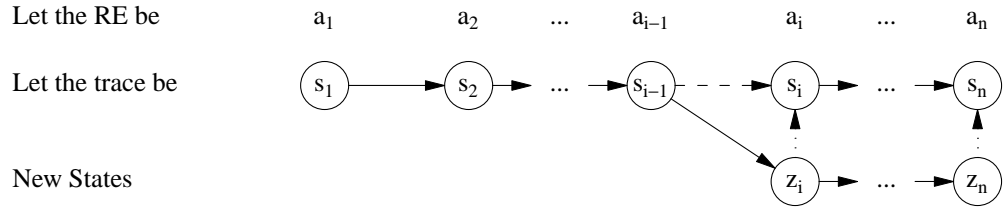
- Case 2: The trace does not contain ambiguous states but it contains NoState:

Let the RE be	a_1	a_2	...	a_i	a_{i+1}	...	a_n
Let the trace be	s_1	s_2	...	s_i	-	...	-
New States					z_{i+1}	...	z_n



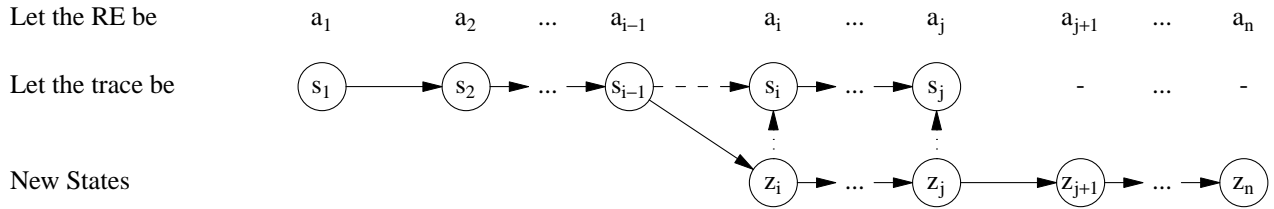
Let s_i be the last state before NoState. Extend the path s_1, \dots, s_i by newly created states z_{i+1}, \dots, z_n . Extend the control function to pass through the states z_{i+1}, \dots, z_n given the input a_{i+1}, \dots, a_n starting from state s_i . Set the semantics of the states z_{i+1}, \dots, z_{n-1} to the empty set. Set the semantics of z_n to the semantics of RE and include z_n into the set of final states. Set the tunnel function of z_{i+1}, \dots, z_n to NoState.

Case 3: The trace does not contain NoState but it contains ambiguous states:



Let s_i be the first ambiguous state in the trace. Create new states z_i, \dots, z_n and extend the control function to pass through the states z_i, \dots, z_n given the input a_i, \dots, a_n starting from state s_{i-1} . Note, that the transition from s_{i-1} to s_i on input a_i is lost this way. Set the semantics of the states z_i, \dots, z_n to the semantics of the corresponding states s_i, \dots, s_n . Include the semantics of RE to the semantics of z_n and include z_n into the set of final states. Set the tunnel function of z_i, \dots, z_n to the corresponding states s_i, \dots, s_n .

Case 4: The trace contains ambiguous states as well as NoState:



Let s_i be the first ambiguous state in the trace. Let s_j be the last state before NoState. Create new states z_i, \dots, z_n and extend the control function to pass through the states z_i, \dots, z_n given the input a_i, \dots, a_n starting from state s_{i-1} . Note again that the transition from s_{i-1} to s_i on input a_i is lost this way. Set the semantics of the states z_i, \dots, z_j to the semantics of the corresponding states s_i, \dots, s_j . Set the semantics of the states z_{j+1}, \dots, z_{n-1} to the empty set. Set the semantics of z_n to the semantics of RE and include z_n into the set of final states. Set the tunnel function of z_i, \dots, z_j to the corresponding states s_i, \dots, s_j and set the tunnel function of the states z_{j+1}, \dots, z_n to NoState.

Step 6: If an unambiguous semantic function is desired convert

$$\text{Semantics}(s) = \{ p_1, \dots, p_n \} \quad \text{to} \quad \text{Semantics}(s) = \{ p_i \}$$

for all states s . We assume the procedures p_1, \dots, p_n to be ordered totally with p_i being the maximal (see below) procedure of p_1, \dots, p_n .

Algorithm 2 describes step 5 more precisely. The problem of an ambiguous semantic function arises already in the running example, as e.g. the string END matches both the REs for IdentSymbol and EndSymbol. Therefore state 12 of Fig. 2 would be associated with two semantic procedures. The generators LEX and Rex for example turn the semantic function into an unambiguous one by considering the sequence of the REs in the given specification. The REs and their associated semantic actions are mapped to priorities in descending order. In case of conflicts the semantic action with greatest priority is preferred. In other words the procedures are ordered totally and the "maximal procedure" out of several ones is selected.

A tunnel automaton extended using Algorithm 2 works correctly because of the following reasons. The constant REs are recognized correctly because of the construction used. We construct a decision tree without any ambiguous states.

The non-constant REs are recognized correctly because: If the tunnel automaton stops at a newly created state we have propagated the semantics of the original final state to the new one. Otherwise the tunnel automaton uses at most one tunnel transition and stops at exactly the same state as it would have stopped at before. That is because of the construction of the tunnel function. As we did not change the semantic function of previously existing states, except in the case where we could use the state as final state of a constant RE, the automaton behaves as before.

Algorithm 2: extend a tunnel automaton to recognize an additional constant RE

```

BEGIN
  State := StartState
  Symbol := NextSymbol ()  /* reading the string specified by the constant RE */

  /* trace and do nothing */

  LOOP
    IF Control (State, Symbol) = NoState OR
       Symbol = EndOfInput OR
       Control (State, Symbol) ∈ AmbiguousStates THEN EXIT END
    State := Control (State, Symbol)  /* trace */
    Symbol := NextSymbol ()
  END
  PreviousState := State
  /* trace and duplicate the path */

  LOOP
    IF Control (State, Symbol) ≠ NoState THEN
      IF Symbol = EndOfInput THEN EXIT END
      NewState := CreateState ()
      State := Control (State, Symbol)  /* trace */
      Control (PreviousState, Symbol) := NewState
      Semantics (NewState) := Semantics (State)
      Tunnel (NewState) := State
      PreviousState := NewState
      Symbol := NextSymbol ()
    ELSE
      IF Tunnel (State) = NoState THEN EXIT END
      State := Tunnel (State)
    END
  END
  /* extend the path */

  WHILE Symbol ≠ EndOfInput DO
    NewState := CreateState ()
    Control (PreviousState, Symbol) := NewState
    Semantics (NewState) := ∅
    Tunnel (NewState) := NoState
    PreviousState := NewState
    Symbol := NextSymbol ()
  END
  /* process new final state */
  FinalStates := FinalStates ∪ { PreviousState }
  Semantics (PreviousState) := Semantics (PreviousState) ∪ Semantics (RE)
END

```

We call a tunnel automaton *minimal* if it has the minimal number of states to do its job, e. g. it must be able to distinguish between different tokens using separate final states. Without a formal proof we state that Algorithm 2 constructs a minimal automaton if the given DFA was minimal. The reason is that a tunnel automaton has the same number of states and the same "structure" as a corresponding DFA except that many regular transitions are replaced by a few tunnel transitions. Compare for example the Figures 2 and 3.

EXAMPLE

Algorithm 2 constructs for the running example the tunnel automaton depicted in Fig. 3. The dotted arrows denote tunnel transitions. The automaton in Fig. 3 is graphically similar to the one in Fig. 2: most of the transitions leading to state 1 have been replaced by tunnel transitions. However, note the difference: whereas a solid arrow of Fig. 2 stands for a big set of transitions a dotted arrow of Fig. 3 denotes a single tunnel transition, only. This is the key to the efficient representation of the control and tunnel functions. As the control function corresponds to a sparse matrix it can advantageously be implemented using the comb-vector technique [ASU86]. The rows of a matrix are merged into a single vector called Next. Two additional vectors called Base and Check are necessary to accomplish the access of the original data. The resulting data structure resembles the merging of several combs into one. In combination with the above a fourth array called Tunnel is used to compress the table even more. This array corresponds directly to our tunnel function. Fig. 4 shows some entries of the comb-vector for the running example. The excerpt is restricted to the characters

E, N, D.

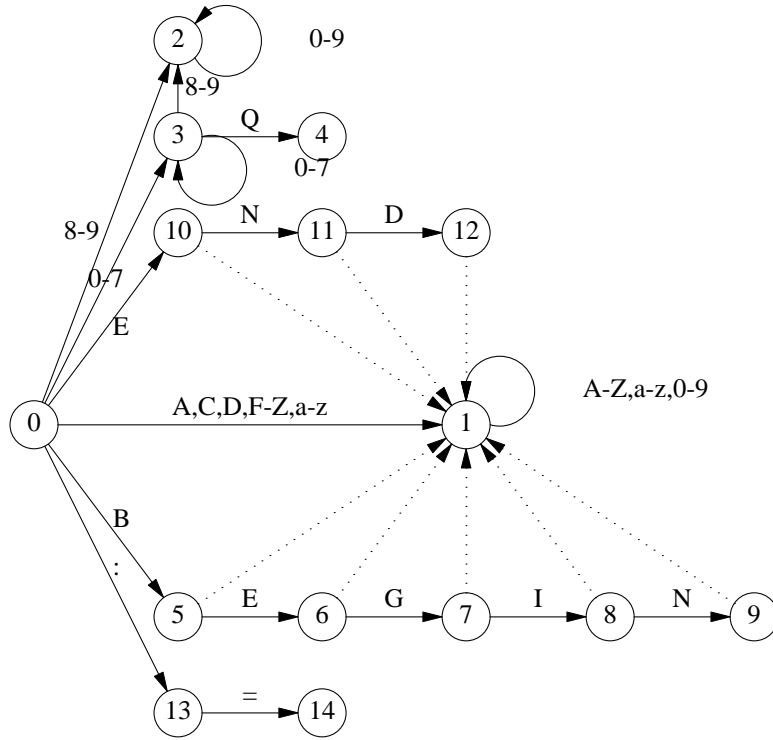


Fig. 3: tunnel automaton for the running example

An advantage of Algorithm 2 is that this data structure is computed in part directly from the set of REs. There is no need to compute a complete classical DFA first and to transform it into the above data structure using time and space consuming table compression algorithms afterwards. The construction of the comb-vector consists primarily of the computation of the Base and the Tunnel values. For each state the Base value is determined by a simple search algorithm which shows reasonable results and performance. A clever computation of the Tunnel values is essential for a good table compression. Its complexity is $O(n^2c)$ where n is the number of states of the DFA and c is the cardinality of the character set. For each state an other state has to be determined which can safely act as "Tunnel state" and which

	0	1	2	...	68	69	70	71	72	...	78	79	80	...
Check	-	-	-	...	0	0	1	1	11	...	0	10	1	...
Next	-	-	-	...	1	10	1	1	12	...	1	11	1	...

State	0	1	...	10	11	12	...
Base	0	2	...	1	4	0	...
Tunnel	-	-	...	1	1	1	...

```

Control (State, Symbol) := IF   Check [Base [State] + Symbol] = State
                           THEN Next [Base [State] + Symbol]
                           ELSE NoState

```

Fig. 4: comb vector data structure for the running example and access procedure
(for characters E, N, D only; ASCII codes: E=69, D=68, N=78)

saves a maximal number of transitions. It is sufficient to perform this expensive algorithm for the few states created for non-constant REs only. The Tunnel values for the states created for constant REs are determined by Algorithm 2 in linear time.

SEVERAL START STATES

Scanner generators like LEX and Rex offer the feature of so-called start states to handle left context. In each start state a different set of REs may be activated and there are special statements to change the current start state. A scanner specification using several start states is turned into an automaton with several start states. Under these circumstances the construction algorithm for a tunnel automaton becomes a little bit more complex. We only mention the problems that arise from this extension.

First we have to refine the computation of ambiguous states. Definition 4 is still valid saying that an ambiguous state must be reachable with more than one string. Now we have to take into account that strings can be processed starting from several start states. The difference of the refined algorithm lies in the treatment of the states being direct successors of the start states. Such a state becomes ambiguous if there exist more than one transition from one start state. It is not sufficient if there are several transitions but each one originates in a different start state.

If a constant RE should be recognized in a set of start states S we would like to construct only one sequence of states which can be reached from all members of S . This is because we want to create as few states as possible. However, we have to be careful, because the trace of the RE could lead to a state which can be reached from a start state t not contained in S . This would erroneously allow the recognition of the RE also from start state t . Therefore we have to construct several sequences of states or we have to branch off an existing sequence in this case. To be able to do this we have to know the set of start states a state can be reached from. Step 5 of the construction of a tunnel automaton has to be extended not only to consider ambiguous states but also to check whether the set of start states S of the RE is a proper subset of the set of start states a state in the trace can be reached from. In this case a side branch with newly created states is necessary.

We extend a tunnel automaton to recognize a RE once for each corresponding start state. Our aim is to use the same sequence of states constructed for the first start state, but we have to check whether this is safely possible. To be able to reuse a sequence of states we have to record it. For every character (index) and every associated state t from the traces of the RE we record the state actually used in the tunnel automaton as target state of the current transition. This is either the state t or a newly created one. The construction method is now extended in the following way. For each character index i of the RE the state t from the trace is examined. Whenever possible the state recorded for i and t is reused.

EXPERIMENTAL RESULTS

The presented method has been used in a scanner generator called *Rex* (Regular EXpression tool) which is implemented by a 5,500 line Modula-2 [Wir85] program. It is able to generate scanners in the languages C and Modula-2. The generated scanners are table-driven implementations of a tunnel automaton. In the following we compare the time to generate a scanner as well as the speed of the generated scanners for LEX [Les75], flex [Pax88], and Rex [Groc, Groc]. Flex is a rewrite of LEX intended to generate lexical analysers much faster and the analysers use smaller tables and run faster.

To compare the scanner generators we used 4 versions of a Modula-2 scanner specification (see Appendix 2 for an example written in the input language of Rex). The versions differ in the number of constant REs as shown in Table 1. Version 1 is incomplete, version 2 omits keywords, version 3 has keywords, and version 4 has lower as well as upper case keywords. The times are given in seconds measured on a MC 68020 processor. The optimization of flex can be controlled by options: usually `-cem` results in the smallest tables and `-cf` in the fastest analyser.

LEX performs well for small specifications but it seems to use a quadratic or exponential algorithm for all the REs. This leads to long generation times and large tables for large specifications (containing e. g. many keywords).

Flex is quite an improvement compared to LEX especially in terms of generation time. The sizes of the tables and the generated scanners depend on the optimization flags: `-cem` reduces the sizes drastically but `-cf` yields sizes even larger than LEX.

The timings of Rex demonstrate its linear behaviour. In general the generation times are not quite as small as those of flex except for specification 4. The expensive algorithms for subset construction, state minimization, and table compression are executed for 40 states, only. An arbitrary number of constant REs like keywords can be added in a small, linear growing time. Although the generated tables are larger than those of flex the total sizes of the scanners

Table 1: Comparison of Scanner Generators

		Spec. 1	Spec. 2	Spec. 3	Spec. 4
# of non-constant REs		10	10	10	10
# of constant REs		2	29	69	109
total # of REs		12	39	79	119
# of states for non-constant REs		40	40	40	40
# of states for constant REs		0	26	181	336
total # of states (generated by Rex)		40	66	221	376
generation time using [seconds]	LEX	3.14	6.71	73.71	215.08
	flex -cem	0.69	0.78	2.01	3.81
	flex -cf	1.39	2.35	7.16	12.16
	Rex	3.56	3.76	4.91	6.16
table size using [bytes]	LEX	2,996	4,708	39,156	67,384
	flex -cem	1,116	1,376	2,868	4,592
	flex -cf	10,744	17,424	57,260	97,096
	Rex	3,114	3,218	4,366	5,530
scanner size using [bytes]	LEX	5,464	8,044	43,756	73,264
	flex -cem	14,240	15,368	18,140	21,144
	flex -cf	15,452	23,000	64,116	105,232
	Rex	8,456	8,884	11,200	13,536

(including the tables) are smaller. Compared to LEX the correct scanner specification 3 is processed nearly 20 times faster by Rex, the size of the table is 10 times smaller, and the scanner is 4 times smaller.

To compare the generated scanners (Table 2) we used a big Modula-2 program as input whose characteristics are as follows: # of lines: 4,171, # of characters: 100,268, # of tokens: 16,948. The timings include input from file, scanning and hashing of identifiers. The Rex options -m and -c determine the target languages Modula-2 and C. Compared to LEX flex yields a speed-up of 1.8 with options -cem and a speed-up between 3.4 and 3.8 with options -cf. The C version of Rex reaches a speed-up between 4 and 5. This speed is reached with analysers that are considerable smaller than flex generated ones. The figures show that a tunnel automaton can be implemented efficiently: Scanners generated by Rex are 4 to 5 times faster than scanners generated by LEX. More details can be found in reference[Grob].

Table 3 compares scanners for different languages generated by Rex. The sizes of the tables and the complete scanners all lie in the same range which is relatively small. Only the generation times for Pascal and Ada are exceptionally long. The reason is that these languages allow keywords to be written with any letters no matter if upper or lower case. Therefore keywords are no longer constant REs and can not be processed with the presented linear algorithm.

Table 2: Comparison of Generated Scanners

generator	with hashing of identifiers and number conversion				without hashing and number conversion	
	table size [bytes]	scanner size [bytes]	time [sec.]	speed [lines/min.]	time [sec.]	speed [lines/min.]
LEX	39,156	43,756	7.21	34,700	6.88	36,400
flex -cem	2,868	18,140	3.99	62,700	3.69	67,800
flex -cf	57,260	64,116	2.12	118,000	1.80	139,000
Rex -m	4,306	13,672	3.00	83,400	2.22	112,700
Rex -c	4,366	11,200	1.77	141,400	1.37	182,700

Table 3: Comparison of Rex-Generated Scanners

language	generator data				scanner data	
	static size [bytes]	dyn. memory [bytes]	total memory [bytes]	time [sec.]	table size [bytes]	scanner size [bytes]
Pascal	102,970	238,464	341,434	87.40	4,426	13,128
Modula	102,970	201,344	304,314	4.93	4,306	13,692
Oberon	102,970	201,344	304,314	5.71	5,122	14,284
C	102,970	201,344	304,314	7.24	5,702	13,296
Ada	102,970	441,984	544,954	300.63	6,222	17,450

CONCLUSION

The presented method allows the conversion of a NFA given by a set of REs to a DFA in practically linear time. This holds under the assumption that only a few non-constant but many constant REs have to be processed which is true in many practical cases. Compared to LEX we gained a speed-up of up to 20. Compared to flex which similarly improves the generation times the generated scanners are smaller and faster. Not only is the generation time reduced drastically but also the space requirement during generation of the scanner. The generator has to perform the subset construction, state minimization, and table compression only for a few states. Therefore the space needed by those algorithms is small. The presented method directly constructs a space efficient representation of the scanner control table which is used in combination with the comb-vector technique [ASU86]. The method allows the generation of lexical analysers in a small amount of time. The generated scanners are powerful and efficient enough to be used in production compilers. Finally, scanner generators could be applied to a broader area than just compiler construction.

REFERENCES

- [AhC75] A. V. Aho and M. J. Corasick, Efficient String Matching: An Aid to Bibliographic Search, *Comm. ACM* 18, 6 (June 1975), 333-340.
- [ASU86] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison Wesley, Reading, MA, 1986.
- [Groa] J. Grosch, Rex - A Scanner Generator, Cocktail Document No. 5, CoCoLab Germany.
- [Grob] J. Grosch, An Efficient Table-Driven Scanner, Cocktail Document No. 1, CoCoLab Germany.
- [Groc] J. Grosch, Selected Examples of Scanner Specifications, Cocktail Document No. 7, CoCoLab Germany.
- [Heu86] V. P. Heuring, The Automatic Generation of Fast Lexical Analysers, *Software—Practice & Experience* 16, 9 (Sep. 1986), 801-808.
- [HoL87] R. N. Horspool and M. R. Levy, Mkscan - An Interactive Scanner Generator, *Software—Practice & Experience* 17, 6 (June 1987), 369-378.
- [Les75] M. E. Lesk, LEX — A Lexical Analyzer Generator, Computing Science Technical Report 39, Bell Telephone Laboratories, Murray Hill, NJ, 1975.
- [Mös86] H. Mössenböck, Alex - A Simple and Efficient Scanner Generator, *SIGPLAN Notices* 21, 12 (Dec. 1986), 139-148.
- [Pax88] V. Paxson, *Flex - Manual Pages*, Public Domain Software, 1988.
- [WaG84] W. M. Waite and G. Goos, *Compiler Construction*, Springer Verlag, New York, NY, 1984.
- [Wai86] W. M. Waite, The Cost of Lexical Analysis, *Software—Practice & Experience* 16, 5 (May 1986), 473-488.
- [Wir85] N. Wirth, *Programming in Modula-2*, Springer Verlag, Heidelberg, 1985. Third Edition.

APPENDIX 1

Algorithm 4: ambiguous states of a tunnel automaton (with one start state)

```

BEGIN
  /* transition function using the tunnel function */
  PROCEDURE NextState (State, Symbol)
    BEGIN
      LOOP
        IF Control (State, Symbol)  $\neq$  NoState THEN
          RETURN Control (State, Symbol)
        ELSE
          State := Tunnel (State)
          IF State = NoState THEN RETURN NoState END
        END
      END
    END
  END

  /* compute the number of predecessors */
  FORALL State  $\in$  StateSet DO
    PredCount (State) := 0
  END

  FORALL State  $\in$  StateSet DO
    FORALL Symbol  $\in$  Vocabulary DO
      PredCount (NextState (State, Symbol)) += 1
    END
  END

  /* iteratively remove states with 1 predecessor */
  AmbiguousStates := StateSet - { NoState }
  UnambiguousStates := { StartState }

  WHILE UnambiguousStates  $\neq \emptyset$  DO
    State := SELECT UnambiguousStates
    UnambiguousStates -= { State }
    AmbiguousStates += { State }
    FORALL Symbol  $\in$  Vocabulary DO
      Successor := NextState (State, Symbol)
      IF PredCount (Successor) = 1 THEN
        UnambiguousStates  $\cup$ := { Successor }
      END
    END
  END
END

```

APPENDIX 2

An example of a scanner specification for Modula-2 in Rex syntax:

```

GLOBAL { VAR NestingLevel: CARDINAL; }
BEGIN { NestingLevel := 0; }
DEFAULT { Error ("illegal character:"); yyEcho; }
EOF { IF yyStartState = comment THEN Error ("unclosed comment"); END; }
DEFINE
    digit      = {0-9}      .
    letter     = {a-z A-Z}   .
    cmt        = - {*(\t\n)  .
START comment
RULES
    "(" *      : {INC (NestingLevel); yyStart (comment);}
#comment# "*" : {DEC (NestingLevel); IF NestingLevel = 0 THEN yyStart (STD); END;}
#comment# "(" | "*" | cmt + : {}
#STD# digit + ,
#STD# digit + / ".." : {RETURN 1;}
#STD# {0-7} + B : {RETURN 2;}
#STD# {0-7} + C : {RETURN 3;}
#STD# digit {0-9 A-F} * H : {RETURN 4;}
#STD# digit + "." digit * (E {+\-} ? digit +) ? : {RETURN 5;}
#STD# ' - {\n'} * ' |
      \" - {\n"} * \" : {RETURN 6;}
#STD# ":" : {RETURN 7;}
#STD# "=" : {RETURN 8;}
#STD# "!=" : {RETURN 9;}
...
#STD# AND : {RETURN 34;}
#STD# ARRAY : {RETURN 35;}
#STD# BEGIN : {RETURN 36;}
...
#STD# letter (letter | digit) * : {RETURN 74;}

```

Contents

1.	Introduction	1
2.	The Method	3
3.	Example	6
4.	Several Start States	8
5.	Experimental Results	8
6.	Conclusion	10
	References	10
	Appendix 1	11
	Appendix 2	12