3 6 Finite Automata

We shall now discover how `Lex` turns its input program into a lexical analyzer At the heart of the transition is the formalism known as *nite automata* These are essentially graphs like transition diagrams with a few di erences

1 Finite automata are *recognizers* they simply say yes or no about each possible input string

2 Finite automata come in two avors

  a *Nondeterministic nite automata* NFA have no restrictions on the labels of their edges A symbol can label several edges out of the same state and the empty string is a possible label

  b *Deterministic nite automata* DFA have for each state and for each symbol of its input alphabet exactly one edge with that symbol leaving that state

Both deterministic and nondeterministic nite automata are capable of rec ognizing the same languages In fact these languages are exactly the same languages called the *regular languages* that regular expressions can describe [4]

3 6 1 Nondeterministic Finite Automata

A *nondeterministic nite automaton* NFA consists of

1 A nite set of states S

2 A set of input symbols the *input alphabet* We assume that which stands for the empty string is never a member of

3 A *transition function* that gives for each state and for each symbol in $\{\ \}$ a set of *next states*

4 A state s_0 from S that is distinguished as the *start state* or *initial state*

5 A set of states F a subset of S that is distinguished as the *accepting states* or *nal states*

We can represent either an NFA or DFA by a *transition graph* where the nodes are states and the labeled edges represent the transition function There is an edge labeled a from state s to state t if and only if t is one of the next states for state s and input a This graph is very much like a transition diagram except

[4]There is a small lacuna as we de ned them regular expressions cannot describe the empty language since we would never want to use this pattern in practice However nite automata *can* de ne the empty language In the theory is treated as an additional regular expression for the sole purpose of de ning the empty language

a The same symbol can label edges from one state to several di erent states and

b An edge may be labeled by the empty string instead of or in addition to symbols from the input alphabet

Example 3 14 The transition graph for an NFA recognizing the language of regular expression **a|b abb** is shown in Fig 3 24 This abstract example describing all strings of *a* s and *b* s ending in the particular string *abb* will be used throughout this section It is similar to regular expressions that describe languages of real interest however For instance an expression describing all les whose name ends in o is **any** o where **any** stands for any printable character

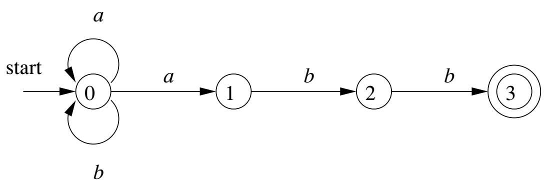


Figure 3 24 A nondeterministic nite automaton

Following our convention for transition diagrams the double circle around state 3 indicates that this state is accepting Notice that the only ways to get from the start state 0 to the accepting state is to follow some path that stays in state 0 for a while then goes to states 1 2 and 3 by reading *abb* from the input Thus the only strings getting to the accepting state are those that end in *abb* □

3 6 2 Transition Tables

We can also represent an NFA by a *transition table* whose rows correspond to states and whose columns correspond to the input symbols and The entry for a given state and input is the value of the transition function applied to those arguments If the transition function has no information about that state input pair we put in the table for the pair

Example 3 15 The transition table for the NFA of Fig 3 24 is shown in Fig 3 25 □

The transition table has the advantage that we can easily nd the transitions on a given state and input Its disadvantage is that it takes a lot of space when the input alphabet is large yet most states do not have any moves on most of the input symbols

STATE	a	b
0	$\{0\ 1\}$	$\{0\}$
1		$\{2\}$
2		$\{3\}$
3		

Figure 3 25 Transition table for the NFA of Fig 3 24

3 6 3 Acceptance of Input Strings by Automata

An NFA *accepts* input string x if and only if there is some path in the transition graph from the start state to one of the accepting states such that the symbols along the path spell out x Note that labels along the path are e ectively ignored since the empty string does not contribute to the string constructed along the path

Example 3 16 The string *aabb* is accepted by the NFA of Fig 3 24 The path labeled by *aabb* from state 0 to state 3 demonstrating this fact is

$$0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$$

Note that several paths labeled by the same string may lead to di erent states For instance path

$$0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$$

is another path from state 0 labeled by the string *aabb* This path leads to state 0 which is not accepting However remember that an NFA accepts a string as long as *some* path labeled by that string leads from the start state to an accepting state The existence of other paths leading to a nonaccepting state is irrelevant □

The *language de ned* or *accepted* by an NFA is the set of strings labeling some path from the start to an accepting state As was mentioned the NFA of Fig 3 24 de nes the same language as does the regular expression **a**|**b** **abb** that is all strings from the alphabet $\{a\ b\}$ that end in *abb* We may use $L\ A$ to stand for the language accepted by automaton A

Example 3 17 Figure 3 26 is an NFA accepting L **aa** |**bb** String *aaa* is accepted because of the path

$$0 \xrightarrow{\varepsilon} 1 \xrightarrow{a} 2 \xrightarrow{a} 2 \xrightarrow{a} 2$$

Note that s disappear in a concatenation so the label of the path is *aaa* □

3 6 4 Deterministic Finite Automata

A *deterministic nite automaton* DFA is a special case of an NFA where

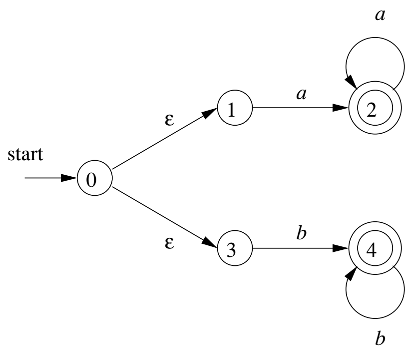


Figure 3 26 NFA accepting **aa** |**bb**

1 There are no moves on input and

2 For each state s and input symbol a there is exactly one edge out of s labeled a

If we are using a transition table to represent a DFA then each entry is a single state We may therefore represent this state without the curly braces that we use to form sets

While the NFA is an abstract representation of an algorithm to recognize the strings of a certain language the DFA is a simple concrete algorithm for recognizing strings It is fortunate indeed that every regular expression and every NFA can be converted to a DFA accepting the same language because it is the DFA that we really implement or simulate when building lexical analyzers The following algorithm shows how to apply a DFA to a string

Algorithm 3 18 Simulating a DFA

INPUT An input string x terminated by an end of le character **eof** A DFA D with start state s_0 accepting states F and transition function *move*

OUTPUT Answer yes if D accepts x no otherwise

METHOD Apply the algorithm in Fig 3 27 to the input string x The function *move* s c gives the state to which there is an edge from state s on input c The function *nextChar* returns the next character of the input string x □

Example 3 19 In Fig 3 28 we see the transition graph of a DFA accepting the language **a**|**b** **abb** the same as that accepted by the NFA of Fig 3 24 Given the input string *ababb* this DFA enters the sequence of states 0 1 2 1 2 3 and returns yes □

$s \quad s_0$
$c \quad nextChar$
while c **eof** {
 $s \quad move\ s\ c$
 $c \quad nextChar$
}
if s is in F **return** `yes`
else return `no`

Figure 3 27 Simulating a DFA

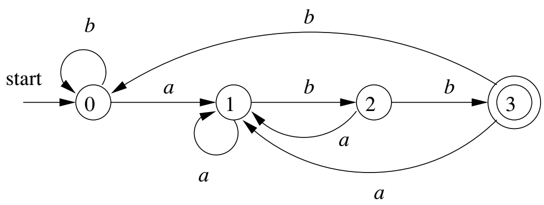


Figure 3 28 DFA accepting **a|b abb**

3 6 5 Exercises for Section 3 6

Exercise 3 6 1 Figure 3 19 in the exercises of Section 3 4 computes the failure function for the KMP algorithm Show how given that failure function we can construct from a keyword $b_1 b_2 \quad b_n$ an $n \quad 1$ state DFA that recognizes $b_1 b_2 \quad b_n$ where the dot stands for any character Moreover this DFA can be constructed in $O\ n$ time

Exercise 3 6 2 Design nite automata deterministic or nondeterministic for each of the languages of Exercise 3 3 5

Exercise 3 6 3 For the NFA of Fig 3 29 indicate all the paths labeled *aabb* Does the NFA accept *aabb*

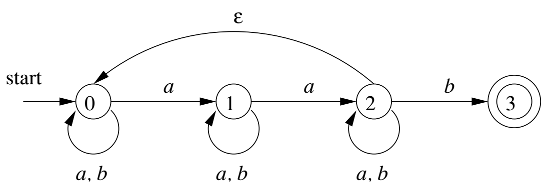


Figure 3 29 NFA for Exercise 3 6 3

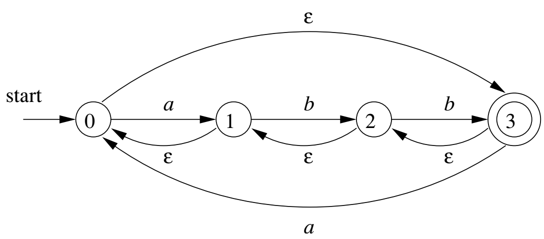


Figure 3 30 NFA for Exercise 3 6 4

Exercise 3 6 4 Repeat Exercise 3 6 3 for the NFA of Fig 3 30

Exercise 3 6 5 Give the transition tables for the NFA of

a Exercise 3 6 3

b Exercise 3 6 4

c Figure 3 26

3 7 From Regular Expressions to Automata

The regular expression is the notation of choice for describing lexical analyzers and other pattern processing software as was re ected in Section 3 5 How ever implementation of that software requires the simulation of a DFA as in Algorithm 3 18 or perhaps simulation of an NFA Because an NFA often has a choice of move on an input symbol as Fig 3 24 does on input a from state 0 or on as Fig 3 26 does from state 0 or even a choice of making a transition on or on a real input symbol its simulation is less straightforward than for a DFA Thus often it is important to convert an NFA to a DFA that accepts the same language

In this section we shall rst show how to convert NFA s to DFA s Then we use this technique known as the subset construction to give a useful algo rithm for simulating NFA s directly in situations other than lexical analysis where the NFA to DFA conversion takes more time than the direct simulation Next we show how to convert regular expressions to NFA s from which a DFA can be constructed if desired We conclude with a discussion of the time space tradeo s inherent in the various methods for implementing regular expressions and see how to choose the appropriate method for your application

3 7 1 Conversion of an NFA to a DFA

The general idea behind the subset construction is that each state of the constructed DFA corresponds to a set of NFA states After reading input

a_1a_2 a_n the DFA is in that state which corresponds to the set of states that the NFA can reach from its start state following paths labeled a_1a_2 a_n

It is possible that the number of DFA states is exponential in the number of NFA states which could lead to di culties when we try to implement this DFA However part of the power of the automaton based approach to lexical analysis is that for real languages the NFA and DFA have approximately the same number of states and the exponential behavior is not seen

Algorithm 3 20 The *subset construction* of a DFA from an NFA

INPUT An NFA N

OUTPUT A DFA D accepting the same language as N

METHOD Our algorithm constructs a transition table *Dtran* for D Each state of D is a set of NFA states and we construct *Dtran* so D will simulate in parallel all possible moves N can make on a given input string Our rst problem is to deal with transitions of N properly In Fig 3 31 we see the de nitions of several functions that describe basic computations on the states of N that are needed in the algorithm Note that s is a single state of N while T is a set of states of N

OPERATION	DESCRIPTION
closure s	Set of NFA states reachable from NFA state s on transitions alone
closure T	Set of NFA states reachable from some NFA state s in set T on transitions alone $_{s \text{ in } T}$ *closure* s
move T a	Set of NFA states to which there is a transition on input symbol a from some state s in T

Figure 3 31 Operations on NFA states

We must explore those sets of states that N can be in after seeing some input string As a basis before reading the rst input symbol N can be in any of the states of *closure* s_0 where s_0 is its start state For the induction suppose that N can be in set of states T after reading input string x If it next reads input a then N can immediately go to any of the states in *move* T a However after reading a it may also make several transitions thus N could be in any state of *closure* *move* T a after reading input xa Following these ideas the construction of the set of D s states *Dstates* and its transition function *Dtran* is shown in Fig 3 32

The start state of D is *closure* s_0 and the accepting states of D are all those sets of N s states that include at least one accepting state of N To complete our description of the subset construction we need only to show how

```
initially   closure s0  is the only state in Dstates  and it is unmarked
while   there is an unmarked state T in Dstates   {
        mark T
        for   each input symbol a   {
                U       closure  move T  a
                if   U is not in Dstates
                        add U as an unmarked state to Dstates
                Dtran T  a      U
        }
}
```

Figure 3 32 The subset construction

closure T is computed for any set of NFA states T This process shown in Fig 3 33 is a straightforward search in a graph from a set of states In this case imagine that only the labeled edges are available in the graph □

```
push all states of T onto stack
initialize   closure T   to T
while   stack is not empty   {
        pop t  the top element  o   stack
        for   each state u with an edge from t to u labeled
                if   u is not in   closure T     {
                        add u to   closure T
                        push u onto stack
                }
}
```

Figure 3 33 Computing *closure* T

Example 3 21 Figure 3 34 shows another NFA accepting **a|b abb** it hap pens to be the one we shall construct directly from this regular expression in Section 3 7 Let us apply Algorithm 3 20 to Fig 3 34

The start state A of the equivalent DFA is *closure* 0 or A $\{0\ 1\ 2\ 4\ 7\}$ since these are exactly the states reachable from state 0 via a path all of whose edges have label Note that a path can have zero edges so state 0 is reachable from itself by an labeled path

The input alphabet is $\{a\ b\}$ Thus our rst step is to mark A and compute *Dtran* A a *closure move* A a and *Dtran* A b *closure move* A b Among the states 0 1 2 4 and 7 only 2 and 7 have transitions on a to 3 and 8 respectively Thus *move* A a $\{3\ 8\}$ Also *closure* $\{3\ 8\}$ $\{1\ 2\ 3\ 4\ 6\ 7\ 8\}$ so we conclude

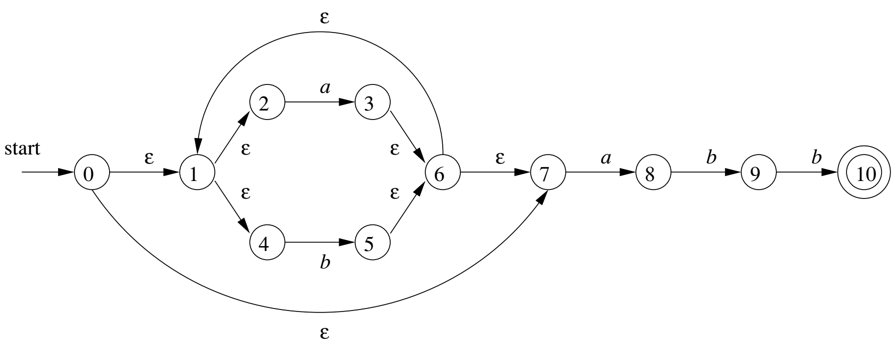


Figure 3 34 NFA N for **a|b abb**

$$Dtran\ A\ a \qquad closure\ \ move\ A\ a \qquad closure\ \{3\ 8\} \qquad \{1\ 2\ 3\ 4\ 6\ 7\ 8\}$$

Let us call this set B so $Dtran\ A\ a \quad B$

Now we must compute $Dtran\ A\ b$ Among the states in A only 4 has a transition on b and it goes to 5 Thus

$$Dtran\ A\ b \qquad closure\ \{5\} \qquad \{1\ 2\ 4\ 5\ 6\ 7\}$$

Let us call the above set C so $Dtran\ A\ b \quad C$

NFA STATE	DFA STATE	a	b
$\{0\ 1\ 2\ 4\ 7\}$	A	B	C
$\{1\ 2\ 3\ 4\ 6\ 7\ 8\}$	B	B	D
$\{1\ 2\ 4\ 5\ 6\ 7\}$	C	B	C
$\{1\ 2\ 4\ 5\ 6\ 7\ 9\}$	D	B	E
$\{1\ 2\ 4\ 5\ 6\ 7\ 10\}$	E	B	C

Figure 3 35 Transition table $Dtran$ for DFA D

If we continue this process with the unmarked sets B and C we eventually reach a point where all the states of the DFA are marked This conclusion is guaranteed since there are only 2^{11} di erent subsets of a set of eleven NFA states The ve di erent DFA states we actually construct their correspond ing sets of NFA states and the transition table for the DFA D are shown in Fig 3 35 and the transition graph for D is in Fig 3 36 State A is the start state and state E which contains state 10 of the NFA is the only accepting state

Note that D has one more state than the DFA of Fig 3 28 for the same lan guage States A and C have the same move function and so can be merged We discuss the matter of minimizing the number of states of a DFA in Section 3 9 6 □

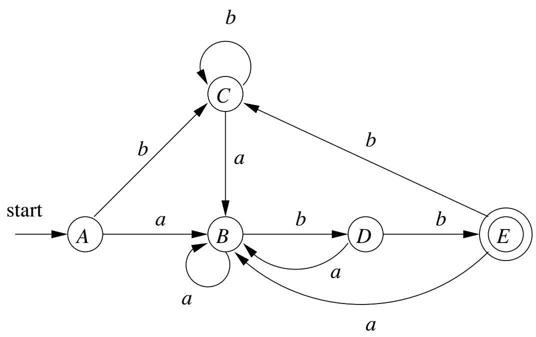


Figure 3 36 Result of applying the subset construction to Fig 3 34

3 7 2 Simulation of an NFA

A strategy that has been used in a number of text editing programs is to con struct an NFA from a regular expression and then simulate the NFA using something like an on the y subset construction The simulation is outlined below

Algorithm 3 22 Simulating an NFA

INPUT An input string x terminated by an end of le character **eof** An NFA N with start state s_0 accepting states F and transition function *move*

OUTPUT Answer yes if N accepts x no otherwise

METHOD The algorithm keeps a set of current states S those that are reached from s_0 following a path labeled by the inputs read so far If c is the next input character read by the function *nextChar* then we rst compute *move* S c and then close that set using *closure* The algorithm is sketched in Fig 3 37 □

S *closure* s_0
c *nextChar*
while c **eof** {
  S *closure move* S c
  c *nextChar*
}
if S F **return** `yes`
else return `no`

Figure 3 37 Simulating an NFA

3 7 3 E ciency of NFA Simulation

If carefully implemented Algorithm 3 22 can be quite e cient As the ideas involved are useful in a number of similar algorithms involving search of graphs we shall look at this implementation in additional detail The data structures we need are

1 Two stacks each of which holds a set of NFA states One of these stacks *oldStates* holds the current set of states i e the value of S on the right side of line 4 in Fig 3 37 The second *newStates* holds the next set of states S on the left side of line 4 Unseen is a step where as we go around the loop of lines 3 through 6 *newStates* is transferred to *oldStates*

2 A boolean array *alreadyOn* indexed by the NFA states to indicate which states are in *newStates* While the array and stack hold the same infor mation it is much faster to interrogate *alreadyOn* s than to search for state s on the stack *newStates* It is for this e ciency that we maintain both representations

3 A two dimensional array *move* s a holding the transition table of the NFA The entries in this table which are sets of states are represented by linked lists

To implement line 1 of Fig 3 37 we need to set each entry in array *al readyOn* to `FALSE` then for each state s in *closure* s_0 push s onto *oldStates* and set *alreadyOn* s to `TRUE` This operation on state s and the implementation of line 4 as well are facilitated by a function we shall call *addState* s This function pushes state s onto *newStates* sets *alreadyOn* s to `TRUE` and calls itself recursively on the states in *move* s in order to further the computation of *closure* s However to avoid duplicating work we must be careful never to call *addState* on a state that is already on the stack *newStates* Figure 3 38 sketches this function

```
addState s  {
        push s onto newStates
        alreadyOn s      TRUE
        for   t on move s
                if    alreadyOn t
                        addState t
}
```

Figure 3 38 Adding a new state s which is known not to be on *newStates*

We implement line 4 of Fig 3 37 by looking at each state s on *oldStates* We rst nd the set of states *move* s c where c is the next input and for each of those states that is not already on *newStates* we apply *addState* to it Note that *addState* has the e ect of computing the *closure* and adding all those states to *newStates* as well if they were not already on This sequence of steps is summarized in Fig 3 39

```
16   for   s on oldStates   {
17        for   t on move s  c
18             if   alreadyOn t
19                       addState t
20        pop s from oldStates
21   }

22   for   s on newStates   {
23        pop s from newStates
24        push s onto oldStates
25        alreadyOn s     FALSE
26   }
```

Figure 3 39 Implementation of step 4 of Fig 3 37

Now suppose that the NFA N has n states and m transitions i e m is the sum over all states of the number of symbols or on which the state has a transition out Not counting the call to *addState* at line 19 of Fig 3 39 the time spent in the loop of lines 16 through 21 is O n That is we can go around the loop at most n times and each step of the loop requires constant work except for the time spent in *addState* The same is true of the loop of lines 22 through 26

During one execution of Fig 3 39 i e of step 4 of Fig 3 37 it is only possible to call *addState* on a given state once The reason is that whenever we call *addState* s we set *alreadyOn* s to `TRUE` at line 11 of Fig 3 38 Once *alreadyOn* s is `TRUE` the tests at line 13 of Fig 3 38 and line 18 of Fig 3 39 prevent another call

The time spent in one call to *addState* exclusive of the time spent in recur sive calls at line 14 is O 1 for lines 10 and 11 For lines 12 and 13 the time depends on how many transitions there are out of state s We do not know this number for a given state but we know that there are at most m transitions in total out of all states As a result the aggregate time spent in lines 12 and 13 over all calls to *addState* during one execution of the code of Fig 3 39 is O m The aggregate for the rest of the steps of *addState* is O n since it is a constant per call and there are at most n calls

We conclude that implemented properly the time to execute line 4 of Fig 3 37 is O n m The rest of the while loop of lines 3 through 6 takes O 1 time per iteration If the input x is of length k then the total work in that loop is O k n m Line 1 of Fig 3 37 can be executed in O n m time since it is essentially the steps of Fig 3 39 with *oldStates* containing only

Big Oh Notation

An expression like O n is a shorthand for at most some constant times n Technically we say a function f n perhaps the running time of some step of an algorithm is O g n if there are constants c and n_0 such that whenever n n_0 it is true that f n cg n A useful idiom is O 1 which means some constant The use of this *big oh notation* enables us to avoid getting too far into the details of what we count as a unit of execution time yet lets us express the rate at which the running time of an algorithm grows

the state s_0 Lines 2 7 and 8 each take O 1 time Thus the running time of Algorithm 3 22 properly implemented is O k n m That is the time taken is proportional to the length of the input times the size nodes plus edges of the transition graph

3 7 4 Construction of an NFA from a Regular Expression

We now give an algorithm for converting any regular expression to an NFA that de nes the same language The algorithm is syntax directed in the sense that it works recursively up the parse tree for the regular expression For each subexpression the algorithm constructs an NFA with a single accepting state

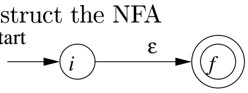
Algorithm 3 23 The McNaughton Yamada Thompson algorithm to convert a regular expression to an NFA

INPUT A regular expression r over alphabet

OUTPUT An NFA N accepting L r

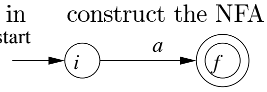
METHOD Begin by parsing r into its constituent subexpressions The rules for constructing an NFA consist of basis rules for handling subexpressions with no operators and inductive rules for constructing larger NFA s from the NFA s for the immediate subexpressions of a given expression

BASIS For expression construct the NFA



Here i is a new state the start state of this NFA and f is another new state the accepting state for the NFA

For any subexpression a in construct the NFA



where again i and f are new states the start and accepting states respectively Note that in both of the basis constructions we construct a distinct NFA with new states for every occurrence of or some a as a subexpression of r

INDUCTION Suppose N s and N t are NFA s for regular expressions s and t respectively

a Suppose r $s|t$ Then N r the NFA for r is constructed as in Fig 3 40 Here i and f are new states the start and accepting states of N r respectively There are transitions from i to the start states of N s and N t and each of their accepting states have transitions to the accepting state f Note that the accepting states of N s and N t are not accepting in N r Since any path from i to f must pass through either N s or N t exclusively and since the label of that path is not changed by the s leaving i or entering f we conclude that N r accepts L s L t which is the same as L r That is Fig 3 40 is a correct construction for the union operator

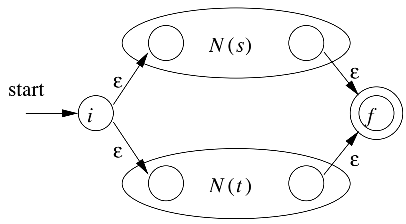


Figure 3 40 NFA for the union of two regular expressions

b Suppose r st Then construct N r as in Fig 3 41 The start state of N s becomes the start state of N r and the accepting state of N t is the only accepting state of N r The accepting state of N s and the start state of N t are merged into a single state with all the transitions in or out of either state A path from i to f in Fig 3 41 must go rst through N s and therefore its label will begin with some string in L s The path then continues through N t so the path s label nishes with a string in L t As we shall soon argue accepting states never have edges out and start states never have edges in so it is not possible for a path to re enter N s after leaving it Thus N r accepts exactly L s L t and is a correct NFA for r st

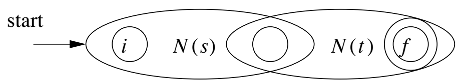


Figure 3 41 NFA for the concatenation of two regular expressions c Suppose r s Then for r we construct the NFA N r shown in Fig 3 42 Here i and f are new states the start state and lone accepting state of N r To get from i to f we can either follow the introduced path labeled which takes care of the one string in L s ^{0} or we can go to the start state of N s through that NFA then from its accepting state back to its start state zero or more times These options allow N r to accept all the strings in L s ^{1} L s ^{2} and so on so the entire set of strings accepted by N r is L s

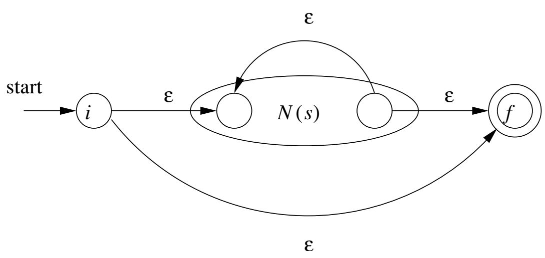


Figure 3 42 NFA for the closure of a regular expression

d Finally suppose r s Then L r L s and we can use the NFA N s as N r

□

The method description in Algorithm 3 23 contains hints as to why the inductive construction works as it should We shall not give a formal correctness proof but we shall list several properties of the constructed NFA s in addition to the all important fact that N r accepts language L r These properties are interesting in their own right and helpful in making a formal proof

1 N r has at most twice as many states as there are operators and operands in r This bound follows from the fact that each step of the algorithm creates at most two new states

2 N r has one start state and one accepting state The accepting state has no outgoing transitions and the start state has no incoming transitions

3 Each state of N r other than the accepting state has either one outgoing transition on a symbol in or two outgoing transitions both on

Example 3 24 Let us use Algorithm 3 23 to construct an NFA for r **a**|**b** **abb** Figure 3 43 shows a parse tree for r that is analogous to the parse trees constructed for arithmetic expressions in Section 2 2 3 For subexpression r_1 the rst **a** we construct the NFA

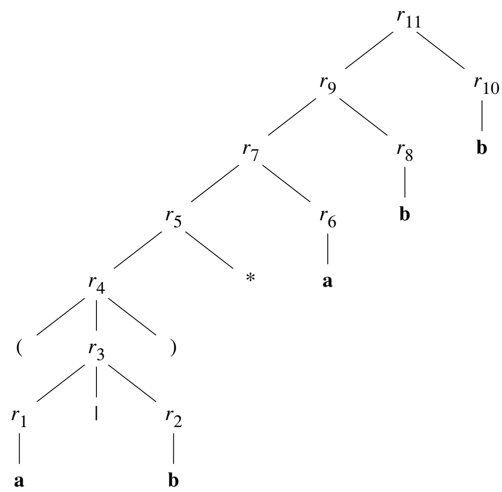
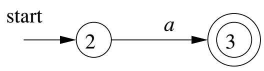
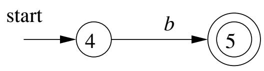


Figure 3 43 Parse tree for **a**|**b** **abb**



State numbers have been chosen for consistency with what follows For r_2 we construct



We can now combine N r_1 and N r_2 using the construction of Fig 3 40 to obtain the NFA for r_3 $r_1|r_2$ this NFA is shown in Fig 3 44

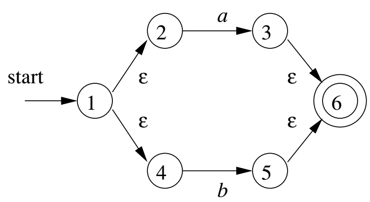


Figure 3 44 NFA for r_3

The NFA for r_4 r_3 is the same as that for r_3 The NFA for r_5 r_3 is then as shown in Fig 3 45 We have used the construction in Fig 3 42 to build this NFA from the NFA in Fig 3 44

Now consider subexpression r_6 which is another **a** We use the basis con struction for a again but we must use new states It is not permissible to reuse

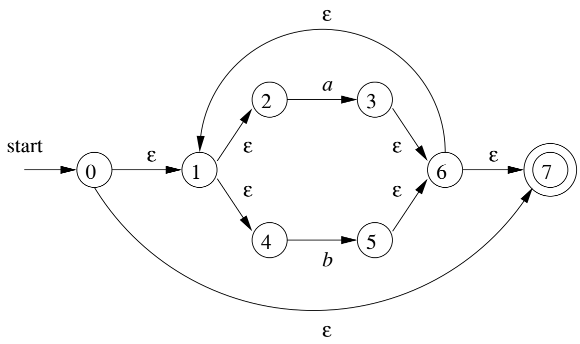
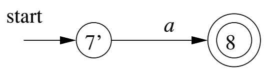


Figure 3 45 NFA for r_5

the NFA we constructed for r_1 even though r_1 and r_6 are the same expression The NFA for r_6 is



To obtain the NFA for r_7 $r_5 r_6$ we apply the construction of Fig 3 41 We merge states 7 and 7′ yielding the NFA of Fig 3 46 Continuing in this fashion with new NFA s for the two subexpressions **b** called r_8 and r_{10} we eventually construct the NFA for **a**|**b** **abb** that we rst met in Fig 3 34 □

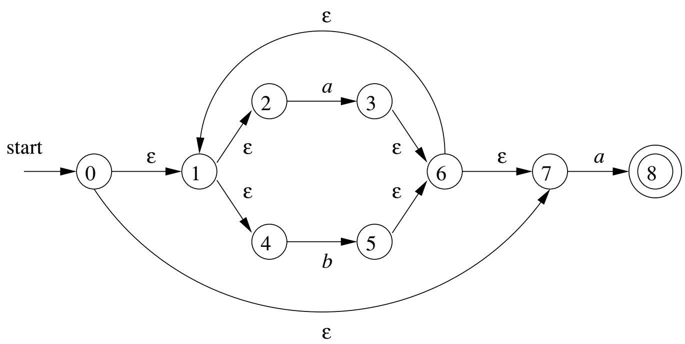


Figure 3 46 NFA for r_7

3 7 5 E ciency of String Processing Algorithms

We observed that Algorithm 3 18 processes a string x in time O $|x|$ while in Section 3 7 3 we concluded that we could simulate an NFA in time proportional to the product of $|x|$ and the size of the NFA s transition graph Obviously it is faster to have a DFA to simulate than an NFA so we might wonder whether it ever makes sense to simulate an NFA

One issue that may favor an NFA is that the subset construction can in the worst case exponentiate the number of states While in principle the number of DFA states does not in uence the running time of Algorithm 3 18 should the number of states become so large that the transition table does not t in main memory then the true running time would have to include disk I O and therefore rise noticeably

Example 3 25 Consider the family of languages described by regular expres sions of the form L_n $\mathbf{a|b}$ $\mathbf{a}$ $\mathbf{a|b}^{n\;\;1}$ that is each language L_n consists of strings of a s and b s such that the nth character to the left of the right end holds a An n 1 state NFA is easy to construct It stays in its initial state under any input but also has the option on input a of going to state 1 From state 1 it goes to state 2 on any input and so on until in state n it accepts Figure 3 47 suggests this NFA

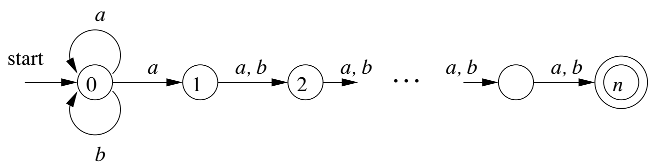


Figure 3 47 An NFA that has many fewer states than the smallest equivalent DFA

However any DFA for the language L_n must have at least 2^n states We shall not prove this fact but the idea is that if two strings of length n can get the DFA to the same state then we can exploit the last position where the strings di er and therefore one must have a the other b to continue the strings identically until they are the same in the last n 1 positions The DFA will then be in a state where it must both accept and not accept Fortunately as we mentioned it is rare for lexical analysis to involve patterns of this type and we do not expect to encounter DFA s with outlandish numbers of states in practice □

However lexical analyzer generators and other string processing systems often start with a regular expression We are faced with a choice of converting the regular expression to an NFA or DFA The additional cost of going to a DFA is thus the cost of executing Algorithm 3 20 on the NFA one could go directly from a regular expression to a DFA but the work is essentially the same If the string processor is one that will be executed many times as is the case for lexical analysis then any cost of converting to a DFA is worthwhile However in other string processing applications such as `grep` where the user speci es one regular expression and one or several les to be searched for the pattern of that expression it may be more e cient to skip the step of constructing a DFA and simulate the NFA directly

Let us consider the cost of converting a regular expression r to an NFA by Algorithm 3 23 A key step is constructing the parse tree for r In Chapter 4 we shall see several methods that are capable of constructing this parse tree in linear time that is in time $O\ |r|$ where $|r|$ stands for the *size* of r the sum of the number of operators and operands in r It is also easy to check that each of the basis and inductive constructions of Algorithm 3 23 takes constant time so the entire time spent by the conversion to an NFA is $O\ |r|$

Moreover as we observed in Section 3 7 4 the NFA we construct has at most $2|r|$ states and at most $4|r|$ transitions That is in terms of the analysis in Section 3 7 3 we have $n\quad 2|r|$ and $m\quad 4|r|$ Thus simulating this NFA on an input string x takes time $O\ |r|\quad |x|$ This time dominates the time taken by the NFA construction which is $O\ |r|$ and therefore we conclude that it is possible to take a regular expression r and string x and tell whether x is in $L\ r$ in time $O\ |r|\quad |x|$

The time taken by the subset construction is highly dependent on the num ber of states the resulting DFA has To begin notice that in the subset con struction of Fig 3 32 the key step the construction of a set of states U from a set of states T and an input symbol a is very much like the construction of a new set of states from the old set of states in the NFA simulation of Algo rithm 3 22 We already concluded that properly implemented this step takes time at most proportional to the number of states and transitions of the NFA

Suppose we start with a regular expression r and convert it to an NFA This NFA has at most $2|r|$ states and at most $4|r|$ transitions Moreover there are at most $|r|$ input symbols Thus for every DFA state constructed we must construct at most $|r|$ new states and each one takes at most $O\ |r|$ time The time to construct a DFA of s states is thus $O\ |r|^2 s$

In the common case where s is about $|r|$ the subset construction takes time $O\ |r|^3$ However in the worst case as in Example 3 25 this time is $O\ |r|^2 2^{|r|}$ Figure 3 48 summarizes the options when one is given a regular expression r and wants to produce a recognizer that will tell whether one or more strings x are in $L\ r$

AUTOMATON	INITIAL	PER STRING
NFA	$O\ \|r\|$	$O\ \|r\|\quad \|x\|$
DFA typical case	$O\ \|r\|^3$	$O\ \|x\|$
DFA worst case	$O\ \|r\|^2 2^{\|r\|}$	$O\ \|x\|$

Figure 3 48 Initial cost and per string cost of various methods of recognizing the language of a regular expression

If the per string cost dominates as it does when we build a lexical analyzer we clearly prefer the DFA However in commands like `grep` where we run the automaton on only one string we generally prefer the NFA It is not until $|x|$ approaches $|r|^3$ that we would even think about converting to a DFA

There is however a mixed strategy that is about as good as the better of the NFA and the DFA strategy for each expression r and string x Start o simulating the NFA but remember the sets of NFA states i e the DFA states and their transitions as we compute them Before processing the current set of NFA states and the current input symbol check to see whether we have already computed this transition and use the information if so

3 7 6 Exercises for Section 3 7

Exercise 3 7 1 Convert to DFA s the NFA s of

a Fig 3 26

b Fig 3 29

c Fig 3 30

Exercise 3 7 2 use Algorithm 3 22 to simulate the NFA s

a Fig 3 29

b Fig 3 30

on input *aabb*

Exercise 3 7 3 Convert the following regular expressions to deterministic nite automata using algorithms 3 23 and 3 20

a **a|b**

b **a |b**

c **|a b**

d **a|b abb a|b**

3 8 Design of a Lexical Analyzer Generator

In this section we shall apply the techniques presented in Section 3 7 to see how a lexical analyzer generator such as `Lex` is architected We discuss two approaches based on NFA s and DFA s the latter is essentially the implemen tation of `Lex`

3 8 1 The Structure of the Generated Analyzer

Figure 3 49 overviews the architecture of a lexical analyzer generated by `Lex` The program that serves as the lexical analyzer includes a xed program that simulates an automaton at this point we leave open whether that automaton is deterministic or nondeterministic The rest of the lexical analyzer consists of components that are created from the `Lex` program by `Lex` itself

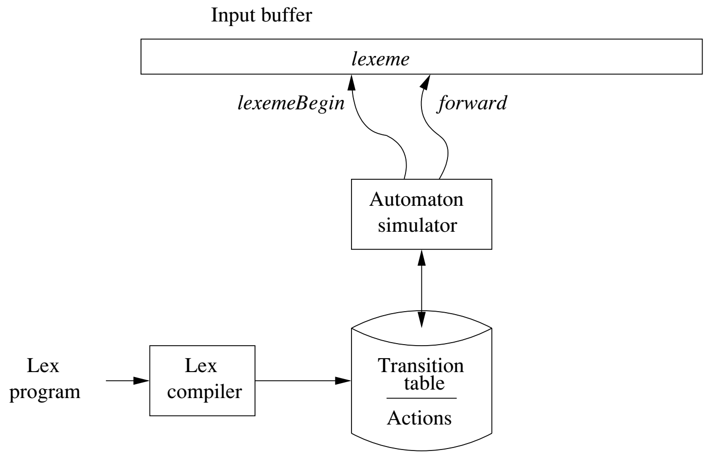


Figure 3 49 A `Lex` program is turned into a transition table and actions which are used by a nite automaton simulator

These components are

1 A transition table for the automaton

2 Those functions that are passed directly through `Lex` to the output see Section 3 5 2

3 The actions from the input program which appear as fragments of code to be invoked at the appropriate time by the automaton simulator

To construct the automaton we begin by taking each regular expression pattern in the `Lex` program and converting it using Algorithm 3 23 to an NFA We need a single automaton that will recognize lexemes matching any of the patterns in the program so we combine all the NFA s into one by introducing a new start state with transitions to each of the start states of the NFA s N_i for pattern p_i This construction is shown in Fig 3 50

Example 3 26 We shall illustrate the ideas of this section with the following simple abstract example

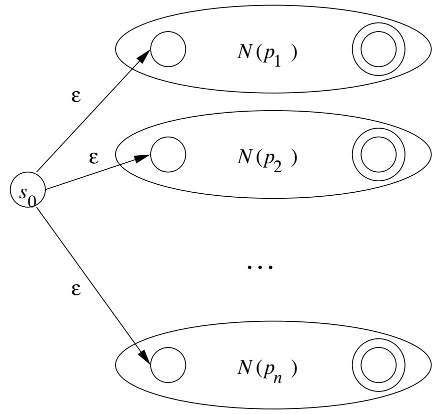


Figure 3 50 An NFA constructed from a `Lex` program

a	{ action A_1 for pattern p_1 }
abb	{ action A_2 for pattern p_2 }
a b	{ action A_3 for pattern p_3 }

Note that these three patterns present some con icts of the type discussed in Section 3 5 3 In particular string *abb* matches both the second and third patterns but we shall consider it a lexeme for pattern p_2 since that pattern is listed rst in the above `Lex` program Then input strings such as *aabbb* have many pre xes that match the third pattern The `Lex` rule is to take the longest so we continue reading *b* s until another *a* is met whereupon we report the lexeme to be the initial *a* s followed by as many *b* s as there are

Figure 3 51 shows three NFA s that recognize the three patterns The third is a simpli cation of what would come out of Algorithm 3 23 Then Fig 3 52 shows these three NFA s combined into a single NFA by the addition of start state 0 and three transitions □

3 8 2 Pattern Matching Based on NFA s

If the lexical analyzer simulates an NFA such as that of Fig 3 52 then it must read input beginning at the point on its input which we have referred to as *lexemeBegin* As it moves the pointer called *forward* ahead in the input it calculates the set of states it is in at each point following Algorithm 3 22

Eventually the NFA simulation reaches a point on the input where there are no next states At that point there is no hope that any longer pre x of the input would ever get the NFA to an accepting state rather the set of states will always be empty Thus we are ready to decide on the longest pre x that is a lexeme matching some pattern

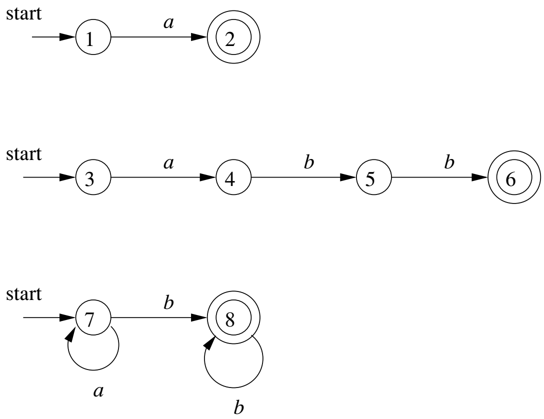


Figure 3 51 NFA s for **a** **abb** and **a** **b**

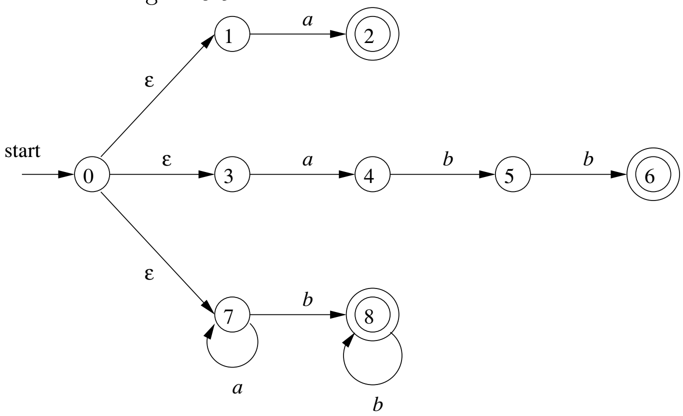


Figure 3 52 Combined NFA

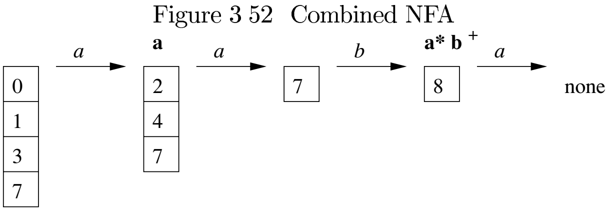


Figure 3 53 Sequence of sets of states entered when processing input *aaba*

We look backwards in the sequence of sets of states until we nd a set that includes one or more accepting states If there are several accepting states in that set pick the one associated with the earliest pattern p_i in the list from the `Lex` program Move the *forward* pointer back to the end of the lexeme and perform the action A_i associated with pattern p_i

Example 3 27 Suppose we have the patterns of Example 3 26 and the input begins *aaba* Figure 3 53 shows the sets of states of the NFA of Fig 3 52 that we enter starting with *closure* of the initial state 0 which is $\{0\ 1\ 3\ 7\}$ and proceeding from there After reading the fourth input symbol we are in an empty set of states since in Fig 3 52 there are no transitions out of state 8 on input a

Thus we need to back up looking for a set of states that includes an ac cepting state Notice that as indicated in Fig 3 53 after reading a we are in a set that includes state 2 and therefore indicates that the pattern **a** has been matched However after reading *aab* we are in state 8 which indicates that **a b** has been matched pre x *aab* is the longest pre x that gets us to an accepting state We therefore select *aab* as the lexeme and execute action A_3 which should include a return to the parser indicating that the token whose pattern is p_3 **a b** has been found □

3 8 3 DFA s for Lexical Analyzers

Another architecture resembling the output of `Lex` is to convert the NFA for all the patterns into an equivalent DFA using the subset construction of Algorithm 3 20 Within each DFA state if there are one or more accepting NFA states determine the rst pattern whose accepting state is represented and make that pattern the output of the DFA state

Example 3 28 Figure 3 54 shows a transition diagram based on the DFA that is constructed by the subset construction from the NFA in Fig 3 52 The accepting states are labeled by the pattern that is identi ed by that state For instance the state $\{6\ 8\}$ has two accepting states corresponding to patterns **abb** and **a b** Since the former is listed rst that is the pattern associated with state $\{6\ 8\}$ □

We use the DFA in a lexical analyzer much as we did the NFA We simulate the DFA until at some point there is no next state or strictly speaking the next state is the *dead state* corresponding to the empty set of NFA states At that point we back up through the sequence of states we entered and as soon as we meet an accepting DFA state we perform the action associated with the pattern for that state

Example 3 29 Suppose the DFA of Fig 3 54 is given input *abba* The se quence of states entered is 0137 247 58 68 and at the nal a there is no tran sition out of state 68 Thus we consider the sequence from the end and in this case 68 itself is an accepting state that reports pattern p_2 **abb** □

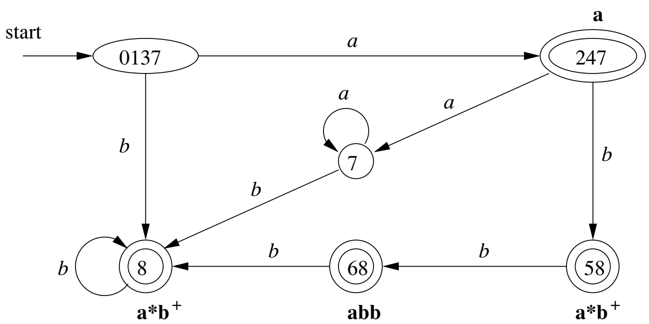


Figure 3 54 Transition graph for DFA handling the patterns **a** **abb** and **a** **b**

3 8 4 Implementing the Lookahead Operator

Recall from Section 3 5 4 that the `Lex` lookahead operator in a `Lex` pattern r_1 r_2 is sometimes necessary because the pattern r_1 for a particular token may need to describe some trailing context r_2 in order to correctly identify the actual lexeme When converting the pattern r_1 r_2 to an NFA we treat the as if it were so we do not actually look for a on the input However if the NFA recognizes a pre x xy of the input bu er as matching this regular expression the end of the lexeme is not where the NFA entered its accepting state Rather the end occurs when the NFA enters a state s such that

1 s has an transition on the imaginary

2 There is a path from the start state of the NFA to state s that spells out x

3 There is a path from state s to the accepting state that spells out y

4 x is as long as possible for any xy satisfying conditions 1 3

If there is only one transition state on the imaginary in the NFA then the end of the lexeme occurs when this state is entered for the last time as the following example illustrates If the NFA has more than one transition state on the imaginary then the general problem of nding the correct state s is much more di cult

Example 3 30 An NFA for the pattern for the Fortran `IF` with lookahead from Example 3 13 is shown in Fig 3 55 Notice that the transition from state 2 to state 3 represents the lookahead operator State 6 indicates the pres ence of the keyword `IF` However we nd the lexeme `IF` by scanning backwards to the last occurrence of state 2 whenever state 6 is entered □

Dead States in DFA s

Technically the automaton in Fig 3 54 is not quite a DFA The reason is that a DFA has a transition from every state on every input symbol in its input alphabet Here we have omitted transitions to the dead state and we have therefore omitted the transitions from the dead state to itself on every input Previous NFA to DFA examples did not have a way to get from the start state to but the NFA of Fig 3 52 does

However when we construct a DFA for use in a lexical analyzer it is important that we treat the dead state di erently since we must know when there is no longer any possibility of recognizing a longer lexeme Thus we suggest always omitting transitions to the dead state and elimi nating the dead state itself In fact the problem is harder than it appears since an NFA to DFA construction may yield several states that cannot reach any accepting state and we must know when any of these states have been reached Section 3 9 6 discusses how to combine all these states into one dead state so their identi cation becomes easy It is also inter esting to note that if we construct a DFA from a regular expression using Algorithms 3 20 and 3 23 then the DFA will not have any states besides that cannot lead to an accepting state

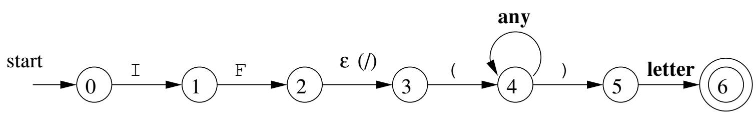


Figure 3 55 NFA recognizing the keyword `IF`

3 8 5 Exercises for Section 3 8

Exercise 3 8 1 Suppose we have two tokens 1 the keyword `if` and 2 id enti ers which are strings of letters other than `if` Show

a The NFA for these tokens and

b The DFA for these tokens

Exercise 3 8 2 Repeat Exercise 3 8 1 for tokens consisting of 1 the keyword `while` 2 the keyword `when` and 3 identi ers consisting of strings of letters and digits beginning with a letter

Exercise 3 8 3 Suppose we were to revise the de nition of a DFA to allow zero or one transition out of each state on each input symbol rather than exactly one such transition as in the standard DFA de nition Some regular expressions would then have smaller DFA s than they do under the standard de nition of a DFA Give an example of one such regular expression

Exercise 3 8 4 Design an algorithm to recognize `Lex` lookahead patterns of the form r_1 r_2 where r_1 and r_2 are regular expressions Show how your algo rithm works on the following inputs

a **abcd|abc d**

b **a|ab ba**

c **aa a**

3 9 Optimization of DFA Based Pattern Matchers

In this section we present three algorithms that have been used to implement and optimize pattern matchers constructed from regular expressions

1 The rst algorithm is useful in a `Lex` compiler because it constructs a DFA directly from a regular expression without constructing an interme diate NFA The resulting DFA also may have fewer states than the DFA constructed via an NFA

2 The second algorithm minimizes the number of states of any DFA by combining states that have the same future behavior The algorithm itself is quite e cient running in time O $n \log n$ where n is the number of states of the DFA

3 The third algorithm produces more compact representations of transition tables than the standard two dimensional table

3 9 1 Important States of an NFA

To begin our discussion of how to go directly from a regular expression to a DFA we must rst dissect the NFA construction of Algorithm 3 23 and consider the roles played by various states We call a state of an NFA *important* if it has a non out transition Notice that the subset construction Algorithm 3 20 uses only the important states in a set T when it computes *closure move* T a the set of states reachable from T on input a That is the set of states *move* s a is nonempty only if state s is important During the subset construction two sets of NFA states can be identi ed treated as if they were the same set if they

1 Have the same important states and

2 Either both have accepting states or neither does

When the NFA is constructed from a regular expression by Algorithm 3 23 we can say more about the important states The only important states are those introduced as initial states in the basis part for a particular symbol posi tion in the regular expression That is each important state corresponds to a particular operand in the regular expression

The constructed NFA has only one accepting state but this state having no out transitions is not an important state By concatenating a unique right endmarker to a regular expression r we give the accepting state for r a transition on making it an important state of the NFA for r In other words by using the *augmented* regular expression r we can forget about accepting states as the subset construction proceeds when the construction is complete any state with a transition on must be an accepting state

The important states of the NFA correspond directly to the positions in the regular expression that hold symbols of the alphabet It is useful as we shall see to present the regular expression by its *syntax tree* where the leaves correspond to operands and the interior nodes correspond to operators An interior node is called a *cat node or node* or *star node* if it is labeled by the concatenation operator dot union operator | or star operator respectively We can construct a syntax tree for a regular expression just as we did for arithmetic expressions in Section 2 5 1

Example 3 31 Figure 3 56 shows the syntax tree for the regular expression of our running example Cat nodes are represented by circles □

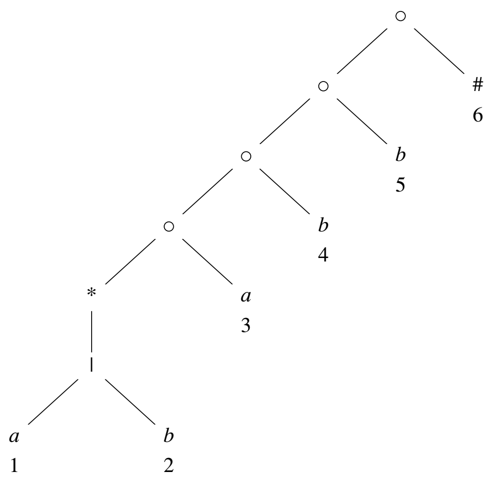


Figure 3 56 Syntax tree for **a|b abb**

Leaves in a syntax tree are labeled by or by an alphabet symbol To each leaf not labeled we attach a unique integer We refer to this integer as the *position* of the leaf and also as a position of its symbol Note that a symbol can have several positions for instance a has positions 1 and 3 in Fig 3 56 The positions in the syntax tree correspond to the important states of the constructed NFA

Example 3 32 Figure 3 57 shows the NFA for the same regular expression as Fig 3 56 with the important states numbered and other states represented by letters The numbered states in the NFA and the positions in the syntax tree correspond in a way we shall soon see □

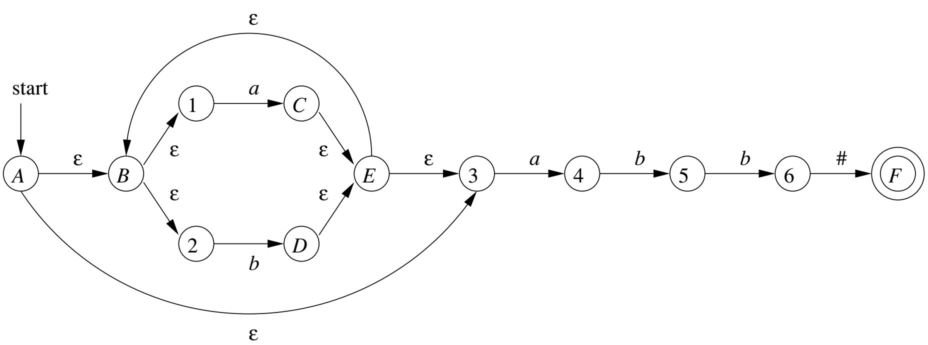


Figure 3 57 NFA constructed by Algorithm 3 23 for **a|b** **abb**

3 9 2 Functions Computed From the Syntax Tree

To construct a DFA directly from a regular expression we construct its syntax tree and then compute four functions *nullable* *rstpos* *lastpos* and *followpos* de ned as follows Each de nition refers to the syntax tree for a particular augmented regular expression r

1 *nullable* n is true for a syntax tree node n if and only if the subexpression represented by n has in its language That is the subexpression can be made null or the empty string even though there may be other strings it can represent as well

2 *rstpos* n is the set of positions in the subtree rooted at n that corre spond to the rst symbol of at least one string in the language of the subexpression rooted at n

3 *lastpos* n is the set of positions in the subtree rooted at n that corre spond to the last symbol of at least one string in the language of the subexpression rooted at n

4 *followpos* p for a position p is the set of positions q in the entire syntax tree such that there is some string x $a_1 a_2$ a_n in L r such that for some i there is a way to explain the membership of x in L r by matching a_i to position p of the syntax tree and a_i $_1$ to position q

Example 3 33 Consider the cat node n in Fig 3 56 that corresponds to the expression **a|b a** We claim *nullable* n is false since this node generates all strings of a s and b s ending in an a it does not generate On the other hand the star node below it is nullable it generates along with all other strings of a s and b s

rstpos n $\{1\ 2\ 3\}$ In a typical generated string like aa the rst position of the string corresponds to position 1 of the tree and in a string like ba the rst position of the string comes from position 2 of the tree However when the string generated by the expression of node n is just a then this a comes from position 3

lastpos n $\{3\}$ That is no matter what string is generated from the expression of node n the last position is the a from position 3 of the tree

followpos is trickier to compute but we shall see the rules for doing so shortly Here is an example of the reasoning *followpos* 1 $\{1\ 2\ 3\}$ Consider a string ac where the c is either a or b and the a comes from position 1 That is this a is one of those generated by the **a** in expression **a|b** This a could be followed by another a or b coming from the same subexpression in which case c comes from position 1 or 2 It is also possible that this a is the last in the string generated by **a|b** in which case the symbol c must be the a that comes from position 3 Thus 1 2 and 3 are exactly the positions that can follow position 1 □

3 9 3 Computing *nullable* *rstpos* and *lastpos*

We can compute *nullable* *rstpos* and *lastpos* by a straightforward recursion on the height of the tree The basis and inductive rules for *nullable* and *rstpos* are summarized in Fig 3 58 The rules for *lastpos* are essentially the same as for *rstpos* but the roles of children c_1 and c_2 must be swapped in the rule for a cat node

Example 3 34 Of all the nodes in Fig 3 56 only the star node is nullable We note from the table of Fig 3 58 that none of the leaves are nullable because they each correspond to non operands The or node is not nullable because neither of its children is The star node is nullable because every star node is nullable Finally each of the cat nodes having at least one nonnullable child is not nullable

The computation of *rstpos* and *lastpos* for each of the nodes is shown in Fig 3 59 with *rstpos* n to the left of node n and *lastpos* n to its right Each of the leaves has only itself for *rstpos* and *lastpos* as required by the rule for non leaves in Fig 3 58 For the or node we take the union of *rstpos* at the

NODE n	*nullable* n	*rstpos* n
A leaf labeled	**true**	
A leaf with position i	**false**	$\{i\}$
An or node n $c_1 \| c_2$	*nullable* c_1 **or** *nullable* c_2	*rstpos* c_1 *rstpos* c_2
A cat node n $c_1 c_2$	*nullable* c_1 **and** *nullable* c_2	**if** *nullable* c_1 *rstpos* c_1 *rstpos* c_2 **else** *rstpos* c_1
A star node n c_1	**true**	*rstpos* c_1

Figure 3 58 Rules for computing *nullable* and *rstpos*

children and do the same for *lastpos* The rule for the star node says that we take the value of *rstpos* or *lastpos* at the one child of that node

Now consider the lowest cat node which we shall call n To compute *rstpos* n we rst consider whether the left operand is nullable which it is in this case Therefore *rstpos* for n is the union of *rstpos* for each of its children that is $\{1\ 2\}$ $\{3\}$ $\{1\ 2\ 3\}$ The rule for *lastpos* does not ap pear explicitly in Fig 3 58 but as we mentioned the rules are the same as for *rstpos* with the children interchanged That is to compute *lastpos* n we must ask whether its right child the leaf with position 3 is nullable which it is not Therefore *lastpos* n is the same as *lastpos* of the right child or $\{3\}$
□

3 9 4 Computing *followpos*

Finally we need to see how to compute *followpos* There are only two ways that a position of a regular expression can be made to follow another

1 If n is a cat node with left child c_1 and right child c_2 then for every position i in *lastpos* c_1 all positions in *rstpos* c_2 are in *followpos* i

2 If n is a star node and i is a position in *lastpos* n then all positions in *rstpos* n are in *followpos* i

Example 3 35 Let us continue with our running example recall that *rstpos* and *lastpos* were computed in Fig 3 59 Rule 1 for *followpos* requires that we look at each cat node and put each position in *rstpos* of its right child in *followpos* for each position in *lastpos* of its left child For the lowest cat node in Fig 3 59 that rule says position 3 is in *followpos* 1 and *followpos* 2 The next cat node above says that 4 is in *followpos* 3 and the remaining two cat nodes give us 5 in *followpos* 4 and 6 in *followpos* 5

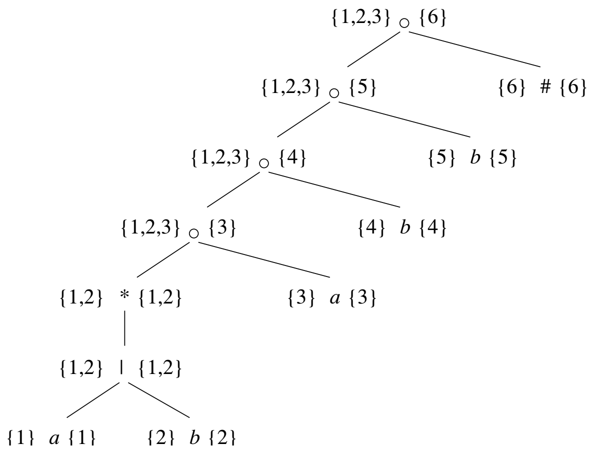


Figure 3 59 *rstpos* and *lastpos* for nodes in the syntax tree for **a|b abb**

We must also apply rule 2 to the star node That rule tells us positions 1 and 2 are in both *followpos* 1 and *followpos* 2 since both *rstpos* and *lastpos* for this node are {1 2} The complete sets *followpos* are summarized in Fig 3 60 □

POSITION n	*followpos* n
1	{1 2 3}
2	{1 2 3}
3	{4}
4	{5}
5	{6}
6	

Figure 3 60 The function *followpos*

We can represent the function *followpos* by creating a directed graph with a node for each position and an arc from position i to position j if and only if j is in *followpos* i Figure 3 61 shows this graph for the function of Fig 3 60

It should come as no surprise that the graph for *followpos* is almost an NFA without transitions for the underlying regular expression and would become one if we

1 Make all positions in *rstpos* of the root be initial states

2 Label each arc from i to j by the symbol at position i and

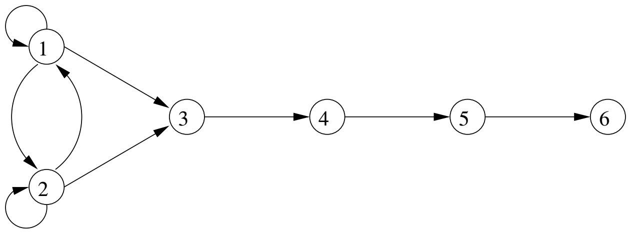


Figure 3 61 Directed graph for the function *followpos*

3 Make the position associated with endmarker be the only accepting state

3 9 5 Converting a Regular Expression Directly to a DFA

Algorithm 3 36 Construction of a DFA from a regular expression r

INPUT A regular expression r

OUTPUT A DFA D that recognizes L r

METHOD

1 Construct a syntax tree T from the augmented regular expression r

2 Compute *nullable* *rstpos* *lastpos* and *followpos* for T using the methods of Sections 3 9 3 and 3 9 4

3 Construct *Dstates* the set of states of DFA D and *Dtran* the transition function for D by the procedure of Fig 3 62 The states of D are sets of positions in T Initially each state is unmarked and a state becomes marked just before we consider its out transitions The start state of D is *rstpos* n_0 where node n_0 is the root of T The accepting states are those containing the position for the endmarker symbol

□

Example 3 37 We can now put together the steps of our running example to construct a DFA for the regular expression r **a**|**b** **abb** The syntax tree for r appeared in Fig 3 56 We observed that for this tree *nullable* is true only for the star node and we exhibited *rstpos* and *lastpos* in Fig 3 59 The values of *followpos* appear in Fig 3 60

The value of *rstpos* for the root of the tree is $\{1\ 2\ 3\}$ so this set is the start state of D Call this set of states A We must compute *Dtran* A a and *Dtran* A b Among the positions of A 1 and 3 correspond to a while 2 corresponds to b Thus *Dtran* A a *followpos* 1 *followpos* 3 $\{1\ 2\ 3\ 4\}$

initialize *Dstates* to contain only the unmarked state *rstpos* n_0
  where n_0 is the root of syntax tree T for r
while there is an unmarked state S in *Dstates* {
  mark S
  for each input symbol a {
    let U be the union of *followpos* p for all p
      in S that correspond to a
    if U is not in *Dstates*
      add U as an unmarked state to *Dstates*
    Dtran S a U
  }
}

Figure 3 62 Construction of a DFA directly from a regular expression

and *Dtran* A b *followpos* 2 $\{1\ 2\ 3\}$ The latter is state A and so does not have to be added to *Dstates* but the former B $\{1\ 2\ 3\ 4\}$ is new so we add it to *Dstates* and proceed to compute its transitions The complete DFA is shown in Fig 3 63 □

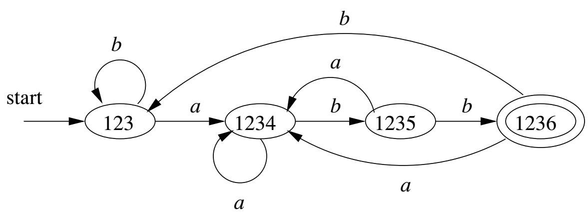


Figure 3 63 DFA constructed from Fig 3 57

3 9 6 Minimizing the Number of States of a DFA

There can be many DFA s that recognize the same language For instance note that the DFA s of Figs 3 36 and 3 63 both recognize language L **a**|**b** **abb** Not only do these automata have states with di erent names but they don t even have the same number of states If we implement a lexical analyzer as a DFA we would generally prefer a DFA with as few states as possible since each state requires entries in the table that describes the lexical analyzer

The matter of the names of states is minor We shall say that two automata are *the same up to state names* if one can be transformed into the other by doing nothing more than changing the names of states Figures 3 36 and 3 63 are not the same up to state names However there is a close relationship between the states of each States A and C of Fig 3 36 are actually equivalent in the sense that neither is an accepting state and on any input they transfer to the same state to B on input a and to C on input b Moreover both states A and C behave like state 123 of Fig 3 63 Likewise state B of Fig 3 36 behaves like state 1234 of Fig 3 63 state D behaves like state 1235 and state E behaves like state 1236

It turns out that there is always a unique up to state names minimum state DFA for any regular language Moreover this minimum state DFA can be constructed from any DFA for the same language by grouping sets of equivalent states In the case of L **a|b** **abb** Fig 3 63 is the minimum state DFA and it can be constructed by partitioning the states of Fig 3 36 as $\{A\ C\}\{B\}\{D\}\{E\}$

In order to understand the algorithm for creating the partition of states that converts any DFA into its minimum state equivalent DFA we need to see how input strings distinguish states from one another We say that string x *distinguishes* state s from state t if exactly one of the states reached from s and t by following the path with label x is an accepting state State s is *distinguishable* from state t if there is some string that distinguishes them

Example 3 38 The empty string distinguishes any accepting state from any nonaccepting state In Fig 3 36 the string *bb* distinguishes state A from state B since *bb* takes A to a nonaccepting state C but takes B to accepting state E □

The state minimization algorithm works by partitioning the states of a DFA into groups of states that cannot be distinguished Each group of states is then merged into a single state of the minimum state DFA The algorithm works by maintaining a partition whose groups are sets of states that have not yet been distinguished while any two states from di erent groups are known to be distinguishable When the partition cannot be re ned further by breaking any group into smaller groups we have the minimum state DFA

Initially the partition consists of two groups the accepting states and the nonaccepting states The fundamental step is to take some group of the current partition say A $\{s_1\ s_2\ \ \ s_k\}$ and some input symbol a and see whether a can be used to distinguish between any states in group A We examine the transitions from each of s_1 s_2 s_k on input a and if the states reached fall into two or more groups of the current partition we split A into a collection of groups so that s_i and s_j are in the same group if and only if they go to the same group on input a We repeat this process of splitting groups until for no group and for no input symbol can the group be split further The idea is formalized in the next algorithm

Algorithm 3 39 Minimizing the number of states of a DFA

INPUT A DFA D with set of states S input alphabet start state s_0 and set of accepting states F

OUTPUT A DFA D' accepting the same language as D and having as few states as possible

Why the State Minimization Algorithm Works

We need to prove two things that states remaining in the same group in $_{\text{nal}}$ are indistinguishable by any string and that states winding up in di erent groups are distinguishable The rst is an induction on i that if after the ith iteration of step 2 of Algorithm 3 39 s and t are in the same group then there is no string of length i or less that distinguishes them We shall leave the details of the induction to you

The second is an induction on i that if states s and t are placed in di erent groups at the ith iteration of step 2 then there is a string that distinguishes them The basis when s and t are placed in di erent groups of the initial partition is easy one must be accepting and the other not so distinguishes them For the induction there must be an input a and states p and q such that s and t go to states p and q respectively on input a Moreover p and q must already have been placed in di erent groups Then by the inductive hypothesis there is some string x that distinguishes p from q Therefore ax distinguishes s from t

METHOD

1 Start with an initial partition with two groups F and S F the accepting and nonaccepting states of D

2 Apply the procedure of Fig 3 64 to construct a new partition $_{\text{new}}$

```
initially let  new
for  each group G of      {
        partition G into subgroups such that two states s and t
                are in the same subgroup if and only if for all
                input symbols a  states s and t have transitions on a
                to states in the same group of
            at worst  a state will be in a subgroup by itself
        replace G in  new by the set of all subgroups formed
}
```

Figure 3 64 Construction of $_{\text{new}}$

3 If $_{\text{new}}$ let $_{\text{nal}}$ and continue with step 4 Otherwise repeat step 2 with $_{\text{new}}$ in place of

4 Choose one state in each group of $_{\text{nal}}$ as the *representative* for that group The representatives will be the states of the minimum state DFA D' The other components of D' are constructed as follows

Eliminating the Dead State

The minimization algorithm sometimes produces a DFA with one dead state one that is not accepting and transfers to itself on each input symbol This state is technically needed because a DFA must have a transition from every state on every symbol However as discussed in Section 3 8 3 we often want to know when there is no longer any possibility of acceptance so we can establish that the proper lexeme has already been seen Thus we may wish to eliminate the dead state and use an automaton that is missing some transitions This automaton has one fewer state than the minimum state DFA but is strictly speaking not a DFA because of the missing transitions to the dead state

a The start state of D' is the representative of the group containing the start state of D

b The accepting states of D' are the representatives of those groups that contain an accepting state of D Note that each group contains either only accepting states or only nonaccepting states because we started by separating those two classes of states and the procedure of Fig 3 64 always forms new groups that are subgroups of previously constructed groups

c Let s be the representative of some group G of $_{\text{nal}}$ and let the transition of D from s on input a be to state t Let r be the rep resentative of t s group H Then in D' there is a transition from s to r on input a Note that in D every state in group G must go to some state of group H on input a or else group G would have been split according to Fig 3 64

□

Example 3 40 Let us reconsider the DFA of Fig 3 36 The initial partition consists of the two groups $\{A\ B\ C\ D\}\{E\}$ which are respectively the nonac cepting states and the accepting states To construct $_{\text{new}}$ the procedure of Fig 3 64 considers both groups and inputs a and b The group $\{E\}$ cannot be split because it has only one state so $\{E\}$ will remain intact in $_{\text{new}}$

The other group $\{A\ B\ C\ D\}$ can be split so we must consider the e ect of each input symbol On input a each of these states goes to state B so there is no way to distinguish these states using strings that begin with a On input b states A B and C go to members of group $\{A\ B\ C\ D\}$ while state D goes to E a member of another group Thus in $_{\text{new}}$ group $\{A\ B\ C\ D\}$ is split into $\{A\ B\ C\}\{D\}$ and $_{\text{new}}$ for this round is $\{A\ B\ C\}\{D\}\{E\}$

In the next round we can split $\{A\ B\ C\}$ into $\{A\ C\}\{B\}$ since A and C each go to a member of $\{A\ B\ C\}$ on input b while B goes to a member of another group $\{D\}$ Thus after the second round $_{\text{new}}$ $\{A\ C\}\{B\}\{D\}\{E\}$ For the third round we cannot split the one remaining group with more than one state since A and C each go to the same state and therefore to the same group on each input We conclude that $_{\text{nal}}$ $\{A\ C\}\{B\}\{D\}\{E\}$

Now we shall construct the minimum state DFA It has four states corre sponding to the four groups of $_{\text{nal}}$ and let us pick A B D and E as the representatives of these groups The initial state is A and the only accepting state is E Figure 3 65 shows the transition function for the DFA For instance the transition from state E on input b is to A since in the original DFA E goes to C on input b and A is the representative of C s group For the same reason the transition on b from state A is to A itself while all other transitions are as in Fig 3 36 □

STATE	a	b
A	B	A
B	B	D
D	B	E
E	B	A

Figure 3 65 Transition table of minimum state DFA

3 9 7 State Minimization in Lexical Analyzers

To apply the state minimization procedure to the DFA s generated in Sec tion 3 8 3 we must begin Algorithm 3 39 with the partition that groups to gether all states that recognize a particular token and also places in one group all those states that do not indicate any token An example should make the extension clear

Example 3 41 For the DFA of Fig 3 54 the initial partition is

$$\{0137\ 7\}\{247\}\{8\ 58\}\{68\}\{\ \}$$

That is states 0137 and 7 belong together because neither announces any token States 8 and 58 belong together because they both announce token **a b** Note that we have added a dead state which we suppose has transitions to itself on inputs a and b The dead state is also the target of missing transitions on a from states 8 58 and 68

We must split 0137 from 7 because they go to di erent groups on input a We also split 8 from 58 because they go to di erent groups on b Thus all states are in groups by themselves and Fig 3 54 is the minimum state DFA recognizing its three tokens Recall that a DFA serving as a lexical analyzer will normally drop the dead state while we treat missing transitions as a signal to end token recognition □

3 9 8 Trading Time for Space in DFA Simulation

The simplest and fastest way to represent the transition function of a DFA is a two dimensional table indexed by states and characters Given a state and next input character we access the array to nd the next state and any special action we must take e g returning a token to the parser Since a typical lexical analyzer has several hundred states in its DFA and involves the ASCII alphabet of 128 input characters the array consumes less than a megabyte

However compilers are also appearing in very small devices where even a megabyte of storage may be too much For such situations there are many methods that can be used to compact the transition table For instance we can represent each state by a list of transitions that is character state pairs ended by a default state that is to be chosen for any input character not on the list If we choose as the default the most frequently occurring next state we can often reduce the amount of storage needed by a large factor

There is a more subtle data structure that allows us to combine the speed of array access with the compression of lists with defaults We may think of this structure as four arrays as suggested in Fig 3 66 [5] The *base* array is used to determine the base location of the entries for state s which are located in the *next* and *check* arrays The *default* array is used to determine an alternative base location if the *check* array tells us the one given by *base* s is invalid

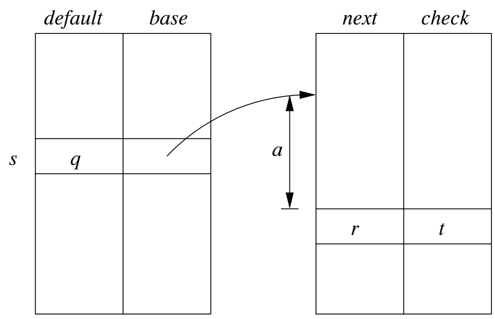


Figure 3 66 Data structure for representing transition tables

To compute *nextState* s a the transition for state s on input a we examine the *next* and *check* entries in location l *base* s a where character a is treated as an integer presumably in the range 0 to 127 If *check* l s then this entry

[5]In practice there would be another array indexed by states to give the action associated with that state if any

is valid and the next state for state s on input a is *next l* If *check l* / s then we determine another state t *default s* and repeat the process as if t were the current state More formally the function *nextState* is de ned as follows

> int *nextState* s a {
> **if** *check base s* a s **return** *next base s* a
> **else return** *nextState default s* a
> }

The intended use of the structure of Fig 3 66 is to make the *next check* arrays short by taking advantage of the similarities among states For instance state t the default for state s might be the state that says we are working on an identi er like state 10 in Fig 3 14 Perhaps state s is entered after seeing the letters `th` which are a pre x of keyword `then` as well as potentially being the pre x of some lexeme for an identi er On input character `e` we must go from state s to a special state that remembers we have seen `the` but otherwise state s behaves as t does Thus we set *check base s* `e` to s to con rm that this entry is valid for s and we set *next base s* `e` to the state that remembers `the` Also *default s* is set to t

While we may not be able to choose *base* values so that no *next check* entries remain unused experience has shown that the simple strategy of assigning *base* values to states in turn and assigning each *base s* value the lowest integer so that the special entries for state s are not previously occupied utilizes little more space than the minimum possible

3 9 9 Exercises for Section 3 9

Exercise 3 9 1 Extend the table of Fig 3 58 to include the operators a and b

Exercise 3 9 2 Use Algorithm 3 36 to convert the regular expressions of Ex ercise 3 7 3 directly to deterministic nite automata

Exercise 3 9 3 We can prove that two regular expressions are equivalent by showing that their minimum state DFA s are the same up to renaming of states Show in this way that the following regular expressions **a|b** **a |b** and **|a b** are all equivalent *Note* You may have constructed the DFA s for these expressions in response to Exercise 3 7 3

Exercise 3 9 4 Construct the minimum state DFA s for the following regular expressions

a **a|b a a|b**

b **a|b a a|b a|b**

c **a|b a a|b a|b a|b**

Do you see a pattern

Exercise 3 9 5 To make formal the informal claim of Example 3 25 show that any deterministic nite automaton for the regular expression

$$\mathbf{a|b} \quad \mathbf{a} \; \mathbf{a|b} \quad \mathbf{a|b} \qquad \mathbf{a|b}$$

where **a|b** appears n 1 times at the end must have at least 2^n states *Hint* Observe the pattern in Exercise 3 9 4 What condition regarding the history of inputs does each state represent

3 10 Summary of Chapter 3

✦ *Tokens* The lexical analyzer scans the source program and produces as output a sequence of tokens which are normally passed one at a time to the parser Some tokens may consist only of a token name while others may also have an associated lexical value that gives information about the particular instance of the token that has been found on the input

✦ *Lexemes* Each time the lexical analyzer returns a token to the parser it has an associated lexeme the sequence of input characters that the token represents

✦ *Bu ering* Because it is often necessary to scan ahead on the input in order to see where the next lexeme ends it is usually necessary for the lexical analyzer to bu er its input Using a pair of bu ers cyclicly and ending each bu er s contents with a sentinel that warns of its end are two techniques that accelerate the process of scanning the input

✦ *Patterns* Each token has a pattern that describes which sequences of characters can form the lexemes corresponding to that token The set of words or strings of characters that match a given pattern is called a language

✦ *Regular Expressions* These expressions are commonly used to describe patterns Regular expressions are built from single characters using union concatenation and the Kleene closure or any number of oper ator

✦ *Regular De nitions* Complex collections of languages such as the pat terns that describe the tokens of a programming language are often de ned by a regular de nition which is a sequence of statements that each de ne one variable to stand for some regular expression The regular ex pression for one variable can use previously de ned variables in its regular expression