Scanner Termination

A scanner reads input characters and partitions them into tokens.

What happens when the end of the input file is reached? It may be useful to create an **`Eof`** pseudo-character when this occurs. In Java, for example, **`InputStream.read()`**, which reads a single byte, returns - 1 when end of file is reached. A constant, **`EOF`**, defined as - 1 can be treated as an "extended" ASCII character. This character then allows the definition of an **`Eof`** token that can be passed back to the parser.

An **`Eof`** token is useful because it allows the parser to verify that the logical end of a program corresponds to its physical end.

Most parsers *require* an end of file token.

Lex and Jlex automatically create an **`Eof`** token when the scanner they build tries to scan an **`EOF`** character (or tries to scan when **`eof()`** is true).

Multi Character Lookahead

We may allow finite automata to look beyond the next input character.

This feature is necessary to implement a scanner for FORTRAN.

In FORTRAN, the statement

```
 DO 10 J = 1,100
```

specifies a loop, with index **`J`** ranging from 1 to 100.
The statement

```
 DO 10 J = 1.100
```

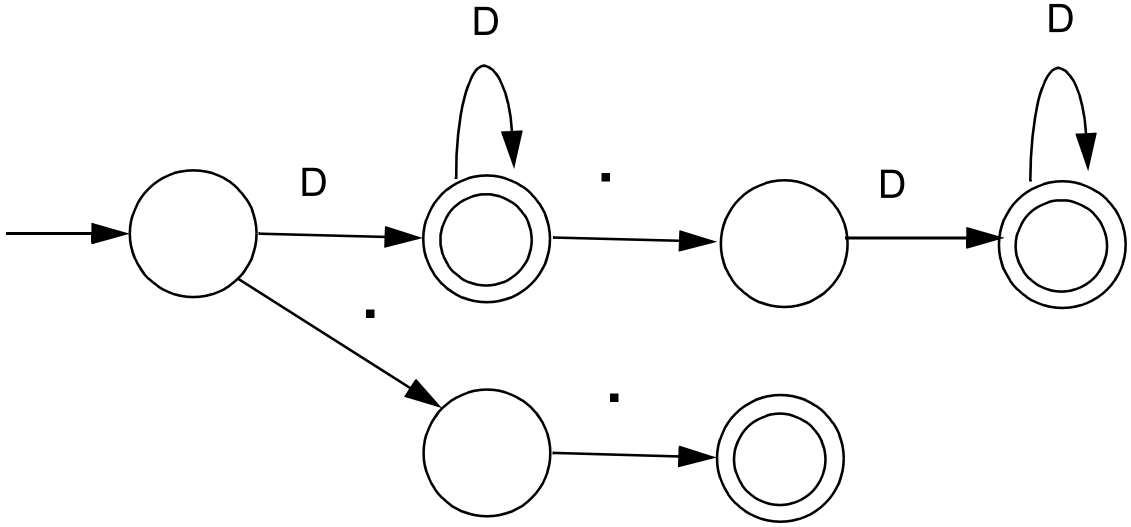
is an assignment to the variable **`DO10J`**. (Blanks are not significant except in strings.)

A FORTRAN scanner decides whether the **`O`** is the last character of a **`DO`** token only after reading as far as the comma (or period).

A milder form of extended lookahead problem occurs in Pascal and Ada.
The token **10.50** is a real literal, whereas **10..50** is three different tokens.

We need two- character lookahead after the **10** prefix to decide whether we are to return **10** (an integer literal) or **10.50** (a real literal).

Suppose we use the following FA.



Given **10..100** we scan three characters and stop in a non-accepting state.

Whenever we stop reading in a non- accepting state, we *back up* along accepted characters until an accepting state is found.

Characters we back up over are *rescanned* to form later tokens. If no accepting state is reached during backup, we have a lexical error.

Performance Considerations

Because scanners do so much character- level processing, they can be a real performance bottleneck in production compilers.

Speed is not a concern in our project, but let’s see why scanning speed can be a concern in production compilers.

Let’s assume we want to compile at a rate of 5000 lines/sec. (so that most programs compile in just a few seconds).

Assuming 30 characters/line (on average), we need to scan 150,000 char/sec.

A key to efficient scanning is to group character- level operations whenever possible. It is better to do one operation on n characters rather than n operations on single characters.

In our examples we’ve read input one character as a time. A subroutine call can cost hundreds or thousands of instructions to execute—far too much to spend on a single character.

We prefer routines that do block reads, putting an entire block of characters directly into a buffer.

Specialized scanner generators can produce particularly fast scanners.

The *GLA scanner generator* claims that the scanners it produces run as fast as:

```
 while(c != Eof) {
    c = getchar();
}
```

Lexical Error Recovery

A character sequence that can't be scanned into any valid token is a *lexical error*.

Lexical errors are uncommon, but they still must be handled by a scanner. We won't stop compilation because of so minor an error.

Approaches to lexical error handling include:

- Delete the characters read so far and restart scanning at the next unread character.
- Delete the first character read by the scanner and resume scanning at the character following it.

Both of these approaches are reasonable.

The first is easy to do. We just reset the scanner and begin scanning anew.

The second is a bit harder but also is a bit safer (less is immediately deleted). It can be implemented using scanner backup.

Usually, a lexical error is caused by the appearance of some illegal character, mostly at the beginning of a token.

(Why at the beginning?)

In these case, the two approaches are equivalent.

The effects of lexical error recovery might well create a later *syntax error*, handled by the parser.

Consider

```
...for$tnight...
```

The **`$`** terminates scanning of **`for`**. Since no valid token begins with **`$`**, it is deleted. Then **`tnight`** is scanned as an identifier. In effect we get

```
...for tnight...
```

which will cause a syntax error. Such “false errors” are unavoidable, though a syntactic error- repair may help.

Error Tokens

Certain lexical errors require special care. In particular, runaway strings and runaway comments ought to receive special error messages.

In Java strings may not cross line boundaries, so a runaway string is detected when an end of a line is read within the string body. Ordinary recovery rules are inappropriate for this error. In particular, deleting the first character (the double quote character) and restarting scanning is a *bad* decision.

It will almost certainly lead to a cascade of “false” errors as the string text is inappropriately scanned as ordinary input.

One way to handle runaway strings is to define an *error token*.

An error token is *not* a valid token; it is never returned to the parser. Rather, it is a *pattern* for an error condition that needs special handling. We can define an error token that represents a string terminated by an end of line rather than a double quote character.

For a valid string, in which internal double quotes and back slashes are escaped (and no other escaped characters are allowed), we can use

" (Not(" | Eol | \ ) | \" | \\ )* "

For a runaway string we use

" (Not(" | Eol | \ ) | \" | \\ )* Eol

(**Eol** is the end of line character.)

When a runaway string token is recognized, a special error message should be issued.

Further, the string may be “repaired” into a correct string by returning an ordinary string token with the closing Eol replaced by a double quote.

This repair may or may not be “correct.” If the closing double quote is truly missing, the repair will be good; if it is present on a succeeding line, a cascade of inappropriate lexical and syntactic errors will follow.

Still, we have told the programmer exactly what is wrong, and that is our primary goal.

In languages like C, C++, Java and CSX, which allow multiline comments, improperly terminated (runaway) comments present a similar problem.

A runaway comment is not detected until the scanner finds a close comment symbol (possibly belonging to some other comment) or until the end of file is reached. Clearly a special, detailed error message is required.

Let’s look at Pascal- style comments that begin with a `{` and end with a `}`. Comments that begin and end with a pair of characters, like `/*` and `*/` in Java, C and C++, are a bit trickier.

Correct Pascal comments are defined quite simply:

{ Not(})* }

To handle comments terminated by **Eof**, this error token can be used:

{ Not(})* Eof

We want to handle comments unexpectedly closed by a close comment belonging to another comment:

```
{... missing close comment
 ... { normal comment }...
```

We will issue a *warning* (this form of comment is lexically legal).

Any comment containing an open comment symbol in its body is most probably a missing **}** error.

We split our legal comment definition into two token definitions.

The definition that accepts an open comment in its body causes a warning message ("Possible unclosed comment") to be printed.

We now use:

{ Not({ | })* } and
{ (Not({ | })* { Not({ | })*)+ }

The first definition matches correct comments that do not contain an open comment in their body.

The second definition matches correct, but suspect, comments that contain at least one open comment in their body.

Single line comments, found in Java, CSX and C++, are terminated by Eol.

They can fall prey to a more subtle error—what if the last line has no Eol at its end?

The solution?

Another error token for single line comments:

// Not(Eol)*

This rule will only be used for comments that don’t end with an Eol, since scanners always match the longest rule possible.

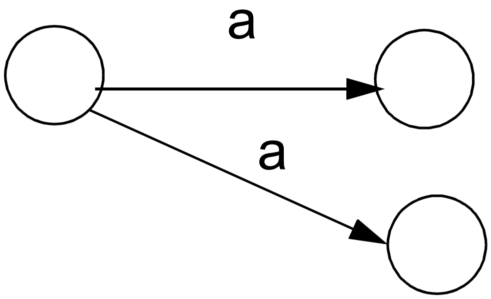
Regular Expressions and Finite Automata

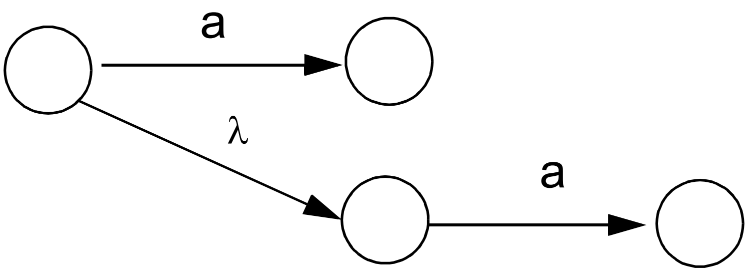
Regular expressions are fully equivalent to finite automata.

The main job of a scanner generator like JLex is to transform a regular expression definition into an equivalent finite automaton.

It first transforms a regular expression into a *nondeterministic finite automaton* (NFA).

Unlike ordinary deterministic finite automata, an NFA need not make a unique (deterministic) choice of a successor state to visit. As shown below, an NFA is allowed to have a state that has two transitions (arrows) coming out of it, labeled by the same symbol. An NFA may also have transitions labeled with λ.





Transitions are normally labeled with individual characters in Σ, and although λ is a string (the string with no characters in it), it is definitely *not* a character. In the above example, when the automaton is in the state at the left and the next input character is a, it may choose to use the transition labeled a *or* first follow the λ transition (you can always find λ wherever you look for it) and *then* follow an a transition. FAs that contain no λ transitions and that always have unique successor states for any symbol are *deterministic.*

Building Finite Automata From Regular Expressions

We make an FA from a regular expression in two steps:

- Transform the regular expression into an NFA.
- Transform the NFA into a deterministic FA.

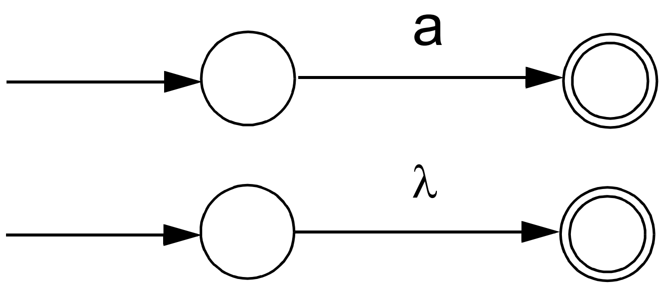
The first step is easy.

Regular expressions are all built out of the *atomic* regular expressions a (where a is a character in Σ) and λ by using the three operations

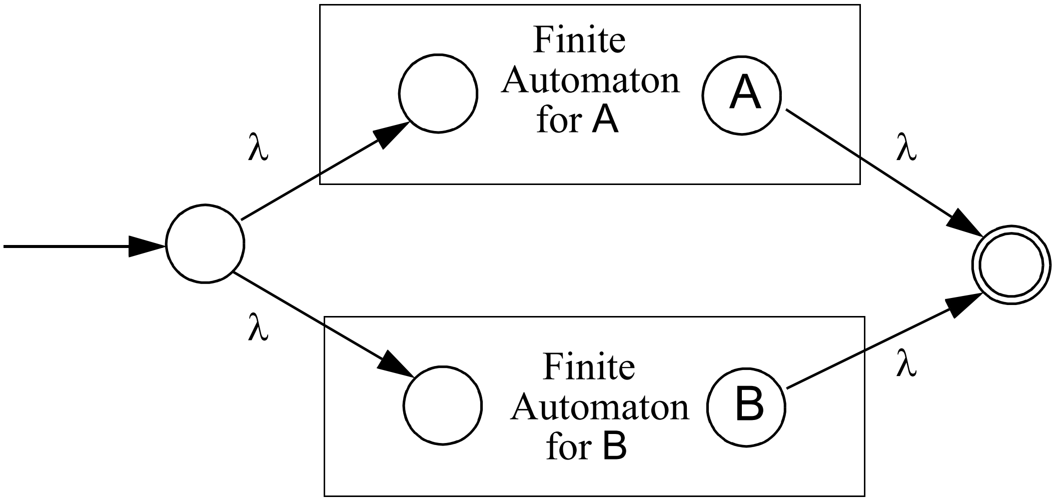
A B and A | B and A^*.

Other operations (like A^+) are just abbreviations for combinations of these.

NFAs for a and λ are trivial:



Suppose we have NFAs for A and B and want one for A | B. We construct the NFA shown below:

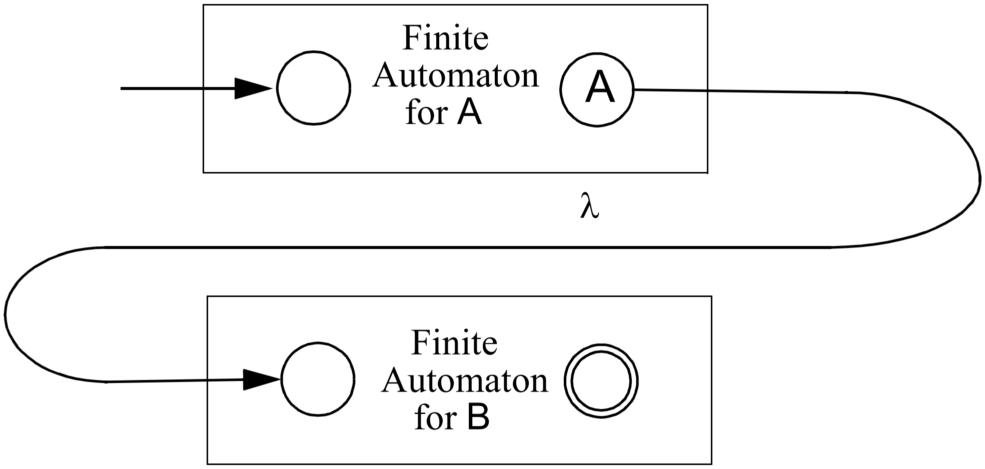


The states labeled A and B were the accepting states of the automata for A and B; we create a new accepting state for the combined automaton.

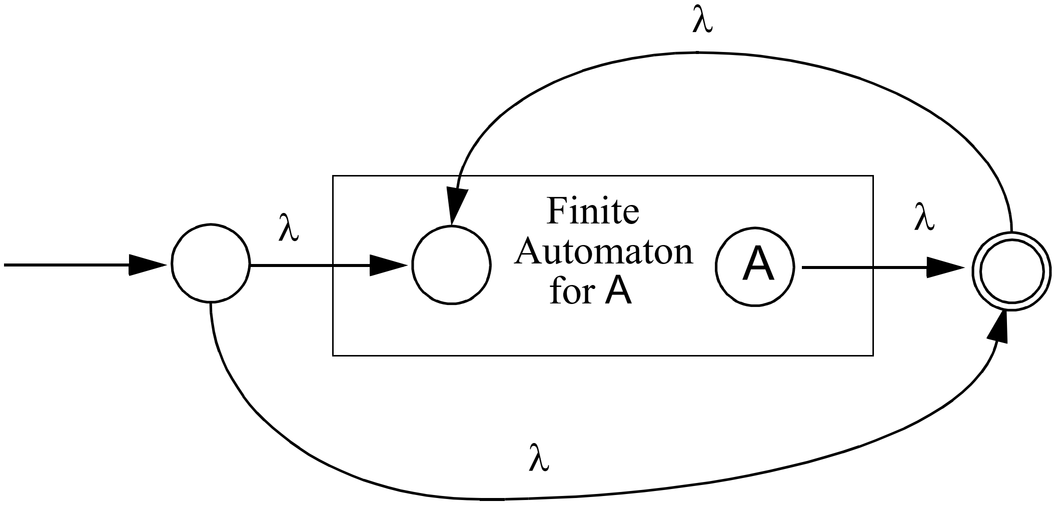
A path through the top automaton accepts strings in **A**, and a path through the bottom automation accepts strings in **B**, so the whole automaton matches **A | B**.

The construction for A B is even easier. The accepting state of the combined automaton is the same state that was the accepting state of B. We must follow a path through **A**’s automaton, then through **B**’s automaton, so overall **A B** is matched.

We could also just merge the accepting state of A with the initial state of B. We chose not to only because the picture would be more difficult to draw.



Finally, let's look at the NFA for A^*. The start state reaches an accepting state via λ, so λ is accepted. Alternatively, we can follow a path through the FA for A one or more times, so zero or more strings that belong to A are matched.



Creating Deterministic Automata

The transformation from an NFA N to an equivalent DFA D works by what is sometimes called the *subset construction.*

Each state of D corresponds to a set of states of N.

The idea is that D will be in state {x, y, z} after reading a given input string if and only if N could be in *any* one of the states *x*, *y*, or *z*, depending on the transitions it chooses. Thus D keeps track of *all* the possible routes N might take and runs them simultaneously.

Because N is a *finite* automaton, it has only a finite number of states. The number of subsets of N's states is also finite, which makes tracking various sets of states feasible.

An accepting state of D will be any set containing an accepting state of N, reflecting the convention that N accepts if there is *any* way it could get to its accepting state by choosing the “right” transitions.

The start state of D is the set of all states that N could be in without reading any input characters—that
is, the set of states reachable from the start state of N following only λ transitions. Algorithm **`close`** computes those states that can be reached following only λ transitions.

Once the start state of D is built, we begin to create successor states:

We take each state S of D, and each character c, and compute S’s successor under c.

S is identified with some set of N’s states, $\{n_1, n_2, \ldots\}$.

We find all the possible successor states to $\{n_1, n_2, \ldots\}$ under c, obtaining a set $\{m_1, m_2, \ldots\}$.

Finally, we compute
T = CLOSE($\{ m_1, m_2, \ldots\}$).

T becomes a state in D, and a transition from S to T labeled with c is added to D.

We continue adding states and transitions to D until all possible successors to existing states are added.

Because each state corresponds to a finite subset of N’s states, the process of adding new states to D must eventually terminate.

Here is the algorithm for λ-closure, called **close**. It starts with a set of NFA states, S, and adds to S all states reachable from S using only λ transitions.

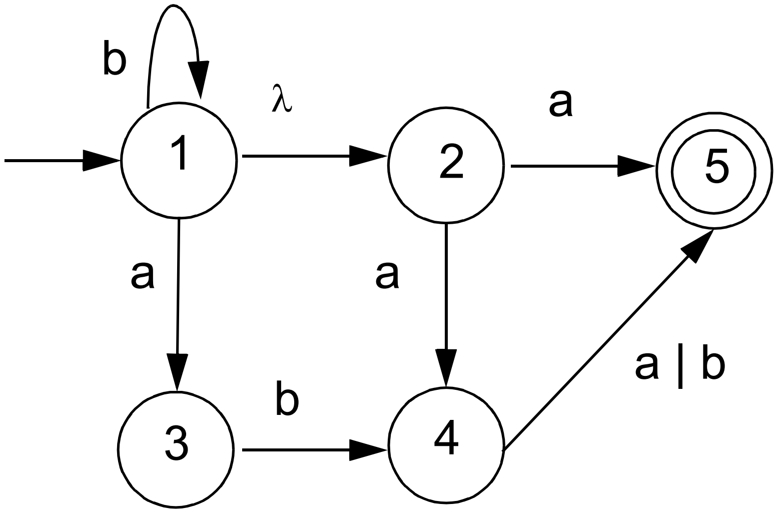
```
void close(NFASet S) {
 while (x in S and $x \xrightarrow{\lambda} y$
       and y notin S) {
          S = S ∪ {y}
}}
```

Using **close**, we can define the construction of a DFA, D, from an NFA, N:

```
DFA  MakeDeterministic(NFA N) {
 DFA  D ; NFASet  T
 D.StartState = { N.StartState }
 close(D.StartState)
 D.States = { D.StartState }
 while (states or transitions can be
        added to D) {
   Choose any state S in D.States
     and any character c in Alphabet
   T = {y in N.States such that
        $x \xrightarrow{c} y$ for some x in S}
   close(T);
   if (T notin D.States) {
        D.States = D.States $\cup$ {T}
        D.Transitions =
           D.Transitions $\cup$
            {the transition $S \xrightarrow{c} T$}
 } }
 D.AcceptingStates =
  { S in D.States such that an
      accepting state of N in S}
}
```

Example

To see how the subset construction operates, consider the following NFA:



We start with state 1, the start state of N, and add state 2 its λ-successor.

D’s start state is {1,2}.

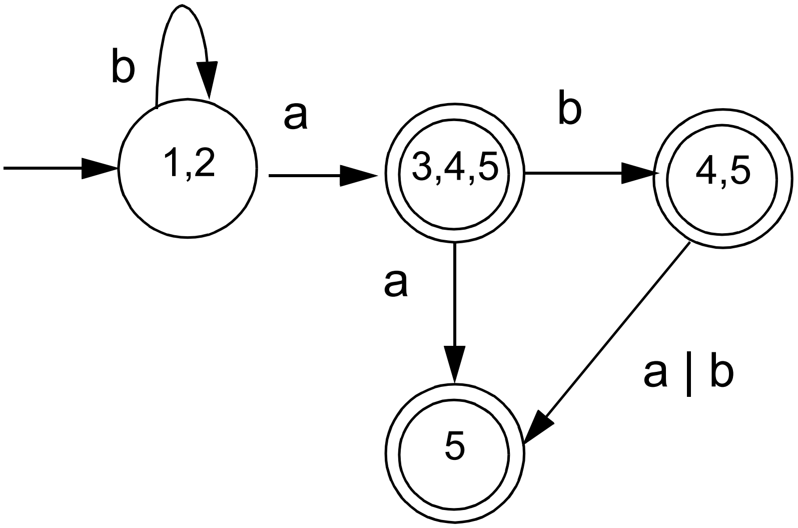
Under a, {1,2}’s successor is {3,4,5}.

State 1 has itself as a successor under b. When state 1's λ-successor, 2, is included, {1,2}'s successor is {1,2}. {3,4,5}'s successors under a and b are {5} and {4,5}.

{4,5}'s successor under b is {5}.

Accepting states of D are those state sets that contain N's accepting state which is 5.

The resulting DFA is:



It is not too difficult to establish that the DFA constructed by `MakeDeterministic` is equivalent to the original NFA.

The idea is that each path to an accepting state in the original NFA has a corresponding path in the DFA. Similarly, all paths through the constructed DFA correspond to paths in the original NFA.

What is less obvious is the fact that the DFA that is built can sometimes be *much larger* than the original NFA. States of the DFA are identified with *sets* of NFA states.

If the NFA has n states, there are 2^n distinct sets of NFA states, and hence the DFA may have as many as 2^n states. Certain NFAs actually exhibit this exponential blowup in size when made deterministic.

Fortunately, the NFAs built from the kind of regular expressions used to specify programming language tokens do not exhibit this problem when they are made deterministic.

As a rule, DFAs used for scanning are simple and compact.

If creating a DFA is impractical (because of size or speed- of-generation concerns), we can scan using an NFA. Each possible path through an NFA is tracked, and reachable accepting states are identified. Scanning is slower using this approach, so it is used only when construction of a DFA is not practical.

Optimizing Finite Automata

We can improve the DFA created by `MakeDeterministic`.

Sometimes a DFA will have more states than necessary. For every DFA there is a unique *smallest* equivalent DFA (fewest states possible).

Some DFA's contain *unreachable states* that cannot be reached from the start state.

Other DFA's may contain *dead states* that cannot reach any accepting state.

It is clear that neither unreachable states nor dead states can participate in scanning any valid token. We therefore eliminate all such states as part of our optimization process.

We optimize a DFA by *merging together* states we know to be equivalent.

For example, two accepting states that have no transitions at all out of them are equivalent.

Why? Because they behave exactly the same way—they accept the string read so far, but will accept no additional characters.

If two states, s_1 and s_2, are equivalent, then all transitions to s_2 can be replaced with transitions to s_1. In effect, the two states are merged together into one common state.

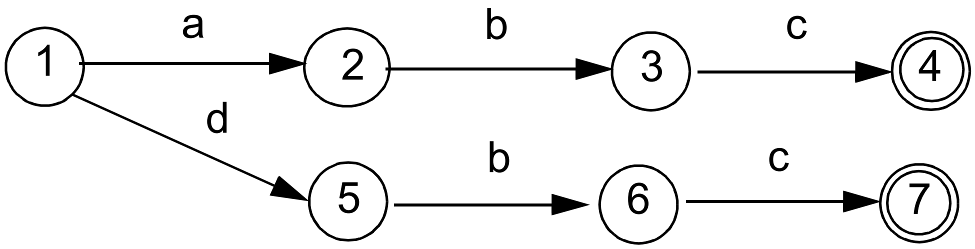
How do we decide what states to merge together?

We take a *greedy* approach and try the most optimistic merger of states. By definition, accepting and non- accepting states are distinct, so we initially try to create only two states: one representing the merger of all accepting states and the other representing the merger of all non- accepting states.

This merger into only two states is almost certainly too optimistic. In particular, all the constituents of a merged state must agree on the same transition for each possible character. That is, for character c all the merged states must have no successor under c or they must all go to a single (possibly merged) state.

If all constituents of a merged state do not agree on the transition to follow for some character, the merged state is *split* into two or more smaller states that do agree.

As an example, assume we start with the following automaton:



Initially we have a merged non-accepting state {1,2,3,5,6} and a merged accepting state {4,7}.

A merger is legal if and only if all constituent states agree on the same successor state for all characters. For example, states 3 and 6 would go to an accepting state given character c; states 1, 2, 5 would not, so a split must occur.

We will add an error state s_E to the original DFA that is the successor state under any illegal character. (Thus reaching s_E becomes equivalent to detecting an illegal token.) s_E is not a real state; rather it allows us to assume every state has a successor under every character. s_E is never merged with any real state.

Algorithm **`Split`**, shown below, splits merged states whose constituents do not agree on a common successor state for all characters. When **`Split`** terminates, we know that the states that remain merged are equivalent in that they always agree on common successors.

```
Split(FASet StateSet) {
 repeat
  for(each merged state S in StateSet) {
    Let S correspond to {$s_1$,…,$s_n$}
    for(each char c in Alphabet){
      Let $t_1$,…,$t_n$ be the successor
       states to $s_1$,…,$s_n$ under c
      if($t_1$,…,$t_n$ do not all belong to
           the same merged state){
            Split S into two or more new
            states such that $s_i$ and $s_j$
            remain in the same merged
            state if and only if $t_i$ and $t_j$
            are in the same merged state}
      }
 until no more splits are possible
}
```

Returning to our example, we initially have states {1,2,3,5,6} and {4,7}. Invoking **`Split`**, we first observe that states 3 and 6 have a common successor under c, and states 1, 2, and 5 have no successor under c (equivalently, have the error state s_E as a successor).

This forces a split, yielding {1,2,5}, {3,6} and {4,7}.

Now, for character b, states 2 and 5 would go to the merged state {3,6}, but state 1 would not, so another split occurs.

We now have: {1}, {2,5}, {3,6} and {4,7}.

At this point we are done, as all constituents of merged states agree on the same successor for each input symbol.

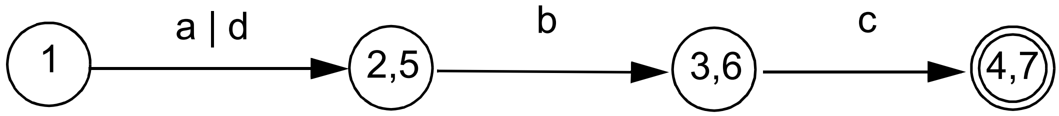
Once **Split** is executed, we are essentially done.

Transitions between merged states are the same as the transitions between states in the original DFA.

Thus, if there was a transition between state s_i and s_j under character c, there is now a transition under c from the merged state containing s_i to the merged state containing s_j. The start state is that merged state containing the original start state.

Accepting states are those merged states containing accepting states (recall that accepting and non- accepting states are never merged).

Returning to our example, the minimum state automaton we obtain is



Properties of Regular Expressions and Finite Automata

- Some token patterns *can't* be defined as regular expressions or finite automata. Consider the set of balanced brackets of the form [[[…]]]. This set is defined formally as

  $\{ [^m]^m \mid m \geq 1 \}$.

  This set is *not* regular.

  No finite automaton that recognizes *exactly* this set can exist.

  Why? Consider the inputs [, [[, [[[, ...

  For two different counts (call them i and j) $[^i$ and $[^j$ must reach the same state of a given FA! (Why?)

  Once that happens, we know that if $[^i]^i$ is accepted (as it should be), the $[^j]^i$ will also be accepted (and that should not happen).

- $\overline{R}$ = V* - R is regular if R is. Why?
  Build a finite automaton for R. Be careful to include transitions to an "error state" s_E for illegal characters.
  Now invert final and non- final states. What was previously accepted is now rejected, and what was rejected is now accepted. That is, $\overline{R}$ is accepted by the modified automaton.

- Not all subsets of a regular set are themselves regular. The regular expression $[^+]^+$ has a subset that isn't regular. (What is that subset?)

- Let R be a set of strings. Define R^{rev} as all strings in R, in reversed (backward) character order.
  Thus if R = {abc, def}

  then R^{rev} = {cba, fed}.

  If R is regular, then R^{rev} is too.
  Why? Build a finite automaton for R. Make sure the automaton has only one final state. Now *reverse* the direction of all transitions, and interchange the start and final states. What does the modified automation accept?

- If R_1 and R_2 are both regular, then $R_1 \cap R_2$ is also regular. We can show this two different ways:
  1. Build two finite automata, one for R1 and one for R2. Pair together states of the two automata to match R1 and R2 simultaneously. The paired-state automaton accepts only if both R1 and R2 would, so R1 $\cap$ R2 is matched.
  2. We can use the fact that R1 $\cap$ R2 is = $\overline{\overline{R_1} \cup \overline{R_2}}$ We already know union and complementation are regular.