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.

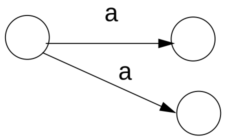
Regular Expressions and Finite Automata

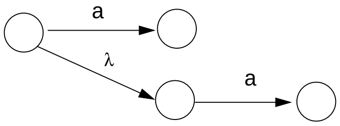
Regular expressions are fully equivalent to finite automata.

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

First it transforms a regular expression into a *nondeterministic finite automaton* (NFA).

Unlike an ordinary deterministic finite automaton, an NFA need not make a unique (deterministic) choice of a successor state to visit. For example, 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:

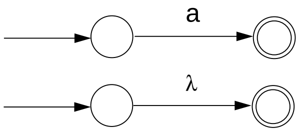
- Transform the regular expression into an NFA.
- Transform the NFA into a deterministic FA.

The first step is easy.

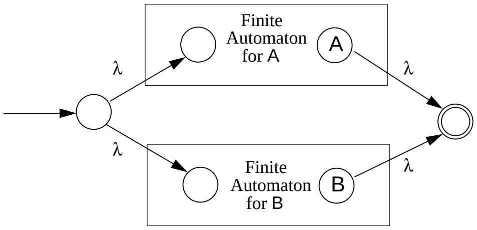
Regular expressions are all built out of the *atomic* regular expressions **a** (where **a** is a character in Σ) and λ by using the three operations **A B** and **A | B** and $\mathbf{A}^*$.

Other operations (like $\mathbf{A}^+$) 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:

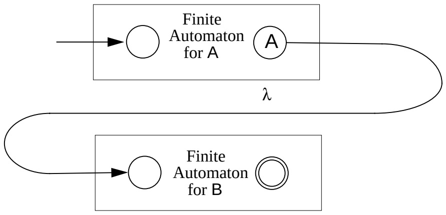


The states labeled **A** and **B** were the accepting states of the automata for **A** and **B**; we create a new accepting state for the combined automaton.

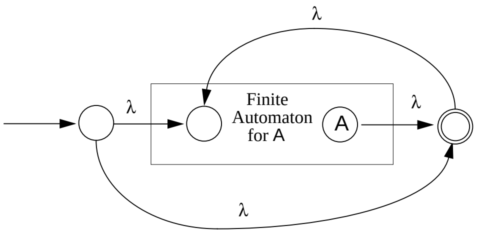
A path through the top automaton accepts strings in **A**, and a path through the bottom automation accepts strings in **B**, so the whole automaton matches **A | B**.

As shown below, 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 $\mathbf{A}^*$. 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, $\{n_1, n_2,...\}$.

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

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

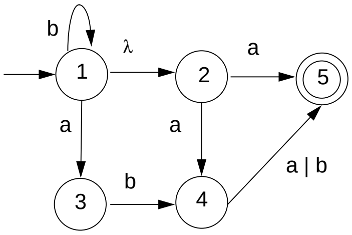
```
void close(NFASet S) {
 while (x in S and $x \xrightarrow{\lambda} y$
       and y notin S) {
        S = S ∪ {y}
}}
```

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 ∪ {T}
        D.Transitions =
           D.Transitions ∪
            {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}.

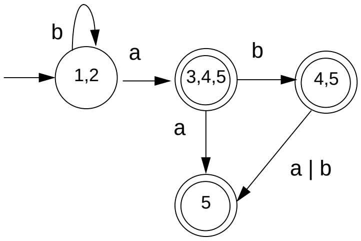
State 1 has itself as a successor under b. When state 1's λ-successor, 2, is included, {1,2}'s successor is {1,2}.

{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:


b
a
b
1,2
3,4,5
4,5
a
a | b
5