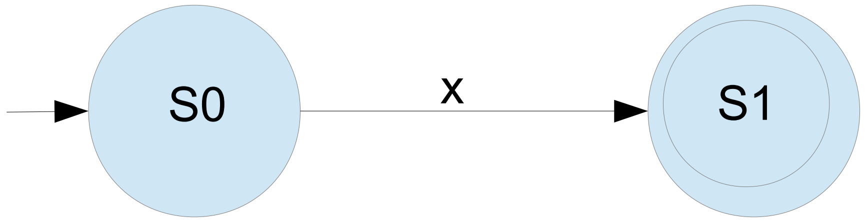
# Thompson's construction

- Thompson's construction provides a rule for NFAs based on a single letter regular expression, e.g. one for a regex that was simply “A”, another NFA for the regex “B”, etc
- It provides a construction rule for each of the core RE operations: concatenation (e.g. AB), kleene star (e.g. A*), alternation/or (e.g. A|B)
- It specifies a precedence order for those operations
- Successive applications of the constructions (following the precedence rules) let us build an NFA matching any regex

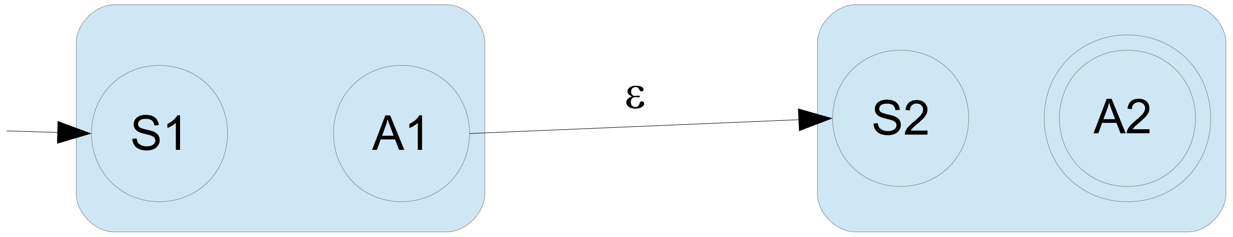
## NFA for single letter, e.g. x

- Each machine will have a single start and single accept state, simplest is for a single letter, assume any transition not shown goes to a reject state



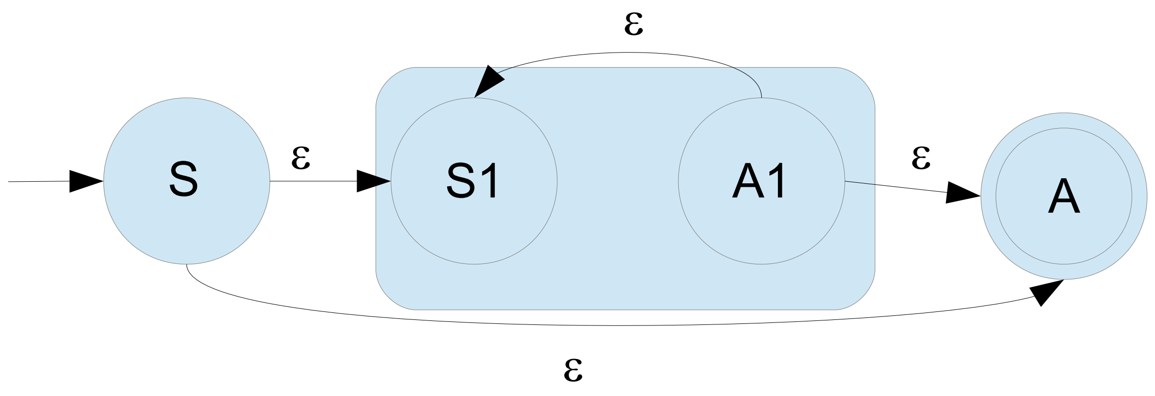
# NFA for concatenate, e.g. xy

- Given machines M1, M2 (just showing their start/accept states), concatenation adds a single null transition from M1 to M2, M1's start is the new machines start, M2's accept is the new machine's accept



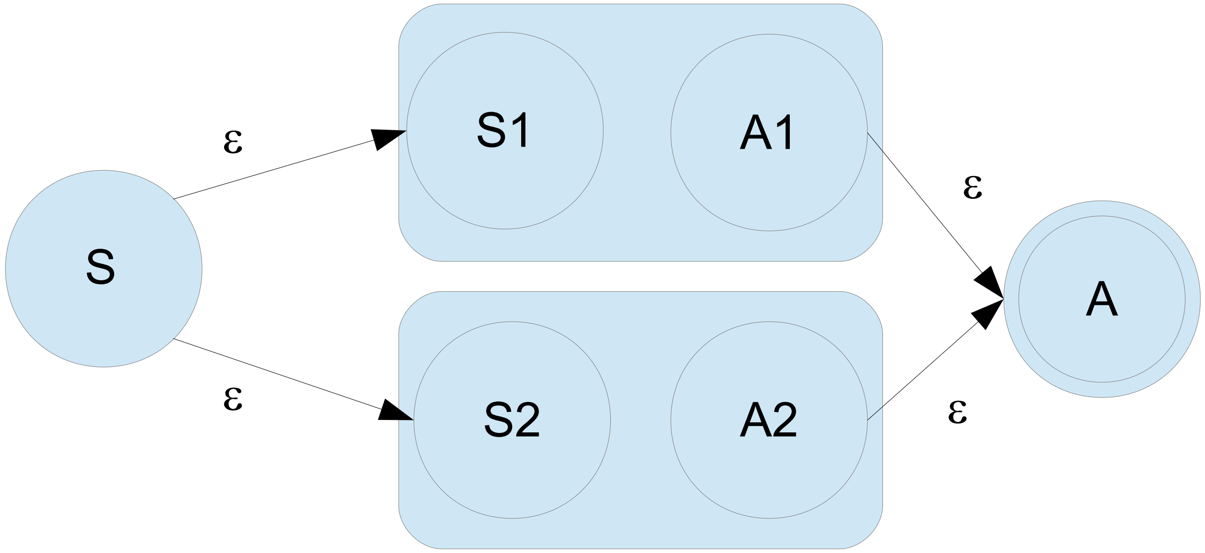
## NFA for kleene star, **e.g. x***

- Given machine M1, adds new start/accept states around M1, and four new null transitions



## NFA for or, e.g x|y

- Given machines M1 and M2, to get M1 | M2 we need new start and accept states and four new null transitions



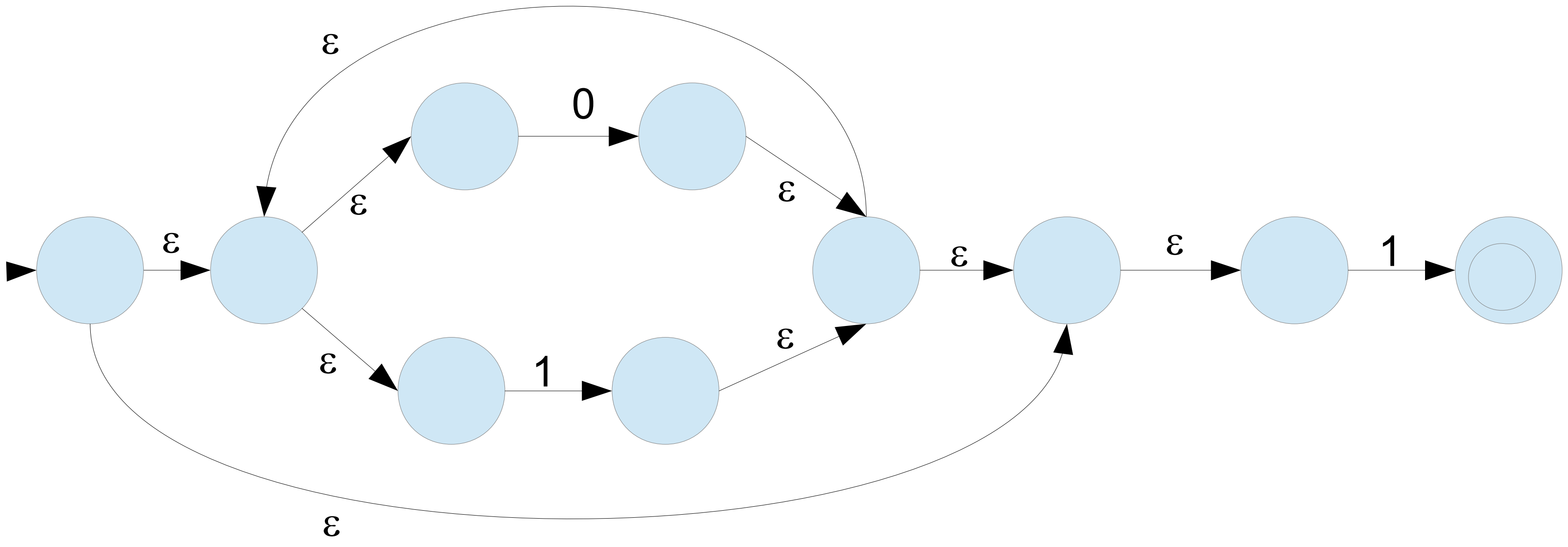
## Operation precedence

- Brackets have highest priority, then *, then concatenation, and finally |
- Note that other regex operations can be represented as combinations of the ones given, e.g. A+ would be represented as AA*

ε

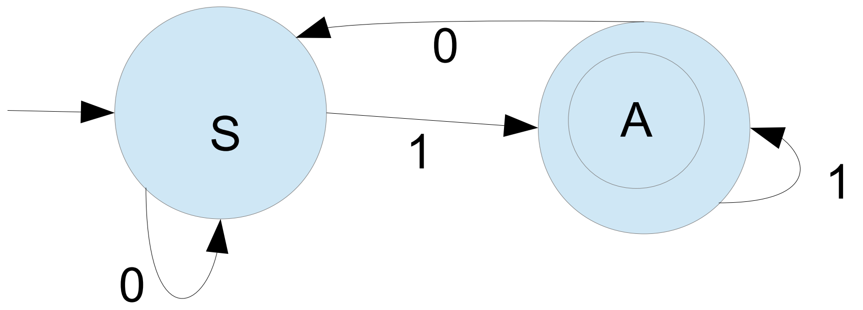
## Example: (0|1)*1

- First we do the 0|1 since it's in parenthesis, then the *, then the concatenation, giving result below



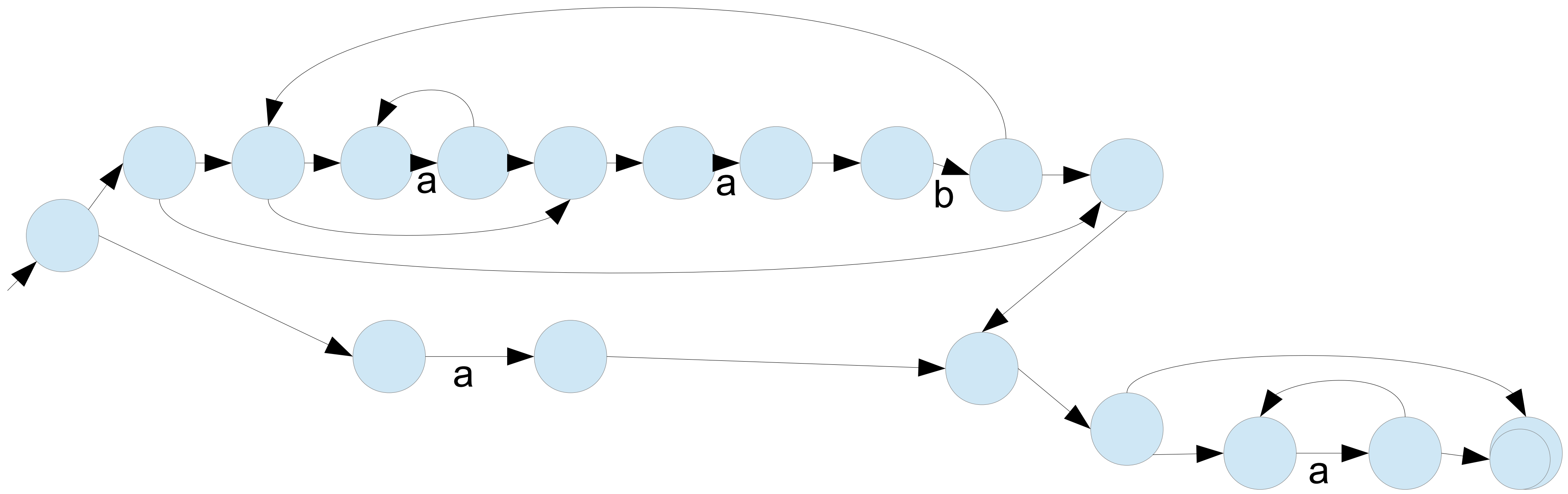
# Lots of extra states

- The construction technique generates many more states than if we came up with an ideal “by hand” version
- e.g. For the (0|1)*1 example all we need are two states as shown below
- This is why our technique goes through a final minimization stage later



# Example: (a|(a*ab)*)a*

- All transitions are nulls except the labelled a/b one



## Creating a language recognizer

- Of course, we want to be able to recognize ALL the tokens in a language, so we want an NFA that combines them all
- If we're given RE r1 for token type 1, r2 for token type 2, etc, then a single regex to cover all the token types is r1 | r2 | r3 .... | rk
- From that regex, we can apply Thompson's construction to get an NFA for the full token set, but it assumes we're consuming the whole string ... that's not enough for cases where we want to handle a string that is a sequence of tokens

## Possible fixes for tokenizing

- require the tokens to be fully delimited – e.g. require whitespace before/after every single token (usually seen as overly restrictive/limiting)
- If the recognizer finished in a non-accepting state, have it back up to a previous accept state that it 'passed through', accept that as the token match then start looking for the next token from that point
- Note that if we want to identify token types as well as recognize them then accept states need to be labelled (and treated as distinct) by the associated token type