From regex to minmized DFA

- Given a set of regular expressions to describe the language tokens, how can we automatically generate a tokenizer:

1. Use Thompson's construction to derive a nondeterministic finite automata from the regular expression (NFAs to be discussed shortly)
2. Use subset construction to derive a deterministic finite automata from the NFA
3. Use Hopcroft's algorithm to minimize the DFA (later we'll consider how to generate actual code based on the DFA)

...We just need to discuss NFA/DFA, Thompson's construction, subset construction, and Hopcroft's algorithm!

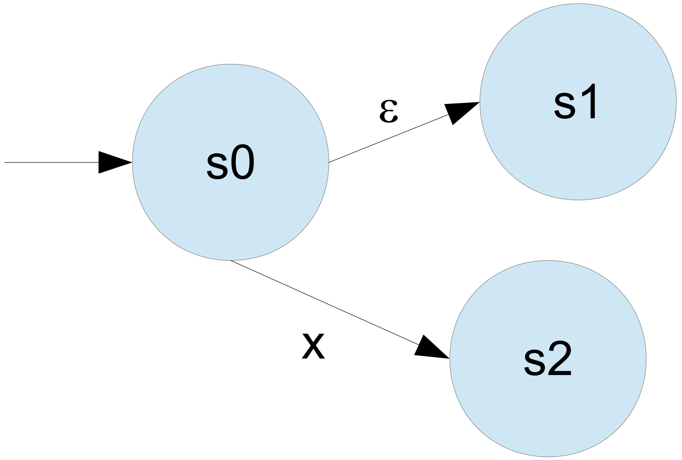
DFA and NFA

- Our earlier FSMs used a deterministic approach: each transition based on current state + next character of input
- Non-deterministic machines allow certain transitions to happen *without* reading a character: these transitions may/may not be used when processing input, and an NFA accepts a string if it is *possible* to end in an accept state
- The two types have equivalent power (proofs in 320)
- We write code based on the deterministic ones, but it's easier to produce NFAs from regular expressions

NFA with null transitions

- Null transitions don't involve reading an input char, e.g. in diagram below if current state is s0 and next char is an x we could go to either s1 (without consuming x) or s2 (consuming x)

ε represents null transition



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

Subset construction

- Given an NFA, we want to build a matching DFA
- Each state in the DFA will actually correspond to a subset of states in the NFA, exactly how many states the DFA has (and which subsets of NFA states each one matches) are determined during construction
- NFA states will be grouped into subsets that combine all the states that can be reached by a particular string *together with possible combinations of null transitions*
- The DFA will be deterministic, identifying its possible states based on string patterns leading to them

Hopcroft's algorithm

- We want to identify and merge any equivalent states in the DFA: states produce identical behaviour across all inputs
- It takes the set of all states, and repeatedly partitions them into smaller and smaller subsets by identifying ways in which some states in a subset behave differently than others in the subset
- First: split the set of all states into accept/non-accept
- Repeat: pick an input character, check if each state in a partition takes you to a common partition (e.g. on input x, do all states in partition P transition to states in partition Q)

What do you suppose lex does?

- if you look in lex.yy.c, you'll find a variety of state tables
- you'll see (buried near a “while (/*CONSTCOND*/1)”) lines matching your .lex token handling code
- you'll see a FSM with a while loop cycling through the input and a set of states
- you'll also find routines to take care of reading the source, setting up yylval, yytext, etc