Non-deterministic Finite Automata (NFA)

- CAN have transitions on the same input to different states
- Can include a $\varepsilon$ or $\lambda$ transition (i.e. move to new state without reading input)
- Often easier to design than equivalent DFA, but more complex to evaluate input
- Can always create a DFA from an NFA (and vice versa)
  - Algorithms exist to handle the conversion
NFA Picture

Bit harder to evaluate if any given input string is part of the language, for example input 00 is accepted since it can end in an accepting state, but depending on the path you follow it could end in any state.
Regular Expression

- Sequence of regular and meta characters that describes a regular language.
  - Alphabet: \( \Sigma \) (Eg. \{a-z\}, \{0,1\}, \{a-z0-9\}, etc.)
  - empty set: \( \emptyset \)
  - empty string: \( \varepsilon \)
  - literal character: a in \( \Sigma \)
Operations

- Alternation
  \( R \mid S = R \cup S \) (union of \( R \) and \( S \))

- Concatenation
  \( RS = \{ \alpha\beta : \alpha \text{ in } R \text{ and } \beta \text{ in } S \} \)

- Kleene star
  \( R^* = \) smallest superset of \( R \) containing \( \varepsilon \) and closed under concatenation
Examples

- $a^* = \{ \epsilon, a, aa, aaa, \ldots \}$

- $b|a^* = \{ b, \epsilon, a, aa, aaa, \ldots \}$

- $(0|1)^* = \text{finite binary, plus empty string}$

- $(h|c)at = \{ \text{hat, cat} \}$

- $(a|b)(c|d) = \{ ac, ad, bc, bd \}$

- $\text{while} = \{ \text{while} \}$
Syntax / Extensions

- square brackets (with ranges)
  - match one of the given letters
  - \([a-z]\) matches all lowercase characters
- plus sign: like star but excluding \(\epsilon\)
  - \([0-9]+\) matches integer numbers
- \([\^]\) - Matches a single symbol NOT inside the brackets.
  - \([\^0-9]\) matches any non numeric character
- dot matches any single letter
  - \.at matches hat, cat, fat, mat, bat, 7at, Aat, etc.
Example

Q: What is the regex to search for all occurrences of the following name in a text with different possible spellings

Georg Friedrich Händel:

- Händel
- Haendel
- Handel
- Hendel

"[Hh](ae|a|e|ä)ndel"
Syntax / Extensions

- \? - matches the previous symbol or pattern 0 or 1 times.
- escape character `\`
  - To use syntactical symbols as constants
  - Eg. `[0-9]+\.[0-9]+` matches positive fractional numbers
- Q: Write a regex that matches real numbers in the decimal number format (Eg. -81.998) or in the scientific notation, so a decimal number followed by E, then a natural number (-5.9E-2)
Cases: x.y, x.yEz, x.yE-z, -x.y, -x.yEz, x.yE-z

-?[0-9]+\.[0-9]+(E-?[0-9]+)?
Scanner: Resolving Conflicts

• Splitting the input stream: How to tokenize “if17”? 
  • One token, variable name “if17” 
  • Variable name “if1” followed by number 7” 
  • Keyword “if” followed by number 17 
  • keyword “if” followed by number 1 then 7 
  • Var name i followed by var name “f17” etc. 

• To solve conflicting rules, the rule that is written first takes priority. (Eg. is “if” a variable or keyword) 

• To solve problems where an input could be broken up multiple times by the same rule we take the longest match (17 is 17 not 1 and 7)
Scanner

- If the input does not match any of our rules then it is not part of our language, in the context of compiling:
  - The code will not compile
  - The user is alerted with a compilation error message.
- After the scanner's execution we are left with a sequence of valid tokens. Can be thought of as having a sequence of correctly spelled English words.
Parsing

- After scanning, step 2 of compilation is Parsing.
- In Parsing, we analyze the sequence of tokens provided by the scanner, and verify that they follow the rules of our language (In English does the series of correctly spelled words follow the rules of grammar?)
- So we need some way to state the rules of our grammar. We need a way to specify what our language is!
Language Specification

- Simple for humans
- Unlike human language, must be unambiguous – not dependent on context.
- Easy to build parsing tools
- We want a formal language defined by what we call a **Context Free Grammar (CFG)**
Example – Simple Sentence

(R1) <sentence>    → <subj phrase> <verb phrase>
(R2) <subj phrase> → <noun phrase>
(R3) <verb phrase> → <verb> <noun phrase>
(R4) <noun phrase> → <article> <noun>
(R5) <verb>        → has
(R6) <article>     → a
(R7) <article>     → the
(R8) <noun>        → man
(R9) <noun>        → dog
CFG Specification Components

- Terminal Symbols (Tokens): actual symbols that appear in our language (word)
- Non-terminal Symbols: Abstract components of our language. Think of these as variable names
  - These do not literally appear in input
  - One nonterminal is chosen as our language’s “Start symbol”. All valid sentences of our language start with it
  - We donate non-terminals with <…>
- Production Rule: Rules of expansion of a non-terminal symbols into zero or more terminals and non-terminals
  - More than one rule per non-terminal possible”
    
    \[<NT> \rightarrow <Expansion1> | <Expansion2>\]
Derivation

Deriving Input strings that are part of the language defined by our context free grammar is done by:

- application of rules to generate valid input string
  - beginning with start symbol
- repeatedly replace one variable by one rule
- continue until there are no more variables
- resulting sequence of terminals is a syntactically correct input string
- formal definition of language: collection of all valid sequences that can be derived from start symbol