**The Compiler – Formal Languages**

* High-level language → compiler → assembly language
* Assembly language
  + Simple structure
  + Easy to parse
  + Straight-forward, unambiguous translation to machine language
* High-level language
  + Complex structure
  + Harder to recognize/parse
  + Usually no single translation to machine language
  + A formal theory of string recognition is needed to handle the complexity
* **Compiler**
  + Scanning – create a token sequence
  + Syntax analysis – create a parse tree
  + Semantic analysis – create a symbol table & type checking
  + Code generation – similar to assembler but more complicated
  + Goal – to find increasingly more sophisticated errors in a program
    - No compiler can find all errors
  + Chomsky hierarch
    - Finite languages
    - → regular languages
      * Can be recognized with a finite amount of memory; uses lexical analysis
    - → context-free languages
      * Recognized with 1 stack; checks syntax
    - → context-sensitive languages
      * Recognized with 2 stacks; checks semantics
  + Scanning:
    - Aka. Lexical analysis – convert code into a stream of token (type, value) pairs
    - Keywords – easy to recognize, no ambiguity, only a fixed number of them
    - Delimiters & operators – easy to recognize, some ambiguity, a fixed number
    - Constants & names – harder to recognize
      * Variable length, ambiguity exists (e.g. different types)
      * An infinite number of them exist
* **Alphabet** – a finite set of symbols, e.g. ∑ = {a, b, c}
* **String/word** – a finite sequence of symbols (from ∑), e.g. a, ab, acb etc.
  + Length of word – # of chars in the word, e.g. |aba| = 3
  + ε = empty word – an empty sequence of symbols; i.e.|ε| = 0
* **Language** – set of words
  + E.g. {a2nb, n ≥ 0} = words made up of an even # of a’s followed by 1 b
  + {} or ∅ = empty language – contains no words
  + {ε} = singleton language (language with one word) – only contains ε
  + How to recognize if a given string belongs to a given language?
    - Depends on how complex the language is – may be impossible
    - We can characterize languages based on the difficulty of their recognition process
* **Finite language**
  + - Has finitely many words
    - Can recognize a word by comparing with each word in the language set
    - E.g. L = {cat, car, cow}

If first char = c, move on, else error

If next char = a

If next char = t

If no more chars, accept, else error

Else if next char = r

… etc.

* + As a state machine (deterministic finite automata – DFA):
    - Start
    - → seen c
    - → seen ca, → seen co
    - → seen cat^, → seen car^, → seen cow^
  + ^ means accept if the program stops here
* **Regular language**
  + Built from the union, concatenation, and repetition of finite languages
  + Union – L1 ∪ L2 = {x | x ∈ L1 or x ∈ L2}
  + Concatenation – L1 ⋅ L2 = {xy | x ∈ L1, y ∈ L2}
  + Repetition – L\* = {ε} ∪ {xy | x ∈ L\*, y ∈ L}
    - = {ε} ∪ L ∪ LL ∪ LLL …
    - = 0 or more occurrences of words in L
    - |L\*| = ∞ i.e. repetition of a language has infinite words
  + E.g. {a2nb, n ≥ 0} = ({aa})\* ⋅ {b}
  + Regular expression vs. set-theoretic notation
    - ∅ = {}
    - ε = {ε}
    - aaa = {aaa}
    - E1 | E2 = E1 ∪ E2
    - E1E2 = E1 ⋅ E2
    - E\* = E\*
  + Is C regular?
    - A C program is a sequence of tokens, each of which is derived from a regular language
    - C ⊆ {valid C tokens}\*
    - Therefore maybe
  + How to recognize membership in a regular language?
  + E.g. {a2nb, n ≥ 0}
  + As a DFA:
    - Start
    - → seen 2n a’s ←→ seen 2n+1 a’s (loop)
    - → seen b^
  + E.g. names in C++
    - Regular expression: [a-zA-Z\_][a-zA-Z\_0-9]\*