**Context-Free Languages**

* What features of C or Scheme programs cannot be verified by a DFA?
  + E.g. ∑ = {(, )}
    - L = {w ∈ L\* | w is a string of balanced parens}
    - E.g. ε ∈ L, () ∈ L, (()()) ∈ L etc.
    - Each new state allows one more level of nesting, but no finite # of states allows all levels of nesting – cannot be modelled by DFA
  + E.g. ∑ = {a, b}
    - L = {w: # of a’s in w = # of b’s in w}
    - Impossible recognize arbitrary #’s of a’s and b’s with DFA
* **Context-free languages**
  + Syntax – structure of code
    - Valid ways to combine valid tokens to form C++ statements
  + Semantics – meaning of code
    - What the program does
    - Two programs written in different languages can have the same semantics but different syntax
  + Context-free languages – languages that can be described by a context-free grammar
  + E.g. balanced parens problem:
    - S → ε a word in the language is either empty,
    - S → (S) surrounded by parens, or
    - S → SS the concatenation of 2 words in the language
    - Shorthand: S → ε | (S) | SS
  + E.g. S ⇒ SS ⇒ (S)S ⇒ (S)(S) ⇒ ((S))(S) ⇒ (())(S) ⇒ (())()
    - ⇒ means “derives” – second string can be obtained from first string by one application of a grammar rule
  + A context-free grammar consists of:
    - An alphabet ∑ of terminal symbols
      * Symbols that appear in the output
    - A finite, non-empty set N of non-terminal symbols
      * Abstract components that do not appear in the output
      * An element S ∈ N – start symbol
    - N ∩ ∑ = ∅ (they have no intersection)
    - V = N ∪ ∑ (“vocabulary”)
    - P = finite set of productions
      * Production has the form A → B, A ∈ N, B ∈ V\*
      * Allows a non-terminal to expand into a repetition of terminals/non-terminals
  + Conventions:
    - a, b, c … − elements of ∑ (characters)
    - w, x, y … − elements of ∑\* (words)
    - A, B, C … S … − elements of N (non-terminals)
    - S – start symbol
    - α, β … − elements of V\*
  + A CFG can use recursion instead of repetition – more powerful
  + Derivation = a sequence of rewriting steps from S until there are no more non-terminals
    - G derives the word w ∈ ∑\* if S ⇒\* w, where w is a concatenation of terminals
      * ⇒\* means there is a finite sequence of productions S ⇒ A ⇒ B … ⇒ w
    - “Directly derives” – takes one derivation step
  + To derive a string:
    - Begin with S, replace single non-terminals using single rules, repeat until there are only terminals left
  + Leftmost derivation – expand leftmost non-terminal first
  + Rightmost derivation – expand rightmost non-terminal first
  + Ex:
    - N = {S, D}
    - ∑ = {a, b, c}
    - P =
      * S → a S b S → D
      * D → c D D → ε
    - E.g. S ⇒ a S b ⇒ aa S bb ⇒ aaa S bbb ⇒ aaa D bbb ⇒ aaabbb
    - E.g. S ⇒ D ⇒ c D ⇒ cc D ⇒ ccc D ⇒ ccc
  + Ex:
    - N = {E, B, D}
    - ∑ = {0, 1}
    - S = E
    - P =
      * E → E + E B → D
      * E → E – E D → 1
      * E → B D → D0
      * B → 0 D → D1
    - E.g. E ⇒ E + E ⇒ B + E ⇒ D + E
      * ⇒ D0 + E ⇒ 10 + E ⇒ 10 + B
      * ⇒ 10 + D ⇒ 10 + 1
  + A context-free language is defined as the collection of all valid strings that can be derived from the start symbol
    - i.e. L(G) = {w ∈ ∑\* | S ⇒\* w}
    - Context-free – means no overlap between blocks; each block can be analyzed in isolation
    - A language L is context-free if ∃ a CFG G . L(G) = L
* **Parse tree**
  + A.k.a. derivation trees
  + Root = start symbol
  + Internal nodes = non-terminals
  + Children of node = given by production rule
  + Lead nodes = terminals
  + Ambiguous grammar
    - Consider “1 – 10 + 11”
    - Grammar can use either E → E + E or E → E – E
      * Result in interpretation as “(1 – 10) + 11” or “1 – (10 + 11)”
    - i.e. this grammar is ambiguous
    - Formally:
      * A string w is ambiguous if there is more than one parse tree for w
      * A CFG G is ambiguous if ∃ at least one string w ∈ L(G) such that w is ambiguous
    - Ambiguity can be caused by having the same non-terminals on the RHS of rule, or allowing both left and right recursion
      * Change E → E + E to B + E
      * Change E → E – E to B – E
      * Thus only allows recursion on the right non-terminal (more operators can only be added on the right)
  + Process parse trees post-order with depth-first traversal
    - Depth-first – process first child before other children
    - Post-order – process children first before processing current node
* Ex: parsing arithmetic expressions
  + N = {E, T, F}
  + ∑ =
    - Operators: + − \* /
    - Delimiters: ( )
    - Constants: int (for numbers)
  + S = E
  + P =
    - E ⇒ E + T | E – T | T
    - T ⇒ T \* F | T / F | F
    - F ⇒ (E) | int
  + E.g. “2 + 3 \* 4”
    - E ⇒ E + T ⇒ T + T ⇒ F + T
    - ⇒ int + T ⇒ int + T \* F ⇒ int + F \* F
    - ⇒ int + int \* F ⇒ int + int \* int
  + CFGs should encode associativity and precedence in grammar