Syntax (IV)

Stages of Compilation
- We now know that the parse trees are for string validation. Then, how does a computer language (e.g., C and Java) validate the programs? [Compiler]
- Compilation has five stages shown as following.

- Compilation not only syntactically check programs, but also does semantically checking.
- **Lexical analyzer (Tokenization)**
  - takes the source file as input
  - converts the source code to a sequence of valid tokens
  - handles part of the production rules that have terminal symbols on the right-hand-side
  - discards invalid tokens after generating an error message
- Valid tokens includes
  - **identifiers**: e.g., variable names, function names
  - **literals**: e.g., numbers, characters, true/false
  - **keywords**: if, else, main, void, for, while, etc.
  - **operators**: +, -, *, /, &&, ||, ==, etc.
  - **punctuations**: ; {} () []
- **Syntactic analyzer**
  - takes a sequence of valid tokens as input
• parse the token sequence and constructs a parse tree/abstract syntax tree according to the grammar
• check syntax errors and ill-formed expressions

- Semantic analysis
  • takes abstract syntax tree as input
  • generates intermediate code, which can be considered as a more explicit, detailed parse tree where operators will generally be specific to the data type they are processing.
  • catches semantic errors such as undefined variables, variable type conflicts, and implicit conversions.

- Code optimization
  • takes the intermediate code as input
  • identify optimizations that speed up code execution without changing the program functionality
  • generate intermediate code that is generally across platforms

- Code generator
  • converts the intermediate code into machine code
  • machine code is tailored to a specific machine

- However, it’s worth noting that both tokenization and syntactic analyzer are for syntax validation. Why do most compilers separate tokenization from syntactic analyzer?
  • Tokenization is not a trivial task. Up to 75% of compilation time is taken by tokenization.

- Because tokenization is such a common process, there are nice tools for generating tokenization automatically based on the lexical syntax of a language.
  • Examples include lex and flex. Both are freely available, but flex is faster. So, we use flex in this course.
  • Flex (fast lexical analyzer generator) written in C around 1987 allows you to write the lexical syntax components of a language as a set of rules based on regular expressions.
  • We will talk about how to use flex later after an introduction to regular expressions.

Regular Expressions

- Regular expression (Regex) is a powerful tool in CS.
  • used in CS231 projects to define a pattern and find all matches
  • used in Vim to find and replace any strings
  • used in the implementation of the Find & Replacement function in a text editor

- Regular expressions are a language on their own designed to compactly represent a set of strings as a single expression.
- The idea of regex is similar to EBNF. It also has a set of meta symbols. Some meta symbols are the same as those in EBNF. But the meanings and usage of these meta symbols in Regex are different from in EBNF.

- We call the meta symbols of Regex special characters.

- **Special characters** in regular expressions:
  - []: used to specify a set of alternatives. Matching any single character in the set is considered as a valid match. *(different from EBNF)*
    - [AEIOU]: one uppercase vowel
    - T[ao]: tap, top
  - \: used as an escape character to permit use of other special character
    - \d: one digit from 0 to 9.
    - \s: whitespace
  - How do we write an regex to match all CS courses? [CS\s\d\d\d matches CS XXX]