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.

  

  **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

  **Semantic analyzer**
  - Generate a detailed, explicit parse tree: catch semantic errors

  **Code optimizer**
  - Generate an improved IC: speed up code execution

  **Code generator**
  - Generate the target machine code

- Compilation not only syntactically check programs, but also does semantically checking.
- 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[a-o]: 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]
- .: matches almost any character except line breaks
  - a.e: water, ate, gate
- *: match the prior expression zero or more times
  - \.: decimal point
  - How do we write a regex to match the floating point values with one digit after the decimal points. [\d*\d.: .3, 12.5, 139.9]
- -: range indicator
  - [a-z]: one lowercase letter from a to z
- ^: negates an expression when inside brackets, permits you to specify strings that don’t include a certain expression, or is the start of the string if it is outside of the brackets.
  - [\^0-9]: matches any character that is not a digit
  - ^a: matches strings start with a
- $: the end of the string
  - the end$: this is the end
  - How do we write a regex that can match any number between 1000 and 9999?
    - ^[1-9][0-9][0-9][0-9]$
    - create a text.txt with following numbers, one on each line, 1231 21 5 57 01001 100001 1000a; key in the command `egrep "^[1-9][0-9][0-9][0-9]" text.txt`; this command returns the matches in text.txt.
- (): group tokens (different from EBNF)
  - th(e|is): the, this
- +: match the prior expression one or more times
  - html tags: <html> </html>, <h1></h1>, <div id="block1"></div>
  - <[A-Za-z][A-Za-z0-9]*>: matches HTML tags without any attributes
  - <[A-Za-z0-9]+>: matches HTML tags without any attributes, but can have invalid tag like <1>
  - <[^< >]+>: matches HTML tags without regard to attributes
- {min, max}: specify how many times a token can be repeated, min >=0 minimum number of matches, max >= min maximum number of matches. If {min, } the maximum number of matches is infinite. {min} repeat exactly min times.
  - {0, } same as *, {1, } same as +
Abstract Syntax Tree

- It's also worth noting that the output of syntactic analyzer is an abstract syntax tree.
- So, what is an abstract syntax tree? We are going to use Clite to illustrate it.
- Before moving forward to the details of abstract syntax tree, let's take a look at Clite syntax first.

- Clite is a subset of C language. Its syntax is given below.

Concrete Syntax of Clite (from Tucker and Noonan, 2007).

\[
\begin{align*}
\text{Program} & \rightarrow \text{int} \ \text{main} \ ( ) \ {\{ \text{Declarations} \ \text{Statements} \}} \\
\text{Declarations} & \rightarrow \{ \text{Declaration} \} \\
\text{Declaration} & \rightarrow \text{Type} \ \text{Identifier} \ [ [ \text{Integer} ] ] \ {\{ , \ \text{Identifier} \ [ [ \text{Integer} ] ] \}} \\
\text{Type} & \rightarrow \text{int} | \text{bool} | \text{float} | \text{char} \\
\text{Statements} & \rightarrow \{ \ \text{Statement} \} \\
\text{Statement} & \rightarrow ; | \ \text{Block} | \ \text{Assignment} | \ \text{IfStatement} | \ \text{WhileStatement} \\
\text{Block} & \rightarrow \{ \ \text{Statements} \} \\
\text{Assignment} & \rightarrow \text{Identifier} \ [ [ \text{Expression} ] ] = \ \text{Expression}; \\
\text{IfStatement} & \rightarrow \text{if} \ ( \ \text{Expression} \ ) \ \text{Statement} \ [ \text{else} \ \text{Statement} ] \\
\text{WhileStatement} & \rightarrow \text{while} \ ( \ \text{Expression} \ ) \ \text{Statement} \\
\text{Expression} & \rightarrow \ \text{Conjunction} \ [ \ | \ \text{Conjunction} \] \\
\text{Conjunction} & \rightarrow \ \text{Equality} \ [ \ \&\& \ \text{Equality} \] \\
\text{Equality} & \rightarrow \ \text{Relation} \ [ \ \text{EquOp} \ \text{Relation} \] \\
\text{EquOp} & \rightarrow \ == | != \\
\text{Relation} & \rightarrow \ \text{Addition} \ [ \ \text{RelOp} \ \text{Addition} \] \\
\text{RelOp} & \rightarrow < | <= | > | >= \\
\text{Addition} & \rightarrow \ \text{Term} \ [ \ \text{AddOp} \ \text{Term} \] \\
\text{AddOp} & \rightarrow + | - \\
\text{Term} & \rightarrow \ \text{Factor} \ [ \ \text{MulOp} \ \text{Factor} \] \\
\text{MulOp} & \rightarrow * | / | % \\
\text{Factor} & \rightarrow \ \text{UnaryOp} \ \text{Primary} \\
\text{UnaryOp} & \rightarrow - | ! \\
\text{Primary} & \rightarrow \ \text{Identifier} \ [ [ \ \text{Expression} \ ] ] | \ \text{Literal} | ( \ \text{Expression} \ ) | \ \text{Type} ( \ \text{Expression} \ ) \\
\text{Identifier} & \rightarrow \ \text{Letter} \ [ \ \text{Letter} | \ \text{Digit} \] \\
\text{Letter} & \rightarrow a | b | \cdots | z | A | B | \cdots | Z \\
\text{Digit} & \rightarrow 0 | \cdots | 9 \\
\text{Literal} & \rightarrow \ \text{Integer} | \ \text{Boolean} | \ \text{Float} | \ \text{Char} \\
\text{Integer} & \rightarrow \ \text{Digit} \ [ \ \text{Digit} \ ] \\
\text{Boolean} & \rightarrow \text{true} | \text{false} \\
\text{Float} & \rightarrow \ \text{Integer} . \ \text{Integer} \\
\text{Char} & \rightarrow ' \ \text{ASCIIChar}'
\end{align*}
\]
- {} and [] in bold are EBNF meta-symbols.

- Read Clite syntax, and answer the following questions.
  - What are the T, N, P, S in Clite?
    - T: A ~ Z, a ~ z, 0 ~ 9, ASCIIChar, true, false, int, bool, float, char, main, if, while, operators
    - N: the symbols on the left-hand-size of the productions
    - P: all the rules
    - S: Program
  - What is the precedence of operators?

- Now, let draw a parse tree for $a = b + 3 \times c$; using Clite syntax.

Concrete grammar for Assignment and Expression (Tucker and Noonan, 2007).

```
Assignment → Identifier [ | Expression | ] = Expression;
Expression → Conjunction { || Conjunction }
Conjunction → Equality { && Equality }
Equality → Relation [ EquOp Relation ]
EquOp → == | !=
Relation → Addition [ RelOp Addition ]
RelOp → < | <= | > | >=
Addition → Term { AddOp Term }
AddOp → + | -
Term → Factor { MulOp Factor }
MulOp → * | / | %
Factor → [ UnaryOp ] Primary
UnaryOp → - | !
Primary → Identifier [ | Expression | ] | Literal | ( Expression ) | Type ( Expression )
```

![Parse tree diagram]

- {} and [] in bold are EBNF meta-symbols.

- Read Clite syntax, and answer the following questions.
  - What are the T, N, P, S in Clite?
    - T: A ~ Z, a ~ z, 0 ~ 9, ASCIIChar, true, false, int, bool, float, char, main, if, while, operators
    - N: the symbols on the left-hand-size of the productions
    - P: all the rules
    - S: Program
  - What is the precedence of operators?

- Now, let draw a parse tree for $a = b + 3 \times c$; using Clite syntax.

Concrete grammar for Assignment and Expression (Tucker and Noonan, 2007).

```
Assignment → Identifier [ | Expression | ] = Expression;
Expression → Conjunction { || Conjunction }
Conjunction → Equality { && Equality }
Equality → Relation [ EquOp Relation ]
EquOp → == | !=
Relation → Addition [ RelOp Addition ]
RelOp → < | <= | > | >=
Addition → Term { AddOp Term }
AddOp → + | -
Term → Factor { MulOp Factor }
MulOp → * | / | %
Factor → [ UnaryOp ] Primary
UnaryOp → - | !
Primary → Identifier [ | Expression | ] | Literal | ( Expression ) | Type ( Expression )
```
- From this parse tree, we can observe that
  • When the syntax becomes complexity, there are many long chains of non-terminal symbols that lead to a single terminal symbol in this large parse tree.
  • Many non-terminal symbols are used to describe the precedence of operators.
  • The use of non-terminal symbols also results in a unique parse tree, which guarantees that the language is unambiguous.

- However, most of the non-terminal symbols are not essential that contribute to the meaning of the expression. Using this large parse tree for semantic checking, the process will be tedious. Therefore, this large parse tree needs to be simplified. Remove all the non-terminal symbols that do not contribute to the semantic meaning, and only keep the essential relationship between components.
- We call the simplified parse tree, abstract syntax tree.