Syntax (V)

More special characters in regular expressions
- ^: 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]$" text.txt`
- (): group tokens (different from EBNF)
  - th(e | is) : the, this
- +: match the prior expression one or more times
  - html tags: <html> </html>, <h1></h1>, <div id="block1"></div>
  - <![A-Za-z][A-Za-z0-9]*]: matches HTML tags without any attributes
  - <![A-Za-z0-9]+>: matches HTML tags without any attributes, but can have invalid tag like `<1>
  -  : matches HTML tags without regard to attributes
- {min, max}: specify how many times a token can be repeated, min >=0 minimum number of matches, max >= min maximum number of matches. If {min, } the maximum number of matches is infinite. {min} repeat exactly min times.
  - {0, } same as *, {1, } same as +
  - ^[1-9][0-9][3]$ : matches a number between 1000 and 9999
  - ^[1-9][0-9][2,4]$ : matches a number between 100 and 99999

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.
  - {} and [] in bold are EBNF meta-symbols.
Concrete Syntax of Clite (from Tucker and Noonan, 2007).

```
Program → int main () { Declarations Statements }
Declarations → {Declaration}
Declaration → Type Identifier [[ Integer ]] { , Identifier [[ Integer ]] }
     Type → int | bool | float | char
Statements → { Statement }
Statement → ; | Block | Assignment | IfStatement | WhileStatement
    Block → { Statements }
Assignment → Identifier [ [ Expression ] ] = Expression;
IfStatement → if ( Expression ) Statement [ else Statement ]
WhileStatement → while ( Expression ) Statement

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 )

Identifier → Letter { Letter | Digit }
    Letter → a | b | … | z | A | B | … | Z
    Digit → 0 | … | 9
    Literal → Integer | Boolean | Float | Char
    Integer → Digit { Digit }
    Boolean → true | false
    Float → Integer . Integer
    Char → ’ASCIIChar’
```

- 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 \textit{Assignment} and \textit{Expression} (Tucker and Noonan, 2007).

\[
\text{Assignment} \rightarrow \text{Identifier} \left[ \left[ \text{Expression} \right] \right] = \text{Expression};
\]

\[
\text{Expression} \rightarrow \text{Conjunction} \left\{ \left|\right| \text{Conjunction} \right\}
\]

\[
\text{Conjunction} \rightarrow \text{Equality} \left\{ \left|\right| \text{Equality} \right\}
\]

\[
\text{Equality} \rightarrow \text{Relation} \left[ \left|\right| \text{EquOp Relation} \right]
\]

\[
\text{EquOp} \rightarrow \left|\right| \left|\right| !\left|\right| =\left|\right|
\]

\[
\text{Relation} \rightarrow \text{Addition} \left[ \left|\right| \text{RelOp Addition} \right]
\]

\[
\text{RelOp} \rightarrow \left|\right| \left|\right| \left|\right| \left|\right| >\left|\right| >\left|\right| =\left|\right|
\]

\[
\text{Addition} \rightarrow \text{Term} \left\{ \left|\right| \text{AddOp Term} \right\}
\]

\[
\text{AddOp} \rightarrow \left|\right| + \left|\right| -
\]

\[
\text{Term} \rightarrow \text{Factor} \left[ \left|\right| \text{MulOp Factor} \right]
\]

\[
\text{MulOp} \rightarrow \left|\right| \left|\right| \left|\right| !\left|\right|
\]

\[
\text{Factor} \rightarrow \left[ \left|\right| \text{UnaryOp Primary} \right]
\]

\[
\text{UnaryOp} \rightarrow \left|\right| !\left|\right|
\]

\[
\text{Primary} \rightarrow \text{Identifier} \left[ \left[ \left|\right| \text{Expression} \right] \right] \left|\right| \text{Literal} \left|\right| \left( \text{Expression} \right) \left|\right| \text{Type} \left( \text{Expression} \right)
\]

- 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.