

Syntax (IV)

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).

$Program \rightarrow \text{int main } () \{ \textit{Declarations} \textit{Statements} \}$
 $\textit{Declarations} \rightarrow \{ \textit{Declaration} \}$
 $\textit{Declaration} \rightarrow \textit{Type} \textit{Identifier} [[\textit{Integer}]] \{ , \textit{Identifier} [[\textit{Integer}]] \}$
 $\textit{Type} \rightarrow \text{int} \mid \text{bool} \mid \text{float} \mid \text{char}$
 $\textit{Statements} \rightarrow \{ \textit{Statement} \}$
 $\textit{Statement} \rightarrow ; \mid \textit{Block} \mid \textit{Assignment} \mid \textit{IfStatement} \mid \textit{WhileStatement}$
 $\textit{Block} \rightarrow \{ \textit{Statements} \}$
 $\textit{Assignment} \rightarrow \textit{Identifier} [[\textit{Expression}]] = \textit{Expression};$
 $\textit{IfStatement} \rightarrow \text{if } (\textit{Expression}) \textit{Statement} [\text{else } \textit{Statement}]$
 $\textit{WhileStatement} \rightarrow \text{while } (\textit{Expression}) \textit{Statement}$

$\textit{Expression} \rightarrow \textit{Conjunction} \{ \mid \mid \textit{Conjunction} \}$
 $\textit{Conjunction} \rightarrow \textit{Equality} \{ \&\& \textit{Equality} \}$
 $\textit{Equality} \rightarrow \textit{Relation} [\textit{EquOp} \textit{Relation}]$
 $\textit{EquOp} \rightarrow == \mid !=$
 $\textit{Relation} \rightarrow \textit{Addition} [\textit{RelOp} \textit{Addition}]$
 $\textit{RelOp} \rightarrow < \mid < = \mid > \mid > =$
 $\textit{Addition} \rightarrow \textit{Term} \{ \textit{AddOp} \textit{Term} \}$
 $\textit{AddOp} \rightarrow + \mid -$
 $\textit{Term} \rightarrow \textit{Factor} \{ \textit{MulOp} \textit{Factor} \}$
 $\textit{MulOp} \rightarrow * \mid / \mid \%$
 $\textit{Factor} \rightarrow [\textit{UnaryOp}] \textit{Primary}$
 $\textit{UnaryOp} \rightarrow - \mid !$
 $\textit{Primary} \rightarrow \textit{Identifier} [[\textit{Expression}]] \mid \textit{Literal} \mid$
 $(\textit{Expression}) \mid \textit{Type} (\textit{Expression})$

$\textit{Identifier} \rightarrow \textit{Letter} \{ \textit{Letter} \mid \textit{Digit} \}$
 $\textit{Letter} \rightarrow \text{a} \mid \text{b} \mid \dots \mid \text{z} \mid \text{A} \mid \text{B} \mid \dots \mid \text{Z}$
 $\textit{Digit} \rightarrow 0 \mid \dots \mid 9$
 $\textit{Literal} \rightarrow \textit{Integer} \mid \textit{Boolean} \mid \textit{Float} \mid \textit{Char}$
 $\textit{Integer} \rightarrow \textit{Digit} \{ \textit{Digit} \}$
 $\textit{Boolean} \rightarrow \text{true} \mid \text{false}$
 $\textit{Float} \rightarrow \textit{Integer} . \textit{Integer}$
 $\textit{Char} \rightarrow ' \textit{ASCIICChar} '$

- Read Clite syntax, and answer the following questions.
 - What are the T, N, P, S in Clite?
 - T: $A \sim Z$, $a \sim z$, $0 \sim 9$, ASCIIChar, true, false, int, bool, float, char, main, if, while, operators
 - N: the symbols on the left-hand-side of the productions
 - P: all the rules
 - S: Program
 - What is the precedence of operators?
 - $\text{UnaryOp} > \text{MulOp} > \text{AddOp} > \text{RelOp} > \text{EquOp} > \&\& > \text{||}$
- 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).

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

$\text{Expression} \rightarrow \text{Conjunction} \{ || \text{Conjunction} \}$

$\text{Conjunction} \rightarrow \text{Equality} \{ \&\& \text{Equality} \}$

$\text{Equality} \rightarrow \text{Relation} [\text{EquOp} \text{Relation}]$

$\text{EquOp} \rightarrow == | !=$

$\text{Relation} \rightarrow \text{Addition} [\text{RelOp} \text{Addition}]$

$\text{RelOp} \rightarrow < | < = | > | > =$

$\text{Addition} \rightarrow \text{Term} \{ \text{AddOp} \text{Term} \}$

$\text{AddOp} \rightarrow + | -$

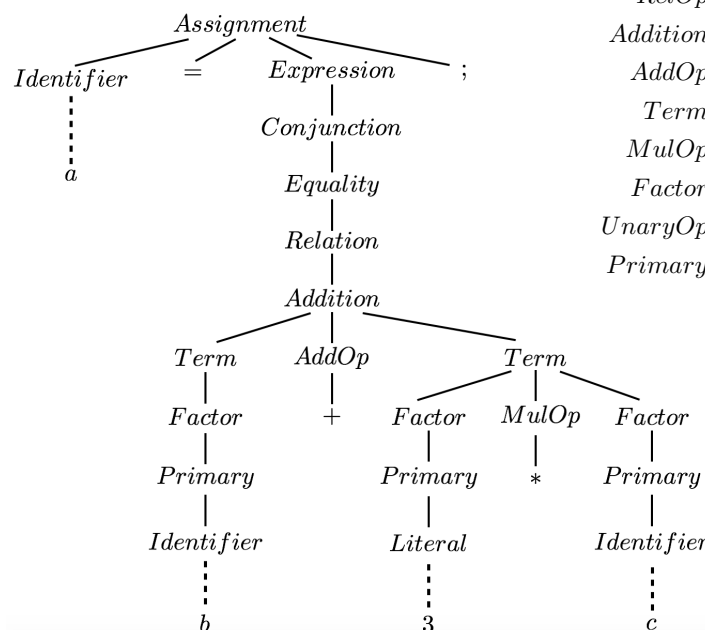
$\text{Term} \rightarrow \text{Factor} \{ \text{MulOp} \text{Factor} \}$

$\text{MulOp} \rightarrow * | / | \%$

$\text{Factor} \rightarrow [\text{UnaryOp}] \text{Primary}$

$\text{UnaryOp} \rightarrow - | !$

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

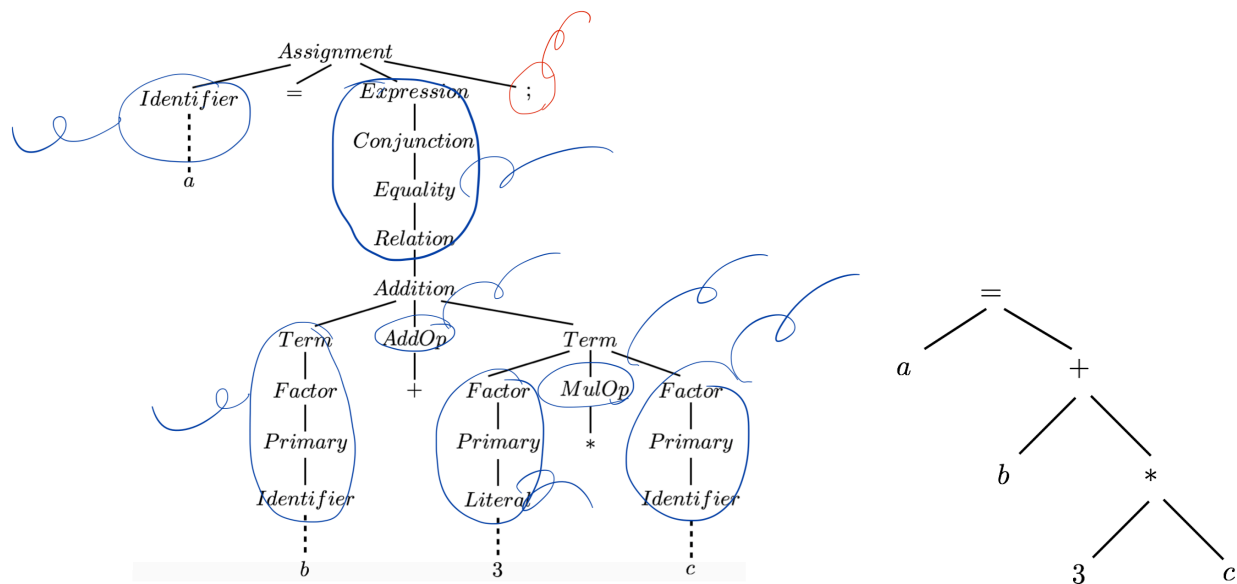


- From the above parse tree, we can observe that:

- 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**.

Build an ABS Tree from a Parse Tree

- There are three steps to get an abstract syntax tree from a parse tree.
 1. **Discard separators or terminating symbols.**
 2. **Discard all non-terminal symbols that are trivial root.** A trivial root is a symbol with only **one subtree**.
 3. Replace remaining non-terminal symbols with operators, which are the immediate subtree.



- Characteristics of abstract syntax tree
 - **No keywords or terminating symbols**

- [Precedence of operators depends on the corresponding parse tree](#). If the precedence of operators in the parse tree is incorrect, operator precedence will be wrong in the abstract syntax tree.
- Abstract syntax tree is general [across languages](#).
- Abstract syntax tree is [not completely language independent](#) due to different semantic rules
 - For example, the variable `c` in the above abstract can be string or number in python and `*` means differently in terms of data types. If variable `c` is a string, then `3 * c` concatenate three strings together. If variable `c` is a number, then `3 * c` times the number by 3.
 - [This is also why semantic analyzer generates a detailed abstract syntax tree](#) that specifies types of operators.