Chi-Hao Tu & Yaoyao Liu

COMP 141: Syntax trees

Instructions: In this exercise, we are going to review parse trees, and abstract syntax trees.

1 CFGs

Consider the following grammar.

$$expr :: = expr + expr \mid expr * expr \mid (expr) \mid number$$

$$number :: = number\ digit \mid digit$$

$$digit :: = 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

1. Redefine the grammar in a way that numbers are defined by regular expressions. That is, remove two grammar rules for *number* and *digit* nonterminal, and replace them with a single regular expressions for numbers.

   expr ::= [0-9]+

2. Modify the grammar to define assignment statements with a syntax similar to C/C++, where expressions are assigned to identifiers. Here are the steps:

   (a) Add identifier token *ID* to the CFG. You may define it with regular expressions (since identifiers are tokens). We have already seen how to define identifiers with regular expressions, in previous sessions.

   ID ::= letter (letter | digit)*

   letter::=[a-z]|[A-Z]|\

   digit::=[0-9]

   (b) Consider a nonterminal for assignment. Let's call this nonterminal *assignment*. You need to define a CFG rule that defines *assignment*. An assignment in C/C++ grammatically consists of an identifier (defined in previous step), followed by equality symbol (a terminal), followed by an expression (an already defined nonterminal). This sequence of terminals and nonterminals should be given in the RHS of your CFG rule.

   assignment ::= ID '=' expr

2 Derivations

Consider the following CFG.

$$expr \ ::= \ expr \ \# \ term \ | \ term$$
$$term \ ::= factor \ @ \ term \ | \ factor$$
$$factor \ ::= (expr) \ | \ 0 \ | \ 1$$

expr is the starting nonterminal. Moreover, (,), 0, 1, #, and @ are terminals. Give derivations for the following expressions:

1. 1 @ 1 # 1

   expr

   => term # term (using expr ::= term # term)
   => factor @ term # term (using term ::= factor @ term)
   => 1 @ term # term (using factor ::= 1)
   => 1 @ factor # term (using term ::= factor)
   => 1 @ 1 # term (using factor ::= 1)
   => 1 @ 1 # factor (using term ::= factor)
   => 1 @ 1 # 1 (using factor ::= 1)

2. 0 @ 0 @ 0

   Expr

   => term (using expr ::= term)
   => factor @ term (using term ::= factor @ term)
   => 0 @ term (using factor ::= 0)
   => 0 @ factor @ term (using term ::= factor @ term)
   => 0 @ 0 @ term (using factor ::= 0)
   => 0 @ 0 @ factor (using term ::= factor)
   => 0 @ 0 @ 0 (using factor ::= 0)

3. 0 @ (0 # 1)

   expr
   => term (using expr ::= term)
   => factor @ term (using term ::= factor @ term)
   => 0 @ term (using factor ::= 0)
   => 0 @ factor (using term ::= factor)
   => 0 @ (expr)(using factor ::=(expr))
   => 0 @ (term # term) (using expr ::= term # term)
   => 0 @ (factor # term) (using term ::= factor)
   => 0 @ (0 # term) (using factor ::= 0)
   => 0 @ (0 # factor) (using term ::= factor)
   => 0 @ (0 # 1) (using factor ::= 1)

3 Syntax trees

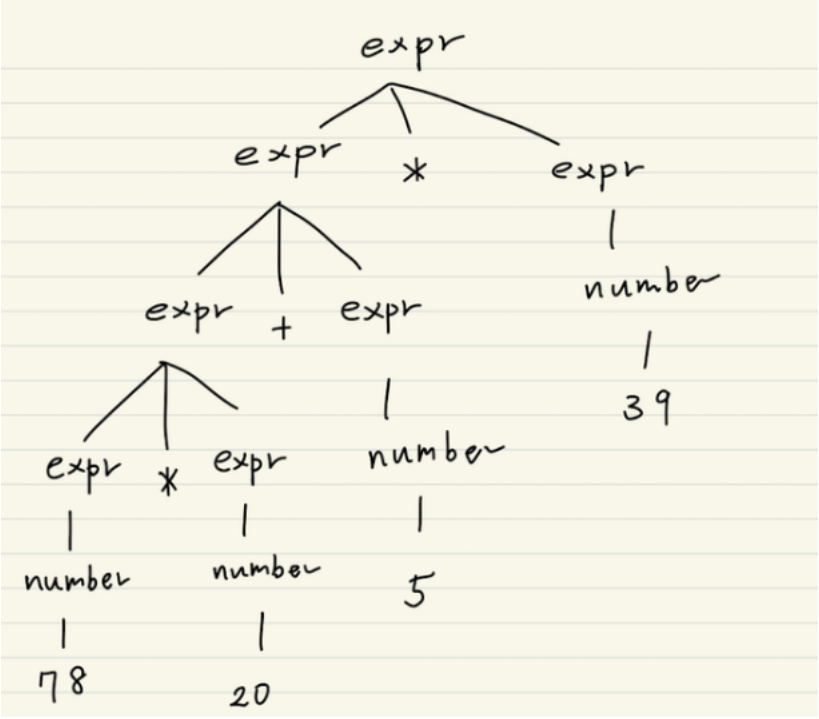
Consider the following grammar with terminals: number, +, ∗, (, and).

$$expr ::= expr + expr \mid expr * expr \mid (expr) \mid \text{number}$$

number = [0-9]+

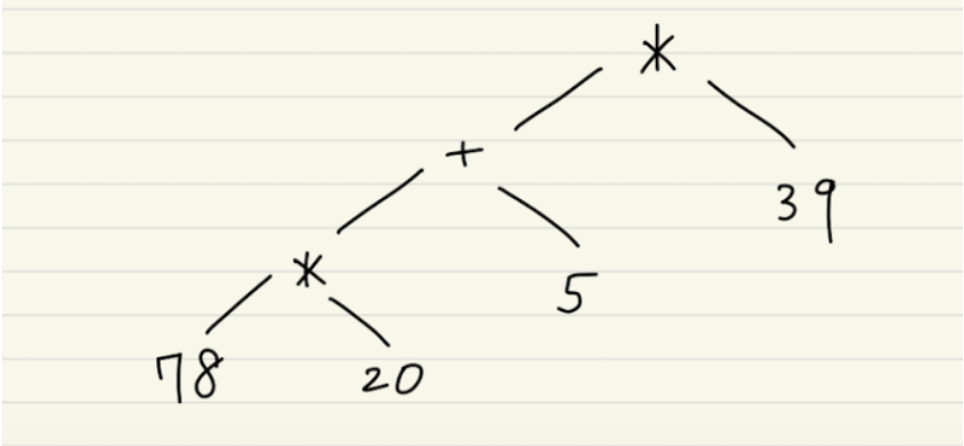
1. We can derive the expression $78 * 20 + 5 * 39$ as follows (among many other derivations).

$$\begin{aligned} expr &\Rightarrow expr * expr \\ &\Rightarrow expr + expr * expr \\ &\Rightarrow expr * expr + expr * expr \\ &\Rightarrow \text{number} * expr + expr * expr \\ &\Rightarrow 78 * expr + expr * expr \\ &\Rightarrow 78 * expr + \text{number} * expr \\ &\Rightarrow 78 * expr + 5 * expr \\ &\Rightarrow 78 * expr + 5 * \text{number} \\ &\Rightarrow 78 * expr + 5 * 39 \\ &\Rightarrow 78 * \text{number} + 5 * 39 \\ &\Rightarrow 78 * 20 + 5 * 39 \end{aligned}$$



Give the parse tree that represents this derivation. Let's call your parse tree P_1.

2. Give the corresponding AST for P_1. Let's call this AST A_1.



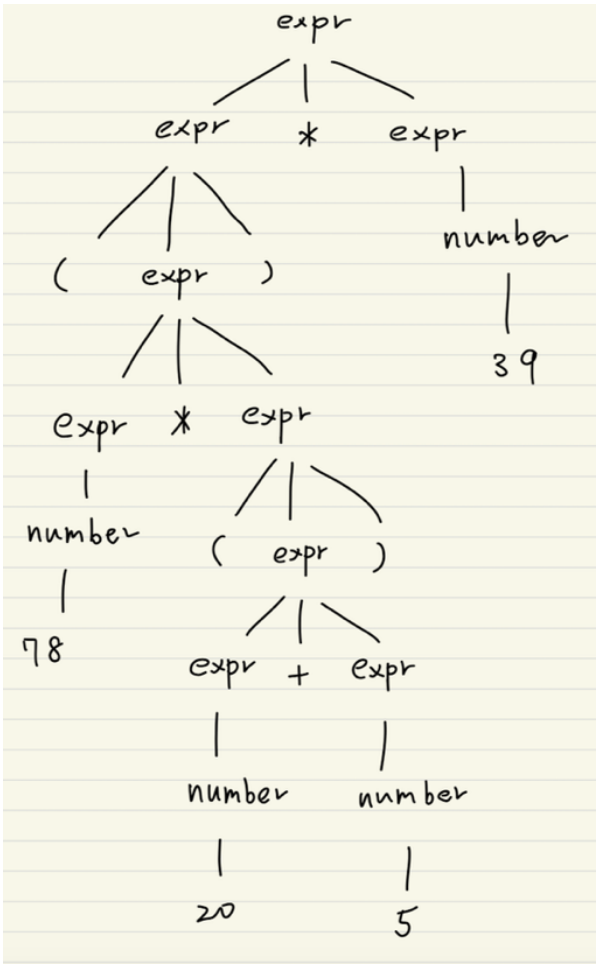
3. If you pass A_1 to some evaluator, what would be the final value?

   The final value of the expression **78 * 20 + 5 * 39**, when evaluated according to the Abstract Syntax Tree (AST), is 1755.

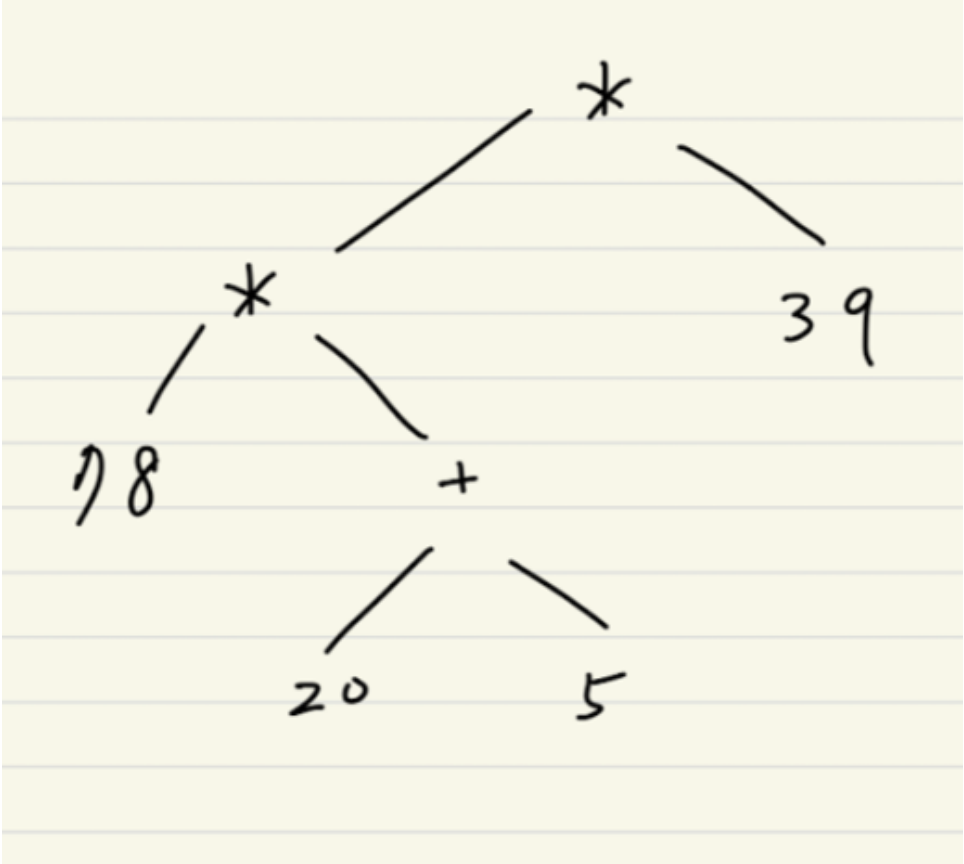
4. We can derive the expression $(78 * (20 + 5)) * 39$ as follows (among many other derivations).

$$
\begin{aligned}
expr &\Rightarrow expr * expr \\
&\Rightarrow (expr) * expr \\
&\Rightarrow (expr * expr) * expr \\
&\Rightarrow (expr * (expr)) * expr \\
&\Rightarrow (expr * (expr + expr)) * expr \\
&\Rightarrow (expr * (expr + expr)) * \text{number} \\
&\Rightarrow (expr * (\text{number} + expr)) * \text{number} \\
&\Rightarrow (expr * (\text{number} + expr)) * 39 \\
&\Rightarrow (expr * (20 + expr)) * 39 \\
&\Rightarrow (expr * (20 + \text{number})) * 39 \\
&\Rightarrow (expr * (20 + 5)) * 39 \\
&\Rightarrow (\text{number} * (20 + 5)) * 39 \\
&\Rightarrow (78 * (20 + 5)) * 39,
\end{aligned}
$$

Give the parse tree that represents this derivation. Let's call your parse tree P_2.



5. Give the corresponding AST for P_2. Let's call this AST A_2.



6. If you pass A_2 to some evaluator, what would be the final value?

The parse tree (P2) and the Abstract Syntax Tree (AST A2) for the expression **(78 * (20 + 5)) * 39** have been created.

4 Boolean expressions

Consider the following CFG with the eight terminals: true, false, ∧, ∨, !, ==, (, and).

expr ::= true | false | *expr* ∧ *expr* | *expr* ∨ *expr* | !*expr* | *expr* == *expr* | (*expr*)

Indeed, the starting symbol is *expr*. Let's call this grammar G. Consider the following expression:

!true ∧ false ∨ true == true

1. Give two different derivations for this expression such that the corresponding parse trees are different from each other.

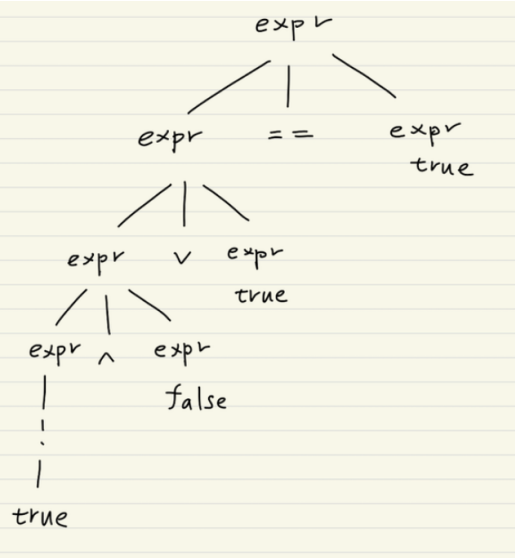
a)

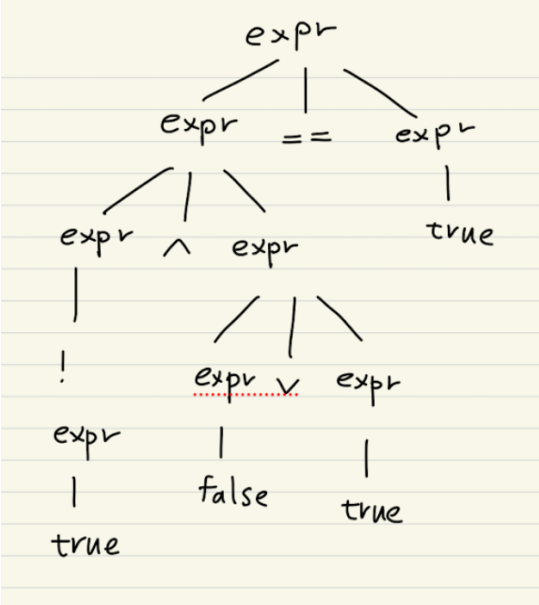
expr → expr == expr
→expr ∨ expr == expr
→ expr ∧ expr ∨ expr == expr
→ !expr ∧ expr ∨ expr == expr
→!true ∧ expr ∨ expr == expr
→!true ∧ false ∨ expr == expr
→!true ∧ false ∨ true == expr
→!true ∧ false ∨ true == true

b)

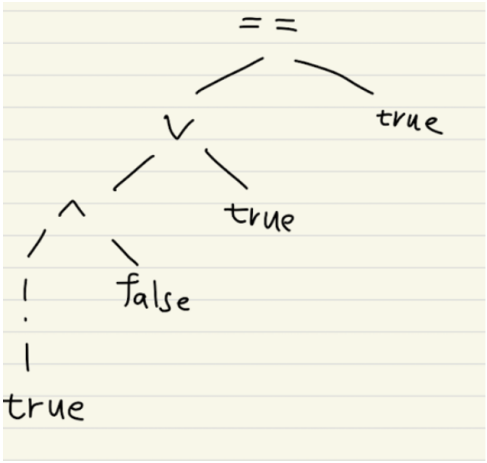
expr → expr == expr
→expr ∧ expr == expr
→!expr ∧ expr == expr
→!true ∧ expr == expr
→!true ∧ expr ∨ expr == expr
→!true ∧ false ∨ expr == expr
→!true ∧ false ∨ true == expr
→!true ∧ false ∨ true == true

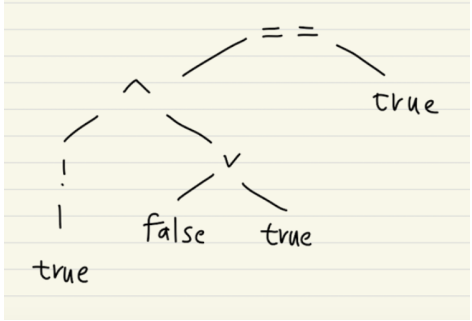
2. Give the corresponding parse trees for each derivation in the previous question.





3. Give the corresponding two ASTs. (Hint: Note that operators ∧, ∨, ==, and ! can appear as interior nodes in ASTs[1]. You may remove the nonterminal *expr* from ASTs as it does not convey any computational information.)





4. If you pass the ASTs from the previous question to an evaluator, what would be the final value in each case?

   For both ASTs, the final evaluated value is **true**.

[1] Operator ! would have a single child, as it is a unary operator.