COMP 141: Ambiguity, EBNFs and Parsers

Instructions: In this exercise, we are going to review EBNFs and parsers.

1 Ambiguous Grammars

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

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

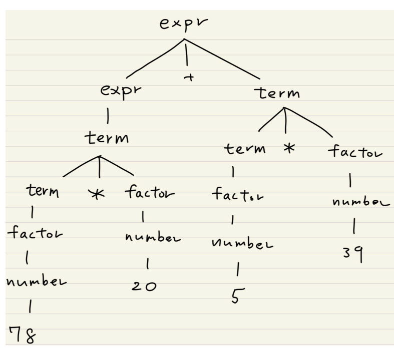
number = [0-9]+

1. As you have seen in the slides, we can disambiguate the grammar by revising it as follows

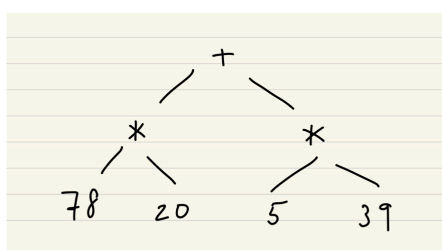
$$expr ::= expr + term \mid term$$
$$term ::= term * factor \mid factor$$
$$factor ::= (expr) \mid \text{number}$$

number = [0-9]+

(a) Since the grammar is disambiguated, there is only one parse tree now for each expression that can be derived from the grammar. What is the unique parse tree for deriving $78 * 20 + 5 * 39$. Let's call this parse tree P.



(b) Give the corresponding AST for P. Let's call it A.



(c) What would be the final value if you pass A to an evaluator (in an interpreter)?

To evaluate this AST:

1. Multiply **78** by **20** to get **1560**.
2. Multiply **5** by **39** to get **195**.
3. Add the results of the two multiplications together: **1560 + 195** to get **1755**.

So the final value after evaluating the AST would be **1755**.

2 EBNFs

1. As discussed in the class, given the BNF grammar

$$
\begin{aligned}
&expr ::= expr + term \mid term \\
&term ::= term * factor \mid factor \\
&factor ::= (expr) \mid number \\
&number ::= \text{NUMBER} \\
&\text{NUMBER} = [0-9]+
\end{aligned}
$$

we can rewrite it in EBNF as follows.

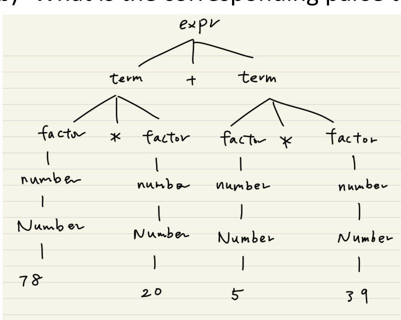
$$
\begin{aligned}
&expr ::= term \ \{+ \ term\} \\
&term ::= factor \ \{* \ factor\} \\
&factor ::= (expr) \mid number \\
&number ::= \text{NUMBER} \\
&\text{NUMBER} = [0-9]+
\end{aligned}
$$

Let's call this definition of grammar g_1.

(a) What is the derivation for $78 * 20 + 5 * 39$ using the rules in g_1?

```
expr
=> term + term
=> factor * factor + term
=> number * factor + term
=> '78' * factor + term
=> '78' * factor + term
=> '78' * number + term
=> '78' * '20' + term
=> '78' * '20' + term
=> '78' * '20' + factor * factor
=> '78' * '20' + number * factor
=> '78' * '20' + '5' * factor
=> '78' * '20' + '5' * factor
=> '78' * '20' + '5' * number
=> '78' * '20' + '5' * '39'
```

(b) What is the corresponding parse tree?



(c) What is the corresponding AST?

```
      '+'
     /   \
   '*'   '*'
  / \   / \
'78''20''5' '39'
```

2. Moreover, we can rewrite the BNF grammar

$$
\begin{aligned}
expr &::= term \ + \ expr \ \mid \ term \\
term &::= factor * term \mid factor \\
factor &::= (expr) \mid number
\end{aligned}
$$

as the EBNF grammar:

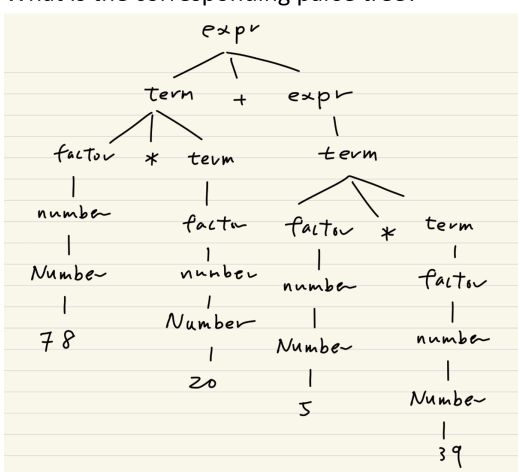
$$
\begin{aligned}
expr &::= term \ [+ \ expr] \\
term &::= factor \ [* \ term] \\
factor &::= (expr) \mid number
\end{aligned}
$$

Let's call this definition of grammar g_2.

(a) What is the derivation for $78 * 20 + 5 * 39$ using the rules in g_2?

```
expr
=> term + expr
=> factor * term + expr
=> number * term + expr
=> '78' * term + expr
=> '78' * term + expr
=> '78' * factor + expr
=> '78' * number + expr
=> '78' * '20' + expr
=> '78' * '20' + expr
=> '78' * '20' + term
=> '78' * '20' + factor * term
=> '78' * '20' + number * term
=> '78' * '20' + '5' * term]
=> '78' * '20' + '5' * term
=> '78' * '20' + '5' * factor
=> '78' * '20' + '5' * number
=> '78' * '20' + '5' * '39'
```

(b) What is the corresponding parse tree?



(c) What is the corresponding AST?

```
       '+'
      /   \
    '*'    '*'
   /  \   /  \
'78''20''5' '39'
```

3 Recursive-descent parser

1. Give the pseudo-code for recursive-descent parser that implements g_1.

```
function parseExpr():
    return parseTerm() + parseExprRest()

function parseExprRest():
    if the next token is '+':
        match('+')
        return parseTerm() + parseExprRest()
    else:
        return ε

function parseTerm():
    return parseFactor() * parseTermRest()

function parseTermRest():
    if the next token is '*':
        match('*')
        return parseFactor() * parseTermRest()
    else:
        return ε

function parseFactor():
    if the next token is '(':
        match('(')
        result = parseExpr()
        match(')')
        return result
    else:
        return parseNumber()

function parseNumber():
    if the next token is a number:
        match(NUMBER)
        return the number
    else:
        throw SyntaxError
```

2. Give the pseudo-code for recursive-descent parser that implements g_2.

```
function parseExpr():
    result = parseTerm()
    while the next token is '+':
        match('+')
        result += parseTerm()
    return result

function parseTerm():
    result = parseFactor()
    while the next token is '*':
        match('*')
        result *= parseFactor()
    return result

function parseFactor():
    if the next token is '(':
        match('(')
        result = parseExpr()
        match(')')
        return result
    else:
        return parseNumber()

function parseNumber():
    if the next token is a number:
        match(NUMBER)
        return the number
    else:
        throw SyntaxError
```

4 Boolean expressions

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

$$expr ::= \text{true} \mid \text{false} \mid expr \wedge expr \mid expr \vee expr \mid !expr \mid expr == expr \mid (expr)$$

Indeed, the starting symbol is $expr$. Let's call this grammar G. This grammar is ambiguous, i.e., there exist at least two parse trees for some expression. For example, in a previous lab you were able to give two different syntax trees for the following expression:

!true ∧ false ∨ true == true

1. Let's disambiguate G. We want to impose the following precedence cascade among operators:
   - The highest precedence is for parentheses,
   - the second highest precedence is for !,
   - the third highest precedence is for ∧,
   - the fourth highest precedence is for ∨, and finally
   - the least precedence is for ==.

   In addition, all binary operators are left-recursive. Define the disambiguated version of the grammar in BNF. Let's call your disambiguated grammar G'.

   expr::=expr==term|term

   term::=term∨factor|factor

   factor::=factor∧excl|excl

   excl::=!excl|paren

   paren::=(expr)|boolean

   boolean::=treu|false

2. In your defined G', is operator ∧ left-associative or right-associative? What about operator ∨?

   In the defined grammar **G'**, both the **∧** (logical AND) and **∨** (logical OR) operators are left-associative.

3. Using G', give the derivation for the expression below:

   ! true ∧ false ∨ true == true

   Note that since G' is disambiguated, there must be a unique parse tree for this expression.

expr→=expr==term

→term==term

→term∨factor==term

→term∨factor==term

→factor∨factor==term

→factor∧excl∨factor==term

→excl∧excl∨factor==term

→!excl∧excl∨factor==term

→!paren∧excl∨factor==term

→!boolean∧excl∨factor==term

→!true∧excl∨factor==term

→!true∧paren∨factor==term

→!true∧boolean∨factor==term

→!true∧false∨factor==term

→!true∧false∨excl==term

→!true∧false∨paren==term

→!true∧false∨boolean==term

→!true∧false∨true==term

→!true∧false∨true==factor

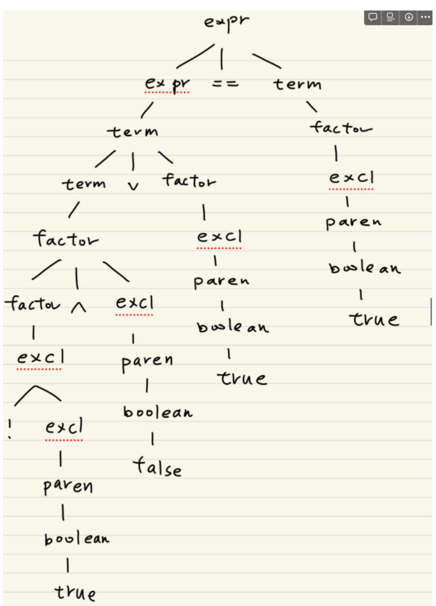
→!true∧false∨true==excl

→!true∧false∨true==paren

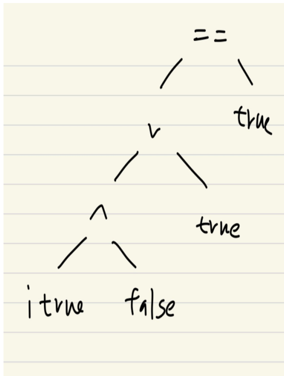
→!true∧false∨true==boolean

→!true∧false∨true==true

4. Give the corresponding parse tree for the derivation in the previous question.



5. Give the corresponding AST for the parse tree in the previous question.



6. If you pass this AST to an evaluator, what would be the final result?

   the final result of evaluating the AST would be **true**.

7. Redefine G' using EBNF. Let's call this version of grammar G''.

   expr::=term{==term}

   term::=factor{∨factor}

   factor::=excl{∧excl}

   excl::={!}paren

   paren::=(expr)|boolean

   boolean::=treu|false

8. Give the pseudo-code for the recursive-descent parser that implements G''. The parser needs to generate the AST (so it is not a recognizer!).

```
bool parseExpr() {
  node_tree = parseTerm()
  if node_tree =='==':
    consume_token();
    node_tree = new PTInteriorNode('==',node_tree,parseTerm())

  return node_tree

}

bool parseTerm() {
  node_tree = parseFactor()
  if node_tree =='v':
    consume_token();
    node_tree = new PTInteriorNode('v',node_tree,parseTerm())

  return node_tree

}
bool parseFactor() {
  node_tree = parseTerm()
  if node_tree =='∧':
    consume_token();
    node_tree = new PTInteriorNode('∧',node_tree,parseFactor()

  return node_tree

}
bool parseExcl() {
  node_tree = parseParen()
  if node_tree =='!':
    consume_token();
    node_tree = new PTInteriorNode('!',node_tree,parseExcl())

  return node_tree
}
bool parseParen() {

  node_tree =  parseBoolean()

  if node_tree == ‘(’:
    consume_token();
    t = parseExpr();
    if node_tree == ‘)’
      return node_tree
    else:
      return false
  else if node_tree == boolean:
    consume_token();
    return node_tree

  else:
    return false


bool parseBoolean() {

  if next_token  == true
    return true;
  else
    return false

}
```