COMP 141: Ambiguity, EBNFs and Parsers

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

# Ambiguous Grammars

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

*expr* ::= *expr* + *expr* | *expr* ∗ *expr* | (*expr*) | 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* | *term term* ::= *term* ∗ *factor* | *factor factor* ::= (*expr*) | number number = [0-9]+

1. 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.

A diagram of a function

Description automatically generated

1. Give the corresponding AST for P. Let’s call it A.

A drawing of a triangle with numbers and stars

Description automatically generated

1. 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**.

# EBNFs

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

*expr* ::= *expr* + *term* | *term term* ::= *term* ∗ *factor* | *factor factor* ::= (*expr*) | *number number* ::= NUMBER

## NUMBER = [0 − 9]+

we can rewrite it in EBNF as follows.

*expr* ::= *term* {+ *term*} *term* ::= *factor* {∗ *factor*} *factor* ::= (*expr*) | *number number* ::= NUMBER

## NUMBER = [0 − 9]+

Let’s call this definition of grammar *g*1.

1. What is the derivation for 78 ∗ 20 + 5 ∗ 39 using the rules in *g*1?

A screenshot of a computer

Description automatically generated

1. What is the corresponding parse tree?

A diagram of a number

Description automatically generated with medium confidence

1. What is the corresponding AST?

A close-up of a number

Description automatically generated

2. Moreover, we can rewrite the BNF grammar

*expr* ::= *term* + *expr* | *term term* ::= *factor* ∗ *term* | *factor factor* ::= (*expr*) | *number*

as the EBNF grammar:

*expr* ::= *term* [+ *expr*] *term* ::= *factor* [∗ *term*] *factor* ::= (*expr*) | *number*

Let’s call this definition of grammar *g*2.

1. What is the derivation for 78 ∗ 20 + 5 ∗ 39 using the rules in *g*2?

A screenshot of a computer

Description automatically generated

1. What is the corresponding parse tree?

A diagram of a number

Description automatically generated

1. What is the corresponding AST?

A triangle with numbers and symbols

Description automatically generated with medium confidence

# Recursive-descent parser

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

A screenshot of a computer program

Description automatically generated

1. Give the pseudo-code for recursive-descent parser that implements *g*2.

A screenshot of a computer program

Description automatically generated

# 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*. 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*’.

A computer screen shot of a computer code

Description automatically generated

1. 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.

1. 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.

A screenshot of a computer program

Description automatically generated

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

A computer screen shot of a tree

Description automatically generated

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

A computer screen shot of a tree

Description automatically generated

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

the operator precedence:

1. Evaluate the **!** operator: **!true** evaluates to **false**.
2. Evaluate the **∧** (AND) operator with the result of the previous operation and **false**: **false ∧ false** evaluates to **false**.
3. Evaluate the **∨** (OR) operator with the result of the previous operation and **true**: **false ∨ true** evaluates to **true**.
4. Evaluate the **==** (equality) operator with the result of the previous operation and **true**: **true == true** evaluates to **true**.

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

1. Redefine *G*’ using EBNF. Let’s call this version of grammar *G*’’.

A computer screen shot of a computer code

Description automatically generated

1. 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!).

function parseExpr():

return parseEquality()

function parseEquality():

left = parseLogicalOr()

if the next token is '==':

match('==')

right = parseLogicalOr()

return ASTNode('==', left, right)

return left

function parseLogicalOr():

left = parseLogicalAnd()

while the next token is '∨':

match('∨')

right = parseLogicalAnd()

left = ASTNode('∨', left, right)

return left

function parseLogicalAnd():

left = parseNotExpr()

while the next token is '∧':

match('∧')

right = parseNotExpr()

left = ASTNode('∧', left, right)

return left

function parseNotExpr():

if the next token is '!':

match('!')

expression = parsePrimary()

return ASTNode('!', expression)

return parsePrimary()

function parsePrimary():

if the next token is 'true':

match('true')

return ASTNode('true')

elif the next token is 'false':

match('false')

return ASTNode('false')

elif the next token is '(':

match('(')

expression = parseExpr()

match(')')

return expression

else:

throw SyntaxError("Expected primary expression")

function match(expectedToken):

if the next token in the input matches expectedToken:

consume and move to the next token

else:

throw SyntaxError("Unexpected token")

class ASTNode:

def \_\_init\_\_(self, type, left=None, right=None):

self.type = type

self.left = left

self.right = right