ECS 140A Discussion 2

Chris Fernandez, Ty Feng

2:10-3:50pm Thursday, Aug 17 2023

UCDAVIS

Today's agenda

BNF

Derivations

Parse Trees

BNF => EBNF

Grammar ambiguity

BNF

- BNF stands for Backus–Naur Form
- A BNF grammar description is an unordered list of rules. Rules are used to define symbols with the help of other symbols.
- There are 2 types of symbols, Terminals and Non Terminals.
- Terminals are the symbols that represent the actual elements of a language. They are meant to be used as they are. Eg. “a”, “1”, “Java”
- Non-terminals are symbols that we use to refer to something else. In BNF, non-terminal names are written within angle brackets. Eg. <statement>, <expression>

BNF

<S>::= <A> b

- **Left-hand side:** Left side will always have a non terminal that we use to define the rule. (<S>)
- **::=** : This character group separates the Left hand side from Right hand side. Read this symbol as "is defined as".
- **Right-hand side**: The definition of the non-terminal specified on the right-hand side. Can have both terminals and non terminals. (<A>b)
- Right side can be a combination of one or more terminals or non-terminals in a sequence and the result is their concatenation, with non-terminals being replaced by their content. Order is important.
- The | operator indicates that the parts separated by it are choices. Order is unimportant.

BNF example

<sentence> ::= <subject> <predicate>

<subject> ::= “I am” | “You are” | “He is” | “She is” | “They are”

<predicate> ::= <feeling> | <occupation>

<feeling> ::= “happy.” | “sad.” | “excited.”

<occupation> ::= “a programmer.” | “a developer.”

Derivations

- Derivations help us understand how code is parsed and analyzed by compiler
- Derivation shows how a sentence is generated from a grammar
- Typically, left-most derivation is used
- Eg. Derive 3*5+2 from the following grammar using left-most derivation

  <expression> ::= <term> + <term>

  <term> ::= <factor> * <factor> | <factor>

  <factor> ::= 1| 2| 3| 4| 5

Derivation Example 1: 3*5+2

Grammar:

<expression> ::= <term> + <term>

<term> ::= <factor> * <factor> | <factor>

<factor> ::= 1| 2| 3| 4| 5

Derivation:

Derivation Example 1: 3*5+2

Grammar:

<expression> ::= <term> + <term>
<term> ::= <factor> * <factor> | <factor>
<factor> ::= 1| 2| 3| 4| 5

Derivation:

<expression> ::= <term> + <term>
<expression> ::= <factor> * <factor> + <term>
<expression> ::= 3 * <factor> + <term>
<expression> ::= 3 * 5 + <term>
<expression> ::= 3 * 5 + <factor>
<expression> ::= 3 * 5 + 2

Derivation Example 2: 3*5+2*4

Grammar:

<expression> ::= <term> + <term>

<term> ::= <factor> * <factor> | <factor>

<factor> ::= 1| 2| 3| 4| 5

Derivation:

Derivation Example 2: 3*5+2*4

Grammar:

<expression> ::= <term> + <term>
<term> ::= <factor> * <factor> | <factor>
<factor> ::= 1| 2| 3| 4| 5

Derivation:

<expression> ::= <term> + <term>
<expression> ::= <factor> * <factor> + <term>
<expression> ::= 3 * <factor> + <term>
<expression> ::= 3 * 5 + <term>
<expression> ::= 3 * 5 + <factor> * <factor>
<expression> ::= 3 * 5 + 2 * <factor>
<expression> ::= 3 * 5 + 2 * 4

Derivation Example 3: A = A * (B + (C * A))

Grammar:

<assign> -> <id> = <expr>

<id> -> A | B | C

<expr> -> <id> + <expr>

| <id> * <expr>

| (<expr>)

| <id>

Derivation Example 3: A = A * (B + (C * A))

<assign> => <id> = <expr>

=> A = <expr>

=> A = <id> * <expr>

=> A = A * <expr>

=> A = A * (<expr>)

=> A = A * (<id> + <expr>)

=> A = A * (B + <expr>)

=> A = A * (B + (<expr>))

=> A = A * (B + (<id> * <expr>))

=> A = A * (B + (C * <expr>))

=> A = A * (B + (C * <id>))

=> A = A * (B + (C * A))

Derivation Example 4: `for(int i = 1; i != 5; i++) {i = 1}`

Grammar:

<expression> ::= for (<initialization> ; <condition> ; <update>) {<statement>}

<initialization> ::= int <identifier> = <number>

<condition> ::= <identifier> != <number>

<update> ::= <identifier>++

<identifier> ::= i | j | k

<statement> ::= <identifier> = <number>

<number> ::= 1|2|3|4|5

Derivation Example 4: `for(int i = 1; i != 5; i++) {i = 1}`

<expression> ::= for (<initialization> ; <condition> ; <update>) {<statement>}

<expression> ::= for (int <identifier> = <number>; <condition> ; <update>) {<statement>}

<expression> ::= for (int i= <number>; <condition> ; <update>) {<statement>}

<expression> ::= for (int i= 1; <condition> ; <update>) {<statement>}

<expression> ::= for (int i= 1; <identifier> != <number>; <update>) {<statement>}

<expression> ::= for (int i= 1; i != <number>; <update>) {<statement>}

Derivation Example 4: `for(int i = 1; i != 5; i++) {i = 1}`

<expression> ::= for (int i= 1; i != 5; <update>) {<statement>}

<expression> ::= for (int i= 1; i != 5; <identifier>++) {<statement>}

<expression> ::= for (int i= 1; i != 5; i++) {<statement>}

<expression> ::= for (int i= 1; i != 5; i++) {<statement>}

<expression> ::= for (int i= 1; i != 5; i++) {<identifier> = <number>}

<expression> ::= for (int i= 1; i != 5; i++) {i = <number>}

<expression> ::= for (int i= 1; i != 5; i++) {i = 1}

Parse Trees

- Parse trees are graphical representations of how a string of symbols is syntactically structured according to a formal grammar.
- They help visualize the hierarchical relationships between different components of a language.
- Nodes: Represent symbols (terminal or non-terminal) in the grammar.
- Edges: Connect nodes to show how symbols are derived from one another.
- Root: The topmost node, representing the starting symbol of the grammar.
- Leaves: The bottom nodes, representing terminal symbols in the string.

BNF example

<sentence> ::= <subject> <predicate>

<subject> ::= “I am” | “You are” | “He is” | “She is” | “They are”

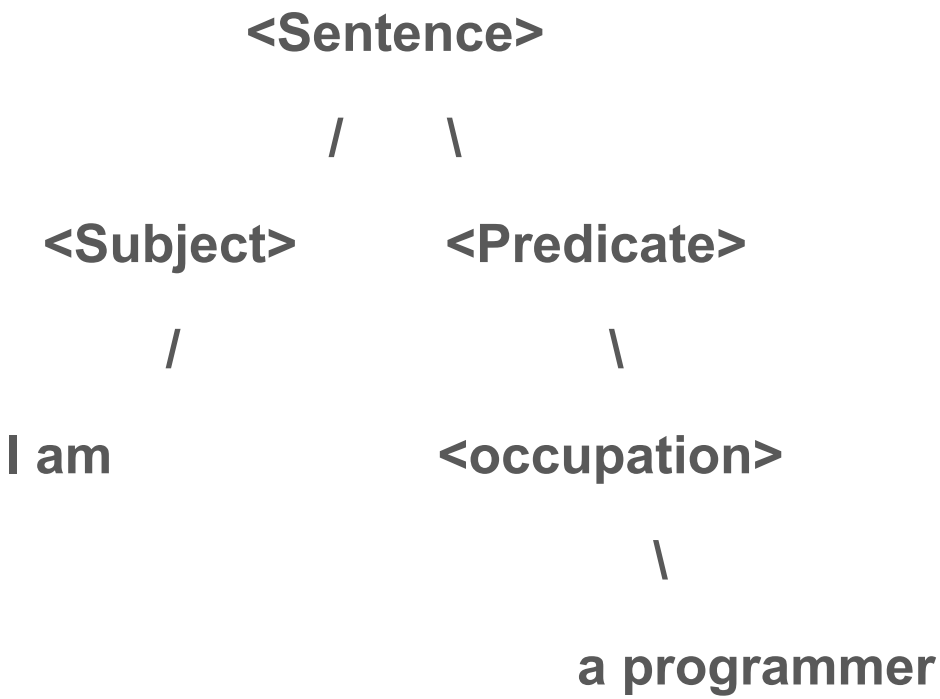
<predicate> ::= <feeling> | <occupation>

<feeling> ::= “happy.” | “sad.” | “excited.”

<occupation> ::= “a programmer.” | “a developer.”

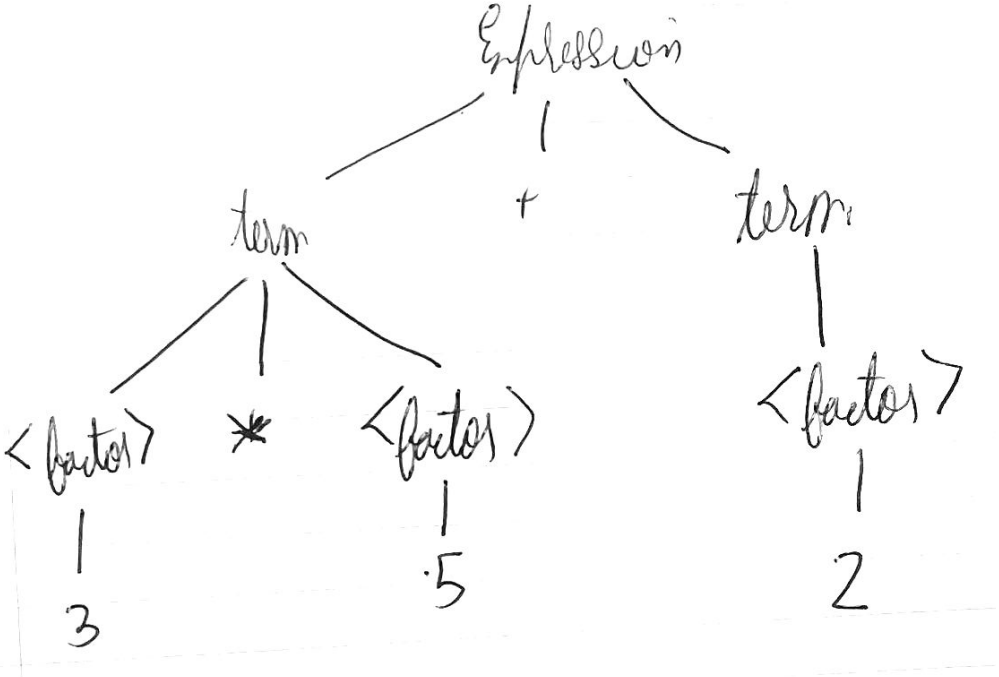
Parse Tree for “I am a programmer”

I am a programmer



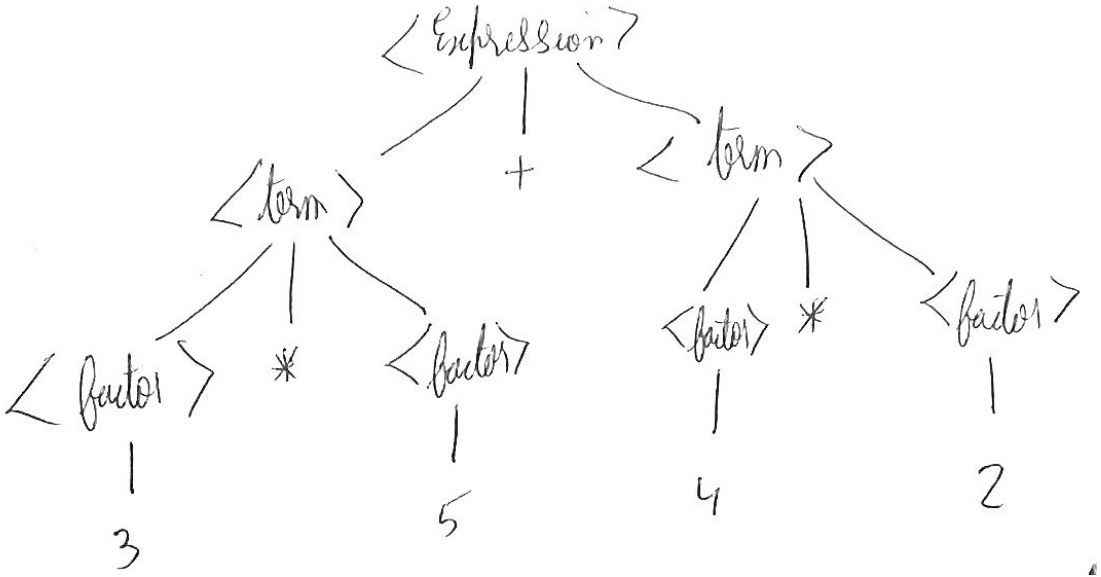
Parse Tree Example 1: 3*5+2

<expression> ::= <term> + <term>
<term> ::= <factor> * <factor> | <factor>
<factor> ::= 1| 2| 3| 4| 5



Parse Tree Example 2: 3*5+2*4

<expression> ::= <term> + <term>
<term> ::= <factor> * <factor> | <factor>
<factor> ::= 1| 2| 3| 4| 5



Parse Tree Example 3: for(int i = 1; i != 5; i++) {i = 1}

<expression> ::= for (<initialization> ; <condition> ; <update>) {<statement>}

<initialization> ::= int <identifier> = <number>

<condition> ::= <identifier> != <number>

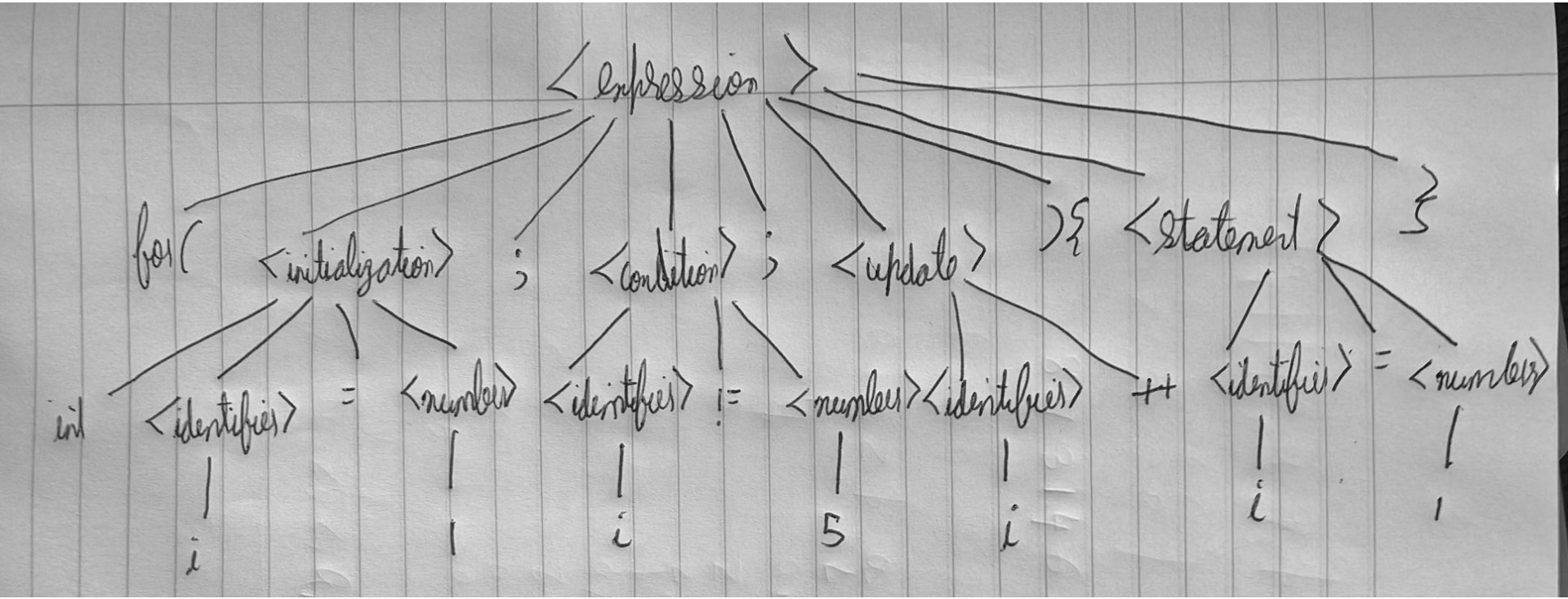
<update> ::= <identifier>++

<identifier> ::= i | j | k

<statement> ::= <identifier> = <number>

<number> ::= 1|2|3|4|5

Parse Tree Example 3: for(int i = 1; i != 5; i++) {i = 1}



EBNF

- EBNF stand for Extended Backus-Naur Form.
- EBNF adds some additional notation and features to BNF to make it more expressive and convenient for describing complex language structures.
- Everything you can express in EBNF can also be expressed in BNF.
- EBNF should have no recursion!
- EBNF extends BNF by adding the following 3 operations:
  1. Option
  2. Repetition
  3. Grouping

Optional Elements: EBNF allows you to use square brackets [ ] to indicate that an element is optional. For example, [<expression>] indicates that <expression> may or may not appear.

Repetition: EBNF introduces curly braces { } to represent repetition of elements. For example, <digit>{<digit>} represents one or more digits.

Grouping: Parentheses can be used to indicate grouping. It means everything they wrap can be replaced with any of the valid strings that the contents of the group represent according to the rules of EBNF. Example -

<ball> ::= <type> "ball"
<type> ::= "basket" | "foot"

BNF

<ball> ::= (“basket” | “foot”) "ball"

EBNF

BNF to EBNF Example 1

BNF:

<program> ::= start<stmtlist> end

<stmtlist> ::= <stmt>|<stmt>;<stmtlist>

<stmt> ::= <var> = <expression>

<var> ::= A | B | C

<expression> ::= <var> + <var>
| <var> - <var> | <var>

EBNF:

<program> ::= start<stmtlist> end

<stmtlist> ::= <stmt> { ; <stmt> }

<stmt> ::= <var> = <expression>

<var> ::= A | B | C

<expression> ::= <var> [(+ | -) <var>]

BNF to EBNF Example 2

BNF:

<assign> -> <id> = <expr>

<expr> -> <id> + <expr>

| <id> * <expr>

| <id>

BNF to EBNF Example 2

EBNF:

<assign> -> <id> = <expr>

<expr> -> <id> {(+|*) <id>}

Grammar Ambiguity

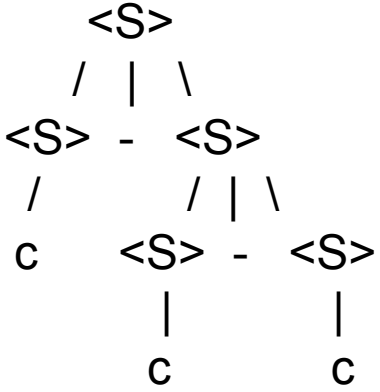
- Grammar ambiguity occurs when a given input string can be parsed or interpreted in more than one way according to the grammar rules.
- Ambiguity can lead to confusion, unexpected behavior, and challenges in compiler design and language specification.
- Ambiguity can arise from:

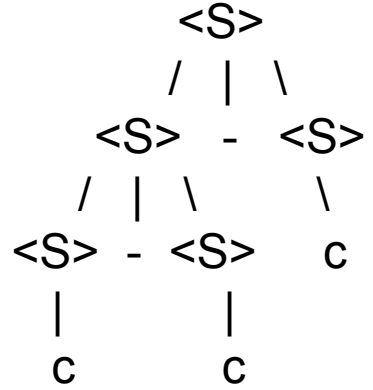
1. Lack of clear operator precedence and associativity rules.
2. Conflicting grammar rules that allow multiple interpretations.
3. Overloading of operators with different meanings in different contexts.

Example 1 of ambiguous grammar

<S> ::= <S>-<S> | c

For the string “c-c-c”, parse tree can be constructed in 2 different ways

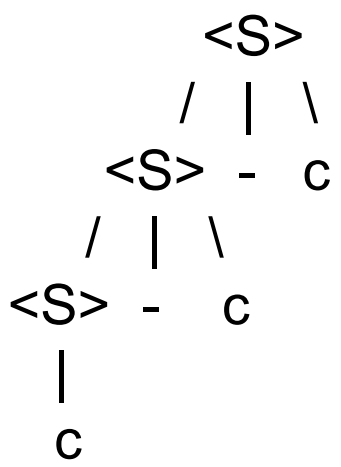




Convert to disambiguous

<S> ::= <S>-c | c

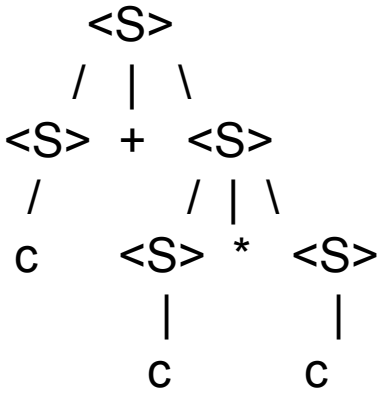
Parse Tree for c-c-c :

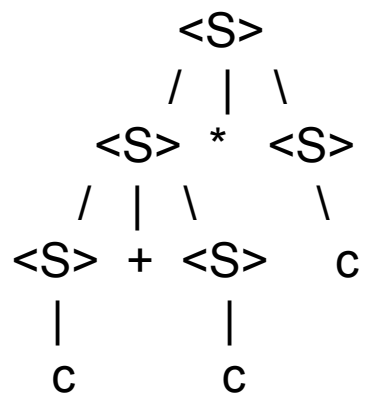


Example 2 of ambiguous grammar

<S> ::= <S>+<S> | <S>*<S> | c

For the string “c+c*c”, parse tree can be constructed in 2 different ways



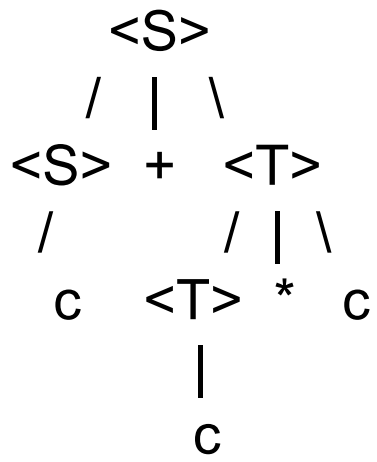


Convert to disambiguous

<S> ::= <S>+<T> | <T>

<T>::= <T> * c | c

Parse Tree for c+c*c :



Quiz Time!

Go to Canvas > Quizzes

Published | Preview | Edit

Discussion 2 Practice Quiz (Not for grade)

This is a preview of the draft version of the quiz

Parse Trees, BNF to EBNF practice.

Quiz Type	Practice Quiz
Points	4
Shuffle Answers	No
Time Limit	15 Minutes
Multiple Attempts	Yes
Score to Keep	Highest
Attempts	Unlimited
View Responses	Always
Show Correct Answers	Immediately
One Question at a Time	No
Require Respondus LockDown Browser	No
Required to View Quiz Results	No
Webcam Required	No

Due	For	Available from	Until
Aug 17 at 4pm	Everyone	Aug 17 at 2:20pm	Aug 17 at 11:59pm

Preview

Submitted Aug 17 at 4:32pm

Question 1

1 / 1 pts

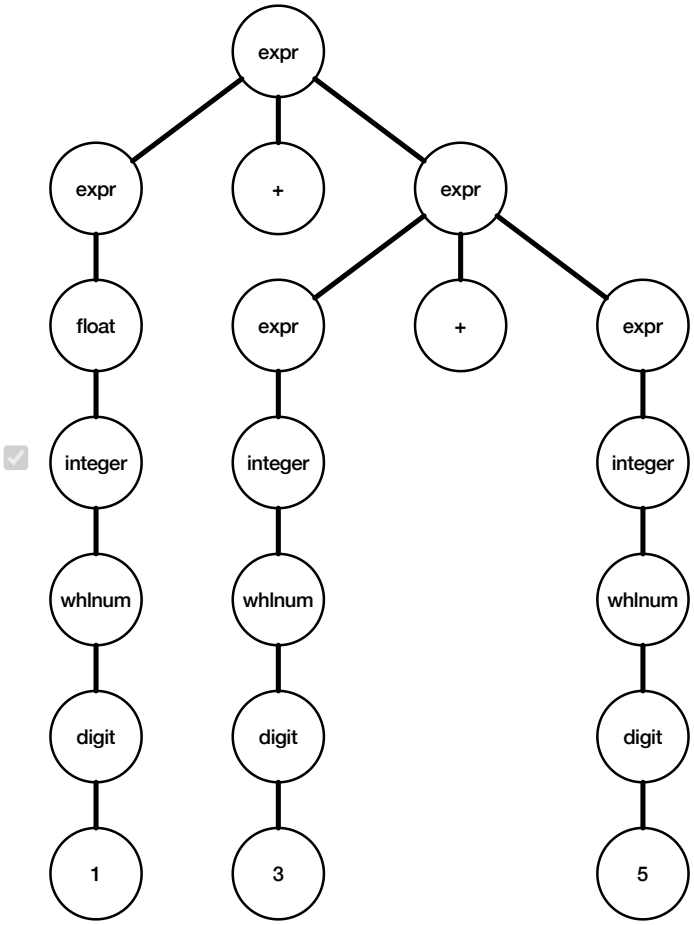
Given the following BNF:

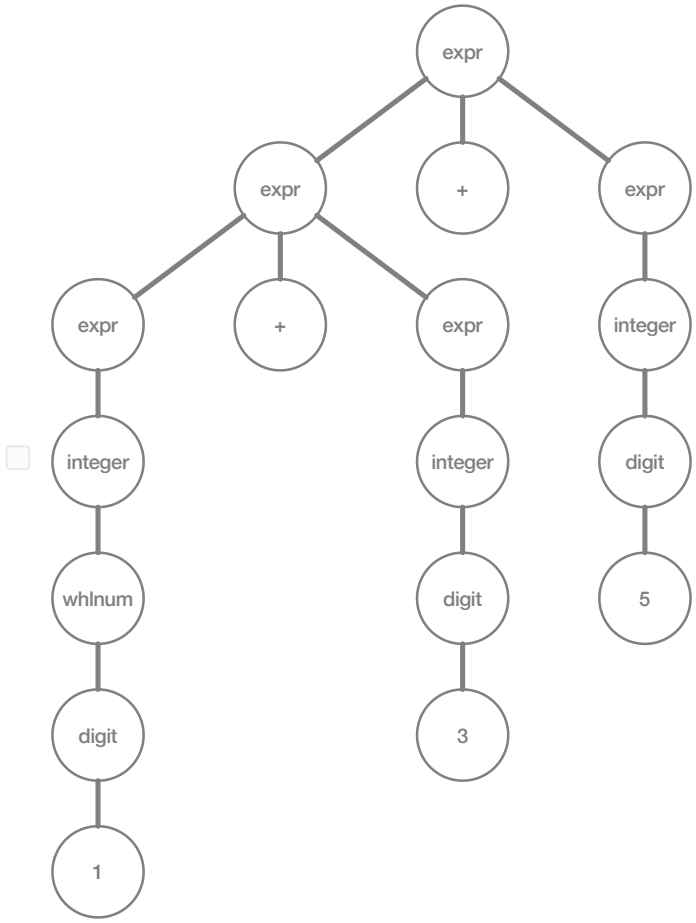
```
<expr> -> <expr> + <expr> | <integer> | <float>
<float> -> <integer> | <integer> . <whlnum>
<integer> -> <whlnum> | - <whlnum>
<whlnum> -> <digit> | <digit> <whlnum>
<digit> -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

What are all the possible parsings of:

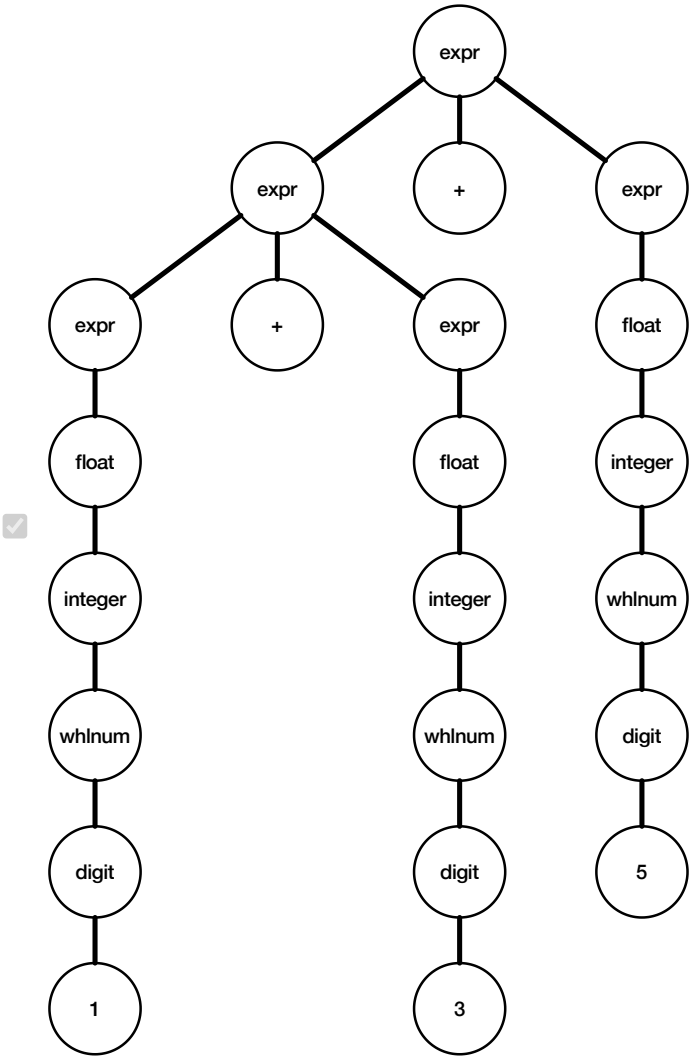
```
1 + 3 + 5
```

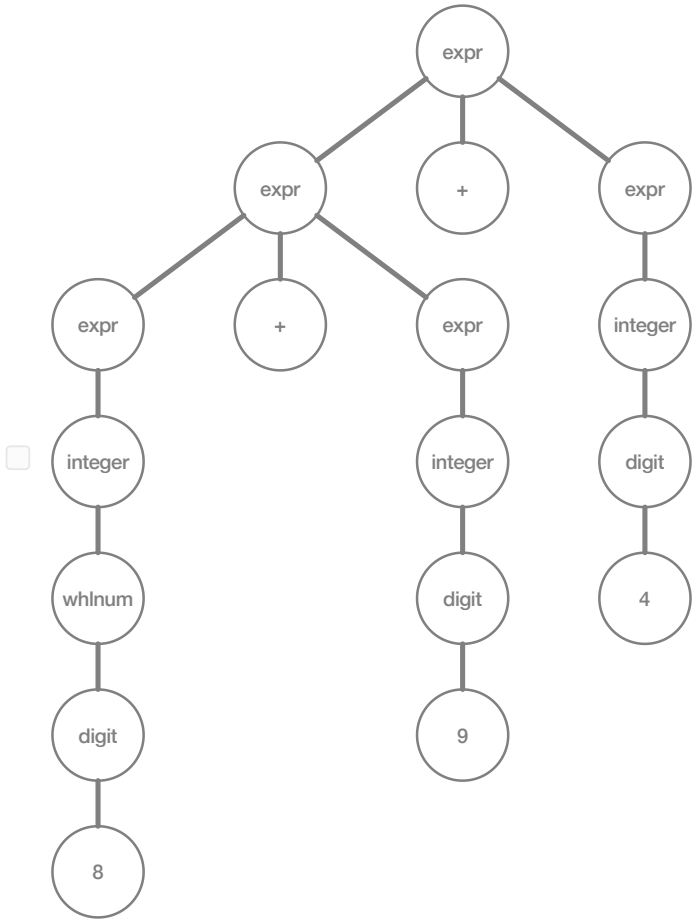
Correct!

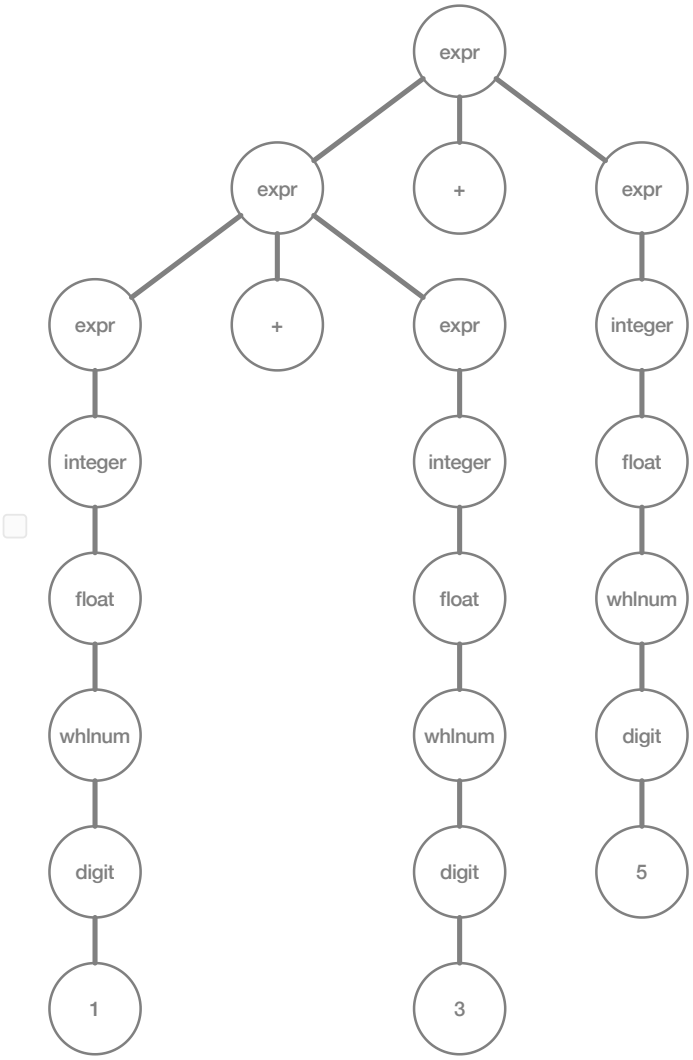



expr
expr
+
expr
expr
+
expr
integer
integer
integer
digit
whlnum
digit
5
digit
3
1

Correct!


expr
expr
+
expr
expr
+
expr
float
float
float
integer
integer
integer
whlnum
whlnum
whlnum
digit
digit
digit
5
1
3


expr
expr
+
expr
expr
+
expr
integer
integer
integer
digit
whlnum
digit
4
digit
9
8



Question 2

1 / 1 pts

Given a grammar for some language, if we can derive more than one parse trees for some sentence using the grammar, it means the grammar is:

Correct!

ambiguous

Correct Answers

Ambiguous

ambiguous

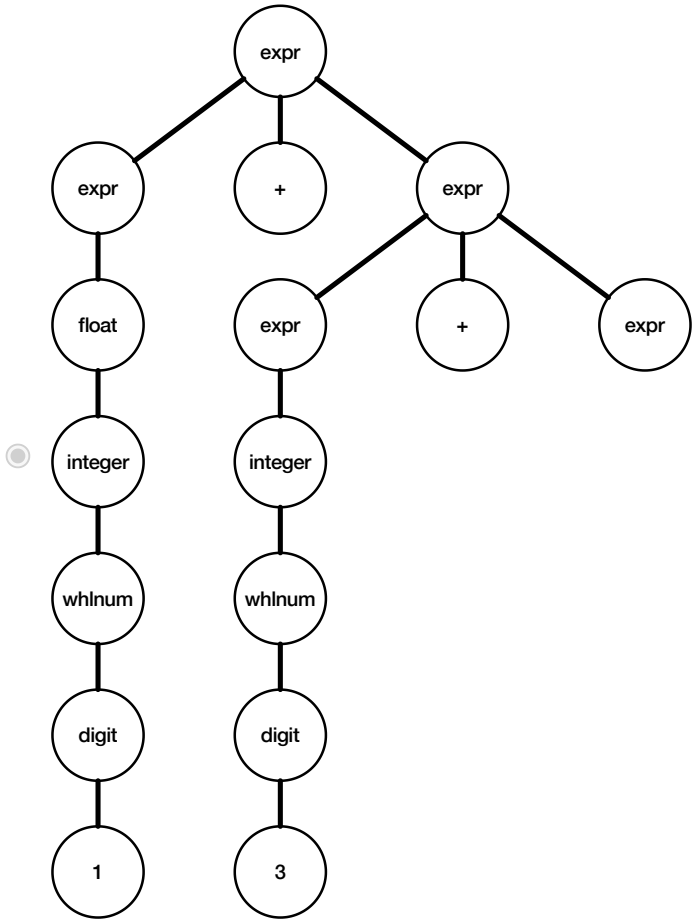
Question 3

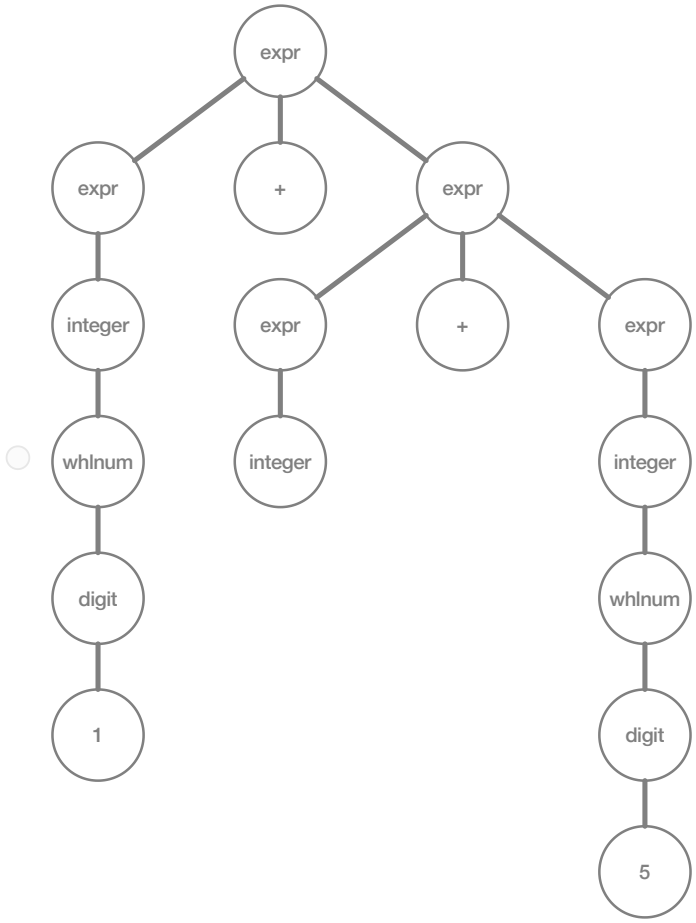
1 / 1 pts

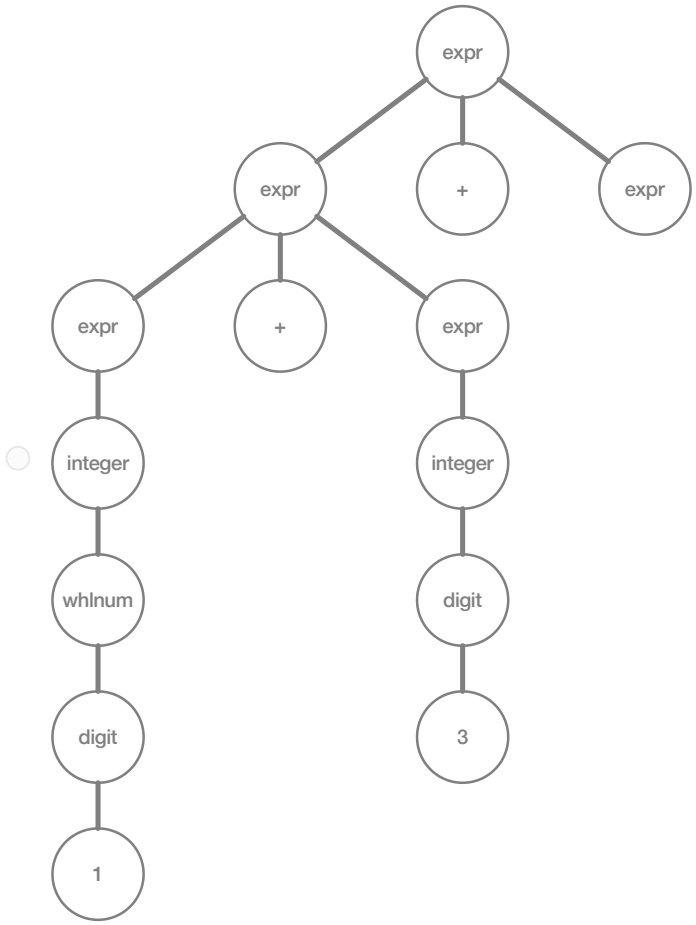
Given the following BNF, select one of the partially completed parse trees that have been derived using leftmost derivation thus far.

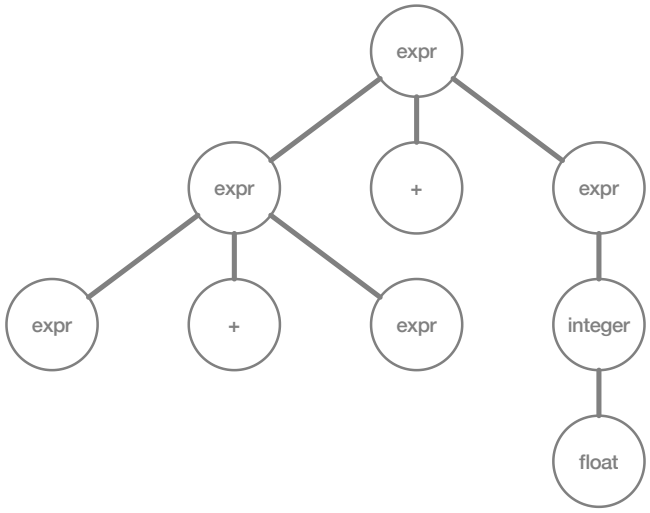
```
<expr> -> <expr> + <expr> | <integer> | <float>
<float> -> <integer> | <integer> . <whlnum>
<integer> -> <whlnum> | - <whlnum>
<whlnum> -> <digit> | <digit> <whlnum>
<digit> -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Correct!




expr
expr
+
expr
integer
expr
+
expr
whlnum
integer
integer
digit
whlnum
1
digit
5


expr
expr
+
expr
expr
+
expr
integer
integer
whlnum
digit
digit
3
1



Question 4

1 / 1 pts

Select all the equivalent BNF/EBNF pairs.

BNF:

```
<float> -> <integer> | <integer> . wholenumber
```

EBNF:

```
<float> -> <integer> {. <wholenumber>}
```

Correct!

BNF:

```
<float> -> <integer> | <integer> . <wholenumber>
```

EBNF:

```
<float> -> <integer> [. <wholenumber>]
```

BNF:

```
<wholenumber> -> <digit> | <digit> <wholenumber>
```

EBNF:

```
<wholenumber> -> {<digit>}
```

BNF:

```
<expr> -> <term> + <expr> | <term>
```

EBNF:

```
<expr> -> <term> [+ <expr> ]
```

BNF:

```
<integer> -> - <wholenumber> | <wholenumber>
```

EBNF:

```
<integer> -> {-} <wholenumber>
```