esercizio 1

→ 6 febbraio 2025

(a) Per ogni stringa elencata sotto stabilire, motivando la risposta, se appartiene al linguaggio definito dalla seguente espressione regolare e, in tal caso, indicare il gruppo di appartenenza (escludendo il gruppo 0).

```
(0|[1-9]\d?\d?(?:_?\d\d\d)*)|([a-zA-Z]+(?:-*\w)*)|(%%(%?[^%])*%%)
```

i. "123_45"
ii. "0_123_4"
iii. "%%ab%%cd%%"
iv. "%%ab%c%%"
v. "a--b--c"
vi. "1a-b-c"

i: l'unico gruppo che definisce un insieme con stringhe che iniziano con 1 è il primo, ma la stringa non appartiene al gruppo perchè _ deve essere seguito da tre cifre

ii: non appartiene a nessun gruppo, perchè nel primo gruppo la stringa può iniziare con 0 solo se è l'unica cifra della stringa

iii: potrebbe appartenere al terzo gruppo, ma non è possibile in quanto il simbolo % centrale non può essere seguito da un secondo %

iv: appartiene al terzo gruppo

v: appartiene al secondo gruppo, con "a" ∈ [a-z A-Z] e "--b" e "--c" che ∈ -*\w, ossia -* vuol dire che - può essere ripetuto zero o più volte

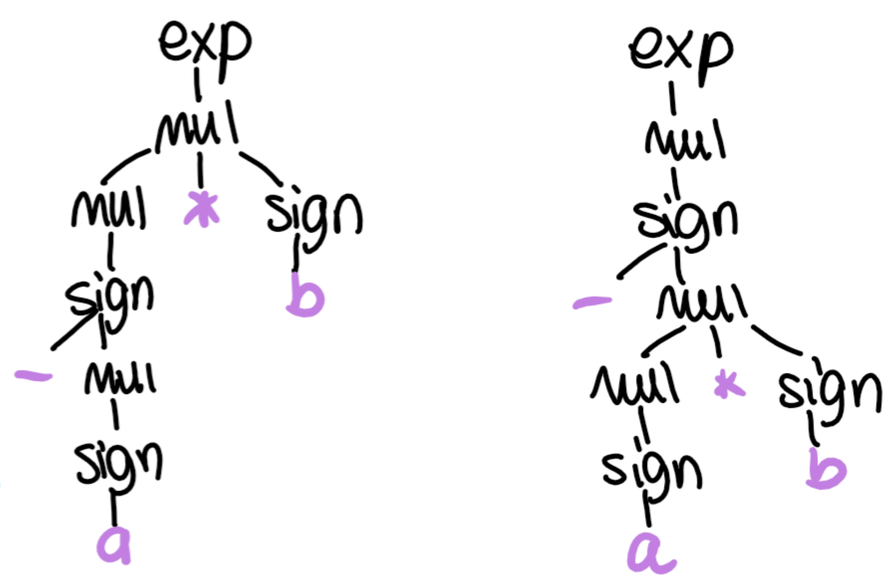
vi: non appartiene a nessun gruppo

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= Exp + Mul | Mul
Mul  ::=  Mul * Sign  | Sign
Sign ::= - Mul | a | b | ( Exp )
```

una grammatica è ambigua se per una stringa ci sono più alberi di derivazione

esempio: -a *b



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

```
Exp  ::= Exp + Mul | Mul
Mul  ::=  Mul * Sign  | Sign
Sign ::= - Sign | a | b | ( Exp )
```

→ 28 gennaio 2025

(a) Per ogni stringa elencata sotto stabilire, motivando la risposta, se appartiene al linguaggio definito dalla seguente espressione regolare e, in tal caso, indicare il gruppo di appartenenza (escludendo il gruppo 0).

```
([A-Z](?::?\w)*)|(<[^>]*>)|((0|[1-9]\d*)(\.(0|[1-9]\d*))*)
```

i. "<<a>>"
ii. "<<a>"
iii. "127.0.0.1"
iv. "212.001"
v. "A_:c"
vi. "A::"

i: l'unico gruppo che definisce un insieme con stringhe che iniziano con < è il secondo, ma non appartiene in quanto > può apparire solo una volta e alla fine della stringa

ii: appartiene al secondo gruppo perchè > è alla fine solo una volta

iii: appartiene al terzo gruppo

iv: non appartiene al terzo gruppo perchè .0 può essere seguito solo da un altro "."

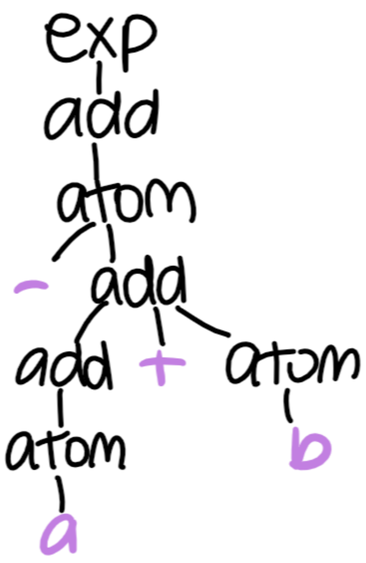
```
Exp  ::= Exp + Unary | Unary
Unary ::= ++ Unary | - Unary | Atom
Atom ::= a | b | ( Exp )
```

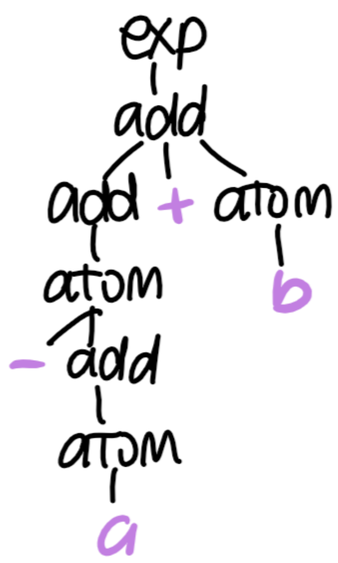
v: appartiene al primo gruppo

vi: non appartiene al primo gruppo perchè : deve essere necessariamente seguito da un \w → [a-z A-Z 0-9 _]

(b) Mostrare che la seguente grammatica è ambigua. $-a+b$

```
Exp ::= Exp * Add | Add
Add ::= Add + Atom  | Atom
Atom ::= - Add | a | b | ( Exp )
```





(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

```
Exp ::= Exp * Add | Add
Add ::= Add + Atom  | Atom
Atom ::= - Atom | a | b | ( Exp )
```

→ 15 gennaio 2025

(a) Per ogni stringa elencata sotto stabilire, motivando la risposta, se appartiene al linguaggio definito dalla seguente espressione regolare e, in tal caso, indicare il gruppo di appartenenza (escludendo il gruppo 0).

```
(/-(?:[^-]|-+[^-/])*-+/)|(0|[1-9]\d?\d?(?:,?\d\d\d)*)|(\w+(\.?\w)*)
```

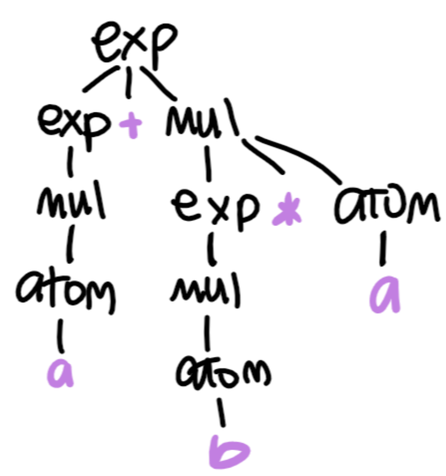
i. `"/--comment--/"`
ii. `"/---/-/"`
iii. `"100,000"`
iv. `"100000"`
v. `"Ab.c.d"`
vi. `"Ab..cde"`

i: la stringa appartiene al primo gruppo, perchè /- ... -+[^-/]→c ...[^-] →comment ...
ii: non appartiene a nessun gruppo
iii: appartiene al secondo gruppo ("1"∈[1-9], "00"∈ \d?\d? e ,"000"∈(?:,?\d\d\d)*)
iv: appartiene al secondo gruppo " "
v: appartiene al terzo gruppo Ab → \w+(\.?\w), .c e .d → \.?\w, ossia il punto . deve essere seguito da una lettera \w
vi: non appartiene al terzo gruppo perchè non è possibile la concatenazione di ..

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp ::= Exp + Mul | Mul
Mul ::=  Exp * Atom  | Atom
Atom ::= - Atom | a | b | ( Exp )
```

esempio: a+b*a



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

```
Exp ::= Exp + Mul | Mul
Mul ::=  Mul * Atom  | Atom
Atom ::= - Atom | a | b | ( Exp )
```

→ 10 Settembre 2024

(a) Per ogni stringa elencata sotto stabilire, motivando la risposta, se appartiene al linguaggio definito dalla seguente espressione regolare e, in tal caso, indicare il gruppo di appartenenza (escludendo il gruppo 0).

```
(\d+(?:[eE][-+]?\d+)?)|(\s+)|(\(\*(?:[^*]*|\*+[^)])*\*+\))
```

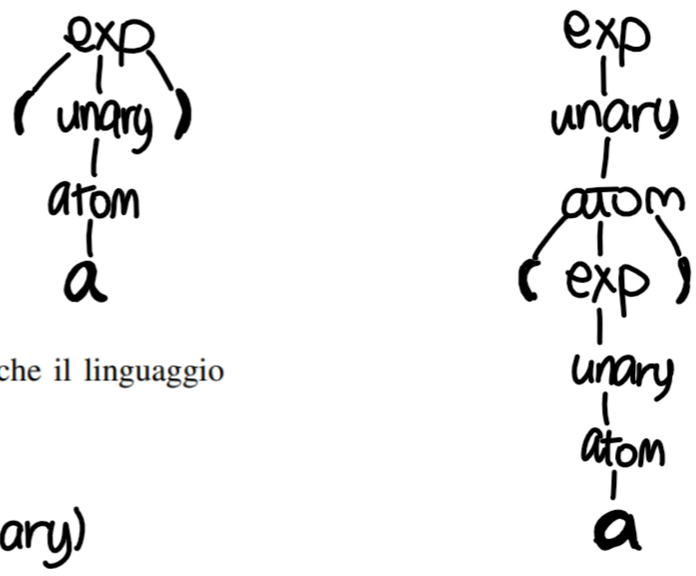
i. "(*****)"
ii. "(*)"
iii. "(**)*)"
iv. "0e+"
v. "000e+00"
vi. "000"

i: appartiene al terzo gruppo, perchè (* ∈ \(* e ****) ∈ *+\)
ii: non appartiene a nessun gruppo, perchè il terzo gruppo deve essere lunga almeno 4 elementi, ossia (**)
iii: non appartiene a nessun gruppo, perchè nel terzo gruppo *) ∉ *+[^)], con ^ che rappresenta che non ci può essere un) in mezzo alla stringa
iv: la stringa non appartiene al primo gruppo perchè deve necessariamente finire con un \d se presenta un eE
v: la stringa appartiene al primo gruppo, con 000 ∈ \d+, e+ ∈ [eE][-+]? e 00 ∈ \d+
vi: la stringa appartiene al primo gruppo perchè 000 ∈ \d+

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp ::= Exp * Unary | Unary | ( Unary )
Unary ::= Unary ++ | Unary -- | Atom
Atom ::= a | b | ( Exp )
```

es: (a)



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

```
Exp ::= Exp * Unary | Unary
Unary ::= Unary ++ | Unary -- | Atom
Atom ::= a | b | ( Exp )
```

elimino exp ::= (Unary)

→ 17 settembre 2024

(a) Per ogni stringa elencata sotto stabilire, motivando la risposta, se appartiene al linguaggio definito dalla seguente espressione regolare e, in tal caso, indicare il gruppo di appartenenza (escludendo il gruppo 0).

```
(\d+(?:[eE]-?\d+)?)|(/[*](?:[^*]*|[*]+[^/])*[*]+/)|(\s+)
```

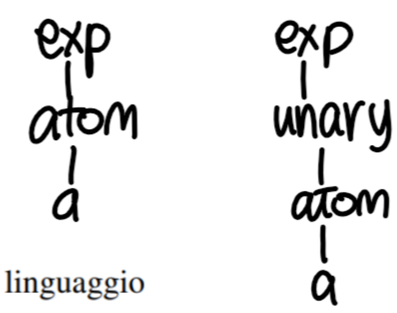
i. "/*/"
ii. "/**/*/"
iii. "/***/"
iv. "314E-02"
v. "3E"
vi. "3-0"

i: non appartiene a nessun gruppo, perchè nel secondo le stringhe devono iniziare con "/*" e finire con "*/"
ii: non appartiene al secondo gruppo perchè la stringa non può avere il carattere / nel mezzo (della stringa)
iii: appartiene al secondo gruppo
iv: appartiene al primo gruppo
v: non appartiene al primo gruppo perchè la stringa deve finire con una cifra decimale
vi: non appartiene al primo gruppo perchè - deve essere preceduto da e o E

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp ::= Exp + Unary | Unary | Atom
Unary ::= ++ Unary | -- Unary | Atom
Atom ::= a | b | ( Exp )
```

esempio: a



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

```
Exp ::= Exp + Unary | Unary
Unary ::= ++ Unary | -- Unary | Atom
Atom ::= a | b | ( Exp )
```

rimuovo Exp::= Atom

→ 21 Giugno 2024

(a) Per ogni stringa elencata sotto stabilire, motivando la risposta, se appartiene al linguaggio definito dalla seguente espressione regolare e, in tal caso, indicare il gruppo di appartenenza (escludendo il gruppo 0).

```
((?:/?[a-zA-Z])*)|(\s+)|(\d*(?:\.?\d|\d\.?)\d*)
```

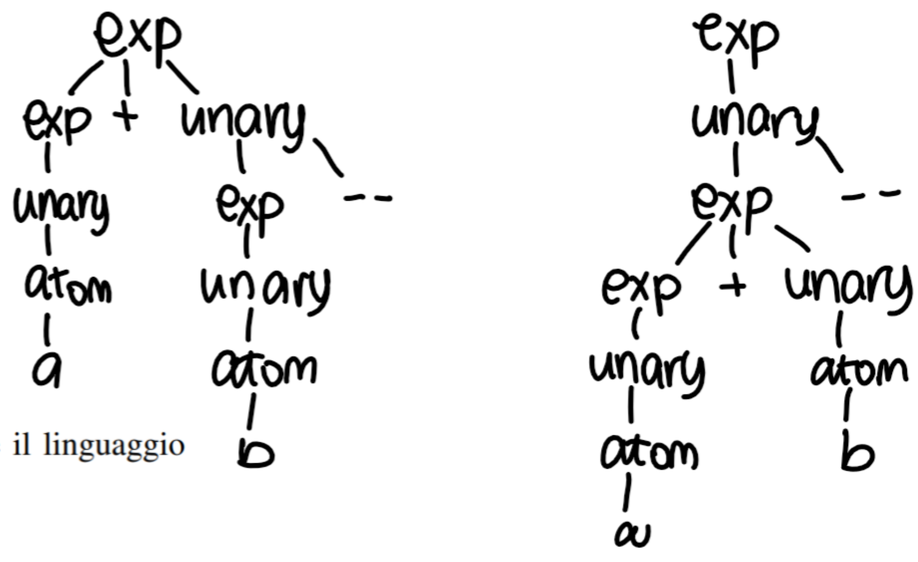
i. "/a/b"
ii. "a/b"
iii. "a/b/"
iv. "10."
v. ".01"
vi. "."

i: appartiene al primo gruppo, dove / deve essere seguito da una lettera
ii: appartiene al primo gruppo
iii: non appartiene al primo gruppo perchè il secondo / non è seguito da una lettera
iv: appartiene al terzo gruppo e ricadiamo nel caso \d\.?, con \d* → d da 0 a n volte
v: appartiene al terzo gruppo e ricadiamo nel caso \.?\d
vi: non appartiene al terzo gruppo perchè "." deve essere almeno seguito o preceduto da un \d

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp ::= Exp + Unary | Unary
Unary ::= Unary ++ | Exp -- | Atom
Atom ::= a | b | ( Exp )
```

a+b--



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

```
Exp ::= Exp + Unary | Unary
Unary ::= Unary ++ | Unary -- | Atom
Atom ::= a | b | ( Exp )
```

→ 23 Febbraio 2024

(a) Per ogni stringa elencata sotto stabilire, motivando la risposta, se appartiene al linguaggio definito dalla seguente espressione regolare e, in tal caso, indicare il gruppo di appartenenza (escludendo il gruppo 0).

```
(\s+)|([a-zA-Z_](?:\.?\w)*)|(\d(_*\d)*)
```

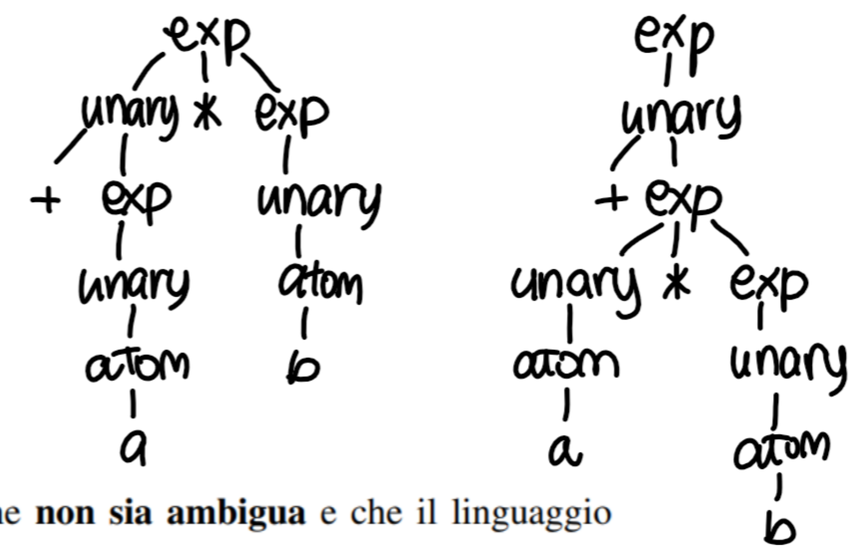
i. " a "
ii. "000"
iii. "0_"
iv. "00___00"
v. "a0.0b23"
vi. "a0..0b23"

i: non appartiene al primo gruppo perche "a" non è uno spazio
ii: appartiene al terzo gruppo
iii: non appartiene al terzo gruppo perchè la stringa deve finire con una cifra decimale
iv: appartiene al terzo gruppo
v: appartiene al secondo gruppo
vi: non appartiene al secondo gruppo perchè "." deve essere seguito da una /w, quindi non può esserci ..

Mostrare che la seguente grammatica è ambigua.

```
Exp ::= Unary * Exp | Unary
Unary ::= - Unary | + Exp | Atom
Atom ::= a | b | ( Exp )
```

esempio: +a*b



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

```
Exp ::= Unary * Exp | Unary
Unary ::= - Unary | + Unary | Atom
Atom ::= a | b | ( Exp )
```

→ 8 febbraio 2024

(a) Per ogni stringa elencata sotto stabilire, motivando la risposta, se appartiene alla seguente espressione regolare e, in caso affermativo, indicare il gruppo di appartenenza (escludendo il gruppo 0).

```
([@a-zA-Z]\w*)|(\s+)|(0(?:,?[0-7])*)
```

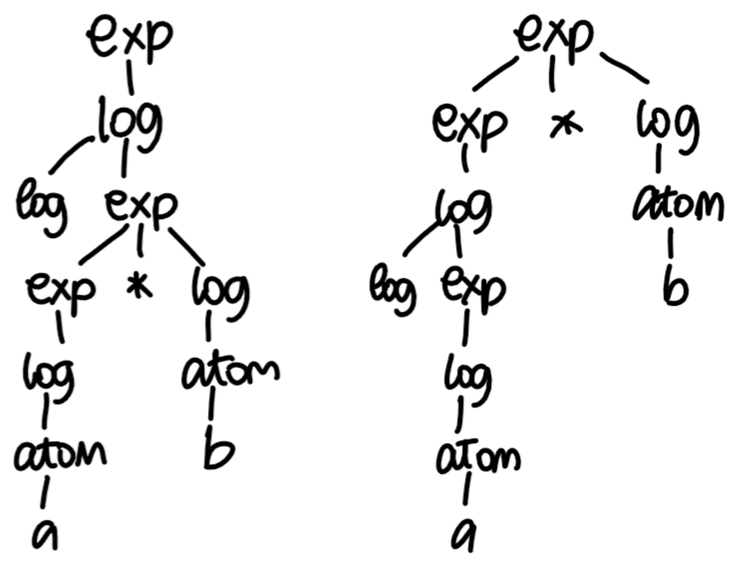
i. "07"
ii. "0,7,7"
iii. "0123,347"
iv. "_____@"
v. "@_____@b"
vi. "@a_____b"

i: la stringa appartiene al terzo gruppo
ii: la stringa appartiene al terzo gruppo, con la virgola "," seguita necessariamente da una cifra ottale
iii: appartiene al terzo gruppo
iv: non appartiene a nessun gruppo, in quanto inizia con "_"
v: non appartiene al primo gruppo perchè "@" può comparire solo come primo carattere
vi: appartiene al primo gruppo, con @ ∈ [@a-z A-Z] e a___b ∈ \w → [a-z A-Z _]

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp ::= Exp / Log | Exp * Log | Log
Log ::= log Exp | Atom
Atom ::= a | b | ( Exp )
```

esempio: log a*b



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

```
Exp ::= Exp / Log | Exp * Log | Log
Log ::= log Log | Atom
Atom ::= a | b | ( Exp )
```

→ 23 gennaio 2024

(a) Per ogni stringa elencata sotto stabilire, motivando la risposta, se appartiene alla seguente espressione regolare e, in caso affermativo, indicare il gruppo di appartenenza (escludendo il gruppo 0).

```
(0[xX][a-fA-F\d](?:_*[a-fA-F\d])*)|([$a-zA-Z]\w*)|(\s+)
```

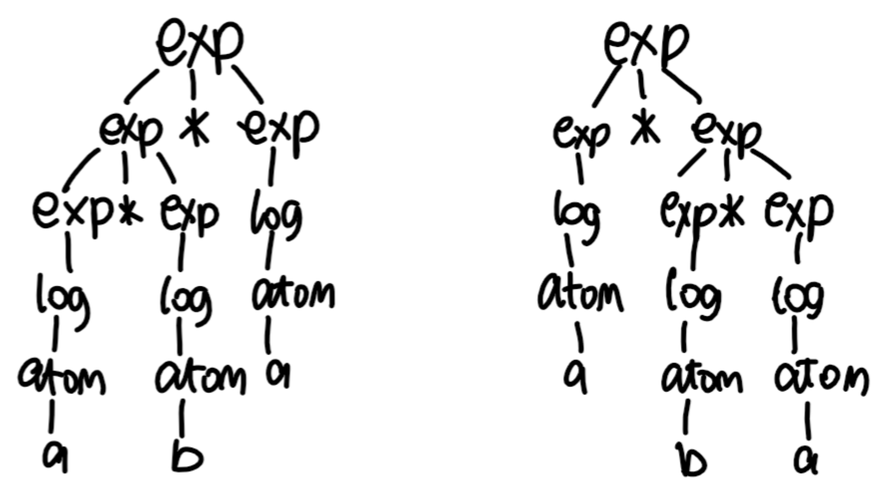
i. "0X_C"
ii. "0XC_"
iii. "0XC__FF__00"
iv. "$__"
v. "a$b"
vi. " "

i: non appartiene al primo gruppo perchè dopo la X ci deve essere [a-f A-F \d]
ii: non appartiene al primo gruppo perchè dopo _ deve esserci [a-f A-F \d]
iii: appartiene al primo gruppo
iv: la stringa appartiene al secondo gruppo, con $ ∈ [$a-z A-Z] e _ ∈ \w
v: $ può essere presente solo come primo elemento, quindi non appartiene al secondo gruppo
vi: appartiene al terzo gruppo (spazi bianchi)

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp ::= Exp / Log | Exp * Exp | Log
Log ::= log Log | Atom
Atom ::= a | b | ( Exp )
```

a*b*a



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

```
Exp ::= Exp / Log | Exp * Log | Log
Log ::= log Log | Atom
Atom ::= a | b | ( Exp )
```

→ Forzare l'associatività dell'operatore a sinistra

→ 20 settembre 2023

(a) Per ogni stringa elencata sotto stabilire, motivando la risposta, se appartiene alla seguente espressione regolare e, in caso affermativo, indicare il gruppo di appartenenza (escludendo il gruppo 0).

```
((?:_|[a-zA-Z])['\w]*)|(0[bB][01](?:_*[01])*)|(\s+)
```

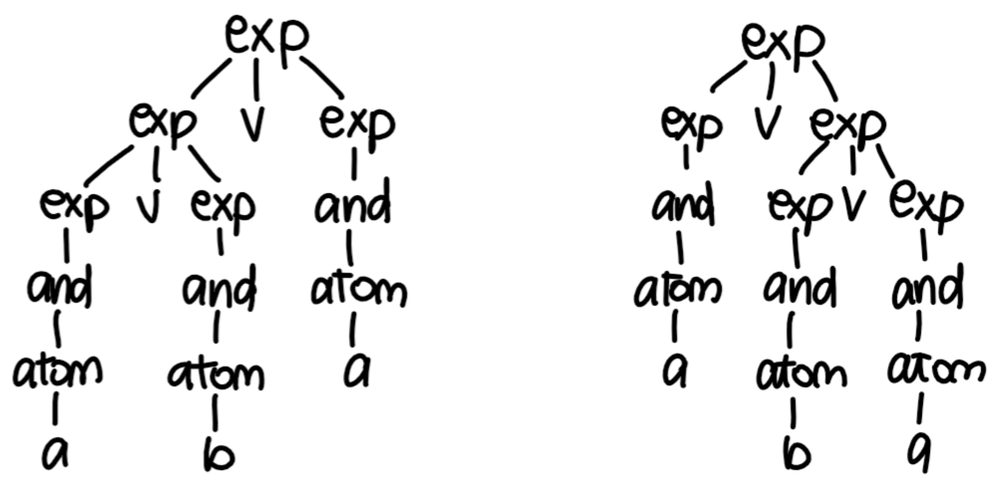
i. "0b_1"
ii. "0b1_"
iii. "0b000_1_0"
iv. "_'"
v. "a'b_'"
vi. "0b0 a3"

i: non appartiene al secondo gruppo poichè dopo la lettera b deve seguire uno 0 o 1
ii: non appartiene al secondo gruppo perchè dopo "_" deve seguire 0 o 1
iii: appartiene al secondo gruppo
iv: appartiene al primo gruppo
v: appartiene al primo gruppo
vi: non appartiene a nessun gruppo perchè nessuno può contenere caratteri E spazi bianchi

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp ::= Exp ∨ Exp | And
And ::= And ∧ And | Atom
Atom ::= a | b | ( Exp )
```

esempio: a∨b∨a



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

```
Exp ::= Exp ∨ And | And
And ::= And ∧ Atom | Atom
Atom ::= a | b | ( Exp )
```

→ forzare l'associatività

→ 10 luglio 2023

(a) Per ogni stringa elencata sotto stabilire, motivando la risposta, se appartiene alla seguente espressione regolare e, in caso affermativo, indicare il gruppo di appartenenza (escludendo il gruppo 0).

```
(\{[^\}]*\})|(\d\d/\d\d)|(\w+(\.\w+)*)
```

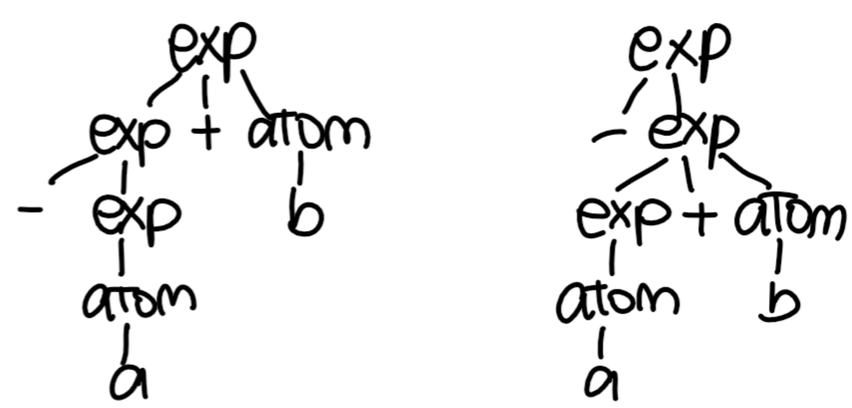
i. "_a."
ii. "_a._"
iii. "12/02"
iv. "12/3"
v. "{}"
vi. "{,}"

i: non appartiene al terzo gruppo perchè "." deve essere seguito da un \w
ii: appartiene al terzo gruppo
iii: appartiene al secondo gruppo
iv: non appartiene al secondo gruppo perchè "/" deve essere preceduto e seguito da 2 cifre
v: appartiene al primo gruppo
vi: appartiene al primo gruppo perchè basta che tra le due {} non ci sia il carattere }

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp ::= Exp + Atom | - Exp | Atom
Atom ::= a | b | ( Exp )
```

esempio: -a+b



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

```
Exp ::= Exp + Minus | Minus
Minus ::= - Minus | Atom
Atom ::= a | b | ( Exp )
```

→ 20 giugno 2023

(a) Per ogni stringa elencata sotto stabilire, motivando la risposta, se appartiene alla seguente espressione regolare e, in caso affermativo, indicare il gruppo di appartenenza (escludendo il gruppo 0).

```
(re:\d\+im:\d+)|([A-Z]\w*)|(<\d+(,\d+)*>)
```

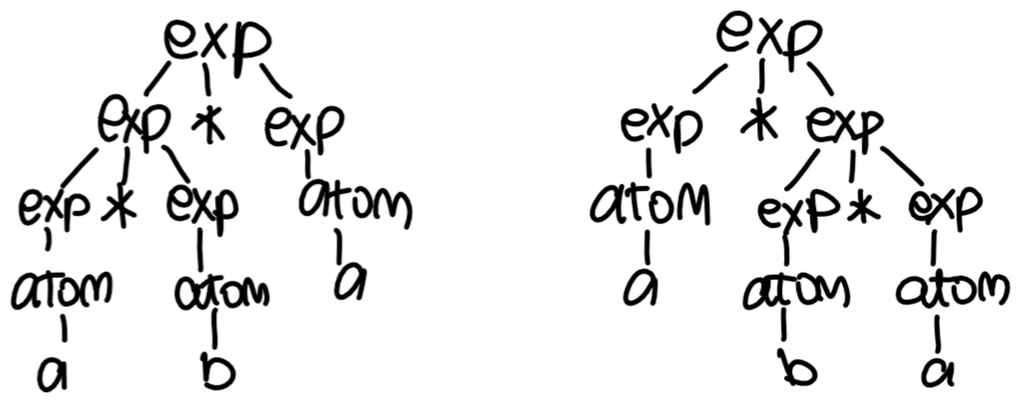
i. `"<1,2,3>"`
ii. `"<1,>"`
iii. `"Class"`
iv. `"MAX_VAL"`
v. `"re:2"`
vi. `"re:1+im:3"`

i: la stringa appartiene al terzo gruppo
ii: non appartiene al terzo gruppo perche ",' deve essere seguito da una o più cifre
iii: appartiene al secondo gruppo
iv: appartiene al secondo gruppo
v: non appartiene al primo gruppo perchè dopo il 2 deve esserci "+im:"
vi: appartiene al primo gruppo

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp ::= Exp + Atom  | Exp * Exp | Atom
Atom ::= a | b | ( Exp )
```

esempio: a*b*a



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

```
Exp ::= Exp + Atom  | Exp * Atom | Atom
Atom ::= a | b | ( Exp )
```

→ 20 febbraio 2023

(a) Per ogni stringa elencata sotto stabilire, motivando la risposta, se appartiene alla seguente espressione regolare e, in caso affermativo, indicare il gruppo di appartenenza (escludendo il gruppo 0).

```
(<[^<>]*>)|([A-Za-z][A-Za-z\d]*)|(\d+[,](\d*[jJ]))
```

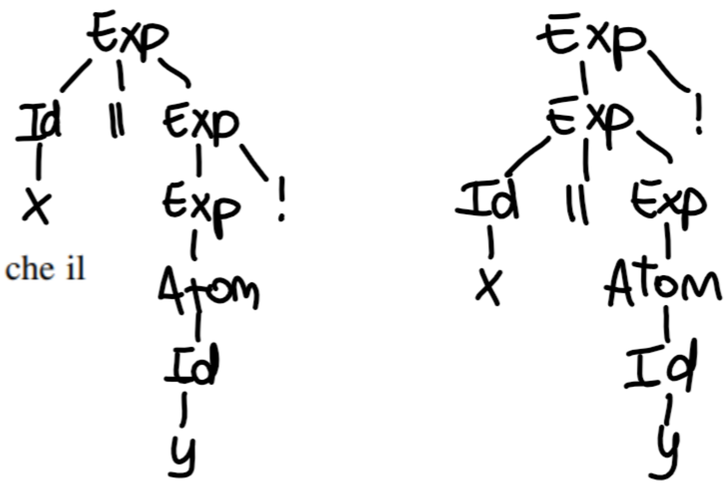
i. `<a>>`
ii. `<a?>`
iii. `a2`
iv. `A_b2`
v. `1,2J`
vi. `,09j`

i: non appartiene al primo gruppo perchè il carattere ">" può essere presente solo alla fine
ii: appartiene al primo gruppo
iii: appartiene al secondo gruppo, con a ∈ [A-Za-z] e 2 ∈ [A-Za-z\d]
iv: non appartiene al secondo gruppo perchè il carattere "_" non può essere presente
v: appartiene al terzo gruppo
vi: non appartiene a nessun gruppo, perchè inizia con ","

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp ::= Exp !  | Id || Exp | Atom
Atom ::= Id | ( Exp )
Id ::= x | y
```

esempio: x||y!



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

```
Exp ::= Bang | Id || Exp
Bang ::= Bang !  | Atom
Atom ::= Id | ( Exp )
Id ::= x | y
```

→ 13 febbraio 2023

(a) Per ogni stringa elencata sotto stabilire, motivando la risposta, se appartiene alla seguente espressione regolare e, in caso affermativo, indicare il gruppo di appartenenza (escludendo il gruppo 0).

```
(\d+[jJ])|([A-Z]\w*)|(-[^-]*-)
```

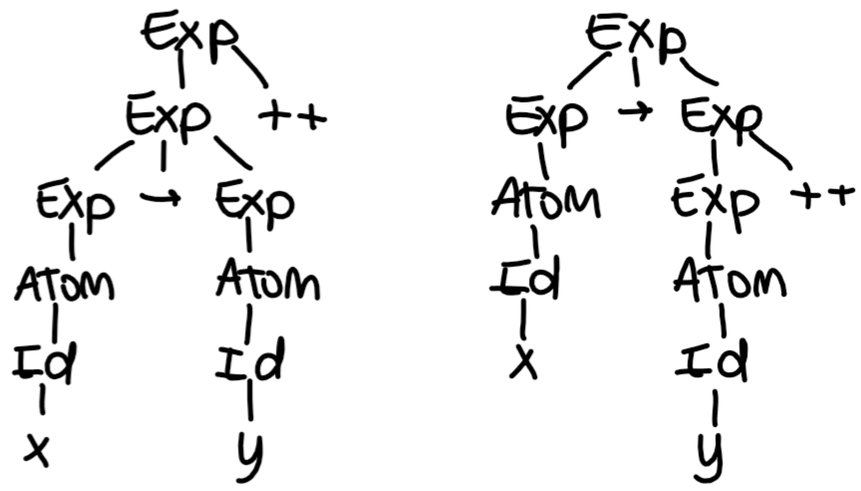
i. `-a--`
ii. `-a?-`
iii. `a2`
iv. `A_b2`
v. `0jJ`
vi. `09j`

i: la stringa non appartiene al terzo gruppo perchè il carattere "-" deve essere presente solo all'inizio e alla fine della stringa
ii: appartiene al terzo gruppo
iii: non appartiene a nessun gruppo perchè inizia con il carattere "a"
iv: appartiene al secondo gruppo, con "A" ∈ [A-Z] e "_b2" ∈ \w*
v: non appartiene al primo gruppo perchè \d deve essere seguita solo da j o J, non entrambe
vi: appartiene al primo gruppo, con "09" ∈ \d+ e "j" ∈ [jJ]

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp ::= Exp ++ | Exp -> Exp | Atom
Atom ::= Id | ( Exp )
Id ::= x | y
```

esempio: x → y ++



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

```
Exp ::= Inc -> Exp | Inc
Inc ::= Inc ++ | Atom
Atom ::= Id | ( Exp )
Id ::= x | y
```

→ 19 gennaio 2023

(a) Per ogni stringa elencata sotto stabilire, motivando la risposta, se appartiene alla seguente espressione regolare e, in caso affermativo, indicare il gruppo di appartenenza (escludendo il gruppo 0).

```
([A-Za-z]\w*)|(\s+)|(0[0-7]*([lL]?))
```

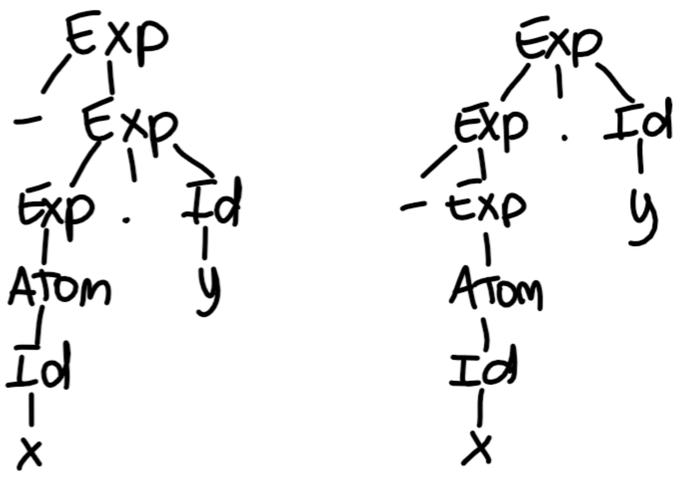
i. `a\w`
ii. `a2`
iii. `a_2`
iv. `00`
v. `0L`
vi. `08`

i: non appartiene al primo gruppo, perchè con \w si intende [A-Z a-z _ 0-9]
ii: appartiene al primo gruppo
iii: appartiene al primo gruppo
iv: appartiene al terzo gruppo, con 00 ∈ 0[0-7]* e [lL]? vuol dire che è opzionale
v: appartiene al terzo gruppo, con 0, [0-7]* vuol dire stringa anche vuota e L
vi: non appartiene al terzo gruppo perchè la seconda cifra può essere da 0 a 7

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp ::= - Exp | Exp . Id | Atom
Atom ::= Id | ( Exp )
Id ::= x | y
```

esempio: - x . y



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

```
Exp ::= - Exp | Sel
Sel ::= Sel . Id | Atom
Atom ::= Id | ( Exp )
Id ::= x | y
```

→ 7 settembre 2022

(a) Per ogni stringa elencata sotto stabilire, motivando la risposta, se appartiene alla seguente espressione regolare e, in caso affermativo, indicare il gruppo di appartenenza (escludendo il gruppo 0).

```
(\s+)|([A-Za-z][_a-zA-Z\d]*)|(\d+(?:[eE]\d\d))?
```

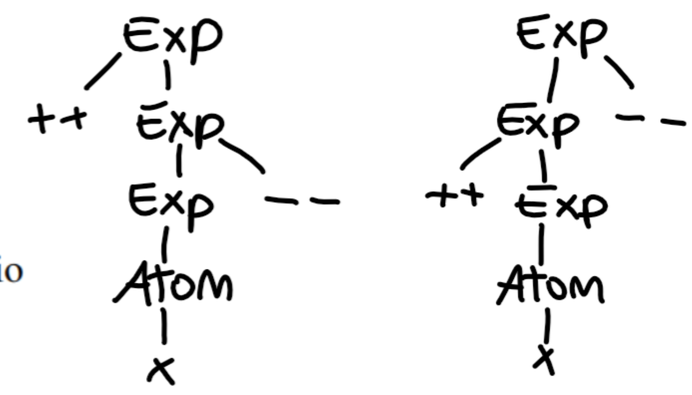
i. "E42"
ii. "42E42"
iii. "4E2"
iv. "e___"
v. "4Ee42"
vi. "E 4"

i: appartiene al secondo gruppo
ii: appartiene al terzo gruppo, con "42" ∈ \d+ e "E42" ∈ (?:[eE]\d\d)
iii: non appartiene al terzo gruppo perchè E è seguito da una sola cifra numerica
iv: appartiene al secondo gruppo, con "e" ∈ [A-Z a-z] e "___" ∈ [_a-z A-Z \d]
vi: non appartiene al secondo gruppo perchè non possono esserci spazi bianchi

Mostrare che la seguente grammatica è ambigua.

esempio: ++ x --

```
Exp ::= ++ Exp | Exp -- | Atom
Atom ::= x | y | ( Exp )
```



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

```
Exp ::= Exp -- | Inc
Inc ::= ++ Inc | Atom
Atom ::= x | y | ( Exp )
```

→ 11 luglio 2022

(a) Per ogni stringa elencata sotto stabilire, motivando la risposta, se appartiene alla seguente espressione regolare e, in caso affermativo, indicare il gruppo di appartenenza (escludendo il gruppo 0).

```
([A-Za-z](?:_*[a-zA-Z\d])*)|((?:0|[1-9]+)L?)|(\s+)
```

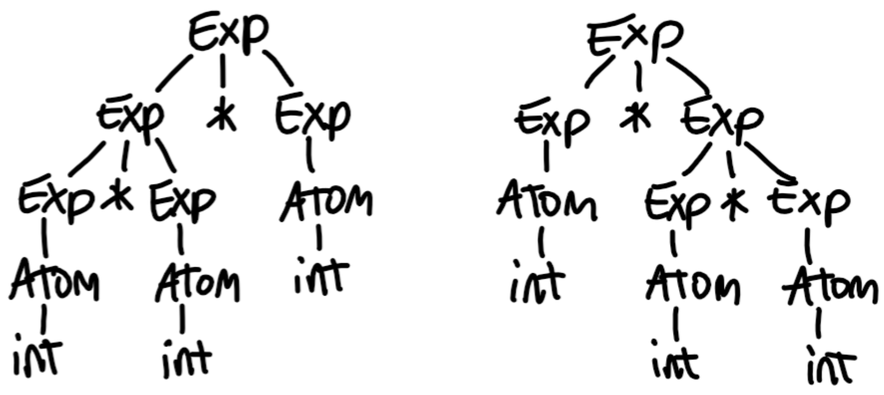
i. "A___b"
ii. "_a"
iii. "a_2_b"
iv. "a_"
v. "0L"
vi. "L"

i: appartiene al primo gruppo
ii: non appartiene a nessun gruppo perchè inizia con "_"
iii: appartiene al primo gruppo
iv: la stringa di caratteri "_" deve essere seguita da [a-z A-Z \d], quindi non appartiene al primo gruppo
v: appartiene al secondo gruppo
vi: appartiene al primo gruppo

(b) Mostrare che la seguente grammatica è ambigua.

esempio: int * int * int

```
Exp ::= Exp [ ] | Exp * Exp | Atom
Atom ::= int | bool | ( Exp )
```



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

```
Exp ::= Exp * Bracket | Bracket
Bracket ::= Bracket [ ] | Atom
Atom ::= int | bool | ( Exp )
```

→ 20 giugno 2022

(a) Per ogni stringa elencata sotto stabilire, motivando la risposta, se appartiene alla seguente espressione regolare e, in caso affermativo, indicare il gruppo di appartenenza (escludendo il gruppo 0).

```
(\s+)|([A-Z][_a-zA-Z\d]*)|([1-9](?:_*[0-9])*|0)
```

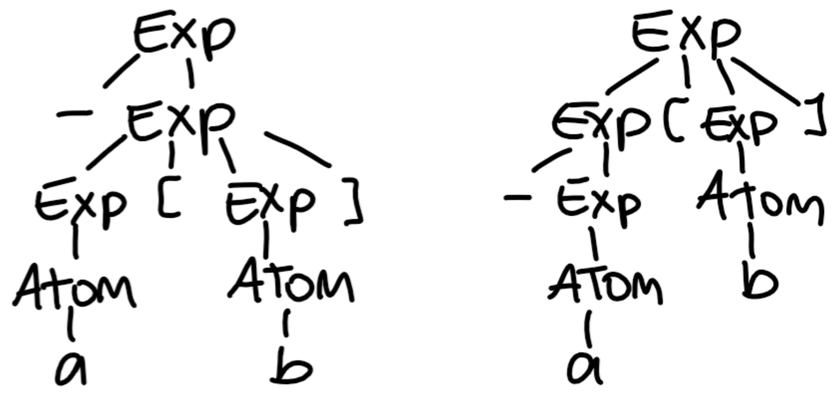
i. "1___2"
ii. "0___2"
iii. "42_"
iv. "A_2"
v. "_A2"
vi. "Acc"

i: appartiene al terzo gruppo
ii: non appartiene al terzo gruppo perchè se la stringa inizia con "0" non può essere seguita da altri caratteri
iii: "_" deve essere seguito da una cifra, quindi non appartiene al terzo gruppo
iv: la stringa appartiene al secondo gruppo
v: non appartiene al secondo gruppo perchè inizia con "_"
vi: appartiene al secondo gruppo

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp ::= Exp [ Exp ] | - Exp | Atom
Atom ::= a | b | ( Exp )
```

esempio: -a[b]



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

```
Exp ::= - Exp | Bracket
Bracket ::= Bracket [ Exp ] | Atom
Atom ::= a | b | ( Exp )
```

→ 16 febbraio 2022

(a) Per ogni stringa elencata sotto stabilire, motivando la risposta, se appartiene alla seguente espressione regolare e, in caso affermativo, indicare il gruppo di appartenenza (escludendo il gruppo 0).
Nota bene: la notazione `(?: )` permette di usare le parentesi in un'espressione regolare **senza** definire un nuovo gruppo.

```
(0[bB](?:[0-1]*|[0-1]+[lL]?))|(\s+)|([a-z]+(?:\.[a-zA-Z\d]*)*)
```

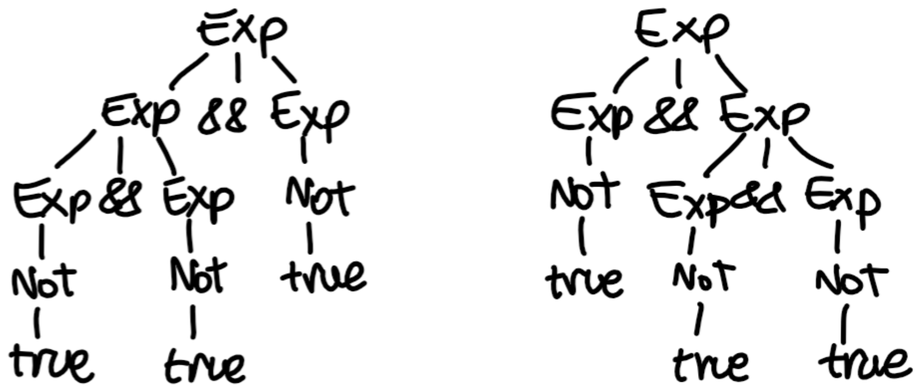
i. "0B"
ii. "0bL"
iii. "a.b"
iv. "a."
v. "a.A3.."
vi. "aA."

i: appartiene al primo gruppo
ii: non appartiene al primo gruppo in quanto L deve seguire uno 0 o un 1
iii: appartiene al terzo gruppo
iv: appartiene al terzo gruppo
v: appartiene al terzo gruppo
vi: non appartiene al terzo gruppo perchè prima di "." possono esserci solo [a-z]

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp ::= Exp && Exp | Not
Not ::= false | true | ! Not | ( Exp )
```

esempio: true && true && true



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

```
Exp ::= Exp && Not | Not
Not ::= false | true | ! Not | ( Exp )
```

→ 10 gennaio 2022

(a) Per ogni stringa elencata sotto stabilire, motivando la risposta, se appartiene alla seguente espressione regolare e, in caso affermativo, indicare il gruppo di appartenenza (escludendo il gruppo 0).
Nota bene: la notazione `(?: )` permette di usare le parentesi in un'espressione regolare **senza** definire un nuovo gruppo.

```
([a-z]+(?:\.[a-z]*)*)|(0[bB][0-1]+[lL]?)|(\s+)|(mv\.ACC|add\.SP|sub\.FM)
```

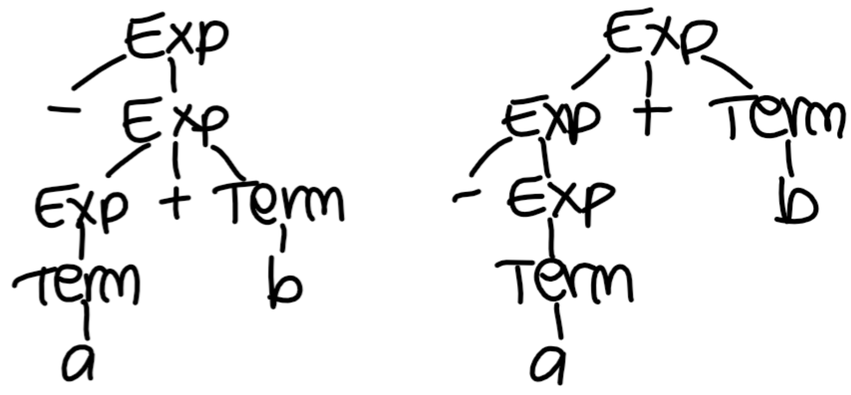
i. `"mv.ACC"`
ii. `"MV.acc"`
iii. `"0B"`
iv. `"b."`
v. `"0BL"`
vi. `"0B10L"`

i: appartiene al quarto gruppo
ii: non appartiene a nessun gruppo perchè parte con "MV"
iii: non appartiene al secondo gruppo perchè B deve essere seguito da 0 o 1
iv: appartiene al primo gruppo
v: non appartiene al secondo gruppo
vi: appartiene al secondo gruppo, con "0B" ∈ 0[bB], "01" ∈ [0-1]+ e "L" ∈ [lL]?

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp ::= Term | Exp + Term | - Exp
Term ::= a | b | - Term | ( Exp )
```

esempio: -a+b



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

```
Exp ::= Term | Exp + Term
Term ::= a | b | - Term | ( Exp )
```

→ elimino Exp::= -Exp

→ 8 settembre 2021

(a) Per ogni stringa elencata sotto stabilire, motivando la risposta, se appartiene alla seguente espressione regolare e, in caso affermativo, indicare il gruppo di appartenenza (escludendo il gruppo 0).

```
([a-z][a-z0-9]*(?:\.[a-z][a-z0-9]*)*)|([0-9]+(?:\.[0-9]*)?)|(\s+)
```

Nota bene: la notazione `(?: )` permette di usare le parentesi in un'espressione regolare **senza** definire un nuovo gruppo.

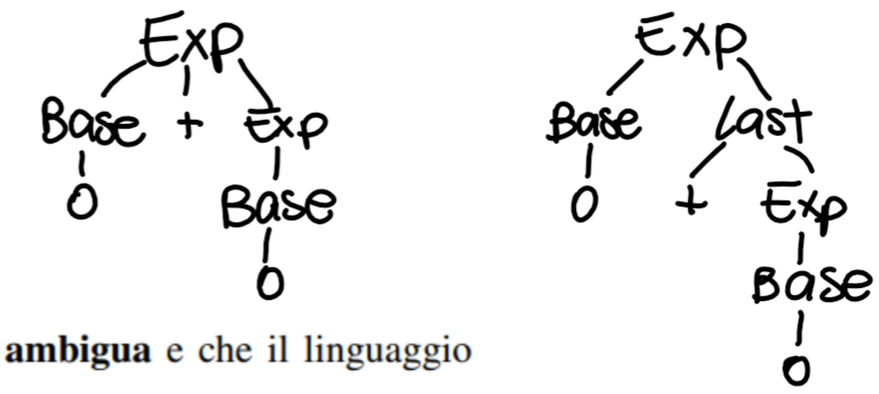
i. `"a.0"`
ii. `".4"`
iii. `"42."`
iv. `"a2.b3"`
v. `"xy."`
vi. `" "`

i: non appartiene al primo gruppo perchè dopo "." deve seguire necessariamente [a-z]
ii: non appartiene a nessun gruppo perchè inizia con "."
iii: appartiene al secondo gruppo, con "42" ∈ [0-9]+ e con il punto che viene seguito da una stringa di numeri che può essere anche vuota
iv: appartiene al primo gruppo
v: non appartiene al primo gruppo perchè dopo "." non è presente nessun carattere
vi: appartiene al terzo gruppo

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp ::= Base + Exp | Base Last | Base
Last ::= + Exp
Base ::= 0 | 1 | ( Exp )
```

esempio: 0+0



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

```
Exp ::= Base + Exp | Base
Base ::= 0 | 1 | ( Exp )
```

→ 12 luglio 2021

(a) Per ogni stringa elencata sotto stabilire, motivando la risposta, se appartiene alla seguente espressione regolare e, in caso affermativo, indicare il gruppo di appartenenza (escludendo il gruppo 0).

```
([0-9]+)|([A-Z][a-zA-Z0-9]*)|(move|store|load)|(\s+)
```

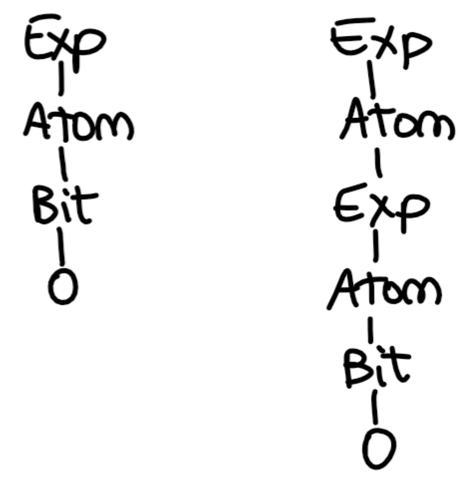
i. "Move"
ii. "move42"
iii. "store "
iv. "000"
v. "0a"
vi. " "

i: appartiene al secondo gruppo
ii: non appartiene a nessun gruppo: nel secondo gruppo la stringa deve iniziare con una lettera maiuscola, mentre nel terzo non possono esserci cifre
iii: non appartiene al terzo gruppo perchè non può contenere spazi bianchi
iv: appartiene al primo gruppo
v: non appartiene a nessun gruppo, perchè nessun gruppo permette di iniziare la stringa con "0" e continuare con una lettera minuscola
vi: appartiene al quarto gruppo

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp ::= Atom + Exp | Atom - Exp | Atom
Atom ::=  Bit | ( Exp ) | Exp
Bit ::= 0 | 1
```

esempio: 0



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

```
Exp ::= Atom + Exp | Atom - Exp | Atom
Atom ::=  Bit | ( Exp )
Bit ::= 0 | 1
```

→ elimino Atom ::= Exp

→ 21 giugno 2021

(a) Per ogni stringa elencata sotto stabilire, motivando la risposta, se appartiene alla seguente espressione regolare e, in caso affermativo, indicare il gruppo di appartenenza (escludendo il gruppo 0).

```
(\s+)|(0[0-7]*)|(0x1|0x2|0x3)|([0-9][a-zA-Z]+)
```

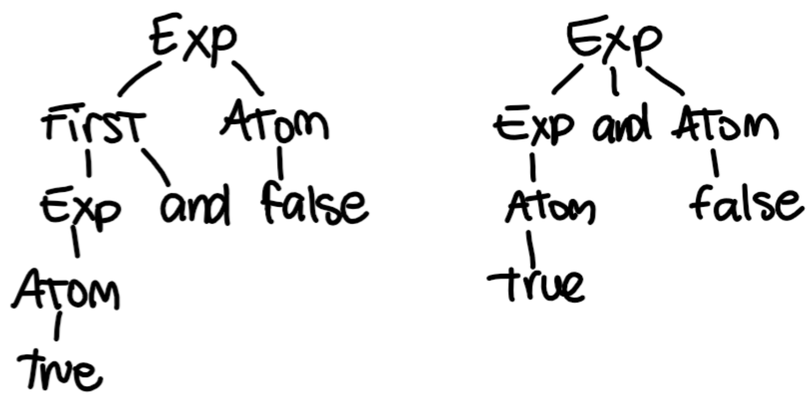
i. "0x4"
ii. "0 0"
iii. "0x"
iv. "0x1"
v. "001"
vi. "0a"

i: non appartiene a nessun gruppo: nè il terzo nè il quarto possono finire con 4
ii: non appartiene a nessun gruppo
iii: appartiene al quarto gruppo, con "0" ∈ [0-9] e "x" ∈ [a-zA-Z]+
iv: appartiene al terzo gruppo
v: appartiene al secondo gruppo
vi: appartiene al quarto gruppo

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp ::= Exp and Atom | First Atom | Atom
First ::= Exp and
Atom ::= false | true | ( Exp )
```

esempio: true and false



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

```
Exp ::= Exp and Atom | Atom
Atom ::= false | true | ( Exp )
```

→ elimino First

→ 3 giugno 2021

(a) Per ogni stringa elencata sotto stabilire, motivando la risposta, se appartiene alla seguente espressione regolare e, in caso affermativo, indicare il gruppo di appartenenza (escludendo il gruppo 0).

```
(0[0-7]+[lL]?)|([A-Za-z]+(?:\.[A-Za-z]+)*)|(\s+)|(\.mv|\.add|\.sub)
```

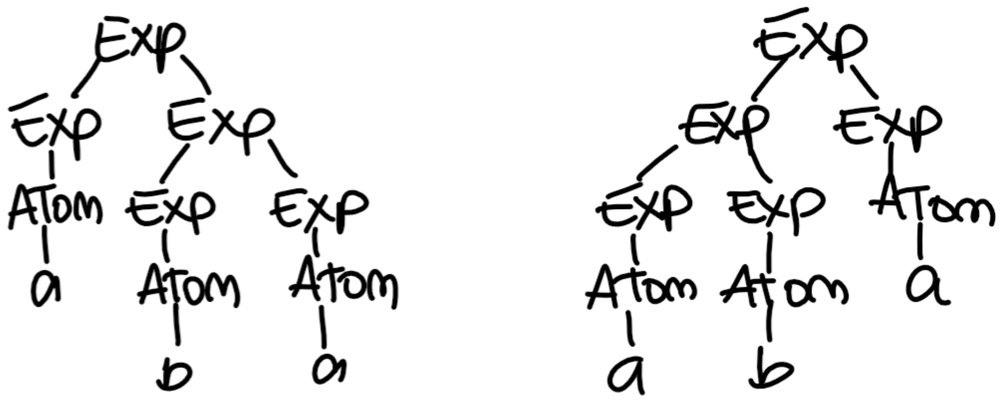
i. "a.b7"
ii. "a."
iii. "mv.sub"
iv. ".sub"
v. "0L"
vi. "00L"

i: non appartiene al secondo gruppo perchè non ammette cifre
ii: non appartiene al secondo gruppo perchè dopo il "." deve seguire almeno una lettera
iii: appartiene al secondo gruppo
iv: appartiene al quarto gruppo
v: non appartiene al primo gruppo perchè dopo il primo 0 deve esserci almeno un [0-7]
vi: appartiene al primo gruppo

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp ::= Exp Exp | Atom
Atom ::= a | b | ( Exp )
```

esempio: a b a



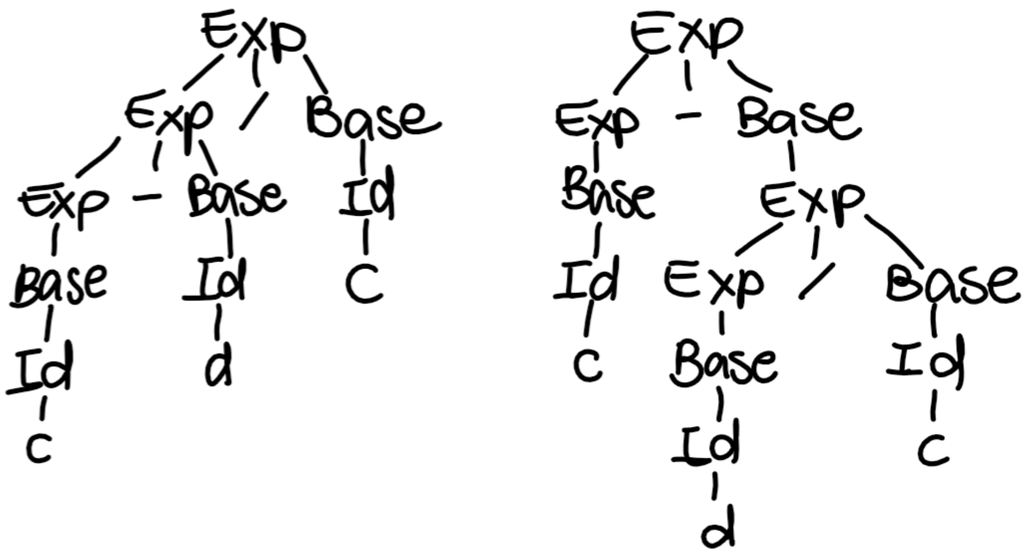
(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

```
Exp ::= Exp Atom | Atom
Atom ::= a | b | ( Exp )
```

→ 18 febbraio 2020

(b) Mostrare che nella seguente grammatica la stringa `c-d/c` è ambigua rispetto a `Exp`.

```
Exp ::= Exp / Base | Exp - Base | Base
Base ::= Id | { Exp } | Exp
Id ::= c | d
```



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

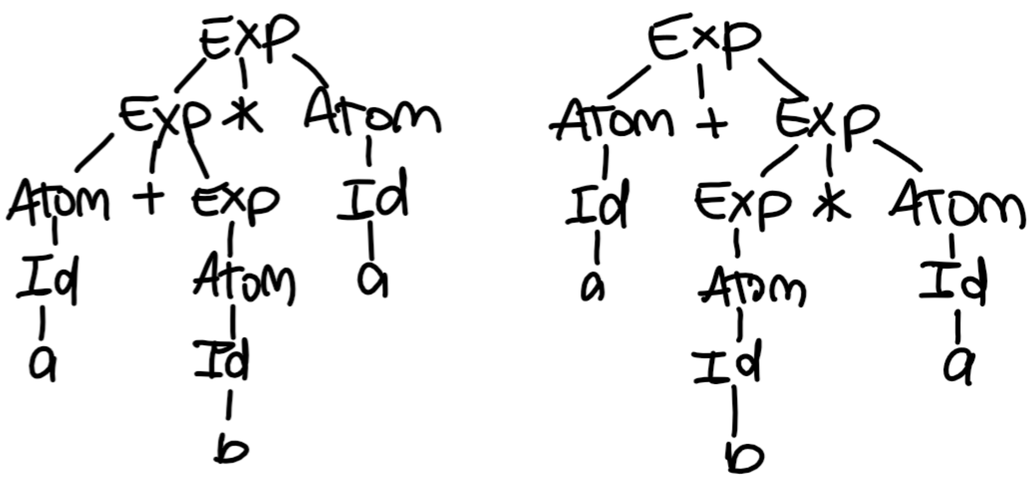
```
Exp ::= Exp / Base | Exp - Base | Base
Base ::= Id | { Exp }
Id ::= c | d
```

→ elimino Base ::= Exp

→ 23 gennaio 2020

(b) Mostrare che nella seguente grammatica la stringa `a+b*a` è ambigua rispetto a `Exp`.

```
Exp ::= Exp * Atom | Atom + Exp | Atom
Atom ::= ( Exp ) | Id
Id ::= a | b
```



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

```
Exp ::= Exp * Atom | Exp + Atom | Atom
Atom ::= ( Exp ) | Id
Id ::= a | b
```

→ 9 settembre 2019

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp ::= Exp * Atom | Exp + Atom | Atom
Atom ::= [ ExpSeq ] | Id
ExpSeq ::= ExpSeq Exp | Exp
Id ::= a | b
```

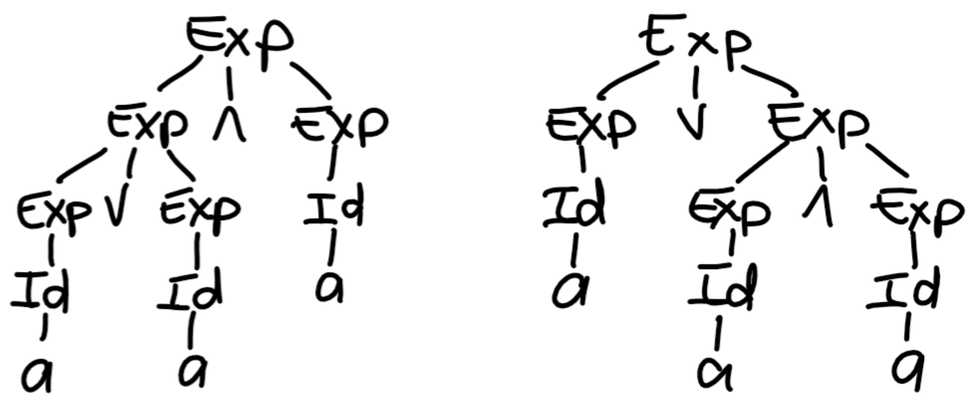
la grammatica NON è ambigua e * e + hanno la stessa precedenza e associano a sinistra

→ 10 luglio 2019

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp ::= Exp ∧ Exp | Exp ∨ Exp | { ExpSeq } | Id
ExpSeq ::= Exp | Exp ExpSeq
Id ::= a | b
```

esempio: a ∨ a ∧ a



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

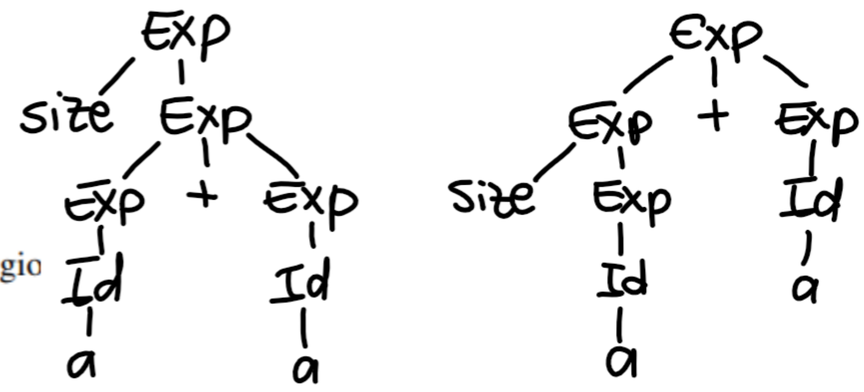
```
Exp ::= Exp ∨ And | And
And ::= And ∧ Atom | Atom
Atom ::= { ExpSeq } | Id
ExpSeq ::= Exp | Exp ExpSeq
Id ::= a | b
```

→ 19 giugno 2019

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp ::= size Exp | Exp + Exp | [ Exps ] | Id
Exps ::= Exp | Exp , Exps
Id ::= a | b
```

esempio: size a+a



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

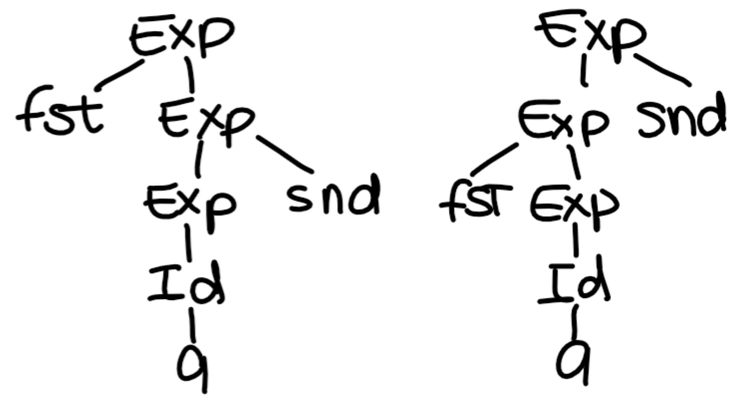
```
Exp ::= Exp + Size | Size
Size ::= size Size | [ Exps ] | Id
Exps ::= Exp | Exp , Exps
Id ::= a | b
```

→ 5 giugno 2019

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp ::= fst Exp | Exp snd | ( Exp , Exp ) | Id
Id ::= a | b
```

esempio : fst a snd



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

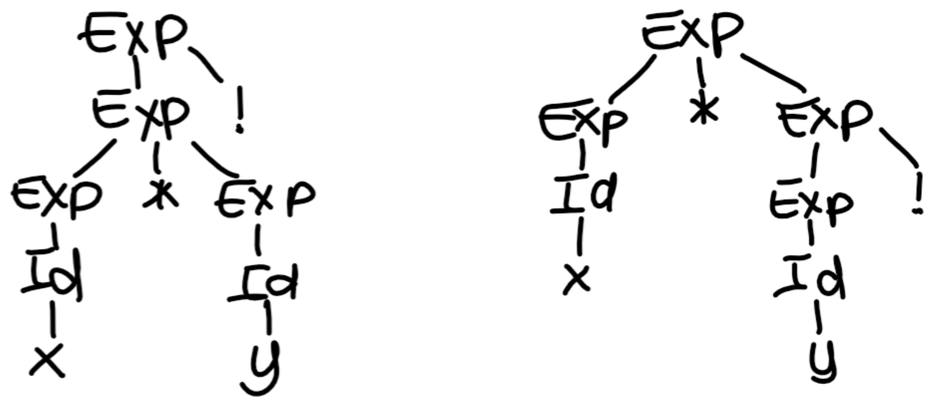
```
Exp ::= fst Exp | Snd
Snd ::= Snd snd | ( Exp , Exp ) | Id
Id ::= a | b
```

→ 11 febbraio 2019

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= Exp ! | Exp * Exp | < Exp > | Id
Id ::= x | y
```

esempio: x*y!



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

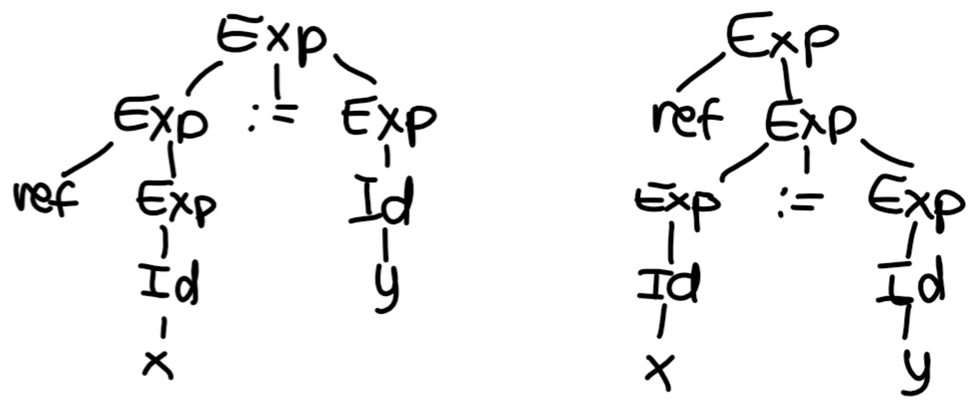
```
Exp  ::= Exp * Bang | Bang
Bang  ::= Bang ! |  < Exp > | Id
Id ::= x | y
```

→ 24 gennaio 2019

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= ref Exp | Exp := Exp | ( Exp ) | Id
Id ::= x | y
```

esempio: ref x:=y



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

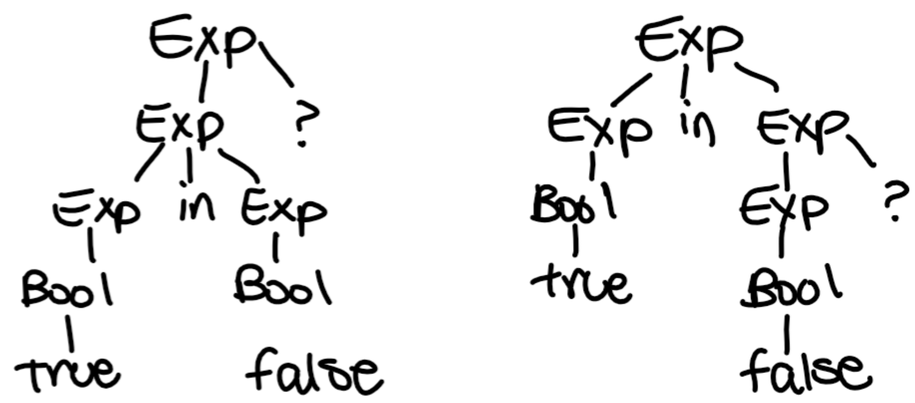
```
Exp  ::= Exp := Ref | Ref
Ref ::= ref Ref | ( Exp ) | Id
Id ::= x | y
```

→ 10 settembre 2018

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= Exp ? | Exp in Exp | ( Exp ) | Bool
Bool ::= false | true
```

esempio: true in false?



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

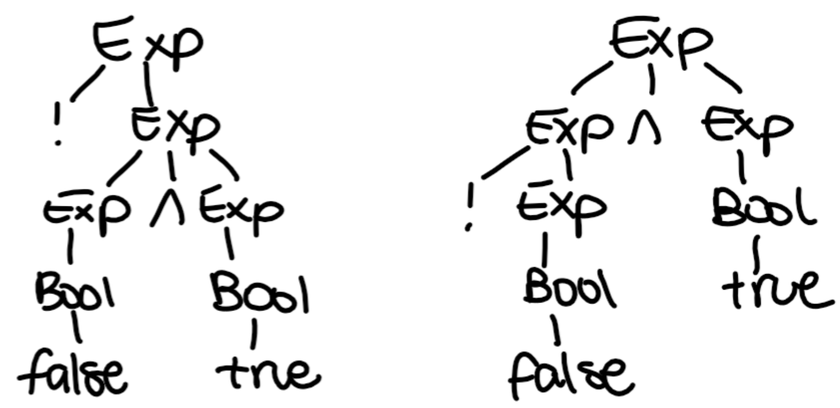
```
Exp ::= Exp in Quest | Quest
Quest ::= Quest ? | ( Exp ) | Bool
Bool ::= false | true
```

→ 7 luglio 2018

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= ! Exp | Exp ∧ Exp | ( Exp ) | Bool
Bool ::= false | true
```

esempio: !false ∧true



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

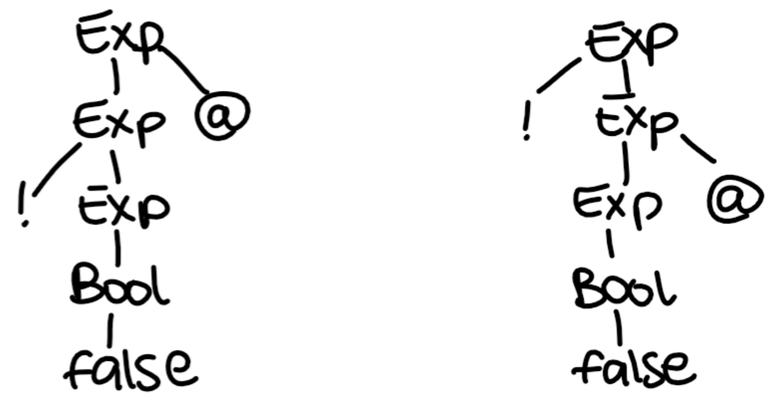
```
Exp ::= Exp ∧ Bang | Bang
Bang ::= ! Bang | ( Exp ) | Bool
Bool ::= false | true
```

→ 20 giugno 2018

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= ! Exp | Exp @ | ( Exp ) | Bool
Bool ::= false | true
```

esempio: ! false @



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

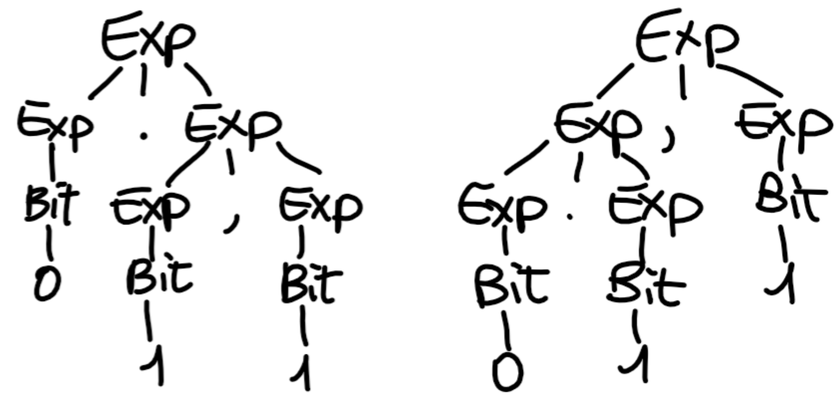
```
Exp  ::= Exp @ | Bang
Bang ::= ! Bang | ( Exp ) | Bool
Bool ::= false | true
```

→ 4 giugno 2018

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= Exp . Exp | Exp , Exp | ( Exp ) | Bit
Bit  ::= 0 | 1
```

esempio: 0.1,1



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

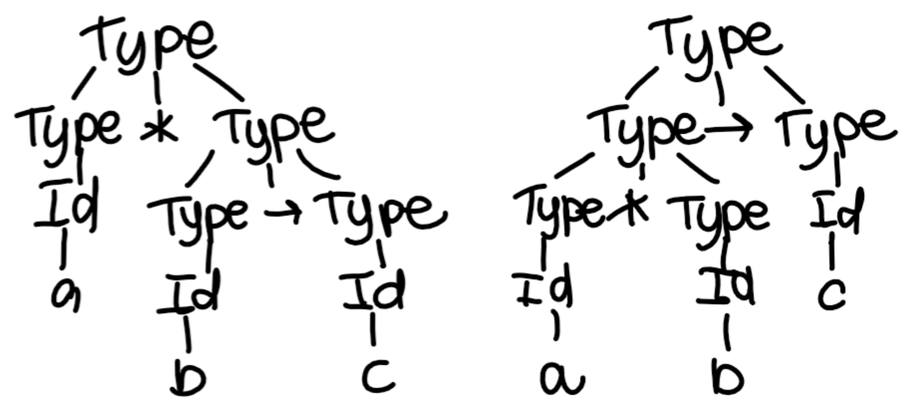
```
Exp  ::= Exp Op Atom | Atom
Op   ::= . | ,
Atom ::= ( Exp ) | Bit
Bit  ::= 0 | 1
```

→ 12 febbraio 2018

(b) Mostrare che la seguente grammatica è ambigua.

```
Type ::= Type * Type | Type -> Type | ( Type ) | Id
Id   ::= a | b | c
```

esempio: a*b→c



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Type` **resti invariato**.

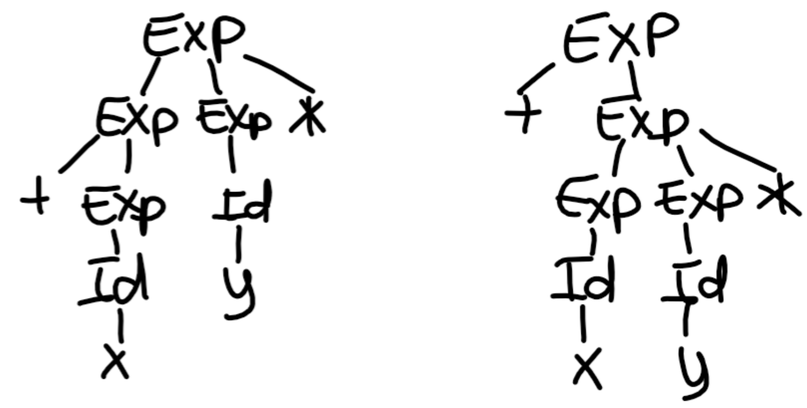
```
Type ::= Prod -> Type | Prod
Prod ::= Prod * Atom | Atom
Atom ::= ( Type ) | Id
Id ::= a | b | c
```

→ 25 gennaio 2018

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= Exp Exp * | + Exp | ( Exp ) | Id
Id   ::= x | y | z
```

esempio: + x y *



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

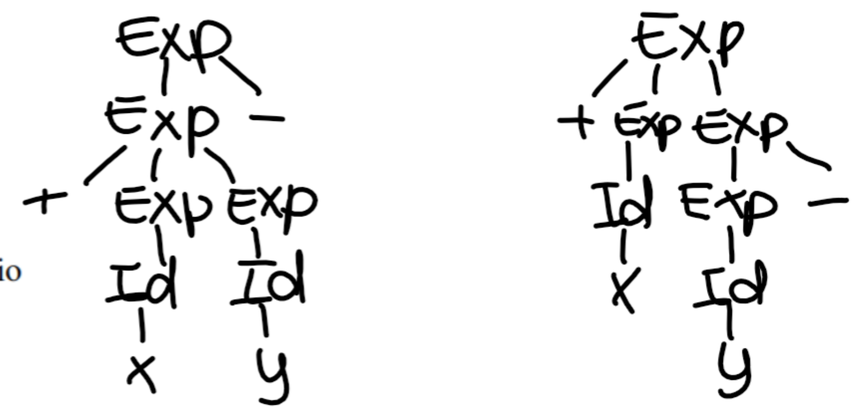
```
Exp  ::= Exp Exp * | Plus
Plus ::= + Plus | ( Exp ) | Id
Id   ::= x | y | z
```

→ 11 settembre 2017

(b) Mostrare che la seguente grammatica è ambigua.

```
1  Exp  ::= + Exp Exp | Exp - | ( Exp ) | Id
2  Id   ::= x | y | z
```

esempio: + x y -



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

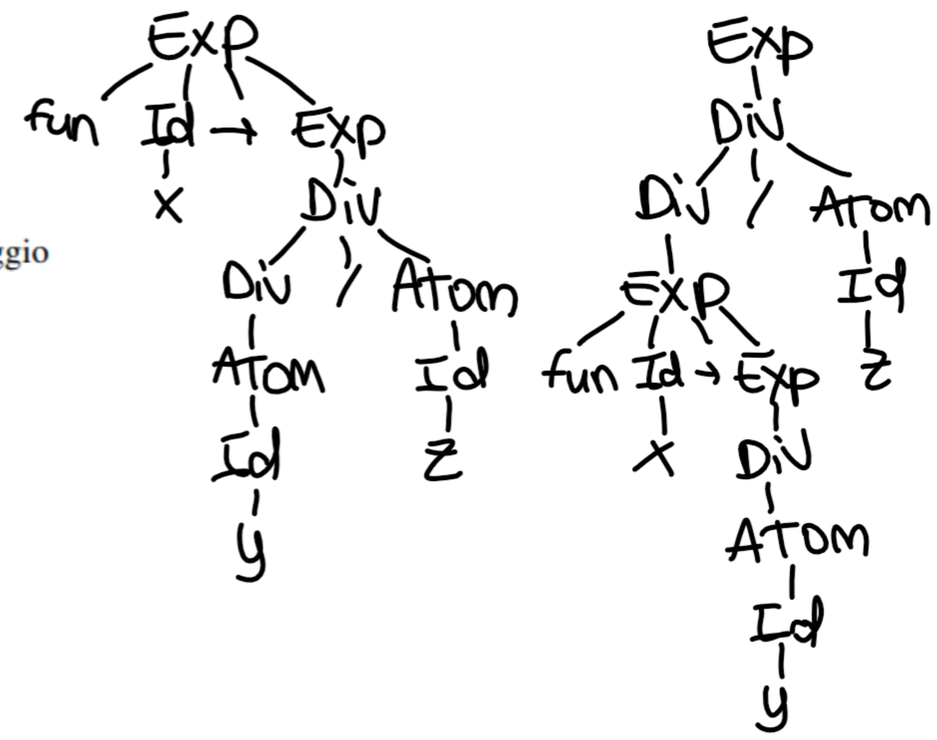
```
Exp   ::= + Exp Exp | Minus
Minus ::= Minus - | ( Exp ) | Id
Id    ::= x | y | z
```

→ 12 luglio 2017

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= fun Id -> Exp | Div
Div  ::= Div / Atom | Atom | Exp
Atom ::= ( Exp ) | Id
Id   ::= x | y | z
```

esempio: fun x→y /z



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

```
Exp  ::= fun Id -> Exp | Div
Div  ::= Div / Atom | Atom
Atom ::= ( Exp ) | Id
Id   ::= x | y | z
```

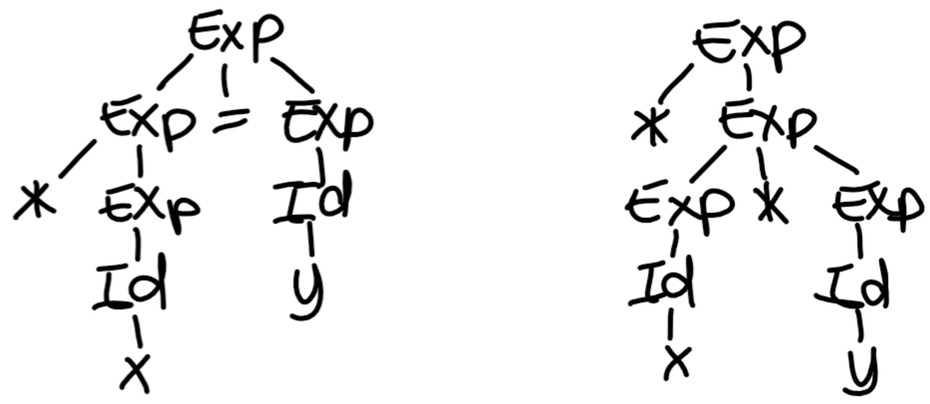
→ elimino Div ::= Exp

→ 5 giugno 2017

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= * Exp | Exp = Exp | ( Exp ) | Id
Id   ::= x | y | z
```

esempio: * x = y



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

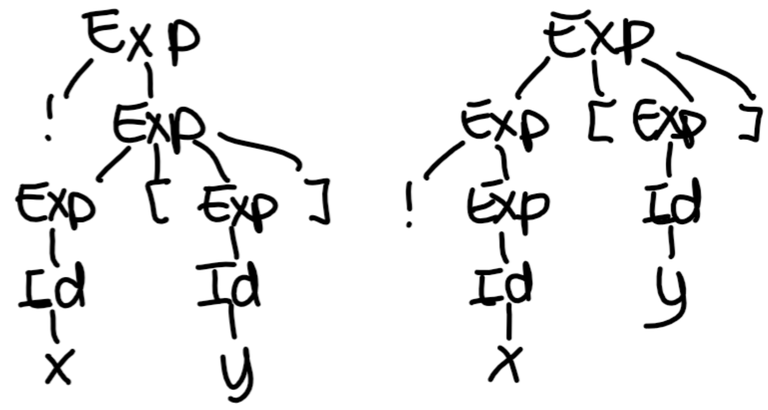
```
Exp   ::= Exp = Deref | Deref
Deref ::= * Deref | ( Exp ) | Id
Id    ::= x | y | z
```

→ 15 febbraio 2017

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= ! Exp | Exp [ Exp ] | ( Exp ) | Id
Id   ::= x | y | z
```

esempio: !x[y]



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

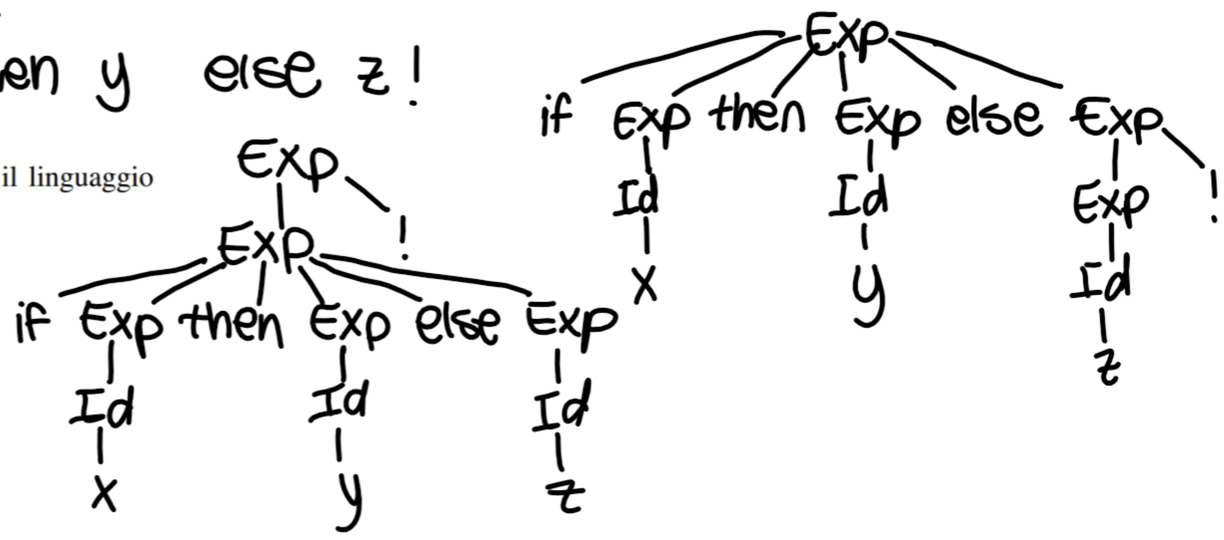
```
Exp   ::= Exp [ Exp ] | Bang
Bang  ::= ! Bang | ( Exp ) | Id
Id    ::= x | y | z
```

→ 8 settembre 2016

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= if Exp then Exp else Exp | Exp ! | ( Exp ) | Id
Id   ::= x | y | z
```

esempio.
if x then y else z!



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

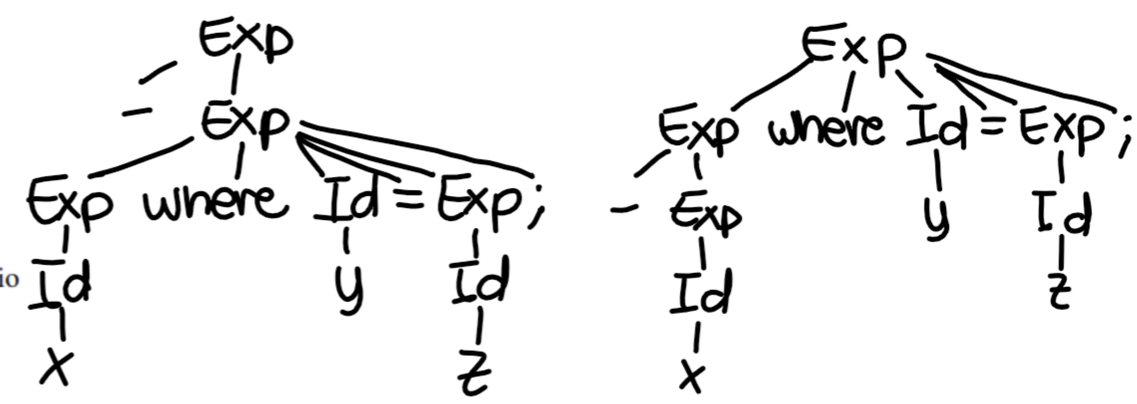
```
Exp  ::= if Exp then Exp else Exp | Bang
Bang ::= Bang ! | ( Exp ) | Id
Id   ::= x | y | z
```

→ 11 luglio 2016

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= Exp where Id = Exp ; | - Exp | ( Exp ) | Id
Id   ::= x | y | z
```

esempio: - x where y = z;



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

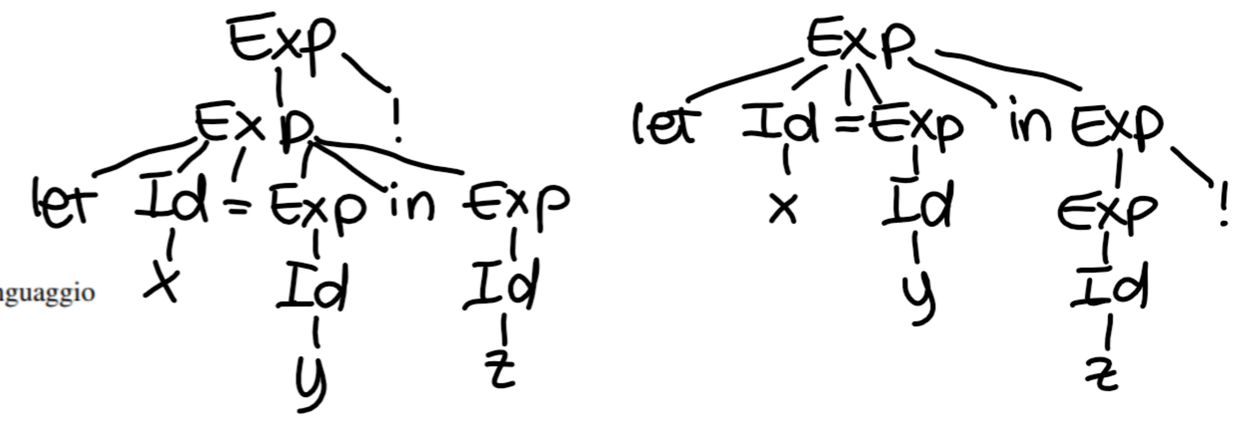
```
Exp   ::= Exp where Id = Exp ; | Minus
Minus ::= - Minus | ( Exp ) | Id
Id    ::= x | y | z
```

→ 9 giugno 2016

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= let Id = Exp in Exp | Exp ! | ( Exp ) | Id
Id   ::= x | y | z
```

esempio: let x = y in z!



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

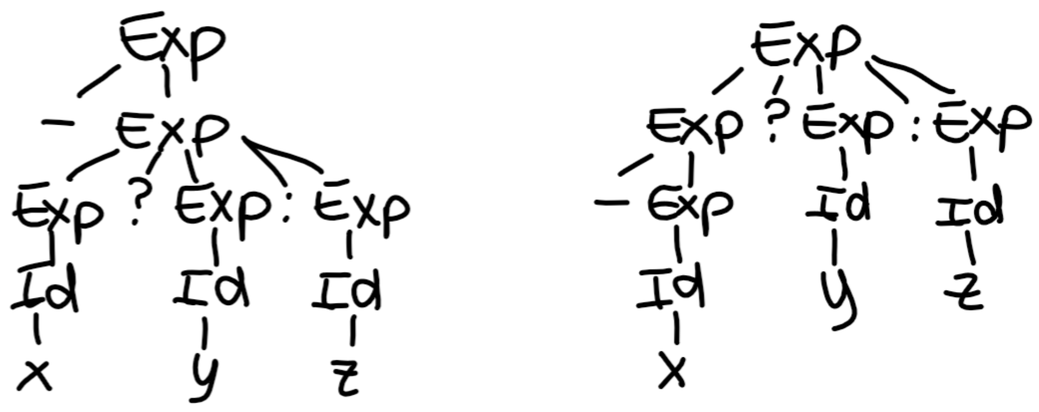
```
Exp  ::= let Id = Exp in Exp | Bang
Bang ::= Bang ! | ( Exp ) | Id
Id   ::= x | y | z
```

→ 11 febbraio 2016

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::=  Exp ? Exp : Exp | - Exp | ( Exp ) | Id
Id   ::= x | y | z
```

esempio: -x? y : z



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

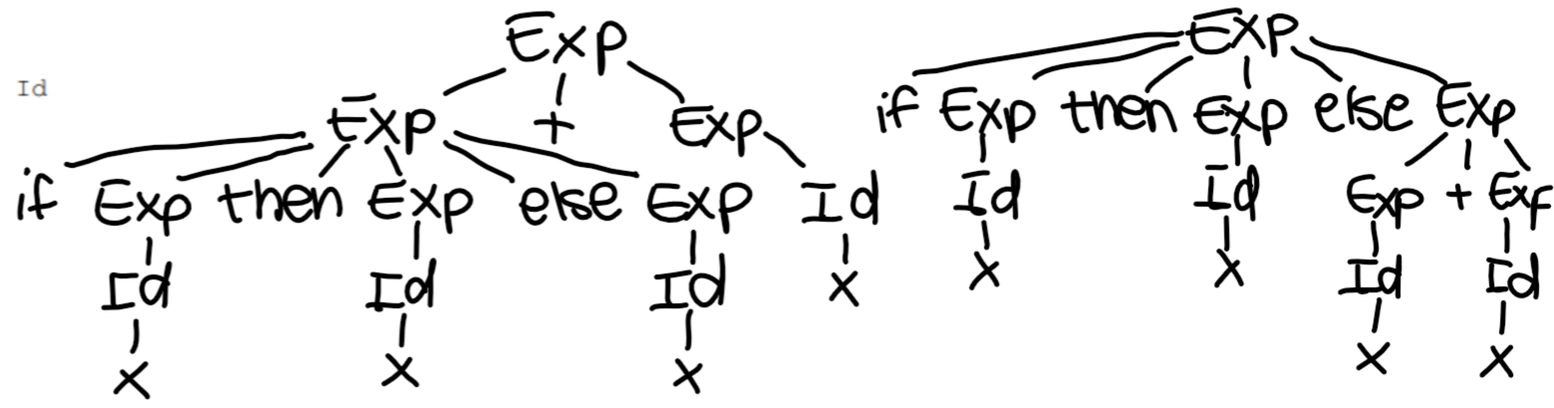
```
Exp   ::=  Exp ? Exp : Minus | Minus
Minus ::= - Minus | ( Exp ) | Id
Id    ::= x | y | z
```

→ 22 gennaio 2016

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= if Exp then Exp else Exp | Exp + Exp | ( Exp ) | Id
Id   ::= x | y | z
```

esempio: if x then x else x + x



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

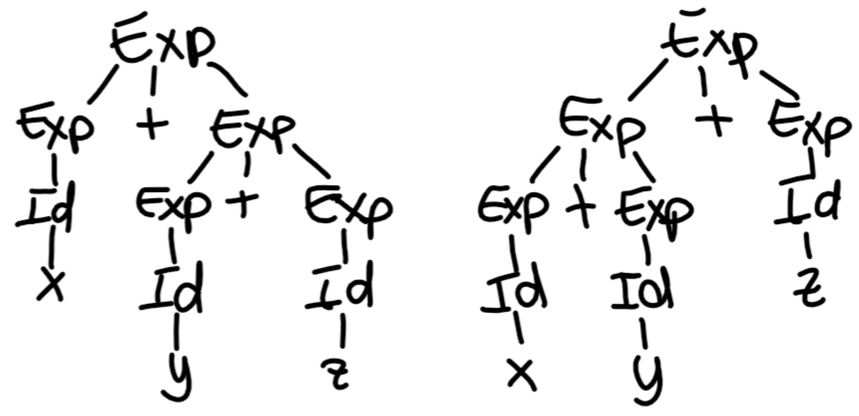
```
Exp  ::= Exp + If | If
If   ::= if Exp then Exp else If | ( Exp ) | Id
Id   ::= x | y | z
```

→ 17 settembre 2015

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= Exp + Exp | + Exp | ( Exp ) | Id
Id   ::= x | y | z
```

esempio: x+y+z



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

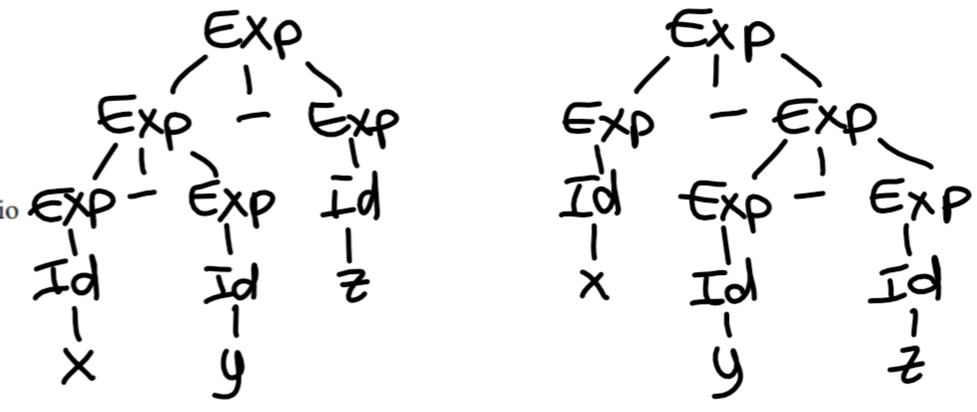
```
Exp  ::= Exp + Atom | Atom
Atom ::= Id | + Atom | ( Exp )
Id   ::= x | y | z
```

→ 21 luglio 2015

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= Exp - Exp | Atom
Atom ::= Id | - Atom | ( Exp )
Id   ::= x | y | z
```

esempio: x-y-z



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

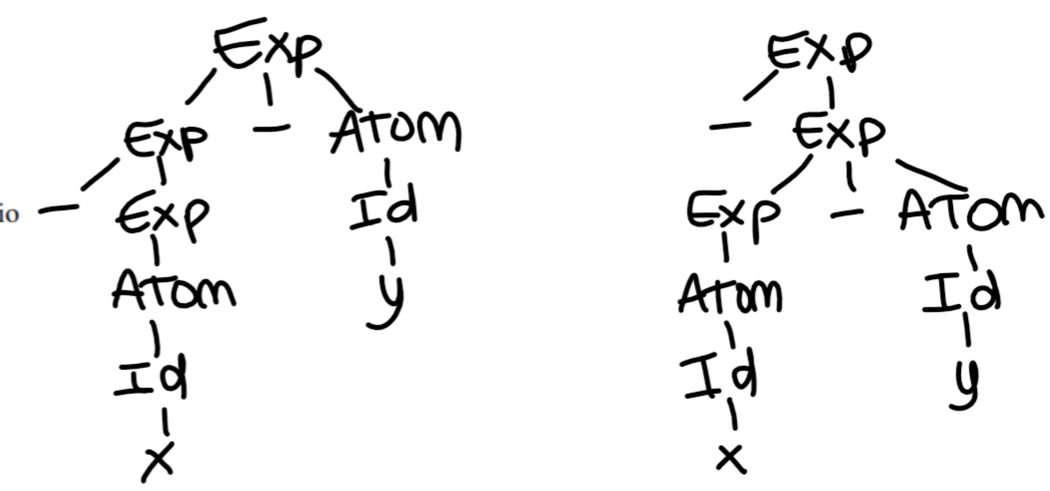
```
Exp  ::= Exp - Atom | Atom
Atom ::= Id | - Atom | ( Exp )
Id   ::= x | y | z
```

→ 5 giugno 2015

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= - Exp | Exp - Atom | Atom
Atom ::= Id | ( Exp )
Id   ::= x | y | z
```

esempio: - x - y



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

Soluzione: La soluzione più semplice consiste nell' aggiungere il non terminale `Minus` per dare precedenza all'operatore unario −.

```
Exp  ::= Exp - Atom | Minus
Minus ::= - Minus | Atom
Atom ::= Id | ( Exp )
Id   ::= x | y | z
```

La grammatica può essere semplificata nella seguente equivalente:

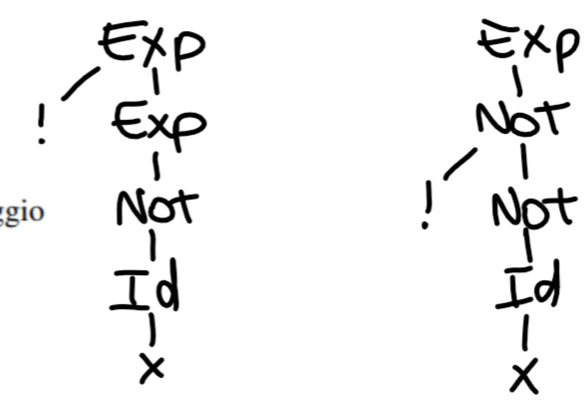
```
Exp  ::= Exp - Atom | Atom
Atom ::= Id | - Atom | ( Exp )
Id   ::= x | y | z
```

→ 5 febbraio 2015

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= Not | ! Exp | Exp && Not
Not  ::= Id | ! Not
Id   ::= x | y | z
```

esempio: !x



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

```
Exp  ::= Not | Exp && Not
Not  ::= Id | ! Not
Id   ::= x | y | z
```

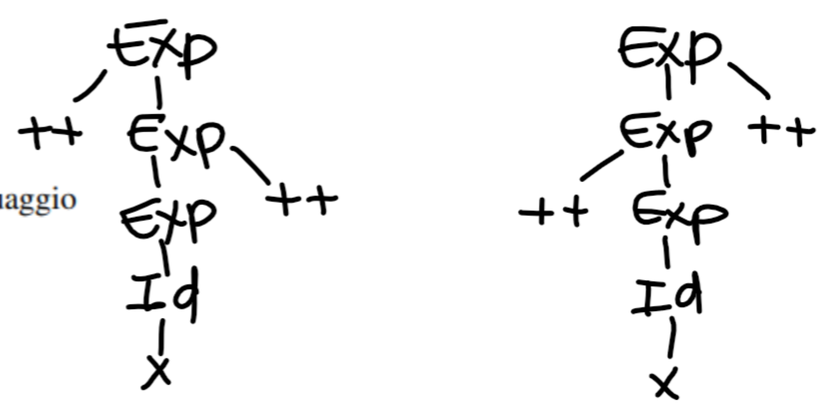
→ elimino Exp::= !Exp

→ 16 gennaio 2015

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= Id | ++ Exp | Exp ++
Id ::= x | y | z
```

esempio: ++x++



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

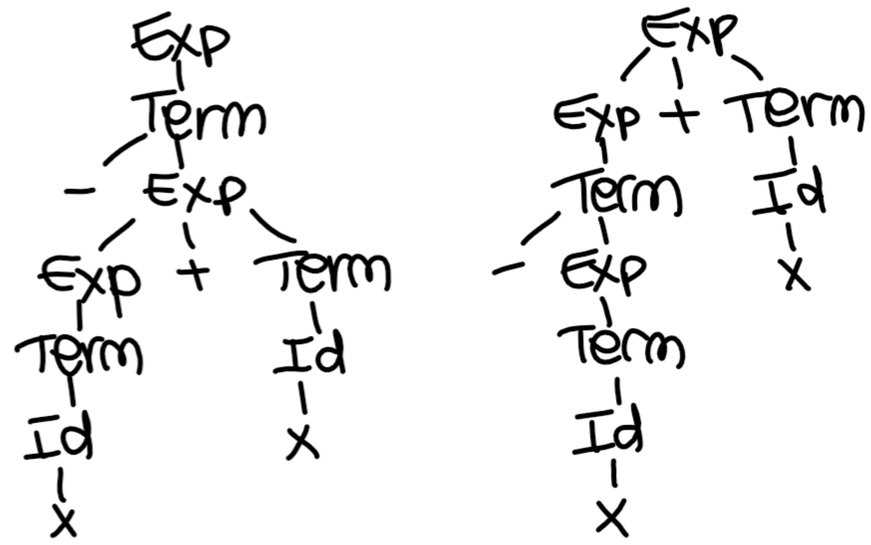
```
Exp  ::= Post | ++ Exp
Post ::= Id | Post ++
Id   ::= x | y | z
```

→ 17 gennaio 2014

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::=  Term | Exp + Term
Term ::= Id | - Exp | ( Exp )
Id ::= x | y | z
```

esempio: -x+x



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

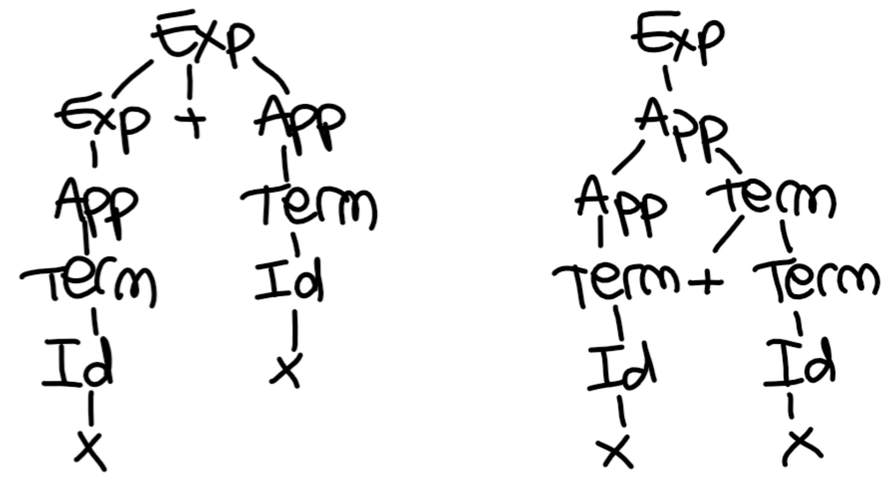
```
Exp  ::=  Term | Exp + Term
Term ::= Id | - Term | ( Exp )
Id ::= x | y | z
```

→ 3 settembre 2013

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= App | Exp + App
App  ::= Term | App Term
Term ::= Id | + Term
Id ::= x | y | z
```

esempio: x+x



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **rimanga lo stesso**.

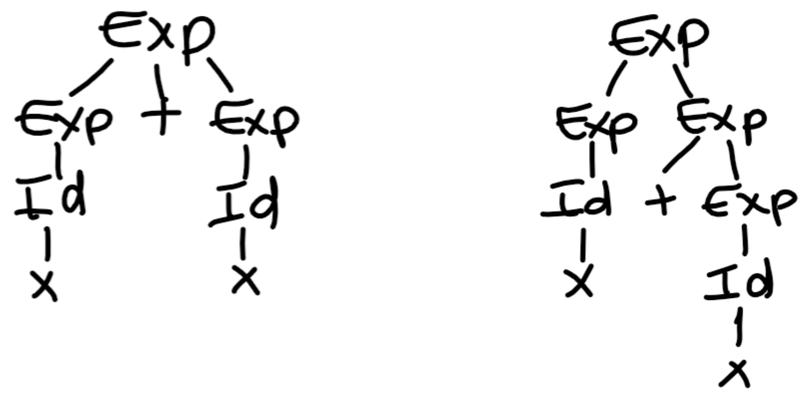
```
Exp  ::= App | Exp + App
App  ::= Term | App Id
Term ::= Id | + Term
Id ::= x | y | z
```

→ 12 luglio 2013

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= Id | Exp Exp | Exp + Exp | + Exp
Id ::= x | y | z
```

esempio x+x



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **rimanga lo stesso**.

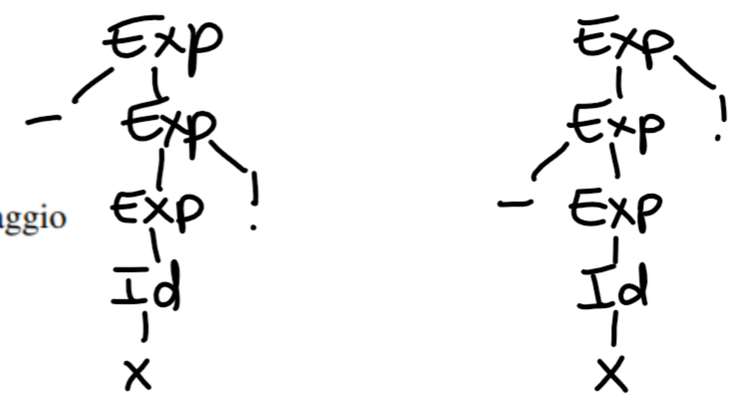
```
Exp  ::= App | Exp + App
App  ::= Term | App Id
Term ::= Id | + Term
Id ::= x | y | z
```

→ 19 giugno 2013

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= Id | Exp ! | - Exp
Id ::= x | y | z
```

esempio: -x!



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **rimanga lo stesso**.

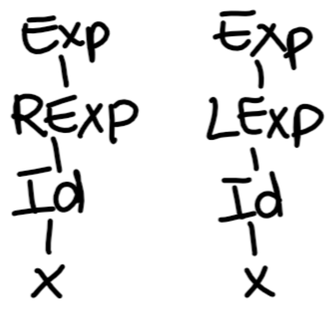
```
Exp  ::= Exp ! | Term
Term ::= - Term | Id
Id ::= x | y | z
```

→ 11 febbraio 2013

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= LExp | RExp
RExp ::= Id | Bit | Id + RExp | Bit + RExp
LExp ::= Id
Id ::= x | y | z
Bit ::= 0 | 1
```

esempio: x



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **rimanga lo stesso**.

```
Exp  ::= RExp
RExp ::= Id | Bit | Id + RExp | Bit + RExp
Id ::= x | y | z
Bit ::= 0 | 1
```

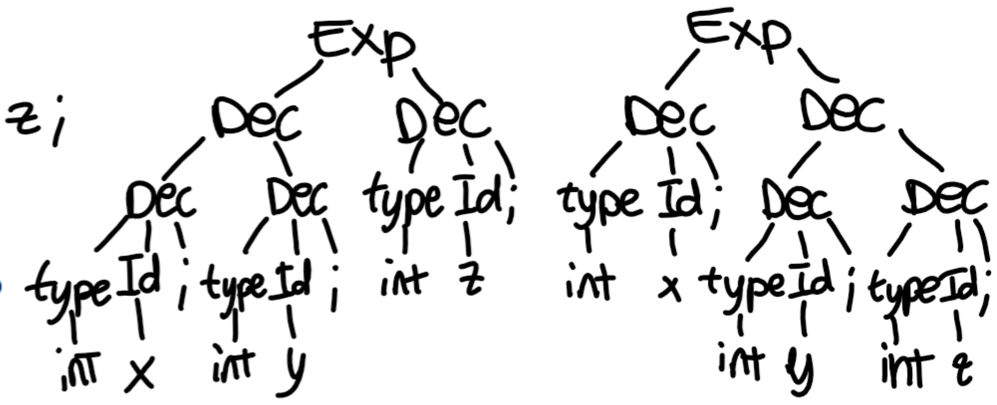
→ elimino completamente LExp

↳ 25 gennaio 2013

(b) Mostrare che la seguente grammatica è ambigua.

```
Dec  ::= Type Id ; | Dec Dec
Id ::= x | y | z
Type ::= int | bool
```

esempio: int x; int y; int z;



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Dec` **rimanga lo stesso**.

```
Dec  ::= Type Id ; | Type Id ; Dec
Id ::= x | y | z
Type ::= int | bool
```