Table of Contents

In [ ]:
```
1
```

Fondamenti di Programmazione

Andrea Sterbini

lezione 19 - 12 dicembre 2022

RECAP:

- Diametro di un albero
- Alberi di gioco
  - Somma di coppie pari o dispari in una sequenza di interi
  - Anagramma e numero di scambi
  - Dare i resti se si hanno certe monete

Errori: try/except/finally e raise

- si definiscono come sottoclassi di **Exception**
- si lanciano con **raise**
- si catturano con **try/except/finally**

In [1]:

```python

# --- definire nuovi errori con nuove classi che ereditano da Exception
class MioErrore(Exception) : pass

# esempio di "cattura" di due tipi di errore
try:
    # codice che normalmente va eseguito e potrebbe generare un errore

    # Per "lanciare" un errore si usa 'raise' (solleva)
    # lancio un mio tipo di errore con un messaggio
    raise MioErrore('è successo qualcosa di strano')

    with open('proibito2') as F:    # provo ad aprire un file che non esiste
        text = F.read()

# qui catturo il tentativo di aprire un file per cui non ho i permessi
except PermissionError as e:
    print(e)                        # e stampo il messaggio della eccezione catturata

# qui catturo il tentativo di aprire un file che non esiste
except FileNotFoundError as e:
    print("file non trovato")       # e stampo un messaggio personalizzato

except Exception as e:              # catturo qualsiasi altro tipo di eccezione
    print(e)          # DA NON USARE (cattura troppo e fallisce il test di ricorsione)
    #raise                          # e posso anche ri-lanciare l'errore

# in ogni caso mi assicuro alla fine di eseguire questo codice "no matter what"
finally:
    # codice da eseguire sempre
    print("fatto")
# codice che segue, che potrebbe non essere eseguito se c'è una eccezione non catturata
print("continuo da qui")
################################################################################

```

```
è successo qualcosa di strano
fatto
continuo da qui
```

GIOCO: Filetto/Tris

- 2 giocatori
- **configurazione**: scacchiera 3x3 con cimboli **o** oppure **x** (oppure spazio per la casella vuota)
- **mosse possibili**: inserire il simbolo del giocatore di turno in una casella vuota
- **vincita**: 3 simboli uguali in fila (riga, colonna o diagonale)
- **convergenza**: max 9 caselle quindi max 9 mosse

In [2]:

```python
import graphviz
class GraphvizNode:
    _num_nodi = 0
    def __init__(self, horizontal=True):
        GraphvizNode._num_nodi += 1           # tengo conto di quanti nodi ho generato
        self.__id = GraphvizNode._num_nodi    # ciascuno ha una ID unica
        self._horizontal = horizontal     # per default lo mostro in orizzontale
        self._sons = []
    def dot(self):
        """Costruisco la rappresentazione dell'albero da visualizzare con Graphviz
           ovvero l'elenco di nodi e di archi da visualizzare"""
        if self._sons:
            s = f'{self.__id} [label="{self.label()}"]\n' # se non foglia colore nero
        else:
            s = f'{self.__id} [label="{self.label()}" color=red, style=bold]\n' # coloro le foglie di rosso
        for son in self._sons:
            s += f'{self.__id} -> {son.__id}\n'   # archi padre -> figlio
            s += son.dot()                        # più tutti i nodi e archi dei sottoalberi
        return s

    def label(self):
        "per default un nodo mostra la propria stringa"
        return self.__repr__()

    def show(self):
        "Creo l'oggetto Digraph che visualizza il grafo diretto"
        G = graphviz.Digraph()
        rankdir = 'LR' if self._horizontal else 'TD'
        G.body.append(f'''rankdir={rankdir}
        node [shape=record]
        ''' + self.dot())
        return G
```

In [3]:

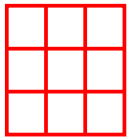
```python
import jdc

# nuovo tipo di errore per comunicare errori del gioco
class FilettoError(Exception):
    pass

class Filetto(GraphvizNode):
    def __init__(self, configurazione=None):
        "creazione di un nuovo nodo a partire da una data configurazione, rappresentata come matrice 3x3"
        super().__init__()
        if configurazione is None:                          # se non viene passata
            configurazione = [ [' ']*3 for i in range(3)]   # ne creiamo una vuota
        self._configurazione = configurazione
        self._sons = []                                     # all'inizio non ci sono figli

    def __repr__(self):
        "rappresentazione sotto forma di stringa della tabella, in graphviz mostra una scacchiera"
        return '{' + '|'.join(['{'+ '|'.join(riga)+'}' for riga in self._configurazione ]) + '}'
```

In [4]:

```python
F = Filetto()
print(F)
F.show()
```

```
{{ | | }|{ | | }|{ | | }}
```

Out[4]:



Le mosse valide sono tutte le caselle libere
(MA SOLO se non è finita la partita)

In [5]:
```
%%add_to Filetto
# --- elenco delle mosse valide
def mosse_valide(self):
    """Trovo le mosse valide (ma non ce ne sono se patta o qualcuno ha vinto)"""
    if self.vincente('x'): return []
    if self.vincente('o'): return []
    if self.patta():       return []
    return [ (x,y)
                    for y,riga    in enumerate(self._configurazione)
                    for x,casella in enumerate(riga)
                    if casella == ' ']

# definirò fra poco i metodi 'patta' e 'vincente'
```

La partita è finita alla pari se non ci sono più mosse disponibili

(ovvero nessuno ha già vinto)

In [6]:
```
%%add_to Filetto
# --- Configurazione che dà patta
def patta(self):
    "torno True se siamo in una patta (non ci sono caselle libere)"
    return self.prossimo_giocatore() is None
```

il prossimo giocatore si ottiene contando le mosse

- si inizia sempre con 'o'
- però torniamo None se non ci sono più caselle libere

In [7]:
```
%%add_to Filetto
# --- Prossimo giocatore
def prossimo_giocatore(self):
    "si inizia sempre col simbolo 'o' quindi basta contare gli ' ' per sapere a chi tocca"
    conteggio = sum( 1
                    for riga in self._configurazione
                    for cell in riga
                    if  cell == ' ')
    if conteggio == 0:         # se non ci sono spazi
        return None            # non è il turno di nessuno
    elif conteggio % 2 == 1:  # inizia sempre 'o' (con 9 caselle libere)
        return 'o'
    else:
        return 'x'
```

In [8]:
```
Filetto().prossimo_giocatore()
```

Out[8]:
```
'o'
```

una configurazione è vincente per un certo giocatore
se ci sono 3 simboli in fila uguali al suo

In [9]:
```
%%add_to Filetto
# --- Configurazione vincente per un giocatore G o K (non intendo strategia)
def vincente(self, giocatore):
    """la configurazione corrente è vincente per il giocatore se ci sono 3
    dei suoi simboli in riga, colonna o diagonale"""
    [[A,B,C],
     [D,E,F],
     [G,H,I]] = self._configurazione
    return (A == B == C == giocatore  # prima riga
         or D == E == F == giocatore  # seconda riga
         or G == H == I == giocatore  # terza riga
         or A == D == G == giocatore  # prima colonna
         or B == E == H == giocatore  # seconda colonna
         or C == F == I == giocatore  # terza colonna
         or A == E == I == giocatore  # diagonale
         or C == E == G == giocatore  # antidiagonale
        )
```

In [10]:
```
Filetto( [[1,1,1],[2,1,2],[3,3,1]]).vincente(1)
```

Out[10]: True

In [11]:
```
Filetto().mosse_valide()
```

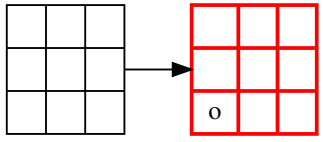
Out[11]: [(0, 0), (1, 0), (2, 0), (0, 1), (1, 1), (2, 1), (0, 2), (1, 2), (2, 2)]

Per applicare una mossa basta mettere il simbolo GIUSTO nella casella

In [12]:
```
%%add_to Filetto
# --- Mosse: inserire il simbolo di turno in una delle caselle vuote
def applica_mossa(self, x, y):
    """data una coordinata x,y inserisco il giocatore di turno,
    e costruisco un nuovo figlio con la nuova configurazione"""
    if self._configurazione[y][x] != ' ':       # se la casella NON è libera
        raise FilettoError(f"La casella {x} {y} è già occupata")
    # altrimenti tutto OK
    copia = [ riga.copy() for riga in self._configurazione ]    # copio la configurazione
    copia[y][x] = self.prossimo_giocatore()                     # e ci metto il simbolo alle coordinate x,y
    self._sons.append(Filetto(copia))      # e aggiungo il nuovo nodo ai figli
    return self
```

In [13]:
```
Filetto().applica_mossa(2,0).show()
```

Out[13]:

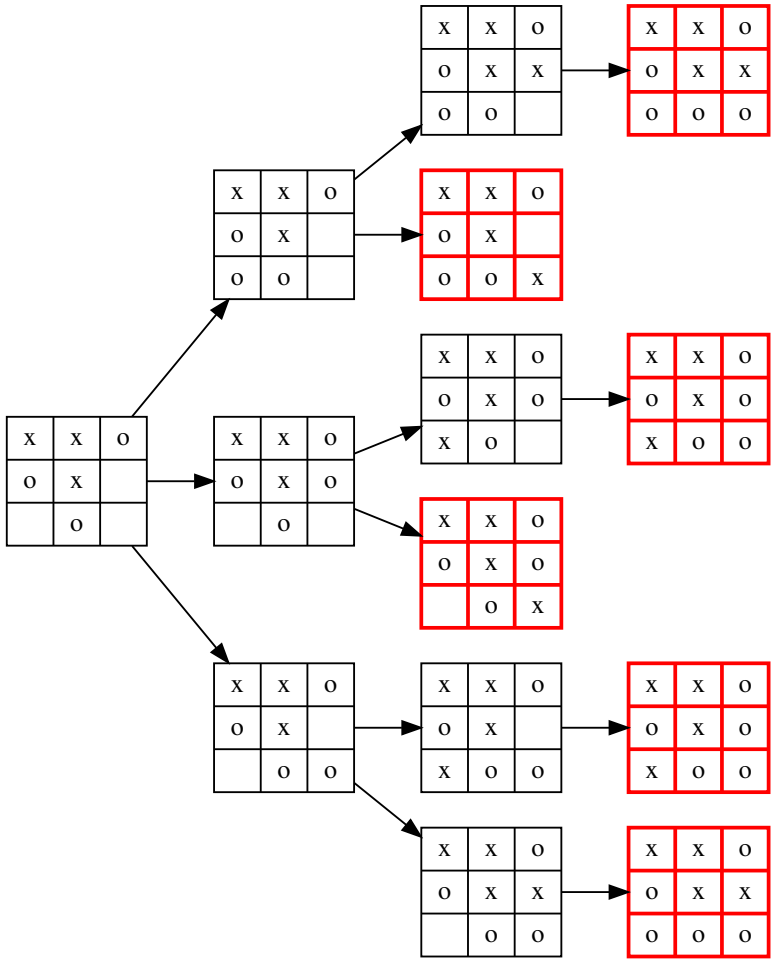


Finalmente generiamo tutto l'albero di gioco

In [14]:
```
%%add_to Filetto
# --- generare l'albero di gioco
def genera(self):
    "Se ci sono mosse valide genero i figli, quindi espando anche i figli"
    mosse = self.mosse_valide()
    for x,y in mosse:
        self.applica_mossa(x, y)
    for figlio in self._sons:
        figlio.genera()
    return self
```

In [15]:
```
board = [['x','o',' '],
         ['x','x','o'],
         ['o',' ',' ']]
Filetto(board).genera().show()
```

Out[15]:



Strategia vincente per il giocatore G

- casi base
  - se patta NO
  - se vincente per G SI

    - se vincente per K NO
- altrimenti: esiste una mossa per G tale che per tutte le mosse di K si arriva sempre ad una posizione vincente x G?
    - se è il turno di G e basta UNA mossa che porti alla vittoria di G
    - se è il turno di K e devono TUTTE portare alla vittoria di G

In [16]:

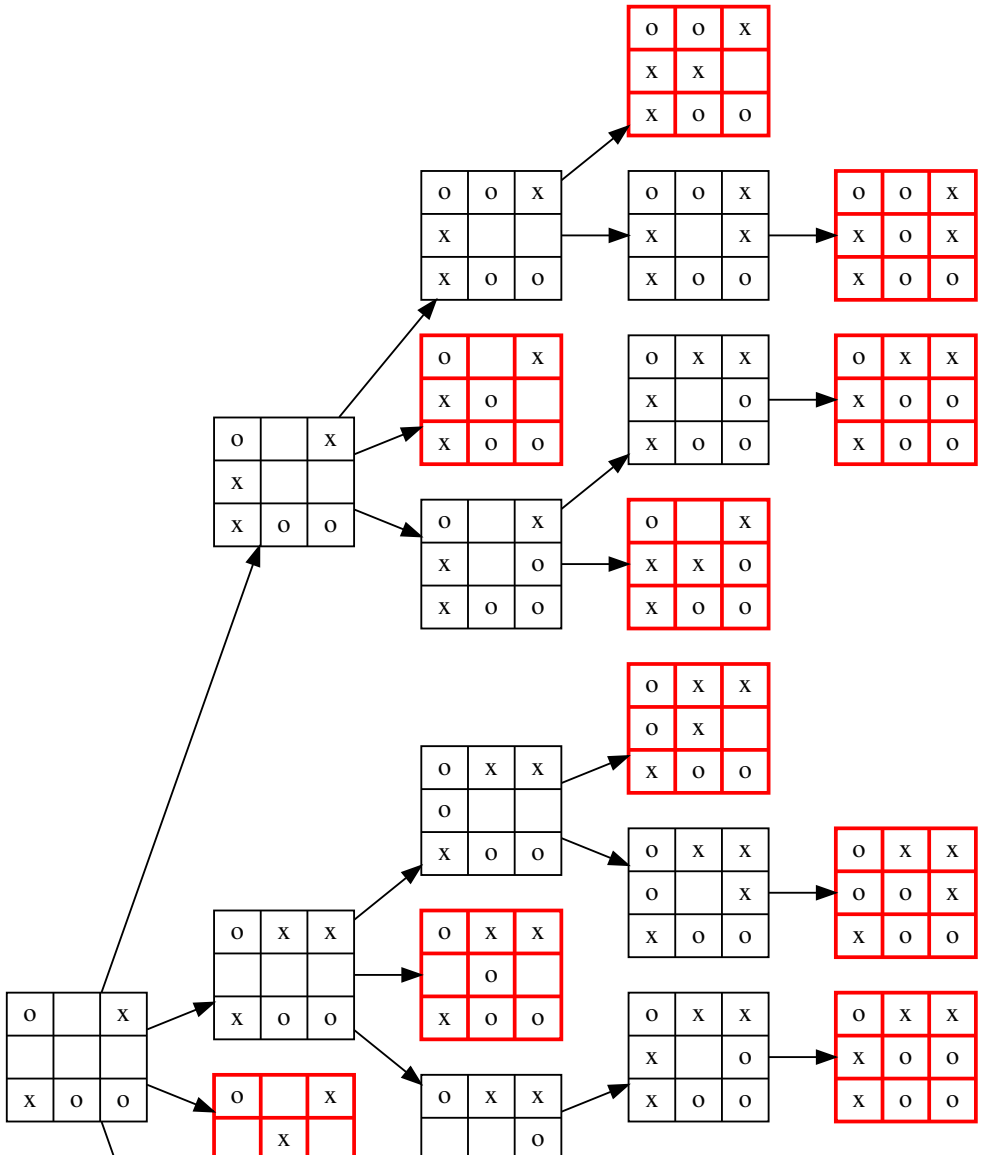
```
%%add_to Filetto
# --- Strategia vincente per il giocatore G
def esiste_strategia_vincente(self, giocatore):
    "vedo se questa posizione ha una strategia vincente per il giocatore"
    if self.vincente(giocatore):    # sì se ho già vinto
        return True
    altro = 'o' if giocatore == 'x' else 'x'
    if self.vincente(altro):        # no se ha vinto l'altro giocatore
        return False
    if self.patta():                # no se siamo alla patta
        return False
    # altro modo: se non ho figli e ho vinto True else False
    pg = self.prossimo_giocatore()
    if giocatore == pg:             # se tocca a me
        for figlio in self._sons:  # e c'è almeno un figlio che è vincente posso muovermi lì e quindi questa posizione è vincente
            if figlio.esiste_strategia_vincente(giocatore):
                return True
        else:
            return False            # altrimenti nessuno dei figli è vincente, questa posizione non è vincente
    else:                           # se invece non è il giocatore a dover giocare
        for figlio in self._sons:  # per vincere, nessuna delle scelte dell'avversario deve essere vincente
            if figlio.esiste_strategia_vincente(altro):    # se una lo è, questa posizione NON è vincente per me
                return False
        else:                       # se nessuno dei figli è vincente per l'avversario
            return True             # allora io posso vincere da qui

# TODO: estrarre la sequenza di mosse vincenti
```

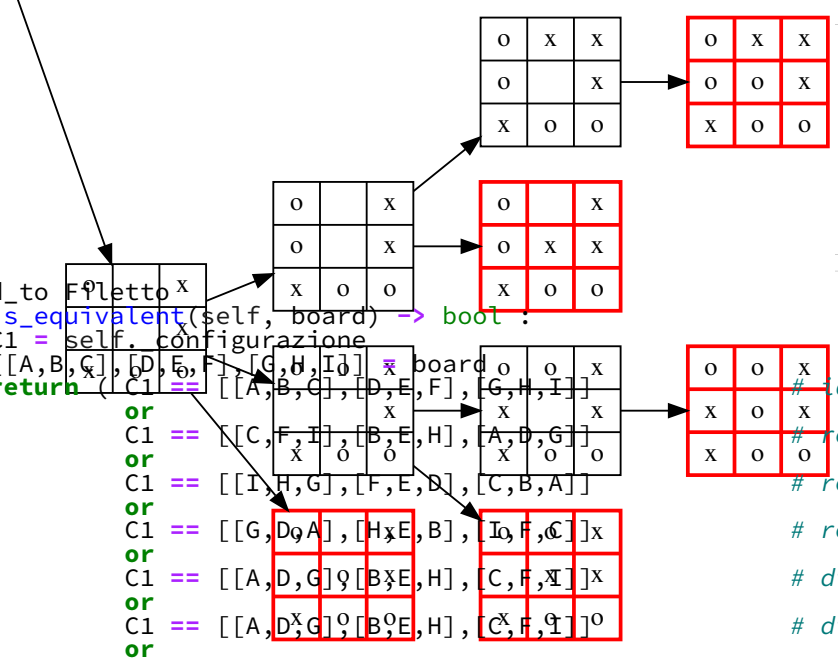
In [17]:

```
f1 = Filetto([
    ['o', ' ', 'x'],
    [' ', ' ', 'o'],
    ['x', ' ', 'o'],
    ])
print('è posizione vincente per x?      ', f1.vincente('x'))
print('prossimo giocatore               ', f1.prossimo_giocatore())
print('è patta?                         ', f1.patta())
#f1.applica_mossa(1, 0, 'x')
f1.genera()
print("c'è una strategia vincente per x?", f1.esiste_strategia_vincente('x'))
f1.show()
```

```
è posizione vincente per x?       False
prossimo giocatore                x
è patta?                          False
c'è una strategia vincente per x? True
```

Out[17]:





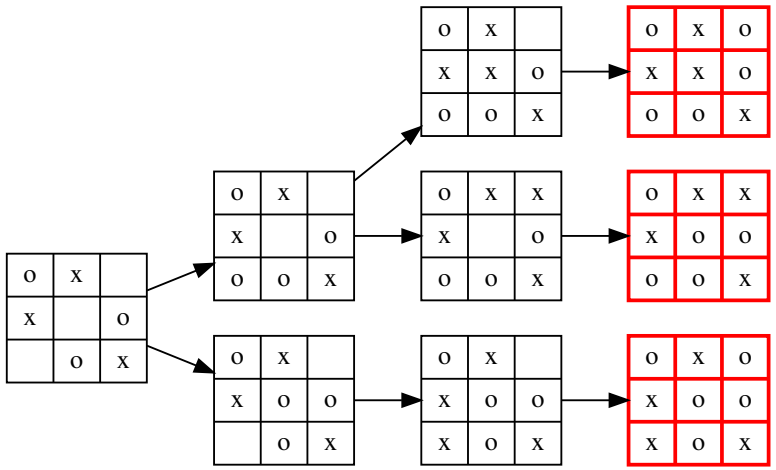
In [18]:
```
%%add_to Filetto
def is_equivalent(self, board) -> bool :
    C1 = self._configurazione
    [[A,B,C],[D,E,F],[G,H,I]] = board
    return ( C1 == [[A,B,C],[D,E,F],[G,H,I]]                # identica
             or
             C1 == [[C,F,I],[B,E,H],[A,D,G]]                # rot 90° antioraria
             or
             C1 == [[I,H,G],[F,E,D],[C,B,A]]                # rot 180°
             or
             C1 == [[G,D,A],[H,E,B],[I,F,C]]                # rot 90° oraria
             or
             C1 == [[A,D,G],[B,E,H],[C,F,I]]                # diag. princ.
             or
             C1 == [[A,D,G],[B,E,H],[C,F,I]]                # diag. second.
             or
             C1 == [[G,H,I],[D,E,F],[A,B,C]]                # sopra sotto
             or
             C1 == [[C,B,A],[F,E,D],[I,H,G]]                # sinistra destra
           )
```

In [19]:
```
%%add_to Filetto
# --- Mosse: inserire il simbolo di turno in una delle caselle vuote
def applica_mossa(self, x, y):
    """data una coordinata x,y provo a vedere se posso inserire il giocatore,
    e costruisco un nuovo figlio con la nuova configurazione"""
    assert self._configurazione[y][x] == ' ', f"La casella {x} {y} è già occupata"
    # altrimenti tutto OK
    copia = [ riga.copy() for riga in self._configurazione ]    # copio la configurazione
    copia[y][x] = self.prossimo_giocatore()       # e ci metto il simbolo alle coordinate x,y
    # se però è una configurazione equivalente ad una già generata non l'aggiungo
    if any( son.is_equivalent(copia) for son in self._sons ):
        print("Skipping a configuration because equivalent to others")
        return self
    self._sons.append(Filetto(copia))      # e aggiungo il nuovo nodo ai figli
    return self
```

In [20]:
```
f2 = Filetto([
    ['o', 'x', ' '],
    ['x', ' ', 'o'],
    [' ', 'o', 'x'],
    ])
f2.genera().show()
```

```
Skipping a configuration because equivalent to others
Skipping a configuration because equivalent to others
```

Out[20]:



Espressioni algebriche e loro manipolazione

Una espressione agebrica è

- un numero intero
- una variabile (1 sola lettera)
- (espressione operatore espressione)
  - operatori: **`* / + - ^`**
  - senza precedenza tra operatori: ogni operazione è racchiusa tra parentesi

In [21]:
```python
# definiamo sue nuovi tipi di errore
class DivisionePerZeroError(Exception):
    pass        # si comporta esattamente come Exception, quindi non ha attributi o metodi suoi

class EspressioneError(Exception): pass

# descrizione della sintassi
# espressione ::= numero
# espressione ::= variabile
# espressione ::= '(' espressione operatore espressione ')'
# operatore   ::= '*' | '+' | '-' | '/' | '^'
# variabile   ::= *un carattere alfabetico*
# numero      ::= una sequenza di *cifre*

class Numero(GraphvizNode):
    '''
    Un numero contiene un valore numerico
    '''
    def __init__(self, valore):
        super().__init__(False)   # visualizzo l'albero in verticale
        self._valore = valore

    # --- stampa della espressione algebrica da un albero
    def __repr__(self):
        "il testo che rappresenta questo oggetto non è altro che il valore come stringa"
        return str(self._valore)
```

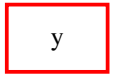
In [22]:
```python
Numero(12).show()
```

Out[22]:

In [23]:
```python
class Variabile(GraphvizNode):
    def __init__(self, nome):
        "una variabile ha un nome (stringa)"
        super().__init__()
        self._nome = nome

    # --- stampa della espressione algebrica da un albero
    def __repr__(self):
        "come stringa la variabile è rappresentata dal suo nome"
        return self._nome
```

In [24]:
```python
Variabile('y').show()
```

Out[24]:

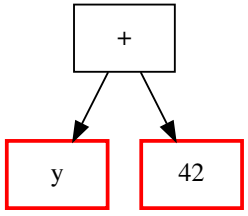


In [25]:
```
class Espressione(GraphvizNode):
    def __init__(self, argomento1, operatore, argomento2):
        "una Espressione ha sempre due argomenti ed un operatore (+-*/^)"
        super().__init__(False)
        self._operatore = operatore
        self._argomento1 = argomento1
        self._argomento2 = argomento2
        self._sons = argomento1, argomento2

    # --- stampa della espressione algebrica da un albero
    def __repr__(self):
        #return self._operatore
        # qua ci sono 2 chiamate ricorsive implicite a __repr__ per inserire gli argomenti nella stringa
        return f'({self._argomento1} {self._operatore} {self._argomento2})'
    def label(self):
        return self._operatore
```

In [26]:
```
E = Espressione(Variabile('y'),'+',Numero(42))
print(E)
E.show()
```

```
(y + 42)
```

Out[26]:



In [27]:
```
%%add_to Numero
def calcola(self, _env=None):
    return self._valore
```

In [28]:
```
Numero(666).calcola()
```

Out[28]: 666

In [29]:
```
%%add_to Variabile
# calcolo del valore della variabile
def calcola(self, environment):
    "dato un dizionario { nome -> valore }, una variabile ha il valore che nel dizionario corrisponde al suo nome"
    try:
        return environment[self._nome]
    except KeyError:
        raise EspressioneError(f"La variabile {self._nome} non è presente nell'environment")
```

In [30]:
```
Variabile('x').calcola({'y':90, 'x':55})
```

Out[30]: 55

In [31]:

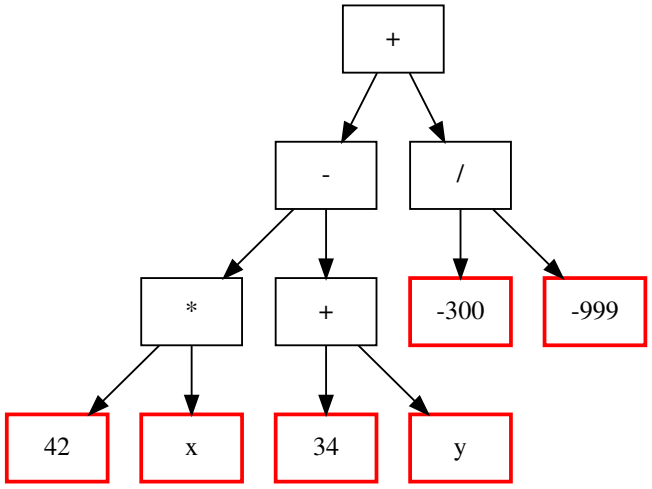
```
%%add_to Espressione
# calcolo della espressione algebrica da un albero (con variabili)
# environment è un dizionario { variabile -> valore } che permette di calcolare
# l'espressione per certi valori delle variabili
def calcola(self, environment):
    "per calcolare il valore di una espressione prima calcolo i due argomenti e poi applico l'operatore"
    arg1 = self._argomento1.calcola(environment)    # passo l'environment alle due sottoespressioni
    arg2 = self._argomento2.calcola(environment)    # che potrebbero contenere variabili
    if self._operatore == '+':
        return arg1 + arg2
    if self._operatore == '-':
        return arg1 - arg2
    if self._operatore == '*':
        return arg1 * arg2
    if self._operatore == '/':
        return arg1 / arg2
    if self._operatore == '^':
        return arg1 ** arg2
```

In [32]:

```

n1 = Numero(42)
n2 = Numero(34)
n3 = Numero(-300)
n4 = Numero(-999)
v1 = Variabile('x')
v2 = Variabile('y')
e1 = Espressione( n1, '*', v1 )  # (42 * x)
e2 = Espressione( n2, '+', v2 )  # (34 * y)
e3 = Espressione( n3, '/', n4 )  # (-300 / -999)
e4 = Espressione( e1, '-', e2 )  # ((42 * x) - (34 * y))
e5 = Espressione( e4, '+', e3 )  # (((42 * x) - (34 * y)) + (-300 / -999))

print(e5)

env = { 'x': 10, 'y': 20, 'z': 3  }

print(e5.calcola(env))
e5.show()
```

```
(((42 * x) - (34 + y)) + (-300 / -999))
366.3003003003003
```

Out[32]:



Come analizzare una espressione come testo e costruire l'albero?

- se inizia per cifra c'è un numero -> leggiamo le altre cifre
- se inizia per lettera è una variabile
- se inizia per '(' è una espressione
  - ricorsivamente leggiamo il primo argomento
  - poi l'operatore
  - poi il secondo argomento
  - poi la ')'

Ad ogni passo guardiamo un solo carattere

Una volta riconosciuto un frammento ci serve sapere cosa ancora va analizzato, quindi torniamo

- l'espressione ottenuta
- il resto della stringa da esaminare (per completare chiamate ricorsive precdenti)

In [33]:

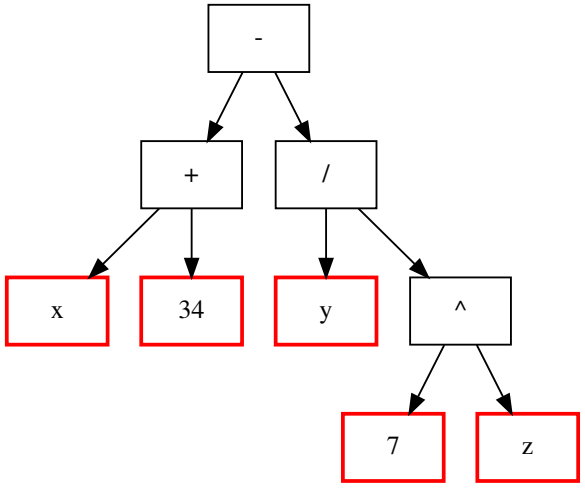
```python
# la funzione analizza la parte iniziale della stringa, riconoscendo l'espressione
# e la torna assieme alla parte di testo che ancoraa non è stata esaminata

def analizza(stringa):
    # FIXME: Prima di chiamare analizza conviene togliere eventuali spazi con replace
    #primo, *resto = stringa         # prendo il primo carattere e lascio in resto i rimanenti
    primo = stringa[0]
    resto = stringa[1:]
    if primo.isdecimal():           # se è una cifra
        # torno un Numero con tutte le cifre che trovo
        valore = primo              # concateno le cifre
        while resto and resto[0].isdecimal():   # se ce ne sono ancora
            valore += resto.pop(0)              # tolgo la prima cifra
        # quando non ne trovo più
        return Numero(int(valore)), resto       # costruisco il numero e torno il resto del testo non analizzato
    if primo == '(':                # se invece il primo carattere è una '(' devo riconoscere una espressione
        # torno una espressione cercando: espressione operatore espressione ')'
        arg1, resto1 = analizza(resto)          # con una chiamata ricorsiva riconosco la prima espressione
        operatore, *resto2 = resto1             # nel resto del testo il primo carattere è l'operatore
        if operatore not in '*+-/^':            # se è sbagliato lancio un errore
            raise EspressioneError("mi aspettavo un operatore '*+-/^' invece di "+operatore)
        arg2, resto3 = analizza(resto2)         # analizzo il testo dopo l'operatore
        if resto3[0] != ')':                    # e subito dopo mi aspetto di trovare la ')'
            raise EspressioneError("mi aspettavo una ) e invece ho trovato una "+resto3[0])
        # se tutto è andato bene posso costruire l'espressione e tornare il testo che segue la ')'
        return Espressione(arg1, operatore, arg2), resto3[1:]
    else:                                       # altrimenti dovrebbe essere una variabile (con un solo carattere)
        # torno una Variabile
        return Variabile(primo), resto          # la costruisco e torno il resto dei caratteri che la seguono
```

In [34]:

```
E, resto = analizza('((x+34)-(y/(7^z)))')

print(E)

print(E.calcola(env))

E.show()
```

```
((x + 34) - (y / (7 ^ z)))
43.94169096209912
```

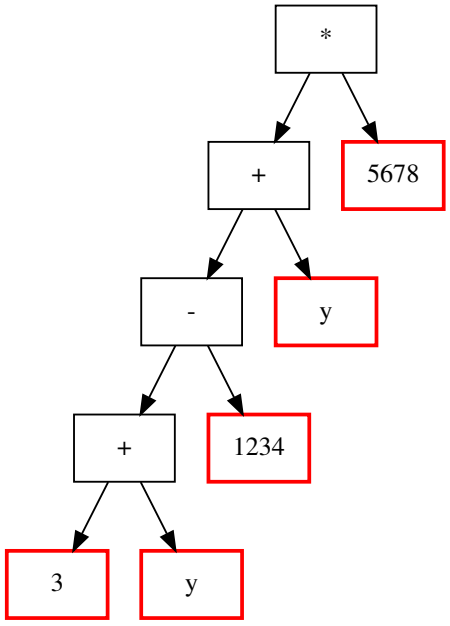
Out[34]:



In [35]:
```
E1,_ = analizza("((((3+y)-1234)+y)*5678)")
print(E1.calcola(env))
E1.show()
```

```
-6762498
```

Out[35]:



cosa possiamo fare con una rappresentazione 'simbolica' di una espressione?

- semplificazione
- derivazione
- ...

In [36]:
```
%%add_to Espressione
def __eq__(self, other):
    return ( isinstance(other, Espressione)
             and
             self._operatore == other._operatore
             and
             self._argomento1 == other._argomento1
             and
             self._argomento2 == other._argomento2
           )
```

In [37]:
```
%%add_to Numero
def __eq__(self, other):
        return ( isinstance(other, Numero)
                 and
                 self._valore == other._valore)
```

In [38]:
```
%%add_to Variabile
def __eq__(self, other):
        return ( isinstance(other, Variabile)
                 and
                 self._nome == other._nome)
```

In [39]:
```
analizza('(x+3)') == analizza('(x-3)')
```

Out[39]: False

In [40]:
```
%%add_to Numero
def semplifica(self):
    return Numero(self._valore)
```

In [41]:
```
%%add_to Variabile
def semplifica(self):
    return Variabile(self._nome)
```

In [42]:

```
%%add_to Espressione
def semplifica(self):
    arg1 = self._argomento1.semplifica()
    arg2 = self._argomento2.semplifica()

    # FIXME:
    # se i due argomenti sono numeri posso direttamente calcolare il valore
    if isinstance(arg1, Numero) and isinstance(arg2, Numero):
        E = Espressione(arg1, self._operatore, arg2)
        return Numero(E.calcola({}))

    if self._operatore == '+':
        if arg1 == Numero(0):
            return arg2
        if arg2 == Numero(0):
            return arg1
    if self._operatore == '-':
        if arg2 == Numero(0):
            return arg1
    if self._operatore == '*':
        if arg1 == Numero(0):
            return Numero(0)
        if arg2 == Numero(0):
            return Numero(0)
        if arg1 == Numero(1):
            return arg2
        if arg2 == Numero(1):
            return arg1
    if self._operatore == '/':
        if arg1 == Numero(0) and arg2 != Numero(0):
            return Numero(0)
        if arg2 == Numero(0):
            raise DivisionePerZeroError()
        if arg2 == Numero(1):
            return arg1
    if self._operatore == '^':
        if arg2 == Numero(1):
            return arg1
        if arg2 == Numero(0):
            return Numero(1)
        if arg1 == Numero(1):
            return Numero(1)
        if arg1 == Numero(0):
            return Numero(0)
    return Espressione(arg1, self._operatore, arg2)


```

In [43]:

```
E,_ = analizza('(3^(1+2))')
E.semplifica().show()
```

Out[43]:

27