lezione21

December 4, 2023

1 Fondamenti di Programmazione

Andrea Sterbini

lezione 21 - 4 dicembre 2023

2 NOTE

- giovedì no lezione

3 RECAP:

- Albero di gioco del Filetto

4 Espressioni algebriche e loro manipolazione

Esempio: **(((2+y)*(3^y))-(1/5)**

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

[34]:
```
from pygraphviz import AGraph
class GraphvizNode:
    _num_nodi = 0
    def __init__(self):
        GraphvizNode._num_nodi += 1
        self.__id = GraphvizNode._num_nodi              # ogni nodo creato ha␣
↪id diversa
        self._sons = []

    def _dot(self, G:AGraph) -> None:
        "Costruisco la rappresentazione dell'albero da visualizzare con␣
↪Graphviz"
        if self._sons:
            G.add_node(self.__id, label=self.label()) # se nodo interno colore␣
↪nero
        else:
```

```
            G.add_node(self.__id, label=self.label(), color='red',␣
 ↪style='bold') # coloro le foglie di rosso
        for son in self._sons:
            G.add_edge(self.__id, son.__id)
            son._dot(G)

    def label(self) -> str:
        return str(self) # f"{self} ({self.__id})"    # per default prendo la␣
 ↪__repr__ del nodo

    def show(self) -> AGraph:
        G = AGraph(rankdir='TD')
        G.node_attr['shape'] = 'box'
        self._dot(G)
        G.layout('dot')
        return G
```

[35]:
```
import jdc
```

[36]:
```
# 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   # idem

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

class EspressioneAlgebricaAstratta(GraphvizNode):
    def calcola(self, variabili : None|dict[str,int|float] = None) -> int|float:
        raise NotImplementedError

class Numero(EspressioneAlgebricaAstratta):
    '''
    Un numero contiene un valore numerico
    '''
    def __init__(self, valore : int|float ):
        super().__init__()
        self._valore = valore

    # --- stampa della espressione algebrica da un albero
```

```
    def __repr__(self) -> str:
        "il testo che rappresenta questo oggetto non è altro che il valore come␣
↪stringa"
        return str(self._valore)
```

[37]:
```
print(Numero(12))
Numero(12).show()
```

```
12
```

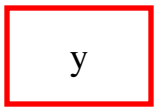
[37]:

[38]:
```
class Variabile(EspressioneAlgebricaAstratta):
    def __init__(self, nome : str):
        "una variabile ha un nome (stringa)"
        super().__init__()
        self._nome = nome

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

[39]:
```
print(Variabile('y'))
Variabile('y').show()
```

```
y
```

[39]:



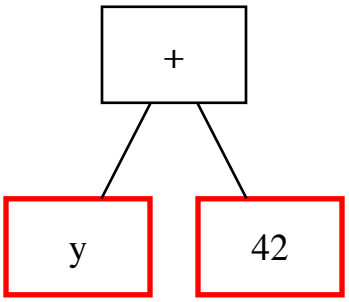
[41]:
```
class Espressione(EspressioneAlgebricaAstratta):
    def __init__(self, argomento1 : GraphvizNode,
                 operatore : str, argomento2: GraphvizNode):
        "una Espressione ha sempre due argomenti ed un operatore (+-*/^)"
        super().__init__()
        self._operatore = operatore
        self._argomento1 = argomento1
        self._argomento2 = argomento2
        self._sons = [argomento1, argomento2]
```

```
    # --- stampa della espressione algebrica come stringa
    def __repr__(self) -> str:
        #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) -> str:         # etichetta da mostrare nell'albero
        return self._operatore
```

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

```
(y + 42)
```

[42]:



[43]:
```
%%add_to Numero
def calcola(self, _env=None) -> int|float :
    return self._valore
```

[44]:
```
Numero(666).calcola()
```

[44]:
```
666
```

[45]:
```
%%add_to Variabile
# calcolo del valore della variabile
def calcola(self, environment : dict[str,int|float]) -> int|float:
    "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")
```

[48]:
```
Variabile('y').calcola({'y':90, 'x':55})
```

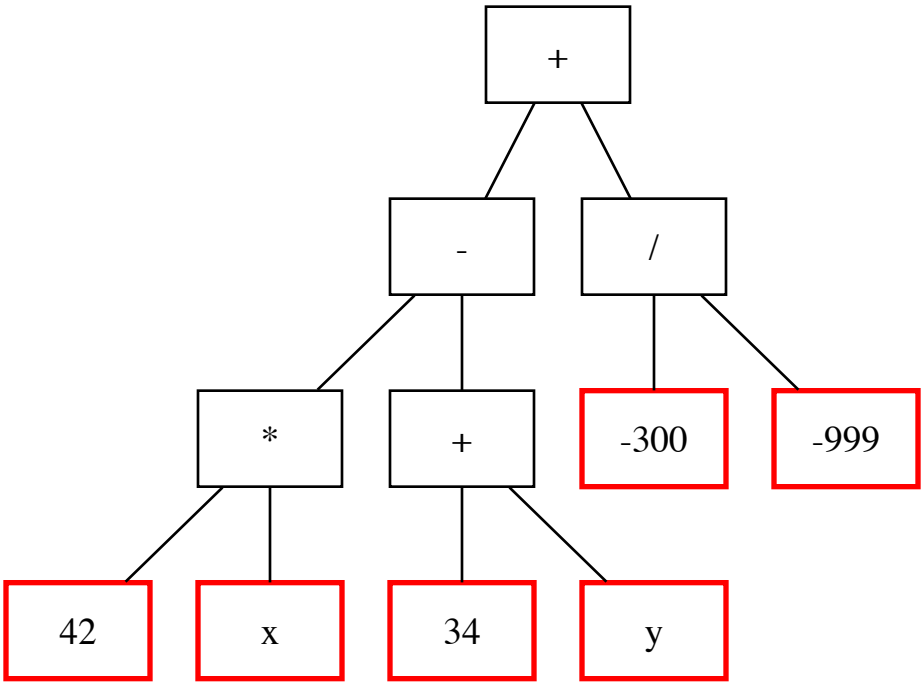
[48]: 90

[49]:
```
%%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 : dict[str,int|float] ) -> int|float:
    "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
```

[50]:
```
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))  # type: ignore
e5.show()
```

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

[50]:



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

Scegliamo una sintassi facile, in cui basta guardare il prossimo carattere - se inizia per cifra c'è un numero -> leggiamo le altre cifre - se inizia per lettera è una variabile (per ora di una sola lettera) - 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 per decidere quale dei tre casi

Una volta riconosciuto un frammento completo ci serve sapere cosa ancora resta da analizzare, quindi torniamo - l'espressione ottenuta (un albero) dall'analisi della prima parte del testo - il resto della stringa da esaminare (per completare chiamate ricorsive precedenti)

[51]:
```
# 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) -> tuple[EspressioneAlgebricaAstratta,str]:
    # 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 nel seguito ci sono ancora␣
↪cifre
            valore += resto.pop(0)                # tolgo la prima cifra e la␣
↪concateno al numero che sto raccogliendo
        # 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(f"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(f"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 # FIXME: dovrei controllare che sia una lettera␣
↪alfabetica?
        return Variabile(primo), resto           # la costruisco e torno il␣
↪resto dei caratteri che la seguono
```

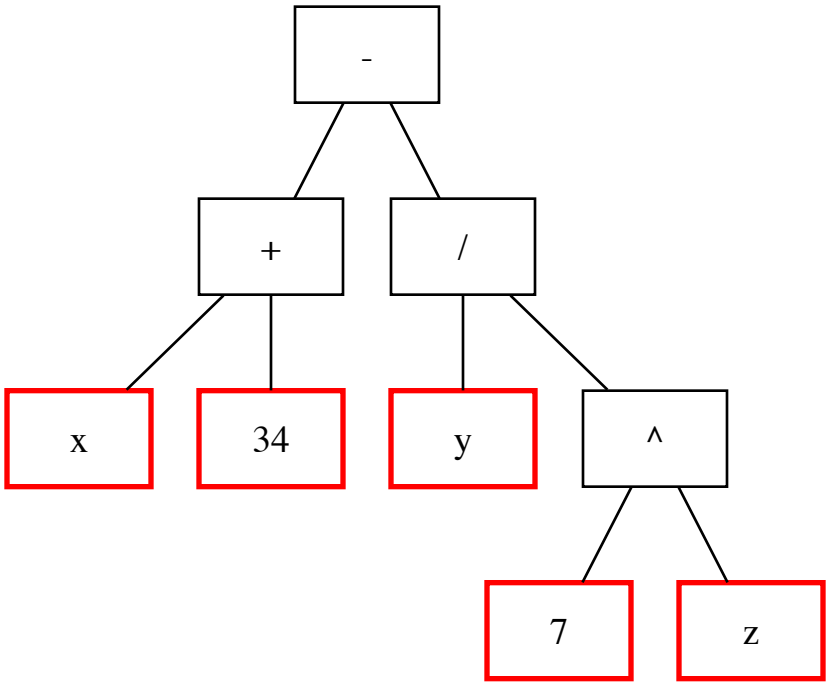
[52]:
```
E, resto = analizza('((x+34)-(y/(7^z)))')

print(E)
```

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

[53]:
```
print(E.calcola(env))

E.show()
```

```
43.94169096209912
```

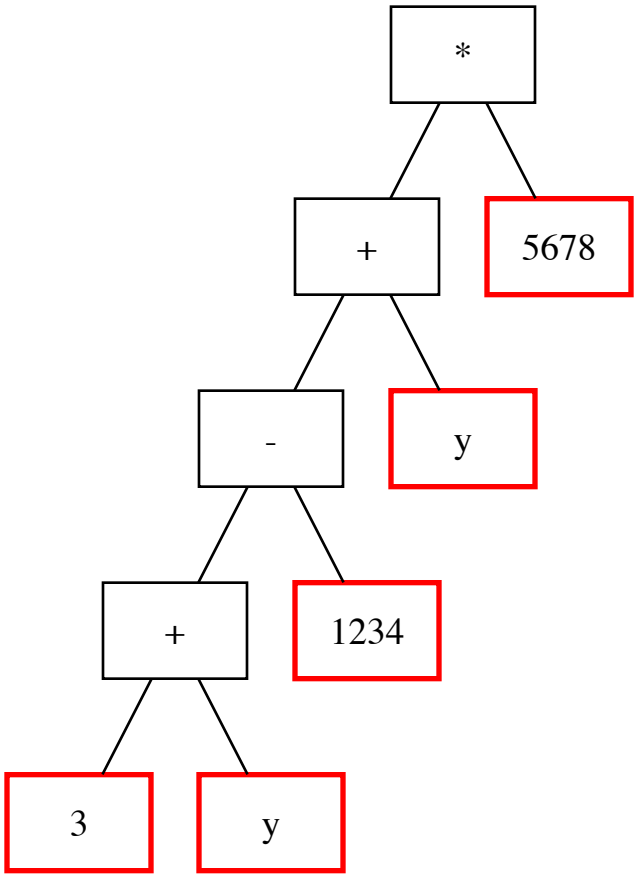
[53]:



[54]:
```
E1,_ = analizza("((((3+y)-1234)+y)*5678)")
print(E1.calcola(env))
E1.show()
```

```
-6762498
```

[54]:



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

- semplificazione
- derivazione
- ...

4.3 Fa comodo poter confrontare due espressioni per vedere se sono uguali

- due Numero sono ugueli se i valori sono uguali
- due Variabile sono uguali se i nomi sono uguali
- due Espressioni sono uguali se hanno lo stesso operatore e operandi uguali

[55]:
```
%%add_to Numero
def __eq__(self, other):
        return ( isinstance(other, Numero)
                and
                self._valore == other._valore)
```

[56]:
```
%%add_to Variabile
def __eq__(self, other):
```

```
        return ( isinstance(other, Variabile)
                and
                self._nome == other._nome)
```

[57]:
```
%%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
           )
```

[60]:
```
# Esempio
analizza('(x+3)') == analizza('(x+3)')
```

[60]:
```
True
```

4.4 Semplificazione algebrica

- se Numero o Variabile si resta uguale

[61]:
```
%%add_to Numero
def semplifica(self):
    "semplificazione algebrica, i numeri vengono ricreati così come sono"
    return Numero(self._valore)
```

[62]:
```
%%add_to Variabile
def semplifica(self):
    "semplificazione algebrica, le variabili vengono ricreate così come sono"
    return Variabile(self._nome)
```

4.4.1 se Espressione

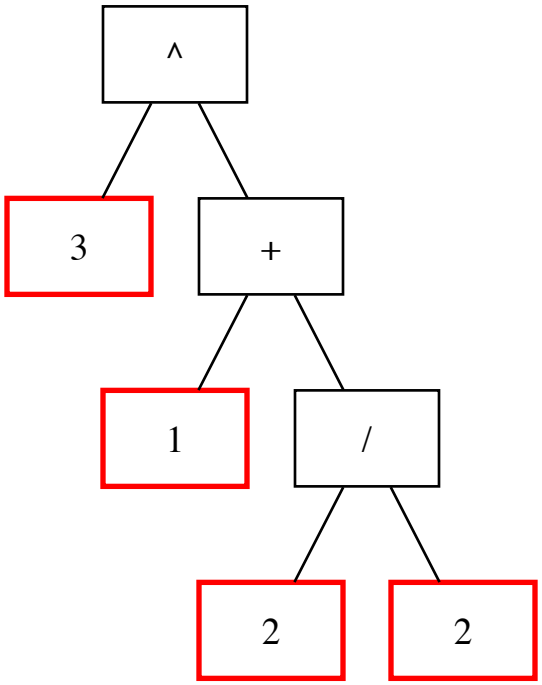
- semplifico gli operandi
- se sono numeri posso calcolare il valore
- altrimenti applico una serie di regole di semplificazioni semplici
- oppure resta uguale (ma con i nuovi operandi semplificati)

[65]:
```
%%add_to Espressione
def semplifica(self):
    "semplificazione algebrica, le variabili vengono ignorate"
    arg1 = self._argomento1.semplifica()             # semplifico il primo␣
 ↪operando
    arg2 = self._argomento2.semplifica()             # semplifico il secondo␣
 ↪operando
```

```
  # 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({}))                    # si ignorano le variabili

  if self._operatore == '+':                          # SOMMA
      if arg1 == Numero(0):    return arg2            # 0 + E = E
      if arg2 == Numero(0):    return arg1            # E + 0 = E
      if arg1 == arg2:
          return Espressione(Numero(2),'*',arg2)     # E + E = 2E

  if self._operatore == '-':                          # SOTTRAZIONE
      if arg2 == Numero(0):    return arg1            # E - 0 = E
      if arg1 == arg2:         return Numero(0)       # E - E = 0
      if arg1 == Numero(0):                           # 0 - E = -1 * E
          return Espressione(Numero(-1),'*',arg2)

  if self._operatore == '*':                          # MOLTIPLICAZIONE
      if arg1 == Numero(0):    return Numero(0)       # 0 * E = 0
      if arg2 == Numero(0):    return Numero(0)       # E * 0 = 0
      if arg1 == Numero(1):    return arg2            # 1 * E = E
      if arg2 == Numero(1):    return arg1            # E * 1 = E
      if arg1 == arg2:
          return Espressione(arg1,'^',Numero(2))     # E * E = E^2

  if self._operatore == '/':                          # DIVISIONE
      if arg1 == Numero(0) and arg2 != Numero(0):     # 0 / E = 0   # se E != 0
          return Numero(0)
      if arg2 == Numero(0):                           # E / 0 -> errore
          raise DivisionePerZeroError()
      if arg2 == Numero(1):    return arg1            # E / 1 = E
      if arg1 == arg2 != Numero(0):                   # E / E = 1   # se E != 0
          return Numero(1)

  if self._operatore == '^':                          # POTENZA
      if arg2 == Numero(1):    return arg1            # E ^ 1 = E
      if arg2 == Numero(0):    return Numero(1)       # E ^ 0 = 1 FIXME: 0^0␣
↪non va bene
      if arg1 == Numero(1):    return Numero(1)       # 1 ^ E = 1
      if arg1 == Numero(0):    return Numero(0)       # 0 ^ E = 0 FIXME: 0^0␣
↪non va bene

  # if nothing applies .... rebuild it with simplified arg1 and arg2
  return Espressione(arg1, self._operatore, arg2)
```

[67]:
```
E,_ = analizza('(3^(1+(2/2)))')
E.show()
```

[67]:



[68]:
```
E.semplifica().show()
```

[68]:

5 e se vogliamo usare anche alcune variabili?

- prima sostituiamo le variabili (con **sostituisci**)
- poi semplifichiamo il risultato

[69]:
```
%%add_to Numero
def sostituisci(self, _):
    "non c'è nulla da sostituire in un numero"
    return self
```

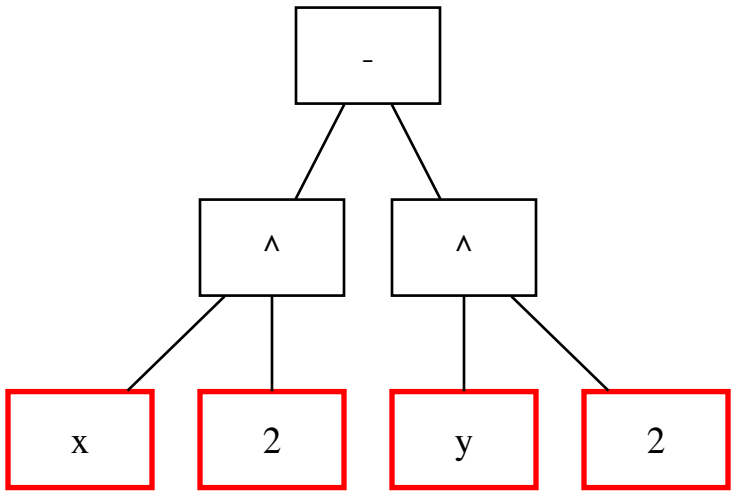
[70]:
```
%%add_to Variabile
def sostituisci(self,dizionario):
```

```
    "sostituzione di una variabile dato un dizionario di sostituzioni␣
↪variabile->numero oppure variabile->espressione"
    if self._nome not in dizionario:              # se il nome non è nel␣
↪dizionario si resta come s'era
        return self
    sostituzione = dizionario[self._nome]         # altrimenti
    if isinstance(sostituzione, (int,float)):     # se il dizionario contiene␣
↪un valore numerico
        return Numero(sostituzione)               # lo ritorno
    elif isinstance(sostituzione, str):           # se il dizionario contiene␣
↪una stringa, è una espressione
        if self._nome in sostituzione:            # la espressione sostituita␣
↪NON PUO' contenere la stessa variabile
            raise Error("una sostituzione di variabile non può contenere la␣
↪stessa variabile")
        # converto la stringa in una espressione analizzando la stringa
        espressione, _ = analizza(sostituzione)   # FIXME: controllare che non␣
↪resti nulla
        return espressione                        # e la ritorno
```

[71]:
```
%%add_to Espressione
def sostituisci(self,dizionario):
    arg1 = self._argomento1.sostituisci(dizionario)
    arg2 = self._argomento2.sostituisci(dizionario)
    return Espressione(arg1, self._operatore, arg2)
```

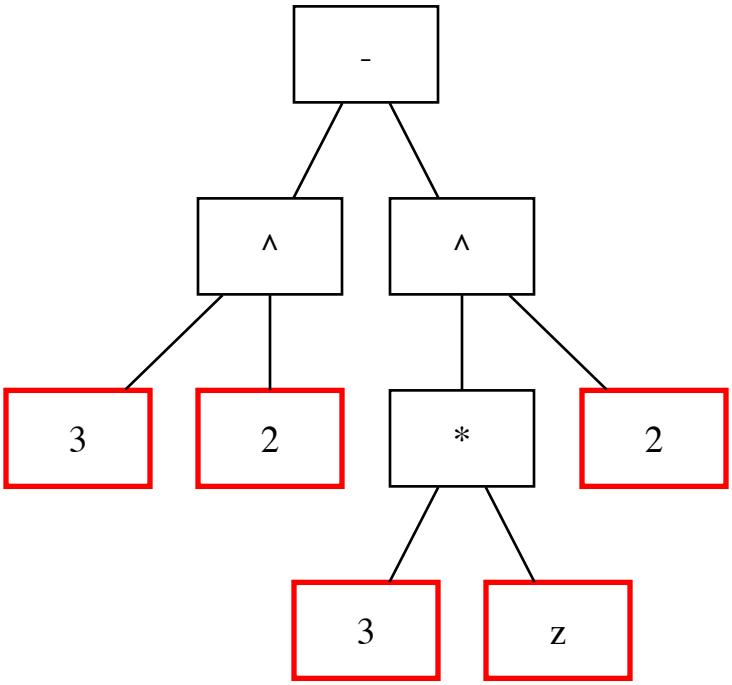
[72]:
```
E,_ = analizza('((x^2)-(y^2))')
E.show()
```

[72]:



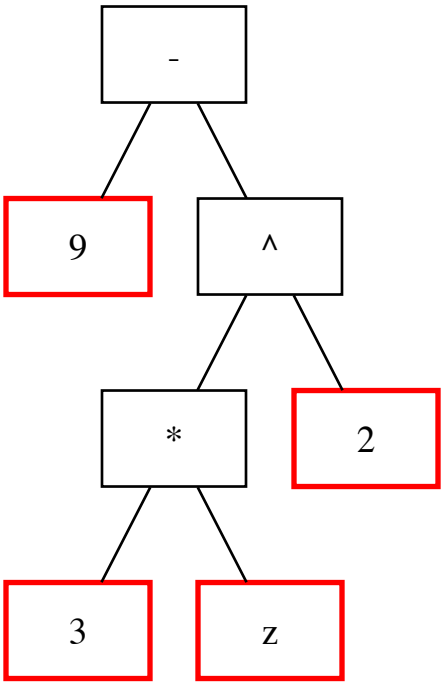
[73]:
```
E1 = E.sostituisci({'x':3, 'y':'(3*z)'})
E1.show()
```

[73]:



[74]:
```
E1.semplifica().show()
```

[74]:



6 TODO:

- altre semplificazioni su espressioni più ampie?
  - $((E1 * E2)^X) = ((E1^X) * (E2^X))$
  - $(E^{(-1*X)}) = (1/(E^X))$
  - $(E^X) * (E^Y) = E^(X + Y)$
  - $(E^X)^Y = E^{(X*Y)}$
  - $((X * E) + (Y * E)) = (E * (X + Y))$
  - $((X/E) + (Y/E)) = ((X + Y)/E)$ (ma solo se E!=0)
  - ...
- derivazione simbolica?