Soluzioni degli esercizi

Queste soluzioni sono proposte soprattutto per favorire un'acquisizione progressiva delle conoscenze. Bisogna partire dall'assunto che esse *non* siano le uniche o le migliori. Prima di studiarle, ognuno deve cercare in autonomia le *proprie*, che potranno anche essere molto diverse da quelle proposte. Alcune delle soluzioni seguenti potrebbero essere incomplete e presentare solo alcune idee per risolvere gli aspetti più critici del problema.

Esercizi capitolo 12 - Strutture ricorsive

Documenti e cartelle

```
ball = Document("ball.gif", "an image")
data = Folder("data", [ball])
a1_0 = Document("a1.txt", "bla bla 0")
cmpt166 = Folder("cmpt166", [a1_0, data])
a1_1 = Document("a1.txt", "a different file")
macm101 = Folder("macm101", [a1_1])
desktop = Folder("Desktop", [cmpt166, macm101])
```

Un nodo astratto può essere in concreto un documento o una cartella. Ciascuna cartella contiene una lista di sottonodi.

Dimensione delle cartelle

```
class Node:
    def size(self):
        raise NotImplementedError("Abstract method")
    def print(self, indent: int):
        raise NotImplementedError("Abstract method")

class Document(Node):  # ...
    def size(self) -> int:
        return len(self._text)
    def print(self, indent: int):
        print(" " * indent + self._name)

class Folder(Node):  # ...
    def size(self) -> int:
```

```
        total_size = 0
        for n in self._subnodes:
            total_size += n.size()
        return total_size
    def print(self, indent: int):
        print(" " * indent + self._name)
        for n in self._subnodes:
            n.print(indent + 4)
```

https://fondinfo.github.io/play/?exs/c12_folders.py

La dimensione di ciascun sottonodo viene ottenuta usando il suo metodo `size`. La stampa di ciascun sottonodo viene ottenuta usando il suo metodo `print`. I metodi `size` e `print` sono quindi polimorfi e ricorsivi. L'indentazione della stampa è proporzionale all'annidamento dei nodi.

Notazione polacca

```
def to_infix(tokens: list) -> str:
    token = tokens.pop(0)

    if "0" <= token[-1] <= "9":
        return token
    else:
        a = to_infix(tokens)
        b = to_infix(tokens)
        return f"({a} {token} {b})"

def evaluate(tokens: list) -> float:
    token = tokens.pop(0)

    if "0" <= token[-1] <= "9":
        return float(token)
    else:
        a = evaluate(tokens)
        b = evaluate(tokens)

        if token == "+": return a + b
        elif token == "*": return a * b  # ...

def main():
    polish = "mod + * + 1 2 + 2 3 4 5".split()
    infix = to_infix(polish[:])
    value = evaluate(polish[:])
```

https://fondinfo.github.io/play/?exs/c12_polish.py

Per valutare una operazione tra due operandi, bisogna prima valutare ciascun operando, ricorsivamente. Infatti, ciascun operando è a sua volta una espressione.

Similmente, per rappresentare una operazione tra due operandi, bisogna prima ottenere la rappresentazione di ciascun operando.

Espressione polacca

```
from operator import add, sub, mul, truediv, neg
ops = {"+": add, "-": sub, "*": mul, "/": truediv, "~": neg}

class Expr:  # ...
    def prefix(self) -> str:
        raise NotImplementedError("Abstract method")

class BinaryOp(Expr):  # ...
    def prefix(self):
        x = self._x.prefix()
        y = self._y.prefix()
        return f"{self._op} {x} {y}"

def main():
    prod1 = BinaryOp("*", Var("x"), Num(2))  #          *  (prod2)
    sum1 = BinaryOp("+", Num(4), prod1)      #         / \
    prod2 = BinaryOp("*", sum1, Num(5))      #        5   +  (sum1)
    print(prod2.eval({"x": 3}))              #           / \
    print(prod2.prefix())                    # (prod1)  *   4
                                             #         / \
                                             #        x   2
```

https://fondinfo.github.io/play/?exs/c12_expr.py

La rappresentazione di ciascun operando viene ottenuta usando il suo metodo `prefix`. Il metodo `prefix` è quindi polimorfo e ricorsivo. In maniera simile, ma cambiando la posizione dell'operatore, si può generare anche la notazione infissa.

Albero da stringa

```
def parse(tokens: list) -> str:
    token = tokens.pop(0)

    if "0" <= token[-1] <= "9":
        return Num(float(token))
    else:
        a = parse(tokens)
        b = parse(tokens)

        if token in "+-*/":
            return BinaryOp(token, a, b)
```

https://fondinfo.github.io/play/?exs/c12_parse.py

Il codice è molto simile a quello del primo esercizio. Nei vari casi, bisogna creare il giusto tipo di nodo, come oggetto.