Linguaggi e Traduttori

simi

February 2025

Chapter 1

Introduzione

Inizio 8.45 fine 10.15

Quello che vedremo noi sono gli interpreti e in particolare riusciremo a fare un eseguibile usando un LLVM (che compila in una sorta di linguaggio macchina IR). Esiste una libreria (che ha fatto lui) che si chiama LibLet che permette di visualizzare l'esecuzione degli algoritmi che vedremo. Tutta questa cosa sarà scritta in Python senza focalizzarci sull'orientazione ad oggetti per vedere che queste cose non si possono fare solo ad oggetti. Vedremo anche un ripassone di algoritmi per vedere alcuni aspetti cruciali nel corso.

1.1 Python

Ci sono alcune strutture dati super comode che si possono usare con una sintassi comoda che sono le liste (strutture non omogenee), gli insiemi e i dizionari. Una cosa comoda di questi oggetti è che sono iterabili, in particolare è possibile costruire questi oggetti attraverso un meccanismo di comprehension in cui racchiudiamo tra i simboli sintattici la costruzione degli oggetti attraverso l'iterazione su un altro oggetto:

```
s = [x * x for x in range(10)]
```

Se ci metto una clausola dopo questa viene valutata:

```
# set comprehension: i numeri pari tra gli interi in [0, 9]
s = {x * x for x in range(9) if x % 2 == 0}
```

Si può imporre l'iterazione usando iter, notiamo che è comodo passare una sentinella per cui quando è finita l'iterazione ci ritorna la sentinella (tipicamente None):

```
#iterazione tramite iter/next

it = iter('alcune parole divise da spazi'.split())
while True:
    w = next(it, None)
    if w is None: break
    print(w)
```

Le funzioni sono cittadini del primo ordine, possiamo assegnarle a variabili e passarle ad altre funzioni. Le useremo per implementare in modo economico i visitor, algoritmi ricorsivi che navigano le strutture dati in modo ricorsivo per fare delle cose. In più vediamo le dispatch table (un modo comodo per fare l'object oriented) e la memorizzazione tramite i decoratori.

```
def quadra(x):
    return x * x

def applica(fun, lst):
    return [fun(x) for x in lst]

applica(quadra, [1, 2, 3])
```

Noi useremo molto le liste di liste, perchè ci rappresentano gli alberi e su queste possiamo definire visite.

```
lol = [1, [2, 3], [4, [5, 6]]]

```

```
#applichiamo una funzione scalare f a tutti gli elementi

def visit(f, lol):
    for elem in lol:
        if instance(elem, list):
            visit(f, elem)
        else:
            f(elem)
```

1.1.1 Dispatch table

Cominciamo a vedere un piccolo esempio di parsing di un esperessione. La prima parte del parsing è suddividere la struttura lineare del testo in chunck concettuali che non è un lavoro banalissimo:

```
    expr = "3 + 12 * 4 + 1 * 2"
    tokens = iter(expr.split())
```

dopo di che dobbiamo definire la semantica delle operazioni, in qualche modo dobbiamo riassociare quello che osserviamo nel flusso dei token con le nostre interpretazioni. Una dispatch table è quella che associa delle informazioni a delle funzioni:

```
#semantica delle operazioni, tramite la dispatch table
    def somma(x, y):
        return x + y
    def prodotto(x, y);
        return x*y

    DT = {
        '+': somma,
        '*': prodotto
    }
```

A questo punto se voglio valutare l'espressione usiamo in modo iterativo la dispatch table, occhio che non stiamo rispettando le regole aritmetiche, ma associamo sempre a sinistra:

```
reuslt = int(next(tokens))

while True:
    t = next(tokens, None)
    if t is None: break
    of = DT[t]
    result = of(result, int(nxt(tokens)))

result
```

1.1.2 Memorizzazione

Spesso e volentieri ci capiterà di esplorare algoritmi ricorsivi per cui per risolvere un problema con un istanza grande risolveremo il problema su sue istanze più piccole, può accadere che nel processo di spezzamento andiamo a risolvere un sottoproblema che è già

stato risolto da qualcun altro, quindi se non adopero accorgimenti particolari ricalcolo gli stessi risultati, l'esempio tipico è il calcolo di Fibonacci.

L'idea è salvare in una cache i risultati parziali di una funzione,

```
def rendi_verbosa(f):
    def f_verbosa(x):
        result = f(x)
        print(f'f({x}) = {result}')
        return result
    return f_verbosa

def quadrato(x):
    x*x

    quadrato_verboso = rendi_verbosa(quadrato)

    q = quadrato_verboso(3)

    #tenere da parte i risultati di una funzione
    cache = {}

    def memoize(f):
      def f_memoized(x):
        if x not in cache: cache[x] = f(x)
        return cache[x]
      return f_memoized
```

Esiste uno zucchero sintattico con cui possiamo annotare la funzione per memorizzare i risultati parziali @memoize

```
@memoize
def cubo(x):
  return x ** 3

  cache = {}

  cubo(1), cubo(4), cubo(6)

  cache

  @memoize
  def fib(n):
    if n == 0 or n == 1: return 1
    return fib(n - 1) + fib(n - 2)
```

1.2 Strutture dati

Le strutture dati che vedremo sono:

- alberi
- grafi
- pile/code

Gli alberi li useremo per rappresentare il testo che parseremo utilizzando una gerarchia del testo, e per visitare questa gerarchia utilizzeremo le visite. Un altro algoritmo che useremo è il backtracking che è molto utile per comprendere gli algoritmi di parsing, è una tecnica ricorsiva che tenta di risolvere i problemi con una sorta di forza bruta ma senza infilarsi nelle chiamate ricorsive anche inutili.

1.2.1 Alberi

Gli alberi sono la struttura dati più comune del corso perchè c'è una forte ricorrenza tra linguaggi/parsing e alberi, inoltre non è difficile immaginare che molti concetti che vedremo sono rappresentabili come alberi (l'espressione aritmetica dell'altra volta).

La forma più comune che useremo è la rappresentazione con liste di liste (lol), la prima è la radice e poi ci sono i sotto-alberi con tutti i loro figli.

```
# [radice]
# [radice alberi]

tree = [1, [11, [111]], [12, [121], [122]], [13]]
```

La cosa comoda di python è che c'è l'assegnamento ordinato, viene particolare comodo l'unpacking iterabile, se metto l'* prima della variabile mi prendo tutto quello che resta:

```
    a, *b = 1, 2, 3
    print(b) #[2, 3]
```

Questo è comodo perchè verrà comodo posso prendere i figli:

```
    root, *children = tree
```

Come già detto useremo la libreria Liblet per visualizzare gli alberi partendo dalle lol (liste di liste). É possibile fare l'unpacking anche di questi alberi con la medesima sintassi.

```
from liblet import Tree, side_by_side

t = Tree.from_lol(tree)

root, *children = t
```

Visite

Quando facciamo una visita possiamo scegliere tante strategie, l'unica questione è decidere quando operare sul nodo, possiamo decidere di operare all'inizio (quando lo incontro) oppure occuparmi del nodo al termine della visita ai figli. Gli ordini vengono chiamati:

- pre-ordine: visito prima il nodo e poi i figli
- post-ordine: visito prima i figli e poi il nodo
- per livello

La visita in preordine si implementa in questo modo, posto che prende la funzione che fa i suoi conti:

```
def preorder(tree, root):
    root, *children = tree
    visitor(root)
    for child in children: preorder(child, visitor)
```

Per fare il postordine scambio lo scarico ricorsivo con la funzione che fa i calcoli, ovviamente visualizzeremo nodi in ordine diverso.

```
def postorder(tree, visitor):
    root, *children = tree
    for child in children: preorder(child, visitor)
    visitor(root)
```

Nel caso degli alberi è possibile visitarli anche per livelli, si chiama level order. É un algoritmo che non fa uso della ricorsione, generalmente si adopera una coda in cui accumulo i figli che vedo in modo tale che mi occupo per primo di quelli che metto dentro:

```
    def levelorder(tree, visitor):
        Q = queue()

        Q.enqueue(tree)
        while Q:
            tree = Q.dequeue()
            root, *children = tree
            visitor(root)
            for st in children: Q.enqueue(child)
```

Alberi con attributi

Una cosa che potrebbe essere molto comoda è che potremmo avere bisogno di usare nodi con informazioni più strututrate per poter visualizare alberi arricchiti di attributi. Per fare questo useremo un albero che abbia dict come valori e che conservi il valore numerico come valore della chiave val:

```
 def add_attr(tree):
  root, *children = tree
  return [{'val': root}] + [add_attr(child) for child in children]

tree = [1, [11, [111]], [1200, [121], [122]], [13]]

add_attr(tree)
```

Gli attributi generalmente vengono calcolati, ad esempio la profondità a cui siamo. Il modo in cui si calcolano segue due direzioni:

- Attributi ereditati: calcolati per il papà e ereditati dai figli. Si usa una visita in pre-ordine.
- Attributi sintetizzati: in qualche modo otteniamo prima le informazioni sui figli e poi sintetizziamo verso l'alto (tipicamente la valutazione delle espressioni è un attributo sintetizzato). Si usa una visita in post-ordine, perchè prima devo visitare i figli.

Ad esempio vediamo come spingere in giù la profondità del nodo, (nella visita ricevo che sono figlio di un nodo profondo 4), quello che fa il visitor è aggiornare quella profondità di 1 per vedere quanto è profondo lui:

```
def preorder_with_value(tree, visitor, value = None):
  root, *children = tree
  visitor(root, value)
  for child in children: preorder_with_value(child, visitor,
     root['depth'])

# visitor che aggiunge l'attributo depth (pari a 1 + il valore
   ereditato, il caso None riguarda la radice)

def add_depth(root, value):
  root['depth'] = value + 1 if value is not None else 0

attr_tree = add_attr(tree)

# la radice ricevera None perche e' il valore di default di value

preorder_with_value(attr_tree, add_depth)

Tree.from_lol(attr_tree)
```

Nella sintesi il codice è più semplice da scrivere perchè possiamo usare il return per ritornare il valore al padre:

```
 def postorder_with_return(tree, visitor):
  root, *children = tree
  values = [postorder_with_return(child, visitor) for child in
     children] # sara' la lista vuota se non ci sono figli
  return visitor(root, values)

# visitor che aggiunge l'attributo max (pari al massimo tra il valore
   del nodo e quelli sintetizzati dai figli)

def add_max(root, values):
  root['max'] = max([root['val']] + values)
  return root['max']

attr_tree = add_attr(tree)

postorder_with_return(attr_tree, add_max)

Tree.from_lol(attr_tree)
```

1.2.2 Grafi

Li avremo i grafi come strumenti, meno concretamente ma avremo delle visite per passare da uno stato all'altro ma non scriveremo esplicitamente grafi. Esistono grafi non orientati e orientati. Idealmente noi useremo i grafi orientati. Tipicamente i grafi si rappresentano con:

- Matrice di adiacenza, NxN dove metto true e false dove c'è un collegamento. É O(1) per il costo computazionale ma $O(n^2)$ nello spazio.

- Tengo tutte le coppie di nodi collegati se la matrice è sparsa (lista dei nodi)
- Tengo le liste di adiacenza di ogni nodo, per ogni nodo tengo in una lista tutti i nodi che ha collegati

Le rappresentazioni più comode che usiamo sono la rappresentazione con lista di archi (una tupla di tuple) oppure le liste di adiacenza rappresentata da un dict di set:

```
# dagli archi alla mappa delle adiacenze


# per ogni nodo n (sia s o t), adjacency[n] = set()

adjacency = dict()
for s, t in arcs:
  adjacency[s] = set()
  adjacency[t] = set()

# aggiungo gli outlink

for s, t in arcs: adjacency[s] |= {t}

adjacency
```

Algoritmi

La visita di un grafo è un procedimento che ci porta da un nodo ad esplorare tutti gli altri, ho in qualche modo bisogno di salvarmi da qualche parte i nodi che ho visitato e quelli che devo visitare. Immaginiamo di avere bisogno di una sacca in cui mettere le cose che abbiamo bisogno di fare (ci mettiamo dentro i figli dei nodi), dobbiamo stare attenti a non rimettere dentro quello che ci è già entrato e ha senso tirare fuori gli elementi in un certo ordine:

- FIFO: il primo nodo che metto è il primo che tolgo, i nodi sono messi in una coda. É la storia del level ordre. Così effettuiamo una visita in ampiezza.
- LIFO: last in first out, è una pila. Effettuiamo così una visita in profondità.

Nella visita in profondità si può dare una implementazione ricorsiva (perchè la pila può essere la pila delle chiamate), quello che dobbiamo avere è una struttura dati che ci tiene quello che abbiamo già visto, per evitare di visitare di nuovo gli stessi nodi:

```
def depthfirst(adjacency, start, visit):
 def walk(src):
   visit(src)
   seen.add(src)
   for dst in adjacency[src]:
     if dst not in seen:
       walk(dst)
 seen = set()
 walk(start)
```

La visita in ampiezza usa la coda, toglo dalla coda e lo visito e quando lo vistio lo aggiungo:

```
from liblet import Queue

def breadthfirst(adjacency, start, visit):

  Q = Queue()

  seen = set()
  Q.enqueue(start)
  while Q:
    src = Q.dequeue()
    visit(src)
    seen.add(src)
    for dst in adjacency[src]:
      if dst not in seen:
        Q.enqueue(dst)
```

Quello da notare nelle visite è che quello che succede nelle visite in profondità è che la pila non aumenta tanto di dimensione, l'unico problema è che se devo vedere nodi vicini è possibile che per visitarlo potrei dover fare un sacco di conti prima di arrivare a visitare quello. La visita in ampiezza d'altro canto è vero che procede per livelli, ma se il grafo è denso la coda può diventare molto grande. Quindi quello che dovremo vedere negli algoritmi di parsing è questa scelta, scendo velocemente nel grafo sperando nel successo oppure è meglio scendere per passi? quello che vedremo è che in generale se non ci sono grandi strategie l'ampiezza è l'unica strada mentre se ho qualcuno che mi dice qualcosa è meglio andare giù veloce nella strada giusta.

1.2.3 backtracking

L'idea è che possiamo decidere che alcuni rami dell'albero sono buoni e altri sono cattivi. Quindi l'idea è che se mi trovo in una strada in cui posso prevedere che se sono in un certo punto in cui non c'è speranza che andando avanti nella ricorsione non troverò mai il risultato allora posso tagliare quel ramo dell'albero. Ho una specie di soluzione parziale che posso piazzare.

Ho una soluzione candidata e se sono già capace di sapere se quella soluzione non va bene allora ritorno, se ho trovato il risultato lo stampo. Altrimenti vado avanti e faccio le chiamate ricorsive.

```
def backtrack(candidate):
    if reject(candidate): return
    if accept(candidate): output(candidate)
    s = first(candidate)
    while s:
        backtrack(s)
        s = next(candidate)
```

Come esempio vediamo come si segmentano le parole, che è un problema fondamentale nei motori di ricerca. Inanzitutto mi procuro un elenco di parole, se segmenti non è vuota e l'ultima parola del dizionario non è in WORDS butto via, altrimenti se non mi rimane niente ho trovato tutti, altrimenti provo a spaccare in due la seguente parole in tutti i modi possibili:

```python
from urllib.request import urlopen

# WORDS sono le parole di almeno 2 caratteri (3 conta anche l'a-capo)

with
    urlopen('https://raw.githubusercontent.com/napolux/paroleitaliane/master/paroleit
    as url:
  WORDS = {word.decode().strip().upper() for word in url if len(word)
     >= 3}

print(len(WORDS))

def segmenta(segmenti, resto):
  if segmenti and not segmenti[-1] in WORDS: return
  if not resto:
    print(segmenti)
    return
  for i in range(1, 1 + len(resto)):
    segmenta(segmenti + [resto[:i]], resto[i:])

segmenta([], 'ILCORRIEREDELLASERAEDIZIONENOTTURNA')
```