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:

```
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):

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.

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

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:

```
def somma(x, y):
    return x + y
```

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:

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,

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