

Linguaggi e Traduttori

Simone Petta

A.A. 2024/2025

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:

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

Se ci metto una clausola dopo questa viene valutata:

```
1 # set comprehension: i numeri pari tra gli interi in [0, 9]
2 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):

```
1 #iterazione tramite iter/next
2
3 it = iter('alcune parole divise da spazi'.split())
4 while True:
5     w = next(it, None)
6     if w is None: break
7     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 decorator.

```
1 def quadra(x):
2     return x * x
3
4 def applica(fun, lst):
5     return [fun(x) for x in lst]
6
7 applica(quadra, [1, 2, 3])
```

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

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

```

3 #applichiamo una funzione scalare f a tutti gli elementi
4
5 def visit(f, lol):
6     for elem in lol:
7         if instance(elem, list):
8             visit(f, elem)
9         else:
10            f(elem)

```

1.1.1 Dispatch table

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

```

1 expr = "3 + 12 * 4 + 1 * 2"
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:

```

1 #semantica delle operazioni, tramite la dispatch table
2 def somma(x, y):
3     return x + y
4 def prodotto(x, y):
5     return x*y
6
7 DT = {
8     '+': somma,
9     '*': prodotto
10 }

```

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 result = int(next(tokens))
2
3 while True:
4     t = next(tokens, None)
5     if t is None: break
6     of = DT[t]
7     result = of(result, int(next(tokens)))
8
9 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,

```
1 def rendi_verbosa(f):
2     def f_verbosa(x):
3         result = f(x)
4         print(f'f({x}) = {result}')
5         return result
6     return f_verbosa
7
8 def quadrato(x):
9     x*x
10
11     quadrato_verboso = rendi_verbosa(quadrato)
12
13     q = quadrato_verboso(3)
14
15     #tenere da parte i risultati di una funzione
16     cache = {}
17
18     def memoize(f):
19         def f_memoized(x):
20             if x not in cache: cache[x] = f(x)
21             return cache[x]
22         return f_memoized
```

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

```
1 @memoize
2 def cubo(x):
3     return x ** 3
4
5     cache = {}
6
7     cubo(1), cubo(4), cubo(6)
8
9     cache
10
11 @memoize
12 def fib(n):
13     if n == 0 or n == 1: return 1
14     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 infilarci 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.

```
1 # [radice]
2 # [radice alberi]
3
4 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:

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

Questo è comodo perchè verrà comodo posso prendere i figli:

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

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

```
1 from liblet import Tree, side_by_side
2
3 t = Tree.from_lol(tree)
4
5 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:

```

1 def preorder(tree, root):
2     root, *children = tree
3     visitor(root)
4     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.

```

1 def postorder(tree, visitor):
2     root, *children = tree
3     for child in children: preorder(child, visitor)
4     visitor(root)

```

Nel caso degli alberi è possibile visitarli anche per livelli, si chiama level order. È 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:

```

1 def levelorder(tree, visitor):
2     Q = queue()
3
4     Q.enqueue(tree)
5     while Q:
6         tree = Q.dequeue()
7         root, *children = tree
8         visitor(root)
9         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 strutturate per poter visualizzare 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:

```

1 def add_attr(tree):
2     root, *children = tree
3     return [{'val': root}] + [add_attr(child) for child in children]
4
5 tree = [1, [11, [111]], [1200, [121], [122]], [13]]
6
7 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:

```

1 def preorder_with_value(tree, visitor, value = None):
2     root, *children = tree
3     visitor(root, value)
4     for child in children: preorder_with_value(child, visitor,
5         root['depth'])
6
7 # visitor che aggiunge l'attributo depth (pari a 1 + il valore
8   ereditato, il caso None riguarda la radice)
9
10 def add_depth(root, value):
11     root['depth'] = value + 1 if value is not None else 0
12
13 attr_tree = add_attr(tree)
14
15 # la radice ricevera None perche e' il valore di default di value
16
17 preorder_with_value(attr_tree, add_depth)
18
19 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:

```

1 def postorder_with_return(tree, visitor):
2     root, *children = tree
3     values = [postorder_with_return(child, visitor) for child in
4         children] # sara' la lista vuota se non ci sono figli
5     return visitor(root, values)
6
7 # visitor che aggiunge l'attributo max (pari al massimo tra il valore
8   del nodo e quelli sintetizzati dai figli)
9
10 def add_max(root, values):
11     root['max'] = max([root['val']] + values)
12     return root['max']
13
14 attr_tree = add_attr(tree)
15
16 postorder_with_return(attr_tree, add_max)
17
18 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, $N \times N$ dove metto true e false dove c'è un collegamento. È $O(1)$ 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:

```

1  # dagli archi alla mappa delle adiacenze
2
3
4  # per ogni nodo n (sia s o t), adjacency[n] = set()
5
6  adjacency = dict()
7  for s, t in arcs:
8      adjacency[s] = set()
9      adjacency[t] = set()
10
11 # aggiungo gli outlink
12
13 for s, t in arcs: adjacency[s] |= {t}
14
15 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. È la storia del level order. 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 (perché 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:

```

1  def depthfirst(adjacency, start, visit):
2      def walk(src):
3          visit(src)
4          seen.add(src)
5          for dst in adjacency[src]:
6              if dst not in seen:
7                  walk(dst)
8      seen = set()
9      walk(start)

```

La visita in ampiezza usa la coda, toglie dalla coda e lo visita e quando lo visita lo aggiunge:

```

1 from liblet import Queue
2
3 def breadthfirst(adjacency, start, visit):
4
5     Q = Queue()
6
7     seen = set()
8     Q.enqueue(start)
9     while Q:
10         src = Q.dequeue()
11         visit(src)
12         seen.add(src)
13         for dst in adjacency[src]:
14             if dst not in seen:
15                 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.

```

1 def backtrack(candidate):
2     if reject(candidate): return
3     if accept(candidate): output(candidate)
4     s = first(candidate)
5     while s:
6         backtrack(s)
7         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:

```

1 from urllib.request import urlopen
2
3 # WORDS sono le parole di almeno 2 caratteri (3 conta anche l'a-capo)
4
5 with
6     urlopen('https://raw.githubusercontent.com/napolux/paroleitaliane/master/paroleit
7         as url:
8     WORDS = {word.decode().strip().upper() for word in url if len(word)
9         >= 3}
10
11 print(len(WORDS))
12
13 def segmenta(segmenti, resto):
14     if segmenti and not segmenti[-1] in WORDS: return
15     if not resto:
16         print(segmenti)
17         return
18     for i in range(1, 1 + len(resto)):
19         segmenta(segmenti + [resto[:i]], resto[i:])
20
21 segmenta([], 'ILCORRIEREDELLASERAEDIZIONENOTTURNA')

```

Chapter 2

Linguaggi

2.1 Closed form

L'idea di poter definire un sotto-linguaggio per ciascuno simbolo è molto comodo perché posso partire da un linguaggio e usare quello. Il linguaggio prodotto è il prodotto dei linguaggi, la production independent è una cosa che utilizziamo molto. Le context-free consentono una struttura che viceversa perdiamo se andiamo nei linguaggi regolari ed è la questione del self embedding.

2.1.1 Self embedding

Si intende come regola ricorsiva quando nel lato destro della produzione compare il simbolo non terminale che si sta definendo:

$$A \rightarrow aAa$$

La regola ricorsiva è l'ingrediente necessario per rendere i nostri linguaggi infiniti. Questa regola ci serve ad esempio ad avere il linguaggio di Dyck, che è un linguaggio di parentesi ben formate.

$$A \rightarrow (A)$$

E questo ci porta a definire correttamente un linguaggio di programmazione. Il Context-Free è un ragionevole punto di incontro tra un linguaggio regolare e un linguaggio ricorsivo (basso ed alto livello).

2.2 Alberi di parsing

L'albero cattura la forma gerarchica di un linguaggio, è un modo per rappresentare la struttura di un linguaggio. Sia a fini linguistici sia per rappresentare la struttura di un linguaggio di programmazione.

Prima di poterci dedicare in maniera serena a questi alberi di parsing dobbiamo fare un po' di pulizia perché abbiamo delle grammatiche con difetti che vorremmo eliminare. La spazzatura che può rimanere dentro una grammatica context free è che possiamo avere dei non terminali non definiti. Cioè abbiamo messo dentro delle variabili (lettere maiuscole) che non abbiamo definito, non stanno mai a sinistra di una produzione. Questo è un problema perché non possiamo mai terminare la produzione. Sono inutili e vanno eliminati. Ci sono altre due circostanze in cui ci sono problemi, potremmo avere definito un terminale che non è mai raggiungibile da nessuna produzione, questi si chiamano simboli irraggiungibili e vanno eliminati. Infine potremmo avere una produzione ricorsiva ma in questo caso non possiamo mai avere una produzione vuota con questa variabile, questa si chiama variabile improduttiva e va eliminata. Un'altra circostanza non bella è avere un loop per derivare delle variabili.

2.2.1 Pulizia di una grammatica

1. Eliminare i non terminali non definiti
2. Eliminare i simboli irraggiungibili

3. Eliminare le variabili improduttive

4. Eliminare i loop

Quello che chiediamo è che una variabile sia derivabile non subito ma dopo un numero n di passi. Quello che chiediamo è una chiusura della funzione, dato un insieme applico f in modo ricorsivo e se questa f è chiusa significa che prima o poi arrivo ad un insieme tale per cui se applico ancora la funzione a quell'insieme rimango in quell'insieme. Un esempio di funzione chiusa è se aumento sempre gli oggetti dell'insieme ma gli oggetti fanno parte di un insieme finito.

Dentro liblet c'è un decoratore `@closure` che ci permette di fare la chiusura di una funzione. È chiaro che con una funzione di questo tipo la pulizia diventa abbastanza semplice. Vediamo un esempio di una grammatica sporca:

```
1 G = Grammar.from_string("""
2 S -> A B | D E
3 A -> a
4 B -> b C
5 C -> c
6 D -> d F
7 E -> e
8 F -> f D
9 """)
10 G
```

Le regole produttive si possono definire, in modo bottom up detrimino le produttive, diventa produttivo a sinistra quello che a destra ha tutte cose produttive:

```
1 def find_productive(G):
2
3     @closure
4     def find(prod):
5         return prod | {A for A, a in G.P if set(a) <= prod}
6
7     return find(G.T)
8
9 find_productive(G)
```

Le raggiungibili invece si possono ottenere con un processo top down, parto dal simbolo distinti e metto dentro tutti i non terminali a quali posso arrivare da qualcosa di raggiungibile:

```
1 from liblet import union_of
2
3 def find_reachable(G):
4
5     @closure
6     def find(reach):
7         return reach | union_of(set(a) for A, a in G.P if A in reach)
8
9     return find({G.S})
10
11 find_reachable(G)
```

Dopo aver definito questo posso pulire la grammatica, garantisco che tutti i simboli sono produttivi e raggiungibili:

```

1 def remove_unproductive_unreachable(G):
2     Gp = G.restrict_to(find_productive(G))
3     return Gp.restrict_to(find_reachable(Gp))
4
5 remove_unproductive_unreachable(G)

```

Attenzione che l'ordine con cui si fa questa operazione è cruciale, se eliminiamo prima i non raggiungibili e poi i non produttivi potrei avere la necessità di dover fare un'altra passata per eliminare altri non raggiungibili.

2.2.2 Dimensione degli alberi di parsing

Ha senso ragionare sulla dimensione degli alberi di parsing? sì perchè se fossero enormi non avrebbero una utilità pratica. La storia è molto semplice ed è legata al fatto che in buona sostanza la frontiera di un albero binario è lineare nel numero di nodi con N nodi abbiamo $O(N)$ foglie. Quindi quello che vogliamo dimostrare che se prendiamo una grammatica non malata un albero di derivazione non può contenere più di N nodi. Quello che facciamo è raginare bottomup, tutte le volte che vengo verso l'alto e faccio un passo agglomerativo (agglomerato con un non terminale) un nodo lo aggiungo ma almeno due ne tolgo, il che vuol dire che se questa cosa allora ho introdotto N nodi e ne ho tolti $2N$ quindi ho inserito linearmente N nodi. Le cose che mi restano da guardare è cosa succede nel caso di regole unitarie, avendo regole unitarie al massimo si va ad esplodere nella dimensione dei non terminali, perchè significa che arrivo ad un agglomeratore tramite una catena.

2.2.3 Derivazioni

Non è detto che una parola derivata abbia sempre la stessa derivazione, ci sono tante derivazioni possibili da una grammatica per la stessa parola. Le derivazioni in termini di alberi di parsing vediamo che per due derivazioni diverse abbiamo lo stesso albero di parsing ed è una situazione spiacevole perchè ci sarebbe piaciuto avere una mappa 1-1, la presenza di più derivazioni più essere più o meno critica a seconda del contesto. Ci possono essere due derivazioni che hanno due alberi di parsing diversi che è la situazione che ci preoccupa, il primo caso è facilmente risolvibile indicando delle derivazioni preferibili (nel primo caso abbiamo per finta più derivazioni perchè alla fine cambia solo l'ordine). Nel secondo caso abbiamo più alberi di parsing per più derivazioni.

```

1 # una grammatica banale per il linguaggio {a^n b^n | n > 0}
2
3 G_ab = Grammar.from_string('''
4 S -> A B
5 A -> a A | a
6 B -> b
7 ''')
8 G_ab
9
10 # due possibili derivazioni
11
12 ab_0 = Derivation(G_ab).step(
13     [(0, 0), (1, 0), (2, 1), (3, 2)]
14 )

```

```

15 ab_1 = Derivation(G_ab).step(
16     [(0, 0), (3,1), (1,0), (2,1)]
17 )
18
19 ab_0, ab_1
20
21 # ma a ben guardare lo stesso albero
22
23 side_by_side(
24     ProductionGraph(ab_0),
25     ProductionGraph(ab_1),
26 )

```

2.2.4 Dalla derivazione all'albero di parsing

Cominciamo a ragionare sul fatto che non è così ovvio il legame tra le parole, le derivazioni e gli alberi di parsing. Adesso cerchiamo di convincerci che almeno uno di questi pezzi è facilmente raggiungibile, esiste un modo semplice data una derivazione costruire un albero di parsing. Con Python posso tenermi la forma sentenziale conservando tutti gli alberi che mano a mano da questa forma sviluppo, la prima forma è il simbolo distinto e poi per ogni passo di derivazione mi dice quale pezzo della forma sentenziale va sostituito e noi sostituiamo questa con un nodo nell'albero. Sostanzialmente per ogni passaggio di derivazione sostituisco mettendo i nodi nell'albero, nella forma sentenziale tengo sempre le foglie e sopra metto da dove derivo. La seguente procedura memorizza in tree l'albero di derivazione e in frontier la sua frontiera, corrispondente alla forma sentenziale a cui è giunta la derivazione (di passo in passo) come una lista di alberi annotati.

Ciascun nodo dell'albero ha due etichette: Symbol che si riferisce ad uno dei simboli della grammatica e prord pari ad una produzione. I figli di ciascun nodo dell'albero hanno i simboli contenuti nel lato destro di prord.

Gli alberi vengono "completati" man mano che la procedura elabora i passi della derivazione; al termine le foglie degli alberi saranno simboli terminali (e prord sarà convenzionalmente definito come None).

```

1  def derivation_to_parsetree(d):
2
3      # questa variabile si riferira all'albero di derivazione di d
4      # inizialmente contiene l'albero annotato col simbolo di partenza
5      tree = Tree({'Symbol': d.G.S, 'prod': None})
6
7      # all'inizio la forma sentenziale e data da tale albero
8      frontier = [tree, ]
9
10     for nprod, pos in d.steps():
11
12         # l'albero da completare e dato dalla posizione in cui e
13         # applicata la produzione
14         curr = frontier[pos]
15
16         # risalgo dal numero alla produzione
17         prod = d.G.P[nprod]
18
19         # i figli sono dati dal lato destro d.G.P[prod].rhs

```



```

19     children = [Tree({'Symbol': X, 'prod': None}) for X in prod.rhs]
20
21     # aggiorniamo l'albero da completare
22     curr.root['prod'] = prod
23     curr.children = children
24
25     # aggiorniamo la forma sentenziale
26     frontier = frontier[:pos] + children + frontier[pos + 1:]
27
28     return tree

```

Per tornare indietro (dall'albero alla derivazione) posso fare una visita in pre-ordine arriviamo ad una derivazione left most:

```

1  def leftmost(tree):
2      return [tree.root['prod']] + [prod for child in tree.children for
3          prod in leftmost(child) if prod]
4
5  lm_0 = leftmost(pt_0)
6  lm_0

```

Da notare che se facciamo una visita in post-ordine non otteniamo una right most, ma una right most al contrario. Questo tipo di ambiguità è ineliminabile. Questo è logicamente un problema quando abbiamo operatori non associativi (es. sottrazione e divisione).

Quello che dovremo fare quando scriveremo una grammatica:

1. Non è sempre possibile
2. Non è automatico
3. Introduco N non "semantici": è chiaro che poi questi N andranno eliminati perchè producono alberi di parsing pieni di mondezze

2.2.5 Possibili soluzioni

La ricorsione a sinistra determina un'associatività a sinistra mentre se è a destra si usa una ricorsione a destra. Quello che facciamo è inserire dei simboli in più nel linguaggio per gestire tutti i casi:

```

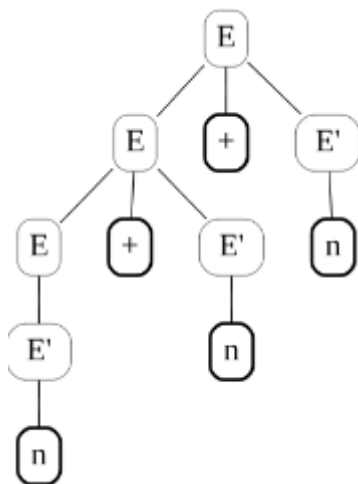
1  G_la = Grammar.from_string("""
2  E -> E + E' | E'
3  E' -> n
4  """)
5  G_la

```

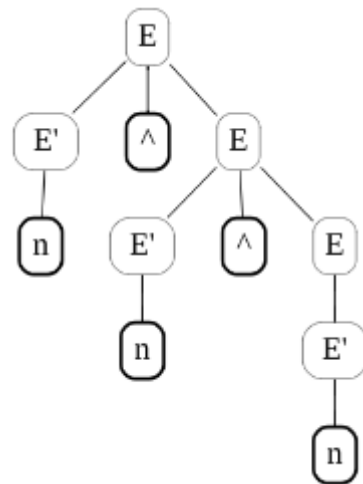
```

1  G_ra = Grammar.from_string("""
2  E -> E' ^ E | E'
3  E' -> n
4  """)
5  G_ra

```



(a) Associatività Sinistra



(b) Associatività Destra

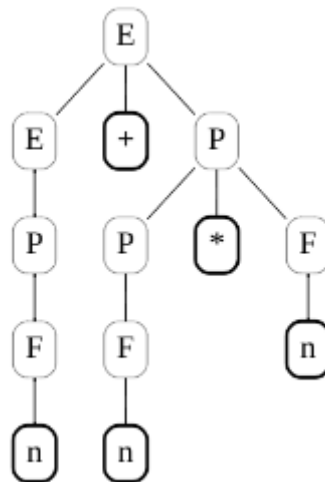
Figure 2.1: Confronto tra associatività sinistra e destra

Quando ho la menata della precedenza, operatori binari con precedenza diversa, questo si traduce con l'introduzione di simboli ulteriori abbiamo ora n . Questi simboli sono terminali che rappresentano la precedenza degli operatori.

```

1  G_p = Grammar.from_string("""
2  E -> E + P | P
3  P -> P * F | F
4  F -> n
5  """)

```



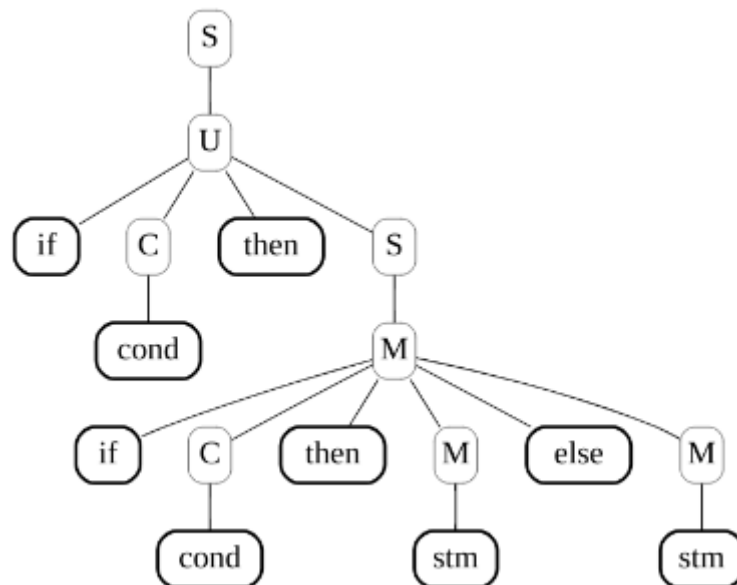
Nel caso del dangling else (if else if) diventa una cosa ancora più dolorosa c'è un'interplate tra SMUC che è difficile da ricordare risolve la cosa perché tiene l'else legato all'if più vicino.

```

1  G_if = Grammar.from_string("""
2  S -> M | U
3  M -> if C then M else M | stm
4  U -> if C then M else U | if C then S

```

```
5 C -> cond
6 """)
```



Il punto cruciale è che context free siamo al giusto livello (non ci siamo persi le parentesi sotto) abbiamo alberi di parsing lineari nella lunghezza della parola, ma dobbiamo stare attenti che avremo ambiguità ed in questo caso dobbiamo o modificare la grammatica che però sarà piena di non terminali e spesso questa procedura passa da trucchi e non c'è niente di teorico (che si possono copiare o inventare).

Chapter 3

Parsing

Quello che vogliamo fare ora è passare partendo da un simbolo distinto espandendo una serie di regole per cui alla fine del processo avrò la parola. L'altro processo è l'opposto parto dalla parola e cerco di capire quali sono gli ultimi pezzi che hanno composto la parola e poi proceso all'indietro fino ad arrivare al simbolo distinto, in questo modo cerco di riaggregare i pezzi andando dal basso verso l'alto. Questi a grandi linee sono le due strategie che applicheremo per il parsing. I due modi si chiamano:

1. **Top-down:** Parto dal simbolo distinto e cerco di arrivare alla parola
2. **Bottom-up:** Parto dalla parola e cerco di arrivare al simbolo distinto

Notiamo che questi approcci possono essere applicati a tutte le grammatiche, ad esempio vediamo una grammatica context sensitive:

Quello che a volte conviene fare è ribaltare la grammatica, le produzioni left producono le right, questo ovviamente va a rovinare la nostra grammatica, poi possiamo mettere anche un simbolo Inizio e Fine per indicare l'inizio e la fine della produzione. Spesso lavorando in questo modo si produce una derivazione right-most (sostituisco il simbolo più a destra della derivazione non sentenziale e sostanzialmente faccio i passi da destra a sinistra dall'ultimo al primo). Posso decidere dall'alto in basso e questo produce derivazioni left-most oppure posso andare dal basso verso l'alto cercando di aggregare e ottengo derivazioni right-most ma che devo leggere dal basso verso l'alto.

3.1 NPDA

In realtà questo tipo di comportamento viene modellato nell'ambito del parsing attraverso la nozione di automa che ha bisogno di un nastro di input ed un area di memoria per tenere il processo di parsing, sostanzialmente deve fare la scansione della parola e mano a mano costruire l'albero di derivazione. Tutti gli algoritmi che vedremo hanno bisogno di una pila (stack) come forma di memoria per tenere traccia del processo di parsing perché il tipo di lavoro che fanno non richiede un accesso causale alla memoria. Ma come fanno a decidere questi automi? hanno un dispositivo di controllo che è in grado di determinare cosa fare, come spostarsi sul nastro e cosa scrivere e prendere dalla pila. La cosa più intuitiva che uno può fare è avere una mente omisciente che è in grado di determinare cosa l'automa in ogni situazione deve fare. Quello che dovremo fare nel nostro lavoro è costruire questo controllo e poi trovare un modo per eliminare il non determinismo perché non avremo l'oracolo che sa tutto.

Quello che accadrà è che data una descrizione della grammatica sarà possibile realizzare la struttura di controllo dell'automa in maniera automatica, potremmo pensare ad un programma che dato in pasto la grammatica G produca l'automa, questi programmi si chiamano parser generator. Un'altro modo molto usuale, che descrive una grande famiglia di algoritmi di parsing, è quello di avere un controllo universale (che funziona per ogni grammatica) trasformando una tabella che rappresenta le informazioni che G contiene in una forma che fa in modo che il controllo sappia cosa fare, questo metodo si chiama table driven.

Nell'analizzare il comportamento di questi parser ci sono due grandezze che vogliamo vedere:

1. Lo spazio in memoria, quanto spazio consuma l'automa nel funzionamento sempre in rapporto alla lunghezza della parola
2. Il tempo di esecuzione, quanto tempo impiega l'automa a processare la parola

Evidente più è alto il tipo di grammatica più è alta la memoria e più scendiamo il contrario. Noi ci occuperemo delle context-free e per quel che concerne gli altri due livelli della gerarchia osserviamo questo:

- Le tipo 0 (unrestricted) rappresentano gli insiemi ricorsivamente enumerabili, quindi chiedersi se una parola appartiene ad una grammatica di tipo 0 equivale a chiedersi se il programma termina, problema indecidibile
- Le tipo 1 (context sensitive) sono più deboli delle unrestricted, sono gli insiemi accettati da una macchina di Turing non deterministica, problema decidibile ma non in tempo polinomiale bensì in tempo e spazio esponenziale. Questo è il livello in cui si colloca il parsing naturale (linguaggi naturali).
- Le tipo 2 (context free) sono gli insiemi accettati da un NPDA, problema decidibile in tempo polinomiale.

Una cosa che vale la pena considerare nell'ambito del parsing context-free esiste una possibile divisione del lavoro che dobbiamo fare raggruppando sulla base di qualche criterio. Una prima grande dicotomia è se gli algoritmi funzionano in maniera:

1. top-down
2. bottom-up

Un'altra dicotomia piuttosto interessante è il modo in cui l'input viene analizzato, perchè l'automa può andare avanti ed indietro nell'input:

1. Parser direzionale: L'automa va avanti e indietro nell'input, in ordine, se lo ciuccia man mano che riceve i byte
2. Parse non direzionale: il parse può adoperare delle porzioni del nastro diverse, ad esempio nell'uso degli editor abbiamo la colorizzazione della sintassi, in questo caso il parser non è direzionale perchè se fosse direzionale dovrei ricostruire tutto l'albero per ogni modifica.

Un'altra possibile dicotomia che abbiamo già in qualche modo accennato è il fatto che purtroppo abbiamo il non determinismo e per risolvere questo potrebbe provare tutti i passi agendo:

1. Depth first
2. Breadth first

Una possibilità cruciale per eliminare il problema del non determinismo è avere un parsing che a priori non accetti qualsiasi grammatica ma solo quelle che sono in una certa forma, quelle deterministiche (si rispippola la grammatica in modo che sia deterministica). Questo ci crea dei parsing molto meno potenti.

Cosa vedremo noi:

1. Non directional methods Bottom-up: CYK parser
2. Deterministic directional:
 - (a) Top-down: LL parser, mescolare parsing e traduzione è più semplice dell LR. Sono meno efficienti ma pace, noi useremo un parser generator LL(*). Noi vedremo LL(1).
 - (b) Bottom-up: LR parser, vedremo LR(0)

3.2 CYK parser

L'algoritmo CYK si basa su una tabella che rappresenta la comprensione temporanea che ha l'algoritmo del processo di parsing (della costruzione dell'albero di parsing) e questa tabella può essere facilmente riempita nel caso in cui la grammatica abbia una forma semplificata che per il momento noi assumeremo. Le regole che vorremo avranno solo questa forma, una produzione non terminale o di una coppia:

1	A -> BC
2	A -> a

La tabella di questo algoritmo è fatta così, sulla base abbiamo la parola e poi ha una serie di celle in cui in ciascuna cella contiene l'informazione della sotto-parola la cui lunghezza è data dalla riga in cui mi trovo e che raccontano l'idea che si è fatto l'algoritmo di parsing della stringa lunga due a partire dalla posizione della tabella sotto. Nella posizione i,l rappresenta l'informazione che ho sul parsing del prefisso della parola che comincia lì ed è lunga l:

Osserviamo che la tabella può essere riempita in due modi, siccome ogni posizione riguarda un sottoinsieme la posso riempire in due modi:

1. offline: parto dal fondo a sinistra e comincio a mettere le lettere singole, poi sopra le coppie e così visita. Offline perchè per riempirla devo aver visto tutto l'input
2. online: la riempio in diagonale partendo dal basso a sinistra, mano mano che vedo caratteri posso riempire sempre più celle

Noi ovviamente non la riempiamo di parti di parola ma di produzioni, per filtrare le produzioni useremo la funzione di python filter che filtra cose in base ad una funzione che gli passiamo. Quando siamo in alto nella tabella dobbiamo mettere una produzione che da A produce BC, perchè vogliamo metterci qualcosa che produce quello che c'è sotto quindi sotto dobbiamo andare a vedere se B produce un pezzo di quello che c'è sotto e la C il resto.

In buona sostanza se sono a lunghezza 1 ci metto tutti i terminali dove A produce a dove a è proprio la parola (lettera negli esempi) che sto cercando, se sono a lunghezza 2 metto tutte le produzioni che producono due terminali B e C dove B produce la parola da i a k e c da k in poi.

Negli esempi si vede bene che sotto mettiamo tutti i terminali delle produzioni a sinistra che producono la parola o lettera (ad esempio A produce a quindi ci metto A per tutte le volte che ci sono a nella parola). Poi per le celle sopra ci chiediamo se c'è una produzione che produce $A \rightarrow BC$ dove B produce la sotto parola da i a k e C da k in poi (ad esempio $S \rightarrow A, S$).

Per guardare se la parola fa parte del linguaggio basta guardare la cella in alto della tabella, se è riempita significa che è ok.

3.2.1 Albero di parsing

Dobbiamo ricostruire l'albero di parsing da questa tabella, al momento ci accontentiamo di costruire l'albero i cui nodi sono i non terminali e gli archi le produzioni (non ha dentro le produzioni). Non è super difficile perchè possiamo approcciare questa cosa ricorsivamente

--