Linguaggi e Traduttori

Simone Petta

A.A. 2024/2025

Contents

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')
```

Chapter 2

Linguaggi

2.1 Closed form

L'idea di poter definire un sotto-linguaggio per ciauscun simbolo è molto comodo perchè 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 nel 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:

$$A \to aAa$$

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.

$$A \to (A)$$

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 perchè abbiamo delle grammatiche con difetti che vorremmo eliminare. La spazzatura che può rimanere dentro una gramamtica 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 perchè 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 chiadiamo è una /underlinechiusura 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. É chiaro che con una funzione di questo tipo la pulizia diventa abbastanza semplice. Vediamo un esempio di una grammatica sporca:

```
G = Grammar.from_string("""
S -> A B | D E
A -> a
B -> b C
C -> c
D -> d F
E -> e
F -> f D
""")
G
```

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

```
def find_productive(G):

  @closure
  def find(prod):
    return prod | {A for A, a in G.P if set(a) <= prod}

  return find(G.T)

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:

```
from liblet import union_of

def find_reachable(G):

  @closure
  def find(reach):
    return reach | union_of(set(a) for A, a in G.P if A in reach)

  return find({G.S})

find_reachable(G)
```

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

```
def remove_unproductive_unreachable(G):
    Gp = G.restrict_to(find_productive(G))
    return Gp.restrict_to(find_reachable(Gp))

remove_unproductive_unreachable(G)
```

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

2.2.2 Dimensione degli alberi di parsing

Ha senso ragionare sulla dimensione degli alberi di parsing? si 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 (agglomero 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.

```
#  una grammatica banale per il linguaggio {a and nb | n > 0}

G_ab = Grammar.from_string('''
S -> A B
A -> a A | a
B -> b
""")
G_ab

# due possibili derivazioni

ab_0 = Derivation(G_ab).step(
  [(0, 0), (1,0), (2,1), (3,2)]
)
```

```
ab_1 = Derivation(G_ab).step(
  [(0, 0), (3,1), (1,0), (2,1)]
)

ab_0, ab_1

# ma a ben guardare lo stesso albero

side_by_side(
  ProductionGraph(ab_0),
  ProductionGraph(ab_1),
)
```

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 aberi che mano a mano da questa forma sviluppo, la prima forma è il simbolo distinto e poi per ogni passo di derviazione mi dice quale pezzo della forma sentneziale 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).

```
def derivation_to_parsetree(d):

    # questa variabile si riferira all'albero di derivazione di d
    # inizialmente contiene l'albero annotato col simbolo di partenza
    tree = Tree({'Symbol': d.G.S, 'prod': None})

    # all'inizio la forma sentenziale e data da tale albero
    frontier = [tree, ]

    for nprod, pos in d.steps():

        # l'albero da completare e dato dalla posizione in cui e
           applicata la produzione
        curr = frontier[pos]

        # risalgo dal numero alla produzione
        prod = d.G.P[nprod]

        # i figli sono dati dal lato destro d.G.P[prod].rhs
```

```
        children = [Tree({'Symbol': X, 'prod': None}) for X in prod.rhs]

        # aggiorno l'albero da completare
        curr.root['prod'] = prod
        curr.children = children

        # aggiorno la forma sentenziale
        frontier = frontier[:pos] + children + frontier[pos + 1:]

    return tree
```

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

```
def leftmost(tree):
 return [tree.root['prod']] + [prod for child in tree.children for
    prod in leftmost(child) if prod]

lm_0 = leftmost(pt_0)
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:

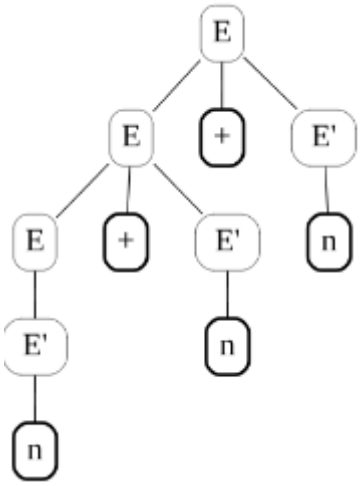
1. Non è sempre possibile
2. Non è automatico
3. Introduco N non "semantici": è chiaro che poi questi N andranno eliminati perchè producono alberi di parsing pieni di mondezza

2.2.5 Possibili soluzioni

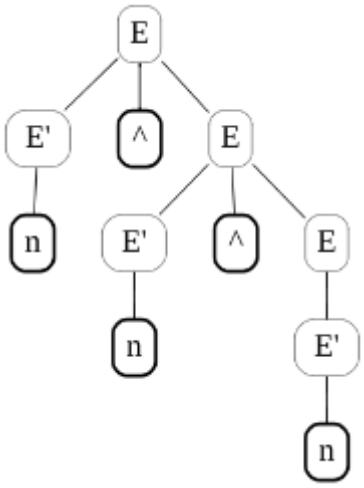
La ricorsione a sinistra determina un associatività a sinistra mentre se è a destra si usa una ricorsione a destra. Quello che facciamo e' inserire dei simboli in piu' nel linguaggio per gestire tutti i casi:

```
  G_la = Grammar.from_string("""
  E -> E + E' | E'
  E' -> n
  """)
  G_la
```

```
  G_ra = Grammar.from_string("""
  E -> E' ^ E | E'
  E' -> n
  """)
  G_ra
```



(a) Associatività Sinistra

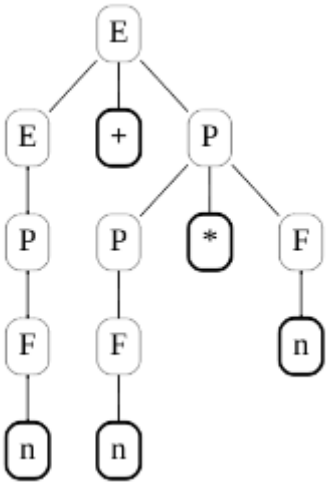


(b) Associatività Destra

Figure 2.1: Confronto tra associatività sinistra e destra

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

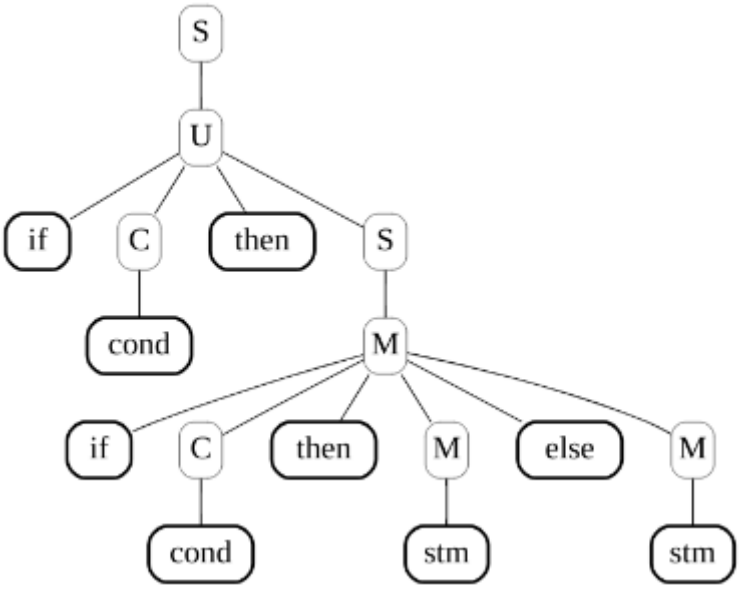
```
G_p = Grammar.from_string("""
E -> E + P | P
P -> P * F | F
F -> n
""")
```



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 perchè tiene l'else legato all'if più vicino.

```
G_if = Grammar.from_string("""
S -> M | U
M -> if C then M else M | stm
U -> if C then M else U | if C then S
```

```
  C -> cond
  """)
```



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 appliacati a tutte le grammatice, ad esempio vediamo una grammatica context sensitive, nel caso top down:

```
w = 'aabbcc'

G = Grammar.from_string("""
S -> a S Q | a b c
b Q c -> b b c c
c Q -> Q c
""", False)

steps = (0, 0), (1, 1), (3, 3), (2, 2)

for nprod, pos in steps: print(pos, G.P[nprod])
```

Nel caso bottom up 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 queso 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.

```
GR = Grammar.from_string("""
Inizio -> a a b b c c
a S Q -> S
a b c -> S
b b c c -> b Q c
Q c -> c Q
S -> Fine
""", False)

steps = (0, 0), (3, 2), (4, 3), (2, 1), (1, 0), (5, 0)

for nprod, pos in steps: print(pos, GR.P[nprod])
```

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 scansinoe 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 perchè 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 omiscente 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 elminare il non determinismo perchè 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 equeivale 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 parser 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 cruciali per eleminare 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 vedermo 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 più 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:

```python
INPUT = 'unaprova'

n = len(INPUT)

R = CYKTable()
for l in range(1, n + 1):
  for i in range(1, n - l + 2):
    R[i, l] = INPUT[(i) - 1: (i + l) - 1]
```

unaprova							
unaprov	naprova						
unapro	naprov	aprova					
unapr	napro	aprov	prova				
unap	napr	apro	prov	rova			
una	nap	apr	pro	rov	ova		
un	na	ap	pr	ro	ov	va	
u	n	a	p	r	o	v	a

Osserviamo che la tabelle 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

```python
 def offline(fill, n):
  R = CYKTable()
  for l in range(1, n + 1):
    for i in range(1, n - l + 2):
      R[i, l] = fill(R, i, l)
  return R

def online(fill, n):
  R = CYKTable()
  for d in range(1, n + 1):
    for i in range(d, 0, -1):
      R[i, d - i + 1] = fill(R, i, d - i + 1)
  return R
```

Noi ovviamente non la riempiremo 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.

```
def cyk_fill(G, INPUT):
 def fill(R, i, l):
   res = set()
   if l == 1:
     for A, (a,) in filter(Production.such_that(rhs_len = 1), G.P):
       if a == INPUT[i - 1]: res.add(A)
   else:
     for k in range(1, l):
       for A, (B, C) in filter(Production.such_that(rhs_len = 2), G.P):
         if B in R[i, k] and C in R[i + k, l - k]: res.add(A)
   return res
 return fill
```

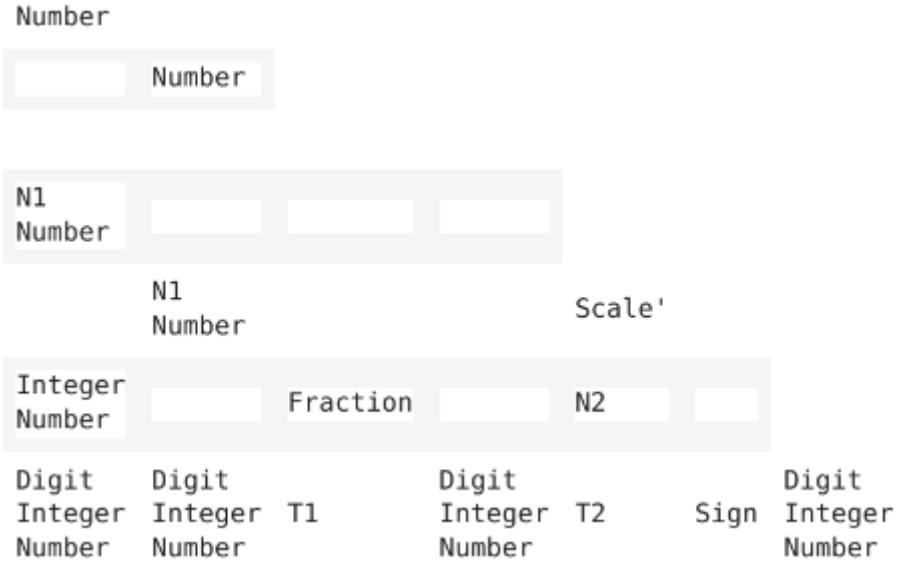
Vediamo degli esempi su una grammatica a^n:

```
G = Grammar.from_string("""
S -> A S
A -> a
S -> .
""")

INPUT = 'aaa.'

online(cyk_fill(G, INPUT), len(INPUT))
```


```
 INPUT = 'aa.a.'

online(cyk_fill(G, INPUT), len(INPUT))
```

```
# fig. 4.15, pag. 123

G = Grammar.from_string("""
Number -> 0|1|2|3|4|5|6|7|8|9
Number -> Integer Digit
Number -> N1 Scale' | Integer Fraction
N1 -> Integer Fraction
Integer -> 0|1|2|3|4|5|6|7|8|9
Integer -> Integer Digit
Fraction -> T1 Integer
```


```
T1 -> .
Scale' -> N2 Integer
N2 -> T2 Sign
T2 -> e
Digit -> 0|1|2|3|4|5|6|7|8|9
Sign -> + | -
""")
```



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 $A \to BC$ dove B produce la sotto parola da i a k e C da k in poi (ad esempio $S \to A, S$).

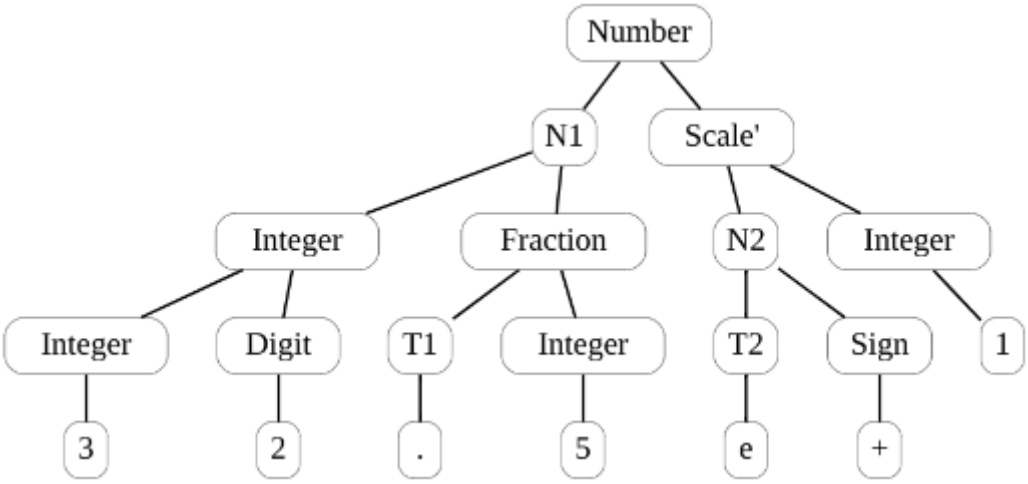
Per guardare se la parola fa parte del linguaggio basta guardare la cella in alto della tabella, se è riempita significa che è ok.

3.2.1 Albero di parsing

Dobbiamo ricostruire l'albero di parsing da questa tabella, al momento ci accontentiamo di costurire 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 ricorsviamente. Scriviamo una funzione ricorsiva fake_parse che (usando la tabella R, la

grammatica G e l'input INPUT) dato un non terminale, il punto d'inizio e la lunghezza, restituisca l'albero di parsing radicato in quel non terminale e che deriva la sottostringa specificata.

```
def cyk_fill(G, INPUT):
  def fill(R, i, l):
    res = set()
    if l == 1:
      for A, (a,) in filter(Production.such_that(rhs_len = 1), G.P):
        if a == INPUT[i - 1]: res.add(A)
    else:
      for k in range(1, l):
        for A, (B, C) in filter(Production.such_that(rhs_len = 2), G.P):
          if B in R[i, k] and C in R[i + k, l - k]: res.add(A)
    return res
  return fill
```



3.2.2 Riduzione in forma normale di Chomsky

La forma normale di Chomsky è una forma normale in cui tutte le produzioni sono della forma:

```
    A -> BC
    A -> a
```

Per ottenere una grammatica in forma normale di Chomsky dobbiamo fare due passaggi:

1. Eliminare le regole unitarie
2. Eliminare le produzioni più lunghe di 2, riduzione in forma normale, trasformazioni produzione non solitarie
3. Eliminare le epsilon regole

```
def remove_unproductive_unreachable(G):
  def find_productive(G):
    @closure
    def find(prod):
      return prod | {A for A, a in G.P if set(a) <= prod}
    return find(G.T)
  def find_reachable(G):
    @closure
    def find(reach):
      return reach | union_of(set(a) for A, a in G.P if A in reach)
    return find({G.S})
  Gp = G.restrict_to(find_productive(G))
  return Gp.restrict_to(find_reachable(Gp))

def cyk(G, INPUT):
  def fill(R, i, l):
    res = set()
    if l == 1:
      for A, (a,) in filter(Production.such_that(rhs_len = 1), G.P):
        if a == INPUT[i - 1]: res.add(A)
    else:
      for k in range(1, l):
        for A, (B, C) in filter(Production.such_that(rhs_len = 2), G.P):
          if B in R[i, k] and C in R[i + k, l - k]: res.add(A)
    return res
  R = CYKTable()
  for l in range(1, len(INPUT) + 1):
    for i in range(1, len(INPUT) - l + 2):
      R[i, l] = fill(R, i, l)
  return R
```

3.2.3 Elminazione epsilon-regole

Una epsilon regola è una regola della forma A -> ϵ, cioè una regola che produce la parola vuota. Per eleminare una epsilon regola poteri duplicare la regola, cioè se ho una regola $A -> \epsilon$ e una produzione $B -> \alpha$ A B posso da qui crearne due con A_1 e A_2. Non sempre è così comodo. La prima osservazione che facciamo è che abbiamo trasformato G in g primo in cui non abbiamo le epsilon regole ma il costa è che per ogni epsilon regola che eliminiamo produce 2^n produzioni (dove n è la lunghezza della regola). Quindi sappiamo che $|G^{'}| \geq 2|G|$. Sostanzialmente con due passi ottnuti tramite chiusura posso rimpiazzarte un simbolo nei lati destri con replace_in_rhs e quindi applicare il primo passo a tutti i simboli che compaiono in una epsilon regolan con inline_epsilon_rules.

La prima cosa che dobbiamo fare è il rimpiazzamento di A con A_1 in tutte le produzioni, poi prendo tutte le produzioni che contengono A e cancello o sostituisco con A primo le occorrenze di A, se invece on l'ho beccato in B lascio così com'è:

```
@closure
def replace_in_rhs(G, A):
  Ap = A + '''
  prods = set()
  for B, Beta in G.P:
    if A in Beta:
```

```
      pos = Beta.index(A)
      prods.add(Production(B, Beta[:pos] + Beta[pos + 1:]))
      prods.add(Production(B, Beta[:pos] + (Ap, ) + Beta[pos + 1:]))
    else:
      prods.add(Production(B, Beta))
  return Grammar(G.N | {Ap}, G.T, prods, G.S)

# esempio d'uso

U = Grammar.from_string("""
S -> x A y A z
A -> a
""")
replace_in_rhs(U, 'A').P
```

$$A \quad a_{(0)}$$

$$S \quad x\ y\ z_{(1)} \mid x\ y\ A'\ z_{(2)} \mid x\ A'\ y\ z_{(3)} \mid x\ A'\ y\ A'\ z_{(4)}$$

A questo punto per sistemarle dobbiamo fare una chiusura, perchè abbiamo prodotto delle epsilon regole, prende tutti i non terminali che non abbimo ancora visto e se quel terminale tra i lati destri di A c'è una epsilon regola faccio il rimpiazzamento e segno che l'ho fatto.

```
@closure
def inline_epsilon_rules(G_seen):
  G, seen = G_seen
  for A in G.N - seen:
    if (epsilon, ) in G.alternatives(A):
      return replace_in_rhs(G, A), seen | {A}
  return G, seen

# esempio d'uso

U = Grammar.from_string("""
S -> A
A -> B C
B -> epsilon
C -> epsilon
""")
U, _ = inline_epsilon_rules((U, set()))

U.P
```

$$A \quad \varepsilon_{(0)} \mid B'_{(1)} \mid C'_{(5)} \mid B'\ C'_{(6)}$$

$$C \quad \varepsilon_{(2)}$$

$$S \quad \varepsilon_{(3)} \mid A'_{(7)}$$

$$B \quad \varepsilon_{(4)}$$

Osserviamo che questo procedimento non ha tolto le epsilon regole ma le ha messe inline, le mette dove accadevano, quello che succede è che queste regole prima o poi diventano unreachable quindi prima o poi questa regola la toglieremo. L'altro effetto è che solleva la epsilon fino al simbolo distinto, quindi quello che accade è che questo linguaggio potrebbe produrre la parola vuota. Usando i passi precedenti è semplice arrivare al passo di eliminazione:

```
def eliminate_epsilon_rules(G):
  Gp, _ = inline_epsilon_rules((G, set()))
  prods = set(Gp.P)
  for Ap in Gp.N - G.N:
    A = Ap[:-1]
    for alpha in set(Gp.alternatives(A)) - {(epsilon, )}:
      prods.add(Production(Ap, alpha))
  return Grammar(Gp.N, Gp.T, prods, Gp.S)

# esempio d'uso (fig. 4.10, pag. 120)

U = Grammar.from_string("""
S -> L a M
L -> L M
L -> epsilon
M -> M M
M -> epsilon
""")

eliminate_epsilon_rules(U).P
```

```
M  ε(0) | M'(6) | M' M'(14)
L  L' M'(1) | M'(10) | ε(11) | L'(12)
M' M'(2) | M' M'(4)
L' L'(3) | L' M'(8) | M'(15)
S  a M'(5) | a(7) | L' a(9) | L' a M'(13)
```

3.2.4 Eliminazione delle unit rules

Le regole unitarie sono quelle della forma A -> B, cioè una variabile che deriva un'altra variabile. In questo caso mi aspetto anche che ci sia una C che deriva B e A che deriva qualcos'altro che non è B. Quello che si fa è in tutte le regole dove c'è la A ci metto tutte le alternative. Dato B -> $\omega_1|\omega_2$ e $C->\alpha AB$ posso fare $C->\alpha\omega_1|\alpha\omega_2$. Faccio questa sostituzione a meno che non mi trovi nel caso del loop $B->B$. Prendo tutte le produzioni che sono lunghe 1, le spacco, dico che A produce B, se B è un non terminale prendo le produzioni e faccio l'inline, faccio in modo che A produca tutte le produzioni di B e poi elimino A produce B:

```
def eliminate_unit_rules(G):
```

```
  @closure
  def eliminate(G_seen):
    G, seen = G_seen
    for P in set(filter(Production.such_that(rhs_len = 1), G.P)) - seen:
      A, (B, ) = P
      if B in G.N:
        prods = (set(G.P) | {Production(A, alpha) for alpha in
           G.alternatives(B)}) - {P}
        return Grammar(G.N, G.T, prods, G.S), seen | {P}
    return G, seen
  return eliminate((G, set()))[0]

# esempio d'uso

U = Grammar.from_string("""
S -> A
A -> B
B -> A | b
""")

eliminate_unit_rules(U).P
```

Notiamo che questo porta ad avere una grammatica con delle cose irraggiungibili (dopo che elminiamo le epsilon regole e i non unitari). Se voglio fare pulizia uso la tecnica dell'altra volta, applico il metodo remove_unproductive_unreachable, ma ancora non è quello da cui siamo partiti perchè ad esempio ci sono dei non terminali oppure delle cose lunghe 3. Ci rimangono due passaggi:

1. Eliminare i non solitari: $A- > \alpha a \beta$. dove alpha e beta non sono terminali
2. Eliminare le produzioni più lunghe di 2

3.2.5 Eliminazione i non solitari

Un solitario e' un simbolo non terminale se esiste almento una derivazione in cui appare, contribuendo alla produzione di stringhe terminali. Per ogni solitario che trovo in giro produco una regola unitaria lunga 1 e sostituisco secco, non ho bisogno neanche di fare le chiusure, cerco le produzioni A B e cerco nel lato destro e guardo cosa sono, se sono dei non terminali li lascio come sono, se sono dei terminali li sostituisco con N e la letterina, dopo di che sostitusico la produzione $Na- > a$ ad esempio:

```
def transform_nonsolitary(G):
  prods = set()
  for A, alpha in G.P:
    prods.add(Production(A, [f'N{x}' if x in G.T else x for x in alpha]
       if len(alpha) > 1 else alpha))
    prods |= {Production(f'N{x}', (x, )) for x in alpha if x in G.T and
       len(alpha) > 1}
  return Grammar(G.N | {A for A, alpha in prods}, G.T, prods, G.S)

U = Grammar.from_string("""
S -> x S y S x
""")

```

```
transform_nonsolitary(U).P

Ny Y(0)
Nx X(1)
S Nx S Ny S Nx
```

Non e' sempre cosi' facile, per le produzioni lunghe faccio produrre ad un primo pezzo A1, x1, x2 poi ad un secondo pezzo A2 faccio produrre A1 x2, così via fino in findo dove avrò A produce A3 x7, partendo da una produzione $A- > x1x2x3x4x5x6x7$:

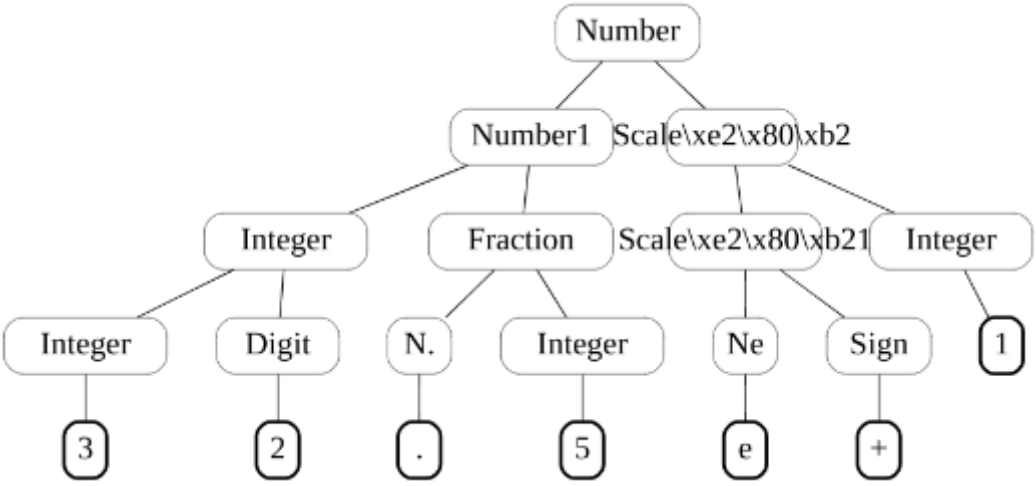
```
def make_binary(G):
  prods = set()
  for A, alpha in G.P:
    if len(alpha) > 2:
      Ai = f'{A}{1}'
      prods.add(Production(Ai, alpha[:2]))
      for i, Xi in enumerate(alpha[2:-1], 2):
          prods.add(Production(f'{A}{i}', (Ai, Xi)))
          Ai = f'{A}{i}'
      prods.add(Production(A, (Ai, alpha[-1])))
    else:
      prods.add(Production(A, alpha))
  return Grammar(G.N | {A for A, alpha in prods}, G.T, prods, G.S)
```

Ora dobbiamo ricostruire l'albero di parsing, ci verrà più facile rispetto a fare una trasformazione di alberi sostituendo la tabella CYK qualcos'altro.

3.2.6 Costruzione albero parsing left-most

Dato un certo non terminale ci diamo come obiettivo costruire un fattore di una parola di input che partiva da una certa posizione con una certa lunghezza. Se la lunghezza è 1 devo andare a cercare una produzione della forma X i -1, questa cosa funziona perchè la tabella è costruita correttamente. Se andiamo a riprenderlo è uguale a fake parse, non metto gli alberi ma metto le produzioni. Quindi per ottenere una derivazione leftmost ragioniamo come per la funzione fake_parse ma invece di restituire un albero restituiamo l'indice della produzione in gioco:

```
 def get_leftmost_prods(G, R, INPUT):
  def prods(X, i, l):
    if l == 1:
      return [G.P.index(Production(X, (INPUT[i - 1],)))]
    for A, (B, C) in filter(Production.such_that(lhs = X, rhs_len = 2),
        G.P):
      for k in range(1, l):
        if B in R[i, k] and C in R[i + k, l - k]:
          return [G.P.index(Production(A, (B, C)))] + prods(B, i, k) +
              prods(C, i + k, l - k)
  return prods(G.S, 1, len(INPUT))

leftmost_prods = get_leftmost_prods(G_cnf, R, INPUT)
leftmost_prods
[28, 31, 9, 12, 7, 37, 20, 11, 1, 6, 32, 15, 30]
d = Derivation(G_cnf).leftmost(leftmost_prods)
ProductionGraph(d)
```



Bisogna considerare anche cosa succede nelle produzioni non binarie, mi piacerebbe avere una funzione derives che prende una sequenza di simboli, una i e una l e restituisce una serie di lunghezze ln tale per cui l0 è la lunghezza derivata da w0 input i + i + l0. Per farlo dovrà fare una ricorsione sulla tabella. Mi sono perso mannaggia a me, ma era abbastanza inseguibile, prova a vedere dal libro.

3.3 Parsing Top-down, caso generale

Andiamo dal basso verso l'alto, da sinistra a destra senza occuparci particolarmente delle grammatiche. Cominciamo ragionando sull'automa, abbiamo un nastro che legge da sinistra verso destra e teniamo una pila in cui manteniamo lo stato del parsing e poi il controllo che facendo uso del nastro che legge in modo direzionale e della pila deve riuscire a risolvere il problema del riconoscimento, anzi meglio dovrebbe sputare l'albero di parsing della parola.

Facciamo un'assunzione sulla forma della grammatica che ci viene comoda per definire il controllo, una volta definito il controllo riusciremo a rilassare questa assunzione.

Proviamo a ragionare su questa grammatica:

```
S -> aBC
A -> aB | b
C -> a

w = aaba
```

Il riconoscimento è molto semplice se ci basiamo sulle produzioni, partendo da S vediamo che produce a, ma cosa ci facciamo della B e della C? adesso il punto cruciale dato che voglio una left-most mi occupo di B, che a questo punto posso solo sostituire con la sua produzione ottenendo aB, a questo punto sempre dato che voglio una left-most devo occuparmi della nuova B generata e non ho scelta se non sostituirla con aB, vado avanti così fino alla fine.

L'idea è che inizio con una pila sulla quale ho il simbolo di partenza S, a questo punto applico la regola 0 e cancello la S e metto sulla pila in ordine inverso le produzioni che ho appena applicato (ci metto solo i non terminali). A questo punto la pila contiene CB,

PILA	HEAD	STACK
S	a	BC
B	a	B
B	b	B
C	a	B

Table 3.1: Tabella di controllo del parser top-down

tolgo la B dalla cima della pila, con la sua produzione mi mangio la seconda a e metto sulla pila la B che resta dalla produzione di B, vado avanti così fino a che non ho più nulla da mangiare (mangiare sono i terminali minuscoli che vogliamo trovare che compongono la parola) e la pila è vuota.

Il controllo che abbiamo possiamo considerarlo come una tabella che ha il top della pila, la head della parola e lo stack che contiene i non terminali che sono stati messi in pila:

Mi voglio tenere anche la storia della derivazioni. Nella funzione della libreria mettiamo un # alla fine della pila e della parola per sapere quando fermarci, ci feriamo quando arriviamo al cancelletto della pila e al cancelletto della parola. Questa descrizione istantanea può evolvere attraverso un passo di predizione, ho decisio che produzione applicare e quindi il passo di predizione ci porta ad avere tolto il lato sininistro della produzione dalla pila e messo il lato destro della produzione sulla pila. Posso semplificare applicando due nuove regole:

1. $(x, x) \rightarrow$ // passo di predizione
2. $(A,) \rightarrow \chi \in N$ passo di Match

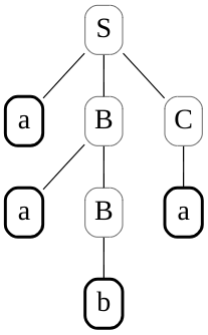
Quindi a questo punto il controllo o mette una cosa sulla pila oppure toglie dalla pila se sulla cima c'è un non terminale che sta anche sul nastro. Sostanzialmente la testina legge le lettere della parola, il controllo se trova sulla cima della pila un terminale (lettera minuscola) lo consuma e lo toglie dalla pila spostando la testina del nasto a destra, se trova un non terminale lo sostituisce con la produzione e lo mette sulla pila. Se la cima della pila è il cancelletto e la testina è sul cancelletto allora ho finito.

In questo esempio vediamo una simulazione su visite del DAG delle computazioni, in ogni nodo i conserviamo la descrizione istantanea piu' la derivazione che ha condotto a tale derivazione (la grammatica e' quella di prima). Notiamo che all'inizio la pila viene mostrata da sinistra a destra, laa cima della pila e' il carattere piu' a sinistra e che sulla pila all'inizio e' stato aggiunto il simbolo terminale e poi la S e in coda al nastro e' stato aggiunto il simbolo terminale mentre tra parentesi abbiamo il passo di derivazione che ha portato a quella situazione.:

```
i = TopDownInstantaneousDescription(G, 'aaba')

output: (), S#, |aaba#

i = i.predict(G.P[0])
output: (s -> a B C,), aBC#, |aaba#

i = i.match()
```

```
output: (S -> a B C,), BC#, a | aba#

i = i.predict(G.P[2])
output: (S -> a B C, B -> a B, B -> b), bC#, aa|ba#

.
.
.

i = i.match()
output: (S -> a B C, B -> a B, B -> b, C -> a), #, aaba|#
```

Notiamo che i passi raccolti (nello stesso ordine di quello con cui sono usati da predict) sono quelli di una derivazione leftmost:



3.3.1 Funzione stato prossimo

Per ora abbiamo visto una grammatica in cui la parte di sinistra non aveva ripetizioni, normalmente potremmo avere una grammatica in cui:

```
    S -> aB | aC
    (S, a) -> aB
    (S, a) -> aC
```

E questo produce una situazione di non determinismo. Per risolvere questo problema esistono tecniche teoriche e pratiche. La tecnica teorica è quella di avere un dio che dice ad ogni passo cosa fare, a noi non basta perchè non abbiamo dio direttamente da interrogare. Possiamo immaginarci un grafo della composizione istantanea in cui i nodi sono le istantanee sulla pila e gli archi sono i passi di predizione e match. Nel caso di non determinismo abbiamo più archi che partono dallo stesso nodo (perchè sono tutte le possibili produzioni che posso applicare). Questa la chiamiamo funzione a stato prossimo, quindi partnendo dalla pila e dal nastro (con la parola) possiamo costruire questo grafo che poi possiamo visitare. In questo grafo troviamo dei nodi morti (dead end) che non portano a nulla e dei nodi che portano a qualcosa. Devo trovare una tecnica di visita che mi porti a trovare i nodi che portano a qualcosa (verdi), la prima idea è una visita in ampiezza in cui posso anche fermarmi quando trovo il primo nodo verde. Il secondo aspetto è che potrei avere delle situazioni in cui arrivo allo stesso nodo ma in modi diversi, sempre il problema dell'ambiguità. Ad esempio in questa grammatica:

```
    S -> A | B
```

```
    A -> a | Ae
```

Quello che faccio è considerare uguali due istantanee se la pila e la testina sono uguali (non mi interessa come arrivo a quella situazione), $i = i' PILA; TESTINA$.

Cominciamo ad immaginare come produrre questo grafo, concentriamoci sulla visita in ampiezza. La cosa più facile è costruire una funzione stato prossimo, voglio costruire una funzione che data una situazione istantanea mi restituisca tutti gli archi. Mi tengo insieme a questi archi un'etichetta che mi dice se questo arco deriva da un passo di produzione o di match.

```
# restituisce un elenco di coppie (s, tid)
# dove s e 'match', o la str(P) dove P e una produzione di G e bid e
   una TopDownInstantaneousDescription
# ottenuta applicando il match o la predizione secondo P a curr

def next_instdescrs(instdescr):
  if instdescr.is_done(): return []
  G = instdescr.G
  top = instdescr.top()
  if top in G.T:
    return [('match', instdescr.match())] if top == instdescr.head()
       else []
  else:
    return [(str(P), instdescr.predict(P)) for P in
       filter(Production.such_that(lhs = top), G.P)]
```

Come detto la prima cosa che vogliamo fare è una derivazione in ampiezza, metto in una coda i nodi che ho visitato sotto l'assunzione che ho visto il nodo che ha la stessa configurazione di pila e nastro. Il problema se abbiamo ricorsioni è che potremmo andare all'infinito perchè il grafo diventerebbe infinito, quindi ci fermiamo al primo stato istantaneo buono (il verde) questo e' quello che fa il parametro first_only.

```
def breadth_first(G, word, verbose = False, first_only = True):
  graph = prepare_graph()
  q = Queue()
  q.enqueue(TopDownInstantaneousDescription(G, word))
  seen = {q.Q[0]}
  derivations = []
  while q:
    if first_only and derivations: break
    if verbose:
      for i in q: print(i)
      print('-' * 60)
    curr = q.dequeue()
    for what, nxt in next_instdescrs(curr):
      graph.edge(curr, nxt, gv_args={'label': what})
      if nxt.is_done(): derivations.append(nxt.steps)
      if not nxt in seen:
        seen.add(nxt)
        q.enqueue(nxt)
  return derivations, graph
```

Ora usiamo una visita in profondità, modifichiamo il tappo di visita, non mi fermiamo alla prima ma contiamo i passi che faccio, se li supero mi fermo.

```
def depth_first(G, word, verbose = False, max_steps = -1, use_seen =
    True):
  graph = prepare_graph()
  s = Stack()
  s.push(TopDownInstantaneousDescription(G, word))
  seen = {s.S[0]}
  derivations = []
  steps = 0
  while s:
    if steps > max_steps > -1: break
    steps += 1
    if verbose:
      for i in s: print(i)
      print('-' * 60)
    curr = s.pop()
    for what, nxt in next_instdescrs(curr):
      graph.edge(curr, nxt, gv_args={'label': what})
      if nxt.is_done(): derivations.append(nxt.steps)
      if not nxt in seen:
        if use_seen: seen.add(nxt)
        s.push(nxt)
  return derivations, graph
```

Con questo nuovo algoritmo notiamo che la coda non contiene più tanti elementi, ma 2/3 alla volta. Vediamo che il grafo scende molto più di prima.

Il problema è proprio la ricorsione, cosa succede se ho regole in cui ho S come ricorsione a sinistra? del tipo:

```
    S -> S a | b
```

Questo produce una situazione in cui, anche se ho una parola breve, facendo una visita in ampiezza la parola la trovo, ma se faccio una visita in profondità le derivazioni verdi non le trova, perchè non ha un loop nella pila (trova i rossi quando ritrova stati uguali) la pila aumenta all'infinito e non trova mai la parola.

Un modo per correggere il tiro c'è, in particolare la presenza di epsilon regole è drammatica (la pila aumenta senza consumare il nastro), però se evitiamo le epsilon regole possiamo, data una grammatica monotonica (in cui la forma sentenziale non si può ridurre, per noi è la pila) possiamo fermarci quando la pila è troppo lunga, ad esempio se devo generare pippo e sulla pila ho più di 5 non terminali non posso generare pippo, ma genererò una parola di più di 5 caratteri. Quindi nella visita se mi infilo in una circostanza in cui so che non incontrerò mai un nodo verde allora posso fermarmi. Allora posso considerare una produzione produttiva se la sua produzione è più corta di quello che resta da mangiare.

```
def next_instdescrs(curr):
  def productive(pair):
    _, instdescr = pair
    return len(instdescr.stack) <= (len(instdescr.tape) -
        instdescr.head_pos)
  return list(filter(productive, original_next_instdescrs(curr)))
```

Considero produttive solo le situazioni in cui sulla pila ho un numero di terminali sensato per la parola che ho da generare. Quindi così anche con una visita in profondità

riesco a trovare la parola. Notiamo che con la perdita delle epsioln regole perdiamo la possibilità di avere una produzione infinita.

Una cosa carina è che potremmo cambiare la scelta di dove andare solamente basandoci sul nostro passato (se sono arrivato allo stesso punto), posso vedere di togliere questa condizione, ovvero non mi pongo il problema di arrivare in un posto nello stesso modo (non mi interessa il passato). Ovvero se introduciamo l'ambiguità quello che troveremo e' alla fine sempre tutte le derivazioni con la differenza che potremmo avere archi doppi allo stesso nodo, questo perche' potrei arrivare a quel nodo ma con storie diverse quindi togliendo la condizione di uguaglianza tra istantanee potrei avere piu' archi che arrivano allo stesso nodo.

3.4 Parsing Ricorsivo Discendente

3.4.1 Eliminare Ricorsione a Sinistra

Eliminare la ricorsione è una cosa molto comoda anche per poter usare alcuni parser generator che funzionano solo se non c'è la ricorsione quindi è un trucco su cui vale la pena fare un discorso.

Il caso più elementare, se abbiamo un terminale che ha una ricorsione a sinistra questa cosa vorrebbe generare un ciclo, una lista finita di elementi:

$$A \to A\alpha_1|A\alpha_2...|\beta_1|\beta_2...$$

Quindi quello che faremo è introdurre dei nuovi terminali, la testa si occupa di generare i beta, poi similmente si definisce un A di coda che permette di ottenere gli alpha:

$$A_H \to \beta_1|\beta_2...B_T \to \alpha_1|\alpha_2...$$

Poi in realtà per generare la lista devo riportare la ricorsione, la spostiamo a destra inserendo un nuovo terminale:

$$A_{TS} \to A_T A_{TS}$$

Dopo di che posso generare uno dei simboli A:

$$A \to A_H A_{TS}|A_H$$

Abbiamo risolto la ricorsione diretta ma manca quella indiretta, una cosa del genere:

$$A \to B\alpha B \to C\gamma C \to A\beta$$

L'idea in questo caso è ancora strutturalmente semplice. Un modo semplice per risolvere il problema sarebbe quello di ordinare i non terminali, se potessi chiamarli:

$$A_1 < A_2 < ... < A_n$$

Il problema di questo caso è che parto da un non terminale e torno allo stesso. Quello che voglio riscrivere tutte le produzioni per cui quelle in cui non c'è un non terminale come primo simbolo deve essere minore di quello che c'è a destra (voglio evitare catene):

$$A_i \to A_j \alpha i < j$$

Il procedimento è iterativo e si parte con $i = 1$, in questo caso non voglio avere cose della ricorsione diretta $A_1 \to A_1\alpha_1|\alpha_2...$ quindi lo abbiamo fatto prima possiamo quindi creare una grammatica senza questa ricorsione. Ora passo a $i = 2$ e qui avrò cose del genere:

$$A_2 \to A_1\alpha A_2\alpha A_3\alpha...$$

Prendo questa grammatica e sostituisco gli A_1 con quelli che abbiamo sistemato prima. Andando avanti così riusciamo ad avere una grammatica G senza ricorsione a sinistra.

3.4.2 Prefix-Free

Adesso c'è un ulteriore richiesta che è più sottile, vogliamo parlare dei prefix-free, non è possibile avere un prefisso proprio dentro un linguaggio. Possiamo dire che una grammatica è prefix free se e solo se tutti i non terminali della grammatica sono prefix free, ossia A non deriva mai sia una parola che un suo prefisso. Non è detto che una grammatica si possa trasformare in prefix free, quindi noi lavoreremo sull'ipotesi che la grammatica sia prefix free.

3.4.3 Parsing Ricorsivo Discendente

L'idea è che vorremmo scrivere una funzione parse che prende in input un simbolo x e un pezzo di forma sentenziale w e restituisce una coppia booleano w' tale per cui se per come è fatta la grammtica x effettua il parsing di un prefisso di w allora restituisce true e il pezzo non parsato di w (nel caso in cui x sia un terminale significa che x deve essere la prima lettera di w).

Vogliamo provare a scrivere il parsing per le espressioni più e meno:

```
    E -> E + T | E - T | T
    T -> t
```

Partendo da una grammatica ricorsiva così si può trasformare in una grammatica non ricorsiva:

```
"""
E -> Eh Ets | Eh
Eh -> T
Ets -> Et Ets | Et
Et -> + T | - T
T -> t
"""
```

Questa grammatica non è prefix free ma noi vogliamo renderla prefix free, in questo caso possiamo mettere un tappo, sostanzialmente vogliamo vedere se dopo Eh mi devo fermare o no, mi fermo se trovo un cancelletto:

```
G = Grammar.from_string("""
E -> Eh Ets | Eh #
Eh -> T
Ets -> Et Ets | Et #
Et -> + T | - T
T -> t
""")
```

Possiamo ragionare per casi, trattando a parte il caso dei non terminali e poi ogni terminale a parte (una sorta di switch). I non terminali devo considerarli sulla scorta delle sua alternative(quel non terminale può produrre questo o quello), nel caso di T l'unica cosa che posso fare per dirti se funziona è vedere se t funziona (richiamando il parse)

```
@show_calls(True)
def parse(X, rest):

  if X in G.T:

    if rest and rest[0] == X: return True, rest[1:]
  elif X == 'T': # T -> t

    return parse('t', rest)

  elif X == 'Eh': # Eh -> T

    return parse('T', rest)

  elif X == 'Et': # Et -> + T | - T

    s, r = parse('+', rest)
    if s: return parse('T', r)
    s, r = parse('-', rest)
    if s: return parse('T', r)

  elif X == 'Ets': # Ets -> Et Ets | Et #

    s0, r0 = parse('Et', rest)
    if s0:
      s1, r1 = parse('#', r0)
      if s1: return True, r1
      return parse('Ets', r0)

  elif X == 'E': # E -> Eh Ets | Eh #

    s0, r0 = parse('Eh', rest)
    if s0:
      s1, r1 = parse('#', r0)
      if s1: return True, r1
      return parse('Ets', r0)

  return False, rest
```

Bello però noi vogliamo l'albero di parsing, modifichiamo la funzione affinchè ritorni anche i passi left-most con cui x produce il prefisso (lo produce usando regola n):

```
# ora parse restituisce: succ, rest, steps

@show_calls(True)
def parse(X, rest):

  if X in G.T:

    if rest and rest[0] == X: return True, rest[1:], None
```

```
9
10  elif X == 'T':
11
12    s, r, _ = parse('t', rest)
13    if s: return True, r, [7]
14
15  elif X == 'Eh':
16
17    s, r, p = parse('T', rest)
18    if s: return True, r, [2] + p
19
20  elif X == 'Et':
21
22    s, r, _ = parse('+', rest)
23    if s:
24      s, r, p = parse('T', r)
25      if s: return True, r, [5] + p
26    s, r, _ = parse('-', rest)
27    if s:
28      s, r, p = parse('T', r)
29      if s: return True, r, [6] + p
30
31  elif X == 'Ets':
32
33    s0, r0, p0 = parse('Et', rest)
34    if s0:
35      s1, r1, _ = parse('#', r0)
36      if s1: return True, r1, [4] + p0
37      s2, r2, p2 = parse('Ets', r0)
38      if s2: return True, r2, [3] + p0 + p2
39
40  elif X == 'E':
41
42    s0, r0, p0 = parse('Eh', rest)
43    if s0:
44      s1, r1, _ = parse('#', r0)
45      if s1: return True, r1, [1] + p0
46      s2, r2, p2 = parse('Ets', r0)
47      if s2: return True, r2, [0] + p0 + p2
48
49  return False, rest, []
```

3.4.4 Generazione automatica del parser

Non è così impensabile partire da una grammatica G e fare un programma che produce la funzione parse, perchè lo abbiamo fatto applicando solamente un pattern (gli if della soluzione sono prodotti, uno per ciascun simbolo, considerando in sequenza le possibili alternative (i lati destri delle produzioni) e restituendo successo al primo tentativo valido), quindi quello che vlgliamo fare è trovare il modo di creare una funzione parse che funziona per una grammatica. Questa cosa si chiama parser generator.

Un esempio e' il simbolo Ets le cui due alternative sono Et Ets e Et # ... supponiamo di scaricarne la soluzione a due funzioni denominate Ets_alt0 e Ets_alt1:

```
def parse(X, rest):

  # ...

  if X == 'Ets':
    succ_alt, rest_alt = Ets_alt0(rest)
    if succ_alt: return True, rest_alt
    succ_alt, rest_alt = Ets_alt1(rest)
    if succ_alt: return True, rest_alt
    return False, rest

  # ...

def Ets_alt0(rest):
  succ, rest = parse('Et', rest)
  if not succ: return False, rest
  succ, rest = parse('Ets', rest)
  if not succ: return False, rest
  return True, rest

def Ets_alt1(rest):
    succ, rest = parse('Et', rest)
    if not succ: return False, rest
    succ, rest = parse('#', rest)
    if not succ: return False, rest
    return True, rest
```

Quello che facciamo in generale è che ogni volta che un non terminale aveva delle alternative le abbiamo eseguite in cascata invocando una certa funzione che usava quella alternativa, se restitueva true tornavamo altrimenti andavamo nell'else. Ci servono due funziononi di utilita': la prima per indentare un blocco di testo al livello di indentazione specificato e la seconda per tradurre il codice sorgente di una funzione Python di nome parse in codice eseguibile:

```
def indent_at(level, source):
  return indent(dedent(source), '  ' * level).strip('\n')

def make_parse(source):
  glb = {'show_calls': show_calls}
  exec(source, glb)
  return glb['parse']
```

Siamo pronti per procedere, useremo le f-stringhe per maggior compattezza:

```
def make_parse_source(G):

  # prima le definizioni delle funzioni A_altN

  defs = []
  for A in G.N:
    for n, a in enumerate(G.alternatives(A)):
      # la funzione "{A}_alt{n}"
      defs.append(f'def {A}_alt{n}(rest):')
      for X in a:
        # una coppia di linee per ogni simbolo
        defs.append(indent_at(1, f'''
```

```
          succ, rest = parse('{X}', rest)
          if not succ: return False, rest
        '''))
      defs.append(indent_at(1,'return True, rest'))

  # poi gli if, uno per terminale

  ifs = []
  for A in G.N:
    # l'if "X == '{A}'"
    ifs.append(f"if X == '{A}':")
    for n, _ in enumerate(G.alternatives(A)):
      # una coppia di linee, una per alternativa
      ifs.append(indent_at(1, f'''
        succ_alt, rest_alt = {A}_alt{n}(rest)
        if succ_alt: return True, rest_alt
      '''))
    ifs.append(indent_at(1,'return False, rest'))

  # in fine i terminali

  ifs.append(indent_at(0, f'''
    if X in {G.T}:
      if rest and rest[0] == X: return True, rest[1:]
      return False, rest
  '''))

  # e ora mettiamo tutto assieme

  parse = '\n'.join((
    indent_at(0, '''
      @show_calls(True)
      def parse(X, rest):
    '''),
    indent_at(1, '\n'.join(defs)),
    indent_at(1, '\n'.join(ifs))
  ))

  return parse
```

Faremo per tutti i non terminali genereremo una serie di or e poi una funzione specifica che fa la concatenazione. L'unica cosa che ci serve è mettere a posto l'indentazione. Python ci permette, data una stringa sorgente renderla eseguibile, prende un sorgente di una parse e restituisce una funzione che fa il parsing (reflection di python). Se generaiamo e stampiamo il sorgente otteniamo questo:

```
source = make_parse_source(G) # G e la grammatica prefix-free
   dell'inizio
print(source)

@show_calls(True)
def parse(X, rest):
  def Ets_alt0(rest):
    succ, rest = parse('Et', rest)
    if not succ: return False, rest
    succ, rest = parse('Ets', rest)
```

```
    if not succ: return False, rest
    return True, rest
  def Ets_alt1(rest):
    succ, rest = parse('Et', rest)
    if not succ: return False, rest
    succ, rest = parse('#', rest)
    if not succ: return False, rest
    return True, rest
  def E_alt0(rest):
    succ, rest = parse('Eh', rest)
    if not succ: return False, rest
    succ, rest = parse('Ets', rest)
    if not succ: return False, rest
    return True, rest
  def E_alt1(rest):
    succ, rest = parse('Eh', rest)
    if not succ: return False, rest
    succ, rest = parse('#', rest)
    if not succ: return False, rest
    return True, rest
  def Et_alt0(rest):
    succ, rest = parse('+', rest)
    if not succ: return False, rest
    succ, rest = parse('T', rest)
    if not succ: return False, rest
    return True, rest
  def Et_alt1(rest):
    succ, rest = parse('-', rest)
    if not succ: return False, rest
    succ, rest = parse('T', rest)
    if not succ: return False, rest
    return True, rest
  def Eh_alt0(rest):
    succ, rest = parse('T', rest)
    if not succ: return False, rest
    return True, rest
  def T_alt0(rest):
    succ, rest = parse('t', rest)
    if not succ: return False, rest
    return True, rest
  if X == 'Ets':
    succ_alt, rest_alt = Ets_alt0(rest)
    if succ_alt: return True, rest_alt
    succ_alt, rest_alt = Ets_alt1(rest)
    if succ_alt: return True, rest_alt
    return False, rest
  if X == 'E':
    succ_alt, rest_alt = E_alt0(rest)
    if succ_alt: return True, rest_alt
    succ_alt, rest_alt = E_alt1(rest)
    if succ_alt: return True, rest_alt
    return False, rest
  if X == 'Et':
    succ_alt, rest_alt = Et_alt0(rest)
    if succ_alt: return True, rest_alt
    succ_alt, rest_alt = Et_alt1(rest)
```

```
    if succ_alt: return True, rest_alt
    return False, rest
  if X == 'Eh':
    succ_alt, rest_alt = Eh_alt0(rest)
    if succ_alt: return True, rest_alt
    return False, rest
  if X == 'T':
    succ_alt, rest_alt = T_alt0(rest)
    if succ_alt: return True, rest_alt
    return False, rest
  if X in frozenset({'+', '-', 't', '#'}):
    if rest and rest[0] == X: return True, rest[1:]
    return False, rest
```

Notiamo che è una ricorsione con backtracking, non scendo all'infinito ma mi fermo prima in alcuni casi.

La cosa drammatica di questo parser è che se ho due alternative tengo sempre buona la prima, ma se il linguaggio non è context free potrei entrare dentro il prefisso e poi dentro li andrei avanti all'infinito. In questo esempio vediamo che parsando alla fine ottengo true con una lista non vuota, quindi sembrerebbe che non ha parsato:

```
# una grammatica non prefix-free (L = {a, ab})

G = Grammar.from_string("""
S -> a | a b
""")

# costruiamo il parser e tentiamo il parse

source = make_parse_source(G)
parse = make_parse(source)
parse('S', 'ab')

(True, 'b')
```

Il problema che non è questa la strada, di fronte ad una disgiunzione $A->B|C$ non posso prendere per buona solo la produzione B ed ignorare il fatto che potrebbe produrre anche C (che potrebbe essere meglio), ed è ovviamente un grande limite. Quello che si potrebbe fare è avere un parsing con le continuazioni, sostanzialmetne restituisco nel parse anche una funzione che prova tutte le alternative. Il problema di questa cosa è che si risolve il problema ma non è più lineare, perchè fa tutte le alternative.

In buona sostanza avere un parsing ricorsivo discendente è fighissimo perchè è lineare, semplice da implementare e permette di avere già in mano l'interprete. Ad esempio il parsing ricorsivo discendente è quello di Python. Forse se stiamo costruendo il nostro DSL, con delle regole semplici (comandare drone, irrigazione ...), possiamo permetterci di avere un parsing ricorsivo discendente, ma se vogliamo fare un parser generico per un linguaggio generico non possiamo permetterci di avere un parsing ricorsivo discendente.

3.5 Parsing Bottom-Up

La realizzazione di parsing bottom up veniva più facile negli anni 70 infatti si e' partiti da questo tipo di parser, ragionare su parsing concreti per le grammatiche context-free

deterministiche era più facile. Aggiustare una grammatica per farlo funzionare in questo contesto è più difficile questo perchè funziona al contrario rispetto agli altri parser. Interessante perchè è a livello storico è il primo parsing che si occupa di un grosso numero di grammatiche in tempo lineare ma è doloroso perchè richiede uno sforzo al progettista di grammatiche.

Abbiamo sempre in mente la pila (che contiene i simbli), il nastro e il controllo. L'idea e' mettere sopra il nastro degli alberi, costruire degli alberi che rappresentano delle porzioni della parola (le derivazioni) e li uniscono verso l'alto per avere l'albero di derivazione costruito dal fondo all'alto. Quindi il punto cruciale di questo algoritmo è cominciare a guardare la base e poi la foresta che mi sono costruito sul nastro, quindi sulla pila devo tenere oltre ai tereminali e non terminali devo tenere anche questi alberi. Allora la cosa più comoda da considerare è che di fatto nella pila voglio tenere le cime degli alberi della foresta, ma correlata a questa informazione i passi che ho fatto.

3.5.1 Shift e Reduce

Quando vedo un carattere sul nastro posso:

- Metterlo sulla cima della pila, facendo uno shift
- Riconoscere che sulla pila c'è il contenuto di una regola produttiva, nella grammatica c'era da qualche parte $A \to CD$ allora potrei togliere dalla pila C e D e mettere A, quindi ho fatto una riduzione (reduce). Da notare che in questo esempio eravamo in una situazione in cui sulla pila c'erano D e C che a loro volta rappresentavano un albero di derivazione.

Quindi il controllo cerca regole della grammatica il cui lato destro è l'unione di elementi sulla pila. Questa configurazone non è proprio un'automa a pila, perchè non guardo solo la cima della pila, ma spio tutta la pila. Il che sembrerebbe suggerire che questa è una macchina ti Touring, ma scopriremo che non è così. Quello che va colto è che è un approccio inerentemente più complesso perchè l'indecisione è più grande, perchè non solo devo decidere in caso di regole multiple cosa fare ma devo anche decidere se fare uno shift o una riduzione e anche nel caso della riduzione devo decidere quale regola applicare.

Adesso come sempre se abbiamo in testa una cosa che non è deterministica non riusiamo nella pratica a trovare la soluzione, mi immagino di avere una forma descrittiva dello stato della computazione (una descrizione che racconta i passi della computazione), produrre una funzione next che dirà data una delle circostante cosa fare (uno shift e quale reduce fare) che mi dirà quali sono le successive istantanee oppure che nessun lato destro della grammatica fa parte della pila (siamo rovinati) o che abbiamo finito il nastro (abbiamo finito di parsare).

A questo punto simulo, parto da uno stadio iniziale e poi con una modalità breadth-first (allargo la vista) o con modalità depth-first (più pericolosa perchè potrei andare giù all'infinito).

Vediamo una grammatica e facciamo un atto di fede in cui la pila è undbound (perchè ci teniamo dentro anche i passi di derivazione):

```
1 G = Grammar.from_string("""
2 S -> A C
3 A -> a b
```

```
C -> c
""")

i = BottomUpInstantaneousDescription(G, 'abc')
# (), , |abc

i = i.shift()
display(i, side_by_side(i.stack))
# (), (a), a|bc

i = i.shift()
display(i, side_by_side(i.stack))
# (), (a)(b), ab|c

i = i.reduce(G.P[1])
display(i, side_by_side(i.stack))
# (A -> a b,), (A: (a), (b)), ab|c

. . .

i = i.reduce(G.P[0])
display(i, side_by_side(i.stack))
# (S -> A C, C -> c, A -> a b), (S: (A: (a), (b)), (C: (c))), abc|
```

Vediamo che la descrizione istantanea è come l'altra, ho i passi di derivazione, la pila vuota, la testina e il nastro. A botte di shift e reduce portiamo un po' di simboli sulla pila e quando vedo che sulla pila ci sono delle regole destre della grammatica tolgo quelle teste e ci metto l'albero di derivazione.

Adesso ragioniamo su come fare la funzione next, l'aspetto cruciale è che abbiamo un automa che ha delle primitive che si riflettono sulla sua natura di automa a pila, dopo di che siccome per ogni condizione in cui si trova ho un certo insieme finito di possibilità (non ristretto ad 1) posso fare una simulazione che devo fare con la funzione next. Il passo di riduzione è più difficile da implementare perchè devo prendere tutto lo stack, corrisponde ad una lettura completa della pila (dal punto di vista di programmazione ad oggetti svelo tutto lo stato dell'oggetto). All'occhio che qui non ci sono le epsilon regole.

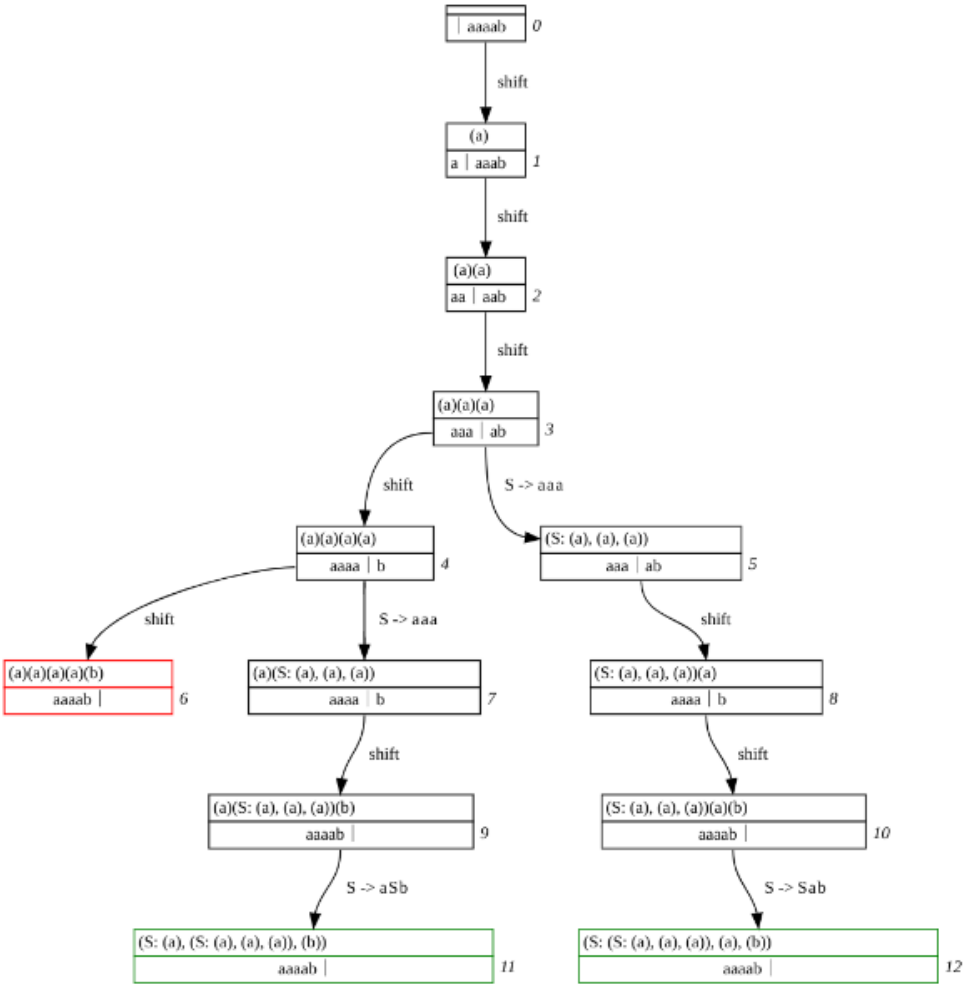
```
G = Grammar.from_string('S -> a S b | S a b | a a a')
derivations, graph = breadth_first(G, list('aaaab'), True, False) #
    raccolgo tutte le derivazioni
```

```
# restituisce un elenco di coppie (s, bid)
# dove s e' 'shift', o la str(P) dove P e' una produzione di G e bid e'
    una BottomUpInstantaneousDescription
# ottenuta applicando lo shift o la riduzione usando P a curr

def next_instdescrs(curr):

  if curr.is_done(): return []

  instdescrs = []

  # shift
  if curr.head_pos < len(curr.tape): instdescrs.append(('shift',
      curr.shift()))
```

```
13    if curr.G.has_epsilon_productions(): instdescrs.append(('e-shift',
          curr.shift(False)))
14
15    # reduce
16    tops = tuple(t.root for t in curr.stack)
17    for P in filter(Production.such_that(rhs_is_suffix_of = tops),
          curr.G.P):
18      instdescrs.append((str(P), curr.reduce(P)))
19
20    return instdescrs
21
22
23  # una funzione che associa ad ogni descrizione un colore
24  # a seconda che sia accettante (verde), o non abbia stati prossimi
        (rosso)
25  def node2color(tid):
26    return {'color': 'green' if tid.is_done() else 'black' if
          next_instdescrs(tid) else 'red'}
```

Possiamo usare lo stesso codice dell'altra volta per fare la visita in ampiezza, abbiamo visto un esempio data una grammatica con ricorsione a sinistra che produce due derivazioni right-most diverse. Vediamo che come l'altra volta abbiamo il parametro first_only che ci permette di fermarci alla prima derivazione verde.

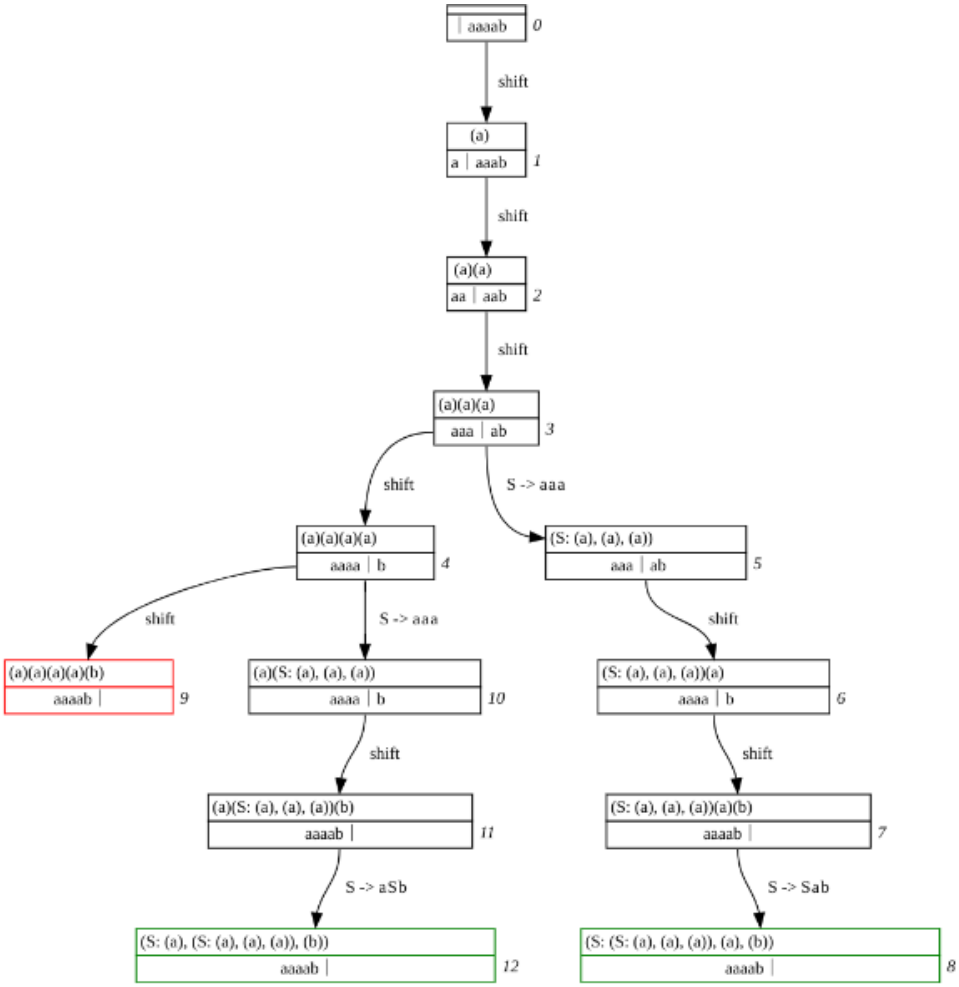
```
1   def breadth_first(G, word, verbose = False, first_only = True):
2     graph = ComputationGraph(node2color)
3     q = Queue()
4     q.enqueue(BottomUpInstantaneousDescription(G, word))
5     derivations = []
6     while q:
7       if first_only and derivations: break
8       if verbose:
9         for i in q: print(i)
10        print('-' * 60)
11      curr = q.dequeue()
12      for what, nxt in next_instdescrs(curr):
13        if nxt.is_done(): derivations.append(nxt.steps)
14        if not graph.seen(nxt):
15          q.enqueue(nxt)
16        graph.step(curr, nxt, what)
17    return derivations, graph
```



Anche l'approccio con visita in profondità è lo stesso dell'altra volta:

```
def depth_first(G, word, verbose = False, max_steps = -1, use_seen =
    True):
  graph = ComputationGraph(node2color)
  s = Stack()
  s.push(BottomUpInstantaneousDescription(G, word))
  derivations = []
  steps = 0
  while s:
    if steps > max_steps > -1: break
    steps += 1
    if verbose:
      for i in s: print(i)
      print('-' * 60)
    curr = s.pop()
    for what, nxt in next_instdescrs(curr):
      if nxt.is_done(): derivations.append(nxt.steps)
      if not (use_seen and graph.seen(nxt)):
        s.push(nxt)
      graph.step(curr, nxt, what)
  return derivations, graph
```

Vediamo un esempio che anche con una grammatica apparentemente elementare ed



una stringa corta ha dei problemi non visibili ad occhio nudo, se la proviamo con la visita in profondità è vero che arriva alla soluzione ma sbaglia un sacco di strade.

```
# fig. 7.8, pag. 204

G = Grammar.from_string("""
S -> E
E -> E Q F | F
F -> a
Q -> + | -
""")
```

Questo algoritmo come abbiamo visto è più difficile da ragionarsi sopra in particolare perchè restano due brutte cose da sistemare:

- Devo leggere tutta la pila
- Devo tenere tutti gli alberi

Ma anche supponendo che questi problemi teorici vengano risolti resta il fatto che purtroppo in questo caso le grammatiche che producono il disastro sono molto più difficili da vedere rispetto alle grammatiche che producono il disastro per i parser top down. Esistono strumenti automatici che ci dicono se la grammatica è ambigua o meno per i parser bottom up (LR(0)) ma è comunque doloroso modificare quelle grammatiche malate. Notiamo che TopDown e BottoUp si usano per grammatiche deterministiche e che senza ambiguità hanno costo computazionale $O(n)$, quindi non è un problema di efficienza ma di correttezza.

Dal lato dei parser Bottom Up ci sarebbe l'algoritmo CYK (che usiamo per le grammatiche generali ed ha costo computazionale n^3) bello, che è l'algoritmo di Early che funziona per le grammatiche generali che in caso di non ambiguità per le grammatiche determinstiche (quelle che usiamo TopDown e BottoUP) diventa $O(n)$.

3.6 Parsing top-down direzionale deterministico

Vogliamo metterci nella condizione per cui la cosa che adoperiamo che ha la testina la pila e la parola ad un certo punto vede la cima della pila e la prossima letterla della parola (sotto la testina) il controllo può fare la predizione. La necessità di seguire tutti i possibili cammini nel grafo che simula la computazione del NPDA deriva dal fatto che per un certo simbolo A in cima alla pila e il primo terminale b sotto la testina, ci sono più produzioni da considerare. In buona sostanza vogiamo una tabella, che dipende dalla grammatica, che mi dice la corripondenza tra simboli terminali e non terminali e grazie a questa corrispondenza riesco a derivare subito la produzione. Grazie a questa tabella implementare il controllo dell'automa è una banalità, se la cima è un terminale ti dico se c'è un match o no, se la cima è un non terminale ti dico se posso fare una predizione o meno. La magia grande è la tabella che deve funzionare, non deve contenere nessuna produzione se sono in un posto in cui non dovrei essere.

Adesso il punto è come produciamo questa tabella, in realtà la abbiamo già vista, perchè ad un certo punto abbiamo giocato con una grammatica che era così:

```
1    S -> aB
2    B -> b | aBb
```

Funziona bene perchè il primo carattere di ogni produzione è diverso, se sono in una circostanza per cui per ogni possibilità mi porta ad una produzione con il primo carattere diverso allora posso costruire la tabella:

	S	**B**
a	S -> ab	B -> b
b		B -> aBb

Queste grammatiche si chiamano SSL(1), dove 1 sta per il numero di simboli che guardiamo dopo la testina.

Per costruire la tabella prendo le produzioni della grammatica, le smonto come lato sinistro e letterina e suffisso e la produco così (se c'è un tentativo di riassegnamento mi avvisa che non è una grammatica che posso trattare), per ogni produzione $A \to b\delta$ basta considerare il primo simbolo che è terminale e distingue le varie alternative:

```
  def compute_simple_table(G):

  TABLE = Table(2, no_reassign = True) # no_reassign sara chiarito in
     seguito

  for P in G.P:
    A, (b, +d) = P
    TABLE[A, b] = P

  return TABLE
```

Ma se la grammatica non è semplice? nella costruzione della tabella mi avverte che c'è un conflitto (cerca di mettere due entry per la stessa coppia).

Questo modo di costruire i parser è facile ma utile, ad esempio è la notazione prefissa (notazione polacca) che è una notazione che non ha bisogno di parentesi, perchè il parser è in grado di capire la precedenza degli operatori. Ad esempio:

```
G = Grammar.from_string("""
E -> + E E | - E E | * E E | / E E | t
""")
```

Questo modo di scrivere le espressioni è molto comodo perchè è immediato parsare le associazioni e le precedenze, ad esempio:

$$*+324 \to (3+2)*4$$
$$+3*24 \to 3+(2*4)$$

Questa costrizione è molto stringente, non avere due simboli uguali in due produzionzioni diverse è una cosa che non è sempre possibile, ad esempio vorrebber dire non poter avere if e if else. Allora una cosa su cui possiamo ragionare è questa, supponiamo di sapere per ogni forma sentenziale qual'è la prima lettera che compare nella derivazione. Mi interessa conoscere data una forma sentenziale i prefissi, di tutte le parole che questa forma sentenziale può produrre quali sono i primi caratteri di queste derivazioni. Se avessi questa roba qui potrei costruire la tabella in un'altra ottica, metto in tebella le produzioni basandomi sui first, ad esempio se voglio collocare A -> w e so che first(w) = a, b, f allora la tabella sarebbe così:

	A
a	A -> w
b	A -> w
...	...
f	A -> w

Il first di una forma sentenziale nella forma in cui nessuna delle parti può scomparire (in quanto non ci sono epsilon regole) è il primo simbolo che compare nella derivazione. Nel riempire la tabella posso fare una fattorizzazione, prendo la A e la spacco e poi prendo tutte le lettere che vengono prodotte da B e riempio la tabella:

```
  def compute_table(G):

  TABLE = Table(2, no_reassign = True)
```

```
4
5   FIRST = compute_first(G)
6
7   for P in G.P:
8     A, (B, *d) = P
9     for a in FIRST[B]:
10      TABLE[A, a] = P
11
12  return TABLE
```

Con questa cosa del first riesco a vedere i conflitti un po' in avanti, un conflitto lo ho quando due produzioni hanno lo stesso first.

Si suggerirebbe un approccio ricorsivo, questo per guardare i first, ricordando che questo è perchè non abbiamo le epsilon regole. Il calcolo di $FIRST(\beta)$ dove si ponda $\beta = C\delta$, può essere semplificato osservando che $FIRST(C\delta) = FIRST(C)$.

```
1  def compute_first(G):
2
3    def recursive_first(B): # B e G.T | G.N
4      return union_of(
5        recursive_first(C)
6        for C, *d in G.alternatives(B) if B != C # occhio alla ricorsione
             a sinistra!
7      ) if B in G.N else {B}
8
9    FIRST = Table(1)
10   for X in G.N | G.T: FIRST[X] = recursive_first(X)
11
12   return FIRST
```

Vediamo un esempio con una garmmatica di un ipotetico sistema esperto nel quale prima si indica un fatto, una sequenza di fatti è una sessione e poi si termina con una domanda:

```
1  # fig. 8.7, pag. 240
2
3  G = Grammar.from_string("""
4  Session -> Fact Session | Question | ( Session ) Session
5  Fact -> ! STRING
6  Question -> ? STRING
7  """)
```

Ricorsivamente possiamo costruire i first, l'idea è che posso usare una regola che ha sessione a sinistra solo se sul nastro vedo uno di questi ?, (), !. E a questo punto posso costruire la tabella di derivazione, in questo caso non ci sono conflitti e quindi posso costruire la tabella:

Question	?
Fact	!
Session	(, !, ?

L'uso della tabella first mi consente di sapere, non tanto sulla scorta della prima letterina dei lati destri, ma sulla scorta dei first della grammatica (le letterine), se questa

tabella non ha conflitti sono a posto, posso chiamare questa roba LL(1) e posso chiamare il parser che funziona:

1. first
2. table if no conflitti:
3. parser

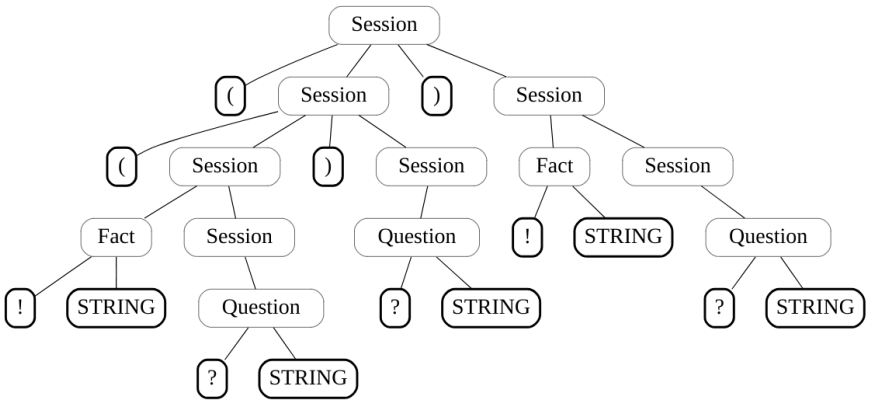
Abbiamo trovato un parser deterministico lineare, lineare perchè legge ogni lettera una volta sola.

Session	Session -> Fact Session	Session -> Question	Session -> (Session) Session
Fact	Fact -> ! STRING		
Question	Question -> ? STRING		

Ora possiamo usare esattamente lo stesso `parse_noε`, passando questa tabella, per derivare una parola

```
INPUT = '( ( ! STRING ? STRING ) ? STRING ) ! STRING ? STRING'.split()
```

```
ProductionGraph(parse_noε(G, TABLE, INPUT))
```



3.6.1 Cosa accade con le epsilon regole

La presenza delle epsilon regole serve a tradurre meglio l'idea semantica che ho in testa, una sessione di colloquio con il mio sistema esperto è una sequenza di fatti, potenzialmente vuota, seguita da una domanda. Per fare questa roba ho bisogno di fare sparire i fatti, quindi ho bisogno di epsilon regole.

```
# fig. 8.9, pag. 242

G = Grammar.from_string("""
Session -> Facts Question | ( Session ) Session
Facts -> Fact Facts | e
Fact -> ! STRING
Question -> ? STRING
""")
```

Cominciamo col ragionare per gradi, come facciamo la tabella dei first con le epsilon regole? Quello che potrei fare è fare in modo che table sbirci non solo il primo simbolo ma un po' più in giù nella pila, in realtà vedremo che non è necessario, si potrà pre-computare una tabella.

Occuparsi di first in presenza di epsilon regole non è così devastante. Qui non possiamo più fattorizzare il fatto che dobbiamo per forza prendere la prima lettera perchè non sparisce, non è vero, perchè ora può sparire perchè ci sono le epsilon regole. Non vale più $FIRST(B\beta) = FIRST(B)$. Nel caso $FIRST(B\gamma)$:

- $\epsilon \notin FIRST(B)$ allora $FIRST(B\gamma) = FIRST(B)$ sono nel caso di prima
- $\epsilon \in FIRST(B)$ allora $FIRST(B\gamma) = (FIRST(B) - \epsilon) + FIRST(\gamma)$, dove gamma sono i simboli che seguono B. Il dramma quindi è che devo capire cosa succede a gamma.

Faccio una chiusura, first la computo per tutte le forme sentenziali, quindi una entry della tabella è una tupla: Riempio mettendo first smontando la forma sentenziale tranne la nullabilità il prefisso e se c'è la nullabilità il suffisso. Consideriamo anche la stringa vuota e il simbolo di fine nastro(per praticità, considereremo anche il caso di suffissi di lunghezza nulla, a cui corrisponderà ϵ). Ad ogni passo la chiusura, per ciascun suffisso $\omega = C\delta$, aggiungerà allàinsieme First(ω) i terminali in:

- First(C) se $\epsilon \notin First(C)$
- First(C) - ϵ e First(δ) se $\epsilon \in First(C)$

Per concludere, la seguente implementazione popola FIRST con $FIRST(\omega)$ per tutti i suffissi dei lati destri delle produzioni di G anche qualora siano presneti epsilon regole:

```
def compute_efirst(G):

 FIRST = Table(1, element = set) # questo significa che gli elementi
    dell tabella sono insiemi

 # i casi base
 for t in G.T | {e, HASH}: FIRST[(t, )] = {t} # attenzione, gli indici
    sono forme sentenziali, ossia tuple!

 # il caso di forma sentenziale di lunghezza nulla
 FIRST[tuple()] = {e}

 @closure
 def update_with_suffixes(FIRST):
   for A, a in G.P:
     FIRST[(A, )] |= FIRST[a]
     for o in suffixes(a):
       C, *d = o
       FIRST[o] |= FIRST[(C, )] - {e}
       if e in FIRST[(C, )]: FIRST[o] |= FIRST[d]
   return FIRST

 return update_with_suffixes(FIRST)
```

Chiamando questa funzione sulla nostra grammatica otteniamo:

Session	{ (, !, ? }
Fact	{ ! }
Question	{ ? }
Facts	{ !, e }

3.7 Full LL(1)

Dobbiamo vedere se riusciamo ad usare questa cosa del First per fare il parsing, quello che mi interessa veramente è essere in un caso deterministico, quindi in qualche modo posso costruire la tabella a patto di poter calcolare questo $FIRST(\alpha\gamma\#)$. In buona sostanza quello che vorremmo e' calcolare TABLE in funzione fi FIRST.

Vediamo che per fare questo ci costruiamo una funzione, partendo da quella dell'altra volta che calcola la tabella dobbiamo estenderla per fare questo calcolo su una forma sentenziale, la chiusura funziona bene se il numero di passi è finito, ma qui non è noto a priori perchè parto da una lunghezza arbitraria, quindi l'unico modo di fare questa cosa è la ricorsione e la logica è questa:

- se la forma sentenziale è lunga uno ritorniamo una epsilon
- se c'è qualcosa da mangiare vuol dire che c'è un C e qualcos'altro. Tutto quello che viene da C lo devo prendere (perchè è il first), tranne epsilon, e poi devo vedere se C è nullabile, se lo è allora devo vedere il suffisso e quindi calcolare il first del suffisso.

```
1  def make_first_function(G):
2
3    FIRST = compute_first(G)
4
5    @show_calls(True)
6    def FIRSTf(w):
7      if not w: return {e} # gestisce il caso provocato dalle
           fattorizzazioni con |w| = 1, ossia |d| = 0
8      C, *d = w
9      if e in FIRST[(C, )]:
10       return (FIRST[(C, )] - {e}) | FIRSTf(d)
11     else:
12       return FIRST[(C, )]
13
14   return FIRSTf
```

Quindi adesso posto che siamo in grado di calcolare questo FIRST posso adoperare la costruzione della tabella con la stessa logica, quindi a questo punto avrei bisogno di una tabella (che non è più una tabella) che data la lettera a mi dice se posso sostituire quella produzione. La posso sostituire guardando tutta la pila, devo in qualche modo considerare tutto la pila, non posso guardare solo la cima proprio perchè se il top sparisce sono fregato. Quindi di nuovo voglio lavorare sulla costruzione della tabella ragionando sulla possibilità che la cima sparisca, questo modifica radicalmente l'algoritmo, perchè non usa più una pila. Come funzina la costruzione della tabella? dato che la pila sta in quelle condizioni le possibili produzioni sono queste: devo smontare la pila in A, gamma e cancelletto $A, \gamma, \#$, lo smonto e vado a prendere tutte le alternative, faccio la forma sentenziale e prendo tutte le alternative di a e se trovo una produzione che è in firts della

forma sentenziale. Una volta che ho testato tutte le alternative possono succedere tre cose:

- Non ne ho trovata nessuna
- Ne ho trovate più di una: non ho più il determinismo, quindi non posso costruire la tabella
- Ne ho trovata una sola: allora posso costruire la tabella

La costruzione non è precomputabile e dipende da un FIRST che deve guardarsi tutta la pila ed è importante notare che in questo caso a differenza di prima non possiamo scartare delle grammatiche, ma capiamo se qualcosa non va solo a runtime.

```
def make_table_function(G):

  FIRSTf = make_first_function(G)

  def TABLEf(stack, a):
    prods = set()
    A, *Ts = list(stack) # AT#
    for a in G.alternatives(A):
      w = a + tuple(Ts) # aT#
      if a in FIRSTf(w): prods.add(Production(A, a))
    if len(prods) == 1:
      return prods.pop()
    elif len(prods) > 1:
      warn(f'Conflict {prods=}')
    return None

  return TABLEf
```

Vediamo anche il parse, che è molto simile al parse no epsilon, con la differenza che non vedo una tabella precomputata ma invoco una funzione che ha il binocolo sullo stack. Le principali differenze sono:

- ignorare le epsilon aggiunte in pila (per via delle epsilon regole)
- usare la funzione TABLEf per calcolare la produzione da usare (invece di consultare la tabella TABLE)

```
def full_parse(G, TABLEf, INPUT):
  tid = TopDownInstantaneousDescription(G, INPUT)
  while not tid.is_done():
    if tid.top() in G.N:
      P = TABLEf(tid.stack, tid.head())
      if P is None:
        warn(f'No production for ({tid.top()}, {tid.head()}) at
           {tid.steps}')
        return None
      tid = tid.predict(P)
    else:
      if tid.top() == tid.head():
        tid = tid.match()
      else:
```

```
        warn(f'Expecting {tid.top()}, got {tid.head()}')
        return None
  return Derivation(G).leftmost(tid.steps)
```

Il punto cruciale è che purtroppo questa cosa $FIRST(\alpha\gamma\#)$ sembrerebbe non precomputabile. Un modo per risolvere un infinito come questo è quello di chiederci se vogliamo risolverlo in ogni circostanza, in questo caso noi sappiamo che se epsilon non fa parte di $FIRST(\alpha)$ allora il FIRST di tutto quello prima è uguale a $FIRST(\alpha) - \epsilon$ (il caso in cui non sparisce alpha) se invece sparisce abbiamo un po' di cosa che produco dalla tabella e la cosa che mi fa veramente male $FIRST(\alpha) - \epsilon \cup FIRST(\gamma\#)$. Quello che vorremmo calcolare è la funzione FOLLOW(A) che sono l'insieme dei terminali che mi porta ad avere $S \to xA\gamma \to xAt\gamma\#$. Stiamo dicendo che per scegliere una produzione A produce alpha guardo:

- i caratteri che produce FIRST(alpha)
- Se alpha sparisce allora guardo i caratteri dopo A, nel nostro caso usando FOLLOW(A)

Il dubbio è se questa cosa la possiamo precomputare. Andiamo per passi, vediamo prima se riusciamo a costruire la tabella, se riesco a costruire la tabella finita abbiamo ricostruito il parser. La calcoliamo così, guardo le letterine dentro FIRST se non c'è epsilon ho finito, se c'è devo guardare le letterine dentro FOLLOW. Quello che ci interessa è non avere conflitto nella tabella, adesso vorremmo distinguere due casi nei conflitti:

- Conflitto FIRST FIRST: due produzioni che hanno la stessa letterina in prima posizione
- Conflitto FIRST FOLLOW: due produzioni che hanno la stessa letterina in prima posizione e una delle due ha epsilon
- Conflitto FOLLOW FOLLOW: due produzioni che hanno la stessa letterina in prima posizione e una delle due ha epsilon

Quindi nella costruzione della tabella mi tengo delle tabelle intermedie giusto per sapere devo è avvenuto il conflitto. Provo tutti i non epsilon e li metto nella tabella, se c'è un epsilon provo tutti i Follow e li metto nella tabella (a parte la gestione degli errori)

```
def compute_table(G, FIRST, FOLLOW):

  TABLE = Table(2)
  FIRST_TABLE = Table(2)
  FOLLOW_TABLE = Table(2)

  for P in G.P:
    A, alpha = P
    for a in FIRST[alpha] - {epsilon}:
      if FIRST_TABLE[A, a] is not None:
        warn(f'First/first conflict on ({A}, {a}) for production {P}
           (was {FIRST_TABLE[A, a]})')
      else:
        FIRST_TABLE[A, a] = P
        TABLE[A, a] = P
```

```
    if epsilon in FIRST[alpha]:
      for a in FOLLOW[A]:
        if FIRST_TABLE[A, a] is not None:
          warn(f'First/follow conflict on ({A}, {a}) for production {P}
             (was {FIRST_TABLE[A, a]})')
        if FOLLOW_TABLE[A, a] is not None:
          warn(f'Follow/follow conflict on ({A}, {a}) for production
             {P} (was {FOLLOW_TABLE[A, a]})')
        if FIRST_TABLE[A, a] is None and FOLLOW_TABLE[A, a] is None:
          FOLLOW_TABLE[A, a] = P
          TABLE[A, a] = P
  return TABLE
```

Se la tabella contiene al più una produzione per ogni cella, allora la grammatica è detta strong LL(1). Osservate che se una grammatica è full LL(1) allora è anche strong LL(1).

Vediamo come computare FOLLOW, prendo questa produzione $z \to \alpha A \beta$ e la uso per calcolare $FOLLOW(A)$. Per stabilire cosa c'è nel FOLLOW(A) devo guardare tutte le possibili fattorizzazioni, quindi ci fa senz'altro i $FIRST(\beta)$ ma il dramma è sempre il solito, se beta sparisce cosa ci metto dentro? se sparisce ci sarà dentro banalmente FOLLOW(x). Per costruire chi segue un non terminale A, devo considerare i lati sinistri delle produzioni, parto da FIRST di beta ma se questo sparisce semplicemente un lato sinisto di A sarà x. Il follow di un terminale non mi interessa, non avrebbe senso.

```
def compute_follow(G, FIRST):

  FOLLOW = Table(1, set)
  FOLLOW[G.S] = {HASH}

  @closure
  def complete_follow(FOLLOW):
    for X, w in G.P:
      for Y in suffixes(w): # X -> alpha A Beta (alpha e' mangiato dal
         suffisso)
        A, *Beta = Y
        if A in G.T: continue
        FOLLOW[A] |= FIRST[Beta] - {epsilon}
        if epsilon in FIRST[Beta]: FOLLOW[A] |= FOLLOW[X]
    return FOLLOW

  return complete_follow(FOLLOW)
```

Vediamo questo con l'esempio dei FACTS and QUESTION, quella con le epsilon regole. La grammatica è questa:

```
G = Grammar.from_string("""
Session -> Facts Question | ( Session ) Session
Facts -> Fact Facts | epsilon
Fact -> ! STRING
Question -> ? STRING
""")
```

Precomputiamo prima le tabelle first e follow:

```
FIRST = compute_first(G)
FIRST.restrict_to({(N, ) for N in G.N})
```

Facts	!, ε
Fact	!
Question	?
Session	!, (, ?

```
FOLLOW = compute_follow(G, FIRST)
FOLLOW
```

Session	♯,)
Facts	?
Question	♯,)
Fact	!, ?
ε	?

Possiamo quindi costruire la tabella per la grammatica:

```
TABLE = compute_table(G, FIRST, FOLLOW)
TABLE
```

	!	?	(
Session	Session -> Facts Question	Session -> Facts Question	Session -> (Session) Session
Facts	Facts -> Fact Facts	Facts -> ε	
Fact	Fact -> ! STRING		
Question		Question -> ? STRING	

A questo punto vediamo il parser che è tornato banale perchè guardiamo la tabella precomputata (è identico al parser senza epsilon regole), l'unica cosa che cambia è la costruzione della tabella e il calcolo del follow.

```
def parse(G, TABLE, INPUT):
  tid = TopDownInstantaneousDescription(G, INPUT)
  while not tid.is_done():
    if tid.top() in G.N:
      P = TABLE[tid.top(), tid.head()]
      if P is None:
        warn(f'No production for ({tid.top()}, {tid.head()}) at
            {tid.steps}')
        return None
      tid = tid.predict(P)
    else:
      if tid.top() == tid.head():
        tid = tid.match()
      else:
        warn(f'Expecting {tid.top()}, got {tid.head()}')
        return None
```

```
16    return Derivation(G).leftmost(tid.steps)
```

3.7.1 Conflitti

Vediamo i vari casi di conflitto attraverso degli esempi. Introduciamo una funzione di comodo che faccia i vari passi di computazione:

```
1  def fft(G):
2    FIRST = compute_first(G)
3    FOLLOW = compute_follow(G, FIRST)
4    FF = Table(2)
5    for N in G.N:
6      FF[N, 'First'] = FIRST[(N, )]
7      FF[N, 'Follow'] = FOLLOW[N]
8    return FF, compute_table(G, FIRST, FOLLOW)
```

Caso first/first

```
G = Grammar.from_string("""
S -> a | A
A -> a
""")
```

```
FF, TABLE = fft(G)
```

```
First/first conflict on (S, a) for production S -> A (was S -> a)
```

```
FF
```

	First	Follow
S	a	♯
A	a	♯

```
TABLE
```

	a
S	S -> a
A	A -> a

Caso first/follow

```
G = Grammar.from_string("""
S -> A a
A -> a | ε
""")
```

```
FF, TABLE = fft(G)
```

```
First/follow conflict on (A, a) for production A -> ε (was A -> a)
```

```
FF
```

	First	Follow
S	a	♯
A	a, ε	a

```
TABLE
```

	a
S	S -> A a
A	A -> a

Caso follow/follow

```
G = Grammar.from_string("""
S -> A a
A -> B | C
B -> ε | b
C -> ε | c
""")
```

```
FF, TABLE = fft(G)
```

```
Follow/follow conflict on (A, a) for production A -> C (was A -> B)
```

```
FF
```

	First	Follow
B	b, ε	a
S	a, b, c	⊣
C	c, ε	a
A	ε, b, c	a

```
TABLE
```

	c	a	b
S	S -> A a	S -> A a	S -> A a
A	A -> C	A -> B	A -> B
B		B -> ε	B -> b
C	C -> c	C -> ε	

3.7.2 Riparare i conflitti

Riparare i conflitti è facile, l'idea è questa. Le cose vanno male quando succedono delle cose così $A \to \alpha\beta|\alpha\gamma$, si rompe quando due produzioni cominciano con la stessa cosa. Possiamo facilmente notare che e' impossibile in questo caso decidere che produzione prendere sulla base di $FIRST(\alpha)$. Una tecnica abbastanza banale per risolvere questi problemi è la fattorizzazione, mediante l'introduzione di un nuovo terminale N possiamo cambiare le due produzioni e sperare che $FIRST(\beta)$ e $FIRST(\gamma)$ siano disgiunti.

$$N \to \beta|\gamma$$
$$A \to \alpha N$$

Questa cosa non risolve sempre perchè se beta e gamma hanno un intersezione non vuota sono a capo, ma spero di poter differenziare i simboli.

Proviamo ad usare, come esempio, una grammatica per operazioni aritmetiche in cui ogni operatore coinvolga esattamente due sottoespressioni:

```
G = Grammar.from_string("""
E -> T + T | T
T -> F * F | F
F -> ( E ) | i
""")
```

Troviamo dei conflitti:

```
FF, TABLE = fft(G)
```

```
First/first conflict on (E, i) for production E -> T (was E -> T + T)
First/first conflict on (E, () for production E -> T (was E -> T + T)
First/first conflict on (T, i) for production T -> F (was T -> F * F)
First/first conflict on (T, () for production T -> F (was T -> F * F)
```

```
FF
```

	First	Follow
F	i, (	*, ♯, +,)
E	i, (	♯,)
T	i, (	♯, +,)

```
TABLE
```

	i	(
E	E -> T + T	E -> T + T
T	T -> F * F	T -> F * F
F	F -> i	F -> (E)

Che possono essere risolti fattorizzando sia T che F:

```
G = Grammar.from_string("""
E -> T E'
E' -> + T | epsilon
T -> F F'
F' -> * F | epsilon
F -> ( E ) | i
""")
```

```
FF, TABLE = fft(G)
```

```
TABLE
```

	i	(	+	♯	)	*
E	E -> T E'	E -> T E'				
E'			E' -> + T	E' -> ε	E' -> ε	
T	T -> F F'	T -> F F'				
F'			F' -> ε	F' -> ε	F' -> ε	F' -> * F
F	F -> i	F -> (E)				

3.8 Parser ricorsivo discendente predittivo

Scriviamo una famiglia di funzioni ricorsive ed una delle quali effettuerà il parsing del documento. Quello che ci aveva bloccato l'altra volta era che non sapevamo cosa fare quando c'erano due regole, prima prendevamo solo la prima possibilità, però non è detto che la prima scelta sia sempre quella giusta. La soluzione a questo problema è avere un parser ricorsivo discendente predittivo, abbiamo messo l'automa nelle condizioni di guardare il nastro e la pila, e avendo la tabella possiamo decidere cosa fare. Prendiamo in mano la TABLE e la adopero per costruire un parser ricorsivo discendente che quando deve decidere cosa fare guarda la tabella e non prende per forza la prima alternativa.

I passi che faremo sono:

- Costruzione manuale del riconoscitore
- Costruzione del parse che restituisca i passi di derivazione

- Costruzione automatica del parser (una funzione che crea la funzione parser)
- Costruire un interprete

Una cosa molto interessante nel parser topdwon è che si può mischiare il parsing al riconoscimento, potrei mano a mano che mangio l'input calcolare l'espressione (nel caso di grammatica aritmetica), possiamo quindi mischiare il processo di parse con quello di interpretazione. Nel caso dell'interprete è cruciale sciogliere i terminali generici (i -> qui ad un certo punto ci sarà un numero). Dovremo rendere più esplicito il processo di parsing e tokenizzazione.

Partiamo da una grammatica per le espressioni aritmetiche:

```
G = Grammar.from_string("""
Expr -> Term Expr'
Expr' -> PLUS Term Expr' | epsilon
Term -> Factor Factor'
Factor' -> TIMES Factor Factor' | epsilon
Factor -> LPAR Expr RPAR | NUMBER
""")
```

Con questa grammatica possiamo costruire la tabella:

```
FIRST = compute_first(G)
FOLLOW = compute_follow(G, FIRST)
compute_table(G, FIRST, FOLLOW)
```

	NUMBER	LPAR	PLUS	RPAR	$	TIMES
Expr	Expr -> Term Expr'	Expr -> Term Expr'				
Expr'			Expr' -> PLUS Term Expr'	Expr' -> ε	Expr' -> ε	
Term	Term -> Factor Factor'	Term -> Factor Factor'				
Factor'			Factor' -> ε	Factor' -> ε	Factor' -> ε	Factor' -> TIMES Factor Factor'
Factor	Factor -> NUMBER	Factor -> LPAR Expr RPAR				

Data questa tabella mi costruisco le funzioni parse per ogni riga della tabella, ogni funzione è costruita sulla base di quello che c'è nella riga della tabella (se incontri questo usa questa produzione), consideriamo che nella tabella gli spazi vuoti sono quelli di errore, non ci sono produzioni e devo segnalare l'errore. Riusciamo a distinguere tra le colonne utilizzando il look-ahed.

```
def parse_Expr():
  if token in {'LPAR', 'NUMBER'}:
    parse_Term()
    parse_Exprp()
  else:
    warn('Error parsing Expr')

def parse_Exprp():
  if token in {'RPAR', HASH}:
    pass # questa e' l'epsilon-produzione
  elif token == 'PLUS':
    consume('PLUS')
    parse_Term()
```

```
14      parse_Exprp()
15    else:
16      warn('Error parsing Expr_primo')
17
18  def parse_Term():
19    if token in {'LPAR', 'NUMBER'}:
20      parse_Factor()
21      parse_Factorp()
22    else:
23      warn('Error parsing Term')
24
25  def parse_Factorp():
26    if token in {'PLUS', 'RPAR', HASH}:
27      pass
28    elif token == 'TIMES':
29      consume('TIMES')
30      parse_Factor()
31      parse_Factorp()
32    else:
33      warn('Error parsing Factor_primo')
34
35  def parse_Factor():
36    if token == 'NUMBER':
37      consume('NUMBER')
38    elif token == 'LPAR':
39      consume('LPAR')
40      parse_Expr()
41      consume('RPAR')
42    else:
43      warn('Error parsing Factor')
```

Usa l'uguaglianza con il token (che al momento è un mistero ma rappresenta quello che c'è sul nastro) e in base a questa uguaglianza decide se fare una produzione o meno, se ho delle epsilon regole non faccio niente, altrimenti usano il lato destro, se c'è un terminale lo consumano e se c'è un non terminale fanno il parser di quello.

3.8.1 Tokenizzazione

Ci manca di capire cosa fa la funzione consume e il tokenizzatore. Ci piacerebbe molto ragionare in questo modo, l'input è una sequenza di caratteri e le cose che adopero per descrivere la grammatica sono oggetti più sofisticati. Nella libreria che stiamo usando stiamo usando le str di Python che hanno un brutto difetto, non esistono i caratteri (ci sono eventualmente stringhe lunghe 1). Noi stiamo usando come parole del linguaggio le tuple di simboli (o stringhe lunghe 1).

- Simbloli -> stringhe
- Parole o forme sentenziali -> tuple di simboli
- Imput -> sequenza di simboli -> tuple

Quello su cui dobbiamo ragionare è come trasformare le sentenze della grammatica e trasformarla correttamente in una tupla formattata correttamente per il parser. Abbiamo quindi bisogno di un processo che prenda la forma sentenziale, la trasformi in una

opportuna forma che il parser possa adoperare:

$$'34 + 2' \rightarrow ('3','4','+','2')$$

Questa cosa si chiama TOKENIZZAZIONE, che ha l'obiettivo di prendere una sequenza di caratteri di un alfabeto (tipo ASCII) e trasformarla in una sequenza di simboli.

Abbiamo una prima grammatica che è quella di tokenizzazione, è quella che vede la stringa vera e propria e ha un insieme di terminali che si chiama alfabeto di macchina (la roba che sta scritta dentro il file). Quella grammatica lì è una collezione di linguaggi regolari, l'idea è che voglio frazionare la stringa usando una serie di grammatiche regolari. Teniamoci divisi i non terminali e le produzioni, distinguiamo i simboli distinti, tanti linguaggi regolai e il simbolo distinto è il nome di quello che vediamo.

I terminali del parser sono i simboli distinti del tokenizzatore. Il parser lavora dalle teste degli alberi del tokenizzatore e il tokenizzatore lavora dalle stringhe alle teste degli alberi del tokenizzatore. Il tokenizzatore associa anche ad ogni token il sua value (il token è il simbolo distinto e il value è il valore che ha), ad esempio un token NUMBER può avere un valore 37. Inoltre deve essere in grado di fare scrorrere il puntatore tra i non terminali solo nel caso in cui la produzione sia corretta.

Quello che vedremo è quella che fa anche ANTLR, che useremo, produce una prima grammatica che è quella di tokenizzazione e poi per ciascun non terminale produce tramite espressioni regolari una grammatica che si mangia i simboli distinti e produce i token. La grammatica di tokenizzazione la rappresentiamo come un unione di espressioni regolari:

```
EPXR_TOKEN2PATTERN = (
  ('LPAR',   r'\('),
  ('RPAR',   r'\)'),
  ('PLUS',   r'\+'),
  ('TIMES',  r'\*'),
  ('NUMBER', r'\d+')
)
```

Adesso devo operativamente usare questa cosa per ottenere il token ed il valore. Abbiamo una funzione consume che consuma il token e il valore, quindi la funzione consume è una funzione che mi dice se il token che ho in testa è quello che mi aspetto, se è così lo consumo e lo restituisco. Se non è così mi da errore. Faccio quello che abbiamo visto con le espressioni regolari, metto in or tutte le espressioni regolari e tra quegli or avrò a non null solo il simbolo che ho matchato. Quando ha finito di iterare invece di lanciare un eccezione restituisce un token speciale di tappo. Da notare che usiamo due variabili globali token e value per immagazzinare il look-ahed:

```
def make_consume(word, token2pattern):
  tokens_re = re.compile('|'.join(f'(?P<{token}>{pattern})' for token,
      pattern in token2pattern))
  token_value_iterator = iter((token, value) for match in
      tokens_re.finditer(word) for token, value in
      match.groupdict().items() if value)
  def _advance():
    global token, value
    token, value = next(token_value_iterator, (HASH, None))
  _advance()
  def consume(expected):
    if token != expected:
```

```
      warn('Expected {}, found {}'.format(expected, token))
    else:
      _advance()
  return consume
```

3.8.2 Interprete, valutazione deglle espressioni

Usando il parser posso tornare indietro con il tokenizzatore usando i valori per tirare fuori il valore dell'espressione. Costruiamo delle espressioni ricorsive per cui per ogni terminale mi danno il valore dell'espressione.

```
def eval_Expr():
  if token in {'LPAR', 'NUMBER'}:
    val = eval_Term()
    val += eval_Exprp()
    return val
  warn('Error parsing Expr')

def eval_Exprp():
  if token in {'RPAR', HASH}:
    return 0 # questa e' l'unita addittiva
  elif token == 'PLUS':
    consume('PLUS')
    val = eval_Term()
    val += eval_Exprp()
    return val
  warn('Error parsing Expr_primo')

def eval_Term():
  if token in {'LPAR', 'NUMBER'}:
    val = eval_Factor()
    val *= eval_Factorp()
    return val
  warn('Error parsing Term')

def eval_Factorp():
  if token in {'PLUS', 'RPAR', HASH}:
    return 1 # questa e l'unita moltiplicativa
  elif token == 'TIMES':
    consume('TIMES')
    val = eval_Factor()
    val *= eval_Factorp()
    return val
  warn('Error parsing Factor_primo')

def eval_Factor():
  if token == 'NUMBER':
    val = int(value) # la variabile globale modificata da consume
    consume('NUMBER')
    return val
  elif token == 'LPAR':
    consume('LPAR')
    val = eval_Expr()
    consume('RPAR')
```

```
    return val
  warn('Error parsing Factor')
```

Posso estendere la grammatica per fare in modo che il parser prenda anche delle espressioni regolari, in particolar modo con l'asterisco indichiamo che possiamo mettere N termini come quello. Nelle funzioni indichiamo questa cosa con un while, quindi se ho un asterisco lo metto dentro un while, devo però stare attento a quando devo fermarmi dentro il while:

```
def eval_Expr():
  if token in {'LPAR', 'NUMBER'}:
    val = eval_Term()
    while token == 'PLUS':
      consume('PLUS')
      val += eval_Term()
    if token in {'RPAR', HASH}: return val
  warn('Error parsing Expr')

def eval_Term():
  if token in {'LPAR', 'NUMBER'}:
    val = eval_Factor()
    while token == 'TIMES':
      consume('TIMES')
      val *= eval_Factor()
    if token in {'PLUS', 'RPAR', HASH}: return val
  warn('Error parsing Term')
```

3.9 Parsing deterministico bottom-up LR(0)

Ci serve un modo per eliminare il non determinismo, in questo caso non possiamo usare la tabella perche' qui non guardiamo neanche il nastro per scegliere se fare shift o reduce. Useremo un automa a pila. L'idea e' che la pila sara' sia la sequenza di stati dell'automa, quindi maneggiare la pila sara' come maneggiare con gli stati dell'automa, con la cosa bella che togliere le cose dalla pila significa andare indietro nella storia dell'automa.

3.9.1 Automa di KNOT

La cosa cruciale e' come costruire questo automa. Usa degli accorgimenti molto simili a quando ci siamo convinti della regolarita' del linguaggio, qui per fare questa costruzione si usa storicamente un'altra rappresentazione rispetto a quella che avevamo usato.

Item

Si costruisce a partire da una cosa che si chiama ITEM, un item e' una produzione annotata in cui scelgo un modo per indicare quanta parte della produzione ho mangiato con un pallino che indica che la parte a sinistra del pallino e' stata mangiata e la parte a destra del pallino e' quella che devo ancora mangiare $A \to \alpha \cdot \beta$. Questi oggetti saranno usati per rappresentare la computazione dell'automa che ci dice se dobbiamo fare shift o reduce. Per usare questi item dobbiamo riflettere su due aspetti:

- Come si passa da un item all'altro
- Vorremmo arrivare a compimento un automa deterministico. Sappiamo che si determizza con la power construction. Noi ragioneremo con una epsilon closure.

Una cosa cruciale dell'item e' che quel pallino si sposta verso destra (ogni volta che consumo una regola) e' il superamento di un simbolo che puo' fare sfruttando un terminale o un non terminale ed in ogni caso quello che mi interessa e' quello che viene dopo il pallino perche' e' la modifica che posso apportare al mio item, possiamo trovarci anche nel caso in cui il pallino e' alla fine della produzione, significa che quella produzione l'ho usata tutta.

La libreria Liblet contiene un po' di questa zuppa, partiamo da una grammatica MCD:

```
G = Grammar.from_string("""
E -> E - T | T
T -> n | ( E )
""")
```

la libreria ci permette di costruire un item a partire da una produzione, e possiamo dirgli di avanzare o meno:

```
# Si costruisce dato rhs e lhs (come le produzioni)

A, alpha = G.P[0]

item = Item(A, alpha)

item
E -> .E - T
# ha un metodo advance che sposta il punto e un metodo
# symbol_after_dot che riporta il simbolo che segue il
# punto (o None se il punto e' oltre la fine del rhs)

for X in alpha:
  item = item.advance(X)
  print(f'item = {item}, symbol_after_dot = {item.symbol_after_dot()}')
item = E -> E.-T, symbol_after_dot = -
item = E -> E - .T, symbol_after_dot = T
item = E -> E - T ., symbol_after_dot = None
```

L'idea di questo item e' che rappresenta un atto di derivazione, se mi trovo a sinistra di un simbolo non terminale potrei proseguire oppure cominciare a consumare una qualunque della alternative che ha quel simbolo a sinistra. Se ho questo item $A \to \alpha \cdot B\beta$ andando avanti mi trovere a sinistra di un non terminale, che significa trovarsi a sinistra di tutte le sue alternative. L'idea della **epsilon chiusura** e' che ogni volta che il pallino e' a sinistra di un non terminale mi trovo nella stessa condizione in cui sono a sinistra di una qualsiasi sua alternativa $B_n \to \cdot\beta$. Per ogni item $A \to \alpha \cdot B\beta$ mettiamo in un set tutte le alternative di B $B \to \cdot\gamma$ dove $B \to \gamma$ e' una produzione di B nella grammatican G. Trovarsi davanti ad un non terminale significa chidersi che sua alternativa adoperare, mettiamo tutte questa alternative in un set, che ci porta ad avere degli item set. Questo e' il codice di questa idea:

```
@closure
def epsilon_closure(items, G):
```

```
  added = set()
  for item in items:
    X = item.symbol_after_dot()
    if X is None or X in G.T: continue
    added |= {Item(X, Beta) for Beta in G.alternatives(X)}
  return frozenset(items | added)
# vediamo come si comporta sul set di item che contiene solo l'item
# ottenuto dalla prima produzione

set(epsilon_closure({Item(*G.P[0])}, G))
{E -> .E - T, E -> .T, T -> .(E), T -> .n}
```

Costruzione dell'automa

Ora vogliamo costruire l'automa, prima abbiamo costruito gli stati (rappresentano la possibile circostanza del nostro parsing)

Per noi lo stato iniziale dell'automa e' l'epsilon chiusura di S, il pallino a sinistra di tutte le alternative di S.

La funzione di transizione di questo automa e' la seguente, posso spostarmi da uno stato ai suoi successivi transendo a partire da qualunque simbolo, prendo un elemento dell'item set, chiamo la funzione advance che mi fa andare avanti e questo mi portera' ad avere altri item set che poi chiudo:

```
def delta(items, X, G):
  return epsilon_closure({item.advance(X) for item in items if
      item.advance(X)}, G)
```

Gli stati finali dell'automa, nell'idea iniziale che la condizione che ci piace e' quando terminiamo una produzione, saranno gli item set in cui sono in queste condizioni $A \to \alpha\cdot$, ho una sola alternativa di sostituzione.

Costruzione del parser

Parto da q0, poi ho una tranzione x che mi porta ad altri stati, li provo tutti, faccio una nuova transizione e vado avanti cosi'. Sostanzialmente e' una visita in ampiezza, ogni visita mi creera' dei nuovi insiemi e vado avanti fino a che non trovo uno stato finale. Da notare che mettiamo in coda alla grammatica un tappo, cosi' da sapere quando abbiamo finito di mangiare che mi evita di essere in una situazione in cui l'automa mi dica che ha finito quando non e' vero.

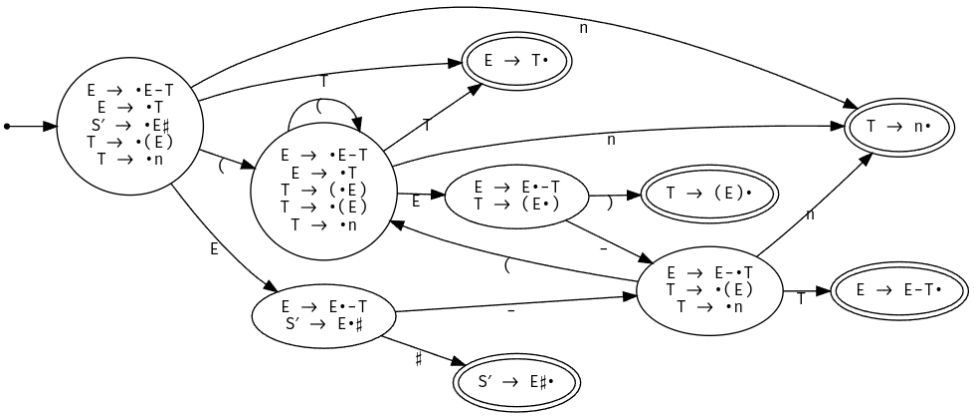
L'algoritmo e' la solita visita in ampiezza, mi tengo tutte le transizioni e gli stati e le provo tutte, sposta il pallino e se vede un non terminale ci mette dentro tutte le alternative:

```
def compute_lr0_automaton(G):

  states = {q0(G)}
  transitions = []
  q = Queue([q0(G)])
  while q:
    items = q.dequeue()
    for X in G.T | G.N:
      next_items = delta(items, X, G)
```

```
      if not next_items: continue
      transitions.append(Transition(items, X, next_items))
      if next_items not in states:
        states.add(next_items)
        q.enqueue(next_items)

  return Automaton(
    states,
    G.T | G.N,
    transitions,
    q0(G),
    set(state for state in states if len(state) == 1 and
       first(state).symbol_after_dot() is None)
  )
```

Esttamente come la costruzione della tabella, la costruzione dell'automa e' l'elemento discriminante che ci dice se la grammatica e' LR(0) o meno. Se la costruzione dell'automa termina con uno stato finale allora la grammatica e' LR(0), altrimenti non lo e' (non ci sono conflitti).

Guardando l'automa si capisce che fa ogni scelta di consumare qualcosa, poi quando arriva ad un punto in cui il pallino e' davanti ad un non terminale mette tutte le sue alternative.



Parsing con un automa a pila

L'idea e' che faccio il parsing seguendo l'automa a pila, mi muovo sul grafo e mano mano che mi muovo pusho gli stati sulla pila, quando arrivo ad uno stato finale tolgo quella produzione, torno indietro togliendo quella produzione dallo stato precedente.

Noi guardiamo solo la pila, non sempre necessariamente consumando lo stato che c'e' sulla cima, quindi l'automa spia quello che c'e' sulla pila, e' un po' diverso da un automa a pila. Se la cima della pila e' uno stato finale lo tiro fuori e lo smonto in un lato sinistro, lato destro e la posizione del pallino, significa che ho visto che un non terminale produce solo alpha, quindi so che la riduzione da fare e' quella. Ho sbattuto su uno stato finale, fai la riduzione che c'e' scritta dentro. Adesso devo riavvolgere la computazione facendo tanti pop quanti elementi ci sono nella parte destra della produzione. A questo punto devo scegliere in che direzione andare, siccome ho deciso di fare una riduzione con un non terminale mi sposto avanti seguendo lo stato del non terminale che ho scelto per la

riduzione. Se invece non sono in uno stato finale, vado a vedere nella cima del nastro e lo shift che devo fare e' decisa da quello che c'e' sulla cima della pila. Quando devo fare uno shift guardo quello che ho sul nastro e faccio lo shift seguendo quello che mi dice il nastro. L'idea cruciale e' che adopero la pila per tenere traccia di quello che sto computando. Da notare che produciamo una derivazione right-most, ad ogni passo di consume tiro fuori una regola di derivazione, che se poi leggo da sinistra verso destra ho proprio i passi di derivazione partendo da S.

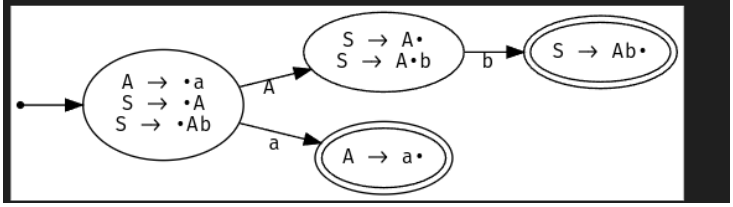
```python
def lr0parse(lr0a, text, G):

  bid = BottomUpInstantaneousDescription(G, text + [HASH, ])
  s = Stack([lr0a.q0])

  while True:
    top = s.peek()
    if top in lr0a.F: # REDUCE
      N, alpha, _ = first(top)
      bid = bid.reduce(Production(N, alpha))
      if bid.is_done(): return bid.steps
      for _ in alpha: s.pop()
      nxt = first(lr0a.delta(s.peek(), N))
    else: # SHIFT
      nxt = first(lr0a.delta(top, bid.head()))
      if nxt is None:
        warn(f'Unexpected token {bid.head()}')
        return None
      bid = bid.shift()
    s.push(nxt)
```

Conflitti

Possiamo vedere degli esempi di grammatiche piu' sfigate per cui il grafo non va bene, la costruzione del grafo mi porta ad avere un conflitto, ho uno stato in cui ho sia un pallino in fondo sia in un pallino che mi porterebbe a fare uno shift e l'automa non sa cosa fare, questo si chiama conflitto shift-reduce.

```python
# ad esempio

Gsr = Grammar.from_string("""
S → A b | A
A → a
""")
```

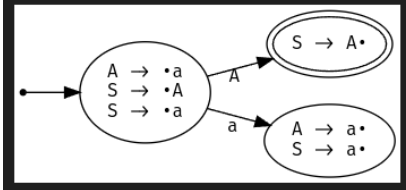
```python
lr0asr = compute_lr0_automaton(Gsr)
StateTransitionGraph.from_automaton(lr0asr, large_labels = True)
```



L'altra circostanza e' questa, sono nel caso di un loop, mi trovo in uno stato in cui posso fare due riduzione, e' un conflitto reduce-reduce.

```
# ad esempio

Grr = Grammar.from_string("""
S → A | a
A → a
""")
```

```
lr0arr = compute_lr0_automaton(Grr)
StateTransitionGraph.from_automaton(lr0arr, large_labels = True)
```



In questo caso è lo stato $\{A \to a^{\bullet}, S \to a^{\bullet}\}$ che può essere interpretato come un reduce $A \to a$ o $S \to a$.

Osserviamo che non possono mai esserci conflitti tra shift, perche' lo shift e' dato dal carattere sul nastro, non avrebbe senso un conflitto.

I conflitti LR non si vedono altrimenti se non costruire l'automa e vedere se e' rotto, non c'e' modo di vederli dalla grammatica. Cosa succede quando sono in una situazione di conflitto in LL o LR, un conflitto e' uno stato di non determinismo quindi la strada generalmente e' aumentare il look ahed, quindi se sono in una situazione di conflitto shift-reduce posso aumentare il look ahed e vedere se il prossimo simbolo mi da un indizio su cosa fare, LR(N). E' una strategia noiosissima perche' dovrei vedere molti piu' stati.

3.9.2 Tabelle GOTO e ACTIONS

Quello che si fa veramente nella pratica e' codificare l'automa, inizialmente si mettonono in una tabella tutti gli stati dell'automa:

```
states = sorted(map(set, lr0a.N))
iter2table(states)
```

0	{S' -> E ♯•}
1	{E -> E - T•}
2	{T -> (E•), E -> E•- T}
3	{T -> •n, T -> •(E), E -> •E - T, E -> •T, T -> (•E)}
4	{E -> T•}
5	{T -> (E)•}
6	{T -> •n, T -> •(E), E -> E -•T}
7	{T -> •n, T -> •(E), S' -> •E ♯, E -> •E - T, E -> •T}
8	{S' -> E•♯, E -> E•- T}
9	{T -> n•}

Se quella tabella ha avuto successo ho anche una mappa per fare le sostituzioni quando devo fare le reduce con le transizioni dell'automa:

```
index_of = states.index

GOTO = Table(2, set)
for frm in states:
  for X in sorted(lr0a.T):
    to = lr0a.δ(frm, X)
    if to:
      to = first(to)
      GOTO[index_of(frm), X] = index_of(to)

GOTO
```

	)	-	(	E	T	n	♯
2	5	6					
3			3	2	4	9	
6			3		1	9	
7			3	8	4	9	
8		6					0

Poi mi ci vuole la tabella delle azioni, sono nello stato n ma qui cosa faccio shift o reduce, contiene tutti gli stati in cui faccio una reduce ed e' una mappa tra il numero dello stato e la riduzione da fare:

```
ACTIONS = Table(1)

for n, state in enumerate(states):
  if len(state) == 1:
    A, α, _ = first(state)
    ACTIONS[n] = Production(A, α)

ACTIONS
```

0	S' -> E ♯
1	E -> E - T
4	E -> T
5	T -> (E)
9	T -> n

A questo punto il parser deve tenere il nastro e la pila e quello che devo tenere sulla pila e' la sequenza degli stati, di fatto si puo' provare a scriverlo anche senza le porcate di python che permette di tenere nella pila i set.

Chapter 4

ANTLR

ANTLR e' un parser generator che genera un parser LL top down multi linguaggio. A partire da una grammatica possiamo generare il parser nel linguaggio che vogliamo e ci genera un runtime nel linguaggio che vogliamo per fare funzionare il parser.

La roba interessante e' che ci sono un sacco di grammatiche pronte per ANTLR.

Abbiamo due modi per usare questo strumento:

- Uso diretto di ANTLR, leggendo il manuale e capire come funziona per scrivere il codice python
- Uso mediato da Liblet, significa che il prof ha scritto un po' di codice in Liblet che serve per importare roba di Java in maniera dinamica

Nel progetto e' una delle cose che possiamo scegliere.

4.1 Uso diretto di ANTLR

La cosa principale di questo strumento e' che ci permette di specificare la grammatica context free che vogliamo usare per produrre il parser tramite file o stringa. Questa grammatica e' fatta cosi', definiamo la grammatica, poi ci sono una serie di regole di tokenizzazzione scritte in maiuscolo alla cui destra c'e' una espressione regolare. Le regole della grammatica (di parsing) iniziano in minuscolo, sono date da un simbolo di tokenizzazzione e poi si possono mettere delle costanti. Alcuni token possono essere scartati dalla parte di parsing.

Adesso il punto cruciale e' utilizzare un tool Java che generi codice Python. Lui genera 4 file:

- lexer
- parser
- visitor: interfaccia che permette di visitare il tree generato dal parser
- listener: interfaccia

Uso del lexer, possiamo vedere i passi del lexer, il token ci fa vedere anche i caratteri del sorgente che rappresentano il token:

Abbiamo visto un esempio piu' complesso di una grammatica di espressioni ed assegnamenti. Vediamo ora come usare il paser, tendenzialmente i parser hanno due modalita' di funzionamento: dato che l'albero e' una struttura ricorsiva ci sono due modi per poterne fruire. Ognuno di questi nodi e' relativo ad una delle regole di parsing, i figli sono i token.

- listener: ha due metodi per ogni regola, entro ed esco. Poi posso invocare un processo di visita che invoca il listener secondo l'ordine che mi aspetto.
- visitor: me lo scrivo io, mi scrivo una funzione per ciascuna regola che sono quelle usate per fare la visita.

Quello che faremo noi a lezione e' usare visitor, perche' a lezione usando LIBLET ci dara' un albero e noi ci scriveremo sempre le visite dell'albero. Cosi' e' come useremo il listener, creo una classe che implementa il listener, poi creo l'istanza e lo passo al walker di liblet che lo usa per fare la visita:

Nel caso del visitor ho sempre la mia classe che implementa il vistor, mi implemento dentro la visita e poi uso la classe per fare la visita che mi sono scritto:

Fino ad ora non ci siamo mai occupati degli errori, ma in caso di ANTLR si possono specificare degli ascoltatori di errore che hanno un metodo che va invocato ogni volta che lui vede un errore nel lexing o nel parser:

4.2 Uso mediato da Liblet

Il modulo ANTLR dentro Liblet ci permette di prendere una grammatica e mi da un oggetto che ha delle competenze. In questa grammatica ci sono delle etichette di ANTLR che ci servono a distinguere le alternative di una regola che ha delle alternative che e' comodo perche' se avviene un match di stat non sapremmo cosa abbiamo matchato:

Dato l'oggetto possiamo fargli fare diverse cose, inizialmente vediamo i tokens, basta un metodo in questo caso:

Posso avere anche il contesto, che mi da dove partire per il visitor, posso dargli anche il token da cui partire:

L'albero di parsing possiamo produrlo con un metodo tree che produce un tree di liblet con dentro le espressioni:

ANTLR accetta anche grammatiche malate (ricorsivita', precedenze...) c'e' pero' una malattia che non puo' essere riconosciuta a tempo di analisi, la ambiguita'. Si puo' accendere una spia che ci dice se il parsing e' ambiguo, il suggerimento e' che quando si progetta una nuova grammatica si preparano dei testcase e vedo se il diagnostici mi dice se e' la grammatica e' ambigua. Va acceso il diag = True nel context:

4.2.1 Grammatiche note

Possiamo guardare alcune grammatiche note che possono essere utili, ad esempio c'e' una grammatica per l'aritmetica, bella ma l'associativita' a destra la sbaglia. Un'altra grammatica carina e CALCULATOR, che parsa le espressioni e permette l'invocazione di funzioni, come il seno o coseno.

Un'altra grammatica super utile, che non viene dal repository standard, ma dal libro di antlr e' la grammatica del linguaggio Cymbol che e' una specie di linguaggetto di programmazione che ha l'invocazione a funzione e gli statement.

La fine di questa cosa e' che e' molto potente, e se incontriamo errori guardiamo in documentazione perche' ci sono alcune porcherie.

4.2.2 Ancora su listener e visitor

Ora ci assegniamo due compiti ed useremo listener e visitor per fare due cose diverse:

- Numerare le righe
- Interpretare un linguaggio

Il linguaggio a cui faremo riferimento e' il linguaggio aritmetico con nomi di variabili ed assegnamenti. Una sequenza lineare di istruzioni in cui le istruzioni sono assegnamenti, stampe o espressioni.

Vogliamo contare gli assegnamenti, lo faremo con un listener, sappiamo che ANTLR ci da' un listener che e' un interfaccia da estendere, noi lo estendiamo e implementiamo i metodi che ci interessano. In questo caso ci interessa il numero di assegnamenti, quindi implementiamo il metodo che ci interessa. Poi creiamo un oggetto del listener e lo passiamo al walker di liblet.

Ora vogliamo interpretare il programma, un compito cosi' sofisiticato spesso si realizza piu' facilmente con un visitor, perche' quando scrivo un linguaggio ho in testa per ogni nodo del linguaggio cosa voglio fare, in particolare noi vogliamo che per ogni nodo restituisca il valore del nodo, questo lo puo' fare facilmente se fa una visita in post-ordine conoscendo i valori dei figli.

In questo caso abbiamo solo variabili locali, quindi la cosa piu' facile da fare e' tenere una memoria che e' un dizionario di python, se vediamo un assegnamento prendiamo il nome della variabile, facciamo lo scarico ricorsivo chiamando expr e salviamo tutto in memoria. Se vediamo una variabile, facciamo un lookup in memoria e restituiamo il valore. Se vediamo un numero lo restituiamo direttamente, da notare che quando restituiamo l'intero e' l'unico passo interpretativo perche' stiamo trasformando testo in intero. Abbiamo un espressione che fa la print, non ritorna niente ma fa una print. Dopo di che c'e' la parte di associazione, di nuovo sono divise perche' sono cose divise, in ogni caso visito tramite scarico ricorsivo il lato sinistro e destro e poi guardo il tipo dell'operatore, la visita delle parentesi e' un trucco sintattico che serve ad indicare diverse precedenze, quindi dal punto di vista interpretativo non serve a niente e si puo' solo mangiare.

Estendere questo parser su linguaggi piu' difficili e' immediato, la logica e' sempre la stessa, basta estendere le regole di parsing e le regole di interpretazione. Potrebbe essere un buon esempio. Il trucco cruciale e' che il visitatore e' ricorsivo quindi posso quando ho un dubbio su che cosa fare posso usare la ricorsione, per valutare un espressione valuto ricorsivamente i figli e poi faccio l'operazione. Il problema di questa cosa e' che la pila di chiamate ricorsive puo' esplodere e non e' facile da tenere sotto controllo, soprattutto se vogliamo dare degli errori durante il parsing. Quindi abbiamo bisogno di fare una traduzione in cui rendo la ricorsione esplicita, significa liberarsi della ricorsione passando ad una struttura iterativa, significa avere una pila di chiamate esplicita invece della pila di record di attivazione, l'idea qui e' che tengo dei risultati parziali. Qui dobbiamo scegliere tra visitor che usa la ricorsione e listener in cui teniamo una pila esplicita per le chiamate.

L'idea in questo caso e' di tenere gli operandi sulla pila e quando siamo pronti a fare l'operazione togliamo gli operandi e mettiamo il risultato intermedio. Quindi il listener avra' due strutture dati per funzionare, una memoria per le variabili ed una pila. Qui ho

bisogno degli exit perche' voglio sapere cosa fare con quello che c'e' sulla pila dopo che ho guardato i figli. Se usciamo da un assegnamento significa che ho gia' valutato l'espressione e sulla pila c'e' il valore da assegnare, quindi assegniamo in memoria. Se usciamo da un id o un intero significa che valutiamo il valore e mettiamo sulla pila il valore. Quando usicamo da una divisione mi aspetto che i valori dei suoi operandi siano sulla pila messi dalla visita in post-ordine, quindi li tolgo dalla pila e metto il risultato della divisione sulla pila. Se usciamo da una somma invece mettiamo il risultato della somma sulla pila.

4.3 Dall'albero di parsing all'AST

L'idea e' di partire da un albero di parsing che ha tanti artefatti, toglierli e raggiungere una forma compatta che e' un abstract syntax tree. Per farlo usiamo una funzione ricorsiva.

Abbiamo visto che in ANTLR si puo' mettere del codice direttamente nella grammatica, non lo faremo mai e' troppo doloroso, poi non ha troppo senso tenere legata la grammatica con l'implementazione del parser.

4.3.1 Dispatch table

Useremo una dispatch table per capire cosa fare quando troviamo una regola (nella table ci saranno le funzioni ricorsive). Inizialmente diciamoche quando troviamo una regola e c'e' nella dispatch table chiamo la sua regola, se non la trovo metto tutti in un catchall:

Devo riempire la tabella, in questo caso definiamo le funzioni che ci servono quando vediamo gli operatori, da notare che come scarico ricorsivo non usano se stesse ma chiamano la funzione che usa la dispatch table, poi popolo la table:

Si e' visto che liblet implementa una classe TreeWalker che fa questo lavoro della funzione ricorsiva, possiamo dirgli dato un albero che attributo considerare, registrare delle funzioni che usa quando incontra dei simboli (con un annotazione o registrarle esplicitamente), inoltre permette anche di avere un catchall (ne esistono di tanti tipi).

Vediamo un esempio in cui eseguiamo il programma partendo da un ATS, notiamo che occupandoci degli atomi abbiamo solo const e var, non abbiamo bisogno di castare ad Int perche' lo abbiamo gia' fatto nella costruzione dell ATS quando mettevamo dentro il valore di una costante.

4.4 Trasformare

Vediamo alcuni esperimenti di trasformazione, in particolare inizialmente vogliamo trasformare un documento JSON in un documento HTML usando ANTLR.

Quello che facciamo noi e' modularizzare il processo, partiamo generando l'AST, si fa con un tree walker ricorsivo che ha la tabella di dispatch che discrimina i nodi in base al nome.

4.4.1 Trasformare HTML in matrici

L'idea e' che ho acceduto ad una pagina HTML con dentro delle tabelle e io voglio farmi una matrice che rappresenta i dati, e' una trasformazione di dominio.

Per farlo partiamo con il definirci una nostra grammatica invece di usare quella completa assumendo per semplicita' che dentro i td possano esserci solo interi.

Il processo e' sempre lo stesso, arriviamo all'AST e qui la cosa che cambia e' che per fare la trasformazione usiamo una struttura dati esterna, uno stack, quando vediamo i dati li pushiamo dentro lo stack e quando vediamo una row togliamo i figli e li mettiamo dentro un array, alla fine dello stack avremo un array di array che rappresenta la matrice.

Abbiamo visto anche che per costruire la tabella potremmo pensare di decorare i nodi con il numero di riga e colonna mettendo nella radice il numero di righe e colonne, cosi' posso poi precomputare una matrice di dimensioni fisse piena di None e poi riempirla visitando l'AST.

4.5 Traspilazione

L'idea e' quella di prendere un linguaggio e trasformarlo in un altro linguaggio. Questa cosa ha origini storiche perche' ad esempio il C non veniva compilato ma traspilato in Fortran. Ci sono anche logiche di portabilita' e di ottimizzazione. La portabilita' e' moderna, uno dei linguaggi dominanti nel web e' JavaScript, il problema di questo linguaggio e' che non e' standardizzato e non coinvolge tutti i browser. Ragione per cui e' nata una famiglia di traspilatori che prendono un JS recente e lo trasformandono in un JS piu' vecchio che gira su tutti i browser. Un esempio e' Babel, che e' un traspilatore di JS, ma ce ne sono anche altri. Se esistono costrutti che nelle vecchie versioni non esistono si usano gli SHIM. E' nato anche typescript, che non esiste nei browser, ma viene traspilato in JS.

Noi partiamo da un linguaggio semplice SimpleLang, prima lo traspiliamo in JavaScript (facendo un text to text trsaplilazione). L'altro passo che faremo e' quello di sfruttare il fatto che alcuni linguaggi hanno una rappresentazione di alto livello, ma che e' un po' piu' snella di un testo (non richiede ulteriori processi di parsing) che puo' essere complata dalla VM e generata senza troppo sforzo, quindi tradurremo questo linguaggio verso Python facendo una traduzione da testo verso l'AST di python (AST = Abstract Syntax Tree).

Dobbiamo decidere come fare l'IO, perche' ha senso compilare un programma solo se c'e' un input. Noi per farla facile cercemo di provvedere modi banali per dare l'input ai nostri programmi, l'interprete preleva delle cose dal mondo esterno e le inietta nel programma. Nel nostro caso assumiamo che ci siano tre variabili INPUT e per farla facile faremo che ha un solo OUTPUT. (dice che il progetto di quest'anno non avra' bisogno di librerie esterne).

Arrivati infondo alla traspilazione dobbiamo cercare di valorizzare la variabile INPUT, facciamo una funzione che cerca tutti gli input e per ciascuni di essi apre un prompt che prende l'input.

Vediamo ora come cercare di evitare il passo di parsing del linguaggio in cui arriviamo. Scriviamo il linguaggio in una forma in cui l'interprete di python sia in grado di interpretarlo direttamente, python perche' e' fornisce delle API per interagire con gli AST. Si puo' usare eval per fare valutare una funzione e generare un parse tree. una volta che

abbiamo un AST possiamo compilarlo con la funzione complie di python che vuole sapere cosa sta compilando e in che modalita'. Per fare la traspilazione verso l'ATS di python dobbiamo visitare il nostro albero e trasformare i nostri nodi nei loro corrispondenti nodi di python.

Dopo siamo di nuovo nella situazione di dover valorizzare gli input, i programmi di python hanno un dizionario dove pesca le variabili locali, noi possiamo aggiungere con un dizionario di python le variabili che ci servono. Ora li stiamo mettendo a mano noi, potremmo anche pensare di mettere delle funzioni per valorizzare le variabili di input.

4.6 Symbol table

Assomiglia ad un dizionario, abbiamo due funzioni Bind e Look up con cui mettiamo dentro un simbolo e lo cerchiamo. Hanno anche la competenza di entrare ed uscire da un contesto (ENTER, EXIT) come contesto potremmo intendere classi o funzioni.

4.6.1 Scope

Nei linguaggi di programmazione moderni le variabili non solo visibili ovunque, ma hanno uno scope. Per farlo utilizzeremo delle opportune symbol table. Esiste uno scoping locale che serve a correggere uno scoping globale. Possiamo avere anche uno scope unico o distinto posso usare lo stesso simbolo in contesti diversi o meno (una variable che si chiama come una funzione).

4.6.2 Implementazione symbol table

Esistono svariate implementazioni, noi ne vedremo due.

Implementazione deterministica: e' la piu' facile ed adoperiamo una pila di coppie (simbolo, informazione), fare il bind di un simbolo significa fare il push della coppia simbolo informazione. Il lookup e' la ricerca in questa pila dall'alto verso il basso. Enter spingo nella pila qualcosa che sono sicuro di riconoscere tipo null, null, in modo tale che quando faccio exit posso fare una catena di pop fino a che non trovo il separatore, sostanzialmente metto un separatore per i contesti. E' un po' inefficiente perche' quando faccio un lookup devo scorrere tutta la pila, e anche l'exit non e' bellissimo.

Possiamo vedere anche la symbol table come albero di dizionario, avremo dei dizionari (chiave - valore) attaccati al loro papa'. Quindi in questo caso dobbiamo fare una ricerca al massimo lineare nel numero di scope, poi dentro i dizionari la ricerca e' 0(1). La cosa bella e' che quando faccio un enter creo un dizionario vuoto e lo metto come figlio del dizionario padre, quando faccio un exit lo tolgo. In questo caso non abbiamo bisogno di mettere i separatori, perche' il dizionario padre e' il separatore.

4.7 Interpretazione

Siamo arrivati al punto in cui possiamo cominciare a parlare di interpretazione, siccome l'albero della AST e' una struttura ricorsiva, possiamo fare una visita ricorsiva dell'albero e interpretare il linguaggio. Questa e' la cosa piu' semplice da fare.

Dopo di che aggiungeremo le funzioni e penseremo a come interpretare le funzioni, con l'idea che le funzioni siano rientranti, quando esco dalla funzione ritorno da dove l'ho chiamata, inoltre i parametri della funzione sono trattati come variabili locali nello scope della funzione, per fare questo useremo la pila dei record di attivazione (chiamato anche call stack).

Come esempio abbiamo sempre il SimpleLang in cui abbiamo introdotto anche il prodExpr. Vediamo come fare l'interprete ricorsivo, inchiodiamo una variabile globale e lo attacchiamo all'interprete, in python si possono incollare attributi agli oggetti, in questo caso lo facciamo con interpreter.GLOBAL_MEMORY. Il difetto dell'interpretazione ricorsiva e' il costo di esecuzione. Non abbiamo fatto robe molto innovative.

4.7.1 Funzioni

Vogliamo dare una nuova espressivita' al linguaggio, vogliamo dare un nome ad un pezzo di codice per eseguirlo piu' volte. Una funzione ha una collezione di parametri (formali) che operano dal punto di vista concettuale come variabili locali di quella funzione. Dobbiamo definire due momenti:

- Definizione di funzione: nome, parametri formali, corpo
- Invocazione di funzione: nome, parametri concreti (actual) che sono di fatto delle espressioni, potremmo avere il valore di ritorno potremmo quindi dividere le funzioni tra quelle che calcolano un valore a quelle che non lo fanno.

In quasi tutti i linguaggi un espressione costituisce uno statement e noi faremo cosi', inoltre ci occuperemo solamente di funzioni di primo livello, non ci occuperemo di chiusure.

Ci interessa ora vedere come usare la pila di chiamate. Inanzitutto abbiamo deciso che un programma e' una sequenza di funzioni, decidiamo che ci sia una funzione che eseguiamo prima delle altre (main), dobbiamo inserire nella grammatica la definizione di funzione con i parametri formali e nell'invocazione delle espressioni possono esserci altre espressioni. Le espressioni diventano statement e ci mettiamo il return statement che e' una comoda convenzione per indicare il valore di ritorno della funzione, questo return e' opzionale se non c'e' return assumeremo che la funzione ritorni None.

Ora vediamo cosa fare quando visitiamo il nodo di dichiarazione di funzione, puo' succedere che ci siano i parametri formali o no, dobbiamo salvarci il nome della funzione e i parametri formali. Da stare attenti che nel caso in cui ci siano i parametri devo andare di due in due perche' devo saltare la virgola.

La chiamata e' sostanzialmente diversa, di nuovo posso avere gli argomenti o meno (anche in questo caso devo saltare la virgola). In questo caso devo creare un nuovo albero che chiama la funzione.

Adesso e' il momento di occuparci della chiamata. Dobbiamo predisporre un record di attivazione, che e' un dizionario in cui mettiamo i parametri formali e i valori dei parametri concreti. Poi dobbiamo fare l'enter della funzione, quindi dobbiamo mettere il record di attivazione nella pila dei record di attivazione, poi dobbiamo visitare il corpo della funzione. Alla fine della visita della funzione dobbiamo fare l'exit della funzione, quindi togliere il record di attivazione dalla pila e restituire il valore di ritorno. E' una pila perche' man mano che chiamo aggiungo record e quando ritorno faccio il pop del record di attivazione. Quindi valutiamo le espressioni dei parametri concreti, prepariamo

il record di attivazione, facciamo il push del record di attivazione nella pila, passiamo il controllo, quando ho finito di fare faccio il pop del record di attivazione e restituisco il valore di ritorno.

Aggiungiamo un altro attributo globale ACTIVATION_RECORDS che e' una pila. La modifica cruciale che invece di cercare i valori nella memoria globale li cerchiamo nella pila di attivazione.

L'interpretazione del programma diventa la raccolta dell'interpretazione delle funzioni, le raccogliamo in un dizionario FUNCTIONS che indicizziamo con il nome delle funzioni e dentro ci mettiamo gli alberi di parsing delle funzioni. Interpretare il programma significa fare il visit del main, che come detto prima mettera' in pila il record di attivazione, a sua volta mettera' in pila i record di tutte le funzioni dopo main.

Il caso dell'istruzione return e' gestito aggiungendo una variable locale dal nome _retval per conservare il valore restituito.

Nella chiamata per preparare il record di attivazione dobbiamo valutare i parametri concreti con le espressioni (che sono proprio numeri nel nostro caso), il record e' un dizionario di coppie.

Abbiamo anche visto come fare il ritorno forzato quando abbiamo un return, ad esempio in un for o in un if, quello che va fatto semplicemente e' che quando valutiamo quelle espressioni non andiamo giu' dritto come al solito, ma dopo il visit(stat) controllo che non ci sia _retval nella memoria, se c'e' faccio return anche dal for o dall'if.

4.8 Tipi

Esistono errori untrapped o trapped. L'obiettivo e' avere un codice safe e well-behaved, per fare questo ci vengono in aiuto i tipi e tutte le tecniche che ci permettono di capire se il programma e' safe.

Alcuni linguaggi non hanno tipi ed altri sono tipizzati, in vario modo misura, nel senso che ogni istanza di variabile o funzione e' decorata con un tipo, questo si chiama tipi espliciti. Ci sono altri sistemi in cui i tipi non sono espliciti, ma sono impliciti, nel senso che il compilatore o l'interprete si occupa di dedurre il tipo della variabile o della funzione. Quello che vogliamo vedere adesso e' se un programma e' well-typed, e' ragionevole pensare che se un programma e' ben tipato allora e' anche safe, in realta' questa implicazione e' delicata perche' oltre ad avere un sistema di tipi dobbiamo averlo sound (ben fatto). Attenzione che la soundness nei linguaggi comuni non e' garantita.

Se viviamo nel mondo dei linguaggi tipati espliciti possiamo usare il typechecker che verifica la well-behavior del programma in fase statica. Invece se stiamo nel mondo dei linguaggi untyped o typed implicit dobbiamo fare un controllo dinamico. Tipicamente si usano tecniche miste.

4.8.1 Typechecking

Quello che faremo e' decorare l'AST con delle symbol table, in modo tale che quando visitiamo l'AST possiamo vedere se il programma e' ben tipato o meno. La cosa bella e' che la symbol table e' una struttura dati che ci permette di tenere sotto controllo i tipi delle variabili e delle funzioni, quindi possiamo usarla per fare il type checking.

Siamo in un mondo con solo tipi primitivi, solo interi e stringhe. Possiamo fare tytte le espressioni binarie con entrambi gli operandi interi, confrontare due stringhe oppure sommare due stringhe (ottenendone la concatenazione) o moltiplicare una stringa per un intero (ottenendone la ripetizione). L'enforcing di queste regole non puo' appartenere all'analisi lessicale del linguaggio, ma deve appartenere ad un analisi successiva.

4.9 Interprete Iterativo

Vorremmo ora vedere un modo per linearizzare l'albero di parsing, in modo tale da non dover usare la ricorsione per interpretare il linguaggio. L'idea e' quella di usare una pila per tenere traccia delle chiamate e dei risultati intermedi. In questo modo possiamo evitare di usare la ricorsione e rendere l'interprete piu' efficiente.

4.9.1 Code threading

L'operazione che impariamo a fare e' il code threading (nel senso di filo sartoriale), l'idea e' di tirare un filo attraverso i nodi dell'albero di parsing, poi se taglio gli archi e tengo i nodi che ho infilato nel filo ho una struttura linearizzata, e' una rappresentazione topologica dell'albero in modo tale che posso camminare su questa struttura iterativa eliminando la ricorsione. Decoriamo i nodi con un campo che punta al nodo successivo (anche piu' campi se ci sono piu' figli).

E' un ulteriore passo dopo l'AST decorato, la decorazione del nodo la facciamo in modo facile, lo decoriamo con un attributo _thread_ e ci mettiamo dentro un nodo, da dire e' facile, il problema di questa cosa e' che la funzione di hash e' mezza complicata, la cosa che dovremmo fare noi e' magari non usare gli alberi di liblet ma usare una roba ad oggetti ed il puntatore al nodo usare un numero, non l'hash del nodo, in modo da evitare la ricorsione infinita.

La cosa cruciale e' concatenare due fili, e' il concatenamento di due liste. E' comodo avere due nodi speciali BEGIN e END per indicare all'interprete dove cominciare e dove finire. Il modo piu' facile di fare il concatenamento e' una visita in post-ordine, in cui concateno i nodi figli e poi concateno il nodo padre. Vediamo che cosi' facendo torniamo anche da program e block, ma sono solo dei segna posti, non ci interessano, possiamo evitarli.

Selezione

In un if-else vogliamo aggiungere un nodo di unione, valutiamo la condizione, torniamo al nodo if-else cosi' l'interprete valuta se la condizione se e' true va al true altrimenti al false, gli alberi dei true e false vanno ad un nodo JOIN.

Iterazione

Funziona in modo abbastanza analogo, il filo nella condizione valuta se deve eseguire ancora il ciclo o meno.

4.9.2 Interprete

Per prima cosa, semplifichiamo i fili tenendo da parte solo le istruzioi ed i salti, iniziamo col trasformare ogni nodo (da usare nei salti) in un numero progerssivo, dopo aver numerato i nodi.

Ho bisogno di strutture dati, memoria globale, stack e numero di istruzione (IP, instruction pointer). Uso il trucco di tenere in cima alla pila quando c'e' un repeat il numero di volte che devo ripetere il ciclo.

Chapter 5

Progetto

L'idea e' quella di ragionare sulla ricorsione investigando la questione che la ricorsione e' una cosa potente ma costosa. Saltino e' un linguaggio che non ha iterazione, se non ho iterazione devo fare la ricorsione. Il punto e' che esistono una serie di strategie per mitigare questo approccio, un esempio di somma senza ricorsione:

```
Somma(n){
    if n == 0 {
        return 0
    } else {
        return n + Somma(n-1)
    }
}
```

Quando la ricorsione e' l'ultimo passo e' facile trasformare la ricorsione in iterazione. Questa cosa si chiama tail recursion, elimini la ricorsione con un while true in cui riassegno quello che ho sulla pila dello stack nel caso della ricorsione, deve essere l'interprete a fare questa cosa.

Quindi l'idea e' inizialmente implementare un interprete ricorsivo e poi implementare un linguaggio saltone che non ha ricorsione, eliminando la ricorsione con un while true eliminando la ricorsione di coda. Ocio che con la somma non funziona quella cosa che dicevamo, la ricorsione non e' l'ultima cosa che faccio, devo ritornare i risultati parziali, ma devo ritornare degli accumulati, di fatto mi porto la somma parziale come secondo argomento:

```
Somma(n, acc){
    if n == 0 {
        return acc
    } else {
        return Somma(n-1, n+acc)
    }
}
```

A questo punto posso scrivere con un while true:

```
Somma(n){
    acc = 0
    while true {
        if n == 0 {
            return acc
        } else {
            new_n = n -1
            new_acc = n + acc
            n = new_n
            acc = new_acc
        }
    }
}
```

Ha detto che fare l'interprete iterativo rende molto facile la trasformazione in saltone, ovvero togliere la ricorsione.

Gli errori devono essere trappati, non devono vedersi stack trace di python.