

CKY

Lingua del Quenya

Patric Reineri, Alessandro Scicolone
Alessandro Mazzei

Scopo del progetto

Vogliamo implementare l'algoritmo *Cocke-Kasami-Younger CKY* su 2 grammatiche differenti, e testare il comportamento finale su di esse.

Le grammatiche prese in considerazione sono quella inglese e quella del Quenya.

“Elen síla lúmenn’ omentielvo”

Una stella splende sull’ora del nostro incontro -Gildor

Algoritmo Cocke-Kasami-Younger CKY

L'algoritmo di Cocke-Younger-Kasami è un algoritmo utilizzato per parsificare stringhe secondo una grammatica libera dal contesto in Forma Normale di Chomsky. L'obiettivo principale di utilizzo di questo algoritmo è quello di costruire l'albero corrispondente.

Il CKY funziona costruendo una tavola triangolare, dove ogni cella $P[i, j]$ contiene l'insieme di simboli non terminali che possono generare la sottostringa della frase di input che inizia alla posizione i e finisce alla posizione j .

Quindi va ad iterare su 3 indici, ovvero j , i e k , scegliendo il simbolo non terminale da inserire nella posizione i, j in base all'esistenza di una regola della grammatica che ha una forma $A \rightarrow BC$, in cui esiste il simbolo B nel intervallo (i, k) e il simbolo C nel intervallo (k, j) , fino a generare l'inizio della sentence S

PSEUDO CODICE

Per l'implementazione su Python del codice del algoritmo, siamo partiti da uno pseudo codice del CKY, cercando di carpirne la struttura dei cicli di iterazione e le strutture dati e/o classi utili per l'implementazione su Python.

```
function CKY-PARSE(words, grammar) returns table

  for j ← from 1 to LENGTH(words) do
    for all {A |  $A \rightarrow words[j] \in grammar$ }
       $table[j-1, j] \leftarrow table[j-1, j] \cup A$ 
    for i ← from j - 2 down to 0 do
      for k ← i + 1 to j - 1 do
        for all {A |  $A \rightarrow BC \in grammar$  and  $B \in table[i, k]$  and  $C \in table[k, j]$ }
           $table[i, j] \leftarrow table[i, j] \cup A$ 
```

Figure 18.12 The CKY algorithm.

Studio sulle strutture dati utilizzate

Abbiamo quindi analizzato le strutture dati utilizzate:

- In primo luogo quella della tabella in cui inserire i dati, riferita nel pseudo-codice come $table[i, j]$ che rappresenta la tabella in cui vengono salvati i simboli non terminali.
- Come secondo punto ci siamo posti il problema di come rappresentare i simboli non terminali, in particolare abbiamo voluto estendere la struttura che contiene il singolo simbolo non terminale, per rendere più semplice in seguito la costruzione del albero ottenuto alla fine delle iterazioni.
- Infine la rappresentazione della grammatica e delle regole, che come vedremo in seguito abbiamo utilizzato una libreria esterna

Tabella

La tabella in cui salvare i nodi non terminale è stata implementata come una Matrice triangolare, sottoforma di array, che tramite trasformazioni degli indici e una corretta inizializzazione va a mappare una matrice triangolare in un singolo array in modo che il suo utilizzo degli indici sia quasi identico a quello di una normale matrice, e quindi il suo utilizzo sia il più simile possibile al utilizzo fatto nello pseudo codice.

La tabella viene inizializzata con elementi vuoti, ogni cella è un set (un insieme) vuoto

```
undefined - CKY.py

1  class Table:
2
3      def __init__(self, lenght: int):
4          self.elements = []
5          for x in range(int((lenght*(lenght+1)) / 2) ):
6              self.elements.append(set())
7
8          self.len = lenght
9          self.sentence_count = 0
10
11     def index0f(self, i, j):
12         return ((2*self.len - i + 1) * i) // 2 + (j - i)
13
14     def get(self, i: int, j: int):
15         if(i > j):
16             raise Exception("Indice non valido")
17         return self.elements[self.index0f(i,j)]
18
19     def insert(self, node: Node, i: int, j: int):
20         if(i > j):
21             raise Exception("Indice non valido")
22         self.elements[self.index0f(i,j)].add(node)
23
```

Nodo (simboli non terminali)

I simboli non terminali che popoleranno la tabella vista sopra, un nodo viene gestito come una classe Nope, che contiene una variabile label che rappresenta il simbolo, inoltre vengono salvati i "children" del nodo, che rappresentano i simboli terminali o non che genera la regola che lo ha prodotto, per esempio se la regola che ci fa produrre un Nodo n è $A \rightarrow BC$, allora il nodo n avrà come figli i 2 nodi che hanno label a B e C, nei rispettivi intervalli.

La struttura relativa ai figli serve per poter costruire facilmente l'albero a fine del algoritmo, ciò ci permette di non dover fare nessun tipo di navigazione del albero, ottenuto il Nodo S, che ha label il simbolo di Sentence della grammatica, per costruire l'albero basterà costruirlo a partire dai figli e così via.

```
undefined - CKY.py
1  class Node:
2
3      def __init__(self, label: str, node1 = None, node2 = None):
4          self.label = label
5          self.children = []
6          if(node1):
7              self.addChildren(node1)
8          if(node2):
9              self.addChildren(node2)
10
11      def isTerminal(self):
12          return len(self.children) == 0
13
```

Grammatica

La grammatica viene gestita e rappresentata tramite la libreria esterna `nltk`, con cui possiamo importare su Python la grammatica da file `.cfg` (con la grammatica scritta similmente a: $A \rightarrow B C / C \rightarrow \text{"word"}$), e di poter trasformare la stessa in Chomsky Normal Form qualora non lo fosse già.

Codice implementazione CKY

```
1 def CKY(words : List[str], grammar: CFG):
2     table = Table(len(words))
3     # first cycle
4     for j in range(0, len(words)):
5         # init node in j, j ( the word )
6         for production in grammar.productions(rhs=words[j]):
7             table.insert(Node(production.lhs().symbol(), Node(words[j])), j, j)
8     # second cycle
9     for i in range(j-1, -1, -1):
10        # third cycle:
11        for k in range(i, j):
12            # cycle of grammar rules:
13            for production in list(filter(lambda x: x.is_nonlexical() and (contains(x.rhs(), table, i, k, j)), grammar.productions())):
14                table.insert(
15                    Node(production.lhs().symbol(),
16                        table.getElement(production.rhs()[0].symbol(), i, k),
17                        table.getElement(production.rhs()[1].symbol(), k+1, j)
18                    ), i, j)
19
20
21     return table
```

Una volta definite le varie strutture dati, abbiamo semplicemente tradotto lo pseudo codice, in codice Python, il codice quindi prende in input la grammatica, già in Chomsky Normal Form e una lista di stringhe, ovvero la frase.

“Elen sila lumenna” viene inserito come: [“elen”, “sila”, “lumenna”]

Una volta inizializzata la tabella, a partire dalla lunghezza della frase data, cominciamo con le iterazioni.

Possiamo vedere come la grammatica venga utilizzata per leggere le regole al suo interno:

- `grammar.productions()` ci da tutte le regole della grammatica, a cui possiamo aggiungere dei filtri, come nel secondo ciclo `for`, in cui richiediamo solo le regole che hanno come elemento a destra della regola una delle parole della frase

- Nel ciclo `for` interno al `for` che ciclo su `k`, iteriamo su un sottoinsieme delle regole della grammatica, cioè solo le regole non unarie i cui figli sono presenti già nella tabella, nei corretti indici, ovvero (i, k) e (k, j)

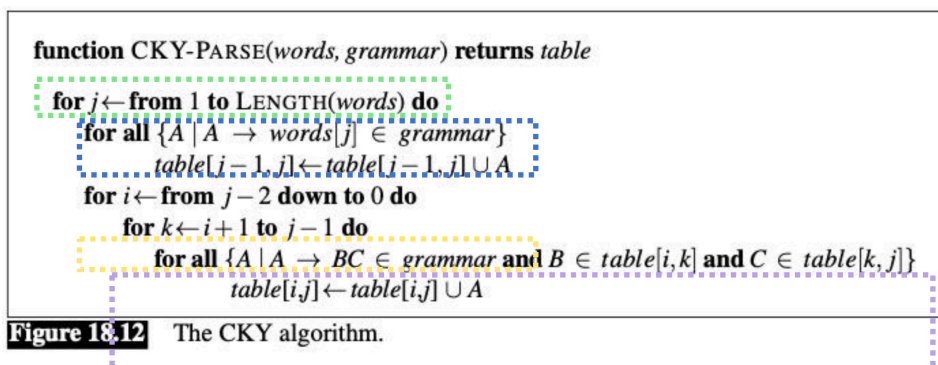
Possiamo notare come la struttura del codice Python sia rimasta veramente molto simile allo pseudo codice, per via della struttura della classe `Nope`, le insera dei simboli nella tabella, richiede più codice, dovuto al fatto di dover salvare e quindi dover recuperare anche i nodi figli, qui sotto mostriamo un paragone tra lo pseudo codice e il codice Python;



```

1 def CKY(words : List[str], grammar: CFG):
2     table = Table(len(words))
3     # first cycle
4     for j in range(0, len(words)):
5         # insert words[j] into table
6         for production in grammar.productions(rhs=words[j]):
7             table.insert(Node(production.lhs().symbol(), Node(words[j])), j, j)
8     # second cycle
9     for i in range(j-1, -1, -1):
10        # build CKY table
11        for k in range(i, j):
12            # cycle of grammar rules:
13            for production in list(filter(lambda x: x.is_nonlexical() and (contains(x.rhs(), table, i, k, j)), grammar.productions())):
14                table.insert(
15                    Node(production.lhs().symbol(),
16                        table.getElement(production.rhs()[0].symbol(), i, k),
17                        table.getElement(production.rhs()[1].symbol(), k+1, j)
18                    ), i, j)
19
20 return table

```



function CKY-PARSE(words, grammar) returns table

```

for j ← from 1 to LENGTH(words) do
  for all {A | A → words[j] ∈ grammar}
    table[j-1, j] ← table[j-1, j] ∪ A
  for i ← from j-2 down to 0 do
    for k ← i+1 to j-1 do
      for all {A | A → BC ∈ grammar and B ∈ table[i, k] and C ∈ table[k, j]}
        table[i, j] ← table[i, j] ∪ A

```

Figure 18.12 The CKY algorithm.

Grammatica Quenya

Per la grammatica abbiamo seguito i file e documentazioni fornite, in modo da poter comprendere le regole fondamentali della lingua, risultando molto simile a quella inglese, con qualche modifica, inoltre abbiamo aggiunto un numero limitato di regole unarie, cioè che hanno come elementi a destra le parole.

Inoltre nota fondamentale per la grammatica Quenya è che utilizza molti prefissi e postfissi, in una grammatica completa, si dovrebbe probabilmente gestire allargando le regole considerando il prefisso/postfisso come una parola a se, per semplificare la creazione della grammatica

```
1  S -> NP VP
2  S -> Pronoun VP
3  S -> Pronoun NP
4  S -> VP
5
6  NP -> Det N
7  NP -> N
8  NP -> N PP
9  NP -> Pronoun
10 VP -> V
11 VP -> V NP
12 VP -> V NP NP
13 VP -> V Adj
14 VP -> V NP Adj
15
16 PP -> Postposition NP
17
18 Det -> "i" | "in"
19 N -> "hesto" | "aran" | "aiwi" | "macil" | "tecil" | "eldar" | "elda" | "atan" | "eldan"
20 V -> "samë" | "tíra" | "nanye" | "nálmë" | "antanë"
21 Pronoun -> "nanye" | "nálmë" | "nán" | "nye" | "se"
22 Adj -> "lumba" | "alta" | "sina" | "nessa"
23 Postposition -> "o" | "arwa" | "rá" | "et" | "ú"
```


Abbiamo infine testato il comportamento del algoritmo sulle frasi di esempio date:

- Does she prefer a morning
- Book the flight through
- I hesto same macil
- I aran tira aiwi
- nanye lumba
- nalme eldar
- I atan antanè i eldan tecil

Ottenendo gli alberi previsti