IUSS - Pavia
A.A.2011/12
CRISTIANO CHESI

Minicorso in 3 puntate

TOP-DOWN SYNTACTIC DERIVATIONS

Indice

Letture, approfondimenti

- **Bibliografia essenziale**
  - Chesi (2007) *An introduction to Phase-based Minimalist Grammars: why move is Top-Down from Left-to-Right*. Studies in Linguistics, 1:38-75
  - Chesi (2007) *Five reasons for building phrase structures top-down from left to right*. Nanzan Linguistics, Special Issue 3.
  - Lazzari, Bianchi, Cadei, Chesi e Maffei (2010) *Informatica umanistica*. McGraw-Hill (Cap. 4)
  - Stabler, E. 1997. *Derivational minimalism*. in Retoré, ed. Logical Aspects of Computational Linguistics. Springer
- **Approfondimenti**
  - Barbara B.H. Partee, A.G. ter Meulen, R. Wall (1993*) Mathematical Methods in Linguistics* (Studies in Linguistics and Philosophy)
  - Hopcroft, Motwani & Ullman (2001) *Introduction to the automata theory, languages and computation*. Addison-Wesley. Boston
  - Sprouse, J., Wagers, M., & Phillips, C. (2012). Working-memory capacity and island effects: A reminder of the issues and the facts. *Language, 88*

I puntata

LE GRAMMATICHE FORMALI

La competenza linguistica

- Conoscere una lingua come l'Italiano significa sapere che:
  - una parola può iniziare per *ma...* (*mare*) ma non per *mr...*
  - la e di case ha un valore diverso da quella di mare
  - "le case sono sulla collina" Vs. *"case le collina sono sulla"
  - il gatto morde il cane > sogg: gatto(agente); verbo: morde(azione); ogg: cane(oggetto)
  - ?il tostapane morde il gatto
  - l'espressione "queste case" si riferisce ad un gruppo preciso di case, evidente dal contesto (Vs. "delle case")
- La nostra competenza linguistica è una conoscenza (finita) che ci permette di:
  - Sapere quali (infinite) espressioni fanno parte della propria lingua
  - Sapere cosa significano

La competenza linguistica

- Tracciare un confine che includa le cose che vogliamo spiegare:
  - Ordine delle parole > significato
    es. ho visto un uomo nel parco con il cannocchiale
  - Accordo
    es. *la mela rosso
    Gianni ha visto Maria vs. Gianni l'ha vista
  - Dipendenze a distanza (legamento pronominale, movimento sintattico)
    es. cosa credi che Maria abbia chiesto a Luigi di comprare _?

    $Gianni_i$ promette a $Maria_j$ di $\__{i/*j}$ andare a $trovarla_{i/k}$
    $Gianni_i$ chiede a $Maria_j$ di $\__{*i/j}$ andare a $trovarla_{*i/k}$

Livelli di adeguatezza di una grammatica

- **Adeguatezza**: una grammatica deve fornire una descrizione adeguata rispetto alla realtà empirica a cui si riferisce. In particolare si può parlare di adeguatezza a tre livelli:
  - **osservativa**: la lingua definita dalla grammatica coincide con quella che si intende descrivere
  - **descrittiva**: l'analisi grammaticale proposta è in linea con le intuizioni linguistiche dei parlanti fornendo descrizioni strutturali adeguate delle frasi accettabili
  - **esplicativa**: i dispositivi generativi utilizzati soddisfano criteri di plausibilità psicolinguistica e riproducono realmente i meccanismi operanti nell'attività linguistica del parlante. Una grammatica si dice esplicativa quando rende conto anche dell'apprendibilità della lingua.

Nozioni formali di base

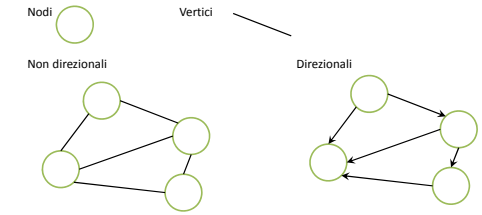
- **Insiemi definiti estensivamente** (finiti): A = {a, b, c}
- **Insiemi definiti induttivamente** (infiniti): A = {x: x ha la proprietà *p*}
- **Insiemi ordinati** (*n*-uple): A = (a, b, c)
- **Cardinalità**: |A| = numero di elementi di A
- **Prodotto cartesiano**: A = {a, b, c} B = {x, y}
  A X B = {(a, x), (b, x), (c, x), (a, y), (b, y), (c, y)}
- **Unione:** $A \cup B = \{x: x \in A \text{ oppure } x \in B\}$
- **Concatenazione:** $A \circ B = \{xy: x \in A \text{ e } y \in B\}$
- **Star**: $A^* = \{x_1x_2 \ldots x_n : n \geq 0 \text{ e ogni } x_i \in A\}$

Nozioni formali di base

- **Indici**: x_k = *k*-esimo elemento di una serie
  x^k = una serie di *k* elementi
  X^R = l'immagine speculare di X
- **Funzione**: f(x) → y (x = Domain, y = Range):
- **Predicati**: f(x) → {vero, falso}
- **Predicati n-ari**: f(x,y … z) → {vero, falso}
- **Relazioni di equivalenza**: predicati binari R per cui valgono le seguenti proprietà:
  - R è **riflessiva**, cioè per ogni x, xRx;
  - R è **simmetrica**, cioè per ogni x e y, se xRy allora yRx;
  - R è **transitiva**, cioè per ogni x, y e z, se xRy e yRz allora xRz;

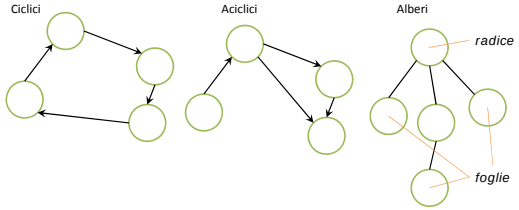
Nozioni formali di base

- **Grafi**



Nozioni formali di base

- **Grafi**



Grado: numero di vertici che esce/entra da un nodo

Come si formalizza una grammatica

- **A** = **Alfabeto**
  insieme finito di caratteri (A* = l'insieme di tutte le stringhe possibili costruite concatenando elementi di A; ε è l'elemento nullo)
- **V** = **Vocabolario**
  insieme (potenzialmente in)finito di parole, costruite concatenando elementi di A ($V \subseteq A^*$)
- **L** = **Linguaggio**
  insieme (potenzialmente in)finito di frasi, costruite concatenando elementi di V ($L \subseteq V^*$)

Come si formalizza una grammatica

- Una **grammatica formale** per il linguaggio **L** è un insieme di regole che permettono di **generare/riconoscere** tutte e sole le frasi appartenenti a **L** e(d eventualmente) di assegnare a queste frasi un'adeguata descrizione strutturale.
- Una grammatica formale G deve essere:
  - **esplicita** (il giudizio di grammaticalità deve essere frutto solo dell'applicazione meccanica delle regole scelte)
  - **consistente** (una stessa frase non può risultare allo stesso tempo grammaticale e non grammaticale)

Come si formalizza una grammatica

- Una grammatica formale G può essere formalizzata (grammatica a struttura sintagmatica o **Phrase Structure Grammar**, PSG Chomsky 1965), come una quadrupla ordinata (**V**, $\mathbf{V_T}$, →, **{S}**) dove:
  - **V** è il vocabolario della lingua
  - $\mathbf{V_T}$ è un sottoinsieme di **V** che racchiude tutti e soli gli elementi terminali (il complemento di $\mathbf{V_T}$ rispetto a **V** sarà l'insieme di tutti i vocaboli non terminali e sarà definito come $\mathbf{V_N}$)
  - → è una relazione binaria, asimmetrica e transitiva definita su V*, detta relazione di riscrittura. Ogni coppia ordinata appartenente alla relazione è chiamata regola di riscrittura.
    Per ogni simbolo $A \in \mathbf{V_N}$ $\phi A \psi \rightarrow \phi \tau \psi$ per qualche $\phi, \tau, \psi \in \mathbf{V^*}$
  - **{S}** è un sottoinsieme di $\mathbf{V_N}$ definito come l'insieme degli assiomi che convenzionalmente contiene il solo simbolo **S**.

Come si formalizza una grammatica

- Date due stringhe ϕ e $\psi \in V^*$ si dice che esiste una **ϕ-derivazione di ψ** se $\phi \rightarrow^* \psi$.
- Se esiste una ϕ-derivazione di ψ allora si può anche dire che **ϕ domina ψ**. Tale relazione è riflessiva e transitiva.
- Una ϕ-derivazione di ψ si dice **terminata** se:
  - $\psi \in V_T^*$
  - per nessun χ esiste una ψ-derivazione di χ
- Data una grammatica **G**, una lingua generata da **G**, detta **L(G)**, è l'insieme di tutte le stringhe ϕ per cui esiste una S-derivazione terminata di ϕ.

Descrizioni strutturali (cioè alberi sintattici)

- Una **Descrizione Strutturale** è una quintupla (**V**, **I**, **D**, **P**, **A**) tale che:
  - **V** è un insieme finito dei vertici (es. v_1, v_2, v_3…)
  - **I** è l'insieme finito degli identificatori (es. S, DP, VP, la, casa…)
  - **D** è la relazione di dominanza. È un ordine debole (cioè una relazione binaria, riflessiva, antisimmetrica e transitiva) definita su V
  - **P** è la relazione di precedenza. È un ordine stretto (cioè una relazione binaria, irriflessiva, antisimmetrica e transitiva) definita su V
  - **A** è la funzione di assegnazione; una funzione non suriettiva da **V** a **I**

Capacità generativa e relazioni di equivalenza

- La capacità generativa denota l'insieme di oggetti generati dalla grammatica; tale capacità è:
  - **debole** se riferita al solo semplice insieme di frasi generabili
  - **forte** se associa a tali frasi l'appropriata descrizione strutturale
- Due grammatiche si dicono **equivalenti** se sono in grado di **generare** lo stesso insieme di oggetti.
  Anche qua si può parlare di **equivalenza debole** o **equivalenza forte**

Decidibilità

- Un insieme Σ si dice
  - **decidibile** (o ricorsivo) se per ogni elemento *e* appartenente all'insieme universo esiste un **procedimento meccanico** M che permette di stabilire in un **numero finito di passi** se $e \in$ oppure $\notin$ a Σ (la non appartenenza a Σ determina l'appartenenza al complemento di Σ definito come $\overline{\Sigma}$)
  - **ricorsivamente enumerabile** quando esiste un procedura che enumera tutti e soli gli elementi di Σ

Grammatiche Regolari – Linguaggi Regolari

- Le grammatiche regolari sono grammatiche che ammettono solo regole sistematicamente di questo tipo:

  $A \rightarrow xB$

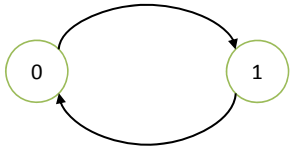
  oppure (sistematicamente) di questo:

  $A \rightarrow Bx$

  I linguaggi generati da queste grammatiche si definiscono **Regolari**

Automi e computazione

- Gli **automi** sono modelli matematici di computazione composti da stati che descrivono la «configurazione» del sistema e transizioni tra stati
- Ecco un esempio di automa (a stati finiti): l'interruttore!
  - 0 = acceso
  - 1 = spento
  - -> = pressione



Macchine e Stati Finiti (FSA)

- **Finite-State Automata** (**FSA**)
  definiti come quintuple **<Q, Σ, q_0, F, δ>** dove:

  Q = insieme finito e non nullo di stati

  Σ = alfabeto finito e non nullo di caratteri accettabili in input

  q_0 = stato iniziale, con $q_0 \in Q$

  F = insieme di stati finali, con $F \in Q$

  δ = insieme delle regole di transizione definite in QxΣ su Q

FSA usati come riconoscitori di parole

- un insieme di FSA non è solo un insieme di macchine che permettono di **riconoscere** o **rifiutare** un elemento lessicale, ma anche di **rappresentare** una frase (o tutta una lingua…).
- FSA che riconosce la parola *casa* ed il suo plurale:

Q = {q_0, q_1, q_2, q_3, q_4},
Σ = {c,a,s,e,#},
Q_0={q_0},
F ={q_4},
δ =

	q_0	q_1	q_2	q_3	q_4
c	q_1				
a		q_2			q_4
s			q_3		
e					q_4

Le Espressioni Regolari

- Le **Espressioni Regolari** sono notazioni algebriche per definire insiemi di stringhe di testo.
- Il cuore dell'espressione regolare è il **pattern di identificazione** composto da caratteri alfanumerici (compresi segni di spaziatura e di interpunzione) e da segni speciali volti a stabilire le relazioni tra i caratteri del pattern.

Espressione Regolare	Corrispondenza	Es. pattern identificato
[Dd]uomo	Duomo oppure duomo	Il duomo è nella piazza
[^a-z]	tutto fuorché lettere minuscole	Il duomo è …
sali?ta	salita oppure salta	Marco deve saltare
sal.ta	accetta ogni carattere tra le i e la t	Marco saluta
bu*	b seguito da un numero imprecisato (anche nullo) di u	buuuuuu! oppure b!
^L Vs. a$	^ = inizio stringa; $ = fine stringa	La casa
cas(a|e)	| è equivalente alla disgiunzione logica	Marco vive in un casale
*	il backslash è il simbolo di escape	A*

RG, FSA e RE sono equivalenti

- **Grammatiche Regolari** (RG, Regular Grammar), le **Macchine a Stati Finiti** (FSA) le **Espressioni Regolari** (RE, Regular Expressions) sono equivalenti (i.e. descrivono lo stesso insieme di linguaggi: i linguaggi regolari).

  Dimostrazione per costruzione

Digressione: Espressioni Regolari ed Eliza

- **Eliza**, un programma del 1966 scritto da Joseph Weizenbaum. Eliza simola il comportamento di uno psicoterapeuta Rogeriano che intraprende una conversazione con un paziente:

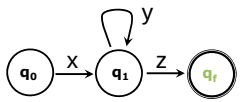
paziente: gli uomini sono tutti uguali
eliza: in che senso
paziente: ci fregano sempre in qualche modo
eliza: puoi fare un esempio specifico
paziente: un mio amico mi ha fatto venire qui
eliza: un tuo amico ti ha fatto venire qui
paziente: dice che sono quasi sempre depressa
eliza: mi dispiace sentire che sei depressa
paziente: è vero sono infelice
eliza: credi che venire qui ti aiuterà a non essere infelice
paziente: ho bisogno di aiuto, questo è sicuro
eliza: cosa intendi come aiuto
paziente: forse potrei riuscire ad andare d'accordo con mia madre
eliza: parlami di tua madre

Digressione: Espressioni regolari ed Eliza

- **Espressioni Regolari** e l'operazione di **Sostituzione**
  - La sostituzione è un'operazione che permette di sostituire l'occorrenza di un'espressione regolare con un'altra espressione regolare e può essere definita come segue:
  - s/espressione_regolare1/espressione_regolare2/
  - s/ www\.[a-z]*\.com / www\.iuss\.it/
- **Registri**: se si usano più blocchi di operatori (ogni parentesi tonda delimita un blocco), si può riutilizzare l'espressione trovata da un determinato blocco nell'espressione da sostituire, facendo riferimento all'ordine dei blocchi nel pattern di ricerca:
  - s/ la **(casa|macchina)** è stata comprata da **(Maria|Gianni)**/ **\2** ha comprato la **\1** / permette di costruire la forma attiva (**Gianni** ha comprato la **casa**) della frase passiva (la **casa** è stata comprata da **Gianni**).
- operazioni di **sostituzione** in ELIZA:
  - s/ sono [.* |]**(depress[o|a]|triste)**/sono spiacente di sapere che sei **\1**/
  - s/ sono tutt[i|e] **(.*)** /in che senso sono **\1**?/
  - s/ sempre / potresti far riferimento ad un esempio specifico?

Ogni espressione linguistica può essere generata dalle Grammatiche Regolari?

- **Pumping lemma per le grammatiche regolari**
  Se A è un Linguaggio Regolare, allora c'è un numero *p* (che esprime l'ampiezza del «pompaggio»), per cui, se *s* è una qualsiasi stringa di A di lunghezza almeno equivalente a *p*, allora può essere divisa in 3 parti, *s* = *xyz* tali che:
  I. Per ogni $i \geq 0$, $xy^iz \in A$
  II. $|y| > 0$
  III. $|xy| \leq p$



- **Proprietà di linguaggi non riconoscibili dalle grammatiche regolari**
  a^nb^n non è una stringa generabile da nessuna grammatica regolare (poiché nessuna sottostringa può essere "pompata" quante volte si vuole garantendo lo stesso numero di *a* e di *b*)

Grammatiche Context-Free

- Le grammatiche Context-Free (CFG) sono grammatiche che ammettono solo regole sistematicamente di questo tipo:

  $A \rightarrow \gamma$ (dove γ è una sequenza qualsiasi di simboli (non)terminali)

  I linguaggi generati da queste grammatiche si definiscono **Context-Free**

- Ogni grammatica CF può essere «convertita» in una grammatica (debolmente) equivalente nella forma chiamata **Chomsky Normal Form** (**CNF**):

  $A \rightarrow BC$
  $A \rightarrow a$

Push-Down Automata

- Un **Push-Down Automata** (PDA) è una macchina a stati finiti dotata di uno **stack di memoria**; i PDA sono definiti come sestuple **<Q, Σ, q_0, F, δ, Γ>** dove:

  Q = insieme finito e non nullo di stati

  Σ = alfabeto finito e non nullo di caratteri accettabili in input

  q_0 = stato iniziale, con $q_0 \in Q$

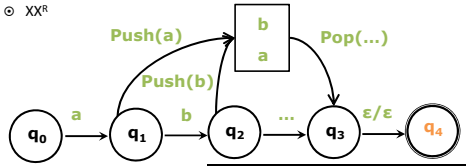
  F = insieme di stati finali, con $F \in Q$

  δ = insieme delle regole di transizione definite in $Q \times \Sigma \times \Gamma$ su $Q \times \Gamma$

  Γ = alfabeto di memoria

PDA usati come riconoscitori di frasi

- XX^R



$Q = \{q_0, q_1, q_2, q_3, q_4\}$,
$\Sigma / \Gamma = \{a, b, \varepsilon\}$,
$Q_0 = \{q_0\}$,
$F = \{q_4\}$,
δ =

	q_0	q_1	q_2	q_n	q_4
a	q_1 *push(a)*				
b		q_2 *push(b)*			
...			q_n *pop(...)*		
ε				q_4	

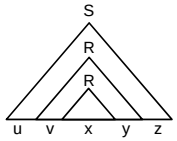
CFG e PDA sono equivalenti

- **Le Grammatiche Context-Free** (CFG, Context-Free Grammars), e i **Push-Down Automata** (PDA) sono equivalenti
  (i.e. descrivono lo stesso insieme di linguaggi: i linguaggi Context-Free).

  «Dimostrazione» per costruzione:
  1. PDA contiene una regola tale che: $(q_0, \varepsilon, \varepsilon) \rightarrow (q_1, S)$
  2. Per ogni regola della CFG tale che $A \rightarrow x$, il PDA contiene un'istruzione di questo tipo: $(q_1, \varepsilon, A) \rightarrow (q_1, x)$
  3. Per ogni simbolo $a : a \in V_T$, il PDA contiene un'istruzione del tipo $(q_1, a, a) \rightarrow (q_1, \varepsilon)$

Ogni espressione linguistica può essere generata dalle Grammatiche CF?

- **Pumping lemma per le grammatiche context-free**
  Se A è un Linguaggio Context-Free, allora c'è un numero *p* (che esprime l'ampiezza del «pompaggio»), per cui, se s è una qualsiasi stringa di A di lunghezza almeno equivalente a *p*, allora può essere divisa in 5 parti, s = uvxyz tali che:
  I. Per ogni $i \geq 0$, $uv^ixy^iz \in A$
  II. $|vy| \rangle 0$
  III. $|vxy| \leq p$



- **Proprietà di linguaggi non riconoscibili dalle grammatiche regolari**
  $a^nb^nc^n$ non è una stringa generabile da nessuna grammatica context-free (poiché nessuna tripla di sottostringhe può essere "pompata" indefinitivamente garantendo lo stesso numero di *a* di *b* e di *c*)

Inclusioni tra classi di grammatiche

- La **gerarchia di Chomsky** (1956, 59):

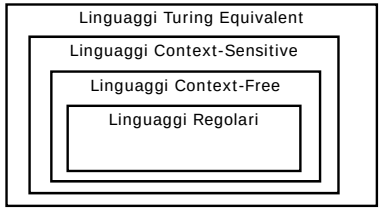
  Tipo 3: grammatiche regolari (**Regular Grammars**):
  $A \rightarrow xB$ (eq. Finite State Automata)

  Tipo 2: grammatiche non-contestuali (**Context Free Grammars**):
  $A \rightarrow \gamma$ (eq. Push-Down Automata)

  Tipo 1: grammatiche contestuali (**Context Sensitive Grammars**):
  $\alpha A \beta \rightarrow \alpha \gamma \beta$ $(\gamma \neq \varepsilon)$ (eq. Linear-Bounded Automata)

  Tipo 0: grammatiche non ristrette (**Turing Equivalent Grammars**):
  $\alpha \rightarrow \beta$ $(\alpha \neq \varepsilon)$ (eq. Augmented Transition Networks)

Inclusioni tra classi di grammatiche



Dove stanno le lingue naturali?

- Le lingue naturali **non** sono **generabili** da **grammatiche regolari** (Chomsky 1956):

  If A **then** B (con A e B potenzialmente anch'esse nella forma "**if** X **then** Y"... quindi linguaggi di tipo a^nb^n)

- Le lingue naturali **non** sono **generabili** da **grammatiche context-free** (Shieber 1985):

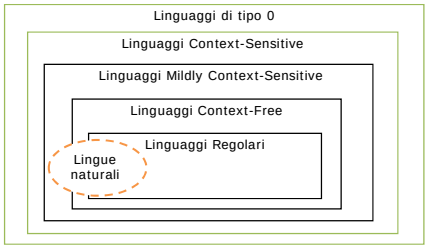
  Jan säit das mer em Hans es huus hälfed aastriiche
  ("famoso" dialetto svizzero tedesco)
  J. dice che noi a H. la casa abbiamo aiutato a dipingere

  Gianni, Luisa e Mario sono rispettivamente
  sposato, divorziata e scapolo
  ("ABC...ABC"... quindi linguaggi di tipo XX)

Dove stanno le lingue naturali?

- **Ricorsività** nelle lingue naturali, ovvero come fare un uso infinito di mezzi finiti:
  - **Incassamento a destra** (ab^n: iterazione):
    [il cane morse [il gatto [che rincorse [il topo [che scappò]]]]]
  - **Incassamento centrale** (a^nb^n: counting recursion):
    [il topo [che il gatto [che il cane morse] rincorse] scappò]
  - **Dipendenze cross-seriali** (xx, identity recursion)
    Gianni, Maria e Marco sono rispettivamente sposato, nubile e divorziato

Dove stanno le lingue naturali?



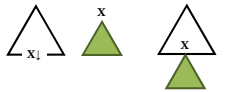
Tree-Adjoining Grammars (TAG, Joshi 75)

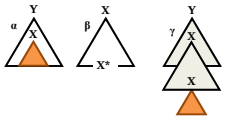
- Le **TAG** (Grammatiche ad aggiunzione di alberi (?)) generano linguaggi "**moderatamente" context-sensitive**
- Una TAG può essere definita come una quintupla **<Σ, N, I, A, S>** in cui:
  - **Σ** è un insieme finito di **simboli terminali**
  - **N** è un insieme finito di **simboli non terminali** ($N \cap \Sigma = \emptyset$)
  - **I** è un insieme finito di **alberi iniziali**, caratterizzati come segue:
    - i. i **nodi interni** sono etichettati con simboli non terminali;
    - ii. i **nodi estremi** possono essere etichettati con nodi non terminali (ed in tal caso dovranno sottostare all'operazione di **sostituzione**, marcata convenzionalmente con una freccia verso il basso ↓), o terminali.
  - **A** è un insieme di **alberi ausiliari**, caratterizzati come segue:
    - i. i **nodi interni** sono etichettati con simboli non terminali;
    - ii. i **nodi estremi** possono essere etichettati con nodi non terminali (ed in tal caso dovranno sottostare all'operazione di **sostituzione**, ad eccezione di un **nodo piede**, marcato convenzionalmente con asterisco *, il cui simbolo è identico al **nodo radice**), o terminali.
  - **S** è il simbolo iniziale della derivazione (Sentence)

Tree-Adjoining Grammars (TAG, Joshi 75)

- **IxA** è l'insieme degli **alberi elementari**.
- Un albero composto da più alberi binari combinati insieme secondo le regole di **sostituzione** e **aggiunzione** si chiama **albero derivato**.

sostituzione – è l'operazione che permette, all'interno di un albero α di rimpiazzare un nodo non terminale estremo marcato per tale operazione





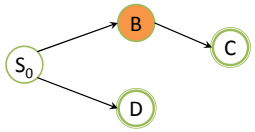
aggiunzione – è l'operazione che permette di costruire un nuovo alberoγ partendo dall'albero ausiliarioβ, il cui nodo radice è X, e daα (albero iniziale, ausiliario o derivato) che comprende un nodo anch'esso etichettato X, non marcato per la sostituzione

Il puntata

TEORIA DELLA COMPUTAZIONE (LINGUISTICA)

L'utilità di un modello computazionale

- Prevedere possibili disfunzioni

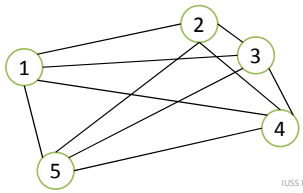


- Calcolare la complessità di certi processi…

Cos'è la complessità

- Ordinate 5 numeri (**sorting problem**)

- Trovate il percorso più breve che collega 5 città (**travelling salesman problem**)

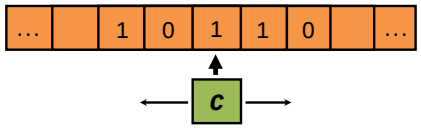


Cos'è computabile

- (informalmente) per **computazione** si intende la specificazione della **relazione tra un input ed un output**. Questa relazione può essere definita in vari modi formalmente chiamati **algoritmi**: descrizioni di una **serie di stadi** e **transizioni** tra questi stati in cui l'informazione di input viene **trasformata** prima di raggiungere la forma dell'output. Una computazione tenta quindi di ridurre ogni input ad output seguendo una serie di **passi consentiti** dal modello computazionale.
- **La tesi di Turing-Church** (semplificata)
  ogni computazione che può essere realizzata da un qualsiasi **dispositivo fisico** può essere realizzata anche da un **algoritmo**; se il dispositivo fisico riesce a completare il compito in *n* passi, tale algoritmo potrà riuscirci in *m* passi, **con *m* che differisce da *n* di al massimo un polinomiale**.
- Alcuni algoritmi impiegano **troppo tempo** per fornire una soluzione
- Altri algoritmi possono **non terminare mai**

Macchine di Turing

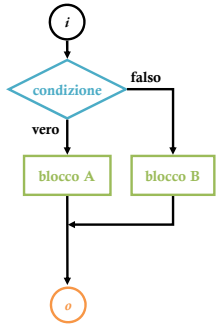
- nastro infinito diviso in celle
- alfabeto ***A*** (di almeno 2 elementi, ad esempio ***A*** ={0, 1})



- cursore ***C*** (che scorre a sinistra e a destra, che può leggere, cancellare e scrivere un carattere)
- insieme finito ***Q*** di stati (q_0, q_1 … q_n)
- input finito ***I*** costituito da caratteri in ***A***
- insieme finito ***S*** di stati della macchina descritte da quintuple del tipo $<q_i a b v q_j>$ dove q_i, $q_j \in Q$; $a,b \in A$; v = {destra, sinistra}

Flow charts

- Grafo orientato etichettato **costituito da:**
  - un **ingresso** (dove viene introdotto l'input *i*)
  - una o più **uscite** (da cui viene, eventualmente, recuperato un output *o*)
  - un insieme finito di **blocchi di istruzioni** tali che ogni istruzione è del tipo X = Y, X = X+1, X = X-1
  - un insieme finito (possibilmente nullo) di blocchi speciali, detti **condizioni** tali da presentare interrogazioni booleane del tipo X = Y?
  - un insieme finito di **connettori** che collegano i blocchi, tali che da ogni blocco esce 1 ed 1 sola freccia, mentre dal blocco condizionale escono 2 frecce



Modularità

- **Macchine di Turing** e **flow charts** sono equivalenti: esprimono la stessa classe di funzioni: le funzioni computabili.
- Entrambi i formalismi godono della proprietà della **composizionalità** (M1 • M2). Questo ci garantisce che un algoritmo può in realtà essere il risultato di una composizione di più macchine di Turing (o di più flow charts).
- Si dice "divide et impera" il paradigma di programmazione che suggerisce di **scomporre un problema nelle sue sottoparti** prima di iniziare a pensare agli algoritmi che lo risolvono.

Nozioni di computazione

- Per **computazione** si intende la **specificazione della relazione** tra un **input** ed un **output**
- tale relazione consiste in una serie di **stadi intermedi** in cui l'informazione di input può venire trasformata prima di raggiungere lo "status" di output
- **obiettivo della computazione**: ridurre ogni input ricevuto ad output significativo seguendo una serie di passi consentiti dal modello computazionale adottato
- **spazio del problema**: serie di **stati raggiungibili** applicando correttamente le regole ammissibili, dati:
  - uno **stato iniziale** del sistema
  - un preciso **input**

Nozioni di complessità

- La **complessità** di una computazione è direttamente proporzionale alla **quantità di risorse impiegate** per mappare un input in un output.
- Tali risorse sono solitamente definite su due dimensioni:
  - **tempo** (time complexity): passi elementari da compiere
  - **memoria** (space complexity): quantità di informazione massima richiesta ad ogni passo
- La complessità è **direttamente proporzionale** alla **dimensione del problema** (ad es. ordinare 1000 parole sarà più complesso che ordinarne 10);
- si può intuire che la complessità di una grammatica sarà direttamente proporzionale alla sua **forza generativa**.

Nozioni di complessità

- La **dimensione del problema** viene espressa in funzione dell'input da processare
- quindi l'**ordine di complessità** si potrà esprimere in funzione di tale input:

  $c \cdot n^2$ (problema con ordine di complessità temporale polinomiale)
  n = dimensione dell'input
  c = costante data (dipendente dal tipo di computazione)

  Si dirà che tale problema ha ordine di complessità pari a n^2 poiché la costante c sarà irrilevante sulla crescita della funzione per n che tende all'infinito. Tale ordine di complessità si definisce: $O(n^2)$.

Nozioni di complessità

- interesse nel **tasso di crescita della funzione** che esprime il mapping tra input e output in termini di dimensione dell'input
- per **problemi limitati** **temporalmente** e **spazialmente** (uso di risorse sicuramente finite) il calcolo della complessità è relativamente rilevante
- Al crescere dell'input a piacimento (***n* che tende all'infinito**), come nel caso di qualsiasi grammatica formale linguisticamente interessante, il **tasso di crescita della funzione** è cruciale per determinarne la **trattabilità**
- si dice che un problema è **trattabile** se esiste una procedura che fornisce una soluzione (positiva o negativa) in un **tempo finito**
- Un problema si dice **umanamente trattabile** se l'algoritmo fornisce una risposta in tempi accettabili

Nozioni di complessità

- Un problema con ordine di **complessità esponenziale** (es. $O(2^n)$) sarà difficilmente umanamente trattabile. Per avere un'idea intuitiva del tasso di crescita di alcune funzioni, ipotizzando di disporre di una macchina che gestisca **un milione di passi al secondo**, ecco il calcolo del tempo che occorrerebbe per risolvere alcune funzioni:

dimensione input → ↓ funzione	10	20	50	100
N^2	0,0001 secondo	0,0004 sec.	0,0025 sec.	0,01 sec
N^5	0,1 sec.	3,2 sec.	5 minuti e 2 sec.	2 ore e 8 min.
2^N	0,001 sec.	1 sec.	35 anni e 7 mesi	400 trilioni di secoli
$N!$	3,6 sec.	circa 771 secoli	un numero di secoli a 48 cifre	un numero di secoli a 148 cifre
N^N	2 ore e 8 minuti	più di 3 trilioni di anni	un numero di secoli a 75 cifre	un numero di secoli a 185 cifre

Nozioni di complessità

- La tesi di **Turing-Church** ci dice che ogni compito computazionale che può essere realizzato da un qualsiasi dispositivo fisico può essere realizzato anche da una **macchina di Turing**
- se il dispositivo fisico riesce a completare il compito in $F(n)$ passi, con n uguale alla dimensione dell'input, la macchina di Turing ci riuscirà in $G(n)$ passi, con F che differisce da G di al massimo un polinomiale
- Questo implica che se una macchina di Turing richiede un tempo esponenziale (cioè t^n) per risolvere un determinato compito, allora non ci sono speranze di trovare un dispositivo fisico qualsiasi che realizzi tale compito.
- Calcolare la **complessità delle proprietà linguistiche** che si intendono catturare è quindi indispensabile per poi scegliere il meccanismo formale che implementerà la grammatica in grado di esprimerle

Problemi classici e calcolo della complessità

- **3SAT problem** (variante del **problema del soddisfacimento, satisfability problem** o **SAT**)
  trovare, se esiste, una particolare **assegnazione di valori verofunzionali** alle lettere proposizionali di una formula booleana in modo che l'intera formula sia vera. La forma che la formula assume è la seguente:

  $(a \vee \neg b \vee c) \wedge (\neg a \vee b \vee \neg c) \wedge (a \vee b \vee c) \wedge \ldots$

- nel **peggiore dei casi**, si dovrebbero provare tutte le combinazioni di assegnazioni possibili, ovvero 2^N (dove **2** è il numero dei possibili valori dei singoli proposizionali, Vero e Falso, e **N** è il numero dei proposizionali **a**, **b**, **c**…).
- Il problema risulta quindi **temporalmente a crescita esponenziale**, ma mostra una caratteristica molto interessante: una volta proposta una soluzione, questa sarà facilissima da verificare! Un problema come **3SAT** si dice quindi **difficile da risolvere ma facile da verificare**.

Problemi classici e calcolo della complessità

- **QBF problem**
  (problema della **Quantificazione di Formule Booleane**)
  trovare un'assegnazione di valori per le variabili quantificate in una formula al fine di far risultare l'intera formula vera. La struttura della formula è la seguente:

  $Qx_1,Qx_2 \ldots Qx_n\ F(x_1,x_2 \ldots x_n)$
  (con Q uguale a $\forall$ oppure $\exists$)

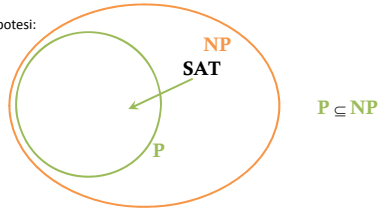
- Il problema è **difficile** da risolvere, **come 3SAT**, ma anche **difficile da verificare**: in effetti, ogni proposizionale di un problema di 3SAT potrebbe essere considerato come **quantificato esistenzialmente**
- Il fatto di introdurre su alcuni proposizionali una **quantificazione universale**, comporta che ogni assegnazione di valore va verificata, quindi la verifica della soluzione avrà lo stesso ordine di complessità del problema 3SAT

Problemi classici e calcolo della complessità

- Ipotizzando che un computer **riesca a risolvere effettivamente** un problema come il 3SAT, tale computer dovrà utilizzare un **algoritmo al peggio polinomiale**.
- vista la natura del problema, tale algoritmo dovrà essere **necessariamente non-deterministico** (quindi implementabile in quella che viene chiamata una **Macchina di Turing** (**MT**) **non-deterministica**).
- Questi problemi vengono definiti di tipo **NP**:
  Non-deterministic Polynomial time
- I problemi di ordine **P** sono invece **deterministici e polinomiali** in funzione del tempo. La classe dei problemi in **P** è probabilmente inclusa in quella dei problemi in **NP**.
- Per quanto riguarda i problemi (come 3SAT) in NP non esiste prova concreta della loro **riducibilità** a problemi di ordine P.

Problemi classici e calcolo della complessità

- ipotesi:



$P \subseteq NP$

- problemi come il **SAT** si dice **NP-hard** (stesse difficoltà dei problemi della classe NP). Ipotizzando un ipotetico dispositivo "preveggente" in grado di risolvere il problema con una funzione di **ordine polinomiale** si dice che il problema è **NP-completo**.
- Un chiaro segnale di NP-completezza è il fatto che tale problema sia **difficile da risolvere ma facile da verificare** una volta proposta una soluzione.

Cos'è il parsing

- data una grammatica ***G*** ed un input ***i*** fare parsing significa applicare una funzione ***p(G, i)*** in grado di:
  1. accettare/rifiutare ***i***
  2. assegnare ad ***i*** una struttura descrittiva adeguata (es. Indicatore Sintagmatico)

  (ad es. il **parsing sintattico** riconosce un input grammaticale e assegna ad esso una struttura sintattica)

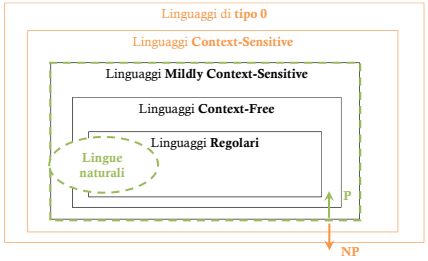
Universal Recognition Problem (URP) e riduzione

- Per scoprire la classe di complessità di un problema si ricorre alla **riduzione**: si prende cioè un problema di cui si conosce già la complessità, si trova un **mapping efficiente** che trasformi ogni istanza del problema noto in un istanza del nuovo problema e che **preservi i risultati richiesti**.
- È questo il caso dell'**Universal Recognition Problem** (**URP**) che viene così formulato:
  Data una grammatica G (in qualsiasi framework grammaticale) e una stringa x, x appartiene al linguaggio generabile da G?

Universal Recognition Problem (URP) e riduzione

- L'**URP** è un problema di parsing generalizzato che può essere ridotto ad un problema come **SAT**
- intuizione di fondo: per una stringa ***x***, come un proposizionale ***a*** in una formula di **SAT**, può esistere un'assegnazione **ambigua di valori** (ad esempio la stringa "*vecchia*" può essere sia *nome* che *aggettivo*, mentre il proposizionale *a* può essere *vero* o *falso*). Inoltre si può parlare di qualcosa come la verifica di accordo sia per la stringa ***x*** (tutte le occorrenze di x devono accordarsi / essere compatibili in senso linguistico) che accordo tra i proposizionali in una formula SAT (inteso come consistenza dell'assegnazione di valori). Si deduce quindi che l'URP è almeno complesso quanto il SAT e che è **NP-completo**

Gerarchia di Chomsky e ordini di complessità



Complessità psicolinguistica

- **complessità = difficoltà** incontrata da un parlante nel comprendere/produrre una frase
  (es. una frase con tre incassamenti centrali è più complessa di una con due, come una che presenta tre dipendenze cross-seriali è più complessa di una con solo due. Mentre non c'è una grande differenza di complessità in frasi con due, tre o quattro relative non incassate: "e venne il cane che morse il gatto che scacciò il topo che mio padre alla fiera comprò")
- **ipotesi 1:** complessità formale = complessità psicolinguistica
- **ipotesi 2:** memoria di processamento limitata
  - da una parte la **capacità di tale buffer** potrebbe non essere sufficiente a contenere più di N strutture;
  - dall'altra inserire in questo deposito più di un certo numero di sintagmi (simili?) incompleti potrebbe causare **confusione**.

Complessità psicolinguistica

- **ipotesi 1**
  - il processamento di **strutture non context-free** causa le difficoltà maggiori (Pullum e Gazdar 1982)
- **ipotesi 2**
  - **memoria limitata** (**limited-size stack**, Yngve 1960)
    il processamento linguistico effettivo è basato su una memoria limitata (limited-size stack) in cui vengono inseriti i risultati parziali delle regole applicate finché i sintagmi non sono completati. Più frasi incomplete vengono inserite nello stack, più difficile da processare sarà il periodo.
  - **Syntactic Prediction Locality Theory** (SPLT, Gibson 1998) **carico mnemonico totale** richiesto da una struttura sintattica = somma dei carichi mnemonici richiesti dalle singole parole **necessarie per completare tale struttura**
    1. DP richiedono VP (in SVO) DP DP DP VP VP VP... è più complessa di DP VP
    2. l'introduzione di un **pronome** che si riferisce ad un'**entità già introdotta** nel discorso è meno complessa dell'introduzione di un **nuovo referente** che dovrà in qualche modo essere integrato nella struttura non ancora completata.

Grammatiche e Parsing

- le grammatiche formali sono (in genere) **dispositivi dichiarativi** che non specificano come un preciso input debba essere analizzato e come una struttura ad albero possa essere costruita.
- Fenomeni come il **non determinismo** (la possibilità di scegliere tra più alternative strutturali equivalenti dal punto di vista dell'informazione posseduta al momento della scelta) pongono seri problemi da questo punto di vista, poiché potrebbe facilmente accadere che la scelta strutturale fatta possa rivelarsi sbagliata proseguendo nel parsing. Si richiedono quindi strategie chiare per gestire efficientemente queste situazioni.

Spazio del problema e strategie di ricerca

- data una **frase** ed una **grammatica** il compito del parser è dire se la frase può essere riconosciuta dalla grammatica (**URP**, **Universal Recognition Problem**) e, in caso affermativo, assegnare alla frase un adeguato Indicatore Sintagmatico
- Lo **spazio del problema** è l'insieme di tutti gli alberi e sottoalberi possibili che possono essere generati applicando adeguatamente le regole grammaticali

Spazio del problema e strategie di ricerca

- frase:
  la vecchia legge la regola
- grammatica:

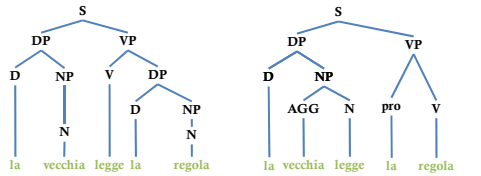
→ (non terminali)	→ (terminali)
S → DP VP	pro → la
VP → V DP	D → la
VP → pro V	AGG → vecchia
DP → D NP	N → vecchia
NP → (AGG) N;	N → legge
	N→ regola
	V → legge
	V→ regola

Spazio del problema e strategie di ricerca

- Due vincoli fondamentali:
  1. le regole grammaticali che predicono come da un **nodo radice S** ci siano solo alcune vie di scomposizione possibili per ottenere i nodi terminali;
  2. le parole della frase, che ricordano come la (s)composizione di S debba **terminare**
- nel primo caso, se si decide di partire dal **nodo radice S** per generare la struttura adatta alla frase data, si parla di strategia di ricerca **Top-Down o goal-driven**
- nel secondo caso, partendo quindi dalle singole parole e cercando di combinarle in strutture compatibili per giungere ad S, la strategia di ricerca si dice **Bottom-Up, o data-driven**.

Algoritmo di parsing Top-Down

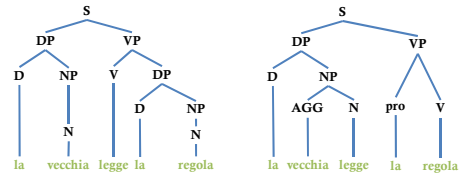
- un semplice algoritmo **top-down** inizia esplorando tutte le possibilità di riscrittura del nodo S che la grammatica offre (assumiamo che ogni albero generato possa poi essere esplorato parallelamente).



- "la regola regola la regola", "la legge legge la vecchia legge"… saranno scartate solo all'ultimo livello, in fase di mappatura con l'input realmente fornito.

Algoritmo di parsing Bottom-Up

- primo vero algoritmo associato storicamente all'operazione del parsing (Yngve 55) e probabilmente il più usato (ad esempio nei parsers per i linguaggi di programmazione). La logica che guida la ricerca è quella di partire dagli elementi della frase, cioè dai nodi terminali, e costruire da qui, applicando tutte le regole possibili, una struttura che termini con S:



Quale è la strategia migliore?

- La strategia Top-Down non perde tempo cercando di costruire alberi non consistenti con la grammatica, ma genererà **tutte le alternative possibili prescindendo dall'input**.
- La strategia Bottom-Up, sarà **consistente** (almeno localmente) **con l'input** fornito, ma potrà generare per molti livelli **sotto-alberi** che alla fine si riveleranno **non combinabili per ottenere S**.
- Anche se talvolta si possono ottenere risultati comparabili, nella stragrande maggioranza dei casi si deve tener conto di due fondamentali caratteristiche dello spazio del problema:
  a. si parte sempre dalla parte dell'albero in cui si dispone dell'**informazione più precisa**
  b. si risale l'albero nella direzione in cui il **fattore di ramificazione è minore**.

Il costo del parallelismo

- Finora abbiamo irrealisticamente assunto che le varie regole potessero essere applicate in parallelo (cioè che ogni ramificazione dello spazio del problema potesse essere espansa in contemporanea). In realtà la quantità di memoria richiesta per registrare gli stati dello spazio di un problema creato da un parser che usa una grammatica credibile sarebbe enorme.
- Di solito la soluzione "brute-force" avviene attraverso l'esplorazione per **ampiezza** (**breadth-first**) o per **profondità** (**deep-first**): con quest'ultimo metodo si esplora lo spazio del problema **incrementalmente**, espandendo fino ai nodi terminali, un nodo alla volta, solitamente **da sinistra a destra** (**top-down**, **depth-first**, **left-to-right parser**). Ogni volta che l'operazione fallisce, si sceglie un nuovo percorso iniziando dalla ramificazione più recente.
- Questo tipo di ricerche sono comunque cieche, nel senso che **non esistono euristiche** che suggeriscono, di fronte ad un'alternativa, la scelta migliore e soprattutto non si rendono conto dell'errore finché l'espansione non è stata completamente effettuata.

Algoritmo di parsing LEFT CORNER

- **idea di base**
  combinare una strategia Top-Down per la generazione di strutture filtrando le soluzioni con considerazioni di natura Bottom-Up.
- **regola dell'angolo sinistro** (**left-corner**)
  ogni categoria alla fine verrà riscritta come una serie di parole linearmente ordinate e conoscere la prima parola della serie aiuterà a prevedere la categoria o almeno ad escludere soluzioni inconsistenti con questa parola
  Formalmente si può dire che B è l'angolo sinistro della categoria A sse $A \rightarrow^* B \rightarrow \alpha$.
- Da tale strategia si ottengono i risultati migliori se preliminarmente viene compilata una tabella che direttamente associa alle categoria un loro possibile **left-corner**:

categoria	**S**	**DP**	**VP**
left-corner	D, $N_{proprio}$, V	D	aux, V

Alcuni problemi irrisolti

- **ricorsività a sinistra**
  una grammatica si dice ricorsiva a sinistra se ammette una regola del tipo:
  $A \rightarrow^* A\alpha$ (es. DP → DP PP)
- **ambiguità**
  a vari livelli si possono trovare molteplici strutture che soddisfano i requisiti dell'input e delle regole grammaticali:
  - **attaccamento dei PP** (ho visto l'uomo con il cannocchiale)
  - **coordinazione** (papaveri e paperi rossi)

  È stato calcolato (Church e Patil 82) che il numero di strutture possibili in corrispondenza di PP cresce **esponenzialmente** con il numero di PP introdotti (se con 3 PP si hanno fino a 5 NP possibili, con 6 PP si arriva a 469 NP possibili… con 8 a 4867).

Alcuni problemi irrisolti

b. **ambiguità** (...continua)

Da una parte il Top-Down parser, se perseguisse il suo compito efficientemente, si dovrebbe **fermare appena recupera una struttura possibile** (nel caso di 3 PP ha una possibilità su 5 di individuare la soluzione più probabile), dall'altra sarebbe **imprudente lasciarlo considerare tutte le strutture generabili**.

L'ambiguità (quella lessicale soprattutto: "vecchia" è Aggettivo o Nome?) causa facilmente errori. La soluzione **deep-first, left-to-right** potrebbe rivelarsi in tal senso **estremamente inefficiente in caso di ambiguità iniziali**.

c. inefficienza nel **ripetere l'analisi dei sottoalberi**
il backtracking causato da un errore di parsing, può facilmente provocare il disfacimento di una porzione di albero che in realtà sarebbe costruita bene e che verrà ricostruita tale e quale al prossimo tentativo, come nel caso seguente:

un volo da Roma per Milano su un 747

Programmazione Dinamica

- Per **programmazione dinamica** (**dynamic programming**) si intende un approccio che fa uso sistematico di tavole per la registrazione di soluzioni ai sottoproblemi incontrati.
- Una volta risolti tutti i **sottoproblemi** (nel caso del parsing sintattico, tutti i sottoalberi), la soluzione al problema globale consiste nel **combinare adeguatamente le singole soluzioni trovate**. Tale metodo risulta essere molto più efficiente degli approcci precedenti e risolve, in linea di principio, almeno il problema dell'inefficienza del ripetere l'analisi corretta dei sottoalberi.

Programmazione Dinamica: l'algoritmo di Earley

- L'**algoritmo di Earley** (Earley 1970) è un esempio classico di programmazione dinamica che usa un approccio top-down parallelo.
- La complessità del problema, in linea teorica sempre **NP-hard**, viene **ridotta a polinomiale** (nel peggiore dei casi abbiamo **$O(n^3)$** eliminando le soluzioni ripetitive dei sottoproblemi dovuti al backtracking da una struttura scorretta.
- L'algoritmo è caratterizzato da **un unico esame**, **da sinistra a destra**, dell'input che permette di riempire una struttura dati chiamata **grafo** (**chart**) che avrà *n*+1 entrate, con *n* uguale al numero delle parole in input.
- Le entrate corrispondono alle **posizioni tra le parole** dell'input (comprese la posizione iniziale, prima della prima parola, e quella finale dopo l'ultima). Per ogni posizione il chart conterrà una lista esaustiva delle strutture fin lì generate ed utilizzabili per ulteriori elaborazioni che cercheranno di integrare posizioni successive. La rappresentazione compatta di questa informazione permette di sfruttare efficientemente, **senza doverla ricomputare in caso di backtrack**, ogni struttura ben formata associata ad un input parziale.

Programmazione Dinamica: l'algoritmo di Earley

- In concreto quindi ogni entrata del **chart** avrà tre tipi di informazioni:
  - un **sottoalbero** corrispondente ad una **singola regola grammaticale**
  - l'**informazione del progresso fatto** al fine di completare il sottoalbero in questione (solitamente si usa un punto • nella parte destra della regola per indicare la posizione a cui si è giunti, la "**regola puntata**" si dice quindi **dotted rule**)
  - la **posizione** del sottoalbero **in relazione all'input** (definito da due numeri indicanti la posizione di inizio della regola e quella dove si trova il punto; es. DP → D • NP [0,1])

Programmazione Dinamica: l'algoritmo di Earley

- L'algoritmo procede con tre operazioni fondamentali:
  - **Previsione (Predictor)**
    crea nuovi stati nell'entrata corrente del chart, rappresentando le aspettative top-down della grammatica; verrà quindi creato un numero di stati uguale alle possibilità di espansione di ogni nodo non terminale nella grammatica.
    es. S → • DP VP [0,0] DP → • D NP [0,0]
  - **Scansione (Scanner)**
    verifica se nell'input esiste, nella posizione adeguata, una parola la cui categoria combacia con quella prevista dallo stato a cui la regola si trova. Se il confronto è positivo, la scansione produce un nuovo stato in cui l'indice di posizione viene spostato dopo la parola riconosciuta. Tale stato verrà aggiunto all'entrata successiva del chart.
    es. DP → • D NP [0,0] (scansione articolo) se D → articolo: DP → D • NP [0,1] (entrata successiva del chart)

Programmazione Dinamica: l'algoritmo di Earley

- L'algoritmo procede con tre operazioni fondamentali:
  - **Completamento (Completer)**
    quando l'indicatore di posizione raggiunge l'estrema destra della regola questa procedura riconosce che un sintagma significativo è stato riconosciuto e verifica se questo avvenuto riconoscimento è utile per completare qualche altra regola rimasta in attesa di quella categoria: ad esempio se nella situazione precedente viene completata positivamente la regola NP → AGG N • [1,3], l'operazione di completamento cercherà tutti gli stati che terminavano nella posizione 1 e che aspettavano un NP per completare la regola. Ma questo era proprio il caso di DP → D • NP [0,1]; quindi il riconoscimento del NP aggiunge anche (all'entrata corrente) lo stato DP → D NP • [0,3];

Programmazione Dinamica: l'algoritmo di Earley

funzione Earley-Parser(*stringa_parole*, *grammatica*) **restituisce** *grafo*
  metti_in_coda(($\gamma \rightarrow \bullet S$, [0,0]), *grafo*[0])
  per ogni *i* **tale che** (*i* va da 0 a lunghezza(*stringa_parole*)) **esegui**
    per ogni *stato* **tale che** (*stato* è in *grafo*[*i*]) **esegui**
      se (incompleto(*stato*) **e non** parte_della_frase
        (prossima_categoria(*stato*)))
      allora esegui Previsione (*stato*)
      altrimenti_se (incompleto(*stato*) **e**
        parte_della_frase(prossima_categoria(*stato*)))
      allora esegui Scansione (*stato*)
      altrimenti esegui Completamento (*stato*)
    fine
  fine
restituisci *grafo*

Programmazione Dinamica: l'algoritmo di Earley

procedura Previsione ($A \rightarrow \alpha \bullet B\beta$, $[i,j]$)
  per ogni ($B \rightarrow \gamma$) **tale che** (Regole_della_Grammatica(*B*, *grammatica*)) **esegui**
    metti_in_coda(($B \rightarrow \bullet\gamma$, $[j,j]$), *grafo*[*j*])
fine
procedura Scansione ($A \rightarrow \alpha \bullet B\beta$, $[i,j]$)
  se (*B* è parte_della_frase(*parola*[*j*])) **allora esegui**
    metti_in_coda(($B \rightarrow$ *parola*[*j*], $[j,j+1]$), *grafo*[*j*+1])
fine
procedura Completamento ($B \rightarrow \gamma \bullet$, $[j,k]$)
  per ogni ($A \rightarrow \alpha \bullet B\beta$, $[i,j]$) **tale che** (($A \rightarrow \alpha \bullet B\beta$, $[i,j]$) è nel *grafo*[*j*]) **esegui**
    metti_in_coda(($A \rightarrow \alpha B \bullet \beta$, $[j,k]$), *grafo*[*k*])
fine
procedura metti_in_coda (*stato*, *entrata_grafo*)
  se (*stato* non è già nell'*entrata_grafo*) **allora esegui**
    inserisci(*stato*, *entrata_grafo*)
fine

Alcune considerazioni su efficienza e prestazioni

- Mentre una grammatica può prescindere da **limiti spazio/temporali** e concentrarsi solo sull'appropriatezza del modello descrittivo fornito, l'architettura del **parser** deve invece crucialmente tener conto di questi vincoli. Questa è una delle principali ragioni per cui **ad una determinata grammatica non corrisponde uno ed un solo parser**.
  La scelta del "più adatto" spesso richiede valutazioni differenti
- Si può parlare di **token trasparency** (Miller e Chomsky 63) o di **isomorfismo stretto** (ipotesi nulla) quando il parser rispecchia esattamente i passi derivazionali ipotizzati nella grammatica.
  (Slobin, 1966, mostra in realtà che, contrariamente a quanto si potrebbe pensare, certe costruzioni passive richiedono meno tempo per essere processate rispetto a quelle attive)
- Si parla invece di **type trasparency** (Bresnan 78) quando alle varie proprietà grammaticali sono associate differenti regole nel modello computazionale che globalmente riproducono la struttura dell'ipotetico parser umano.

Alcune considerazioni su efficienza e prestazioni

- Berwick e Weinberg (83, 84) introducono il concetto di **covering grammars**: un modello della grammatica proposto per il parsing può comunque essere una realizzazione della competenza linguistica purché questo riesca a modellare lo stesso linguaggio per cui la competenza linguistica umana è teoricamente adeguata. Questo modello non sarà psicologicamente plausibile, ma potrà essere ottimale in senso computazionale.

III puntata

L'APPROCCIO TOP-DOWN ALLA COMPUTAZIONE LINGUISTICA

Grammatiche Minimaliste

- Stabler's (1997) formalization of a Minimalist Grammar, **MG** (Chomsky 1995) as a 4-tuple (V, Cat, Lex, F) such that:

  V is a finite set of non-syntactic features, (P ∪ I) where
    P are phonetic features and I are semantic ones;
  Cat is a finite set of syntactic features,
    Cat = (base ∪ select ∪ licensors ∪ licensees) where
    base are standard categories {comp, tense, verb, noun ...},
    select specify a selection requirement {=x | x base}
    licensees force phrasal movement {–wh, –case ...},
    licensors satisfy licensee requirements {+wh, +case ...}
  Lex is a finite set of expressions built from V and Cat (the lexicon);
  F is a set of two partial functions from tuples of expressions to expressions : {merge, move};

Grammatiche Minimaliste

V = *P* = {/what/, /did/, /you/, /see/},
I = {[what], [did], [you], [see]}

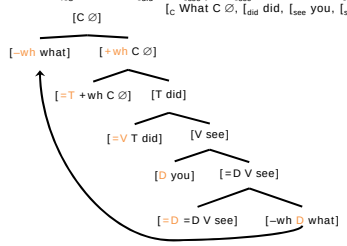
Cat = *base* = {D, N, V, T, C}
select = {=D, =N, =V, =T, =C}
licensors = {+wh}
licensees = {–wh}

Lex = { [–wh D what], [=V T did], [D you], [=D =D V see], [=T +wh C ∅] }

F = {*merge*, *move*} such that:
merge ([=F X] , [F Y]) = [$_X$ X Y]
("simple merge" on the right, "complex merge" on the left)
move ([+g X] , [W [–g Y]]) = [[$_X$ Y X] W, t_Y]

Grammatiche Minimaliste

1. merge ([=D =D V see], [-wh D what]) → [$_{see}$ =D V see, –wh what]
2. merge ([D you], [=D V see, -wh what]) → [$_{see}$ you, [$_{see}$ V see, –wh what]]
3. merge ([=V T did], [$_{see}$ you, [$_{see}$ V see, -wh what]]) → ([$_{did}$ T did, [$_{see}$ you, [$_{see}$ see, –wh what]]]
4. merge ([=T +wh C ∅], [$_{did}$ T did, [$_{see}$ you, [$_{see}$ see, –wh what]]]) → ([$_C$ +wh C ∅, [$_{did}$ did, [$_{see}$ you, [$_{see}$ see, –wh what]]]])
5. move ([$_C$ +wh C ∅, [$_{did}$ did, [$_{see}$ you, [$_{see}$ see, –wh what]]]]) → [$_C$ What C ∅, [$_{did}$ did, [$_{see}$ you, [$_{see}$ see, t_{what}]]]]



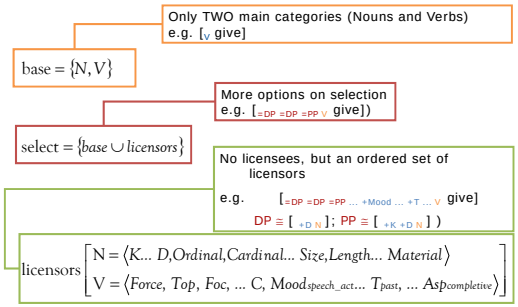
Phase-based MGs (PMGs, Chesi 2004-07)

1. Feature Structures
(lexicon + parameterization)

2. Structure Building Operations
(merge, move, phase)

3. Universals
(structural constraints + economy conditions)

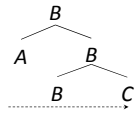
Phase-based MGs (PMGs, Chesi 2004-07)



PMGs Universals: Linearization Principle

⊙ Linearization Principle *(inspired by LCA, Kayne 1994)*

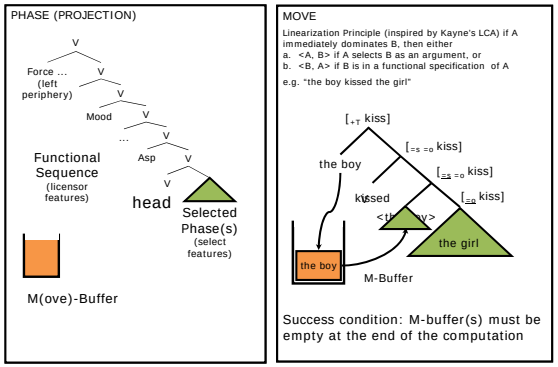
if A dominates B, then either
a. A precedes B if B is a complement of A (that is, A selects B), or
b. B precedes A if B is in a functional projection of A



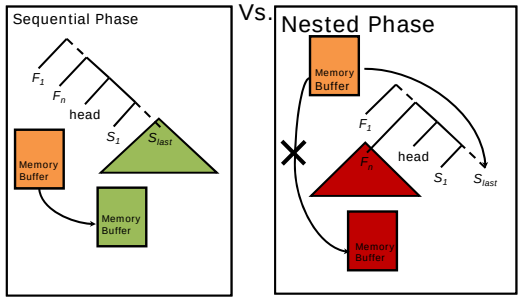
dominance:
B dominates A
B dominates C

precedence:
<A, B, C>

Two more Structure Building Operations

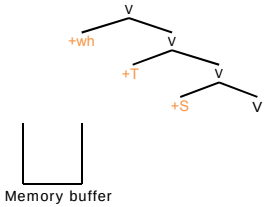


"Internal" Merge



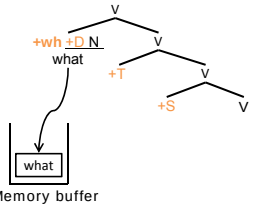
Formal background: Sample derivation of wh-question

- *(default)* Expand(Lex: CP_{wh} = **<+wh +T +S V>**)



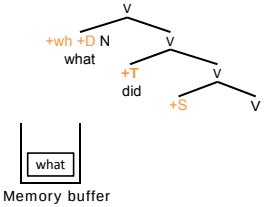
Formal background: Sample derivation of wh-question

- *(default)* Expand(Lex: CP_{wh} = **<+wh +T +S V>**)
- Insert(Lex: [+wh +D N what])

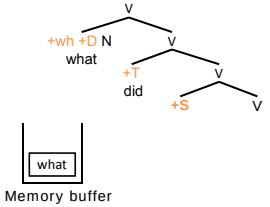


Formal background: Sample derivation of wh-question

- Insert(Lex: [**+T** did])

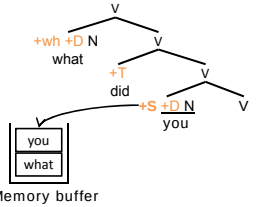


Formal background: Sample derivation of wh-question



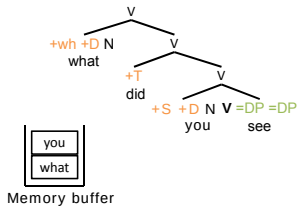
Formal background: Sample derivation of wh-question

- Insert(Lex: [+S **+D N** you])



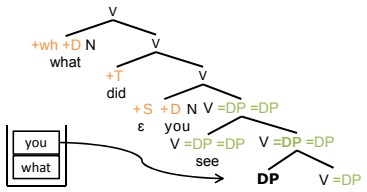
Formal background:
Sample derivation of wh-question

- Insert(Lex: [**V** =DP =DP see)



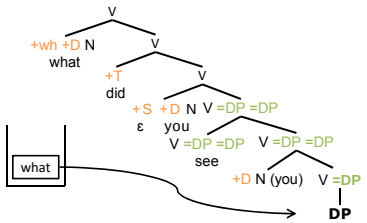
Formal background:
Sample derivation of wh-question

- Expand(V **=DP**)
- Move([+D N you])



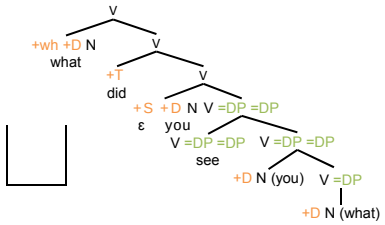
Formal background:
Sample derivation of wh-question

- Expand(V **=DP**)
- Move([+D N what])



Formal background:
Sample derivation of wh-question

- Expand(V **=DP**)
- Move([+D N what])

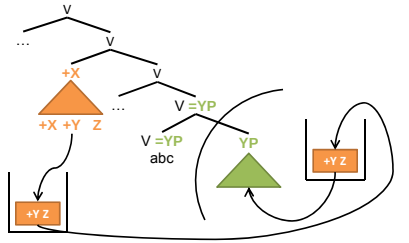


Locality in a Top-Down Left-Right derivation

1. Every computation is a **top-down process** divided in **phases**.
2. A phase gets closed when all the functional and selectional specifications of its head are satisfied. Any internal constituent will be a computationally **nested phase**.
3. The **Move operation** stores an **unselected** element in the local M-buffer of the current phase, and discharges it in a selected position if possible; if not, when the phase is closed the content of the M-buffer is inherited by the M-buffer of the **next selected phase** (the **sequential phase**, Chesi 2004).
4. A **nested phase** cannot inherit the content of the M-buffer of the containing phase (Chesi 2004, Bianchi & Chesi 2005)

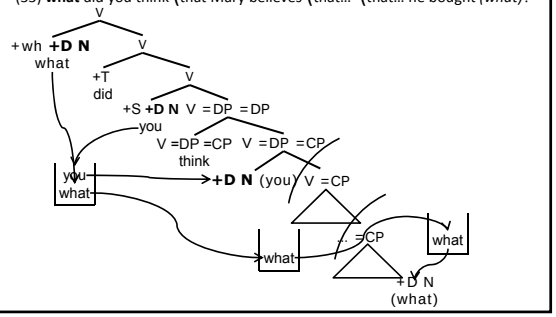
Locality in a Top-Down Left-Right derivation

- **Nested** vs. **selected** **phases**



Successive Cyclic A'-movement

(33) **what** did you think ⟨that Mary believes ⟨that… ⟨that… he bought *(what)*?



$\{a^n b^n : n \geq 0\} \notin strings(\text{PMG}_0)$

- **PMG_0:** ***F = <Merge, PhaseProjection, Move>***
- Using the ***Pumping Lemma*** for RGs (Partee and al. 1993) we can show that it is in fact impossible to "pump" (i.e. *selecting* recursively: [=X X ***x***]) any portion of an $a^n b^n$ expression so as to keep the number of *a*s and *b*s equal:

1. [=X X ***ab***], [X ***ab***]
   would produce $(ab)^n$;

1. [=X X ***a***], [X ***a***], [=Y Y ***b***], [Y ***b***], [=X =Y R ***ε***]
   would produce the same number of *a*s and *b*s only occasionally.

- This is sufficient to guarantee that **PMG_0**s (with local ***PhaseProjection***) are as powerful as **RGs**.

$\{a^n b^n : n \geq 0\} = strings(\text{PMG}_1)$

PMG_1:

V = *P* = {*a*, *b*, ε};

Cat = *base* ∪ *licensors* ∪ *select* ∪ *interveners*:
base= {*R(ecursion)*, *A*, *B*, *a*, *b*};
licensors = {<+*A*, +*B*>};
select = {=*S*, =*R*, =*A*, =*B*, =*a*, =*b*};

Lex = [+A ***a***], [b ***b***],
[+B_0 =b C ***ε***], [+B =b =C C ***ε***],
[+A +B R ***ε***], [=C R ***ε***], [+A +B_0 =R S ***ε***]

F = <*Merge*, *Move*, *PhaseProjection*>

$\{a^n b^n c^n d^n e^n : n \geq 0\} = strings(\text{PMG}_{1a})$

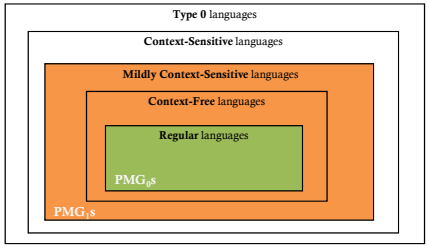
PMG_{1a}:

V = *P* = {*a*, *b*, *c*, *d*, *e*, ε};

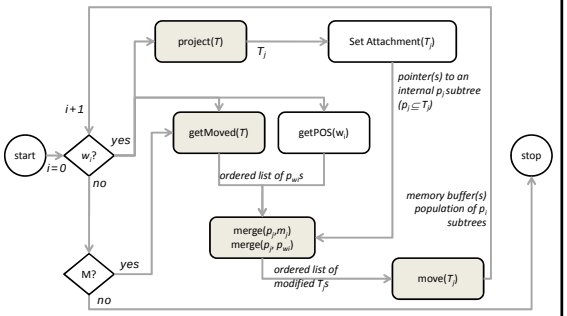
Cat = *phase* ∪ *functional* ∪ *select* ∪ *interveners*:
phase = {S(tart), R(ecursion), A, B, C, D, E, a, b, c, d, e};
functional = {<+A, +B, +C, +D, +E>};
select = {=[+A +B +C +D +E R], =A, =B, =C, =D, =E,
=a, =b, =c, =d, =e};
interveners = {/*A*, /*B*, /*C*, /*D*, /*E*};

Lex = [a *a*], [b *b*], [=a A_0 ε], [=b B_0 ε], [=a =A A ε], [=b =B B ε],
[c *c*], [d *d*], [=c C_0 ε], [=d D_0 ε], [=c =C C ε], [=d =D D ε],
[e *e*], [=e E_0 ε], [=e =E E ε], [+A_0 +B_0 +C_0 +D_0 +E_0 =[+A +B +C +D +E R] S ε],
[+A +B +C +D +E =[+A +B +C +D +E R] R ε], [=A =B =C =D =E R ε]

How powerful are PMGs?



Parsing Algorithm with PMGs (PMG-pa)



Parsing States: CFG-Earley Vs. PMG-pa

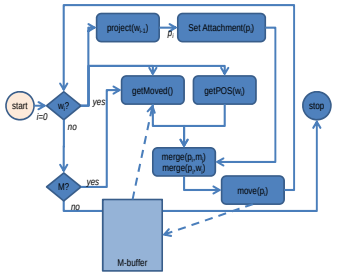
CFG-Earley	PMG-pa
Input position (e.g. the man • runs)	Input position (the man • runs)
Grammar rule (e.g. S → NP VP)	Phase inspected (e.g. Verbal)
Dot-position (e.g. S → NP • VP)	Constituent completion status (is the phase headed? yes are all thematic requirements satisfied? yes are non-thematic dependencies licensed? yes are non-thematic dependencies unique? yes further non-thematic dependencies available? yes status: potentially complete)
Leftmost edge of the substring this rule generates (e.g. the man runs)	Constituent prefix (e.g. the man runs)
	Memory buffer status (e.g. one N(ominal), determined, potentially complete phase)

Parsing Operations: CFG-Earley Vs. PMG-pa

CFG-Earley	PMG-pa
Predict top-down expansion of non-terminal rules in the grammar; Result: new rules (NP, VP, PP, AP…) added	Phase Projection new constituents insertion based on selection features in the lexicon; Result: Rooted Trees (RTrees) decorated with empty NP, VP or AP
Scan bottom-up inspection of the lexicon given a word; Result: PoS list to be integrated in the rules	Move / Lexical insertion unselected lexicalized sub-trees (moved LTrees) are available in this list, plus lexicalized sub-trees projected from a Top-Down inspection of the processed word-token; Result: ordered list (the pending list) of lexicalized sub-trees (LTrees) to be integrated in the structure
Complete top-down expansion of non-terminal rules in the grammar	Merge unification algorithm among pending rooted structures (RTrees) and lexicalized sub-trees in the pending list (LTrees)

Getting asymmetries with PMG-pa

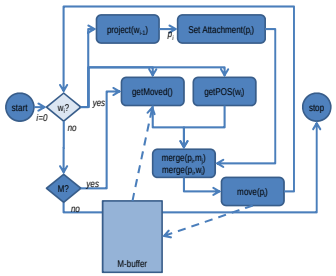
Subject relatives in head initial languages
a. The reporter who attacked the senator admitted the error.
b. The reporter who the senator attacked admitted the error.



Getting asymmetries with PMG-pa

Subject relatives in head initial languages
a. The reporter who attacked the senator admitted the error.
b. The reporter who the senator attacked admitted the error.

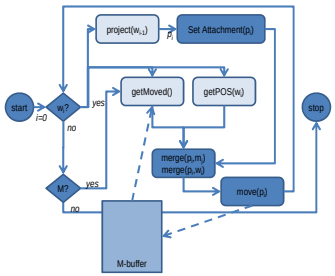
1. w_0 = "the"



Getting asymmetries with PMG-pa

Subject relatives in head initial languages
a. The reporter who attacked the senator admitted the error.
b. The reporter who the senator attacked admitted the error.

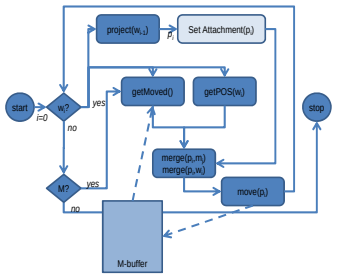
1. w_0 = "the"
2. project(*default*): [$_{VP}$], [$_{NP}$]
   getMoved(): nothing
   getPOS(*the*): [$_{NP\ +D}$ *the*]



Getting asymmetries with PMG-pa

Subject relatives in head initial languages
a. The reporter who attacked the senator admitted the error.
b. The reporter who the senator attacked admitted the error.

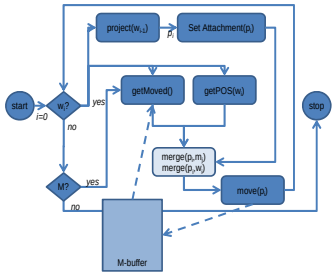
1. w_0 = "the"
2. project(*default*): [$_{VP}$], [$_{NP}$]
   getMoved(): nothing
   getPOS(*the*): [$_{NP\ +D}$ *the*]
3. setAttachment([$_{VP}$]): [$_{VP}$]
   setAttachment([$_{NP}$]): [$_{NP}$]



Getting asymmetries with PMG-pa

Subject relatives in head initial languages
a. The reporter who attacked the senator admitted the error.
b. The reporter who the senator attacked admitted the error.

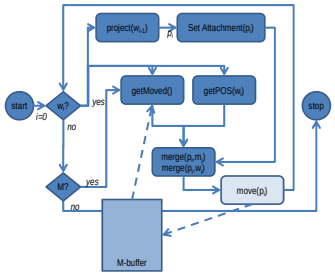
1. w_0 = "the"
2. project(*default*): $[_{VP}\ ]$, $[_{NP}\ ]$
   getMoved(): nothing
   getPOS(*the*): $[_{NP\ +D}\ the]$
3. setAttachment($[_{VP}\ ]$): $[_{VP}\ ]$
   setAttachment($[_{NP}\ ]$): $[_{NP}\ ]$
4. merge($[_{VP}\ ]$, $[_{NP\ +D}\ the]$): $[_{VP}\ [_{NP\ +D}\ the]]$
   merge($[_{NP}\ ]$, $[_{NP\ +D}\ the]$): $[_{NP\ +D}\ the]$



Getting asymmetries with PMG-pa

Subject relatives in head initial languages
a. The reporter who attacked the senator admitted the error.
b. The reporter who the senator attacked admitted the error.

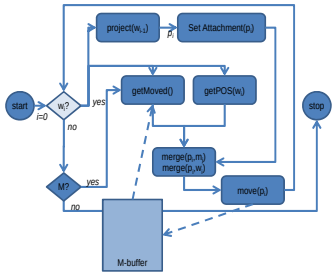
1. w_0 = "the"
2. project(*default*): $[_{VP}\ ]$, $[_{NP}\ ]$
   getMoved(): nothing
   getPOS(*the*): $[_{NP\ +D}\ the]$
3. setAttachment($[_{VP}\ ]$): $[_{VP}\ ]$
   setAttachment($[_{NP}\ ]$): $[_{NP}\ ]$
4. merge($[_{VP}\ ]$, $[_{NP\ +D}\ the]$): $[_{VP}\ [_{NP\ +D}\ the]]$
   merge($[_{NP}\ ]$, $[_{NP\ +D}\ the]$): $[_{NP\ +D}\ the]$
5. move($[_{VP}\ [_{NP\ +D}\ the]]$): nothing
   move($[_{NP\ +D}\ the]$): nothing



Getting asymmetries with PMG-pa

Subject relatives in head initial languages
a. The reporter who attacked the senator admitted the error.
b. The reporter who the senator attacked admitted the error.

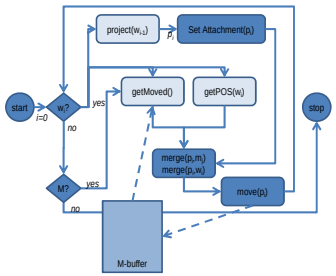
1. w_1 = "reporter"



Getting asymmetries with PMG-pa

Subject relatives in head initial languages
a. The reporter who attacked the senator admitted the error.
b. The reporter who the senator attacked admitted the error.

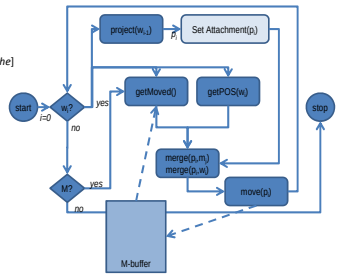
1. w_1 = "reporter"
2. project(*the*): nothing
3. getMoved(): nothing;
   getPOS(*reporter*): $[_{NP\ N}\ reporter]$



Getting asymmetries with PMG-pa

Subject relatives in head initial languages
a. The reporter who attacked the senator admitted the error.
b. The reporter who the senator attacked admitted the error.

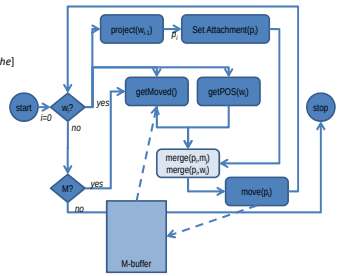
1. w_1 = "reporter"
2. project(*the*): nothing
3. getMoved(): nothing;
   getPOS(*reporter*): $[_{NP\ N}\ reporter]$
4. setAttachment($[_{VP}\ [_{NP\ +D}\ the]]$): $[_{NP\ +D}\ the]$
   setAttachment($[_{NP\ +D}\ the]$): $[_{NP\ +D}\ the]$



Getting asymmetries with PMG-pa

Subject relatives in head initial languages
a. The reporter who attacked the senator admitted the error.
b. The reporter who the senator attacked admitted the error.

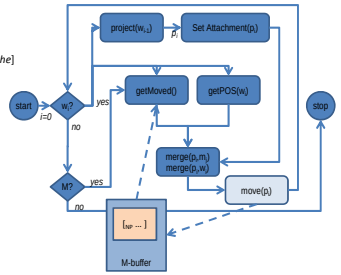
1. w_1 = "reporter"
2. project(*the*): nothing
3. getMoved(): nothing;
   getPOS(*reporter*): $[_{NP\ N}\ reporter]$
4. setAttachment($[_{VP}\ [_{NP\ +D}\ the]]$): $[_{NP\ +D}\ the]$
   setAttachment($[_{NP\ +D}\ the]$): $[_{NP\ +D}\ the]$
5. merge($[_{NP\ +D}\ the]$, $[_{NP\ N}\ reporter]$):
   $[_{VP}\ [_{NP\ +D}\ the\ _{N}\ reporter]]$
   merge($[_{NP\ +D}\ the]$, $[_{NP\ N}\ reporter]$):
   $[_{NP\ +D}\ the\ _{N}\ reporter]$



Getting asymmetries with PMG-pa

Subject relatives in head initial languages
a. The reporter who attacked the senator admitted the error.
b. The reporter who the senator attacked admitted the error.

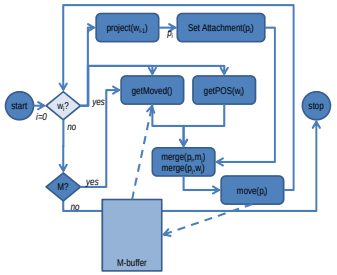
1. w_1 = "reporter"
2. project(*the*): nothing
3. getMoved(): nothing;
   getPOS(*reporter*): [$_{NP\ N}$ *reporter*]
4. setAttachment([$_{VP}$ [$_{NP\ +D}$ *the*]]): [$_{NP\ +D}$ *the*]
   setAttachment([$_{NP\ +D}$ *the*]): [$_{NP\ +D}$ *the*]
5. merge([$_{NP\ +D}$ *the*], [$_{NP\ N}$ *reporter*]):
   [$_{VP}$ [$_{NP\ +D}$ *the* $_{N}$ *reporter*]]
   merge([$_{NP\ +D}$ *the*], [$_{NP\ N}$ *reporter*]):
   [$_{NP\ +D}$ *the* $_{N}$ *reporter*]
6. move([$_{VP}$ [$_{NP\ +D}$ *the* $_{N}$ *reporter*]]):
   A-Buffer[[$_{NP}$]]
   move([$_{NP\ +D}$ *the* $_{N}$ *reporter*]): [$_{NP}$ …]



Getting asymmetries with PMG-pa

Subject relatives in head initial languages
a. The reporter who attacked the senator admitted the error.
b. The reporter who the senator attacked admitted the error.

1. w_2= "who"

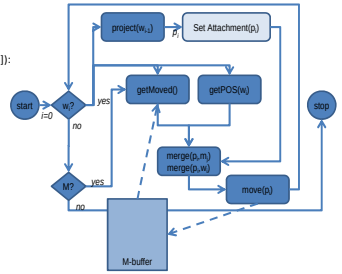


Getting asymmetries with PMG-pa

Subject relatives in head initial languages
a. The reporter who attacked the senator admitted the error.
b. The reporter who the senator attacked admitted the error.

1. w_2= "who"
2. project(*who*): nothing
3. getMoved(): [$_{NP}$];
   getPOS(*who*): [$_{VP\ +C}$ *who*]
4. setAttachment([$_{VP}$ [$_{NP\ +D}$ *the reporter*]]):
   [$_{NP}$ … reporter]

…



Getting asymmetries with PMG-pa

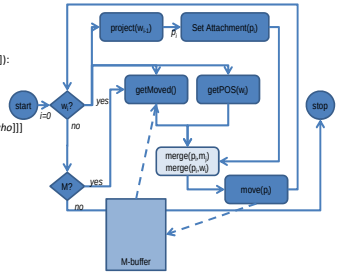
Subject relatives in head initial languages
a. The reporter who attacked the senator admitted the error.
b. The reporter who the senator attacked admitted the error.

1. w_2= "who"
2. project(*who*): nothing
3. getMoved(): [$_{NP}$];
   getPOS(*who*): [$_{VP\ +C}$ *who*]
4. setAttachment([$_{VP}$ [$_{NP\ +D}$ *the reporter*]]):
   [$_{NP}$ … reporter]

…

5. merge([$_{NP}$ … reporter], [$_{NP}$]): failed
   merge([$_{NP}$ … reporter], [$_{VP\ +C}$ *who*]):
   [$_{VP}$ [$_{NP\ +D}$ *the* $_{N}$ *reporter* [$_{VP\ +C}$ *who*]]]

…



Getting asymmetries with PMG-pa

Subject relatives in head initial languages
a. The reporter who attacked the senator admitted the error.
b. The reporter who the senator attacked admitted the error.

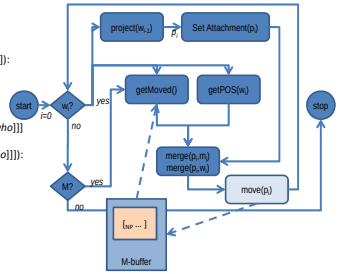
1. w_2= "who"
2. project(*who*): nothing
3. getMoved(): [$_{NP}$];
   getPOS(*who*): [$_{VP\ +C}$ *who*]
4. setAttachment([$_{VP}$ [$_{NP\ +D}$ *the reporter*]]):
   [$_{NP}$ … reporter]

…

5. merge([$_{NP}$ … reporter], [$_{NP}$]): failed
   merge([$_{NP}$ … reporter], [$_{VP\ +C}$ *who*]):
   [$_{VP}$ [$_{NP\ +D}$ *the* $_{N}$ *reporter* [$_{VP\ +C}$ *who*]]]

…

6. move([$_{VP}$ [$_{NP\ +D}$ *the* $_{N}$ *reporter* [$_{VP\ +C}$ *who*]]]):
   A-Buffer<[$_{NP}$]>



Getting asymmetries with PMG-pa

Subject relatives in head initial languages
a. The reporter who attacked the senator admitted the error.
b. The reporter who the senator attacked admitted the error.

1. project(*who*): nothing
2. getMoved(): [$_{NP}$];
   getPOS(*attacked*):
   [$_{VP\ +T\ V\ =NP\ =NP}$ *attacked*]
3. setAttachment([$_{VP}$ [$_{NP}$ … *reporter* [$_{VP}$ *who*]]]): [$_{VP}$ *who*]
   …
4. merge([$_{VP\ +C}$ *who*], [$_{NP}$]): failed
   merge([$_{VP\ +C}$ *who*], [$_{VP\ +T\ V\ =NP\ =NP}$ *attacked*]):
   [$_{VP}$ [$_{NP\ +D}$ *the* $_{N}$ *reporter* [$_{VP\ +C}$ *who*
   $_{+T\ V\ =NP\ =NP}$ *attacked*]]]
5. move([$_{VP\ +C}$ *who* $_{+T\ V\ =NP\ =NP}$ *attacked*]):
   A-Buffer<[$_{NP\ who}$]>

1. project(*who*): nothing
2. getMoved(): [$_{NP}$]; getPOS(*the*):
   [$_{NP\ +D}$ *the*]
3. setAttachment([$_{VP}$ [$_{NP}$ … *reporter* [$_{VP}$ *who*]]]):
   [$_{VP}$ *who*]
   …
4. merge([$_{VP\ +C}$ *who*], [$_{NP}$ who]): failed
   merge([$_{VP\ +C}$ *who*], [$_{NP\ +D}$ *the*]):
   [$_{VP\ +C}$ *who* [$_{NP\ +D}$ *the*]]
   …
5. move([$_{VP}$ [$_{NP}$ … [$_{VP\ +C}$ *who* [$_{NP\ +D}$ *the*]]]]):
   A-Buffer<[$_{NP}$ *who*]>

Getting asymmetries with PMG-pa

Subject relatives in head initial languages
a. The reporter who attacked the senator admitted the error.
b. The reporter who the senator attacked admitted the error.

1. project(*who*): nothing
2. getMoved(): [$_{NP}$];
   getPOS(*attacked*):
   [$_{VP\ +T\ V\ =NP\ =NP}$ *attacked*]
3. setAttachment([$_{VP}$ [$_{NP}$ … reporter [$_{VP}$ *who*]]]): [$_{VP}$ *who*]
   …
4. merge([$_{VP\ +C}$ *who*], [$_{NP}$]): failed
   merge([$_{VP\ +C}$ *who*], [$_{VP\ +T\ V\ =NP\ =NP}$ *attacked*]):
   [$_{VP}$ [$_{NP\ +D}$ *the* $_{N}$ *reporter* [$_{VP\ +C}$ *who*
   $_{+T\ V\ =NP\ =NP}$ *attacked*]]]
5. move([$_{VP\ +C}$ *who* $_{+T\ V\ =NP\ =NP}$ *attacked*]):
   A-Buffer<[$_{NP\ who}$]>

1. project(*the*): nothing
2. getMoved(): [$_{NP}$]; getPOS(*senator*):
   [$_{NP\ N}$ *senator*]
3. setAttachment([$_{VP}$ [$_{NP}$ … [$_{VP}$ *who* [$_{NP}$ *the*]]]]):
   [$_{NP}$ *the*]
   …
4. merge([$_{NP\ +D}$ *the*], [$_{NP\ N}$ *senator*]):
   [$_{VP\ +C}$ *who* [$_{NP\ +D}$ *the* $_{N}$ *senator*]]
   …
5. move([$_{VP}$ [$_{NP}$ … [$_{VP}$ … [$_{NP}$ *the senator*]]]]):
   A-Buffer<[$_{NP}$ *who*], [$_{NP}$ *the senator*]>

Getting asymmetries with PMG-pa

Subject relatives in head initial languages
a. The reporter who attacked the senator admitted the error.
b. The reporter who the senator attacked admitted the error.

1. project(*who*): nothing
2. getMoved(): [$_{NP}$];
   getPOS(*attacked*):
   [$_{VP\ +T\ V\ =NP\ =NP}$ *attacked*]
3. setAttachment([$_{VP}$ [$_{NP}$ … reporter [$_{VP}$ *who*]]]): [$_{VP}$ *who*]
   …
4. merge([$_{VP\ +C}$ *who*], [$_{NP}$]): failed
   merge([$_{VP\ +C}$ *who*], [$_{VP\ +T\ V\ =NP\ =NP}$ *attacked*]):
   [$_{VP}$ [$_{NP\ +D}$ *the* $_{N}$ *reporter* [$_{VP\ +C}$ *who*
   $_{+T\ V\ =NP\ =NP}$ *attacked*]]]
5. move([$_{VP\ +C}$ *who* $_{+T\ V\ =NP\ =NP}$ *attacked*]):
   A-Buffer<[$_{NP\ who}$]>

1. project(*senator*): nothing
2. getMoved(): <[$_{NP}$],[$_{NP}$]>; getPOS(*attacked*):
   [$_{VP\ +T\ V\ =NP\ =NP}$ *attacked*]
3. setAttachment([[[$_{VP}$ [$_{NP}$ … [$_{VP}$ who [$_{NP}$ …]]]]):
   [$_{VP}$ *who* [$_{NP}$ …]] …
4. merge([$_{VP\ +C}$ *who* [$_{NP}$ …]], [$_{NP}$]): failed
   …
   merge([$_{VP}$ *who* [$_{NP}$ …]],
   [$_{VP\ +T\ V\ =NP\ =NP}$ *attacked*]):
   [$_{VP}$ [$_{NP\ +D}$ *the* $_{N}$ *reporter* [$_{VP\ +C}$ *who*
   [$_{NP\ +D}$ *the* $_{N}$ *senator*] $_{+T\ V\ =NP\ =NP}$ *attacked*]]]
5. move([$_{VP}$ [$_{NP}$ … [$_{VP}$ … [$_{NP}$ *the senator*]]]]):
   A-Buffer<[$_{NP}$ *who*], [$_{NP}$ *the senator*]>

Concetti fondamentali del minicorso

- Cos'è una **grammatica formale** e cosa serve per specificarne una
  - Regole di riscrittura e ricorsività
  - Restrizioni sulle regole di riscrittura per creare classi di grammatiche generativamente più o meno potenti (gerarchia di Chomsky)
  - dove stanno i linguaggi naturali e quali sono i limiti di decidibilità
- Che cos'è una **computazione**
  - Cos'è lo spazio del problema e come lo spazio viene esplorato da un algoritmo
  - Come si calcola la complessità di un problema
  - Cosa vuol dire fare parsing di una frase e alcuni algoritmi
- Cos'è una grammatica **Top-Down**
  - Competenza e Performance… riunite
  - Ricostruzione di elementi dalle isole
  - Aspettative e regolamentazione

Puntata X

BONUS SLIDES…

Linguaggio e visione