Luigi Rizzi (rizzil@unisi.it) - Cristiano Chesi (chesi@media.unisi.it) 4 Marzo 2003

IL PARSING (SINTATTICO)

Definizione del problema

(1) Fare **parsing** significa riconoscere un input e assegnare a questo una struttura adeguata; il parsing sintattico riconosce un input grammaticale e assegna ad esso una struttura sintattica.

(2) **dalla teoria alla pratica**: 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 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.

(3) specificare i **requisiti di interfaccia**
a. fonologia (Phonetic Form) o stringhe di testo
b. forma logica (Logical Form)

a. definisce l'informazione base (ovvero l'input) che potremo fornire al parser; b. precisa il tipo di struttura (sintattica) che verrà resa come output e che potrà essere a sua volta un input fondamentale per sistemi di traduzione, estrazione di informazione, correzione ortografica, risposta a domande, applicazione lessicografiche...

(4) **spazio del problema** e **strategie di ricerca**: data una frase ed una grammatica (ad esempio una CFG) il compito del parser è dire se la frase può essere generata dalla grammatica (URP, Universal Recognition Problem, vedi hand-out precedente) 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.

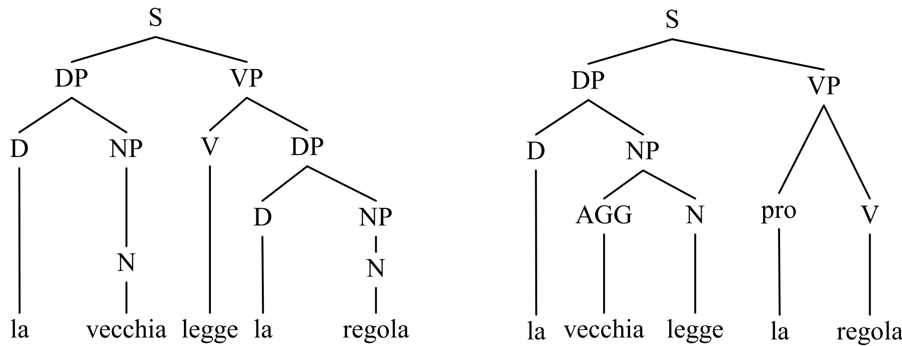
frase: la vecchia legge la regola

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

Due sono i vincoli a cui questo compito sottostà: le regole grammaticali che predicono come da un nodo radice S ci siano solo alcune vie di scomposizione possibili per ottenere i nodi terminali; dall'altro ci sono 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**.

(5) **Top-Down** parsing:
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).
Data la grammatica in (4), la sola regola utile è S → DP VP; questo sarà perciò il primo livello (**ply** o **level**) di espansione. Si prosegue quindi cercando di espandere DP e VP: anche DP ha una sola possibilità di riscrittura; VP invece presenta due alternative. Al secondo livello lo spazio del problema si ramificherà e i due nodi figli generati saranno differenziabili per la regola scelta: VP → V DP oppure VP → pro V. Al terzo livello l'opzionalità del nodo AGG all'interno del NP provocherà un'ulteriore ramificazione. Alla fine i nodi pre-terminali verranno riscritti in quelli terminali e i risultati che combaciano con l'input accettati come buone soluzioni al problema sintattico posto:



Oltre a queste strutture accettabili, anche le strutture di frasi come "la regola regola la regola", "la legge legge la vecchia legge"… verranno generate e solo all'ultimo livello, in fase di mappatura con l'input realmente fornito, saranno scartate.

(6) **Bottom-Up** parsing:
questo è il 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. Mentre nella strategia Top-Down si cercava una corrispondenza tra la parte sinistra della regola e l'input da espandere, in questo caso, la parte che deve combaciare sarà la parte destra della regola. Nella grammatica in (4) ogni parola causerà al primo livello di espansione un'ambiguità che causerà la ramificazione dell'albero.

(7) 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 incombinabili 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.**

(8) 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 meglio utilizzando quella **per ampiezza** (**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 da fare e soprattutto non si rendono conto dell'errore finché l'espansione non è stata completamente effettuata.

(9) Esiste un modo migliore per sfoltire lo spazio del problema cercando di evitare di esplorare regole più improbabili senza ricorrere ad euristiche sofisticate: ad esempio utilizzando una strategia Top-Down per la generazione di strutture filtrata attraverso considerazioni di natura Bottom-Up. Il parsing procederà incrementalmente proponendo una soluzione di espansione Top-Down ed in caso di alternative verrà considerata la prima parola dell'input che dovrà essere integrata nella struttura seguendo l'idea dell'**angolo sinistro** (**left-corner**): l'idea è che 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 $A \rightarrow^* B\alpha$.
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

(10) Nonostante il filtraggio Bottom-Up, le strategie di parsing Top-Down presentano evidenti **problemi** che rendono questo approccio difficilmente praticabile:

a. **ricorsività a sinistra**
una grammatica si dice ricorsiva a sinistra se ammette una regola del tipo A →* Aα (es. DP → DP PP). È facile intuire come una strategia di ricerca che tenta di espandere fino al nodo terminale ogni categoria partendo da sinistra, incontri qualche difficoltà ad espandere un elemento infinite volte.

b. **ambiguità**
a vari livelli si possono trovare molteplici strutture che soddisfano i requisiti dell'input e delle regole grammaticali. Le ambiguità risiedono ad esempio a livello di attaccamento dei PP (ho visto l'uomo con il cannocchiale) e di 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). 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 locale soprattutto, del genere di "vecchia" o "legge") causa facilmente errori da cui poi si deve riuscire a proseguire. 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

(11) 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 descritto in
(10.c).

(12) L'**algoritmo di Earley** (Earley 70) 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)**, con *n* uguale al numero delle parole in input) 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 (del tutto simile ad un **array**) 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.

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])

L'algoritmo procede con tre operazioni fondamentali:

a. **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. Ad esempio questa funzione applicata ad una grammatica come

quella fornita in (4) riscriverà nell'entrata **corrente** del chart la prima regola come S → • DP VP [0,0] e oltre a questa aggiungerà lo stato DP → • D NP [0,0].

b. **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 di 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. Ad esempio se dopo la regola precedentemente generata DP → • D NP [0,0] si esegue una scansione e si trova un articolo tale che nella grammatica in questione D → articolo, l'operazione produrrà lo stato DP → D • NP [0,1] che verrà aggiunto all'entrata successiva del chart.

c. **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];

(13) L'algoritmo di Earley completo, in pseudo-codice, è il seguente:

funzione EARLEY-PARSER(*stringa_parole*, *grammatica*) **restituisce** *grafo*

METTI_IN_CODA((*γ* → •*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*

procedura PREVISIONE (*A* → *α*•*Bβ*, [*i*,*j*])
    per ogni (*B* → *γ*) **tale che** (REGOLE_DELLA_GRAMMATICA(*B*, *grammatica*)) **esegui**
        METTI_IN_CODA((*B* → •*γ*, [*j*,*j*]), *grafo*[*j*])
    fine

procedura SCANSIONE (*A* → *α*•*Bβ*, [*i*,*j*])
    se (*B* è PARTE_DELLA_FRASE(*parola*[*j*])) **allora esegui**
        METTI_IN_CODA((*B* → *parola*[*j*], [*j*,*j*+1]), *grafo*[*j*+1])
    fine

procedura COMPLETAMENTO (*B* → *γ*•, [*j*,*k*])
    per ogni (*A* → *α*•*Bβ*, [*i*,*j*]) **tale che** ((*A* → *α*•*Bβ*, [*i*,*j*]) è nel *grafo*[*j*]) **esegui**
        METTI_IN_CODA((*A* → *αB*•*β*, [*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

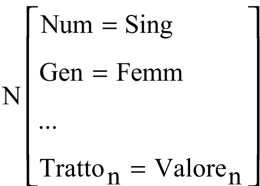
Formalismi basati su restrizioni (Contraint-based formalisms)

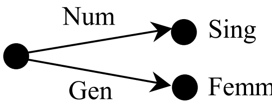
(14) Ricordiamo che **accordo**, **sottocategorizzazione** e **relazioni a distanza** erano problemi che avevano messo in evidente crisi l'approccio proposto dalle grammatiche a struttura sintagmatica (sia le context-free che le context-sensitive): in particolare l'impossibilità di catturare questi fenomeni, se non al costo di una proliferazione esponenziale di regole e categorie che poteva facilmente portare dall'ipergeneralizzazione di certi fenomeni, all'impossibilità di catturarne altri, ha giustificato la ricerca in direzione di una rappresentazione più efficiente e significativa dell'informazione linguistica. Le grammatiche basate su restrizioni nascono con questa vocazione.

Si è cercato quindi un formalismo **leggermente più potente delle CFG**, in cui fosse possibile render conto in modo **compatto** (quindi più elegante) ed **efficiente** delle restrizioni linguistiche imposte da fenomeni produttivi quali quelli appena elencati. Una risposta a queste richieste viene dalle grammatiche che implementando **gerarchie di tratti** come proprietà aggiuntive alle regole di riscrittura.

(15) L'introduzione dei tratti richiede però anche un ulteriore test di "buona costituzione" oltre alle normali regole di concatenazione derivanti dai dispositivi di riscrittura. Questa operazione, che ovviamente riguarderà direttamente i tratti, viene chiamata **unificazione** e verrà formalmente definita in (18). Il suo compito è verificare la compatibilità di due elementi che dovranno essere uniti determinando inoltre i tratti che potranno essere ereditati dal nuovo elemento formatosi in seguito tale unione.

(16) Una semplice **struttura di tratti** (**FS**, **Feature Structures**) è un insieme di coppie del tipo **tratto-valore** (es. numero > singolare). Tali coppie vengono organizzate in una **Matrice di Attributi e Valori** (**AVM**, **Attribute Value Matrix**) o diagramma ad archi orientati ed etichettati (**DAG**, Direct Acyclic Graph)

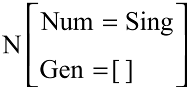




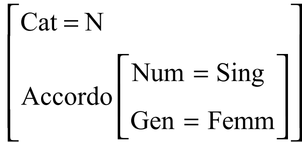
AVM DAG

Alcune proprietà interessanti delle strutture di tratti:

a. **parzialità**, **>** o **<** **specificità**, ovvero alcuni elementi posso restare non specificati, ad esempio il genere:



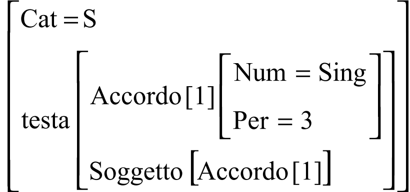
b. la struttura delle AVM può essere **rientrante**, cioè un tratto che ha una qualche significatività dal punto di vista empirico, può essere definito da più sottotratti, come nel caso dell'accordo:



c. **percorsi**, il valore di un tratto è definito in base ad un percorso univoco che lo identifica, cioè una lista di tratti lungo la struttura del tipo: accordo>num>sing

d. **condivisione di tipo** (type sharing), una struttura può essere condivisa tra più elementi anche se i valori non lo sono

e. **condivisione di occorrenza** (token sharing), l'occorrenza di un determinato valore può essere condivisa in tal caso può essere indicato con l'uso di un indice, es [1]:



f. **significatività empirica**, i tratti sembrano catturare adeguatamente, almeno a livello descrittivo, fenomeni linguisticamente produttivi

g. **aciclicità**, i grafi non possono essere ricorsivi

(17) **Sussunzione**
stabilisce una relazione ordinata tra due strutture di tratti FS; la FS più generica sussume quella più specifica. Si può quindi dire che:
$FS_a \prec FS_b$
sse FS_b *ha tutti i tratti di* FS_e *nella stessa configurazione strutturale e con uguali assegnazioni di valore*

(18) **Unificazione**
permette di combinare le informazioni per rappresentarle in formato più compatto e significativo:
$FS_a \cup FS_b = FS_x$ *(se esiste) tale che* FS_x *è la più generale delle FS sussunte da* FS_a *e* FS_b

(19) Per implementare l'unificazione ad una grammatica CFG si può procedere nel seguente modo:
a. si associano strutture di tratti complesse sia agli **elementi lessicali** che alle **categorie grammaticali**
b. si definiscono composizionalmente delle regole di combinazione di questi tratti
c. si stabiliscono i vincoli di compatibilità tra strutture di tratti

Utilizzando il formalismo introdotto da Shieber (1986) le regole nella grammatica saranno espresse come segue:
S → DP VP
<DP accordo = VP accordo>

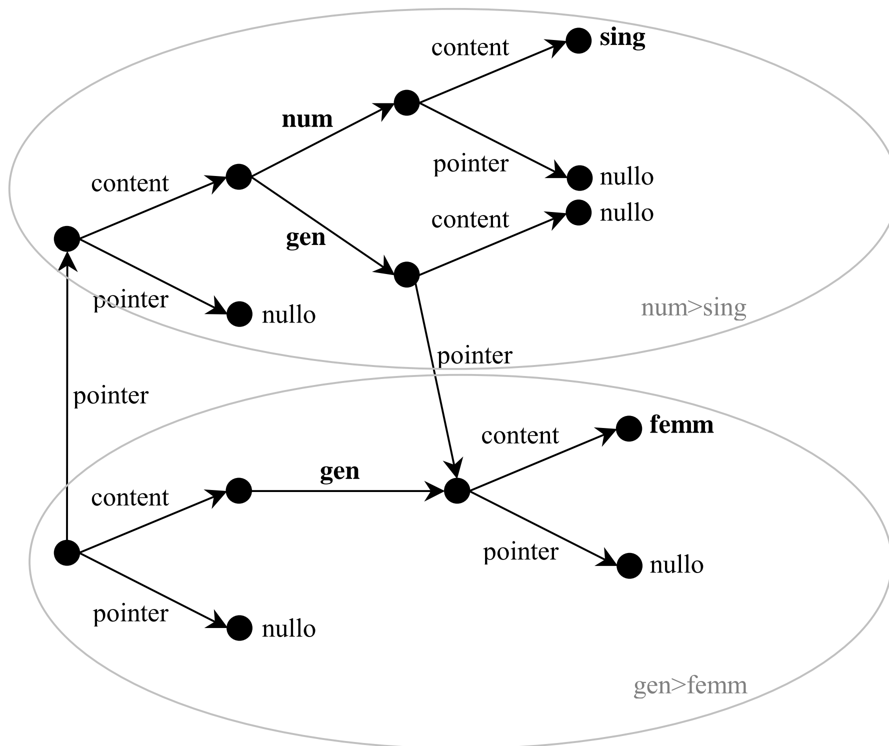
(20) Per quanto riguarda l'**accordo**, si può notare come se all'interno di un NP un nome possiede un tratto singolare, l'intero DP deve essere accordato con tale tratto. In genere è sempre la testa del sintagma lessicale che determina i valori di accordo. Per catturare questa generalizzazione si deve permettere alla testa lessicale di **trasmettere** il valore dei propri tratti ai sintagmi funzionali che lo dominano direttamente: il DP ad esempio **erediterà** i tratti dalla testa nominale, completando attraverso l'operazione di unione gli eventuali tratti sottospecificati e verificando la compatibilità con quelli specificati. Il tratto di testa è quindi un tratto fondamentale, che dovrà essere associato a determinati elementi lessicali e che determinerà la "direzionalità" dell'accordo.

(21) Avevamo visto che per rendere conto dell'asimmetria mostrata dai verbi nel prendere certi argomenti piuttosto che altri si doveva ricorrere ad uno **schema di sottocategorizzazione** che nelle CFG portava ad un'esplosione dei simboli non terminali che dovevano render conto delle varie possibili categorie verbali. Nelle grammatiche su restrizioni, la sottocategorizzazione può essere resa attraverso un tratto **subcat**, associato ad esempio al verbo:

$$\begin{bmatrix} \text{Ortografia} = \text{voglio} \\ \\ \text{Cat} = \text{V} \\ \text{Testa} \begin{bmatrix} \textbf{Subcat} < \begin{bmatrix}\text{Cat} = \text{NP}\end{bmatrix}, \begin{bmatrix} \text{Cat} = \text{VP} \\ \text{Testa} = \begin{bmatrix}\text{forma_infinitiva}\end{bmatrix} \end{bmatrix} > \end{bmatrix} \end{bmatrix}$$

(22) L'**implementazione** dell'algoritmo di unificazione richiede un dispositivo che riceva in input due strutture di tratti, ne verifichi la compatibilità e restituisca una singola struttura unita oppure rifiuti le strutture in input come incompatibili.

L'idea di base è quindi banalmente quella di creare un loop sui DAG da combinare, fino ad esaurimento tratti, creando riferimenti da un DAG all'altro quando strutture e valori sono compatibili. Per fare questo si ricorre ad una piccola modifica dei DAG: ad ogni tratto si associa un **valore di campo** (**content**) e un **puntatore** (**pointer**); il primo indicherà il valore del tratto nel DAG, il secondo stabilirà un link con l'altra struttura dati compatibile.
Questo il risultato dell'unione dei due DAG modificati (num>sing ∪ gen>femm):



(23) Per evitare cicli che occorrerebbero se si cercasse di unire una struttura con un'altra che la contiene già propriamente come sottoparte, si deve ricorrere a quello che si chiama un **occur check**, che impedisce l'unione nel caso citato.

(24) Siamo a questo punto pronti ad integrare l'operazione di unificazione con l'algoritmo di parsing prescelto. Il livello di astrazione della definizione permette di applicare l'unificazione a qualsiasi algoritmo. Si possono distinguere almeno due modalità sostanzialmente diverse di integrazione: la prima consiste nell'applicare l'operazione come **filtro** (escludendo quindi le soluzione incompatibili una volta che tutte le soluzioni sono state generate dalla CFG, ad esempio), la seconda sceglie di porre dei **constraints** al parser a valutare di volta in volta i requisiti di unificazione in modo da impedire immediatamente la generazione di strutture che poi alla fine verrebbero rifiutate. Questa seconda soluzione è spesso la migliore (se si riesce a minimizzare lo sforzo computazionale che i constraints richiedono) e sarà quella che cercheremo di perseguire integrando l'unificazione con l'algoritmo di Earley.
Le sostanziali modifiche rispetto all'algoritmo originale sono essenzialmente quattro:

a. le regole di riscrittura vengono integrate con informazioni sui tratti, come mostrato in (19)
b. agli stati che rappresentano le regole parzialmente parseggiate vengono associate informazioni sui tratti in forma di DAG:
   S → • DP V, [0,0], [], DAG
c. quando si applica l'operazione di completamento, il nuovo costituente che deve essere integrato, verrà valutato in base all'algoritmo di unificazione per verificarne la compatibilità con il DAG fino a quel momento costruito dall'applicazione della regola
d. anche l'operazione di incolonnamento (che inseriva in coda agli stati regole solo regole non presenti nello stack) adesso dovrà verificare la compatibilità del DAG prima di escludere l'incolonnamento di regole che per la grammatica precedente potevano apparire identiche.

vista la natura dell'algoritmo di unificazione che essenzialmente tenderebbe a modificare i DAG compatibili, perderemmo l'utile proprietà dell'algoritmo di Earley di riutilizzare, senza rigenerarle, strutture già create durante l'applicazione delle regole. In effetti se l'unificazione fallisse, ci ritroveremmo con una struttura modificata ed inutilizzabile per future operazioni. Per evitare questo si introduce un'operazione di COPYDAG, che ogni volta conserva una copia del DAG che la regola cerca di unificare.

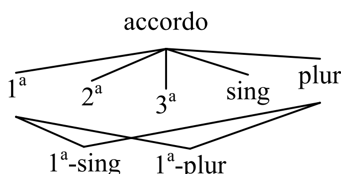
(25) **Tipologie ed ereditarietà**
Con l'introduzione dei tratti non è stato affrontato il problema dei **vincoli nell'assegnazione di valore**: niente in effetti vieterebbe di assegnare valore *femminile* al tratto *numero*.

Inoltre non si riesce a cogliere **nessuna generalizzazione** laddove sottoparti della struttura dei tratti sono ripetute (ad esempio i complementi selezionati per la sottocategorizzazione).

Per queste ragioni si introducono i **tipi**, cioè classi di strutture di tratti che rappresentano uniformemente elementi che hanno un comportamento linguistico significativamente omogeneo (accordo, caso...); in particolare il sistema dei tipi permette di definire:

a. **condizioni di appropriatezza** che specificano quali tratti sono adeguati in un determinato tipo
b. **gerarchia di tipi**, attraverso cui i tipi più specifici ereditano le caratteristiche da quelli più astratti
c. **unificazione di tipi** oltre che semplici tratti

ecco un esempio di gerarchia di tipi:



molti sforzi si sono concentrati sull'arricchimento delle proprietà espresse dai tratti tipizzati, tra queste l'idea dei valori di **default** (valori assegnati ad un tratto quando questo rimane sottospecificato) o la **priorità di unione** (viene specificato l'ordine di combinazione di certi tratti all'interno della gerarchia).

Alcune considerazioni su efficienza e prestazioni

(26) 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 (66) 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.

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.

(27) Applicazioni del parsing: **cascate di automi a stati finiti**;
il presupposto che giustifica questi semplici dispositivi è che non tutti i tipi di processamento linguistico richiedono un parsing completo. In particolare molte strutture risultano **localmente non ambigue** e quindi elaborabili indipendentemente dal resto della frase. Questo principio è quello che guida molti algoritmi ad esempio per l'**estrazione di informazioni** che si avvalgono di **analisi parziali** (**partial parse** o **shallow parse**).
Il costo di usare FSA anziché CFG porta ad esempio all'esclusione dalla grammatica utilizzata di regole ricorsive in senso stretto (es. NP → NP PP) e quindi ad una minore potenza generativa corrispondente ad una difficoltà nel catturare fenomeni realmente presenti.

🗐 Bibliografia essenziale:

Jurafasky & Martin 2000 *Speech & Language Processing*. Pretience Hall, NJ (**Cap. 10,11**)

🗐 Approfondimenti:

- Allegranza, V., Mazzini G. (2000) *Linguistica generativa e grammatiche a unificazione*. Paravia scriptorium.
- Allen J. (1987) *Natural Language Understanding*. MIT Press