Linguaggi Formali e Compilatori

Grammatiche con Attributi

Prof. Stefano Crespi Reghizzi

(impaginazione a cura di prof. Luca Breveglieri)

20 novembre 2008

Introduzione e prime nozioni

La compilazione richiede funzioni non definibili con i metodi puramente sintattici, vale a dire:

- automa finito o a pila, con due nastri di ingresso o con uno di ingresso e uno di uscita

- grammatica libera di traduzione (schema sintattico di traduzione)

Esempio: la traduzione di un numero frazionario dalla base 2 alla base 10.

Esempio: la traduzione di una dichiarazione di struttura (record), calcolando lo spiazzamento in memoria centrale di ogni campo del record.

Esempio: la costruzione della tabella dei simboli di un record (con sintassi di tipo Pascal):

```
LIBRO: record
  AUTORE:   array[1..8] of char;
  TITOLO:   array[1..20] of char;
  PREZZO:   real;
  QUANTITA: integer;
end;
```

simbolo	*tipo*	*dimensione* (in byte)	*indirizzo* (in byte)
LIBRO	record	34	3401
AUTORE	string	8	3401
TITOLO	string	20	3409
PREZZO	real	4	3428
QUANTITA	integer	2	3432

È evidente come la traduzione richieda funzioni aritmetiche, necessarie per calcolare gli indirizzi.

Traduttori guidati dalla sintassi:

- Usano funzioni operanti sull'albero sintattico, e calcolano variabili, o *attributi semantici*.

- I valori degli attributi costituiscono la traduzione ovvero esprimono il *significato*, o *semantica*, della frase sorgente.

- La grammatica con attributi non è un modello formale, poiché le procedure di calcolo degli attributi sono programmi non formalizzati.

- Essa è piuttosto un metodo di ingegneria del software per progettare i compilatori in modo ordinato e coerente, evitando scelte infelici.

Compilazione a due passate:

1. *Parsificazione* (parsing) o *analisi sintattica* $\longrightarrow$ albero sintattico *astratto*.

2. *Valutazione* o *analisi semantica* $\longrightarrow$ albero sintattico *decorato*.

In genere la sintassi astratta è la più semplice possibile, compatibilmente con la struttura semantica del linguaggio.

L'ambiguità nella sintassi astratta non distrugge l'univocità di traduzione, poiché il parsificatore passa al valutatore *un solo* albero astratto.

Nei traduttori più semplici si possono riunire le due fasi in una sola passata, usando la sintassi concreta del linguaggio.

Esempio: la conversione da base 2 a base 10.

L. sorgente: $L = \{0,1\}^* \bullet \{0,1\}^*$

Il punto '$\bullet$' separa la parte intera del numero (visto come stringa di alfabeto $\Sigma = \{0,1,\bullet\}$) da quella frazionaria.

Il significato o traduzione della stringa $1101 \bullet 01_{due}$ (in base due) è $13,25_{dieci}$ (in base dieci).

Grammatica con attributi:

sintassi	*funzioni semantiche*	
$N \to D \bullet D$	$v_0 := v_1 + v_2 \times 2^{-l_2}$	
$D \to DB$	$v_0 := 2 \times v_1 + v_2$	$l_0 := l_1 + 1$
$D \to B$	$v_0 := v_1$	$l_0 := 1$
$B \to 0$	$v_0 := 0$	
$B \to 1$	$v_0 := 1$	

Consiste di regole sintattiche (produzioni) appaiate a regole semantiche ausiliarie.

Attributi e loro significato:

nome	*significato*	*dominio*	*nonterm. assoc.*
v	valore	num. fraz.	N, D, B
l	lunghezza	num. int.	D

Ogni funzione semantica è associata a una produzione di *supporto*. Una produzione di supporto può avere parecchie funzioni semantiche.

Il pedice, v_0, v_1, v_2, l_0 e l_2, specifica a quale simbolo della produzione sia associato l'attributo:

$$\underbrace{N}_{0} \rightarrow \underbrace{D}_{1} \bullet \underbrace{D}_{2}$$

La regola $v_0 := \ldots$ assegna a v_0 il valore dell'espressione contenente gli argomenti v_1, v_2, l_2. Per esempio: $v_0 := f(v_1, v_2, l_2) = v_1 + v_2 \times 2^{-l_2}$.

Dunque la grammatica con attributi di prima, denotata in modo completo, è la seguente:

sintassi	*funzioni semantiche*	
$N_0 \to D_1 \bullet D_2$	$v_0 := v_1 + v_2 \times 2^{-l_2}$	
$D_0 \to D_1 B_2$	$v_0 := 2 \times v_1 + v_2$	$l_0 := l_1 + 1$
$D_0 \to B_1$	$v_0 := v_1$	$l_0 := 1$
$B_0 \to 0$	$v_0 := 0$	
$B_0 \to 1$	$v_0 := 1$	

Se si desidera essere ancora più espliciti, si scriva:

sintassi	*funzioni semantiche*	
$N_0 \to D_1 \bullet D_2$	$v_{0,N} := v_{1,D} + v_{2,D} \times 2^{-l_{2,D}}$	
$D_0 \to D_1 B_2$	$v_{0,D} := 2 \times v_{1,D} + v_{2,B}$	$l_{0,D} := l_{1,D} + 1$
$D_0 \to B_1$	$v_{0,D} := v_{1,B}$	$l_{0,D} := 1$
$B_0 \to 0$	$v_{0,B} := 0$	
$B_0 \to 1$	$v_{0,B} := 1$	

Così è ultrachiaro, ma forse un po' noioso.

Comunque, di solito agli attributi basta mettere i pedici numerici e a parte specificare a quali (non) terminali ciascun attributo sia associabile, insieme al dominio dell'attributo e a una spiegazione sintetica circa il significato (semantica) dell'attributo stesso.

Se un certo (non) terminale comparisse una sola volta in una data produzione, basterebbe applicare all'attributo soltanto il nome del (non) terminale, omettendo il pedice (ma non guasta metterlo comunque).

Qualcuno scrive 'v_0 of N' invece di '$v_{0,N}$' o semplicemente di 'v_0' (la numerazione, se si sa quale sia la produzione da guardare, basta), o anche scrive 'v of N_0'; è solo un modo diverso di denotare. Per esempio:

$$v_0 \text{ of N} := v_1 \text{ of D} + v_2 \text{ of D} \times 2^{-l_2 \text{ of D}}$$

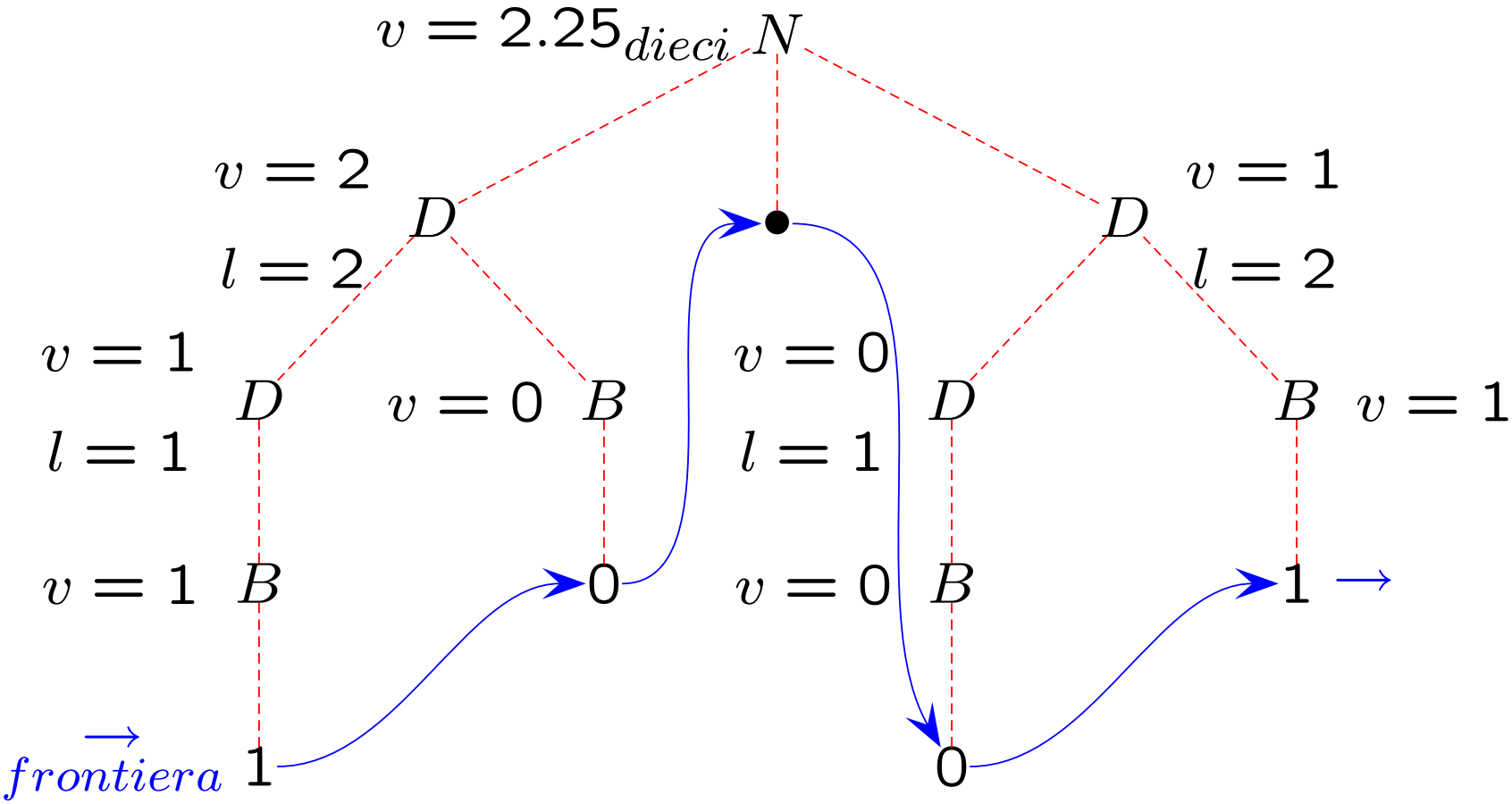
oppure

$$v \text{ of } N_0 := v \text{ of } D_1 + v \text{ of } D_2 \times 2^{-l \text{ of } D_2}$$

Dato un albero sintattico, in ogni nodo si applica una funzione semantica, cominciando dai nodi dove gli argomenti sono noti, in genere i nodi terminali (cioè le foglie dell'albero).

L'albero decorato con i valori è la traduzione della stringa data (la quale si legge sulla frontiera dell'albero). Si veda l'esempio seguente:

albero decorato con gli attributi



Nel disegno non serve mettere pedici agli attributi, perché la vicinanza ai nodi indica da sola quale sia il (non) terminale di riferimento.

Sono possibili più ordini di calcolo degli attributi: tutti però rispettano la condizione di non eseguire una certa funzione prima di avere eseguito quelle che ne calcolano gli argomenti.

Il risultato finale è il valore che, terminato il calcolo, si legge nella radice dell'albero.

Gli attributi degli altri nodi sono intermedi e dopo il calcolo non servono più.

Quelli delle foglie spesso sono attributi iniziali (calcolati dallo scanner), da dove partono il calcolo e la propagazione.

Attributi di due tipi: *sinistri* (sintetizzati o synthesized) e *destri* (ereditati o inherited):

- sinistro $\Rightarrow$ la funzione $a_0 = f(...)$ associata al non.term. della parte sx della produzione

- destro $\Rightarrow$ la funzione $a_k = f(...)$, $k \geqslant 1$ associata a un simbolo della parte dx della produzione

Esempio sopra: tutti sinistri (caso semplice)

Un esempio più articolato e complesso

Problema: come impaginare un testo in formato libero in modo da avere righe di $\leqslant W$ caratteri.

Il testo è una lista di una o più parole separate da uno spazio ($\perp$); il simbolo c sta per un carattere.

Formato di impaginazione: ogni riga contiene il numero massimo possibile di parole indivise.

L'attributo significativo è $ultimo$ (di seguito abbreviato ult): indica il numero della colonna dove si trova l'ultima lettera di una parola.

Si prenda la frase seguente:

"la torta ha gusto ma la grappa ha forza"

con $W = 13$ (max lunghezza di riga).

Testo impaginato in modo corretto:

1	2	3	4	5	6	7	8	9	10	11	12	13
l	a		t	o	r	t	a		h	a		
g	u	s	t	o		m	a		l	a		
g	r	a	p	p	a		h	a				
f	o	r	z	a								

L'attributo $ultimo$ vale 2 per 'la' e 5 per 'forza'.

Attributi e loro significato:

- lun è sinistro, ed esprime la *lunghezza* in caratteri della parola corrente
- pre è destro, ed esprime la colonna dell'ultimo carattere della parola *precedente*
- ult è sinistro, ed esprime la colonna dell'*ultimo* carattere della parola corrente

Per calcolare l'attributo ult della parola corrente w_k, si deve prima conoscere il numero della colonna dell'ultimo carattere della parola immediatamente precedente w_{k-1}, numero che è indicato dall'attributo pre, incrementarlo di 1 e aggiungere la lunghezza della parola corrente w_k.

$$ult(w_k) := pre(w_{k-1}) + 1 + lun(w_k)$$

Per la prima parola del testo si pone $pre = -1$.

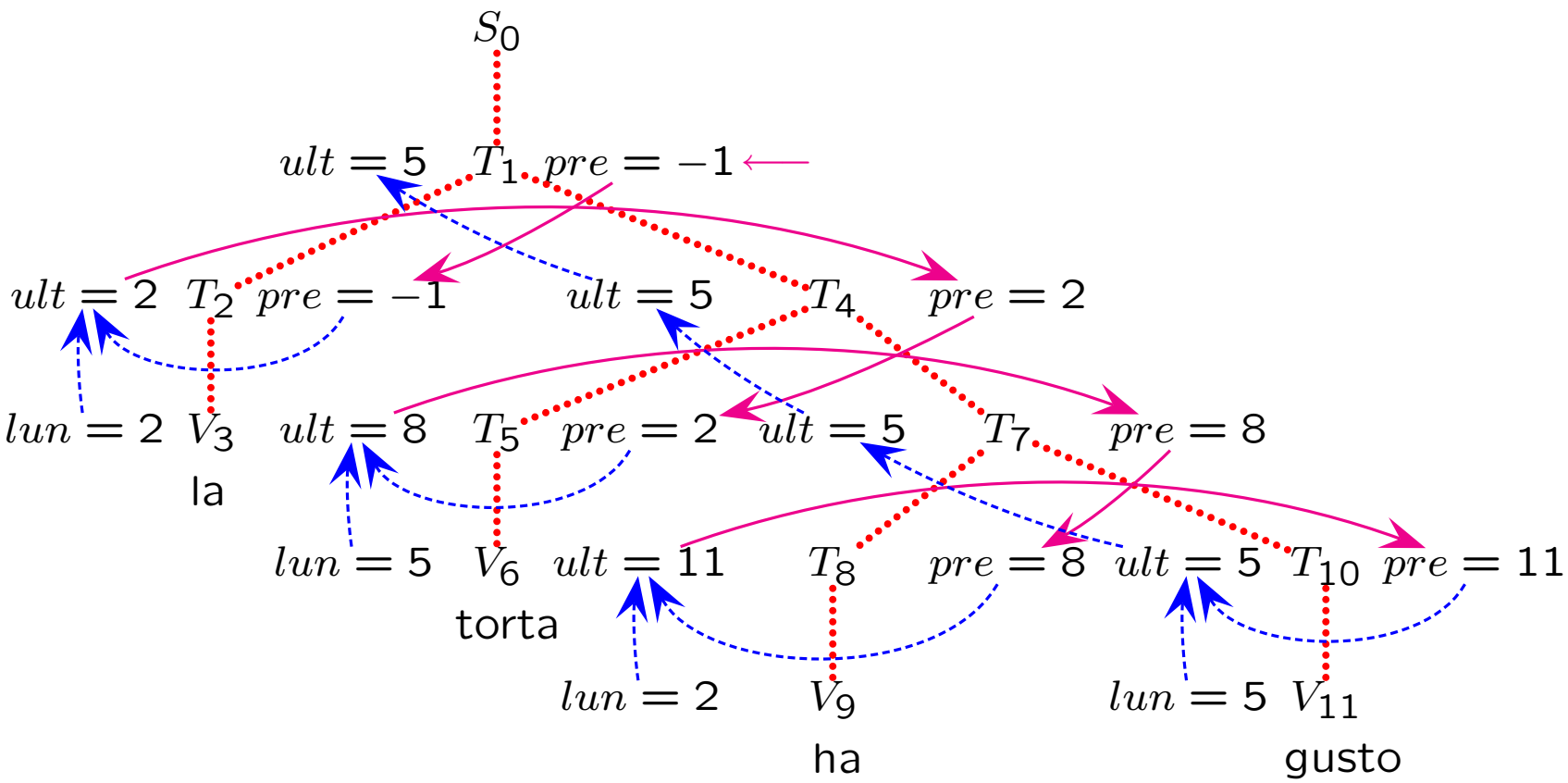
Grammatica e regole semantiche ($\perp$ = spazio):

sintassi	*funzioni semantiche*	
	attributi destri	*attributi sinistri*
1 $S_0 \to T_1$	$pre_1 := -1$	
2 $T_0 \to T_1 \perp T_2$	$pre_1 := pre_0$ $pre_2 := ult_1$	$ult_0 := ult_2$
3 $T_0 \to V_1$		$ult_0 :=$ **if** $(pre_0 + 1 + lun_1 \leqslant W)$ **then** $(pre_0 + 1 + lun_1)$ **else** (lun_1) **end if**
4 $V_0 \to c\, V_1$		$lun_0 := lun_1 + 1$
5 $V_0 \to c$		$lun_0 := 1$

La sintassi è *ambigua* (per la regola $T \rightarrow T \perp T$ con ricorsione bilaterale), ma ciò non dà guai: al valutatore arriva un solo albero sintattico, scelto in qualche modo tra i vari possibili.

La versione ambigua della grammatica è preferibile per la sua maggiore semplicità rispetto alla versione non ambigua.

Grafo delle dipendenze tra le var. degli ass.:



Convenzioni notazionali per disegnare il grafo:

- Lati tratteggiati: relazioni sintattiche.
- Frecce continue: calcolo di pre.
- Frecce tratteggiate: calcolo di ult.
- Si mettono gli attributi destri (pre) e sinistri (lun, ult) a dx e sx del nodo, risp.

Il grafo delle dipendenze è *privo di circuiti*.

Ogni ordine di calcolo che soddisfi le precedenze permette di calcolare i valori degli attributi.

Se la grammatica rispetta certe condizioni (che si diranno in seguito), il risultato non dipende dall'ordine di applicazione delle funzioni.

Domanda: questa grammatica usa attributi destri e sinistri, ma si potrebbe esprimere lo stesso calcolo con una grammatica priva di attributi destri ? *Risposta*: sì. Ecco come:

1. calcola l'attributo sinistro lun

2. costruisci un nuovo attributo sinistro $lista$, che abbia come dominio una lista ordinata di interi, rappresentanti le lunghezze delle parole

Nella figura il nodo T da cui deriva la frase

"la torta ha gusto"

avrebbe l'attributo: $lista = \langle 2, 5, 2, 5 \rangle$.

Il valore di $lista$ nella radice permette di calcolare, conoscendo W, la posizione dell'ultimo carattere di ogni parola.

Difetti della nuova proposta:

- il calcolo da fare sulla lista nella radice è sostanzialmente lo stesso del problema iniziale
- il problema non è dunque decomposto in sottoproblemi più semplici o almeno un po' differenti da quello di partenza

- l'informazione finale rimane concentrata nella radice e non può decorare i nodi interni

- è necessario l'uso di attributi non scalari, aventi un dominio complesso (liste o insiemi)

La soluzione più elegante ed efficiente è spesso quella che fà uso sia di attributi destri sia di attributi sinistri, scambiantisi informazioni.

Definizione di grammatica con attributi

1. Si dà una sintassi $G = (V, \Sigma, P, S)$, dove V e Σ sono gli insiemi dei nonterminali e terminali, P è l'insieme delle produzioni e S è l'assioma. Conviene che l'assioma non figuri in alcuna parte destra di produzione e che la produzione assiomatica sia unica.

2. Si dà un insieme di *attributi* (semantici), associati ai simboli nonterminali e terminali. Gli attributi associati a un simbolo D (non) terminale sono denotati α, β, ..., e sono raggruppati nell'insieme $attr(D) = \{\alpha, \beta, \ldots\}$.
   L'insieme degli attributi è spartito in due insiemi *disgiunti*: *attributi sinistri* (o sintetizzati, synthesized, p. es. σ) e *attributi destri* (o ereditati, inherited, p. es. δ o η).

3. Per ogni attributo α (sia esso sx o dx) è specificato un *dominio*, cioè un insieme finito o infinito da dove l'attributo prende valore.

   Uno stesso attributo α può essere associato a più simboli (non) terminali. L'attributo $\alpha \in attr(D_i)$ associato al (non) terminale D_i si indica scrivendo α_i, con il pedice. Se non c'è pericolo di confusione qualcuno scrive pure "α of D", "α_D", o con altre notazioni similari.

4. Si dà un insieme di *funzioni* (o regole) semantiche. Ogni funzione è associata a una produzione di supporto:

$$p\colon\ D_0 \to D_1 D_2 \ldots D_r \qquad r \geqslant 1$$

Più funzioni possono avere lo stesso supporto. L'insieme di funzioni semantiche associate a una data produzione p si denota $fun\,(p)$.

5. Una funzione semantica:

$$\alpha_k := f\left(attr(\{D_0, D_1, \ldots, D_r\} \setminus \{\alpha_k\})\right)$$

con $0 \leqslant k \leqslant r$, assegna un valore a α di D_k mediante l'applicazione di una regola di calcolo f avente come argomenti attributi della *stessa* produzione p, escluso il risultato della funzione (nessuna ricorsione). Le funzioni semantiche sono *totali*.

6. Le funzioni semantiche sono scritte in una notazione apposita (*metalinguaggio semantico*): in genere in linguaggio programmativo, oppure in pseudocodice o anche in linguaggio di specifica del software. In concreto:
   - $\sigma_0 := f(\ldots)$ definisce un attributo *sinistro*, del padre
   - $\delta_i := f(\ldots)$, con $1 \leqslant i \leqslant r$, definisce un attributo *destro*, di un figlio

7. Attributi dei simboli terminali:

   - sono sempre di tipo destro
   - sono spesso definiti non per mezzo di funzioni semantiche, bensì tramite valori costanti loro assegnati nella fase di analisi lessicale, a monte della valutazione (...lessemi...)
   - non di rado un attributo di un terminale prende come valore il terminale stesso

8. Per l'insieme $fun(p)$ delle funzioni semantiche aventi la produzione p come supporto, devono valere le condizioni seguenti:

(a) per ogni attributo sinistro σ_i si abbia che:

- se $i = 0$, *esista una*, e una sola, *funzione* che lo definisca: $\exists\,!\ (\sigma_0 := f\,(\ldots)) \in fun(p)$
- se $1 \leqslant i \leqslant r$, *non esista nessuna funzione* che lo definisca: $\nexists\ (\sigma_i := f\,(\ldots)) \in fun(p)$

(b) per ogni attributo destro δ_i si abbia che:

- se $1 \leqslant i \leqslant r$, *esista una*, e una sola, *funzione* che lo definisca: $\exists\,!\ (\delta_i := f\,(\ldots)) \in fun(p)$
- se $i = 0$, *non esista nessuna funzione* che lo definisca: $\nexists\ (\delta_0 := f\,(\ldots)) \in fun(p)$

Dunque: se σ è sinistro mai avere $\sigma_i := \ldots$ con $i \neq 0$; se δ è destro mai avere $\delta_0 := \ldots$.

9. Gli attributi sinistri σ_0 e destri δ_i, con $i \neq 0$, essendo quelli definiti dalle funzioni semantiche aventi come supporto la produzione p, sono detti *interni* per p.

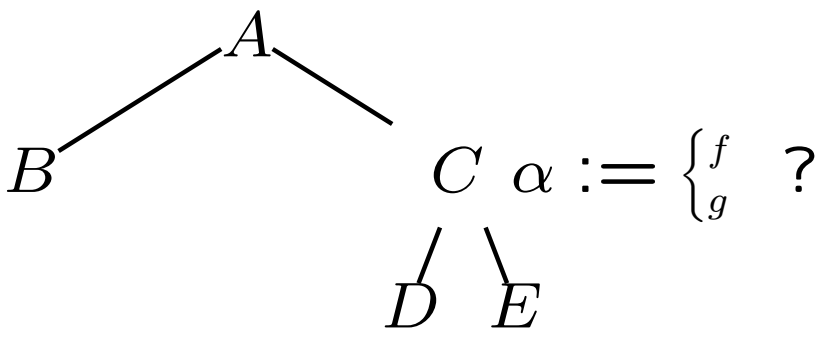
   Gli attributi destri δ_0 e sinistri σ_i, con $i \neq 0$, sono detti *esterni* per p, essendo definiti da funzioni semantiche aventi come supporto un'altra produzione $\neq p$.

10. Si possono inizializzare alcuni attributi con valori costanti o calcolati da funzioni esterne.

    Ciò accade sempre per gli attributi associati ai simboli terminali del linguaggio.

Unicità della definizione: è *vietato* a un attributo α essere sia sinistro sia destro, perché altrimenti vi sarebbero sull'albero due assegnamenti conflittuali allo stesso attributo. Per esempio:

	supporto	*funzione semantica*
1	$A \to BC$	- - attributo destro $\alpha_C := f(attr(A, B))$
2	$C \to DE$	- - attributo sinistro $\alpha_C := g(attr(D, E))$



L'attributo α_C, interno alle produzioni 1 e 2, è destro in 1, sinistro in 2: non va bene !

Il valore di α_C dipende dall'ordine di applicazione delle funzioni semantiche relative (f e g).

La semantica perde dunque la qualità di indipendenza dall'implementazione del valutatore.

Principio di località delle funzioni semantiche:

Errore: porre come argomento o risultato di una funzione semantica, di supporto p, un attributo **estraneo** alla produzione p.

Esempio: modificando le regole 2 dell'esempio precedente, si otterrebbe quanto segue

	sintassi	*funzioni semantiche*
1	$S_0 \to T_1$	$\ldots$
2	$T_0 \to T_1 \perp T_2$	$pre_1 := pre_0 \quad + \underbrace{lun_0}_{\text{attr. non locale}}$
3	$\ldots$	

L'attributo lun è associato (per definizione) solo a V; ma V non compare nella produzione numero 2, dunque si violerebbe la condizione di località, ciò che precluderebbe la visibilità degli attributi dei nodi che non fossero il padre o i figli.

Costruzione del valutatore semantico

Il valutatore semantico è una grammatica che specifica la traduzione, ma senza l'onere di programmarne *ad hoc* l'ordine di calcolo.

La procedura per il calcolo degli attributi sarà costruita (automaticamente o a mano) in base alle dipendenze funzionali degli attributi.

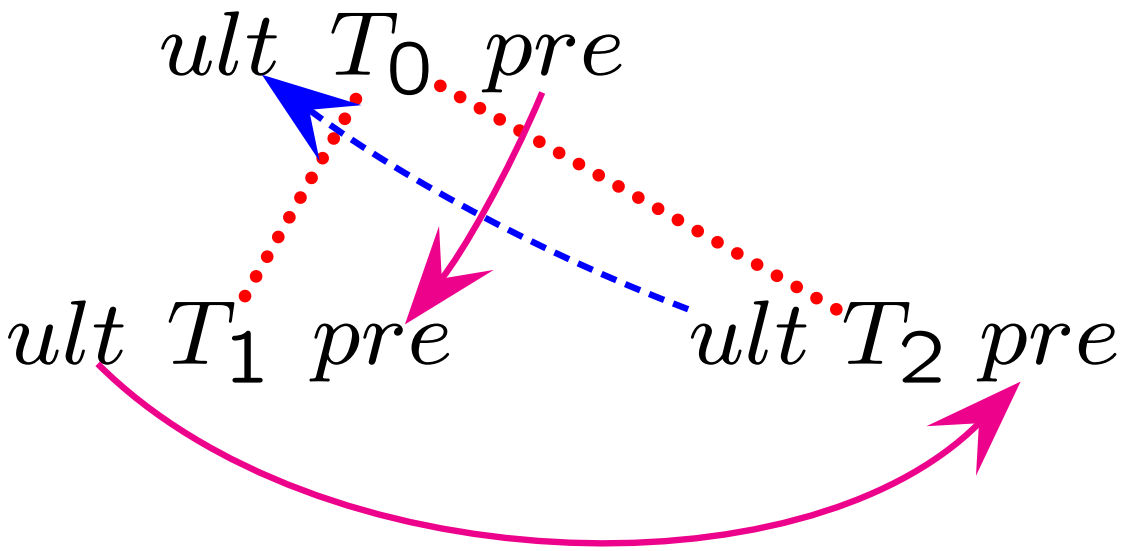
Grafo delle dipendenze di funzione semantica:

È un *grafo orientato* (directed graph) con nodi, attributi e archi disposti nel modo seguente:

- i nodi sono gli argomenti e il risultato
- vi è un arco da ogni argomento al risultato
- e di solito si mettono gli attributi *sintetizzati* (o sinistri) a *sx* del nodo cui si associano, quelli *ereditati* (o destri) a *dx* del nodo

Il grafo va disegnato sovrapposto al supporto sintattico. Ecco l'esempio per la produzione 2:

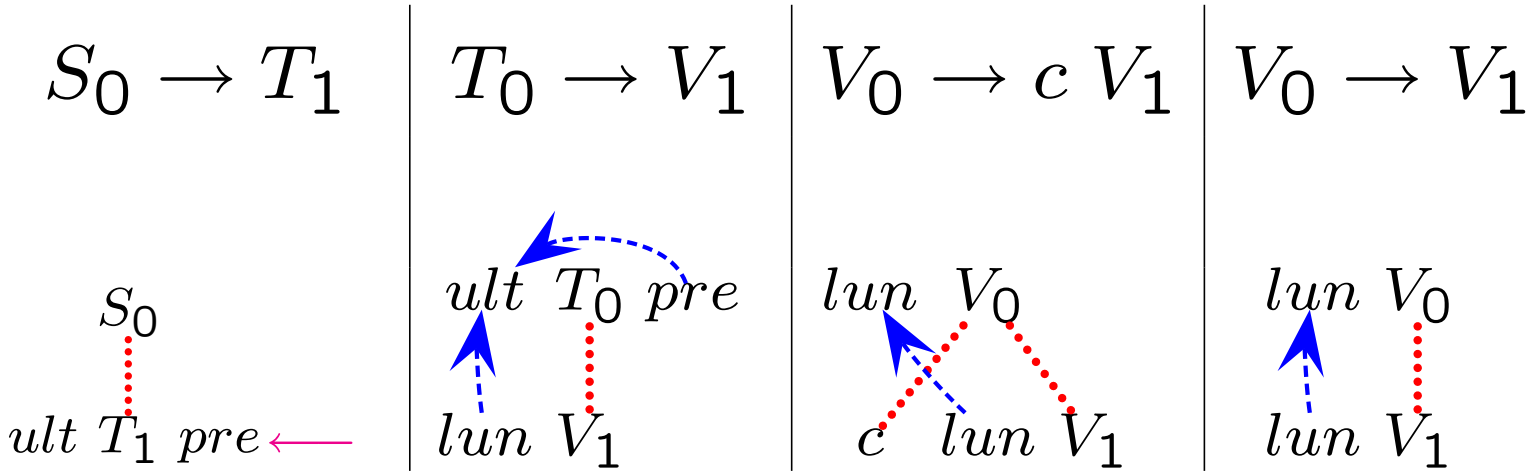
$$
\begin{aligned}
2 : T_0 &\rightarrow T_1 \perp T_2 \\
ult_0 &:= f_1(ult_2) \\
pre_1 &:= f_2(pre_0) \\
pre_2 &:= f_3(ult_1)
\end{aligned}
$$



Attr. sin.: frecce in su; des.: in giù o di lato !

(per semplicità il terminale $\perp$ è omesso)

Altre produzioni della stessa grammatica:



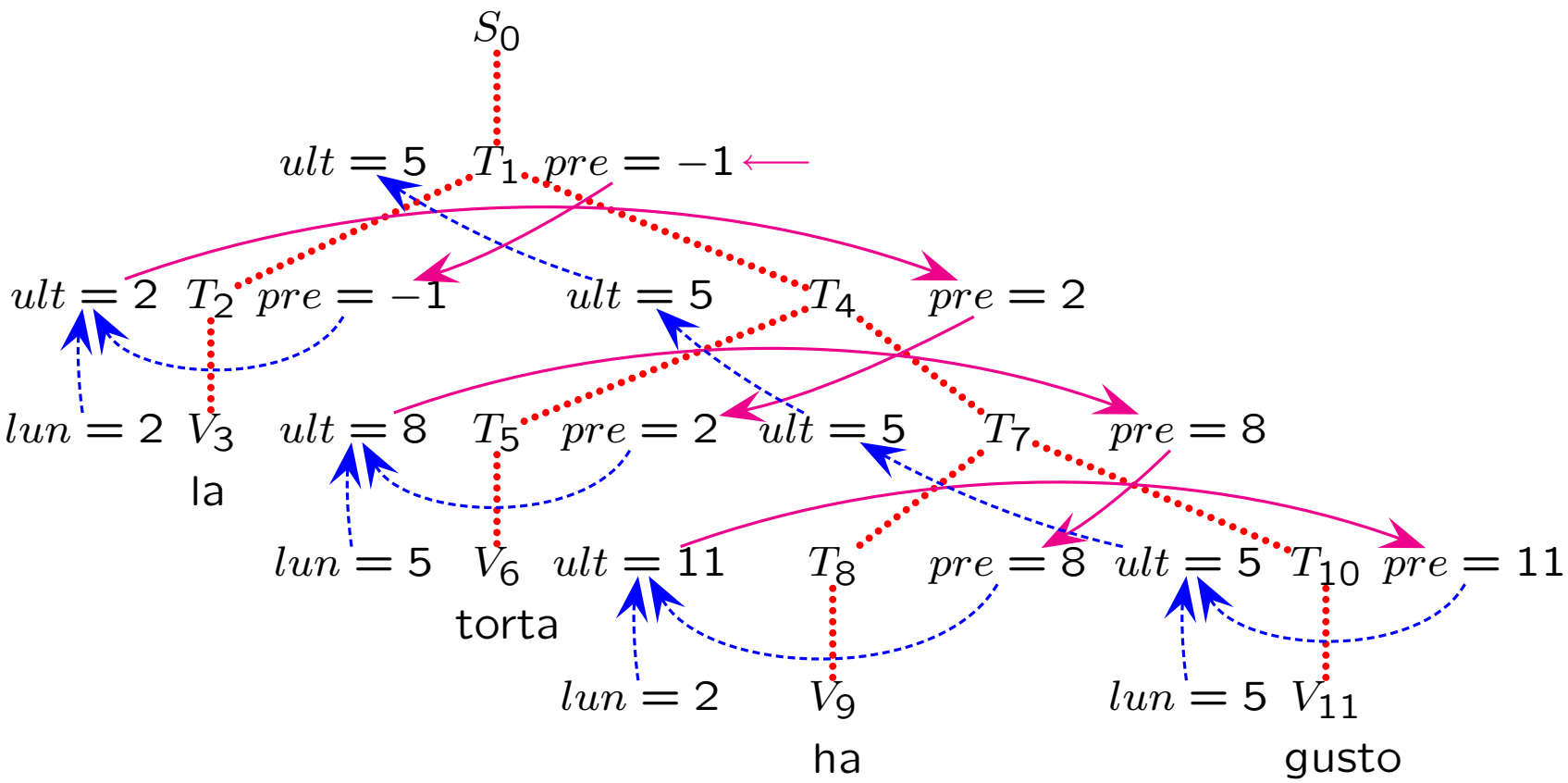
I nomi con e senza archi entranti indicano attributi interni ed esterni, rispettivamente.

Grafo delle dipendenze di un albero sintattico:

È ottenuto incollando insieme i grafi delle singole produzioni usate nei nodi dell'albero.

Il tutto si riporta sull'albero sintattico astratto completo della frase, che dunque fà da supporto per la trama delle dipendenze funzionali.

albero decorato con le dipendenze funzionali



Esistenza e unicità della soluzione

Data una grammatica con attributi che soddisfa le condizioni della definizione, se il grafo delle dipendenze tra gli attributi di un albero è aciclico, esiste un solo insieme di valori per i suoi attributi conforme alle dipendenze.

La *grammatica* è detta *aciclica* (loop-free), se ogni albero ha un grafo delle dipendenze aciclico.

Ipotesi: la grammatica sia sempre aciclica (poi si vedrà come garantire che ciò valga).

Bisogna prima vedere come ordinare linearmente gli assegnamenti agli attributi, in modo che ognuno di essi venga eseguito dopo quelli che ne calcolano gli argomenti.

Algoritmo di ordinamento topologico del grafo:

Sia dato $G = (V, E)$ grafo aciclico, dove i nodi hanno etichette numeriche, $V = \{1, 2, \ldots, |V|\}$.

L'algoritmo calcola un ordine totale tra i nodi, detto *ordine topologico* (topological sorting).

$ord\,[i]$, $i \in 1..|V|$, contiene in posizione i il nodo che occupa l'i-sima posizione nell'ordinamento

input $G = (V, E)$

output ord - - vettore di nodi ordinati

begin - - inizia acquisendo il grafo G

$m := 1$ - - variabile locale contatore di nodi

while $V \neq \emptyset$ **do** - - toglie in nodi in un ordine topologico
- - e termina quando esauriti tutti i nodi

$n :=$ *un nodo qualsiasi di* V *senza archi entranti*
- - il nodo esiste: G non ha cicli

$ord[m] := n$ - - inserisce n nell'ordinamento in posiizone m

$m := m + 1$ - - incrementa il contatore di nodi

$E := E \setminus \{$archi uscenti da n $\}$

$V := V \setminus \{$n$\}$ - - NB: G rimane aciclico

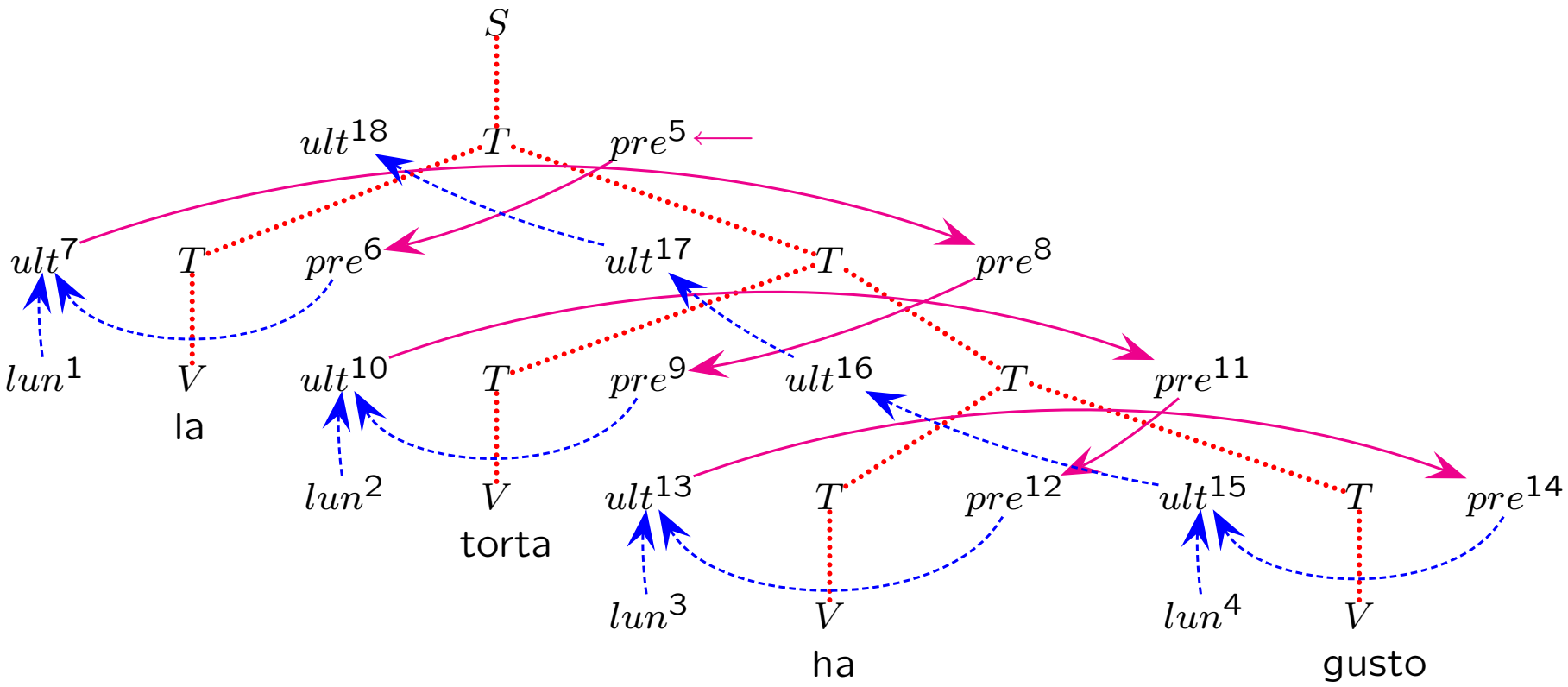
end while

end - - termina restituendo ord

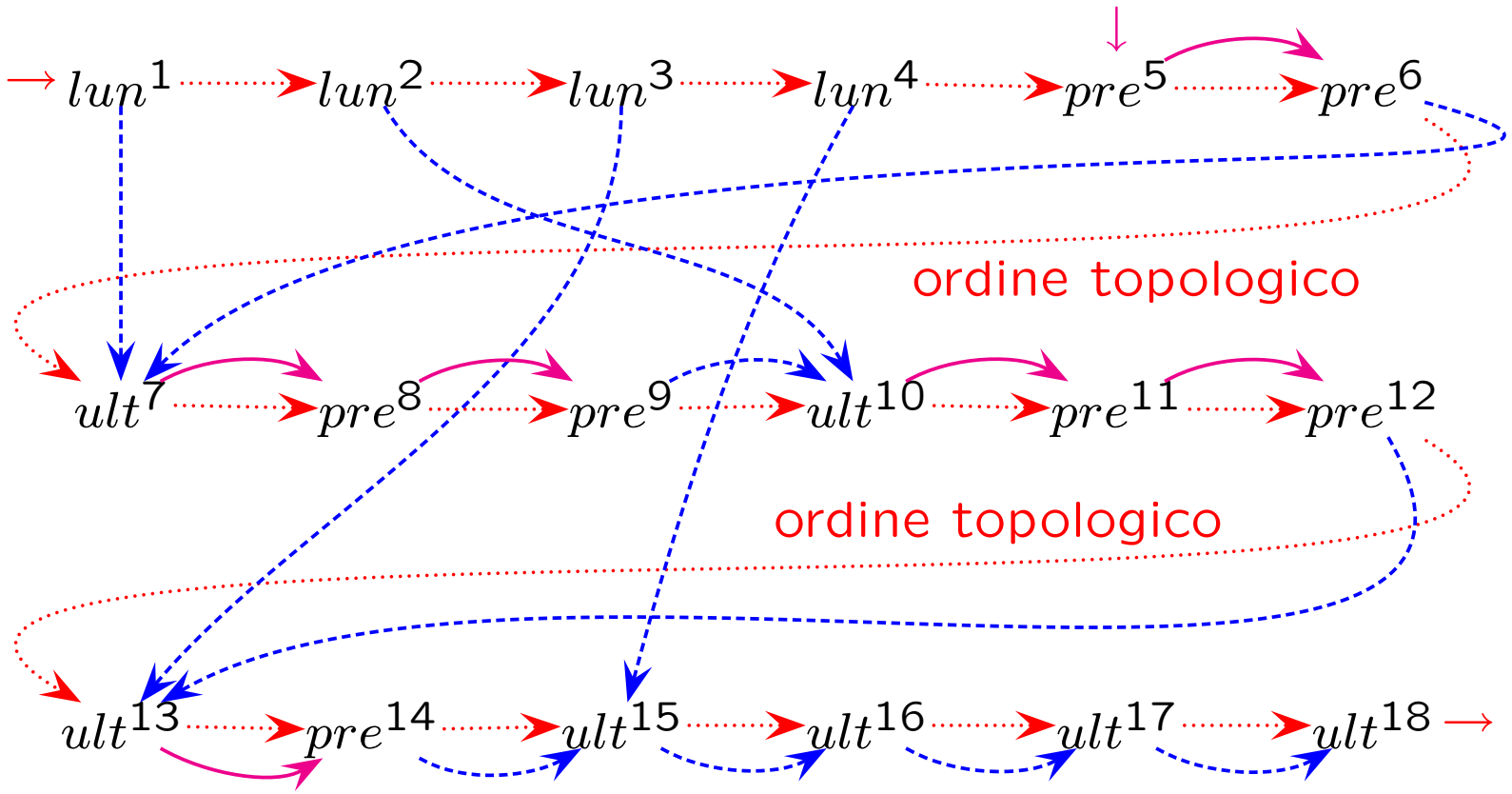
Applicando l'algoritmo al grafo di prima, un ordine topologico valido è il seguente:

$$\begin{array}{cccccc} lun^1 & lun^2 & lun^3 & lun^4 & pre^5 & pre^6 \\ ult^7 & pre^8 & pre^9 & ult^{10} & pre^{11} & pre^{12} \\ ult^{13} & pre^{14} & ult^{15} & ult^{16} & ult^{17} & ult^{18} \end{array}$$

ordine topologico di valutazione degli attributi



e in forma lineare ...



... tutte le frecce di dip. puntano in avanti !

Per il primo nodo nell'ordinamento, l'assegnamento di valore all'attributo è necessariamente costante (non essendoci predecessori). Anche altri attributi possono essere costanti, però.

Poi si calcolano gli assegnamenti di valori agli attributi seguendo l'ordine topologico, usando l'algoritmo di cui sopra per stabilirlo.

Schedulazione con scansione fissa

Questa via non è efficiente: si deve applicare l'algoritmo di ordinamento ai nodi (= attributi) dell'albero, prima di calcolare gli assegnamenti.

Ecco un valutatore più veloce: si predetermina un ordine fisso di visita (*schedulazione*) dei nodi, valido per ogni albero, in accordo con le dipendenze funzionali tra gli attributi.

Secondo problema lasciato in sospeso, in merito all'ipotesi di aciclicità della grammatica: come si fà a verificare che nessun albero possa mai presentare circuiti nel grafo delle dipendenze?

Poiché il linguaggio sorgente è infinito, non si può fare in modo esaustivo il *test di aciclicità*.

Un algoritmo per decidere se una grammatica è aciclica esiste ma è complesso (NP-completo rispetto alle dimensioni della grammatica).

Tuttavia il test di aciclicità è necessario in pratica: si danno dunque certe condizioni sufficienti per costruire la schedulazione, escludendo ipso facto la ciclicità della grammatica.

Valutazione a una sola scansione

Un valutatore semantico veloce deve visitare l'albero passando *una sola volta* su ogni nodo, calcolandone via via gli attributi pertinenti.

Si chiama *valutazione a una sola scansione* (one sweep); somiglia all'elaborazione in tempo reale.

Visita in profondità dell'albero:

1. inizia dalla radice dell'albero (assioma)
2. visita il sottoalbero t_N avente radice nel nodo N; siano $N_1, \ldots, N_r$ i figli di N
   (a) visita in profondità i sottoalberi t_1, t_2, ..., t_{r-1}, t_r, in un ordine che non è necessariamente quello naturale $1, 2, \ldots, r-1, r$, ma può essere una permutazione di questo

Prima di visitare e valutare il sottoalbero t_N, si calcolano gli attributi destri del nodo N. Al termine della visita del sottoalbero t_N si calcolano gli attributi sinistri di N.

Non tutte le grammatiche consentono di valutare gli attributi con una sola scansione.

Certe dipendenze funzionali richiedono un ordine di visita diverso da quello in profondità.

Condizioni affinché la visita in profondità sia compatibile con la valutazione

Verificabili *localmente* in modo rapido sul grafo delle dipendenze dip_p di ogni produzione p.

Nel progetto di una grammatica è spesso agevole rispettare tali condizioni, pensandoci in tempo.

Ciò permette la costruzione del valutatore semantico a una sola scansione.

grafo dei fratelli $frat_p$: relazione binaria tra simboli (non tra attributi) D_i, con $i \geqslant 1$.

Data la produzione p: $D_0 \rightarrow D_1 D_2 \ldots D_r$, con $r \geqslant 1$, i nodi di $frat_p$ sono i simboli della parte destra di p, cioè $\{D_1, D_2, \ldots, D_r\}$.

In $frat_p$ esiste l'arco $D_i \rightarrow D_j$, con $i \neq j$ e $i, j \geqslant 1$, se e solo se nel grafo delle dipendenze dip_p esiste un arco $\alpha_i \rightarrow \beta_j$ tra un attributo qualsiasi α del simbolo D_i e uno qualsiasi β del simbolo D_j.

Nota: i nodi di $frat_p$ sono i simboli nonterminali della sintassi, non gli attributi della semantica.

Dunque tutti gli attributi di dip_p aventi lo stesso pedice j si fondono nel nodo D_j di $frat_p$.

Tra i due grafi dip_p e $frat_p$ vi è relazione di omomorfismo ($frat_p$ è immagine omomorfa di dip_p, essendo ottenuto fondendo nodi di dip_p).

Condizioni per l'esistenza della grammatica a una sola scansione

$$\forall\, p\colon\ \ D_0 \to D_1 D_2 \ldots D_r \qquad\qquad r \geqslant 1$$

1. Nel grafo dip_p non esiste nessun circuito.
2. Nel grafo dip_p non esiste nessun cammino $\sigma_i \to \ldots \to \delta_i$, con $i \geqslant 1$, da un attributo sinistro σ_i a uno destro δ_i, entrambi associati allo stesso simbolo D_i della parte destra di p.

3. Nel grafo dip_p non esiste nessun arco $\sigma_0 \longrightarrow \delta_i$, con $i \geqslant 1$, da un attributo sinistro del padre D_0 a uno destro di un qualsiasi figlio D_i.

4. Nel grafo $frat_p$ non esiste nessun circuito.

Spiegazioni circa le varie condizioni:

1. La condizione è necessaria affinché la grammatica risulti aciclica.
2. Se vi fosse un cammino $\sigma_i \longrightarrow \ldots \longrightarrow \delta_i$, con $i \geqslant 1$, non si potrebbe calcolare l'attributo destro δ_i prima di averne visitato il sottoalbero t_i, perché il valore dell'attributo sinistro σ_i sarebbe noto soltanto al termine della visita; ciò contrasterebbe la schedulazione scelta.

3. L'attributo δ_i non sarebbe ancora disponibile quando iniziasse la visita del sottoalbero t_i.

4. La condizione permette di ordinare topologicamente i fratelli, ossia i sottoalberi, e di organizzare la loro visita $t_1, \ldots, t_r$ in un ordine che va bene per tutte le dipendenze di dip_p. Se il grafo dei fratelli fosse ciclico, non esisterebbe una schedulazione valida per tutti gli attributi della parte destra di p.

Algoritmo per la costruzione del valutatore a una sola scansione

Si deve progettare una procedura per ogni non-terminale, i cui argomenti in ingresso siano:

- il sottoalbero con radice nel nonterminale
- gli attributi destri della radice del sottoalbero

La procedura visita il sottoalbero, ne calcola gli attributi e restituisce quelli sinistri della radice.

Si elencano ora i vari passi di costruzione della procedura di valutazione semantica.

$$\forall\, p\colon\ \ D_0 \to D_1 D_2 \ldots D_r \qquad\qquad r \geqslant 1$$

1. Costruisci un Ordine Topologico dei nonterminali $D_1, D_2, \ldots, D_r$ rispetto al grafo dei Fratelli $frat_p$, chiamato OTF (Ordinamento Topologico dei Fratelli).

2. Per ogni simbolo D_i, con $1 \leqslant i \leqslant r$, costruisci un Ordine Topologico degli attributi Destri (ereditati) del simbolo D_i, chiamato OTD.

3. Costruisci un Ordine Topologico degli attributi Sinistri (sintetizzati) del nonterminale D_0, chiamato OTS.

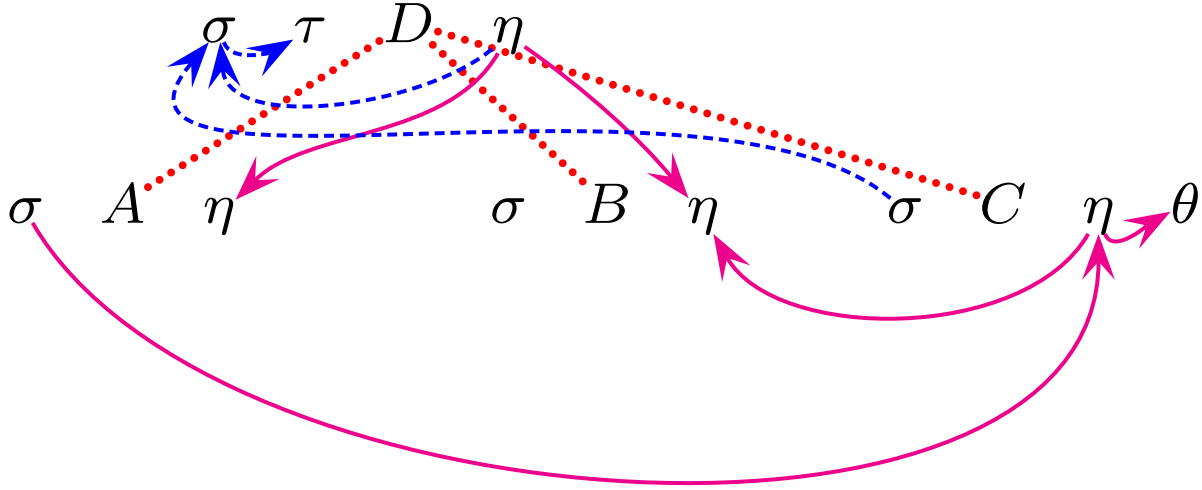
I tre ordinamenti OTF, OTD e OTS determinano la sequenza delle istruzioni della procedura.

Esempio di procedura a una sola scansione

Produzione p e grafo delle dipendenze dip_p:

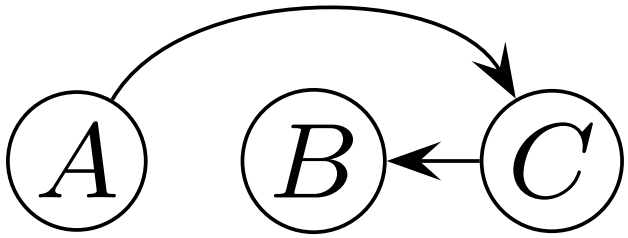
produzione

p: $D \to ABC$



Il grafo dip_p soddisfa i punti 1, 2 e 3 della condizione a una sola scansione.

Il grafo dei fratelli $frat$ è aciclico:



Gli archi del grafo sono ricavati così:

$A \to C$ dalla dipendenza $\sigma_A \to \eta_C$

$C \to B$ dalla dipendenza $\eta_C \to \eta_B$

Ciò soddisfa il punto 4 della condizione.

Ordini topologici possibili:

- grafo dei fratelli: $OTF = A, C, B$
- attributi destri di ogni figlio:
  - OTD di $A = \eta$ (c'è un solo attr.)
  - OTD di $B = \eta$ (c'è un solo attr.)
  - OTD di $C = \eta, \theta$
- attributi sinistri: OTS di $D = \sigma, \tau$

Procedura semantica di $D \rightarrow ABC$

procedure D (**in** t, η_D; **out** σ_D, τ_D)

var $\eta_A, \sigma_A, \eta_B, \sigma_B, \eta_C, \theta_C$ - - variabili attributo locali per passaggio parametri

begin - - inizia ricevendo in ingresso t e η di D

$\eta_A := f_1(\eta_D)$ - - OTD di A calcola η di A

A (t_A, η_A; σ_A) - - OTF chiama A e decora il sottoalbero di A

$\eta_C := f_2(\sigma_A)$ - - OTD di C calcola η e θ di C

$\theta_C := f_3(\eta_C)$

C (t_C, η_C, θ_C; σ_C) - - OTF chiama C e decora il sottoalbero di C

$\eta_B := f_4(\eta_D, \eta_C)$ - - OTD di B calcola η di B

B (t_B, η_B; σ_B) - - OTF chiama B e decora il sottoalbero di B

$\sigma_D := f_5(\eta_D; \sigma_C)$ - - OTS calcola σ e τ di D

$\tau_D := f_6(\sigma_D)$

end - - termina restituendo in uscita σ e τ di D

Si immagini di disporre i nodi in linea da sx verso dx nell'ordine di visita determinato (secondo il grafo $frat$), e di piazzare gli attributi:

- ereditati, a sinistra del nodo cui si associano
- sintetizzati, a destra di tutti i figli, nipoti, ecc (e dei loro attr.), del nodo cui si associano

È come dire che gli attributi ereditati sono operatori *prefissi*, quelli sintetizzati *postfissi*.

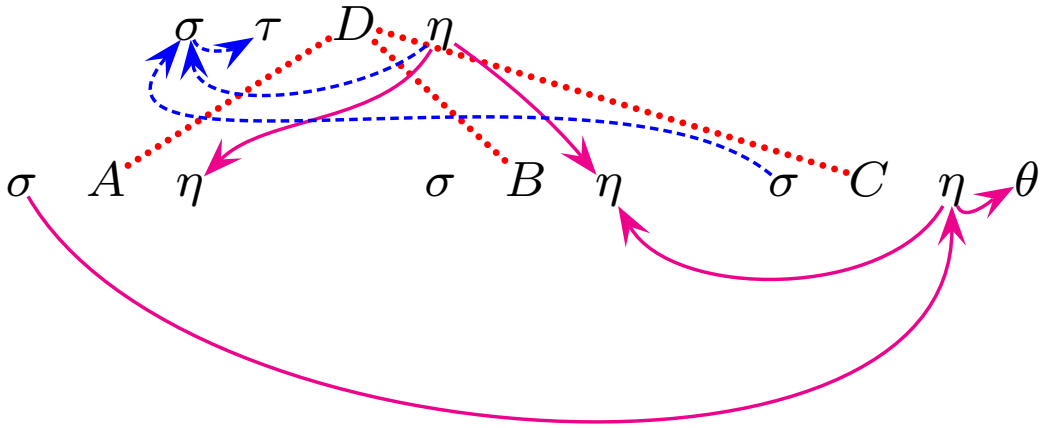
Ecco un esempio (lo stesso di prima):

produzione

$D \to ABC$

ordine di visita

$D,\ A,\ C,\ B$



L'ordine lineare prefisso-postfisso è il seguente:

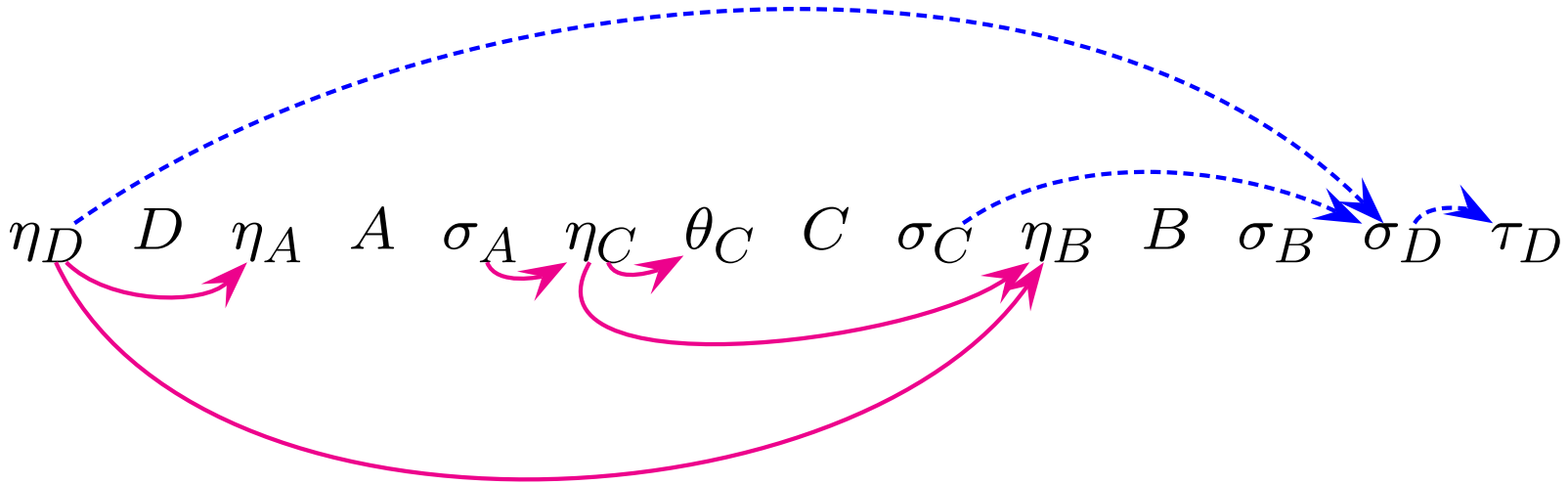
$$\eta_D\ \ D\ \ \eta_A\ \ A\ \ \sigma_A\ \ \eta_C\ \ \theta_C\ \ C\ \ \sigma_C\ \ \eta_B\ \ B\ \ \sigma_B\ \ \sigma_D\ \ \tau_D$$

ed esprime l'ordine di calcolo degli argomenti.

Ora si riportino le frecce dipendenza tra gli attributi così ordinati linearmente.
Dato che gli attributi ereditati vanno calcolati non appena si entra nel nodo, e quelli sintetizzati quando tutti i sottoalberi del nodo sono stati completamente decorati, la condizione a una sola scansione riesce soddisfatta se e solo se:

le frecce di dipendenza puntano verso dx !

Produzione: $D \longrightarrow ABC$, ordine di visita: D, A, C, B



Le frecce puntano tutte verso destra ! Dunque scandire e calcolare gli argomenti da sx verso dx è compatibile con tutte le dipendenze.

Analisi sintattica e semantica integrate

Qualora la valutazione semantica possa essere svolta direttamente dal parsificatore, integrare analisi sintattica e semantica è un metodo che risulta molto efficiente.

Si presta bene per traduzioni poco complesse.

Ci sono però varie situazioni da considerare, a seconda della natura del linguaggio sorgente:

- *linguaggio sorgente regolare*: analisi lessicale con attributi lessicali
- *sintassi di tipo* $LL(k)$: parsificatore a discesa ricorsiva con attributi
- *sintassi di tipo* $LR(k)$: parsificatore a spostamento e riduzione con attributi

Il primo caso (regolare) è sostanzialmente noto dagli strumenti SW *flex* o *lex*.

Il secondo caso è facile da realizzare a mano con attributi solo di tipo sinistro.

Il terzo caso è noto dagli strumenti SW *bison* o *yacc*: gli attributi destri sono esclusi o fortemente limitati nelle loro dipendenze.

Parsificatore a discesa ricorsiva con attributi

Ipotesi di lavoro:

- sintassi adatta all'analisi discendente det.
- grammatica adatta a una sola scansione
- inoltre le dipendenze funzionali tra attributi devono soddisfare *restrizioni supplementari*

L'algoritmo a una scansione visita i sottoalberi:

$$t_1, \ldots, t_r$$

associati alla produzione $D_0 \rightarrow D_1 \ldots D_r$, in un ordine anche diverso da quello naturale.

L'ordine è topologico, così da rispettare le dipendenze funzionali tra gli attributi dei nodi $1, \ldots, r$.

Invece, il parsificatore a discesa ricorsiva costruisce l'albero nell'ordine naturale.

Il sottoalbero t_j $(1 \leqslant j \leqslant r)$ verrà dunque costruito *dopo* i sottoalberi $t_1, t_2, \ldots, t_{j-2}, t_{j-1}$.

Vanno vietate le dipendenze funzionali che imporrebbero una visita dei sottoalberi in un ordine con permutazione diversa da $1, 2, \ldots, r-1, r$.

Condizione L (left-to-right) per disc. ric.

$$\forall\, p\colon\ \ D_0 \to D_1 \ldots D_r \qquad\qquad r \geqslant 1$$

1. la cond. a una sola scansione sia soddisfatta

2. nel grafo dei fratelli $frat_p$ non esistano archi

$$D_j \to D_i \qquad\qquad \text{con } j > i$$

La condizione 2 vieta che un attributo destro del nodo D_i dipenda da un attributo (destro o sinistro) di un nodo D_j posto alla destra di D_i.

Dunque l'ordine naturale di scansione dei sottoalberi, cioè $1, 2, \ldots, r-1, r$, soddisfa le precedenze che condizionano la visita dei sottoalberi.

Proprietà: se una grammatica è tale che

- la sintassi soddisfi la condizione $LL(k)$, e
- le regole semantiche soddisfino la cond. L

si può costruire un parsificatore deterministico a discesa ricorsiva (*analizzatore sintattico semantico*) che calcola gli attributi.

Esempio di analizzatore sintattico-semantico a discesa ricorsiva

Converte un numero frazionario minore di 1 dalla base 2 alla base 10.

Linguaggio: $L = \bullet(0 \mid 1)^*$

Traduzione: $\bullet 01_{due} \Rightarrow 0,25_{dieci}$

Grammatica con regole semantiche:

sintassi	*funzioni semantiche*		
$N_0 \to \bullet D_1$	$v_0 := v_1$	$l_1 := 1$	
$D_0 \to B_1 D_2$	$v_0 := v_1 + v_2$	$l_1 := l_0$	$l_2 := l_0 + 1$
$D_0 \to B_1$	$v_0 := v_1$	$l_1 := l_0$	
$B_0 \to 0$	$v_0 := 0$		
$B_0 \to 1$	$v_0 := 2^{-l_0}$		

Gli attributi v e l sono associati ai gruppi di nonterminali N, D, B e D, B, rispettivamente.

Attributi e loro significato:

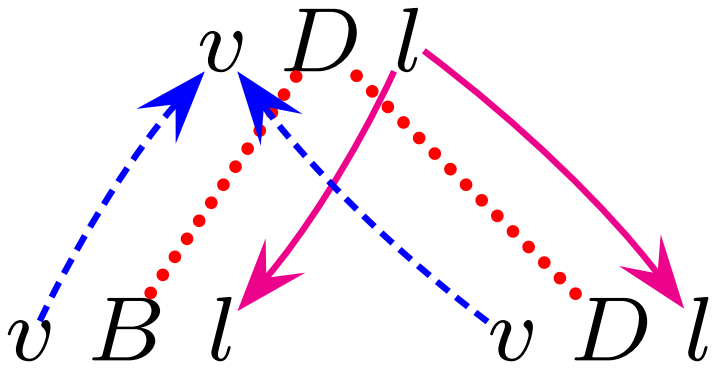
nome	*significato*	*tipo*	*nonterm. assoc.*
v	valore	sinistro	N, D, B
l	lunghezza	destro	D, B

Il valore di un bit è pesato con esponente negativo pari alla distanza dal punto frazionale.
La sintassi è deterministica $LL(2)$ (verificare ...).

Verifica della condizione L per ogni prod.

- $N \rightarrow \bullet D$: il grafo dip delle dipendenze ha soltanto l'arco $v_1 \rightarrow v_0$, pertanto:
  1. nel grafo non esiste nessun circuito
  2. non esiste nessun cammino da attributo sinistro v a destro l dello stesso figlio
  3. non esiste nessun arco da attributo v del padre ad attributo destro l di un figlio
  4. il grafo dei fratelli $frat$ è privo di archi

- $D \to BD$: il grafo delle dipendenze della prod.



dove le frecce continue e tratteggiate puntano verso l'attributo destro l e sinistro v, rispettivamente, ha le proprietà seguenti

- non ha circuiti
- non ha nessun cammino da attributo sinistro v a destro l di uno stesso figlio
- non ha nessun arco da attributo sinistro v del padre ad attributo destro v di un figlio
- il grafo dei fratelli $frat$ è privo di archi

- $D \rightarrow B$: idem come sopra ($D \rightarrow BD$)

- $B \rightarrow 0$: il grafo delle dipendenze della produzione è privo di archi

- $B \rightarrow 1$: il grafo delle dipendenze della produzione ha soltanto l'arco $l_0 \rightarrow v_0$ che sia compatibile con una sola scansione

Procedura semantica:

- parametri in ing.: attributi destri del padre
- parametri in usc.: attributi sinistri del padre
- variabili: $cc1$ e $cc2$ indicano il *terminale corrente* e *successivo*, rispettivamente, e ci sono alcune variabili locali (con tipo conforme all'attributo corrispondente) per passare attributi alle chiamate interne di altre procedure

La funzione "`leggi`" aggiorna $cc1$, terminale corrente, e $cc2$, successivo (si deve ricorrere alla prospezione, la sintassi è $LL(2)$ ma non $LL(1)$):

procedure N (**in** $\emptyset$; **out** v_0)

var l_1 - - variabili attributo locali

begin
    if ($cc1 =$ '•') - - verifica insieme guida
        then leggi - - leggi nuovo terminale
        else errore - - caso di errore
    end if
    $l_1 := 1$ - - calcola l_1 di D
    D (l_1; v_0) - - chiama D
end

```
procedure D (in l0; out v0)

var v1, v2, l2                    - - variabili attributo locali

begin
    case cc2 of                   - - prospeziona il nastro di ingresso
        '0','1': begin            - - caso D → BD
            B (l0; v1)            - - chiama B
            l2 := l0 + 1          - - calcola l2 di B
            D (l2; v2)            - - chiama D
            v0 := v1 + v2         - - calcola v0 di D
        end
        '⊣': begin                - - caso D → B
            B (l0; v1)            - - chiama B
            v0 := v1              - - calcola v0 di D
        end
        otherwise errore          - - caso di errore
    end case
end
```

```
procedure B (in l_0; out v_0)

begin
    case cc1 of                    - - verifica insieme guida

        '0': v_0 := 0              - - caso B → 0

        '1': v_0 := 2^{-l_0}       - - caso B → 1

        otherwise  errore          - - caso di errore

    end case
end
```

Per lanciare il programma si chiama la procedura sintattico-semantica associata all'assioma.

Vari miglioramenti ovvi per un programmatore.

Applicazioni tipiche delle gram. con attr.

- Controlli semantici (p. es. verifica dei tipi).
- Generazione di codice (p. es. assembler).
- Analisi sintattica guidata dalla semantica.

Controlli semantici

Il linguaggio formale L_F definito dalla sintassi è soltanto una grossolana approssimazione per eccesso del linguaggio tecnico L_T da compilare.

Vale l'inclusione:

$$\underbrace{L_F}_{\text{libero dal contesto}} \supset \underbrace{L_T}_{\text{famiglia più complessa contestuale}}$$

Si sa che le sintassi di tipo contestuale (tipo 1 o 0) non sono utilizzabili in pratica.

Ci si deve accontentare dell'approssimazione fornita da sintassi libere (se non regolari).

Si pensi a un linguaggio L_T di programmazione.

Le frasi di L_F sono *sintatticamente corrette* ma possono violare prescrizioni del manuale:

- compatibilità tra i tipi degli operandi di espressioni o assegnamenti (type checking)
- corrispondenza tra parametri formali e attuali di procedura (parameter checking)
- corrispondenza tra dichiarazione e uso di variabile, e regole di visibilità (scope rule)

Controllo delle prescrizioni di manuale mediante regole semantiche:

- Le regole semantiche calcolano degli attributi booleani, detti anche *predicati semantici*.
- Se la prescrizione è violata, la valutazione semantica produce predicato falso.
- I predicati semantici possono a loro volta dipendere da altri attributi che rappresentano proprietà varie del testo sorgente.

Esempio: la concordanza tra la dichiarazione di variabile e il suo uso in un assegnamento.

Nel testo la dichiarazione può essere molto distante dagli assegnamenti che ne fanno uso.

Il tipo secondo cui è dichiarata la variabile è posto in un attributo complesso, chiamato *tabella dei simboli* (TS) o *ambiente*.

La TS viene propagata sull'albero fino ai punti dove la variabile è usata negli assegnamenti.

Propagare la TS sarebbe inefficiente se andasse fatto per davvero, ma è soltanto concettuale.

In pratica nel compilatore la TS è realizzata come struttura dati (o come oggetto) globale visibile da ogni funzione semantica.

L'esempio seguente schematizza la creazione di TS e l'impiego per controllare le variabili usate.

Tabella dei simboli e controllo dei tipi

Di seguito si mostra un esempio semplice di controllo dei tipi mediante tabella dei simboli, che contempla le due comuni funzioni generali:

- Dichiarazione di variabili scalari e vettori.
- Uso di tali variabili negli assegnamenti.

Perscrizioni per la correttezza semantica:

1. è vietato (ri)dichiarare due o più volte la variabile (anche se la dich. fosse identica)

2. è vietato usare la variabile prima di averla dichiarata (né dichiararla dopo)

3. sono permessi assegnamenti soltanto tra variabili di tipo scalare (o elementi di vettore), oppure tra vettori di eguale dimensione

La sintassi deve distinguere la dichiarazione della variabile dall'uso nell'assegnamento:

- La TS ha per chiave di ricerca il nome (l'identificatore) n della variabile.

- La TS contiene il descrittore $desc$ della variabile, con tipo (scalare o vettore) e dimensione.

In costruzione la TS è attributo sinistro t.

Il predicato dd (di tipo sinistro) denuncia l'esistenza di una *doppia dichiarazione*.

L'attributo t è propagato in tutto il programma.

Le parti sinistra L e destra R di un assegnamento hanno lo stesso attributo $desc$, che ha la funzione di specificare:

- Una variabile, senza indice o con indice (elemento di array), oppure una costante.

- Se il nome $\notin TS$, il descrittore denuncia errore.

Regola supportata dall'assegnamento:

- Controlla compatibilità e dichiarazione.

- Calcola il predicato ae *assegnamento errato*.

Nel seguito le produzioni non hanno pedici numerici, per brevità, ma si immagini di applicarli per associare correttamente nonterm. e attr.

sintassi	*funzioni semantiche*	- - *commento*
$S \rightarrow P$	$t_1 := \emptyset$	- - inizializza tab. sim.
$P \rightarrow DP$	$t_2 := inserisci(t_0, n_1, desc_1)$	- - mette desc. in tab.
$P \rightarrow AP$	$t_1 := t_0$ $t_2 := t_0$	- - propaga tab. sim.
$P \rightarrow \varepsilon$		
$D \rightarrow id$	$dd_0 := presente(t_0, n_{id})$ **if** $(\neg dd_0)$ **then** $desc_0 :=$ 'sca' **end if** $n_0 := n_{id}$	- - dichiara var. sca.
$D \rightarrow id\ [cost]$	$dd_0 := presente(t_0, n_{id})$ **if** $(\neg dd_0)$ **then** $desc_0 := (\text{'}vet\text{'}, v_{cost})$ **end if** $n_0 := n_{id}$	- - dichiara var. vet.
$A \rightarrow L := R$	$ae_0 := \neg\langle desc_1$ è compatibile con $desc_2\rangle$	- - verifica dei tipi
$L \rightarrow id$	$desc_0 := \langle$tipo di n_{id} in $t_0\rangle$	- - ass. var. sca. o vet.
$L \rightarrow id\ [id]$	**if** $\big(\langle$tipo di n_{id_1} in $t_0\rangle =$ 'vet' $\wedge \langle$tipo di n_{id_2} in $t_0\rangle =$ 'sca'$\big)$ **then** $desc_0 := \langle$descr di n_{id_1} in $t_0\rangle$ **else** errato **end if**	- - ass. var. con indice
$R \rightarrow id$	$desc_0 := \langle$tipo di n_{id} in$t_0\rangle$	- - uso var. sca. o vet.
$R \rightarrow cost$	$desc_0 :=$ 'sca'	- - uso di costante
$R \rightarrow id\ [id]$	**if** $\big(\langle$tipo di n_{id_1} in $t_0\rangle =$ 'vet' $\wedge \langle$tipo di n_{id_2} in $t_0\rangle =$ 'sca'$\big)$ **then** $desc_0 := \langle$descr di n_{id_1} in$t_0\rangle$ **else** errato **end if**	- - uso var. con indice

Testo sintatticamente corretto:

$$\overbrace{a[10]}^{D_1}\ \overbrace{i}^{D_2}\ \overbrace{b}^{D_3}\ \overbrace{i := 4}^{A_4}\ \overbrace{c := a[i]}^{A_5:\, ae=true}\ \overbrace{c[30]}^{D_6}\ \overbrace{i}^{D_7:\, dd=true}\ \overbrace{a := c}^{A_8:ae=true}$$

Sono stati riconosciuti vari errori semantici:

- negli assegnamenti A_5 e A_8
- nella dichiarazione di variabile D_7

Perfezionamenti e completamenti necessari in un compilatore reale:

- rendere la *diagnostica più precisa*

- discriminare i generi di errori (variabile indefinita, tipo non compatibile, dimensione di vettore errata, e altri ancora ...)

- comunicare all'autore l'indicazione del punto (numero di linea) dove l'errore si è manifestato, e magari anche la colonna

- ciascun predicato calcolato in un punto dell'albero, unitamente alla coordinata del punto, è propagato verso la radice

- una funzione apposita emette la diagnostica in maniera coerente e comprensibile

Generazione di codice macchina

Si hanno situazioni diverse a seconda della distanza semantica tra i costrutti dei linguaggi sorgente e pozzo (destinazione).

Il problema della generazione di codice macchina corretto ed efficiente è molto ampio e articolato.

Se differenze sono piccole, la traduzione è fatta direttamente dal parsificatore:

- p. es. tramite la forma infissa o polacca
- tradurre in linguaggio macchina un linguaggio di alto livello come Java o C, è più difficile

  corso (interessantissimo): ANALISI E OTTIMIZZAZIONE DEI PROGRAMMI

Traduzione a più stadi:

- uno stadio traduce un linguaggio *intermedio* in un altro, più vicino al linguaggio macchina
- il primo stadio (analizzatore sintattico) ha Java o C come linguaggio sorgente
- l'ultimo stadio produce il linguaggio macchina del processore (linguaggio assemblatore)

Gamma di linguaggi intermedi impiegati:

- in forma prefissa o postfissa
- strutturati come alberi o grafi
- oppure simili al codice assemblatore

Primo stadio (*tronco o front-end*): traduttore guidato dalla sintassi del linguaggio Java.
Gli stadi finali scelgono le istruzioni macchina, mirando a ottimizzare vari parametri, come:

- aumentare la velocità di esecuzione
- e/o ridurre il consumo di energia elettrica

Metodi: "tree pattern matching", ovvero copertura dell'albero sintattico mediante forme

Traduzione in istruzioni macchina di strutture di controllo condizionali e iterative

I costrutti di alto livello *if then else*, *while do*, *repeat until*, *case* e *switch*, *loop exit*, *break* e *continue*, ecc., vanno convertiti in *istruzioni macchina di salto*, condizionato e non.

Il traduttore ha bisogno di n*uove etichette di arrivo* delle istruzioni macchina di salto, diverse dalle etichette usate altrove.

La funzione $nuovo$, a ogni invocazione, assegna all'attributo n un intero diverso.

La traduzione di un costrutto viene effettuata mediante l'attributo generico tr.

Si usa l'operatore di concatenamento ($\bullet$) per giustapporre le traduzioni parziali alle istruzioni macchina di salto e alle nuove etichette.

Si usano etichette del tipo e397, f397, i23, ...

Grammatica del costrutto condizionale if:

sintassi	*funzioni semantiche*
$F_0 \rightarrow I_1$	$n_1 := nuovo$
$I_0 \rightarrow$ **if** $(cond)$	$tr_0 := tr_{cond}$ • 'jump-if-false' • ' e' • n_0 • ';'•
then L_1	tr_{L_1} • 'jump-uncond' • ' f' • n_0 • ';'•
else L_2	'e' • n_0 • ':'tr_{L_2}•
end if	'f' • n_0 • ':'

Il simbolo • indica concatenamento, 'e' sta per "else" e 'f' sta per "finish".

Le traduzioni della condizione booleana $cond$ e delle altre frasi (rami "then" ed "else") sono specificate in regole omesse (attributo tr).

Si suppone che tali regole mettano ';' (separatore) al termine di ogni frase tradotta.

Traduzione di un frammento di programma condizionale (supponendo che $nuovo$ restituisca 7):

codice sorgente	*codice tradotto*
if $(a > b)$	$trad(a > b)$;
	jump-if-false e7;
then $a := a - 1$	$trad(a := a - 1)$;
	jump-uncond f7;
else $a := b$	e7: $trad(a := b)$;
end if	f7: - - seguito del prog.

Grammatica del costrutto iterativo while

sintassi	*funzioni semantiche*
$F_0 \rightarrow W_1$	$n_1 := nuovo$
$W_0 \rightarrow$ **while** $(cond)$	$tr_0 :=$ 'i' $\bullet\, n_0 \bullet$ ':' $\bullet\, tr_{cond} \bullet$
	'jump-if-false' $\bullet$ ' f' $\bullet\, n_0 \bullet$ ';' $\bullet$
L_1	$tr_{L_1} \bullet$
	'jump-uncond' $\bullet$ ' i' $\bullet\, n_0 \bullet$ ';' $\bullet$
end while	'f' $\bullet\, n_0 \bullet$ ':'

Il simbolo $\bullet$ indica concatenamento, 'i' sta per "iterate" e 'f' sta per "finish".

Traduzione di un frammento di programma iterativo (supponendo che $nuovo$ restituisca 8):

codice sorgente	*codice tradotto*
while $(a > b)$	i8: $trad(a > b)$;
	jump-if-false f8;
$a := a - 1$	$trad(a := a - 1)$;
	jump-uncond i8;
end while	f8: - - seguito del prog.

Analisi sintattica guidata dalla semantica

Nello schema classico l'analisi sintattica precede quella semantica; sono dunque indipendenti.

Quando la sintassi concreta è ambigua, la parsificazione produce più alberi sintattici, tra cui l'analisi semantica dovrebbe in seguito selezionare quelli semanticamente validi.

In campo tecnico tale evenienza è rara: di solito i linguaggi sono progettati in modo da renderne deterministica la sintassi (non ambigua).

Nel trattamento dei testi in lingua naturale (italiano, inglese, ecc) la situazione si ribalta: lì la sintassi è spesso molto ambigua, senza rimedio.

corso (affascinante): TRATTAMENTO DEL LINGUAGGIO NATURALE

In un linguaggio tecnico ben progettato le frasi *non sono ambigue semanticamente*, cioè hanno un solo significato (= una sola traduzione).

Spesso si può eliminare l'incertezza tra i vari alberi sintattici ammissibili già durante la parsificazione, sfruttando le varie informazioni semantiche disponibili al compilatore.

Con riferimento all'analisi $LL(k)$ discendente:

- se il parsificatore non sa scegliere tra due alternative, aventi insiemi guida sovrapposti
- risolve il dubbio consultando il valore di un attributo semantico, detto *predicato guida*
- il predicato guida è stato calcolato prima di allora dall'analizzatore sintattico-semantico

Gli attributi sono ripartiti in due insiemi:

1. I predicati guida e gli attributi da cui essi dipendono, che vanno valutati durante la parsificazione.
2. Gli attributi restanti, che vanno valutati dopo avere costruito l'albero sintattico, unico.

L'insieme (1) deve soddisfare la condizione L.

Il predicato guida sarà così già disponibile quando andrà fatta la scelta tra due produzioni alternative per espandere il nonterminale:

$$D_i \qquad 1 \leqslant i \leqslant r$$

Nella produzione:

$$D_0 \rightarrow D_1 \ldots D_i \ldots D_r$$

l'albero sintattico sarà già stato costruito dalla radice fino ai sottoalberi $D_1 \ldots D_{i-1}$.

Per la condizione L, il predicato guida può dipendere dagli attributi seguenti:

- quelli destri del padre D_0
- quelli destri o sinistri dei simboli che, nella parte destra della produzione, precedono la radice del sottoalbero figlio D_i da costruire

Esempio di linguaggio senza interpunzione

Nel linguaggio PLZ-SYS, progettato negli anni '70 (da IBM) per un microprocessore di poche risorse, mancano le virgole e gli altri segni di interpunzione.

Si ha pertanto forte ambiguità nella lista dei parametri formali di procedura.

Vi sono tre possibili interpretazioni degli argomenti (parametri formali) della procedura P.

P proc $(X\ Y\ T1\ Z\ T2)$

1. X ha tipo Y
   $T1$ e Z hanno tipo $T2$
2. X e Y hanno tipo $T1$
   Z ha tipo $T2$
3. $X, Y, T1$ e Z hanno tipo $T2$

Si adottano le regole convenzionali seguenti:

- Ogni argomento è specificato con il suo tipo, e più argomenti possono condividere il tipo.

- Le dichiarazioni dei tipi devono precedere le dichiarazioni delle procedure.

Siano le dichiarazioni di tipo precedenti per P:

$$\text{type } T1 = \text{record} \ldots \text{end}$$

$$\text{type } T2 = \text{record} \ldots \text{end}$$

Caso 1: è escluso perché Y non è un tipo, mentre $T1$ non è una variabile.

Similmente si può escludere il caso 3.

Resta dunque solo il caso 2, quello valido.

Così l'ambiguità sintattica è stata eliminata, mediante il ricorso a informazioni semantiche.

Ora si mostra come sfruttare la conoscenza delle dichiarazioni di tipo per guidare l'analizzatore sintattico-semantico tra i casi 1, 2 e 3.

Le parti rilevanti della sintassi della parte dichiarativa D sono le seguenti:

1. T, cioè le dichiarazioni di tipo

2. I, cioè le testate di procedura (qui non ne interessano i corpi esecutivi)

La semantica inserisce i descrittori di tipo nella tabella dei simboli t, come attributi sinistri.

Il simbolo n è il nome o la chiave di ricerca (nella tabella dei simboli) di un identificatore.

Al termine dell'analisi delle dichiarazioni di tipo, la tabella dei simboli viene propagata verso le parti successive del programma.

L'attributo t è copiato nell'attributo destro td per la propagazione verso il basso dell'albero.

Il descrittore $desc$ del tipo di ogni identificatore consente al parsificatore di scegliere la produzione corretta, anche quando c'è un'alternativa non risolubile in modo puramente sintattico.

Si adottano varie semplificazioni:

- l'ambiente di visibilità degli oggetti dichiarati (tabella dei simboli) è unico
- si modella solo il tipo record, non ulteriormente specificato (id "record")
- si omette il controllo di doppia dichiarazione
- si omette di inserire in tabella la dichiarazione di procedura (e dei relativi argomenti)

sintassi	*funzioni semantiche*	- - *commento*
$D_0 \rightarrow T_1\ I_2$	$td_2 := t_1$	- - tab. sim. copiata e propagata
$T_0 \rightarrow type\_id_1 =$ **record . . . end** T_2	$t_0 := inserisci(t_2, n_1,$ 'record')	- - inserisce desc. in tab. sim.
$T_0 \rightarrow \varepsilon$	$t_0 := \emptyset$	- - inizializza tab. sim.
$I_0 \rightarrow proc\_id_1\ (L_2)\ I_3$	$td_2 := td_0$ $td_3 := td_0$	- - passa tab. sim. a L_2 (lista param.) - - e a I_3 (procedura successiva)
$I_0 \rightarrow \varepsilon$		
$L_0 \rightarrow V_1\ type\_id_2\ L_3$	$td_1 := td_0$ $td_3 := td_0$	- - passa tab. sim. a V_1 (param.) - - e a L_3 (resto della lista param.)
$L_0 \rightarrow \varepsilon$		
$V_0 \rightarrow var\_id_1\ V_2$	$td_1 := td_0$ $td_2 := td_0$	- - passa tab. sim. a var_id_1 (param.) - - per la verifica semantica - - e a V_2 (altri param. stesso tipo)
$V_0 \rightarrow var\_id_1$	$td_1 := td_0$	- - passa tab. sim. a var_id_1 (param.) - - per la verifica semantica
$type\_id_0 \rightarrow \ldots$		- - id generico
$proc\_id_0 \rightarrow \ldots$		- - id generico
$var\_id_0 \rightarrow \ldots$		- - id generico

La coppia di produzioni alternative che espandono V vìola la condizione $LL(2)$ (verifica).
Gli identificatori di tipo e variabile sono sintatticamente indistinguibili ($type\_id$ e var_id sono ambedue identificatori qualunque).
Il parsificatore effettua una verifica semantica per risolvere l'alternativa sintattica di V.
I simboli cc_1 e cc_2 rappresentano i due ultimi terminali letti, detti anche lessemi (prospezione).

#	*produzione*	#	*predicato guida*
1	$V_0 \rightarrow \underbrace{var\_id_1}_{cc_1} \; \underbrace{V_2}_{cc_2}$	1	$\langle$desc. di cc_2 in tab. $td_0\rangle$ = 'id di var.' $\wedge$
		1'	$\langle$desc. di cc_1 in tab. $td_0\rangle$ = 'id di var.'
2	$V_0 \rightarrow \underbrace{var\_id_1}_{cc_1}$	2	$\langle$desc. di cc_2 in tab. $td_0\rangle$ = 'id di tipo' $\wedge$
		2'	$\langle$desc. di cc_1 in tab. $td_0\rangle$ = 'id di var.'

I predicati sintattici 1 e 2 guidano nell'alternativa sintattica "$V \rightarrow var\_id\, V \;\mid\; var\_id$".

I predicati semantici 1' e 2' verificano se l'identificatore in cc_1 si riferisca a una variabile.

Idem, al nonterminale "$type\_id$", nella produzione "$L_0 \rightarrow V_1\ type\_id_2\ L_3$", va associato un predicato per verificare se l'identificatore in cc_2 (corrispondente a "$type\_id$") si riferisca a un tipo.

Tale verifica ha scopo solo semantico, le produzioni che espandono L sono di tipo $LL(1)$.

Concludendo, il parsificatore può costruire deterministicamente l'albero sintattico decorato.