Linguaggi Formali e Compilatori
(Formal Languages and Compilers)

prof. S. Crespi Reghizzi, prof. Angelo Morzenti
(prof. Luca Breveglieri)

Prova scritta - 6 febbraio 2009 - Parte I: Teoria

CON SOLUZIONI - A SCOPO DIDATTICO LE SOLUZIONI SONO MOLTO ESTESE E COMMENTATE VARIAMENTE - NON SI RICHIEDE CHE IL CANDIDATO SVOLGA IL COMPITO IN MODO ALTRETTANTO AMPIO, BENSÌ CHE RISPONDA IN MODO APPROPRIATO E A SUO GIUDIZIO RAGIONEVOLE

NOME:

COGNOME:

MATRICOLA: FIRMA:

ISTRUZIONI - LEGGERE CON ATTENZIONE:

- L'esame si compone di due parti:
  - I (80%) Teoria:
    1. espressioni regolari e automi finiti
    2. grammatiche libere e automi a pila
    3. analisi sintattica e parsificatori
    4. traduzione sintattica e analisi semantica
  - II (20%) Esercitazioni Flex e Bison
- Per superare l'esame l'allievo deve sostenere con successo entrambe le parti (I e II), in un solo appello oppure in appelli diversi, ma entro un anno.
- Per superare la parte I (teoria) occorre dimostrare di possedere conoscenza sufficiente di tutte le quattro sezioni (1-4), rispondendo alle domande obbligatorie.
- È permesso consultare libri e appunti personali.
- Per scrivere si utilizzi lo spazio libero e se occorre anche il tergo del foglio; è vietato allegare nuovi fogli o sostituirne di esistenti.
- Tempo: Parte I (teoria): 2h.30m - Parte II (esercitazioni): 45m

1 Espressioni regolari e automi finiti 20%

1. Sono dati due linguaggi regolari L_1 e L_2, definiti come segue:

$$L_1 \subseteq \{a,b\}^* \qquad\qquad L_2 \subseteq \{a,b,c\}^*$$

$$\begin{aligned} L_1 &= \{\, w \mid \ |w|_a \text{ è pari e} \geq 0 \ \wedge \ |w|_b = 1\} \\ L_2 &= \{\, w \mid \ |w|_a \geq 0 \ \wedge \ |w|_b = |w|_c = 1 \ \wedge \ \text{le } a \text{ precedono } b\} \end{aligned}$$

Si ricorda che $|w|_a$ indica il numero di lettere a che compaiono nella stringa w, e similmente per le altre lettere dell'alfabeto.

Si risponda alle domande seguenti:

(a) Si scrivano due espressioni regolari R_1 e R_2, non ambigue, che generano i linguaggi L_1 e L_2, rispettivamente.

(b) Si traccino i grafi stato-transizione di due automi A_1 e A_2 (non necessariamente deterministici) che riconoscono i linguaggi L_1 e L_2, rispettivamente, procedendo a scelta in modo intuitivo o algoritmico.

Soluzioni

(a) Ecco le due espressioni regolari R_1 e R_2, ottenute in modo del tutto intuitivo:

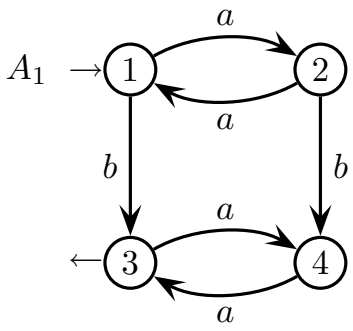
$$\begin{aligned} R_1 &= (\ a\ a\ )^*\ b\ (\ a\ a\ )^* \ \mid\ a\ (\ a\ a\ )^*\ b\ a\ (\ a\ a\ )^* \\ R_2 &= a^*\ c\ a^*\ b \ \mid\ a^*\ b\ c \end{aligned}$$

L'espressione R_1 genera stringhe contenenti esattamente una lettera b, preceduta e seguita da numeri pari ≥ 0 di lettere a oppure preceduta e seguita da numeri dispari ≥ 1 di lettere a; dunque in entrambi i casi la stringa completa contiene numero pari di lettere a. L'espressione R_2 genera la coppia $c\ b$ o $b\ c$, con numero arbitrario di lettere a, ma mai nessuna lettera a a destra della lettera b.

L'espressione R_1 non è ambigua: i due membri dell'unione generano insiemi disgiunti di stringhe; e ciascun membro non è ambiguo, giacché i fattori $(\ a\ a\ )^*$, presi individualmente, non sono ambigui, mentre la lettera b figura una sola volta e pertanto non è generata ambiguamente. Lo stesso accade all'espressione R_2 e l'argomentazione è simile (si lascia al lettore l'esame dettagliato).

(b) Per ricavare i due automi A_1 e A_2, equivalenti a R_1 e R_2, si potrebbe ricorrere a un metodo algoritmico: Thompson per versione indeterministica, o Berri-Seti per versione deterministica; ma qui è semplice procedere in modo intuitivo.

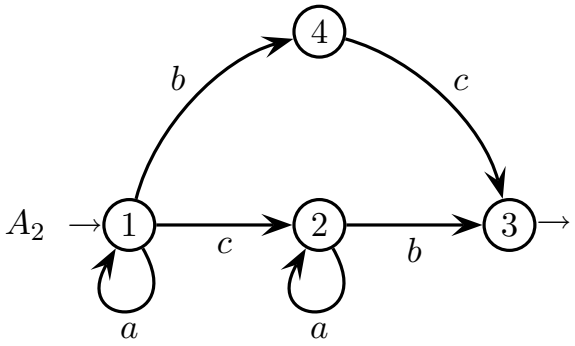
Ecco l'automa A_1 riconoscitore dell'espressione R_1:



Ecco la giustificazione dell'equivalenza tra automa A_1 ed espressione R_1. I due cicli $1 \underset{a}{\overset{a}{\rightleftarrows}} 2$ e $3 \underset{a}{\overset{a}{\rightleftarrows}} 4$ contano numeri pari di lettere a. Le transizioni $1 \xrightarrow{b} 3$ e $2 \xrightarrow{b} 4$ corrispondono al membro sinistro e destro, rispettivamente, dell'unione che figura nell'espressione R_1: la prima inserisce una lettera b tra numeri pari di lettere a; la seconda tra numeri dispari. Ciò basta a concludere.

Si può notare che l'automa A_1 è in forma ridotta (ogni stato è raggiungibile e definito) ed è deterministico. È anche facile verificare che è in forma minima. Infatti si ha quanto segue: lo stato 3 è l'unico finale e pertanto è distinguibile dagli altri tre; lo stato 4 manca di arco uscente con etichetta b e pertanto è distinguibile dagli stati 1 e 2; infine a pari etichetta b dagli stati 1 e 2 si va negli stati 3 e 4 rispettivamente, che si è già visto essere distinguibili, e pertanto anche gli stati 1 e 2 sono distinguibili. Dunque A_1 è l'automa minimo. Non è in forma naturale completa, cioè lo stato d'errore non è messo in evidenza (si ricordi peraltro che tale stato è sempre indefinito).

Ed ecco l'automa A_2 riconoscitore dell'espressione R_2:



Ecco la giustificazione dell'equivalenza tra automa A_2 ed espressione R_2. I cammini $1 \xrightarrow{c} 2 \xrightarrow{b} 3$ e $1 \xrightarrow{b} 4 \xrightarrow{c} 3$ corrispondono ai membri sinistro e destro, rispettivamente, dell'unione che figura nell'espressione R_2: la prima inserisce una lettera c tra numeri arbitrari di lettere a; la seconda mette tutte le lettere a a sinistra della lettera b. Ciò basta a concludere.

Anche l'automa A_2 è in forma ridotta, deterministico e in forma minima. Infatti si ha quanto segue: lo stato 3, finale, è distinguibile dagli altri tre stati, non finali; lo stato a, senza arco uscente con etichetta a, è distinguibile dagli stati 1 e 2, che lo hanno; infine lo stato 2, senza arco uscente con etichetta c, è distinguibile dallo stato 1, che lo ha. Dunque A_2 è l'automa minimo. Non è in forma naturale completa (non si mostra lo stato d'errore).

2. È data la grammatica G seguente, di alfabeto terminale $\{a, b\}$ e nonterminale $\{S, X\}$, lineare a sinistra (assioma S):

$$G \begin{cases} S & \to & S\,a \mid X\,a \mid \varepsilon \\ X & \to & X\,b \mid S\,b \mid \varepsilon \end{cases}$$

Si risponda alle domande seguenti:

(a) Si costruisca un automa A a stati finiti (a scelta deterministico o no) riconoscitore del linguaggio $L(G)$.

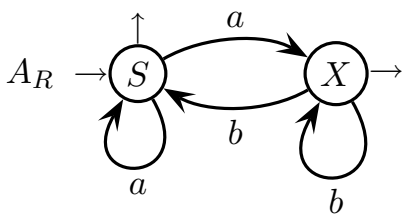
(b) (facoltativa) Se necessario, si costruisca l'automa minimo A_{min} equivalente all'automa A.

Soluzione

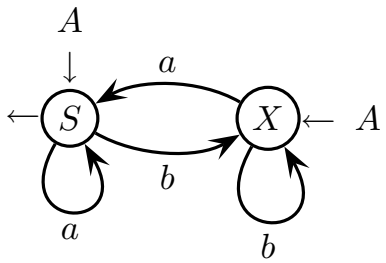
(a) Invertendo specularmente le regole delle grammatica G, si ottiene la grammatica G_R seguente, di tipo lineare a destra (assioma S):

$$G_R \begin{cases} S & \to & a\,S \mid a\,X \mid \varepsilon \\ X & \to & b\,X \mid b\,S \mid \varepsilon \end{cases}$$

La grammatica G_R genera il linguaggio riflesso $L(G_R) = L(G)^R$. Essendo G_R lineare a destra, è immediato costruire il riconoscitore a stati finiti A_R equivalente, che riconosce il linguaggio riflesso $L(A_R) = L(G_R) = L(G)^R$. Eccolo:

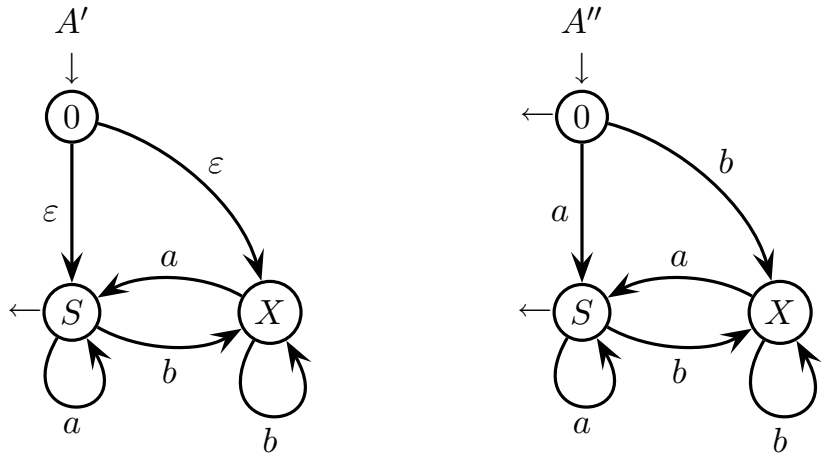


Per avere l'automa A che riconosce il linguaggio originale $L(G)$, si trasforma questo automa A_R: si scambiano stati finali e iniziali e s'inverte l'orientamento degli archi. In tale modo si ha un nuovo automa A che riconosce il linguaggio riflesso $L(A) = L(A_R)^R = \left(L(G)^R\right)^R = L(G)$. Eccolo:



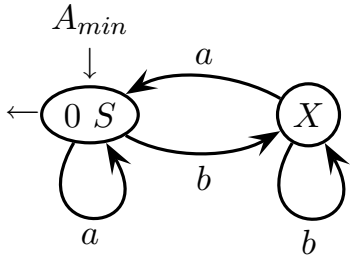
Tanto basta per rispondere, senza vincolare l'automa A al determinismo.

(b) L'automa A è indeterministico giacché ha due stati iniziali (per il resto è deterministico). Si crea un automa equivalente A' con un solo stato iniziale 0, da dove con mossa spontanea si va negli originali stati iniziali di A (sotto a sinistra)



Tagliando i due archi etichettati con ε (mediante regola di retropropagazione) si ottiene l'automa deterministico A'' (sopra a destra). Non va dimenticato che eliminando la transizione $0 \xrightarrow{\varepsilon} S$, lo stato iniziale 0 deve diventare anche finale (la marca di stato finale va retropropagata da S a 0).

L'automa A'' non è in forma minima. Se ci si trova nel gruppo di stati finali $\{\ 0,\ S\ \}$, con ingresso a si resta nel gruppo, mentre con ingresso b si va nello stesso stato X (che fa gruppo a sé stante). Pertanto gli stati 0 e S sono indistinguibili e si fondono in uno solo, chiamato 0 S (che è sia iniziale sia finale), così ottenendo l'automa minimo richiesto A_{min} (i due stati, finale e non, sono distinguibili):



Per inciso si noti che l'automa deterministico minimo A_{min} ha due stati come l'automa indeterminstico A, che pertanto è anch'esso minimo. Ciò mostra che rinunciando al determinismo l'automa minimo non è necessariamente unico.

2 Grammatiche libere e automi a pila 20%

1. Le stringhe del linguaggio L contengono numero pari ≥ 2 di gruppi di lettere a (due gruppi, quattro, sei, ecc), separati da lettere b isolate, come segue:

$$L = \{\ a^{n_1}\ b\ a^{n_2}\ b\ \ldots\ b\ a^{n_{2k}} \mid \quad k \geq 1\ \wedge\ \forall\ i \in [1, 2k] : n_i \geq 1\ \}$$

Si considerino i linguaggi L_1 e L_2, sottoinsiemi di L, descritti di seguito.

In L_1 ogni gruppo di lettere a in posizione dispari (primo gruppo, terzo, ecc) ha la stessa lunghezza del gruppo consecutivo (secondo gruppo, quarto, ecc). In formula:

$$\forall\ i \in [1, k] : n_{2i-1} = n_{2i} \qquad \text{ossia} \quad n_1 = n_2 \quad n_3 = n_4 \quad \ldots$$

Ecco un esempio di stringa di L_1:

$$a\ a\ b\ a\ a\ b\ a\ a\ a\ b\ a\ a\ a\ b\ a\ b\ a$$

In L_2 ogni gruppo di lettere a ha la stessa lunghezza del gruppo che si trova in posizione simmetrica rispetto alla lettera b centrale (la quale è individuata univocamente giacché le stringhe contengono un numero dispari di b). In formula:

$$\forall\ i \in [1, k] : n_i = n_{2k+1-i} \qquad \text{ossia} \quad n_1 = n_{2k} \quad n_2 = n_{2k-1} \quad \ldots$$

Ecco un esempio di stringa di L_2:

$$a\ a\ b\ a\ a\ a\ b\ a\ b\ a\ b\ a\ a\ a\ b\ a\ a$$

Si risponda alle domande seguenti:

(a) Si definisca una grammatica G_1 per il linguaggio L_1 e si disegni l'albero sintattico della stringa seguente:

$$a\ a\ b\ a\ a\ b\ a\ a\ a\ b\ a\ a\ a\ b\ a\ b\ a$$

(b) (facoltativa) Si caratterizzi il linguaggio intersezione $L_3 = L_1 \cap L_2$ e si dica se esso è libero o no. In caso positivo se ne dia una grammatica G_3, in caso negativo si dia una ragione, quanto meno intuitiva, della non-libertà di L_3.

Soluzione

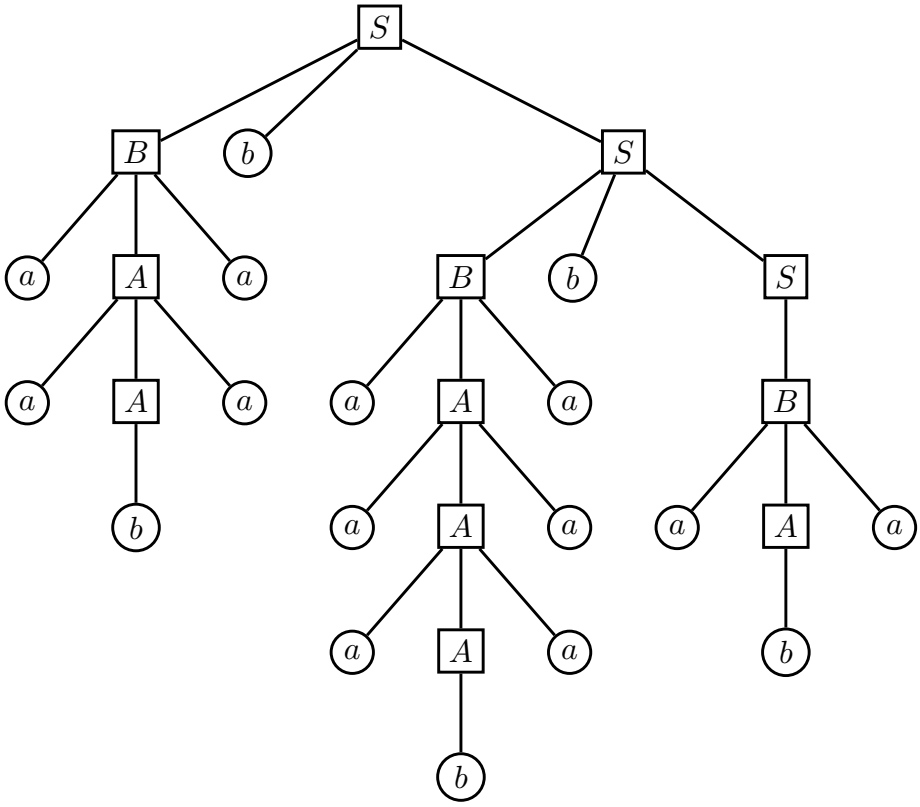
(a) Una grammatica G_1 che genera il linguaggio L_1 è la seguente (assioma S):

$$G_1 \begin{cases} S & \rightarrow & B\ b\ S \\ S & \rightarrow & B \\ B & \rightarrow & a\ A\ a \\ A & \rightarrow & a\ A\ a \\ A & \rightarrow & b \end{cases}$$

La grammatica G_1 genera una lista (non vuota) di nonterminali B separati dalla lettera b. Ciascun nonterminale B viene espanso in due gruppi consecutivi (non vuoti) di lettere a di uguale lunghezza, separati dalla lettera b. Beninteso ci possono essere altre formulazioni di G_1, più o meno naturali.

Ecco l'albero sintattico della stringa di L_1 proposta:

$a\,a\,b\,a\,a\,b\,a\,a\,a\,b\,a\,a\,a\,b\,a\,b\,a$



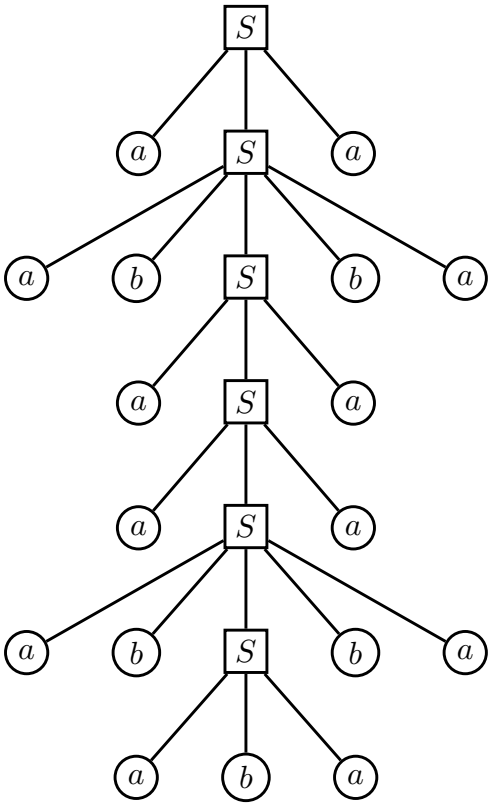
(b) Può essere utile (benché non sia richiesto dall'esercizio) guardare una grammatica G_2 per il linguaggio L_2 (assioma S):

$$G_2 \begin{cases} S & \rightarrow & a\ S\ a \\ S & \rightarrow & a\ b\ a \\ S & \rightarrow & a\ b\ S\ b\ a \end{cases}$$

Mediante la prima regola la grammatica G_2 genera due gruppi simmetrici di lettere a di uguale lunghezza; mediante la seconda regola G_2 separa i due gruppi con una lettera b isolata e termina; le due regole impediscono ai gruppi d'essere vuoti; quando ci sono più di due gruppi (ossia quattro gruppi, sei, ecc), mediante la terza regola G_2 separa i gruppi già generati inserendo due lettere b in mezzo e prepara l'inserimento di due nuovi gruppi tramite l'assioma S centrale; e così via ricorsivamente.

Ecco l'albero sintattico della stringa di L_2 proposta:

$a\,a\,b\,a\,a\,a\,b\,a\,b\,a\,b\,a\,a\,a\,b\,a\,a$



Intersecando i linguaggi L_1 e L_2 si ottengono stringhe che ne riuniscono le caratteristiche. Pertanto le stringhe del linguaggio L_3 presentano le seguenti uguaglianze di gruppi di lettere a: primo, secondo, ultimo e penultimo gruppo hanno la stessa lunghezza; terzo, quarto, terz'ultimo e quart'ultimo gruppo hanno la stessa lunghezza; e così via convergendo verso il centro della stringa; al centro si possono avere quattro gruppi consecutivi di uguale lunghezza, oppure solo due gruppi quando il numero di gruppi non è multiplo di quattro; e come caso particolare la stringa si può ridurre ad avere solo due gruppi di lunghezza identica. Ecco tre esempi di stringhe di L_3, con due, quattro e sei gruppi:

$$a\,a\,a\,b\,a\,a\,a$$

$$a\,a\,b\,a\,a\,b\,a\,a\,b\,a\,a$$

$$a\,a\,b\,a\,a\,b\,a\,a\,a\,b\,a\,a\,a\,b\,a\,a\,b\,a\,a$$

Il linguaggio L_3 non è libero. Ecco una spiegazione intuitiva costruita per confutazione: se L_3 fosse libero ammetterebbe un automa riconoscitore a pila; tale automa deve impilare la lunghezza n_i del gruppo i^{esimo}, codificandola in modo opportuno, per confrontarla sia con la lunghezza n_{i+1} del gruppo consecutivo (tale è la caratteristica di L_1) sia con la lunghezza n_{2k-i+1} del gruppo simmetrico (tale è la caratteristica di L_2). Ma controllando per esempio se $n_1 = n_2$, l'automa spila il valore n_1 e pertanto non è in grado di ricordarlo quando legge l'ultimo gruppo di lettere a, dove sarebbe necessario controllare $n_1 = n_{2k}$. Tanto basta per concludere che L_3 non è libero.

Una dimostrazione rigorosa è la seguente, basata sulle proprietà di chiusura della famiglia di linguaggi liberi rispetto all'intersezione con i linguaggi regolari e alla traduzione razionale (o regolare). Si veda il testo per le due proprietà.

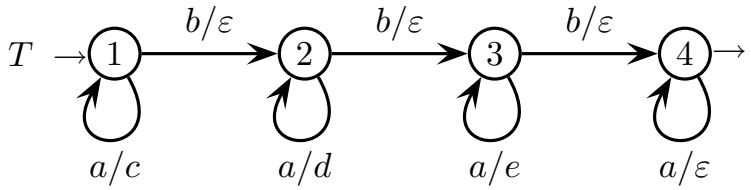
Si prenda il linguaggio regolare L_R definito tramite espressione regolare seguente:

$$L_R = a^+ \; b \; a^+ \; b \; a^+ \; b \; a^+$$

Pertanto si ha l'intersezione seguente:

$$L_4 = L_3 \cap L_R = \{ \; a^n \; b \; a^n \; b \; a^n \; b \; a^n \; | \quad n \geq 1 \; \}$$

Insomma il linguaggio L_R seleziona in L_3 tutte e sole le stringhe con quattro gruppi (non vuoti) di lettere a, che come detto prima devono avere lunghezza uguale. Si veda anche il secondo degli esempi dati sopra.
Ora si consideri il traduttore a stati finiti T seguente, con alfabeto d'ingresso e uscita $\{a, b\}$ e $\{c, d, e\}$, rispettivamente:



Il traduttore T è deterministico (giacché l'automa riconoscitore soggiacente lo è) e pertanto definisce una funzione di traduzione τ razionale (ossia a stati finiti):

$$\tau \colon L_s \rightarrow L_d$$

$$L_s = \{ \; a^p \; b \; a^q \; b \; a^r \; b \; a^s \; | \quad p, q, r, s \geq 0 \; \} = a^* \; b \; a^* \; b \; a^* \; b \; a^*$$

$$L_d = \{ \; a^p \; b \; a^q \; b \; a^r \; | \quad p, q, r \geq 0 \; \} = c^* \; d^* \; e^*$$

dove i linguaggi sorgente L_s e destinazione L_d di τ sono regolari, e τ traduce così la stringa sorgente:

$$\tau \left(\; a^p \; b \; a^q \; b \; a^r \; b \; a^s \; \right) \mapsto c^p \; d^q \; e^r \qquad\qquad p, q, r, s \geq 0$$

Si vede subito che vale la traduzione seguente:

$$L_5 = \tau \left(L_4 \right) = \{ \; c^n \; d^n \; e^n \; | \quad n \geq 1 \; \}$$

Infatti il linguaggio L_4 è un sottinsieme del linguaggio L_s. Se si prende una stringa di L_s e si applica la restrizione $p = q = r = s \geq 1$, si ottiene una stringa di L_4, e tutte le stringhe di L_4 sono ottenibili così. Insomma il traduttore T scandisce la stringa $a^n \; b \; a^n \; b \; a^n \; b \; a^n$ di L_4, con quattro gruppi di lettere a separati da tre lettere b, e traduce il primo gruppo a^n in c^n (per ogni a emette una c), e similmente il secondo e terzo gruppo a^n in d^n e e^n, rispettivamente, mentre le tre lettere b e l'ultimo gruppo a^n non danno luogo a emissione.

Giacché il linguaggio L_5 è ottenuto dal linguaggio L_3 tramite prima intersezione con linguaggio regolare e poi traduzione razionale, se per ipotesi L_3 fosse libero anche L_5 dovrebbe esserlo, in forza delle due proprietà di chiusura citate prima. Ma L_5 è il noto linguaggio a tre esponenti, che si sa bene non essere libero: la dimostrazione si trova nel libro di testo e si basa sull'uso del lemma di iterazione o "pumping lemma". Pertanto l'ipotesi che L_3 sia libero è falsa. In conclusione l'intersezione $L_1 \cap L_2$ non è un linguaggio libero, come si voleva dimostrare.

2. Si consideri un linguaggio di programmazione semplificato, tratteggiato come segue:

   - il linguaggio definisce l'identificatore alfanumerico, con sintassi analoga al linguaggio C, e la costante numerica intera o reale in virgola fissa, come

     ```
     alfa     beta_1     12230     12045,37     1245,037
     ```

   - il linguaggio ammette variabili e il nome di variabile è un identificatore
   - il linguaggio ammette i tre tipi scalari seguenti: char, int e float
   - il linguaggio ammette espressioni con variabili, costanti intere e reali, operatori infissi '+' e '∗' (addizione e moltiplicazione), dove come d'uso la moltiplicazione ha precedenza sull'addizione, e parentesi '(' e ')'
   - la frase del linguaggio è un programma completo, dotato di intestazione obbligatoria, sezione di dichiarazione facoltativa e sezione esecutiva obbligatoria
   - l'intestazione è introdotta dalla parola chiave program seguita dal nome del programma, ossia un identificatore, come

     ```
     program  nome_programma
     ```

   - la sezione di dichiarazione è una lista di dichiarazioni di variabile, separate da ';' (punto e virgola) tranne l'ultima dichiarazione
   - la singola dichiarazione di variabile è in stile linguaggio C

     ```
     nome_di_tipo  lista_di_nomi_di_variabile
     ```

     non è consentito inizializzare la variabile in sede di dichiarazione
   - la sezione esecutiva è introdotta e terminata dalle parole chiave begin e end, rispettivamente, e contiene una lista (non vuota) di istruzioni separate da ';' (punto e virgola), compresa l'ultima istruzione
   - l'istruzione è un assegnamento con '=' (uguale) da espressione a variabile, come

     ```
     a = espressione
     ```

     oppure una chiamata a procedura, con parametri attuali facoltativi, come

     ```
     nome_procedura ( lista_di_parametri_attuali )
     ```

     non ci sono istruzioni condizionali, cicli o altre strutture di controllo
   - i parametri attuali sono separati da ',' (virgola) e sono espressioni

   Ecco un breve esempio di programma:

   ```
   program SAMPLE
     char letter_1 ;
     int a, sum ;
     float r1, r2a
   begin
     r1 = sum + 1 + 3,2 * (a + sum) ;
     write (letter_1, r2a + 2) ;
   end
   ```

   Si scriva una grammatica G, non ambigua e di tipo EBNF, che genera il linguaggio di programmazione semplificato così descritto.

Soluzione

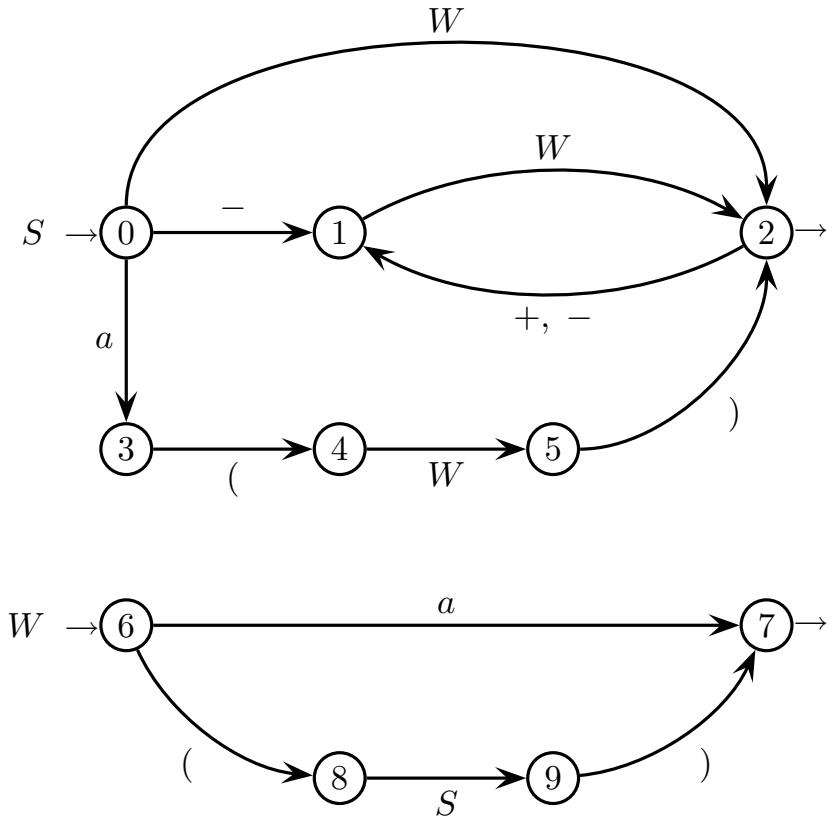
Ecco la grammatica G richiesta (assioma PROG):

intestazione e corpo del programma		
⟨PROG⟩	→	program ⟨ID⟩ ⟨BODY⟩
⟨BODY⟩	→	[⟨DECL_SECT⟩] ⟨EXEC_SECT⟩
sezione di dichiarazione		
⟨DECL_SECT⟩	→	⟨VAR_DECL⟩ (';' ⟨VAR_DECL⟩)*
⟨VAR_DECL⟩	→	⟨TYPE_NAME⟩ ⟨ID⟩ (',' ⟨ID⟩)*
⟨TYPE_NAME⟩	→	int | char | float
sezione esecutiva		
⟨EXEC_SECT⟩	→	begin ⟨STAT_LIST⟩ end
⟨STAT_LIST⟩	→	(⟨STAT⟩ ';')$^+$
⟨STAT⟩	→	⟨ASSIGN⟩ | ⟨PROC_CALL⟩
⟨ASSIGN⟩	→	⟨ID⟩ '=' ⟨EXPR⟩
⟨PROC_CALL⟩	→	⟨ID⟩ '(' [⟨PARAM_LIST⟩] ')'
⟨PARAM_LIST⟩	→	⟨EXPR⟩ (',' ⟨EXPR⟩)*
sintassi dell'espressione aritmetica (forma estesa standard)		
⟨EXPR⟩	→	⟨TERM⟩ ('+' ⟨TERM⟩)*
⟨TERM⟩	→	⟨FACT⟩ ('$*$' ⟨FACT⟩)*
⟨FACT⟩	→	⟨ID⟩ | ⟨INT_CONST⟩ | ⟨FLOAT_CONST⟩
⟨FACT⟩	→	'(' ⟨EXPR⟩ ')'
sintassi di identificatore e costanti (stile linguaggio C)		
⟨ID⟩	→	($[a,z]$ | $[A,Z]$) ($[a,z]$ | $[A,Z]$ | $[0,9]$ | '_')*
⟨INT_CONST⟩	→	$[1,9]$ $[0,9]^*$ | '0'
⟨FLOAT_CONST⟩	→	[$[1,9]$ $[0,9]^*$] ',' $[0,9]^*$ $[1,9]$

Le parentesi quadre indicano opzionalità. Una notazione come $[0,9]$ indica un elemento a scelta nell'intervallo $0 \ldots 9$, e così via per le altre similari. La grammatica G è in forma estesa (EBNF) e ha struttura modulare, dove ciascun modulo contiene elementi notoriamente non ambigui: liste con separatore, grammatica standard estesa delle espressioni aritmetiche, ecc. Pertanto G non è ambigua per costruzione. La struttura ordinata e modulare di G giustifica a sufficienza l'equivalenza con la specifica informale (in linguaggio naturale) del linguaggio data nell'esercizio.

3 Analisi sintattica e parsificatori 20%

1. Si considerino l'alfabeto terminale { 'a', '+', '−', '(', ')' } e nonterminale { S, W }. È data la seguente rete di macchine ricorsive sull'alfabeto totale (unione di terminali e nonterminali), che definisce una grammatica G (assioma S):



Si risponda alle domande seguenti:

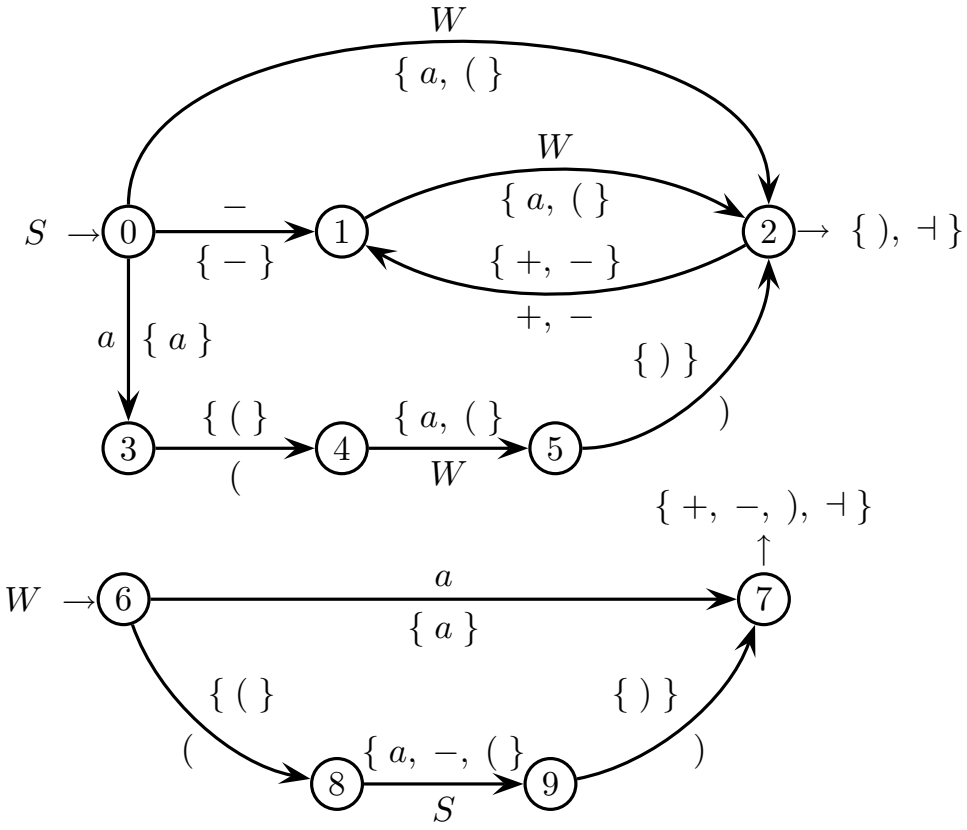
(a) Si calcolino gli insiemi guida su tutti gli archi e si dica se la grammatica G è di tipo $LL(1)$.

(b) Dove necessario, si calcolino gli insiemi guida di tipo $LL(k)$ con $k = 2$.

(c) (facoltativa) Si scriva il codice della procedura sintattica a discesa ricorsiva per il nonterminale S.

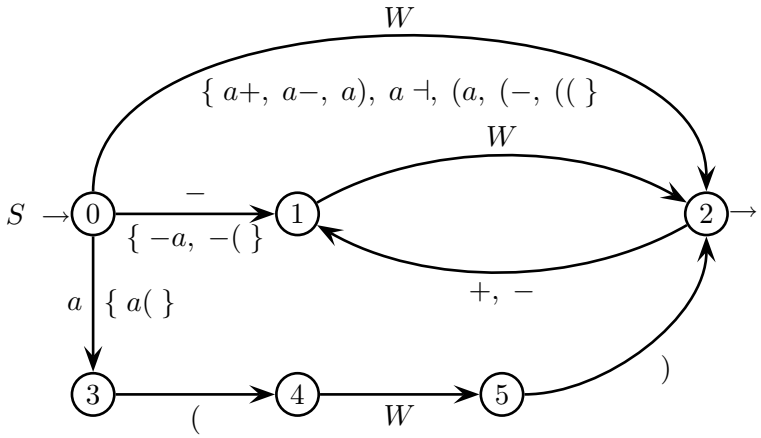
Soluzione

(a) Ecco l'analisi $LL(1)$ della rete di macchine rappresentante la grammatica G:



La grammatica G non è $LL(1)$, nello stato 0. Infatti c'è conflitto tra le transizioni $0 \xrightarrow[a,\,(]{W} 2$ e $0 \xrightarrow[a]{a} 3$: gli insiemi guida condividono il terminale a. Per il resto la grammatica G è $LL(1)$.

(b) L'analisi LL con $k = 2$ è necessaria soltanto nello stato 0. Eccola:



Ora si vede che la grammatica G è $LL(2)$: gli insiemi guida in uscita dallo stato 0 sono disgiunti (e più grandi di quelli con $k = 1$).

(c) La procedura a discesa ricorsiva per l'analisi del nonterminale S esegue la macchina di S. Eccone lo codifica in pseudo-linguaggio simile a C e Pascal:

```
procedure S
    if ( window ∈ { 'a+', 'a−', 'a)', 'a ⊣', '(a', '(−', '((' } ) then
        call W
    else if ( window ∈ { '−a', '−(' } ) then
        shift
        if ( current ∈ { 'a', '(' } ) then
            call W
        else error
    else if ( window ∈ { 'a(' } ) then
        shift
        if ( current ∈ { '(' } ) then
            shift
            if ( current ∈ { 'a', '(' } ) then
                call W
                if ( current ∈ { ')' } ) then
                    shift
                else error
            else error
        else error
    else error
end procedure
```

La variabile *window* è la finestra di prospezione estesa su due caratteri consecutivi posizionata sulla stringa da analizzare, e la variabile *current* è il primo carattere (a sinistra) di tale finestra. Il comando **shift** fa scorrere la finestra *window* un carattere verso destra e conseguentemente aggiorna la variabile *current*. Per fare il test dell'insieme guida si suppone che lo pseudo-linguaggio usato per codificare l'analizzatore sintattico disponga del tipo insieme di stringhe, come { 'str_1', 'str_2', ... }, e dell'operatore '$\in$' di appartenenza a insieme.
Per completezza, ecco la procedura sintattica che analizza il nonterminale W:

```
procedure W
    if ( current ∈ { 'a' } ) then
        shift
    else if ( current ∈ { '(' } ) then
        shift
        if ( current ∈ { 'a', '+', '(' } ) then
            call S
            if ( current ∈ { ')' } ) then
                shift
            else error
        else error
    else error
end procedure
```

Tecnicamente si potrebbero implementare *window* e *current* come variabile globale array di due caratteri e variabile globale carattere, rispettivamente, e **shift** come procedura utente ausiliaria. Per esempio:

```
char current, window[2]

procedure SHIFT
        window[0] = window[1]
        current = window[0]
        read (window[1])
end procedure
```

È un semplice dettaglio implementativo e ci sono codifiche alternative altrettanto valide. Si ricorda che la procedura assiomatica S va chiamata avendo già posizionato la finestra di prospezione all'inizio della stringa da analizzare, come segue:

```
program SYNTAX_ANALYSER
        read (window[0])
        read (window[1])
        current = window[0]
        call S
end program
```

Ecco infine la vista d'insieme dell'intero analizzatore sintattico della grammatica G:

```
program SYNTAX_ANALYSER
        read (window[0])
        read (window[1])
        current = window[0]
        call S
end program
```

```
char current, window[2]
procedure SHIFT
        window[0] = window[1]
        current = window[0]
        read (window[1])
end procedure
```

```
procedure S
        if ( window ∈ { 'a+', 'a−', 'a)', 'a ⊣', '(a', '(−', '((' } ) then
                call W
        else if ( window ∈ { 'a', '−(' } ) then
                shift
                if ( current ∈ { 'a', '(' } ) then
                        call W
                else error
        else if ( window ∈ { 'a(' } ) then
                shift
                if ( current ∈ { '(' } ) then
                        shift
                        if ( current ∈ { 'a', '(' } ) then
                                call W
                                if ( current ∈ { ')' } ) then
                                        shift
                                else error
                        else error
                else error
        else error
end procedure
```

```
procedure W
        if ( current ∈ { 'a' } ) then
                shift
        else if ( current ∈ { '(' } ) then
                shift
                if ( current ∈ { 'a', '+', '(' } ) then
                        call S
                        if ( current ∈ { ')' } ) then
                                shift
                        else error
                else error
        else error
end procedure
```

2. È data la grammatica G seguente, di alfabeto terminale $\{a, b, c\}$ e nonterminale $\{A, B, S\}$ (assioma S):

$$G \begin{cases} S & \rightarrow & A\ c\ S \mid b\ B\ S \mid b\ a\ c \\ A & \rightarrow & b\ a \\ B & \rightarrow & a\ c \end{cases}$$

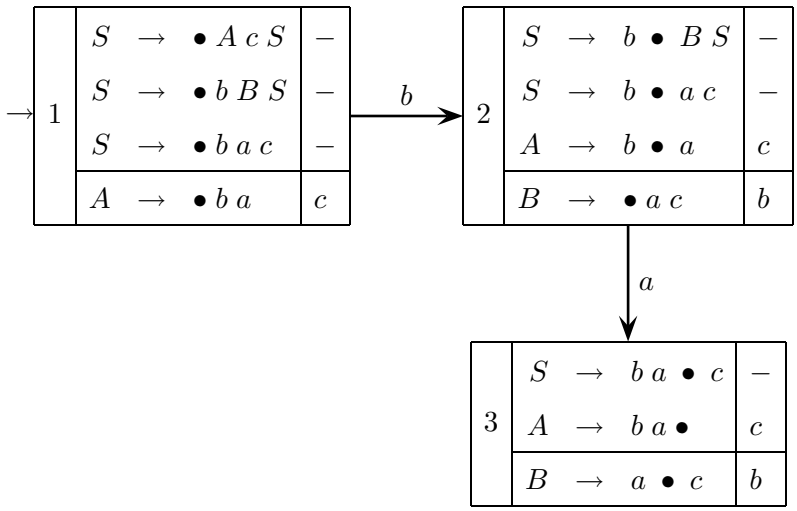
Si risponda alle domande seguenti:

(a) Si mostri che la grammatica G non è $LR(1)$, costruendo (anche solo in parte) il grafo dell'automa pilota e indicando quale condizione violano gli stati inadeguati.

(b) S'indichi qual è il linguaggio generato dalla grammatica G e si proponga un rimedio (altra grammatica o metodo di analisi alternativo) per analizzarlo in modo deterministico.

Soluzione

(a) Tracciando il grafo pilota $LR(1)$ del riconoscitore della grammatica G, si ottiene il frammento seguente che basta a risolvere la domanda (il resto non serve):

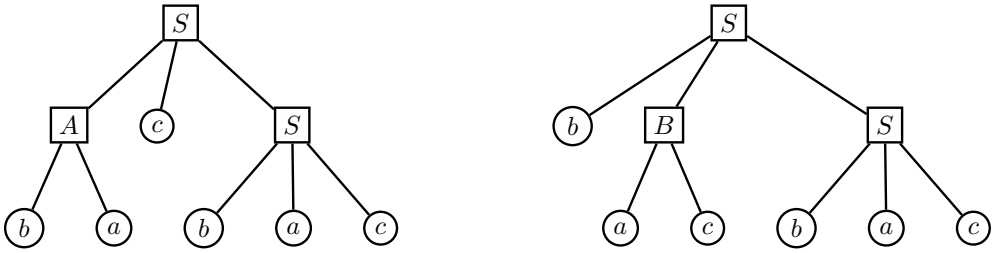


Si vede subito che il macrostato 3 contiene le tre candidate seguenti (tre graffe i terminali di prospezione):

$$S \rightarrow b\ a\ \bullet\ c\ \{-\} \qquad A \rightarrow b\ a\ \bullet\ \{c\} \qquad B \rightarrow a\ \bullet\ c\ \{b\}$$

Questo macrostato è inadeguato a causa del conflitto tra candidata di riduzione $A \rightarrow b\ a\ \bullet\ \{c\}$ e candidata di spostamento $B \rightarrow a\ \bullet\ c\ \{b\}$: la prospezione di riduzione è la lettera c e coincide con lo spostamento su c. Pertanto la grammatica G non è $LR(1)$ (naturalmente potrebbe avere altri conflitti).

Per dimostrare che la grammatica G non è $LR(1)$, senza neppure tracciare parte del grafo pilota dell'analizzatore, basterebbe comunque osservare che G è ambigua: ammette due alberi sintattici diversi per la stringa $b\ a\ c\ b\ a\ c$. Eccoli:



Equivalentemente la grammatica G ammette due derivazioni diverse per la stringa $b\ a\ c\ b\ a\ c$. Eccole (sono entrambe sinistre):

$$S \overset{S\to A\,c\,S}{\Longrightarrow} A\ c\ S \overset{A\to b\,a}{\Longrightarrow} b\ a\ c\ S \overset{S\to b\,a\,c}{\Longrightarrow} b\ a\ c\ b\ a\ c$$

$$S \overset{S\to b\,B\,S}{\Longrightarrow} b\ B\ S \overset{B\to a\,c}{\Longrightarrow} b\ a\ c\ S \overset{S\to b\,a\,c}{\Longrightarrow} b\ a\ c\ b\ a\ c$$

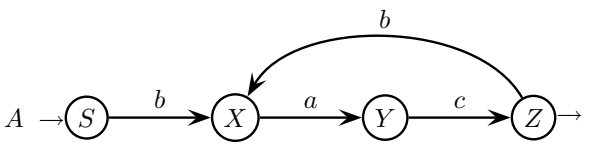
(b) Il linguaggio generato dalla grammatica G è regolare. Basta sostituire le regole terminali $A \to b\ a$ e $B \to a\ c$ nelle regole assiomatiche $S \to A\ c\ S \mid b\ B\ S \mid b\ a\ c$, che in tale modo si unificano come segue:

$$S \to b\ a\ c\ S \mid b\ \ a\ c$$

ossia in una grammatica lineare a destra, ma non unilineare perché nella parte destra figura più di un terminale. Tramite la regola di Arden si ottiene subito la soluzione seguente:

$$L(S) = (\ b\ a\ c\ )^+$$

ossia un'espressione regolare. Dunque il linguaggio $L(G) = L(S)$ è analizzabile tramite automa finito deterministico A, ottenibile dall'espressione in uno qualunque dei metodi disponibili: Thompson, Berri-Seti o anche intuizione. Poi da A è facile ricavare una grammatica unilineare a destra G_{uni} che permette l'analisi deterministica. Un'ovvia formulazione intuitiva è la seguente:



$$G_{uni}\begin{cases} S \to b\,X \\ X \to a\,Y \\ Y \to c\,Z \\ Z \to b\,X \\ Z \to \varepsilon \end{cases}$$

L'automa A ha quattro stati: S (iniziale), X, Y e Z (finale). Chiaramente si ha $L(A) = L(S)$. La grammatica unilineare a destra G_{uni} (assioma S) è equivalente ad A: le regole di G_{uni} sono le transizioni di A. Giacché l'automa A è deterministico, G_{uni} è di tipo $LL(1)$ e pertanto anche $LR(1)$, ma non $LR(0)$ a motivo della regola nulla $Z \to \varepsilon$ od osservando che il linguaggio ha prefissi.

4 Traduzione e analisi semantica 20%

1. Si considerino i linguaggi sorgente L_s e destinazione L_d seguenti, di alfabeto $\{a, b\}$ e $\{c, d\}$, rispettivamente:

$$L_s = a^*\ b^* \qquad\qquad L_d = c^*\ \mid\ d^+$$

Su tali linguaggi si definisce la traduzione sintattica τ seguente:

$$\tau \colon L_s \to L_d \qquad\qquad \tau\left(a^h\ b^k\right) \mapsto \begin{cases} c^{h-k} & \text{se } h \geq k \\ d^{h+k} & \text{se } h < k \end{cases}$$

dove si ha $h, k \geq 0$.

Ecco tre esempi di traduzione:

$$\begin{aligned} \tau\left(a^2\ b^2\right) &= \varepsilon \\ \tau\left(a^3\ b^2\right) &= c \\ \tau\left(a^2\ b^3\right) &= d^5 \end{aligned}$$

Si progetti una grammatica di traduzione G_τ, o uno schema sintattico di traduzione, non ambigua, che realizza la traduzione τ.

Soluzione

Ecco la grammatica di traduzione G_τ, in forma unita (assioma S):

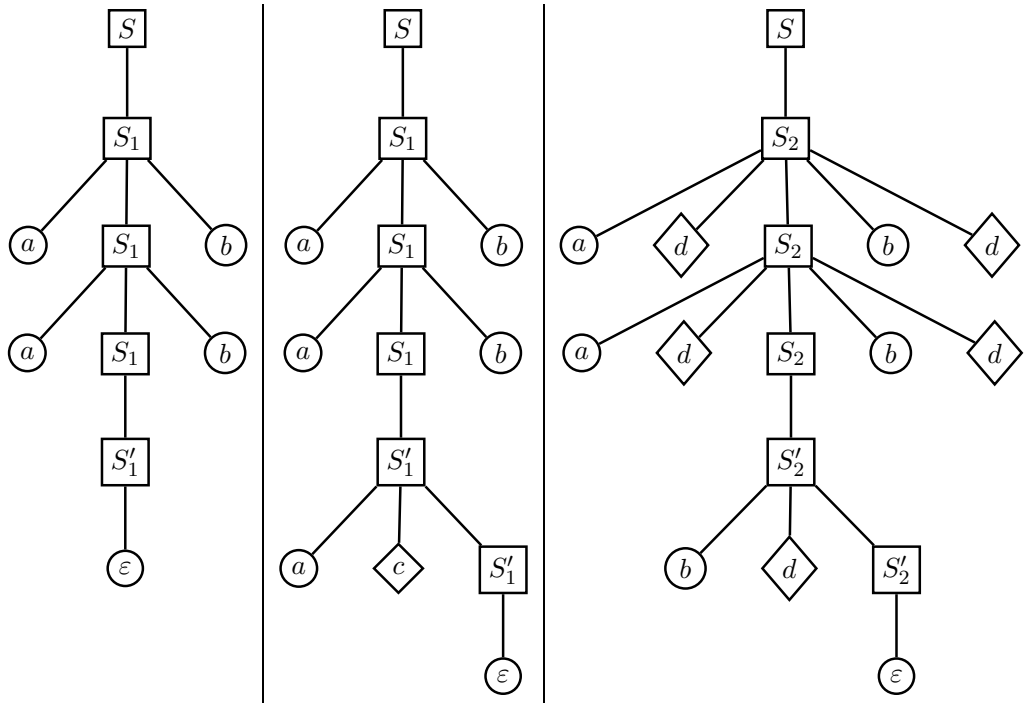
$$G_\tau \begin{cases} S & \to & S_1 \\ S & \to & S_2 \\ S_1 & \to & a\ S_1\ b \\ S_2 & \to & a\ \{\ d\ \}\ S_2\ b\ \{\ d\ \} \\ S_1 & \to & S_1' \\ S_2 & \to & S_2' \\ S_1' & \to & a\ \{\ c\ \}\ S_1' \\ S_2' & \to & b\ \{\ d\ \}\ S_2' \\ S_1' & \to & \varepsilon \\ S_2' & \to & b\ \{\ d\ \} \end{cases}$$

I nonterminali S_1 e S_2 generano le opzioni $h \geq k$ e $h < k$ della traduzione τ, rispettivamente. Si noti che il meccanismo di generazione della stringa sorgente è molto simile nelle due opzioni: per entrambe viene prima generata una struttura parentetica $a^n\ b^n$ bilanciata ($n \geq 0$), e poi si possono aggiungere in mezzo lettere a eccedenti per la prima opzione, oppure lettere b eccedenti (almeno una) per la seconda opzione. Ma il meccanismo differisce profondamente nella generazione della stringa destinazione: nella prima opzione si genera una lettera c per ciascuna lettera a eccedente; nella seconda opzione si genera una lettera c per ogni lettera a (eccedente o no) e b.

La grammatica sorgente non è ambigua: le due opzioni generano insiemi disgiunti di stringhe sorgente e ciascuna opzione, presa singolarmente, ha struttura non ambigua; si tratta infatti delle due note grammatiche che generano la struttura parentetica $a^m\ b^n$ con $m \geq n$ e $m < n$, rispettivamente. Dunque anche la grammatica di traduzione G_τ non è ambigua, come richiesto.

Sia il linguaggio sorgente L_s sia quello destinazione L_d sono regolari (in quanto dati come espressioni regolari), ma la traduzione τ richiede uno grammatica libera, non riducibile a una grammatica unilineare a destra ossia regolare. Infatti la traduzione differisce secondo l'opzione $h \geq k$ o $h < k$, e il confronto di due numeri interi di valore arbitrario, come sono h e k, non è fattibile tramite automa a stati finiti, e dunque neppure tramite grammatica di traduzione unilineare a destra (o a sinistra).

Per completezza ecco gli alberi sintattici delle tre traduzioni di esempio:



$\tau\left(a^2\ b^2\right) = \varepsilon$ | $\tau\left(a^3\ b^2\right) = c$ | $\tau\left(a^2\ b^3\right) = d^5$

Non sarebbe difficile progettare un trasduttore a pila indeterministico che realizza la traduzione τ. È più difficile progettarne uno deterministico. L'intuizione non è immediata, ma si può ricorrere all'analisi sintattica. Tuttavia si vede rapidamente che la grammatica sorgente non è di tipo LL, pertanto bisogna ripiegare sull'analisi LR. Non è comunque detto che funzioni. Si lascia il resto al lettore.

2. Una base di dati contenente delle t-uple con tre campi, come segue:

$$(\mathit{Mother\_Name},\ \mathit{Mother\_Age},\ \mathit{Child\_Age})$$

è descritta dalla sintassi G seguente (assioma S):

$$G \begin{cases} 1: & S & \rightarrow & D \\ 2: & D & \rightarrow & T\ D \\ 3: & D & \rightarrow & T \\ 4: & T & \rightarrow & (\ \mathbf{mn},\ \mathbf{ma},\ \mathbf{ca}\ ) \end{cases}$$

Ciascuno dei tre simboli terminali **mn**, **ma** e **ca** ha un attributo semantico inizializzato con il valore del terminale stesso. Ognuno dei tre attributi ha nome coincidente con il terminale di riferimento: *mn* di tipo stringa, *ma* di tipo intero e *ca* di tipo intero.

Si risponda alle domande seguenti.

(a) Negli spazi appositi (pagine successive) si scrivano le regole semantiche di una grammatica con attributi che calcola l'interrogazione SQL seguente:

$$\text{SELECT } MN \text{ FROM } D \text{ WHERE } (MA - CA) < 18$$

Il risultato è un attributo semantico, chiamato *select*, associato all'assioma S di G (e se occorre anche ad altri nonterminali).

(b) (facoltativa) Negli spazi appositi (pagine successive) si scrivano le regole semantiche che calcolano, per ogni t-upla, un attributo semantico, chiamato dif, con il valore seguente:

$$dif = MA - AVG(MA)$$

dove AVG è la media degli attributi MA. Il risultato dif è un attributo associato al nonterminale T di G (e se occorre anche ad altri nonterminali).

(c) (facoltativa) Si verifichi se la grammatica con attributi è a una scansione (one sweep), e si codifichi la procedura semantica associata al nonterminale D o se ne dia una descrizione sommaria.

Domanda (a) - calcolo interrogazione SQL:

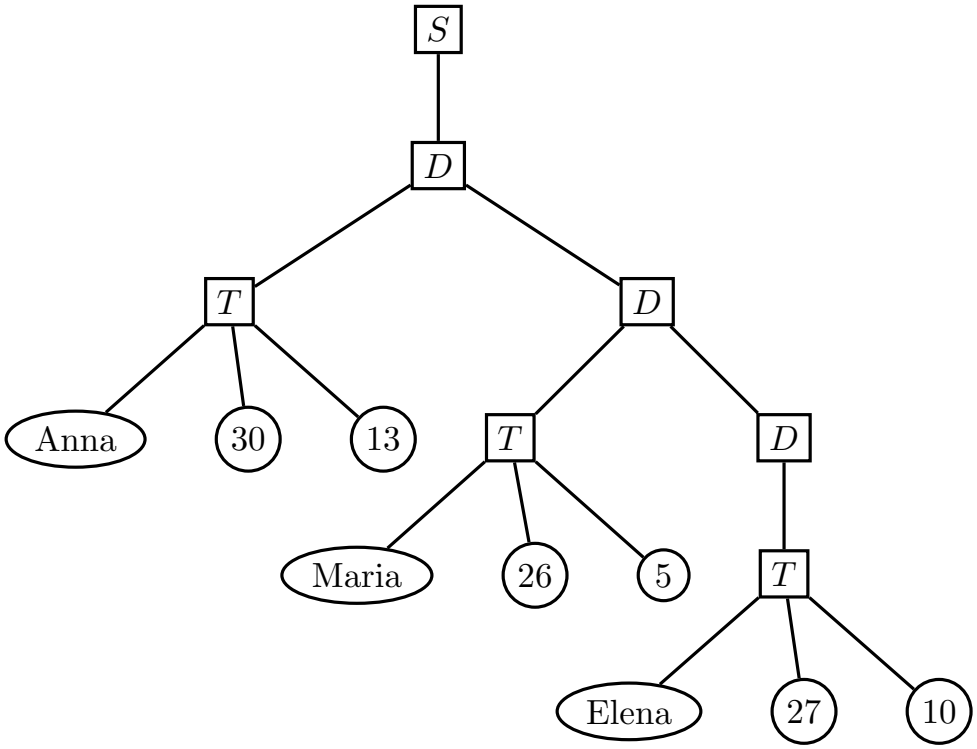
sintassi	funzioni semantiche
1: $S_0 \rightarrow D_1$	
2: $D_0 \rightarrow T_1\ D_2$	
3: $D_0 \rightarrow T_1$	
4: $T_0 \rightarrow$ (**mn**, **ma**, **ca**)	

Domanda (b) facoltativa - attributo semantico del nonterminale T:

sintassi	funzioni semantiche
1: $S_0 \rightarrow D_1$	
2: $D_0 \rightarrow T_1\ D_2$	
3: $D_0 \rightarrow T_1$	
4: $T_0 \rightarrow$ (**mn**, **ma**, **ca**)	

Soluzioni

Prima di procedere ecco un esempio di albero sintattico già costruito (con tre triple) per aiutare a comprendere meglio le funzioni semantiche:



(a) Per calcolare l'attributo *select* basta uno schema puramente sintetizzato, giacché il calcolo procede direttamente da foglie a radice. Eccolo:

	sintassi	calcolo attributi
1	$S_0 \to D_1$	$select_0 = select_1$

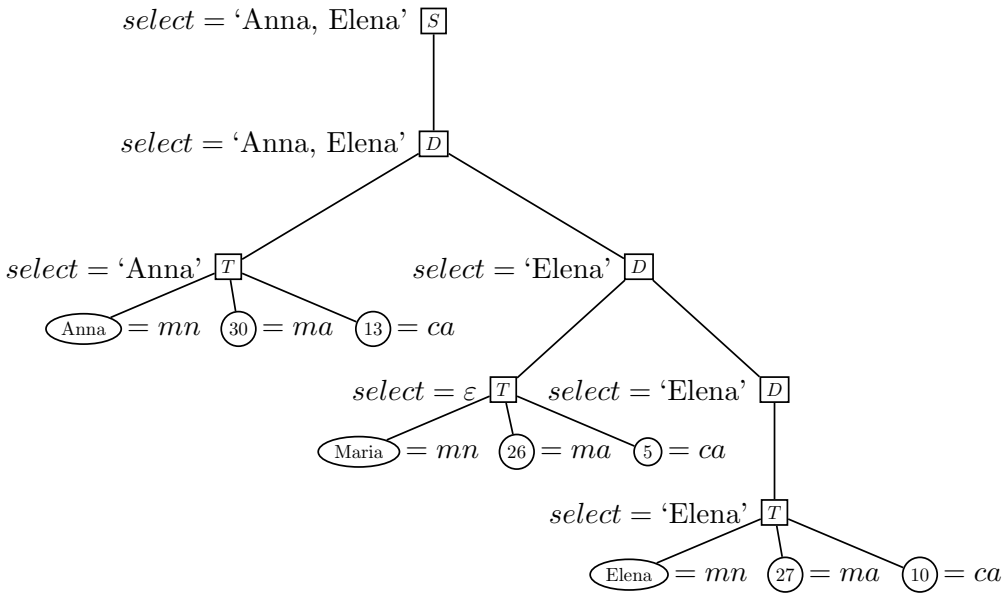
2	$D_0 \to T_1 D_2$	**if** $(select_1 != \varepsilon)$ **then**
		$select_0 = \mathbf{cat}\,(select_1, \text{`,'}, select_2)$
		else
		$select_0 = select_2$
		endif

3	$D_0 \to T_1$	$select_0 = select_1$

4	$T_0 \to (\mathbf{mn}, \mathbf{ma}, \mathbf{ca})$	calcola attributi mn, ma e ca
		if $(ma - ca < 18)$ **then**
		$select_0 = mn$
		else
		$select_0 = \varepsilon$
		endif

L'attributo sinistro *select* di tipo stringa, associato ai nonterminali S, D e T, raccoglie progressivamente la lista dei nomi di madre, separati da virgola, le cui

età soddisfano il test. La lista è costruita aggiungendo il nuovo elemento in testa tramite la funzione **cat** di concatenamento fra stringhe. Gli attributi mn, ma e ca, rispettivamente funzioni dei terminali **mn**, **ma** e **ca**, sono convenzionalmente considerati destri, ma sono calcolabili insieme agli attributi sinistri. Ecco il calcolo esemplificato:



(b) Per questo problema occorre fare uso anche di attributi ereditati. Per ogni t-upla si devono scrivere le regole semantiche che calcolano l'attributo dif di tipo reale, associato al nonterminale T, dandogli il valore seguente:

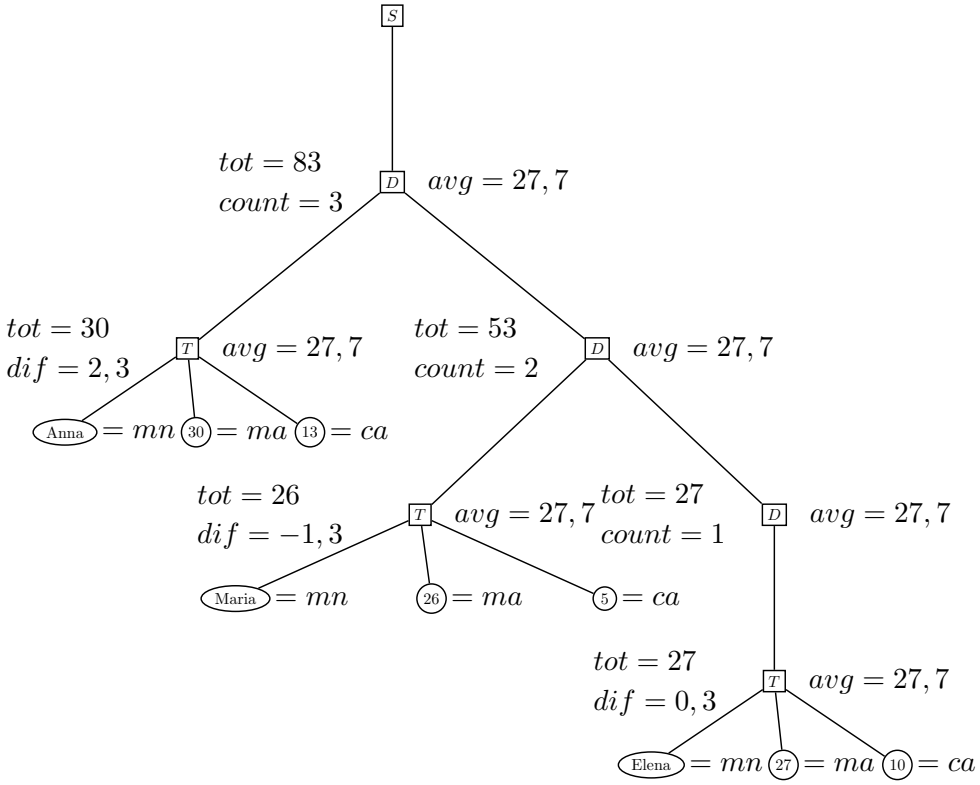
$$dif = MA - AVG(MA)$$

dove AVG è la media degli attributi MA.

Per calcolare AVG si usano due attributi sinistri di tipo intero: tot, associato a D e T; e $count$, associato a D. Essi totalizzano il numero di t-uple e la sommatoria dei valori ma, rispettivamente. Nella radice S si calcola l'attributo destro avg di tipo reale, associato a D e T, spedito in giù verso le t-uple. Infine in ogni t-upla si calcola l'attributo sinistro dif di tipo reale associato a T. Ecco lo schema:

	sintassi	calcolo attr. sinistri	calcolo attr. destri
1	$S_0 \to D_1$		$avg_1 = tot_1/count_1$
2	$D_0 \to T_1 D_2$	$tot_0 = tot_1 + tot_2$ $count_0 = count_2 + 1$	$avg_1 = avg_0$ $avg_2 = avg_0$
3	$D_0 \to T_1$	$tot_0 = tot_1$ $count_0 = 1$	$avg_1 = avg_0$
4	$T_0 \to (\mathbf{mn}, \mathbf{ma}, \mathbf{ca})$	$tot_0 = ma$ $dif_0 = ma - avg_0$	calcola attributi mn, ma e ca

Ecco il calcolo esemplificato sull'albero dato inizialmente:



(c) Si può scegliere quale grammatica con attributi esaminare. Di seguito sono considerate entrambe.

Grammatica (a) La grammatica con attributi che risolve la domanda (a) è puramente sintetizzata e pertanto senz'altro di tipo a una scansione (one sweep). Ecco la codifica della procedura semantica D:

```
procedure D (out select; in tree)
    var select_T, select_D
    if node D → T D then
        call T (select_T, tree → T)
        call D (select_D, tree → D)
        if (select_T != ε) then
            select = cat (select_T, ',', select_D)
        else
            select = select_D
    else if node D → T then
        call T (select_T, tree → T)
        select = select_T
    else error
end procedure
```

La funzione **cat** concatena le stringhe date come argomento. Per completezza si dà la codifica dell'intero valutatore semantico. Ecco la procedura semantica del nonterminale T:

```
procedure T (out select; in tree)
        var mn, ma, ca
        if node T → (mn, ma, ca) then
                – compute attributes mn, ma and ca
                if (ma − ca < 18) then
                        select = mn
                else
                        select = ε
        else error
end procedure
```

E infine ecco la procedura semantica dell'assioma S:

```
procedure S (out select; in root)
        var select_D
        if node S → D then
                call D (select_D, root → D)
                select = select_D
        else error
end procedure
```

Per chiarezza qui la codifica di tutte e tre le procedure per i nonterminali D, T e S, è data in modo da riprodurre fedelmente le funzioni semantiche della grammatica, senza mirare all'efficienza. Tuttavia si potrebbero fare alcune ottimizzazioni programmative. Per esempio alcune variabili locali sono eliminabili utilizzando direttamente gli argomenti. Ecco infine la vista d'insieme del valutatore semantico:

```
program SEMANTIC_ANALYSER
        var S_pointer
        – parse text and build tree
        call S (S_pointer)
end program
```

```
procedure S (out select; in root)
        var select_D
        if node S → D then
                call D (select_D, root → D)
                select = select_D
        else error
end procedure
```

```
procedure D (out select; in tree)
        var select_T, select_D
        if node D → T D then
                call T (select_T, tree → T)
                call D (select_D, tree → D)
                if (select_T != ε) then
                        select = cat (select_T, ',', select_D)
                else
                        select = select_D
        else if node D → T then
                call T (select_T, tree → T)
                select = select_T
        else error
end procedure
```

```
procedure T (out select; in tree)
        var mn, ma, ca
        if node T → (mn, ma, ca) then
                – compute attr. mn, ma, ca
                if (ma − ca < 18) then
                        select = mn
                else
                        select = ε
        else error
end procedure
```

Grammatica (b) Ecco ora l'analisi della grammatica che risolve la domanda (b). Per ciascuna regola si verifica se le dipendenze funzionali soddisfano la condizione a una scansione (one sweep). In generale sarebbe opportuno disegnare il grafo delle dipendenze per ogni regola, ma qui lo schema è semplicissimo e basta procedere intuitivamente esaminando le regole una per una. Ecco l'esame:

regola 1 non va bene: l'attributo destro avg_1, associato al nodo figlio D, dipende dagli attributi sinistri tot_1 e $count_1$ dello stesso nodo; ciò vìola la condizione a una scansione giacché il valutatore semantico deve disporre di avg_1 prima di invocare la procedura semantica associata al nodo D, laddove i dati tot_1 e $count_1$ necessari per tale calcolo sono disponibili solo dopo avere invocato la procedura semantica del nodo D

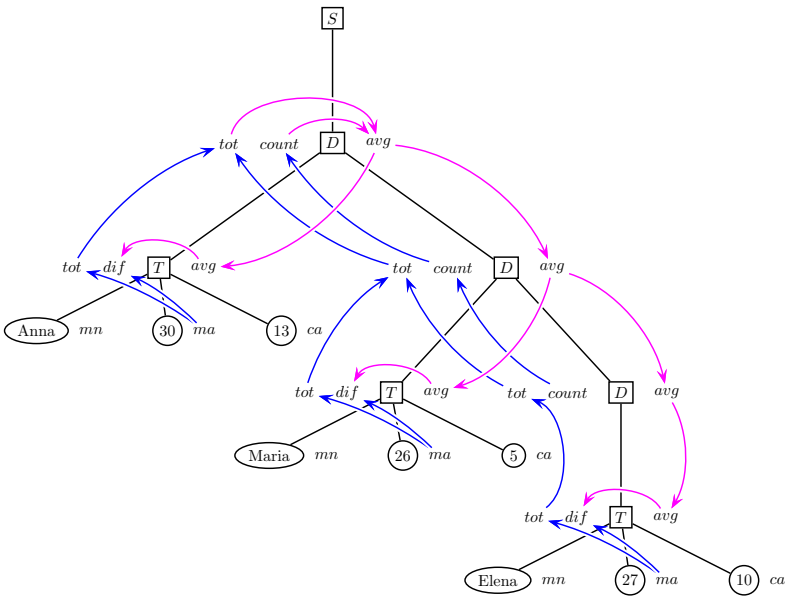
regola 2 va bene: gli attributi sinistri tot_0 e $count_0$ non danno problema in quanto dipendenti solo da attributi sinistri, e gli attributi destri avg_1 e avg_2 dei figli T e D dipendono solo dall'attributo destro avg_0 del padre D

regola 3 va bene: idem come sopra (e c'è solo il figlio T)

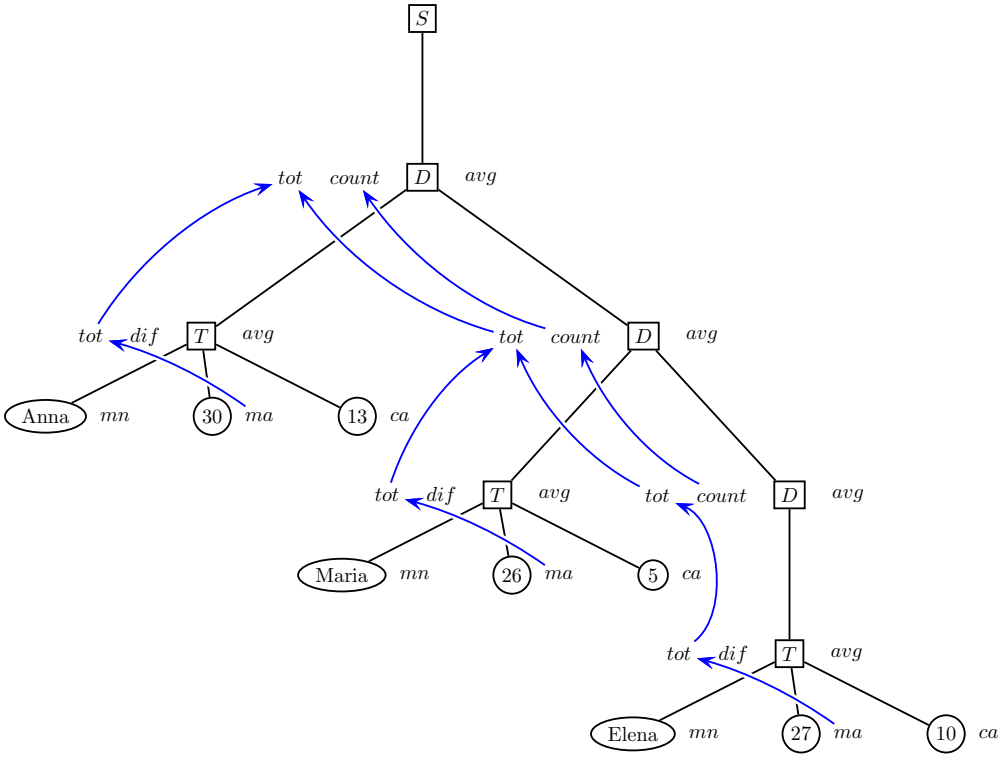
regola 4 va bene: l'attributo sinistro tot_0 dipende solo dall'attributo ma calcolato localmente come funzione del terminale corrispondente, e l'attributo sinistro dif_0 dipende dall'attributo ma, calcolato localmente come funzione del terminale corrispondente, e dall'attributo destro avg_0 del padre T

Pertanto per questa grammatica è impossibile costruire il valutatore semantico a una scansione (one sweep). In definitiva è l'attributo destro avg che lo impedisce, giacché richiede prima il calcolo completo degli attributi sinistri tot e $count$.

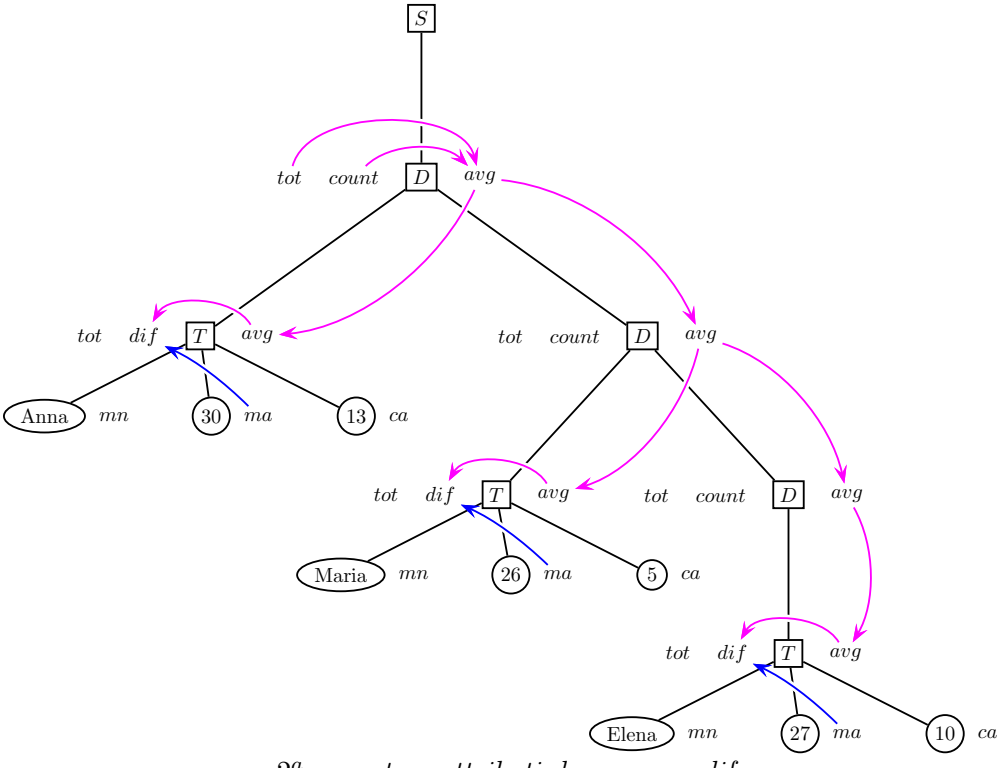
Tuttavia si possono valutare gli attributi con due scansioni. La prima scansione calcola gli attributi sinistri tot e $count$. Essendo noti i loro valori, la seconda scansione calcola gli attributi rimanenti avg (destro) e dif (ancora sinistro ma dipendente da avg che è destro). Dunque si può costruire il valutatore semantico a due passate supponendo di avere l'albero sintattico già interamente costruito e disponibile. Ciò si vede anche esaminando le dipendenze tra attributi, per esempio sull'albero mostrato prima (qui non sarebbe richiesto). Gli archi blu e rosa rappresentano le dipendenze in salita e in discesa o laterali, rispettivamente:



Dettagliatamente ecco l'albero con le dipendenze funzionali tra attributi diviso in due per prima e seconda passata, sopra e sotto rispettivamente:



1^a *passata - attributi sx tot e count*



2^a *passata - attributi dx avg e sx dif*

La procedura semantica D va divisa in due versioni D_{first} e D_{second} per prima e seconda scansione, rispettivamente. La prima versione calcola gli attributi *tot* e *count*, la seconda *avg*. La procedura semantica del nonterminale T va divisa in modo analogo, e T_{first} e T_{second} vanno chiamate in D_{first} e D_{second}, rispettivamente. Ecco la codifica della procedura D (prima e seconda versione):

```
procedure D_first (out tot, count; in tree)
    var tot_T, count_D, tot_D
    if node D → T D then
        call T_first (tot_T, tree → T)
        call D_first (tot_D, count_D, tree → D)
        tot = tot_T + tot_D
        count = count_D + 1
    else if node D → T then
        call T_first (tot_T, tree → T)
        tot = tot_T
        count = 1
    else error
end procedure
```

1^a *passata - attributi sx tot e count*

```
procedure D_second (in tree, avg)
    var avg_T, avg_D
    if node D → T D then
        avg_T = avg
        avg_D = avg
        call T_second (null, tree → T, avg_T)
        call D_second (tree → D, avg_D)
    else if node D → T then
        avg_T = avg
        call T_second (null, tree → T, avg_T)
    else error
end procedure
```

2^a *passata - attributo dx avg*

Si noti che l'ordine di visita dei nodi è quello naturale da sinistra a destra. Gli assegnamenti identici alle variabili locali avg_T e avg_D sono un po' formali e si potrebbero semplificare passando direttamente il parametro *avg* alle procedure T_{second} e D_{second}; qui sono mostrati perché riproducono fedelmente le funzioni semantiche della grammatica e pertanto rendono più esplicito il codice. Formalmente l'attributo sinistro dif è parametro in uscita alla procedura T_{second}; tuttavia il testo dell'esercizio chiede semplicemente di calcolarlo senza poi farne uso, pertanto quando la procedura D_{second} chiama T_{second}, passa il puntatore *null* per significare che il parametro dif in uscita da T_{second} è irrilevante.

Per generalità qui si dà la codifica dell'intero valutatore semantico a due passate. Prima e seconda versione T_{first} e T_{second} della procedura semantica T calcolano gli attributi *tot* e dif, rispettivamente. Ecco le due versioni:

```
procedure T_first (out tot; in tree)
    var mn, ma, ca
    if node T → (mn, ma, ca) then
        – compute attr. mn, ma, ca
        tot = ma
    else error
end procedure
```

1^a *passata - attributo sx tot*

```
procedure T_second (out dif; in tree, avg)
    var mn, ma, ca
    if node T → (mn, ma, ca) then
        – compute attr. mn, ma, ca
        dif = ma − avg
    else error
end procedure
```

2^a *passata - attributi sx dif e dx avg*

Formalmente anche la procedura semantica della radice S andrebbe divisa in due versioni: la prima calcola gli attributi *count* e *tot*, la seconda *avg* e dif. Però le due versioni sono unificabili in quanto vanno chiamate consecutivamente: infatti è qui dove avviene la transizione da prima a seconda passata. Pertanto ecco la procedura semantica unificata $S_{unified}$ dell'assioma S:

```
procedure S_unified (in root)
        var tot_D, count_D, avg_D
        if node S → D then
                call D_first (tot_D, count_D, root → D)
                avg_D = tot_D/count_D
                call D_second (root → D, avg_D)
        else error
end procedure
```

Si noti il nesso cruciale tra chiamata a prima e seconda versione della procedura D, dove si calcola l'attributo destro *avg* in funzione degli attributi sinistri *tot* e *count*: lì si ha passaggio da informazione sintetizzata a ereditata.

Ciò conclude il progetto dell'intero valutatore semantico a due passate. Per maggiore chiarezza ecco la vista d'insieme del valutatore semantico a due passate:

```
program SEMANTIC_ANALYSER
        var S_pointer
        – parse text and build tree
        call S_unified (S_pointer)
end program
```

```
procedure S_unified (in root)
        var tot_D, count_D, avg_D
        if node S → D then
                call D_first (tot_D, count_D, root → D)
                avg_D = tot_D/count_D
                call D_second (root → D, avg_D)
        else error
end procedure
```

```
procedure D_first (out tot, count; in tree)
        var tot_T, count_D, tot_D
        if node D → T D then
                call T_first (tot_T, tree → T)
                call D_first (tot_D, count_D, tree → D)
                tot = tot_T + tot_D
                count = count_D + 1
        else if node D → T then
                call T_first (tot_T, tree → T)
                tot = tot_T
                count = 1
        else error
end procedure
```

```
procedure D_second (in tree, avg)
        var avg_T, avg_D
        if node D → T D then
                avg_T = avg
                avg_D = avg
                call T_second (null, tree → T, avg_T)
                call D_second (tree → D, avg_D)
        else if node D → T then
                avg_T = avg
                call T_second (null, tree → T, avg_T)
        else error
end procedure
```

```
procedure T_first (out tot; in tree)
        var mn, ma, ca
        if node T → (mn, ma, ca) then
                – compute attr. mn, ma, ca
                tot = ma
        else error
end procedure
```

```
procedure T_second (out dif; in tree, avg)
        var mn, ma, ca
        if node T → (mn, ma, ca) then
                – compute attr. mn, ma, ca
                dif = ma − avg
        else error
end procedure
```

1^a passata | **2^a passata**

Va da sé che presumibilmente si possono fare ottimizzazioni puramente programmative alla codifica proposta per il valutatore semantico.