Linguaggi Formali e Compilatori
(Formal Languages and Compilers)

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

Prova scritta - 31 gennaio 2011 - 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:

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. È data l'espressione regolare R seguente:

$$R = \Big(\Big(\big(\, (\, a \, \cup \, c \,)^* \; c \, b^+ \, \big) \, \cap \, \big(\, c \, b^* \, \big)^* \, \Big) \, \cup \, a \Big)^*$$

che contiene anche l'operatore intersezione '$\cap$'.

Si risponda alle domande seguenti:

(a) Si elenchino le cinque frasi più corte del linguaggio $L(R)$, in ordine alfabetico.

(b) Si costruisca un'espressione regolare R', equivalente a R ma con i soli operatori di base (unione, concatenamento, stella e croce), e si mostri come si è ottenuta tale espressione R'.

Soluzione

(a) Sub-expression E is isolated as follows:

$$\Big(\underbrace{\Big(\big(\, (\, a \, \cup \, c \,)^* \; c \, b^+ \, \big) \, \cap \, (\, c \, b^* \,)^* \, \Big)}_{E} \cup a \Big)^*$$

and generates the following language:

$$L(E) = \{ \, cb, \; acb, \; ccb, \; cbb, \; aacb, \; accb, \; cacb, \; cccb,$$
$$acbb, \; ccbb, \; cbbb, \; \ldots \} \cap$$
$$\cap \, \{ \, \varepsilon, \; c, \; cb, \; cc, \; cbb, \; ccb, \; ccc, \; cbcb, \; \ldots \} =$$
$$= \; \{ \, cb, \; ccb, \; cbb, \; \ldots \}$$

To get the full language $L(R)$, these short strings of $L(E)$ should be freely concatenated with a, as $L(R) = (\, L(E) \; | \; a \,)^*$. Here are the five shortest strings of the language $L(R)$, in alphabetical order (assuming as usual $a < b < c$):

$$\varepsilon \qquad a \qquad aa \qquad cb \qquad aaa \qquad \ldots$$

There actually are more strings of length three, namely acb and cba, but aaa is the first one in alphabetical order.

(b) The common strings in the two members of the intersection contained in the sub-expression E, are $c^+ \, b^+$, i.e.:

$$L(E) = c^+ \, b^+$$

which have to be united to a and then closed by the star operator to generate the full language $L(R)$. Thus a valid expression R' is as follows:

$$R' = \Big(\; (\, c^+ \, b^+ \,) \; | \; a \, \Big)^*$$

By the way, expression R' is surely not ambiguous as it is linear: each generator occurs only once in the expression.

2. È data l'espressione regolare R seguente:

$$R = b\ (\ a\ b\ \mid\ b\ a\ )^*\ a$$

Si risponda alle domande seguenti:

(a) Utilizzando il metodo BS (Berri-Sethi), si costruisca un automa deterministico A che riconosce il linguaggio $L(R)$.

(b) Si dica, dandone una dimostrazione il più possibile formale, se il linguaggio $L(R)$ è di tipo locale.

Soluzione

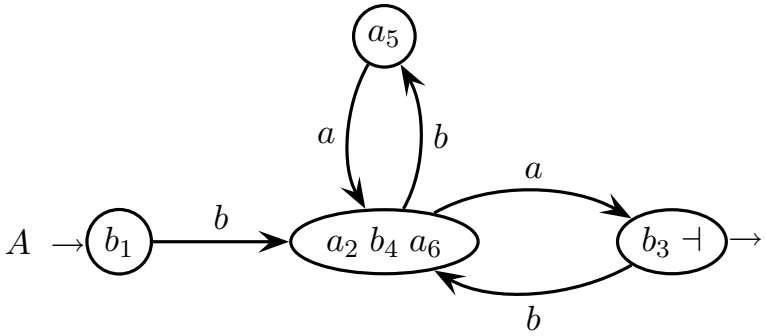
(a) Ecco l'espressione $R_\#$ con i generatori numerati e il terminatore:

$$R_\# = b_1 \ (\ a_2\ b_3\ \mid\ b_4\ a_5\ )^*\ a_6\ \dashv$$

Ecco la tabella degli inizi e dei seguiti:

inizi	b_1
gen.	seguiti
b_1	$a_2\ b_4\ a_6$
a_2	b_3
b_3	$a_2\ b_4\ a_6$
b_4	a_5
a_5	$a_2\ b_4\ a_6$
a_6	$\dashv$

Ed ecco l'automa A deterministico di Berri-Sethi:



(b) Il linguaggio $L(R)$ non è locale. Ecco gl'insiemi degli inizi, delle fini e dei digrammi:

$$Ini(R)\ =\ \{\ b\ \} \qquad Fin(R)\ =\ \{\ a\ \}$$

$$Dig(R) = \{\ a\,a,\ a\,b,\ b\,a,\ b\,b\ \}\ =\ \Sigma^2$$

Il linguaggio L' generato componendo in tutti i possibili modi $Ini(R)$, $Dig(R)$ e $Fin(R)$, è il seguente:

$$L' = b\ (\ a\ \mid\ b\ )^*\ a$$

cioè l'insieme di tutte le stringhe di lunghezza maggiore o uguale a due, che iniziano con una lettera b e terminano con una lettera a. Ma il linguaggio $L(R)$ è un sottoinsieme proprio di tale linguaggio L' (per esempio $L(R)$ non contiene la stringa $b\,a\,a$), pertanto $L(R)$ non è locale.

2 Grammatiche libere e automi a pila 20%

1. Si consideri la grammatica libera G seguente (assioma S):

$$G \begin{cases} S & \rightarrow & a\ S\ X\ b \\ S & \rightarrow & \varepsilon \\ X & \rightarrow & c\ X \\ X & \rightarrow & \varepsilon \end{cases}$$

Si considerino ora le due espressioni regolari R_1 e R_2 seguenti:

$$R_1 = a^*\ (\ c\ c\ b\ )^*$$

$$R_2 = a^*\ (\ c\ c\ b^*\ )^*$$

Si risponda alle domande seguenti:

(a) Si scriva una grammatica G_1 per il linguaggio $L_1 = L(G) \cap L(R_1)$.

(b) Si scriva una grammatica non ambigua G_2 per il linguaggio $L_2 = L(G) \cap L(R_2)$.

(c) (facoltativa) Si disegni un automa a pila deterministico P che riconosce il linguaggio $L(G_2)$ (a scelta con pila vuota o stato finale).

Soluzione

(a) Ogni lettera b è preceduta da esattamente due lettere c.

$$G_1 \begin{cases} S & \to & a\ S\ c\ c\ b \\ S & \to & \varepsilon \end{cases}$$

(b) Ogni lettera b è preceduta da un numero pari di lettere c, anche nessuna, tranne la lettera b più interna che dev'essere preceduta da almeno due lettere c.

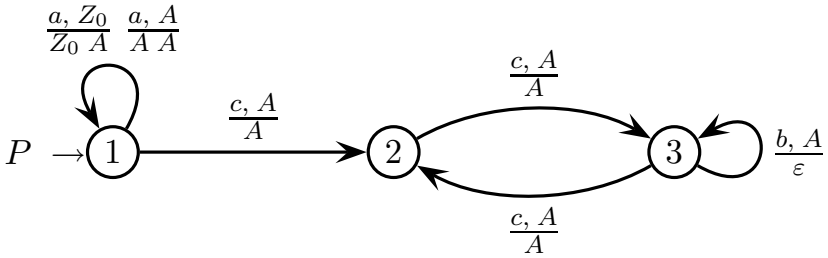
$$G_2 \begin{cases} S & \to & Y \\ S & \to & \varepsilon \\ Y & \to & a\ Y\ X\ b \\ Y & \to & c\ c \\ X & \to & c\ c\ X \\ X & \to & \varepsilon \end{cases}$$

La grammatica G_2 non è ambigua: la struttura $a^n\ b^n$ $(n \geq 0)$ è generata nel noto modo autoinclusivo non ambiguo, e le regole unilineari di X non sono ambigue (le s'immagini come automa a stati finiti deterministico).

(c) Il linguaggio $L(G_2)$, rappresentato come insieme di stringhe, è il seguente:

$$L(G_2) = \left\{ \varepsilon,\ \ a^n \prod_{i=1}^{n} \left(c^{2e_i}\ b \right) \ \middle|\ \ n \geq 1\ \wedge\ e_1,\, e_{i\geq 2} \geq 1,\, 0 \right\}$$

Con la pila si riconosce la struttura autoinclusiva $a^n\ b^n$ $(n \geq 0)$, mentre con gli stati finiti si riconoscono i fattori regolari $(c\,c)^*$ di lunghezza pari. Ora è facile tracciare l'automa riconoscitore P a pila vuota. Eccolo:



L'automa P è chiaramente deterministico: prima conta le lettere a impilando altrettanti simboli A, poi conta almeno due lettere c senza modificare la pila, poi conta le lettere b spilando altrettanti simboli A, riconosce a pila vuota ossia quando i numeri di a e b sono uguali, e prima di ogni lettera b accetta numeri pari di lettere c senza modificare la pila; la stringa vuota è riconosciuta all'inizio.

2. Si consideri uno stralcio di linguaggio di programmazione contenente dichiarazioni di array (a una o più dimensioni), eventuali inizializzazioni delle stesse e istruzioni di assegnamento. Un esempio tipico è il seguente:

```
-- integer matrix with size 2 by 2 and integer vector with size 3
int X [2, 2], Y [3] ;

real Z [10, 10, 10], W [2, 2] ;  -- real 3D and 2D arrays

Y = < 77, 14, 8 > ;  -- vector initialized with the value list

W = X ;  -- assignment of a whole array to another of equal size

-- assignment of an expression to an array element
W [1, Y [2] ] = 2 * ( Z [1, 2, 1] + Z [10, 10, 9] ) * W [1, 1] + 9 ;
```

   Ecco le specifiche del linguaggio:

   - gli identificatori come X, Y, W, ... sono istanze del terminale *id*
   - le costanti come 10, 2, ... sono istanze del terminale *const*
   - le dichiarazioni precedono le inizializzazioni che precedono gli assegnamenti
   - l'inizializzazione è facoltativa e si possono inizializzare soltanto vettori
   - un assegnamento a un array ha come parte destra un array, non un'espressione
   - un assegnamento a un elemento di array ha come parte destra un'espressione aritmetica che può contenere come operandi elementi di array e costanti, e come operatori addizione e moltiplicazione; inoltre:
     - l'indice di array può essere una costante o un elemento di array
     - la moltiplicazione ha precedenza sull'addizione
     - ci possono essere parentesi

   Si risponda alle domande seguenti:

   (a) Si progetti una grammatica, in forma BNF estesa (EBNF) e non ambigua, per il linguaggio specificato.

   (b) (facoltativa) Si perfezioni la grammatica progettata in modo che il numero di inizializzazioni sia sempre minore o uguale al numero di vettori dichiarati.

Soluzione

(a) Grammar (axiom PRG):

$$
\left\{
\begin{array}{rcl}
\langle \mathsf{PRG} \rangle & \rightarrow & \underbrace{\langle \mathsf{DCL} \rangle}_{declarations} \quad \underbrace{\langle \mathsf{INI} \rangle}_{initializations} \quad \underbrace{\langle \mathsf{USE} \rangle}_{uses} \\
\hline
\langle \mathsf{DCL} \rangle & \rightarrow & \Big(\underbrace{\langle \mathsf{TYP} \rangle}_{type} \; \underbrace{\langle \mathsf{OBJ} \rangle}_{object} \; (\text{`,'} \; \langle \mathsf{OBJ} \rangle \; )^* \; \text{`;'} \Big)^+ \\
\langle \mathsf{INI} \rangle & \rightarrow & \Big(id \; \text{`='} \; \text{`}\langle\text{'} \; const \; (\text{`,'} \; const \; )^* \; \text{`}\rangle\text{'} \; \text{`;'} \Big)^* \\
\langle \mathsf{USE} \rangle & \rightarrow & \Big(id \; \text{`='} \; id \; \text{`;'} \; | \; \underbrace{\langle \mathsf{ARF} \rangle}_{array\ ref.} \; \text{`='} \; \underbrace{\langle \mathsf{EXP} \rangle}_{expression} \; \text{`;'} \Big)^+ \\
\hline
\langle \mathsf{TYP} \rangle & \rightarrow & id \\
\langle \mathsf{OBJ} \rangle & \rightarrow & id \; \text{`['} \; \underbrace{const}_{array\ range} \; ( \text{`,'} \; const \;)^* \; \text{`]'} \\
\langle \mathsf{ARF} \rangle & \rightarrow & id \; \text{`['} \; \underbrace{\langle \mathsf{IND} \rangle}_{index} \; ( \text{`,'} \; \langle \mathsf{IND} \rangle \;)^* \; \text{`]'} \\
\langle \mathsf{EXP} \rangle & \rightarrow & \underbrace{\langle \mathsf{TRM} \rangle}_{term} \; (\text{`+'} \; \langle \mathsf{TRM} \rangle \;)^* \\
\hline
\langle \mathsf{IND} \rangle & \rightarrow & const \; | \; id \; \text{`['} \; \langle \mathsf{IND} \rangle \; ( \text{`,'} \; \langle \mathsf{IND} \rangle \;)^* \; \text{`]'} \\
\langle \mathsf{TRM} \rangle & \rightarrow & \underbrace{\langle \mathsf{FCT} \rangle}_{factor} \; (\text{`}*\text{'} \; \langle \mathsf{FCT} \rangle \;)^* \\
\hline
\langle \mathsf{FCT} \rangle & \rightarrow & const \; | \; \langle \mathsf{ARF} \rangle \; | \; \text{`('} \; \langle \mathsf{EXP} \rangle \; \text{`)'}
\end{array}
\right.
$$

Array initialisation might be extended to allow expressions, not just constants, as the example only shows constants while there is not any textual specification on this detail; that is:

$$\langle \mathsf{INI} \rangle \rightarrow \Big(id \; \text{`='} \; \text{`}\langle\text{'} \; \langle \mathsf{EXP} \rangle \; (\text{`,'} \; \langle \mathsf{EXP} \rangle \; )^* \; \text{`}\rangle\text{'} \; \text{`;'} \Big)^*$$

Clearly such extension implies expressions are static, that is can be evaluated at declaration time. This is not a syntactic aspect anyway, but semantic.

(b) The constraint to be introduced is similar to the following abstract language:

$$V^m\ I^n \qquad m \geq n \geq 0$$

where m is the number of vectors declared and n is the number of initializations. This is the well-known language:

$$V^m\ V^*\ I^m \qquad m \geq 0$$

generated by the grammar $S \rightarrow V\ S\ I\ \mid\ V^*$.

To count the vectors declared, we have to rearrange the grammar so that different rules are used to generate 1D vector declarations and 2D, 3D, etc, matrix declarations. Attention should be paid to that vector and matrix declarations can be placed in any order. Of course matrix declarations should not be counted.

3 Analisi sintattica e parsificatori 20%

1. Si consideri la grammatica G seguente (assioma S):

$$G \begin{cases} S & \rightarrow & S\ p\ S \ \mid\ u\ S \ \mid\ A \\ A & \rightarrow & \text{`('}\ B\ \text{`)'} \ \mid\ t \\ B & \rightarrow & B\ q \ \mid\ q \end{cases}$$

   Si risponda alle domande seguenti:

   (a) Si mostri che la grammatica non è né $LL(1)$ né $LL(k)$ per alcun $k > 1$.

   (b) Si scriva una grammatica G_L, eventualmente in forma estesa (EBNF), equivalente a G e di tipo $LL(1)$.

Soluzione

(a) La grammatica G è ambigua per via della ricorsione bilaterale nella regola $S \to S\ p\ S$ e inoltre c'è ricorsione sinistra nella regola $B \to B\ q$. Pertanto la grammatica G non è $LL(k)$ per alcun $k \geq 1$.

(b) Ecco la grammatica G_L, in forma estesa EBNF (assioma S):

$$G_L \begin{cases} S & \to & U\ A\ P \\ P & \to & p\ U\ A\ P \\ P & \to & \varepsilon \\ U & \to & u\ U \\ U & \to & \varepsilon \\ A & \to & \text{`('}\ B\ \text{`)'} \\ A & \to & t \\ B & \to & q^* \end{cases}$$

La grammatica G_L è equivalente alla grammatica G. Eccone la giustificazione. In G le regole di S fanno sì che da S derivi una lista di elementi di tipo u^*A separati da una p; in G_L questa lista viene resa dalle regole di S, P e U. Le regole di A sono identiche in G e G_L. In G le regole di B (unilineari) fanno sì che da B derivi una stringa q^*; in G_L la regola estesa di B provvede direttamente allo stesso scopo.

La grammatica G_L è di tipo $LL(1)$. Si possono tracciare gli automi dei non-terminali, ma basta anche esaminare gl'insiemi guida delle varie alternative, qui molto semplici da ricavare. Eccoli:

$$G_L \begin{cases} & & & \text{guida} \\ S & \to & U\ A\ P & u,\ (,\ t \\ P & \to & p\ U\ A\ P & p \\ P & \to & \varepsilon & \dashv \\ U & \to & u\ U & u \\ U & \to & \varepsilon & (,\ t \\ A & \to & \text{`('}\ B\ \text{`)'} & (\\ A & \to & t & p \\ B & \to & q^* & q,\ ) \end{cases}$$

Si vede subito che gl'insiemi guida di regole alternative (che negli automi corrisponderebbero a biforcazioni) sono disgiunti, mentre la regola estesa finale ha guida q per i casi di tipo $B \to q^+$ (che nell'automa corrisponderebbero a restare in ciclo) e guida ')' per il caso vuoto $B \to \varepsilon$ (che corrisponderebbe a uscire dal ciclo), dunque anche qui le guide sono disgiunte. Volendo, si traccino gli automi.

2. Si consideri la grammatica G seguente (assioma S):

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

Si risponda alle domande seguenti:

(a) Si tracci il grafo pilota completo di tipo $LR(1)$ della grammatica G e si indichino tutti gli eventuali conflitti presenti nel grafo, precisandone il tipo.

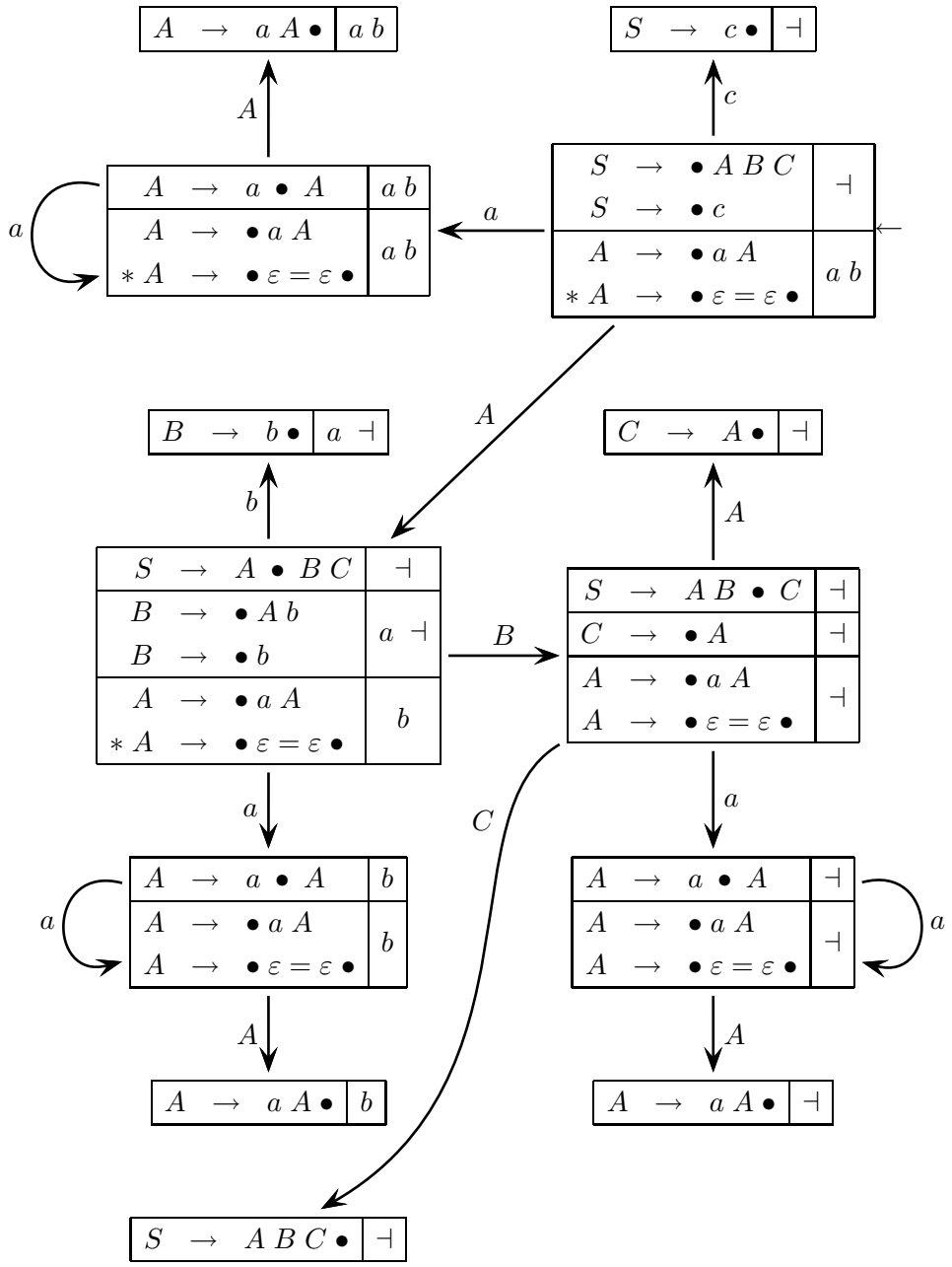
(b) (facoltativa) Se la grammatica G non è di tipo $LR(1)$, se ne trovi una equivalente G' che lo è.

Soluzione

(a) Conviene prima dare la tabella di nullabilità e dei caratteri iniziali dei nonterminali, che si ottiene facilmente:

nonterm.	nullabile	car. iniziali
S	no	$a\ b\ c$
A	sì	a
B	no	$a\ b$
C	sì	a

Ecco il grafo pilota completo di G:



Si vedano le candidate di riduzione con prefisso $*$. Le due riduzioni $A \to \varepsilon$ con prospezione a e quella con b sono conflittuali con gli spostamenti, in quanto esitono archi uscenti (o autoanelli) con etichetta a o b. Il resto del grafo è privo di conflitti. Pertanto la grammatica G non è di tipo $LR(1)$.

(b) Il linguaggio $L(G)$ è regolare, poiché

$$L(A) = a^* \qquad L(B) = L(A)\ b = a^*\ b \qquad L(C) = L(A) = a^*$$

dunque

$$L(S) = L(A)\ L(B)\ L(C)\ \mid\ c = a^*\ a^*\ b\ a^*\ \mid\ c = a^*\ b\ a^*\ \mid\ c$$

e una qualunque grammatica unilineare a destra, ricavata da un automa a stati finiti deterministico, è anche $LR(1)$. Si lascia al lettore di ricavarla.

4 Traduzione e analisi semantica 20%

1. Si progetti un traduttore finito (ossia IO-automa) per calcolare la seguente traduzione. Il linguaggio sorgente è $L = (a\ a)^*$, l'alfabeto del linguaggio immagine è $\{ 1,\ 2,\ 3,\ t \}$ e la traduzione è la seguente:

$$\tau (a^n) = (1\ 2\ 3)^{n \text{ div } 3}\ t^{n \bmod 3} \qquad\qquad n \text{ è pari e} \geq 0$$

Esempi:

$$\begin{aligned} \tau (a\ a) &= (1\ 2\ 3)^{2 \text{ div } 3}\ t^{2 \bmod 3} &&= t\ t \\ \tau (a\ a\ a\ a\ a\ a) &= (1\ 2\ 3)^{6 \text{ div } 3}\ t^{6 \bmod 3} &&= 1\ 2\ 3\ 1\ 2\ 3 \\ \tau (a^{10}) &= (1\ 2\ 3)^{10 \text{ div } 3}\ t^{10 \bmod 3} &&= (1\ 2\ 3)^3\ t \end{aligned}$$

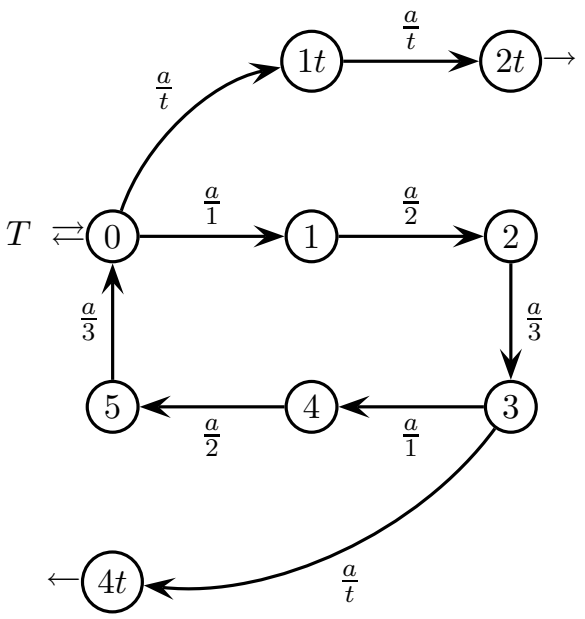
Si risponda alle domande seguenti:

(a) Si disegni il grafo dell'automa finito traduttore T per la traduzione τ e si dica se il traduttore T è deterministico.

(b) (facoltativa) In caso negativo, si discuta come ottenere un traduttore finito deterministico T' per la stessa funzione di traduzione τ (se lo si ritiene utile si supponga pure che la stringa d'ingresso abbia un terminatore $\dashv$).

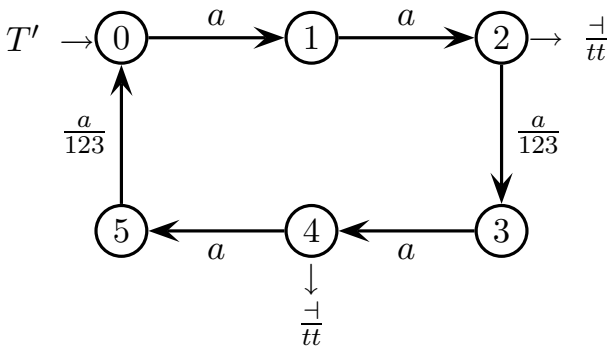
Soluzione

(a) Ecco il grafo dell'automa traduttore T per la traduzione τ:



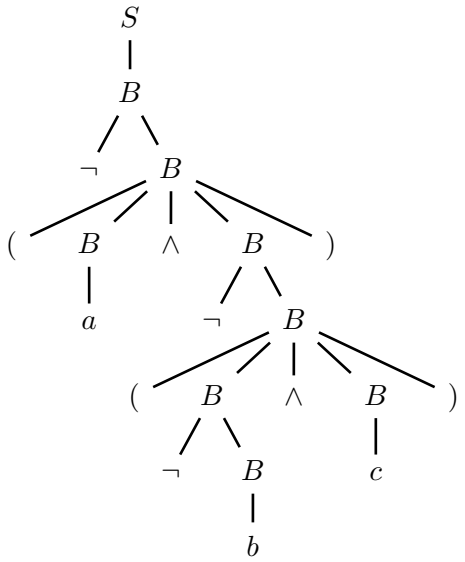
Si tenga presente che il linguaggio sorgente consiste di stringhe di lettere a di lunghezza pari: 0, 2, 4, ecc; in ogni caso non sono riconoscibili e dunque neppure traducibili, stringhe sorgente di lunghezza dispari. Il traduttore T non è deterministico negli stati 0 e 3.

(b) Il non determinismo è dovuto a che l'automa T non sa se la prima lettera a sarà seguita da altre due a o no, e di conseguenza che non può scegliere se tradurla come 1 o t. La soluzione sta nel ritardare l'emissione a quando l'automa sa di avere letto tre lettere a o di avere terminato prima di leggerle. Il traduttore deterministico di tipo *sequenziale* (con terminatore $\dashv$) T' è mostrato qui:



2. Si consideri la seguente grammatica astratta (assioma S) per le espressioni di logica proposizionale con lettere predicative o letterali, e connettivi logici di negazione '$\neg$', congiunzione '$\wedge$' e disgiunzione '$\vee$'. Per comodità gli operatori binari sono sempre racchiusi tra parentesi. Il simbolo terminale lp rappresenta tutti i letterali.

$$\begin{cases} S & \rightarrow & B \\ B & \rightarrow & \text{`}\neg\text{'}\ B \\ B & \rightarrow & \text{`(' } B \text{ `}\vee\text{' } B \text{ `)'} \\ B & \rightarrow & \text{`(' } B \text{ `}\wedge\text{' } B \text{ `)'} \\ B & \rightarrow & \text{`}lp\text{'} \end{cases}$$



Per esempio la formula $\neg\ (\ a\ \wedge\ \neg\ (\ \neg\ b\ \wedge\ c\ )\ )$ corrisponde all'albero sintattico sopra.

Si risponda alle domande seguenti:

(a) Si scriva una grammatica con attributi, usando solo due attributi sintetizzati p e d che determinino i numeri di letterali racchiusi in un numero *pari* e *dispari* di negazioni annidate, rispettivamente (si veda la tabella a pagina seguente). Nell'esempio due letterali, a e b, sono racchiusi in un numero dispari (ossia 1 e 3 negazioni rispettivamente), mentre il solo letterale c è racchiuso in un numero pari (ossia 2 negazioni).

(b) (facoltativa) Le formule di logica proposizionale ammettono una forma normale and-or con l'operatore di negazione applicato soltanto ai letterali. Da una generica formula proposizionale se ne ottiene una equivalente in forma normale applicando ripetutamente le leggi di De Morgan e di cancellazione della doppia negazione. L'esempio sopra si normalizza così:

$$\begin{aligned} &\neg\ (\ a\ \wedge\ \neg\ (\ \neg\ b\ \wedge\ c\ )\ ) \Rightarrow \\ &(\neg\ a\ \vee\ \neg\ \neg\ (\ \neg\ b\ \wedge\ c\ )\ ) \Rightarrow \\ &(\ \neg\ a\ \vee\ (\ \neg\ b\ \wedge\ c\ )\ ) \end{aligned}$$

– la neg. è solo sui letterali

Si scriva una grammatica con attributi a una passata che calcoli la versione in forma normale come attributo della radice dell'albero.

Allo scopo si usino un attributo ereditato *neg* indicante se una (sotto)espressione è preceduta da numero dispari o pari di negazioni annidate e uno sintetizzato τ per costruire la traduzione normalizzata (si veda la tabella a pagina seguente).

attributi da usare per la grammatica - domanda (a)

tipo	nome	(non)terminali	dominio	significato
sin	p	S, B	intero	numero di letterali racchiusi in numero pari di negazioni annidate
sin	d	S, B	intero	numero di letterali racchiusi in numero dispari di negazioni annidate

attributi da usare per la grammatica - domanda (b)

tipo	nome	(non)terminali	dominio	significato
des	neg	S, B	booleano	VERO se la (sotto)espressione è preceduta da numero DISPARI di negazioni annidate; FALSO se la (sotto)espressione è preceduta da numero PARI di negazioni annidate
sin	τ	S, B	stringa	traduzione normalizzata della (sotto)espressione

sintassi	calcolo attributi (domanda (a))
1: $S_0 \rightarrow B_1$	
2: $B_0 \rightarrow$ '$\neg$' B_1	
3: $B_0 \rightarrow$ '(' B_1 '$\vee$' B_2 ')'	
4: $B_0 \rightarrow$ '(' B_1 '$\wedge$' B_2 ')'	
5: $B_0 \rightarrow$ 'lp'	

sintassi	calcolo attributi (domanda (b))
1: $S_0 \rightarrow B_1$	
2: $B_0 \rightarrow$ '$\neg$' B_1	
3: $B_0 \rightarrow$ '(' B_1 '$\vee$' B_2 ')'	
4: $B_0 \rightarrow$ '(' B_1 '$\wedge$' B_2 ')'	
5: $B_0 \rightarrow$ 'lp'	

Soluzione

(a) Ecco la soluzione, piuttosto intuitiva, che si avvale solo dei due attributi dati:

sintassi	calcolo attributi (domanda (a))
1: $S_0 \to B_1$	$p_0 = p_1$ $d_0 = d_1$
2: $B_0 \to$ '$\neg$' B_1	– scambio pari-dispari ! $p_0 = d_1$ $d_0 = p_1$
3: $B_0 \to$ '(' B_1 '$\vee$' B_2 ')'	– somma conteggi $p_0 = p_1 + p_2$ $d_0 = d_1 + d_2$
4: $B_0 \to$ '(' B_1 '$\wedge$' B_2 ')'	– somma conteggi $p_0 = p_1 + p_2$ $d_0 = d_1 + d_2$
5: $B_0 \to$ 'lp'	– inizializza conteggi $p_0 = 1$ $d_0 = 0$

La grammatica è puramente sintetizzata e dunque a una passata.

(b) Si propaga l'attributo ereditato *neg* da radice a foglie, commutandolo ogniqualvolta si trova una negazione (nella radice è inizializzato a falso - zero ossia numero pari di negazioni). Risalendo da foglie a radice si calcola la traduzione: se numero dispari applica negazione a letterale e cambia and in or o or in and; e cancella tutte le negazioni intermedie. Ecco la soluzione, che si avvale solo dei due attributi dati:

	sintassi	calcolo attributi (domanda (b))
1:	$S_0 \rightarrow B_1$	– inizializza parità di # di neg. annidate $neg_1 = false$ $\tau_0 = \tau_1$
2:	$B_0 \rightarrow$ '$\neg$' B_1	– commuta parità di # di neg. annidate $neg_1 =$ **not** neg_0 $\tau_0 = \tau_1$

3:	$B_0 \rightarrow$ '(' B_1 '$\vee$' B_2 ')'	$neg_1 = neg_0$ $neg_2 = neg_0$
		if $(neg_0 == true)$ **then**
		– cambia $\vee$ in $\wedge$
		$\tau_0 =$ **cat** ('(', τ_1, '$\wedge$', τ_2, ')')
		else
		$\tau_0 =$ **cat** ('(', τ_1, '$\vee$', τ_2, ')')
		endif

4:	$B_0 \rightarrow$ '(' B_1 '$\wedge$' B_2 ')'	$neg_1 = neg_0$ $neg_2 = neg_0$
		if $(neg_0 == true)$ **then**
		– cambia $\wedge$ in $\vee$
		$\tau_0 =$ **cat** ('(', τ_1, '$\vee$', τ_2, ')')
		else
		$\tau_0 =$ **cat** ('(', τ_1, '$\wedge$', τ_2, ')')
		endif

5:	$B_0 \rightarrow$ 'lp'	**if** $(neg_0 == true)$ **then**
		– applica $\neg$ al letterale
		$\tau_0 =$ **cat** ('$\neg$', 'lp')
		else
		$\tau_0 =$ 'lp'
		endif

Come d'uso la funzione **cat** concatena in una sola stringa la lista di argomenti di tipo stringa, siano essi stringhe costanti o attributi di tipo stringa.

La grammatica è a una passata: in discesa si calcola e propaga l'attributo ereditato *neg* e in risalita quello sintetizzato τ.