Linguaggi Formali e Compilatori
(Formal Languages and Compilers)

prof. S. Crespi Reghizzi, prof.ssa L. Sbattella
(prof. Luca Breveglieri)

Prova scritta - 7 luglio 2006 - Parte I: Teoria

CON SOLUZIONI

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 e automi a pila
    3. analisi sintattica e parsificatori
    4. traduzione e analisi semantica
  - II (20%) Esercitazioni Flex e Bison
- Per superare l'esame l'allievo deve avere sostenuto con successo entrambe le parti (I e II), in un solo appello oppure in appelli diversi, ma entro due sessioni d'esame.
- Per superare la parte I (teoria) occorre dimostrare di possedere sufficiente conoscenza di tutte le quattro sezioni (1-4).
- 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 date due espressioni regolari:

L_1	L_2
$a\,((b \mid c)\,a)^+$	$a\,((c \mid d)\,a)^*$

(a) Si scrivano le tre stringhe più corte appartenenti alle differenze insiemistiche:

$L_1 \setminus L_2$	$L_2 \setminus L_1$

(b) Si verifichi se l'espressione L_1 sia ambigua.

Soluzione

(a) Confrontando L_1 con la classica lista $e\,(s\,e)^+$ si vede che il primo linguaggio consiste di liste aventi almeno due elementi a, intercalati dal separatore b o c.
Similmente, confrontando L_2 con l'espressione classica $(e\ s)^*\,e$ è evidente che il secondo linguaggio contiene le liste formate da almeno un elemento a, con il separatore c o d.
Di conseguenza si possono descrivere le differenze insiemistiche così:

$L_1 \setminus L_2$ $(= L_1 - L_2 = L_1 \cap \neg L_2)$	$L_2 \setminus L_1$ $(= L_2 - L_1 = L_2 \cap \neg L_1)$
liste di almeno due elementi, con almeno un separatore b	lista di un elemento e liste con almeno un separatore d
aba, *ababa*, *abaca*, *acaba*	*a*, *ada*, *adaca*, *acada*, *adada*

(b) Ambiguità: l'espressione L_1 ha la stessa struttura della lista astratta $e\,(s\,e)^+$, che si sa essere inambigua. Ciò significa che per ogni lista, con riferimento all'espressione numerata $e_1\,(s_2\,e_3)^+$, ogni comparsa della lettera s è numerata 2 e ogni comparsa della lettera e è numerata 1 o 3, in modo mutuamente esclusivo. Del resto è facile vedere che la prima comparsa di e è numerata 1 e che le successive sono numerate 3. Ne viene che una data lista può dunque avere una sola numerazione e ciò non lascia spazio ad alcuna forma di ambiguità.

Volendo precisare, si osservi che l'espressione regolare di L_1 si traduce nella lista astratta tramite la traslitterazione seguente:

$$a \to e \qquad b \to s \qquad c \to s$$

Data allora una lista astratta numerata, come per esempio la seguente:

$$e_1\ s_2\ e_3\ s_2\ e_3$$

la traslitterazione inversa produce le liste numerate seguenti:

$$a_1\ b_2\ a_3\ b_2\ a_3 \qquad a_1\ b_2\ a_3\ c_2\ a_3 \qquad a_1\ c_2\ a_3\ b_2\ a_3 \qquad a_1\ c_2\ a_3\ c_2\ a_3$$

Tali liste chiaramente conservano la proprietà di non permettere di numerare una stessa frase in due modi diversi, com detto prima in generale. In conclusione l'espressione data non è ambigua.

2. Il linguaggio L di alfabeto Σ seguente:

   $$\Sigma = \{\ v,\ \text{`}+\text{'},\ \text{`)'},\ \text{`('}\ \}$$

   contiene le espressioni aritmetiche con la variabile v, l'operatore somma "+" ed eventualmente le parentesi tonde, ma con il vincolo che le parentesi non possano essere racchiuse da parentesi. Esempi:

   $$v \qquad v+v \qquad (v+v+v) \qquad (v+v+v)+(v+v)+v \qquad (v)+(v+v)$$

   Si svolgano i punti seguenti:

   (a) Si scriva un'espressione regolare non ambigua del linguaggio L.
   (b) Si costruisca l'automa riconoscitore del linguaggio $\neg L$ (il complemento di L), spiegando il procedimento seguito per ottenerlo.

   Nota: l'automa costruito può essere det. o non-det., a scelta.

Soluzione

(a) Conviene vedere il linguaggio come una lista a due livelli. Al primo livello vi è una lista astratta di elementi appartenenti a L', separati dal segno "+":

$$L' = L''\ \left(\text{`}+\text{'}\ L''\right)^*$$

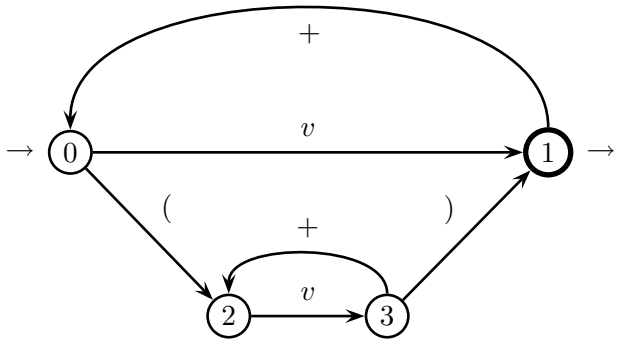
Al secondo livello L'' si sviluppa come:

$$L'' = v\ \mid\ \text{`('}\ v\ \left(\text{`}+\text{'}\ v\ \right)^*\ \text{`)'}$$

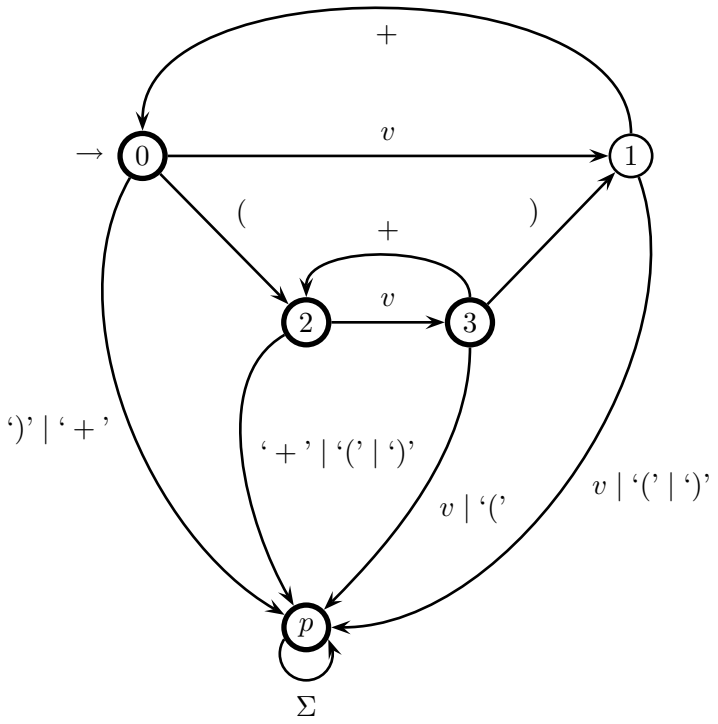
Per dimostrare che l'espressione regolare è inambigua si osservi quanto segue:

- ogni sottostringa parentesizzata, come per esempio $(v+v+v)$, è derivabile in un solo modo dalla e.r. L'', la forma classica delle liste associative a destra
- ogni stringa valida è definita dalla L' in un solo modo, essendo anche tale e.r. quella classica delle liste associative a destra

(b) Per costruire il riconoscitore del complemento in modo sistematico conviene prima costruire il riconoscitore M deterministico di L e poi applicargli la costruzione dell'automa complementare.



Verificato che l'automa è deterministico (ed è anche palesemente minimo), dopo averlo completato con lo stato pozzo p, scambiando gli stati finali e non si ottiene il riconoscitore del complemento (usando gli apici qua e là, dove serve per evitare confusione con il metasimbolo "|").



Gli stati finali sono ora 0 (che è anche iniziale), 2, 3 e p. L'automa potrebbe non essere minimo (il lettore esamini se sia vero o no).

2 Grammatiche libere e automi a pila 20%

1. Si consideri il linguaggio delle espressioni con operatori binari "+" e "×" (somma e prodotto), variabili schematizzate come "*i*", e parentesi tonde "(" e ")". Si deve scrivere la grammatica BNF (non estesa) delle espressioni dove in ogni termine il numero di segni "×" sia pari o nullo.

   Esempi: $i \quad i+i+i \quad i+i\times(i+i)\times i$

   Controesempi: $i\times i+i \quad i\times(i+i\times i)\times i$

Soluzione

Ecco la grammatica, che è molto semplice: si mette un vincolo di parità sul numero di segni "×" in ciascun termine separatamente.

$$G \begin{cases} S & \to & E \\ E & \to & T_1 \text{ ‘+’ } E \mid T_1 \\ T_1 & \to & F \text{ ‘×’ } T_2 \mid F \\ T_2 & \to & F \text{ ‘×’ } T_1 \\ F & \to & i \mid \text{‘(’ } E \text{ ‘)’} \end{cases}$$

Il nonterminale T_1, che dà origine al termine, può generare solo un numero pari di segni "×" (cioè 0, 2, 4, ecc). Per il resto la grammatica ha la solita forma BNF (non estesa) non ambigua di una generica grammatica per espressioni aritmetiche.

2. Si progetti la grammatica EBNF non ambigua che modella il linguaggio, semplificato, delle espressioni della teoria elementare dei vettori di elementi scalari. Sono presenti i concetti seguenti:

   - gli elementi scalari sono schematizzati con il puro terminale s
   - una lista (non vuota) di elementi scalari separati da "," e racchiusa tra parentesi quadre "[" e "]" è un vettore
   - uno scalare si assimila a un vettore monodimensionale e può non avere quadre
   - tra vettori sono consentite le operazioni seguenti (si scrive *vett.* per indicare un vettore generico):
     (a) prodotto di scalare per vettore $s \cdot vett.$ o $vett. \cdot s$, con risultato *vettoriale*
     (b) prodotto interno (o scalare) di due vettori $\langle vett., vett. \rangle$, con risultato *scalare*
     (c) prodotto esterno (o vettoriale) di due vettori $vett. \vee vett.$, con risultato *vettoriale*
     (d) somma $vett. + vett.$ di vettori, con risultato *vettoriale*
     (e) somma diretta $vett. \oplus vett.$ di vettori (ovvero concatenamento di vettori), con risultato *vettoriale*

     precedenze: (a) precede (b), che precede (c), che precede (d), che precede (e)
   - sono ammesse sotto-espressioni parentetizzate mediante "(" e ")"

   Esempi: $s \quad [s, s] \quad [s] \cdot s \quad \langle [s, s], [s] \vee [s, s] \rangle \quad ([s, s] \vee [s, s]) \oplus [s, s, s]$

   Si scriva la grammatica G in questione (in forma EBNF non ambigua). Quali aspetti semantici non sono esprimibili sintatticamente ?

Soluzione

La grammatica EBNF richiesta è abbastanza ovvia. Eccone una forma possibile:

$$
G \begin{cases}
S & \to & E \\
E & \to & T_1 \ (\text{`}\oplus\text{'}\ T_1\ )^* \\
T_1 & \to & T_2 \ (\text{`}+\text{'}\ T_2\ )^* \\
T_2 & \to & T_3 \ (\text{`}\vee\text{'}\ T_3\ )^* \\
T_3 & \to & \text{`}\langle\text{'}\ T_4\ \text{`,'}\ T_4\ \text{`}\rangle\text{'} \ \mid \ T_4 \\
T_4 & \to & N\ \text{`}\cdot\text{'}\ T_5 \ \mid \ T_5\ \text{`}\cdot\text{'}\ N \ \mid \ T_5 \\
T_5 & \to & N \ \mid \ V \ \mid \ \text{`('}\ E\ \text{`)'} \\
V & \to & \text{`['}\ s\ (\text{`,'}\ s\ )^+\ \text{`]'} \\
N & \to & s \ \mid \ \text{`['}\ s\ \text{`]'}
\end{cases}
$$

Essa genera le espressioni richieste, con le precedenze corrette tra operatori; inoltre nel caso dell'operatore "$\cdot$" tiene conto se uno dei due argomenti sia uno scalare (o un vettore monodimensionale, assimilabile a uno scalare). A grandi linee la grammatica data qui si ispira al modello standard di grammatica EBNF non ambigua di espressioni aritmetiche, ampliando l'insieme di operatori disponibili.

Con tale grammatica, e in generale con una qualsiasi grammatica non contestuale, non si possono però modellare gli aspetti semantici legati alle operazioni che richiedono argomenti vettore della stessa dimensione; cioè "$\langle\ \rangle$", "$\vee$" e "$+$"; non si ha invece nessun problema per "$\oplus$" perché non presenta tale vincolo.

Nota bene: tutte le operazioni tranne "$\oplus$" danno risultato monodimensionale se gli operandi sono monodimensionali, e inoltre il prodotto interno "$\langle\ \rangle$" dà sempre risultato scalare (ovvero vettore monodimensionale); si potrebbe pertanto modellare più finemente la semantica distinguendo in modo sistematico tra vettori a una dimensione (cioè scalari) e a più dimensioni; qui tale distinzione è fatta solo per l'operatore "$\cdot$" ... si lascia l'esercizio al lettore, la grammatica sarà più voluminosa ma più precisa.

3 Analisi sintattica e parsificatori 20%

1. È data la grammatica estesa (EBNF) G seguente:

   $$S \to (\ a\ b\ )^+\ \ c^*\ \mid\ a\ S\ c$$

   Si svolgano i punti seguenti:

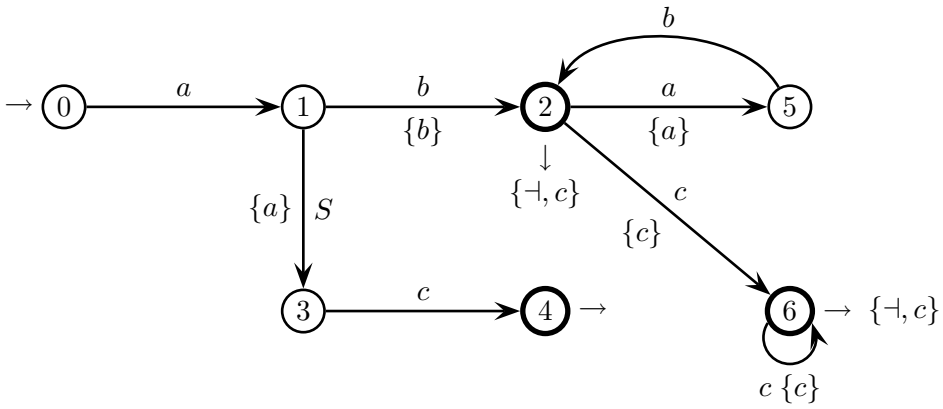
   (a) Si disegni la rete delle macchine ricorsive equivalente a G.

   (b) Si calcolino gli insiemi guida e si verifichi se la rete delle macchine sia $LL(k)$.

   (c) Se necessario, si trasformi la grammatica G per ottenere una grammatica equivalente $LL(k)$.

Soluzione

(a) Si disegna già nella forma deterministica la rete di automi, composta in realtà dalla sola macchina S:



(b) Gli insiemi guida rilevanti (dove ci sono biforcazioni) sono già sul grafo. Gli stati 2 e 6 violano la condizione $LL(1)$, a causa della presenza del carattere c sia sulle transizioni sia sulle frecce d'uscita. È piuttosto evidente che ciò non cambia con $k > 1$, a motivo del suffisso c^*, il quale genera $cc\ldots c$, e della ricorsione aSc, la quale genera anch'essa $cc\ldots c$. Dunque il conflitto persiste per $k > 1$.
Per comprendere meglio l'origine del conflitto si osservino le frasi seguenti:

$$\overbrace{a\ b\ c}^{S}\ \dashv \qquad a\ \overbrace{a\ b}^{S}\ c\ \dashv \qquad a\ a\ \overbrace{a\ b}^{S}\ c\ c\ \dashv \qquad \ldots$$

Anche aumentando la lunghezza k della prospezione, il conflitto permane.

(c) Per ottenere la grammatica $LL(k)$ si osservi il linguaggio generato da G:

$$L(G) = \{a^n\ (a\ b)^+\ c^*\ c^n \mid \quad n \geq 0\}$$

Esso esibisce lo stesso tipo di conflitto dell'esempio noto:

$$L_N = \{a^n\ c^*\ c^n \mid \quad n \geq 0\} = \{a^m\ c^n \mid \quad n \geq m \geq 0\}$$

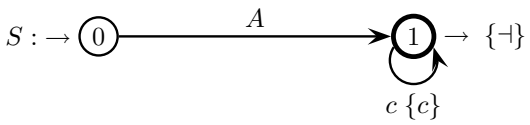
Ma si sa che quest'ultimo si può anche scrivere, spostando il termine c^* fuori dalla ricorsione, nella forma seguente:

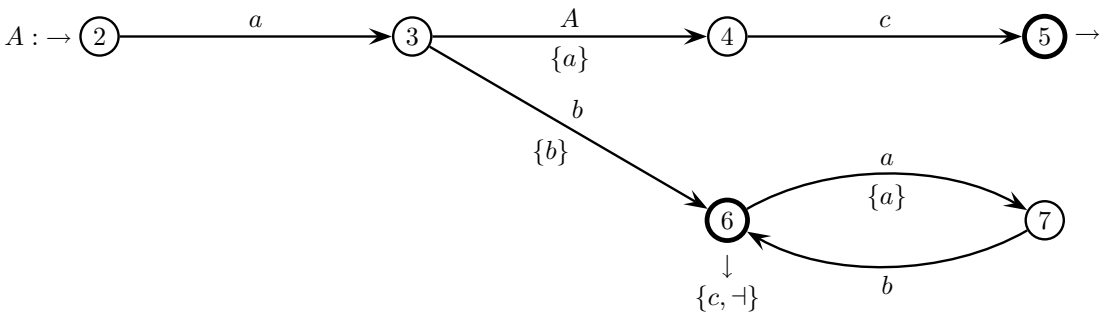
$$L_N = \{a^n\ c^n\ c^* \mid \quad n \geq 0\}$$

da dove si trova facilmente una grammatica LL(1).
Applicando la stessa idea al problema, si ottiene la grammatica equivalente G':

$$S \to A\ c^* \qquad\qquad A \to a\ A\ c\ \mid\ (a\ b)^+$$

con la corrispondente rete di macchine:





Ora gli insiemi guida sono disgiunti in tutti gli stati di scelta.

2. È data la grammatica G seguente:

$$S \rightarrow abc \mid ab \mid a \mid Acd \mid Ace$$
$$A \rightarrow ab \mid \varepsilon$$

Nota: non è permesso modificare tale grammatica.

Si svolgano i punti seguenti:

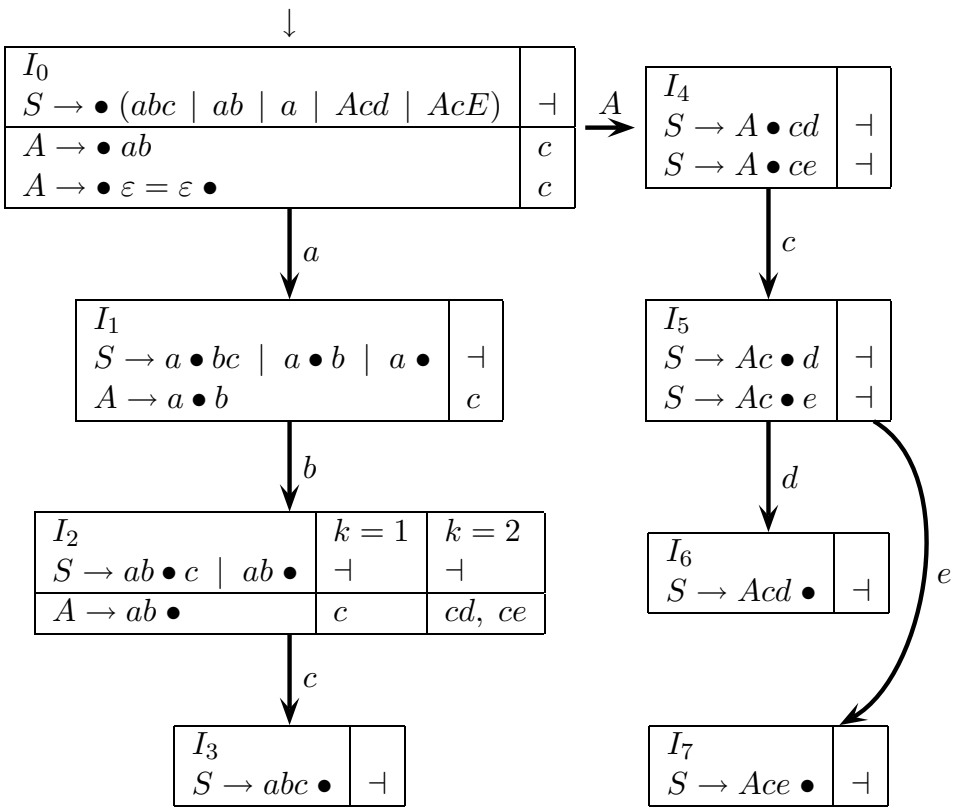
(a) Per G si costruisca l'automa pilota $LR(1)$ (cioè il riconoscitore dei prefissi).

(b) Per ogni macrostato del pilota si indichi se per $k = 1$ vi siano conflitti, di tipo spostamento-riduzione o di tipo riduzione-riduzione.

(c) {facoltativo} Per ogni macrostato si indichi quale sia il valore *minimo* di k ($k \geq 0$) necessario per renderlo deterministico.

Soluzione

(a) Ecco la macchina pilota $LR(k)$ (con k fino a 2 dove serve):



(b) Per il valore di profondità di prospezione $k = 0$ (cioè senza prospezione) si ha quanto segue:

- i macrostati I_3, I_4, I_5, I_6 e I_7 sono deterministici (nessun conflitto)
- i macrostati I_0 e I_1 hanno entrambi un conflitto di tipo spostamento-riduzione
- e il macrostato I_2 ha due conflitti:
  - uno di tipo spostamento-riduzione (o se si vuole ha due conflitti, considerando le due riduzioni separatamente in rapporto allo spostamento; ma non occorre distinguere così finemente)
  - e uno di tipo riduzione-riduzione

Dunque G non è $LR(0)$.

Per risolvere il conflitto spostamento-riduzione sia di I_0 sia di I_1 basta la prospezione con profondità $k = 1$ (vedi grafo pilota), ma per I_2 il valore $k = 1$ non basta a risolvere né il conflitto spostamento-riduzione né quello riduzione-riduzione, perché c'è un arco uscente etichettato con il carattere c e le due riduzioni hanno entrambe prospezione c, rispettivamente; esse sono pertanto in conflitto con lo spostamento e tra sé. Dunque G non è neppure $LR(1)$.

(c) Entrambi i conflitti di I_2 si risolvono invece con prospezione di profondità $k = 2$ perché il cammino uscente è etichettato con il carattere c e successivamente (nel macrostato I_3) c'è riduzione con prospezione $\dashv$, ovvero si ha $c \dashv$ (vedi grafo sopra), mentre le due prospezioni di riduzione diventano cd e ce; pertanto i loro conflitti con lo spostamento e tra sé scompaiono. Dunque G è $LR(2)$.

4 Traduzione e analisi semantica 20%

1. Si considerino come linguaggio sorgente L_1 le espressioni aritmetiche definite dalla grammatica EBNF non ambigua G_1 seguente:

$$G_1 \begin{cases} E & \rightarrow & T\ (\text{`}+\text{'}\ T\ )^* \\ T & \rightarrow & F\ (\text{`}\times\text{'}\ F\ )^* \\ F & \rightarrow & a \end{cases}$$

Le espressioni contengono variabili, schematizzate con a, e gli operatori binari infissi somma "+" e prodotto "×", ma non parentesi. Nell'ordine di calcolo il prodotto precede la somma.

Esempi: $a + a$ $a + a + a$ $a \times a$ $a + a \times a$ $a \times a + a$

Si svolgano i punti seguenti:

(a) Si scrivano tre esempi di espressione infissa la cui traduzione in forma postfissa contenga la coppia di operatori "× +" adiacenti.

(b) Si scriva, modificando se necessario la grammatica G_1, lo schema sintattico di traduzione G_2 EBNF non ambiguo che traduce in forma postfissa le espressioni di L_1, ma dove ogni coppia di operatori "× +" adiacenti sia rimpiazzata dal terminale singolo "$m\&a$", il quale rappresenta l'operazione "multiply-and-add".

Soluzione

(a) Condizione necessaria e sufficiente affinché la forma postfix contenga la coppia adiacente "× +" è che la forma infix contenga un termine avente struttura di tipo seguente (con $k \geq 1$):

$$\ldots\ a\ \underbrace{+_1\ a\ \times_2\ a\ \ldots\ a\ \times_k\ a}_{\text{termine}}\ \ldots$$

Il termine deve essere preceduto dal segno "+" e deve contenere almeno un segno "×". Traducendo in forma postfix diventa:

$$\ldots\ a\ \underbrace{a\ a\ \times_2\ \ldots\ a\ \times_k\ +_1}_{\text{termine}}\ \cdots = \ldots\ a\ \underbrace{a\ a\ \times_2\ \ldots\ a\ m\&a_{k,1}}_{\text{termine}}\ \ldots$$

Esempi (numerando gli operatori e cambiando i nomi di variabili per rintracciarli facilmente nella forma postfissa):

$$a\ +_1\ b \times_2 c = abc \times_2\ +_1\ = abc\ m\&a_{2,1}$$

$$a\ +_1\ b\ \times_2\ c \times_3 d = abc\ \times_2\ d \times_3\ +_1\ = abc\ \times_2\ d\ m\&a_{3,1}$$

$$a \times_1 b\ +_2\ c \times_3 d\ +_4\ e = ab \times_1 cd \times_3 +_2\ e\ +_4\ = ab \times_1 cd\ m\&a_{3,2}\ e\ +_4$$

e così via.

(b) Si procede per gradi, allungando un po' il procedimento che di per sé è semplice. Primo passo: grammatica G_1' EBNF non ambigua di traduzione in forma postfix qualunque (si ottiene spostando l'operatore a destra del secondo operando):

$$G_1' \begin{cases} & & \text{sorgente} & \text{pozzo} \\ E & \to & T\ (\text{`}+\text{'}\ T\ )^* & T\ (T\ \text{`}+\text{'})^* \\ T & \to & F\ (\text{`}\times\text{'}\ F\ )^* & F\ (F\ \text{`}\times\text{'})^* \\ F & \to & a & a \end{cases}$$

Secondo passo: grammatica G_1'' EBNF non ambigua sorgente che espone e isola ogni sottoespressione formata da un termine preceduto da segno "+" e contenente almeno un segno "×", l'ultimo dei quali è quello da esporre (la nuova grammatica G_1'' è equivalente a G_1'):

$$G_1'' \begin{cases} & & \text{sorgente} \\ E & \to & T \left(\underbrace{\text{`}+\text{'}\ T\ \text{`}\times\text{'}\ F}_{\substack{\text{ecco la subsepr.} \\ \text{“}+\ a\ \ldots\ \times\ a\text{”}}} \ \mid\ (\text{`}+\text{'}\ F\ )^* \right)^* \\ T & \to & F\ (\text{`}\times\text{'}\ F\ )^* \\ F & \to & a \end{cases}$$

Terzo passo: la grammatica di traduzione richiesta G_2 EBNF non ambigua è allora la seguente (solo parte pozzo, ottenuta come prima):

$$G_2 \begin{cases} & & \text{pozzo} \\ E & \to & T \left(\underbrace{T\ F\ \text{`}\times\text{'}\ \text{`}+\text{'}}_{\text{eccola}} \ \mid\ (F\ \text{`}+\text{'})^* \right)^* \\ T & \to & F\ (F\ \text{`}\times\text{'})^* \\ F & \to & a \end{cases}$$

vale a dire (sostituendo $m\&a$ e usando apici):

$$G_2 \begin{cases} & & \text{pozzo} \\ E & \to & T \left(\underbrace{T\ F\ \text{`}m\&a\text{'}}_{\text{eccola}} \ \mid\ (F\ \text{`}+\text{'})^* \right)^* \\ T & \to & F\ (F\ \text{`}\times\text{'})^* \\ F & \to & a \end{cases}$$

e G_2 si può considerare la soluzione completa al problema proposto.

Volendo si possono combinare la parte sorgente e pozzo, nel solito modo. Eccolo (si omettono gli apici per abbreviare):

$$G_2 \left\{ \begin{array}{lcl} & & \text{sorgente e } \{\text{pozzo}\} \\ E & \rightarrow & T \left(\underbrace{+\, T \times F \ \{m\&a\}}_{\text{eccola}} \mid (+\, F \ \{+\})^* \right)^* \\ T & \rightarrow & F \ (\times\, F \ \{\times\})^* \\ F & \rightarrow & a \ \{a\} \end{array} \right.$$

Nota: il linguaggio L_1 è regolare ed è facile convincersi che la traduzione richiesta si può modellare anche mediante un trasduttore a stati finiti; il lettore provi a farlo.

Nota: naturalmente ci possono essere altre soluzioni di tipo grammaticale, ma quella presentata qui appare essere bene armonizzata con la forma EBNF standard di espressioni senza parentesi data nel testo, nel senso che è una specializzazione della forma standard; altre soluzioni sono possibili ma probabilmente meno naturali da leggere, ancorché magari del tutto corrette.

Addendum: si potrebbe usare G_2 come traduttore di espressioni per una macchina a pila potenziata con l'istruzione "$m\&a$", la quale spila tre elementi, ne calcola la multiply-and-add e ne impila il risultato.

Addendum: si vede subito che le espressioni di tipo (con $n \geq 1$):

$$c + a_1 \times b_1 + a_2 \times b_2 \ + \ \ldots \ + a_n \times b_n$$

si traducono in:

$$c \ a_1 \ b_1 \times + \ a_2 \ b_2 \times + \ \ldots \ a_n \ b_n \times +$$

ovvero sono interamente calcolabili tramite "$m\&a$", senza "$+$" e "$\times$" separati:

$$c \ a_1 \ b_1 \ m\&a \ a_2 \ b_2 \ m\&a \ \ldots \ a_n \ b_n \ m\&a$$

e sono le sole espressioni ad avere tale proprietà. Esse di fatto corrispondono al calcolo del prodotto scalare (o interno) di vettori $A = [a_1, \ldots, a_n]$ e $B = [b_1, \ldots, b_n]$, più la costante c:

$$c + \langle A, B \rangle = c \ a_1 \ b_1 \ m\&a \ a_2 \ b_2 \ m\&a \ \ldots \ a_n \ b_n \ m\&a$$

Si provi a validare la trasduzione tramite tali particolari espressioni.

Nota: spesso i processori dispongono di un'istruzione macchina "MADD" (multiply-and-add), tipo "MADD D1, D2, D3" dove D3 $\leftarrow$ [D1] $\times$ [D2] + [D3][1], proprio allo scopo di ottimizzare il calcolo del prodotto scalare di vettori (e di tutte le operazioni similari: convoluzione discreta, filtraggio lineare, ecc); l'espressione precedente si codificherebbe per esempio così:

codice		commento
MOVE	c, D3	# D3 $\leftarrow$ [c]
MOVE	a1, D1	# D1 $\leftarrow$ [a1]
MOVE	b1, D2	# D2 $\leftarrow$ [b1]
MADD	D1, D2, D3	# D3 $\leftarrow$ [D1] $\times$ [D2] + [D3]
MOVE	a2, D1	# D1 $\leftarrow$ [a2]
MOVE	b2, D2	# D2 $\leftarrow$ [b2]
MADD	D1, D2, D3	# D3 $\leftarrow$ [D1] $\times$ [D2] + [D3]
...		

La grammatica di traduzione G_2 potrebbe dunque fornire il supporto sintattico per la generazione, con l'aggiunta di attributi semantici opportuni, di codice macchina ottimizzato nel senso appena detto.

[1] La coppia di parentesi quadre "[ ]" indica il contenuto dell'argomento; per esempio [D3] indica il contenuto del registro D3 e [a1] indica il contenuto della cella di memoria di indirizzo a1; è la notazione RTL (register transfer language) standard, si trova in tutti i libri dedicati alla struttura del processore.

2. Si consideri il prodotto di uno scalare s per una matrice rettangolare di ordine $r \times c$:

$$s \times \underbrace{\begin{bmatrix} e_{1,1} & e_{1,2} & \dots & e_{1,c} \\ e_{2,1} & e_{2,2} & \dots & e_{2,c} \\ \dots & \dots & \dots & \dots \\ e_{r,1} & e_{r,2} & \dots & e_{r,c} \end{bmatrix}}_{\substack{\text{matrice di partenza} \\ \text{(generata sintatticamente)} \\ \text{(check semantico righe)}}} = \underbrace{\begin{bmatrix} se_{1,1} & se_{1,2} & \dots & se_{1,c} \\ se_{2,1} & se_{2,2} & \dots & se_{2,c} \\ \dots & \dots & \dots & \dots \\ se_{r,1} & se_{r,2} & \dots & se_{r,c} \end{bmatrix}}_{\substack{\text{matrice risultato} \\ \text{(da generare come traduzione)}}}$$

È dato un supporto sintattico G che genera il membro sinistro: scalare e matrice di partenza. Lo scalare s e gli elementi e della matrice sono numeri interi (qui sono trattati come terminali). La matrice è codificata e generata come lista (non vuota) di righe con sep. ";" e la riga è una lista (non vuota) di elementi con sep. ",".

$$G \begin{cases} S & \to & s \text{ ' } \times \text{ ' } \text{'[' } M \text{ ']'} \\ M & \to & R \text{ ';' } M \\ M & \to & R \\ R & \to & E \text{ ',' } R \\ R & \to & E \\ E & \to & e \end{cases}$$

esempio di matrice generata:
$[\ M\ ] \overset{*}{\underset{G}{\Rightarrow}} [e; e, e, e, e; e, e]$

che codifica: $\begin{bmatrix} e & & & \\ e & e & e & e \\ e & e & & \end{bmatrix}$

Come si vede da sopra, il supporto sintattico G non assicura che le righe abbiano tutte la stessa lunghezza e non genera la matrice risultato: membro destro dell'uguaglianza.

Si chiede pertanto di progettare una grammatica con attributi G', estendendo la sintassi G con regole semantiche, che controlli se la matrice M abbia righe di uguale lunghezza (check semantico righe) e, se è così, emetta come traduzione la matrice risultato, altrimenti non emetta nulla. Ci devono essere almeno gli attributi seguenti:

- un predicato di correttezza "φ_S", vero se e solo se la matrice M è corretta
- una funzione di traduzione "τ_E" che, se M è corretta, emette come traduzione il prodotto del valore dell'elemento E per lo scalare s, altrimenti niente

Per esempio, φ applicato alla derivazione di M data sopra risulterebbe falso. Invece, un esempio corretto è: $S \overset{*}{\underset{G}{\Rightarrow}} s \times [e, e, e; e, e, e]$, dove $\varphi =$ vero.

Si possono estendere gli attributi φ e τ ad altri nonterminali e introdurre nuovi attributi, come sembra necessario od opportuno.

Ecco i punti da svolgere:

(a) Elencare gli attributi, con il rispettivo tipo e significato.
(b) Scrivere le funzioni semantiche che calcolano gli attributi (alle pagine successive sono già pronti gli schemi da compilare).
(c) Disegnare i grafi delle dipendenze funzionali tra attributi, per ciascuna produzione separatamente.
(d) Stabilire se la grammatica sia di tipo a una sola scansione.
(e) Stabilire se la grammatica sia di tipo L.

Soluzione

(a) Intuitivamente occorrono gli attributi importanti seguenti (oltre a φ e τ): un attributo η ereditato per propagare lo scalare s sugli elementi della matrice; un attributo λ sintetizzato per calcolare la lunghezza della riga; e un attributo ρ ereditato per propagare il predicato di correttezza sugli elementi della matrice, dove serve per calcolare la traduzione τ; più eventualmente qualche attributo di servizio, dalle funzioni limitate (p. es. σ, vedi sotto).

Ecco la tabella degli attributi:

tipo	nome	nonterminali	dominio	significato
		già dati		
sx	φ	S	$\{\mathrm{V}, \mathrm{F}\}$	predicato di correttezza
sx	τ	E	intero	prodotto scalare $\times$ elemento
		da aggiungere o estendere		
sx	φ	M	$\{\mathrm{V}, \mathrm{F}\}$	estensione di φ a M
sx	σ	S	intero	valore dello scalare s
dx	η	M, R, E	intero	valore ereditato di σ
sx	λ	M, R	intero (≥ 1)	lunghezza della riga
dx	ρ	M, R, E	$\{\mathrm{V}, \mathrm{F}\}$	valore ereditato di φ

(b) Le funzioni semantiche non presentano diffcoltà soverchie. Eccole:

sintassi	*funzioni semantiche*
$S_0 \to s$ ' $\times$ ' '[' M_1 ']'	$\sigma_0 =$ valore di s $\eta_1 = \sigma_0$ $\varphi_0 = \varphi_1$ $\rho_1 = \varphi_0$
$M_0 \to R_1$ ';' M_2	$\eta_1 = \eta_0$ $\eta_2 = \eta_0$ $\varphi_0 = \begin{Bmatrix} \text{vero} & \text{se } \lambda_1 = \lambda_2 \\ \text{falso} & \text{se } \lambda_1 \neq \lambda_2 \end{Bmatrix} \wedge \varphi_2$ $\lambda_0 = \lambda_2$ $\rho_1 = \rho_0$ $\rho_2 = \rho_0$
$M_0 \to R_1$	$\varphi_0 =$ vero $\eta_1 = \eta_0$ $\lambda_0 = \lambda_1$ $\rho_1 = \rho_0$

sintassi	*funzioni semantiche*
$R_0 \to E_1$ ‘,’ R_2	$\eta_1 = \eta_0$ $\eta_2 = \eta_0$ $\lambda_0 = \lambda_1 + \lambda_2$ $\rho_1 = \rho_0$ $\rho_2 = \rho_0$
$R_0 \to E_1$	$\eta_1 = \eta_0$ $\lambda_0 = 1$ $\rho_1 = \rho_0$
$E_0 \to e$	$\tau_0 = \begin{cases} \text{stampa } \eta_0 \times e & \text{se } \rho_0 = \text{vero} \\ \text{stampa } \varepsilon & \text{se } \rho_0 = \text{falso} \end{cases}$

Volendo, si potrebbero propagare fino alla radice S dell'albero tutti i valori emessi da τ e lì aggregarli per costruire l'intera matrice risultato, invece di emettere la traduzione elemento per elemento; occorrerebbero attributi appositi, facili da definire. Non è richiesto dal testo e si lascia l'esercizio al lettore. È possibile che riformulando così il problema la soluzione risulti essere di tipo one-sweep se non addirittura di tipo L.

(c) Da fare ... (semplice, si lascia al lettore)

(d) La grammatica ad attributi G' non è di tipo a una sola passata (one-sweep), a motivo della necessità prima di propagare verso l'alto l'attributo sintetizzato φ (predicato di correttezza) e poi di ripropagarne verso il basso il valore (mediante l'attributo ereditato η), allo scopo di calcolare correttamente la traduzione τ di ogni nodo E dell'albero sintattico (infatti la traduzione τ dipende dal predicato φ, per come è definita). Naturalmente, la soluzione si calcola senz'altro in due passate ...

(e) No, G' non è di tipo L a maggior ragione per il punto precedente.