Linguaggi Formali e Compilatori
(Formal Languages and Compilers)

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

Prova scritta - 24 giugno 2008 - 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:

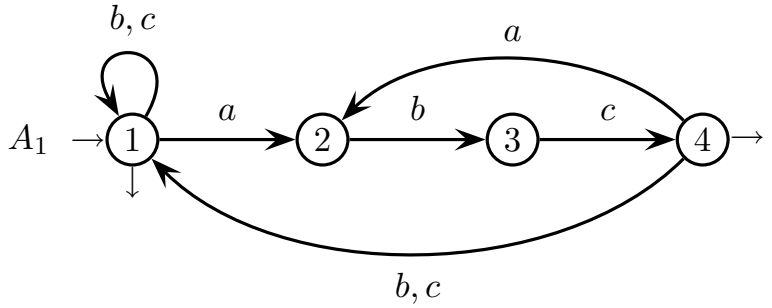
MATRICOLA: FIRMA:

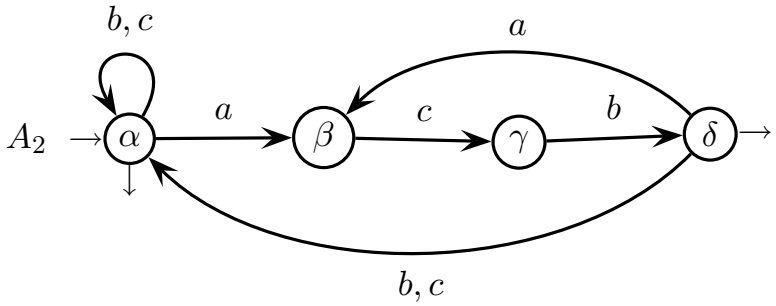
ISTRUZIONI - LEGGERE CON ATTENZIONE:

- 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 avere sostenuto 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. Si considerino i due automi finiti seguenti A_1 e A_2 che riconoscono i linguaggi L_1 e L_2, rispettivamente, entrambi di alfabeto $\{ a, b, c \}$.





Le stringhe dei due linguaggi L_1 e L_2 hanno la descrizione rispettiva seguente:

- in L_1 ogni lettera a è seguita prima da una lettera b e poi da una lettera c
- in L_2 ogni lettera a è seguita prima da una lettera c e poi da una lettera b

Si risponda alle domande seguenti:

(a) Si verifichi se gli automi A_1 e A_2 siano minimi, e se non lo sono li si minimizzi.

(b) Si ricavi un'espressione regolare R_3 che genera il linguaggio intersezione $L_3 = L_1 \cap L_2$, in uno a scelta tra i due modi seguenti: ragionando in modo informale sulla descrizione dei due linguaggi; oppure procedendo in modo algoritmico a partire dai due automi (per esempio trovando l'automa di L_3 e ricavando da quest'ultimo l'espressione regolare).

(c) (facoltativa) Si ricavino le espressioni regolari R_1 e R_2 che generano i due linguaggi L_1 e L_2, in uno a scelta tra i due modi seguenti: ragionando in modo informale sulla descrizione dei due linguaggi; oppure procedendo in modo algoritmico a partire dai due automi.

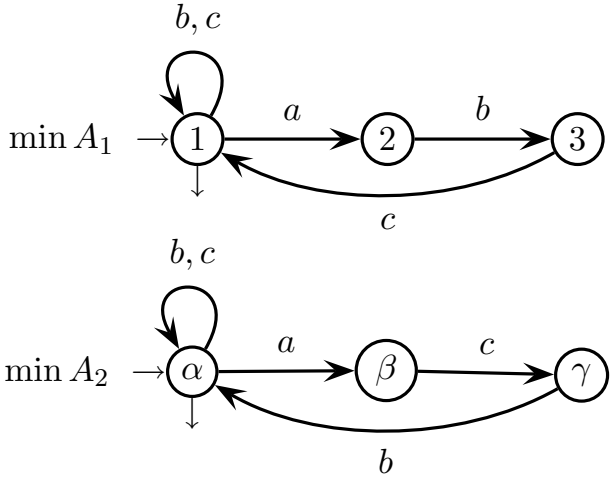
Soluzione

(a) Si vede subito che gli automi A_1 e A_2 non sono minimi. Per A_1 si ha infatti la relazione di indistinguibilità mostrata nella tabella triangolare seguente (a sinistra), che porta a ottenere l'automa minimo dove i due stati indistinguibili 1 e 4 sono fusi. In modo del tutto analogo si ottiene un automa minimo equivalente ad A_2, dove gli stati α e δ sono fusi (tabella triangolare a destra).

2	×		
3	×	×	
4	1 4 2 2	×	×
	1	**2**	**3**

β	×		
γ	×	×	
δ	$\alpha\,\delta$ $\beta\,\beta$	×	×
	α	β	γ

Ecco i grafi stato-transizione dei due automi A_1 e A_2 così minimizzati (dove le coppie di stati indistinguibili 1 4 e α δ sono rinominate 1 e α rispettivamente):



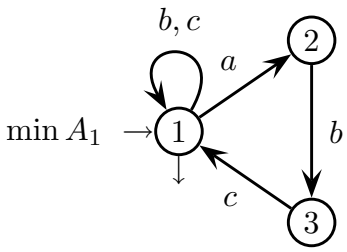
(b) Informalmente, dall'automa A_1 si vede che le stringhe del linguaggio L_1 sono costruite concatenando un numero arbitrario ≥ 0 di fattori presi fra i tre seguenti: $a\,b\,c$, b e c. Pertanto un'espressione regolare R_1 che genera L_1 è la seguente:

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

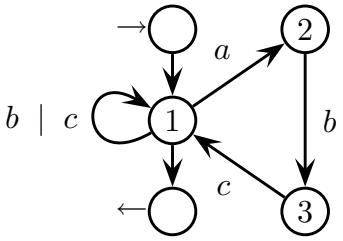
Con un ragionamento analogo, basandosi sulla somiglianza tra le descrizioni informali di L_1 e L_2 date nel testo, per il linguaggio L_2 si ricava l'espressione regolare R_2 seguente:

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

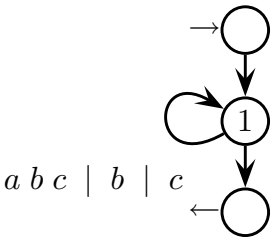
Algoritmicamente, per progettare le espressioni regolari R_1 e R_2 basta applicare il metodo di eliminazione dei nodi (algoritmo di Brozozowski) ai due automi A_1 e A_2 separatemente. Conviene naturalmente usare le forme minime trovate prima. Ecco il caso di A_1:



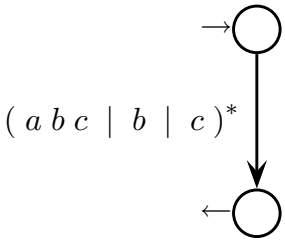
(1) inizio - automa originale



(2) nodi iniziale e finale unici e distinti



(3) eliminati i nodi 2 e 3



(4) eliminato il nodo 1 - fine

Pertanto si ottiene l'espressione regolare R_1 seguente:

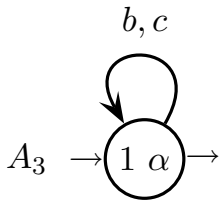
$$R_1 = (\ a\ b\ c\ \mid\ b\ \mid\ c\ )^*$$

Il caso di A_2 è analogo ed è lasciato al lettore. Infatti l'automa A_2 ha stati e connessioni identici a quelli di A_1, e solo le etichette sugli archi differiscono. Naturalmente per R_1 e R_2 si ottengono soluzioni equivalenti (qui addirittura identiche) a quelle ricavate in modo informale.

(c) Informalmente, dato che differiscono le stringhe $b\,c$ e $c\,b$ che nei due linguaggi L_1 e L_2 devono rispettivamente seguire ogni lettera a, quest'ultima non può mai comparire in nessuna stringa appartenente al linguaggio L_3, giacché questo è l'intersezione di L_1 e L_2. Pertanto il linguaggio L_3 contiene le sole stringe senza a, comuni a L_1 e L_2, ed è generato dall'espressione regolare R_3 seguente:

$$R_3 = (\ b\ \mid\ c\ )^*$$

Se si preferisce, il linguaggio intersezione L_3 è quello universale di alfabeto $\{b, c\}$. Algoritmicamente, nella costruzione dell'automa prodotto $A_3 = A_1 \times A_2$ si vede subito che lo stato prodotto $(1, \alpha)$, che è sia iniziale sia finale, è l'unico utile: leggendo la lettera a si va nello stato prodotto $(2, \beta)$, che è uno stato pozzo (non finale) e non va considerato; mentre leggendo la lettera b o c si rimane nello stesso stato $(1, \alpha)$. Pertanto l'automa prodotto A_3 è quello mostrato in figura (sotto):



Evidentemente l'automa A_2 (che ovviamente è minimo) accetta il linguaggio $(\ b\ \mid\ c\ )^*$, conformemente alla conclusione informale raggiunta prima.

2. È data l'espressione regolare R seguente, di alfabeto $\{a, b, c\}$:

$$R = \big(\; (\; a \; b \;)^+ \;\; | \;\; (\; a \;\; | \;\; c \;)^* \; \big) \;\; (\; c \; b \;\; | \;\; c \; a \;)^+$$

Si risponda alle domande seguenti:

(a) Dall'espressione regolare R si ricavi un riconoscitore finito deterministico A, utilizzando il metodo di Berry e Sethi.

(b) (facoltativa) Si verifichi se l'automa deterministico A ottenuto al punto (a) sia minimo e, in caso negativo, lo si minimizzi.

Soluzione

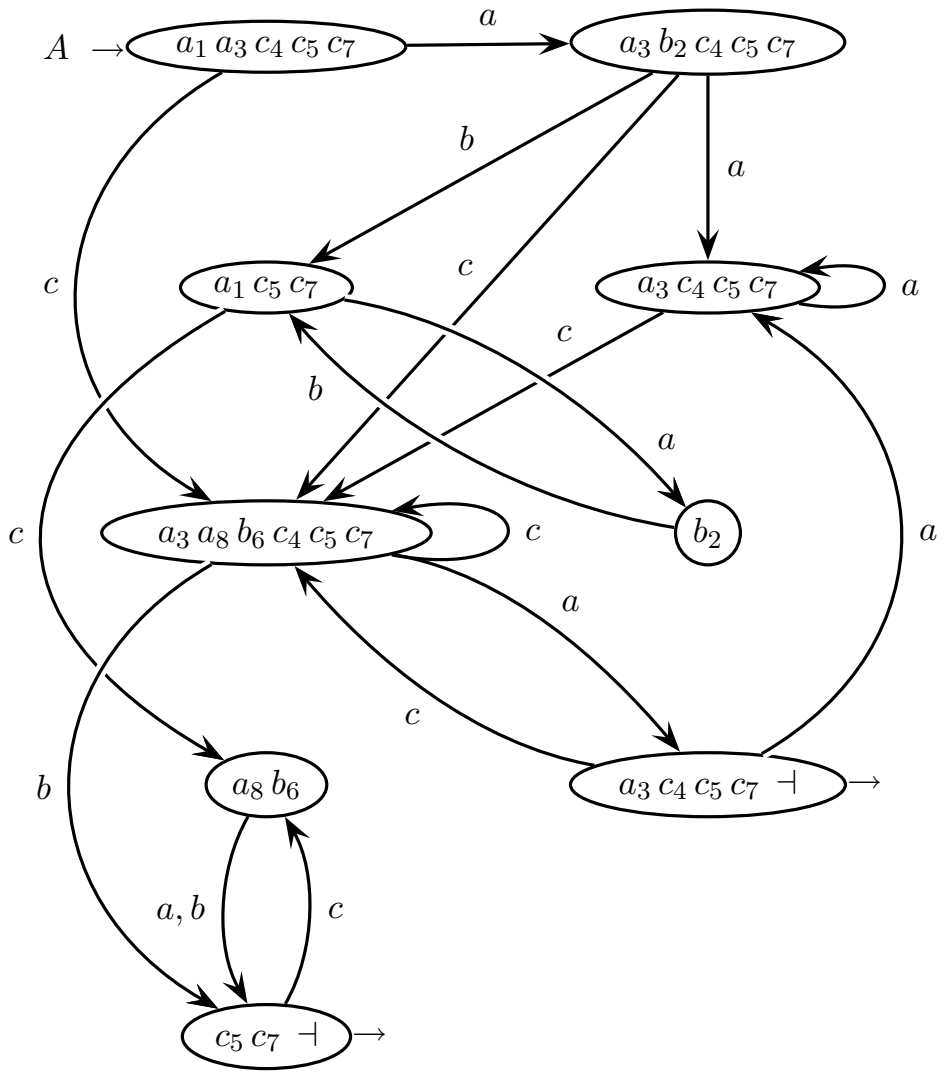
(a) Ecco il riconoscitore A ottenuto con il metodo di Berry e Sethi. Si dà prima la versione numerata R' dell'espressione regolare R (con terminatore $\dashv$):

$$R' = \big(\; (\; a_1 \; b_2 \;)^+ \;\; | \;\; (\; a_3 \;\; | \;\; c_4 \;)^* \; \big) \;\; (\; c_5 \; b_6 \;\; | \;\; c_7 \; a_8 \;)^+ \; \dashv$$

Poi si compilano l'insieme degli inizî e quelli dei séguiti dell'espressione R:

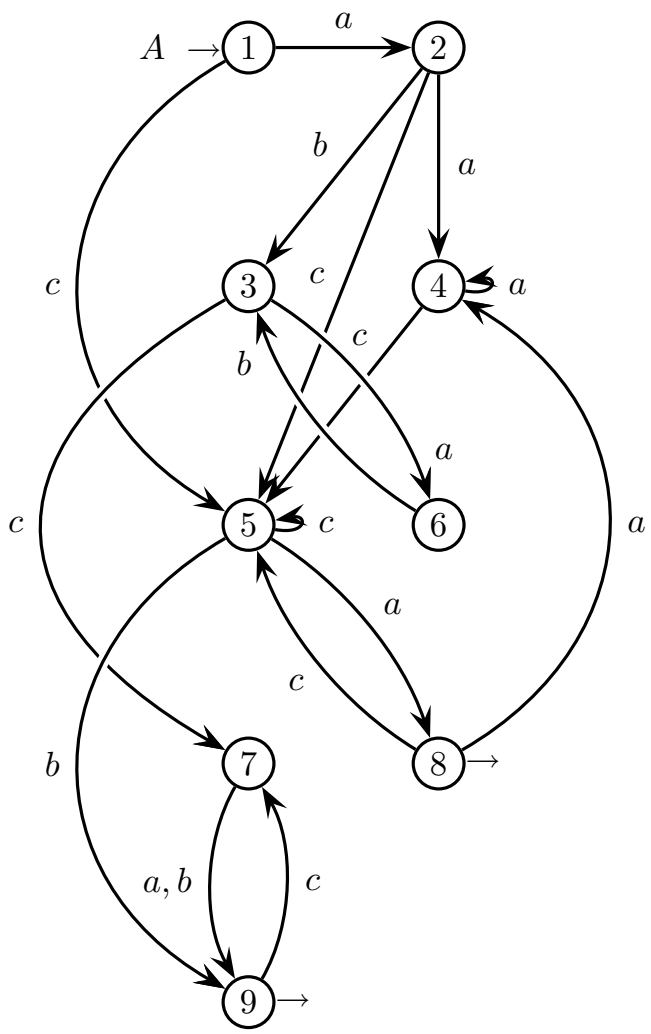
inizî	$a_1 \; a_3 \; c_4 \; c_5 \; c_7$
generatore	**séguiti**
a_1	b_2
b_2	$a_1 \; c_5 \; c_7$
a_3	$a_3 \; c_4 \; c_5 \; c_7$
c_4	$a_3 \; c_4 \; c_5 \; c_7$
c_5	b_6
b_6	$c_5 \; c_7 \; \dashv$
c_7	a_8
a_8	$c_5 \; c_7 \; \dashv$

E infine si traccia il grafo stato-transizione dell'automa deterministico A, equivalente all'espressione regolare R:



L'automa A non ha stati inutili ed è deterministico, ma potrebbe non essere in forma minima.

(b) Si rinominano gli stati dell'automa A e si ha il grafo stato-transizione seguente:



Esaminando gli stati, non è difficile convincersi che l'automa A è già in forma minima.

2 Grammatiche libere e automi a pila 20%

1. È dato un linguaggio L di alfabeto $\{a, b, c\}$, definito come segue:

$$L = \{\ a^n\, b\, c^m\, a\ \} \cup \{\ a^n\, b\, c^m\, a\, a\ \} \cup \{\ a^n\, b\, b\, c^m\, a\, a\ \}$$

Il linguaggio L è espresso come unione di tre componenti, dove si ha $0 \leq m \leq n$.

Si risponda alle domande seguenti:

(a) Si progetti una grammatica G, non ambigua e non estesa (BNF), che genera il linguaggio L.

(b) (facoltativa) Si progetti una grammatica G', non ambigua e non estesa (BNF), che genera il linguaggio stella L^* (cioè la chiusura di Kleene di L).

Soluzione

(a) Si vede subito che i tre componenti del linguaggio L sono disgiunti (dunque non danno luogo ad ambiguità di unione) e che contengono la ben nota struttura $a^n\, c^m$ $(0 \leq m \leq n)$, tipicamente context-free e facilmente generabile in modo non ambiguo. Ecco una prima versione della grammatica G, non estesa (assioma S), che genera i tre componenti di L separatamente tramite le tre forme sentenziali $A\, S_1\, a$, $A\, S_2\, a\, a$ e $A\, S_3\, a\, a$, derivabili in alternativa dall'assioma:

$$G \begin{cases} S & \rightarrow & A\ S_1\ a \ \mid\ A\ S_2\ a\ a \ \mid\ A\ S_3\ a\ a \\ S_1 & \rightarrow & a\ S_1\ c \ \mid\ b \\ S_2 & \rightarrow & a\ S_2\ c \ \mid\ b \\ S_3 & \rightarrow & a\ S_3\ c \ \mid\ b\ b \\ A & \rightarrow & a\ A \ \mid\ \varepsilon \end{cases}$$

Dato che le tre forme sentenziali generano componenti disgiunti e sono intrinsecamente non ambigue, per costruzione la grammatica G risulta non ambigua. Basterebbe questo per rispondere alla domanda, ma si nota subito che i nonterminali S_1 e S_2 sono equivalenti giacché sono espansi da regole identiche. Essi pertanto si possono compattare in uno solo, chiamato espressivamente S_{12}. Si ottiene così la versione ottimizzata seguente della grammatica G (assioma S):

$$G_{ottimizzata} \begin{cases} S & \rightarrow & A\ S_{12}\ a \ \mid\ A\ S_{12}\ a\ a \ \mid\ A\ S_3\ a\ a \\ S_{12} & \rightarrow & a\ S_{12}\ c \ \mid\ b \\ S_3 & \rightarrow & a\ S_3\ c \ \mid\ b\ b \\ A & \rightarrow & a\ A \ \mid\ \varepsilon \end{cases}$$

Le derivazioni che originano dalle tre forme sentenziali $A\, S_{12}\, a$, $A\, S_{12}\, a\, a$ e $A\, S_3\, a\, a$ generano componenti disgiunti e sono non ambigue (per gli stessi motivi di prima), pertanto anche la grammatica G in versione ottimizzata non lo è.

(b) Come primo tentativo si potrebbe ottenere una grammatica G' che genera il linguaggio stella L^* tramite la costruzione canonica della chiusura di Kleene, aggiungendo alla grammatica G data prima la regola assiomatica $S_0 \to S\ S_0 \mid \varepsilon$ (il nuovo assioma è il nonterminale S_0). Conviene naturalmente fare riferimento alla versione ottimizzata di G. Tuttavia la grammatica G' così ottenuta presenterebbe ambiguità di concatenamento. Infatti si ha quanto segue:

$$L^* \ni a\,b\,c\,a\,a\,a\,b\,c\,a = \begin{cases} \overbrace{\underbrace{\varepsilon}_{A}\ \underbrace{a\,b\,c\ a}_{S_{12}}}^{S} \cdot \overbrace{\underbrace{a}_{A}\ \underbrace{a\,b\,c\ a}_{S_{12}}}^{S} \\ \overbrace{\underbrace{\varepsilon}_{A}\ \underbrace{a\,b\,c\ a\,a}_{S_{12}}}^{S} \cdot \overbrace{\underbrace{\varepsilon}_{A}\ \underbrace{a\,b\,c\ a}_{S_{12}}}^{S} \end{cases}$$

Insomma ci sono due derivazioni diverse di tipo $S_0 \overset{*}{\Rightarrow} S\,S \overset{*}{\Rightarrow} a\,b\,c\,a\,a\,a\,b\,c\,a \in L^*$. Si nota subito che l'ambiguità compare concatenando due stringhe del linguaggio L tali che la prima contiene una sola lettera b e ci sono almeno tre lettere a in mezzo (nell'esempio anche la seconda stringa contiene una sola b). Il motivo è che una stringa di L contenente una sola lettera b può avere una o due lettere a in fondo (mentre se ci sono due lettere b è d'obbligo avere due lettere a in fondo), e nel secondo caso l'ultima lettera a è attribuibile anche alla stringa di L seguente. Tutto ciò continua a valere concatenando tre o più stringhe.

In termini formali, qui succede che la condizione necessaria e sufficiente per avere ambiguità di concatenamento è verificata. Infatti si ha quanto segue:

$$L^* \ni x = \begin{cases} \underbrace{a\,b\,c\,a\,a}_{u} \cdot \underbrace{a\,b\,c\,a}_{v} = u\,v & u, v \in L \\ \underbrace{a\,b\,c\,a}_{u'} \cdot \underbrace{a\,a\,b\,c\,a}_{v'} = u'\,v' & u', v' \in L \end{cases}$$

dove il fattore u' è prefisso del fattore u, avendosi $u = u'\,a$; e similmente il fattore v è suffisso del fattore v', avendosi $v' = a\,v$ (si veda anche il libro di testo).

Per evitare l'ambiguità si può imporre che se la (prima) stringa del linguaggio L contiene una sola lettera b, debba per esempio avere necessariamente una sola lettera a in fondo, a meno che la stringa non sia l'ultima concatenata. Ecco una grammatica G' progettata secondo questa linea di ragionamento (assioma S_0):

$$G' \begin{cases} S_0 \to A\ S_{12}\ a\ S_0 \mid A\ S_3\ a\ a\ S_0 \mid \\ \qquad\quad A\ S_{12}\ a \mid A\ S_{12}\ a\ a \mid A\ S_3\ a\ a \\ S_{12} \to a\ S_{12}\ c \mid b \\ S_3 \to a\ S_3\ c \mid b\ b \\ A \to a\ A \mid \varepsilon \end{cases}$$

Tramite le regole assiomatiche, che sono lineari a destra nell'assioma S_0, la grammatica G' genera successioni libere di fattori $A\,S_{12}\,a$ e $A\,S_3\,a\,a$; il fattore $A\,S_{12}\,a\,a$, se pure compare, può figurare solo come ultimo della successione.

Detto altrimenti, per costruzione la grammatica G' deriva (facoltativamente) il fattore $A\,S_{12}\,a\,a$ (con due a in fondo) solo alla fine della derivazione, mentre prima può comparire solo il fattore $A\,S_{12}\,a$ (con una sola a in fondo), oltre al fattore $A\,S_3\,a\,a$ che ha d'obbligo due lettere a in fondo e che dunque non dà problema di ambiguità. Pertanto questa versione della grammatica G' genera il linguaggio stella L^* ma non è ambigua, come richiesto.

La costruzione così proposta è un raffinamento di quella canonica per la chiusura di Kleene, che non evita l'ambiguità di concatenamento se le condizioni si verificano (come in effetti succede qui). Si sarebbe potuta dare anche sulla versione non ottimizzata delle grammatica G, con qualche lungaggine in più.

2. Si consideri il sottinsieme di linguaggio di programmazione di alto livello (come C, Pascal, ecc) comprendente gli aspetti sintattici seguenti:

   - ci sono variabili nominali, di tipo intero, e costanti numeriche intere
   - la variabile e la costante sono denotate semplicemente dai simboli "var" e "const", rispettivamente, che non occorre espandere ulteriormente
   - ci sono espressioni con variabili, costanti, addizione "+" e sottrazione "−"
   - l'istruzione (statement) del programma è di due tipi:
     - istruzione di assegnamento, con la sintassi seguente:

       ```
       nome-di-variabile = espressione
       ```

     - struttura di controllo condizionale di tipo `if-then` o `if-then-else`, come:

       ```
       if ( espressione ) then
          blocco-di-istruzioni
       ```

       ```
       if ( espressione ) then
          blocco-di-istruzioni
       else
          blocco-di-istruzioni
       ```

   - un blocco di istruzioni è una lista, non vuota e racchiusa tra parentesi graffe "{" e "}", di istruzioni separate dal carattere ";" (punto e virgola); il separatore non va messo dopo l'ultima istruzione della lista; per esempio:

     ```
     { istruzione; istruzione; istruzione }
     ```

   - se il blocco consta di una sola istruzione le parentesi graffe sono facoltative
   - in caso di else pendente (dangling else), come in questi due esempi:

     ```
     if ( espressione ) then
        if ( espressione ) then
           blocco-di-istruzioni
     else
        blocco-di-istruzioni
     ```

     ```
     if ( espressione ) then {
        if ( espressione ) then
           if ( espressione ) then
              blocco-di-istruzioni
        else
           blocco-di-istruzioni
     }
     ```

     il ramo else pendente va sintatticamente attribuito alla struttura di controllo if più esterna, compatibilmente con la suddivisione in blocchi di istruzioni, come l'allineamento nei due esempi dati sopra suggerisce - QUESTA SPECIFICA DIFFERISCE DAL LINGUAGGIO C STANDARD

   - infine, il programma consiste di un blocco di istruzioni (non c'è sezione di dichiarazione delle variabili)

   Si chiede di scrivere la grammatica, non ambigua e in forma estesa (EBNF), che genera il minilinguaggio di programmazione così descritto.

Soluzione

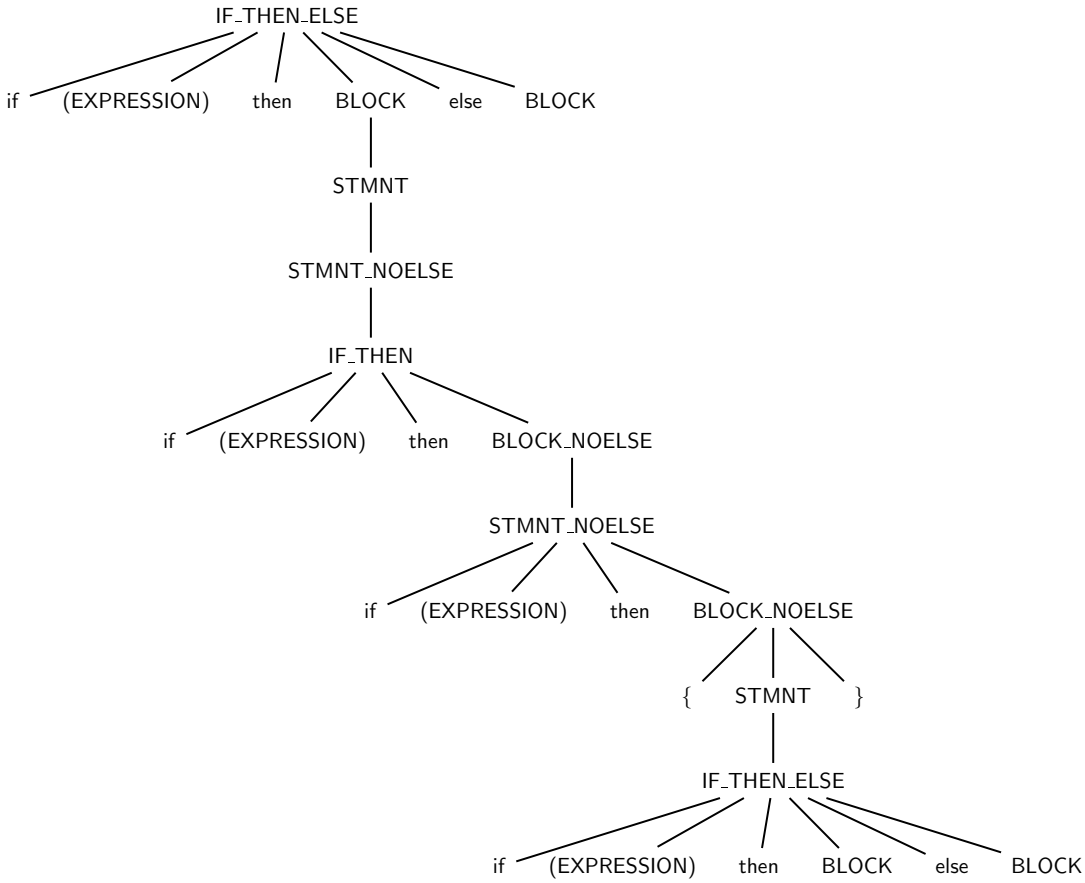
Ecco una soluzione EBNF non ambigua (assioma ⟨PROGRAM⟩):

⟨PROGRAM⟩	→	⟨BLOCK⟩
⟨BLOCK⟩	→	‘{’ (⟨STMNT⟩ ‘;’)* ⟨STMNT⟩ ‘}’ | ⟨STMNT⟩
⟨BLOCK_NOELSE⟩	→	‘{’ (⟨STMNT⟩ ‘;’)* ⟨STMNT⟩ ‘}’ | ⟨STMNT_NOELSE⟩
⟨STMNT⟩	→	⟨STMNT_NOELSE⟩ | ⟨IF_THEN_ELSE⟩
⟨STMNT_NOELSE⟩	→	⟨ASSIGNMENT⟩ | ⟨IF_THEN⟩
⟨IF_THEN_ELSE⟩	→	‘if’ ‘(’ ⟨EXPRESSION⟩ ‘)’ ‘then’ ⟨BLOCK⟩ ‘else’ ⟨BLOCK⟩
⟨IF_THEN⟩	→	‘if’ ‘(’ ⟨EXPRESSION⟩ ‘)’ ‘then’ ⟨BLOCK_NOELSE⟩
⟨ASSIGNMENT⟩	→	⟨VARIABLE⟩ ‘=’ ⟨EXPRESSION⟩
⟨EXPRESSION⟩	→	([‘+’ | ‘−’] ⟨TERM⟩)* | ⟨TERM⟩
⟨TERM⟩	→	⟨CONSTANT⟩ | ⟨VARIABLE⟩
⟨VARIABLE⟩	→	‘var’
⟨CONSTANT⟩	→	‘const’

Qui le parentesi quadre “[” e “]” denotano opzionalità. La grammatica è strutturata modularmente, secondo schemi (di tipo esteso) non ambigui, e dunque non è ambigua.

Per definizione si ha else pendente (dangling else) quando c’è una catena di strutture if innestate nei rami then e in fondo figura anche un ramo else, ambiguamente attribuibile a una qualunque delle strutture if. La maggior parte dei linguaggi di programmazione (linguaggio C e molti altri) risolve l’ambiguità associando sintatticamente lo else pendente allo if più interno (vedi per esempio il libro di testo). Qui si chiede di procedere diversamente, associandolo allo if più esterno. Con ciò si intende di associarlo allo if più in alto nella catena di strutture if innestate nei rami then, posto che non si debbano scavalcare parentesi graffe (che deliminato i blocchi) o un altro ramo else già piazzato.

La grammatica proposta risolve il problema in modo corretto. Lo else pendente è sempre attribuito alla struttura di controllo if più esterna, compatibilmente con la presenza di parentesi graffe. Infatti il ramo then di una struttura if senza ramo else è espanso dalla classe sintattica BLOCK_NOELSE, la quale o genera una lista di istruzioni racchiuse tra parentesi graffe e non può dare luogo a un else pendente, o si riduce alla classe sintattica STMNT_NOELSE e genera solo un’altra struttura if senza ramo else oppure un assegnamento. Se si immagina l’albero sintattico, è chiaro che così un nodo else pendente o è figlio del nodo if più vicino alla radice o a una coppia di parentesi graffe, oppure è figlio di un nodo if già dotato di ramo else (ossia in generale è figlio del nodo if più esterno possibile). Si veda per esempio l’albero sintattico seguente (frammentario ma serve semplicemente per capire):



L'albero genera il frammento seguente di programma, dove ci sono due rami else pendenti attribuiti in modo corretto alla struttura if di riferimento.

```
if ... then                 /* struttura if-then-else */
   if ... then              /* struttura if-then      */
      if ... then {         /* struttura if-then      */
         if ... then ...    /* struttura if-then-else */
         else ...           /* else pendente          */
      }
else ...                    /* else pendente          */
```

L'allineamento mostra come funziona l'attribuzione: lo else a profondità maggiore è attribuito allo if più esterno ma senza uscire dalla coppia di parentesi graffe, lo else a profondità minore è attribuito alla radice. Naturalmente ci possono essere soluzioni diverse, altrettanto valide di quella sviluppata qui.

3 Analisi sintattica e parsificatori 20%

1. La grammatica G seguente (in forma estesa EBNF) genera liste i cui elementi, separati dal carattere d, sono stringhe ben parentesizzate di alfabeto $\{\, a,\, c \,\}$ (assioma S).

$$G \begin{cases} S & \rightarrow & P\ (\ d\ P\ )^* \\ P & \rightarrow & a\ P\ c\ \mid\ P\ a\ c\ \mid\ \varepsilon \end{cases}$$

Si risponda alle domande seguenti:

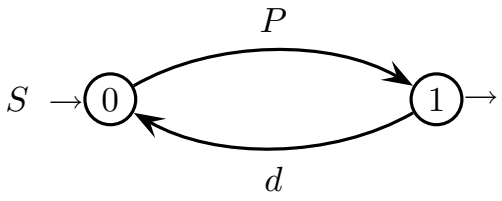
(a) Si rappresenti la grammatica G come una rete di macchine a stati finiti, disegnando i grafi corrispondenti.

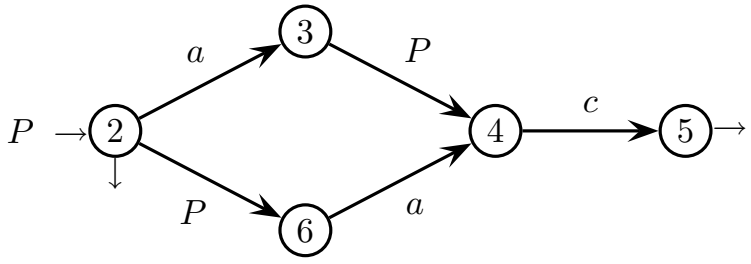
(b) Si calcolino gli insiemi guida $LL(1)$ per ogni arco e freccia della rete, e si indichino gli eventuali conflitti presenti.

(c) (facoltativa) Se è necessario si modifichi la grammatica G al fine di ottenerne una equivalente che soddisfi la condizione $LL(1)$.

Soluzione

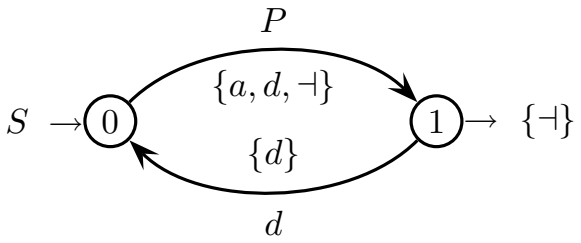
(a) Ecco le due macchine della rete ricorsiva di automi finiti rappresentante la grammatica G, dove le due macchine espandono i nonterminali S e P:

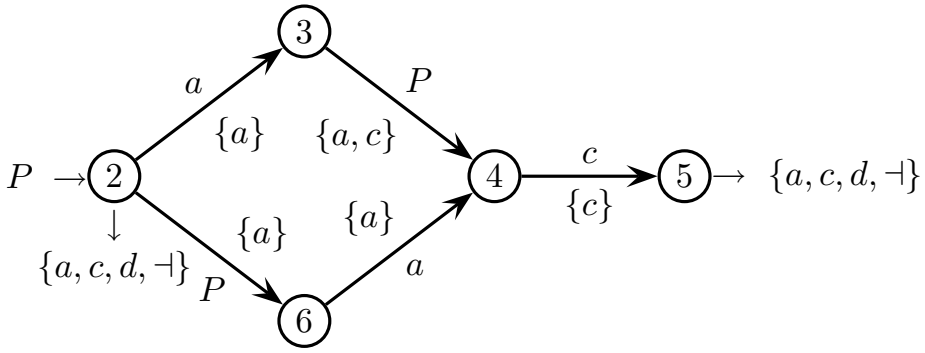




Si vede facilmente che entrambe le macchine sono automi deterministici, sull'alfabeto totale (unione di nonterminali e terminali), e sono anche minimi.

(b) Ed ecco tutti gli insiemi guida richiesti, con profondità di prospezione $k = 1$:





Si noti che il nonterminale P è nullabile e che di conseguenza lo è anche l'assioma S. Ne va tenuto conto nel calcolare gli insiemi guida.

(c) La sovrapposizione tra gli insiemi guida delle tre frecce uscenti dallo stato 2 (la lettera a è comune) rende la grammatica G non di tipo $LL(1)$; qui non ci sono altre violazioni della proprietà $LL(1)$. Comunque G è ambigua (per esempio lo è la stringa $a\,c$) e pertanto non è $LL(k)$ per nessun $k \geq 1$.

Per trovare una grammatica tipo $LL(1)$ si può fare il ragionamento seguente. Una breve riflessione mostra che il linguaggio generato del nonterminale P (preso come assioma) è un sottinsieme del linguaggio di Dyck su alfabeto $\{a, c\}$ (aperta e chiusa). La restrizione consiste in questo: dove ci sono più coppie concatenate di parentesi associate (cioè coppie $a\,c$) solo la prima coppia può eventualmente avere profondità di annidamento maggiore di uno, cioè può contenere altre coppie ricorsivamente. Ecco un esempio (a sinistra) e un controesempio (a destra):

$$\overbrace{a\ \underbrace{a\ c}_{1.1}\ c}^{1}\ \overbrace{a\ c}^{2}\ \overbrace{a\ c}^{3} \in L(P) \qquad \overbrace{a\ c}^{1}\ \overbrace{a\ \underbrace{a\ c}_{2.1}\ c}^{2}\ \overbrace{a\ c}^{3} \notin L(P)$$

Nell'esempio (sinistra) ci sono tre coppie concatenate: la prima ha profondità due, la seconda e la terza hanno profondità uno. Nel controesempio (destra) ci sono ancora tre coppie, ma la seconda ha profondità due (e ciò non va bene).

Per avere una grammatica di tipo $LL(1)$ (dunque non ambigua), si può allora ricorrere alla regola di Dyck standard (che è di tipo $LL(1)$) applicandole la

restrizione opportuna per togliere le stringhe $\notin L(P)$. Ecco una grammatica G' equivalente a G, non ambigua e di tipo $LL(1)$:

$$G' \begin{cases} S & \rightarrow & P\ (\ d\ P\ )^* \\ P & \rightarrow & a\ P\ c\ Q\ \mid\ \varepsilon \\ Q & \rightarrow & a\ c\ Q\ \mid\ \varepsilon \end{cases}$$

Per giustificare l'equivalenza tra le grammatiche G e G', si osservi che in sostanza la regola $P \rightarrow a\ P\ c\ Q\ \mid\ \varepsilon$ è quella standard di Dyck eccetto che la seconda comparsa del nonterminale P a destra è sostituita dal nuovo nonterminale Q, il quale tramite la regola lineare a destra $Q \rightarrow a\ c\ Q\ \mid\ \varepsilon$ si limita eventualmente a concatenare (a destra) coppie di parentesi tutte di profondità uno.

Nella grammatica G' la condizione $LL(1)$ risulta facilmente verificata: basta osservare che gli insiemi guida delle regole alternative $P \rightarrow a\ P\ c\ Q$ e $P \rightarrow \varepsilon$ sono $\{a\}$ e $\{c, d, \dashv\}$, rispettivamente, e che ovviamente sono disgiunti; inoltre la nuova regola lineare di Q si comporta come quella di P e non dà alcun problema.

Un modo elegante di vedere le due regole della grammatica G' che espandono i nonterminali P e Q, è di considerarle entrambe restrizioni della regola standard di Dyck, genericamente esprimibile come $X \rightarrow a\ X\ c\ X\ \mid\ \varepsilon$, così:

$$\begin{array}{lcl} P & \rightarrow & a\ P\ c\ P\ \mid\ \varepsilon \\ Q & \rightarrow & a\ Q\ c\ Q\ \mid\ \varepsilon \end{array} \quad \underset{\text{restrizione}}{\Rightarrow} \quad \begin{array}{lcl} P & \rightarrow & a\ P\ c\ Q\ \mid\ \varepsilon \\ Q & \rightarrow & a\quad c\ Q\ \mid\ \varepsilon \end{array}$$

La regola di P è ristretta in modo da potere generare direttamente coppie annidate ricorsivamente, ma da dovere passare alla regola di Q per concatenarle. La regola di Q, dalla quale non si può più tornare a quella di P, è ristretta in modo da potere concatenare coppie, ma non da poterle annidare. Questo è proprio il comportamento desiderato. La soluzione proposta è pertanto semplicemente un rimaneggiamento oculato della grammatica standard di Dyck, restringendo le derivazioni possibili. Per tale motivo risulta subito non ambigua e di tipo $LL(1)$. Va da sé che ci possono essere altre soluzioni, pure compatte ed eleganti.

2. Si deve costruire un parsificatore deterministico per il linguaggio L generato dalla grammatica G seguente, di alfabeto $\{a, b\}$ (assioma S):

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

Si risponda alle domande seguenti:

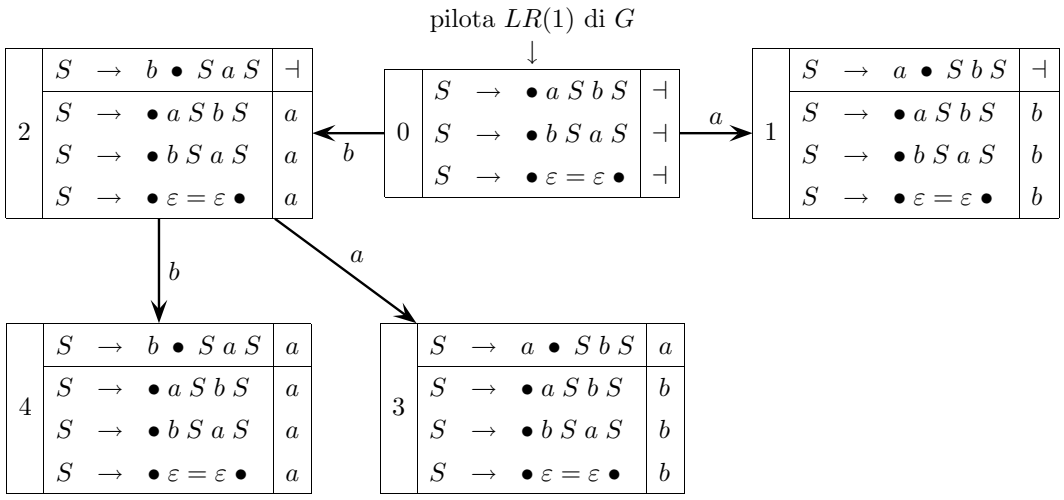
(a) Si costruisca il grafo del pilota $LR(1)$ della grammatica G, anche solo parzialmente per quanto serve per la domanda successiva.

(b) Si metta in evidenza un conflitto presente nell'automa pilota $LR(1)$ della grammatica G, spiegando di che tipo sia.

(c) (facoltativa) Si descriva, a parole o mediante il grafo stato-transizione, un automa a pila deterministico A che riconosce il linguaggio L.

Soluzione

(a) Ecco la parte del grafo pilota $LR(1)$ di G sufficiente per rispondere alle domande:



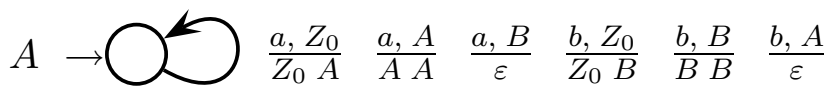
Con un po' di pazienza si completa il grafo pilota. Del resto il grafo completo non aggiungerebbe grandi novità a quanto già si vede sul frammento disegnato.

(b) Nel grafo pilota $LR(1)$ della grammatica G, il macrostato 2 ha un conflitto tra spostamento (lettura di a) e riduzione con la regola nulla $S \to \varepsilon$ (con prospezione a), e il macrostato 1 ha lo stesso problema (qui non mostrato). Si noti che le regole di G sono simmetriche rispetto alle lettere a e b, e di conseguenza anche i macrostati godono della stessa proprietà di simmetria.

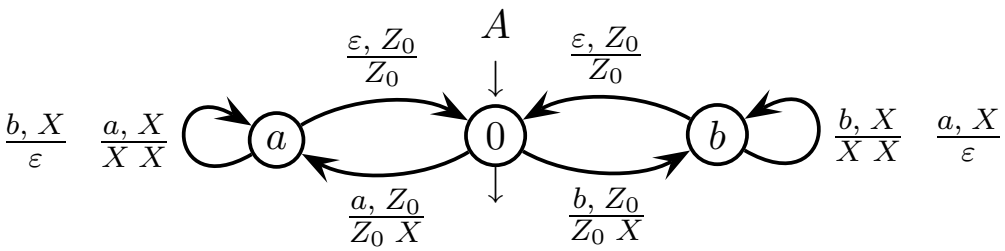
(c) Per progettare un automa a pila deterministico capace di riconoscere il linguaggio L, conviene fare le osservazioni seguenti. Le regole della grammatica G sono quelle standard del linguaggio di Dyck (su alfabeto binario), dove però le parentesi associate a coppie possono comparire sia nell'ordine aperta-chiusa $(a \dots b)$ sia in quello chiusa-aperta $(b \dots a)$. Da ciò segue subito che ogni frase di L contiene un numero uguale di lettere a e b. Ma segue anche il viceversa, cioè che ogni frase con uguale numero di lettere a e b appartiene a L: infatti se si sa che ci sono tante a quante b e se si concede che in una coppia di parentesi associate la chiusa b possa precedere l'aperta a corrispondente (oppure seguirla), sarà sempre possibile associare a e b al modo di Dyck. Così si conclude che L è costituito proprio da tutte e sole le stringhe di alfabeto $\{a, b\}$ con uguale numero di a e b.

Ora è facile modellare l'automa a pila riconoscitore del linguaggio L: esso si limita a scandire la stringa (da sinistra verso destra) tenendo traccia dell'eccesso di lettere a su b o viceversa (scandendo la stringa la situazione si può invertire più volte), e accetta alla fine solo se non c'è eccesso (né di a su b né di b su a). La pila dell'automa andrà pertanto usata come contatore delle lettere in eccesso. Ciò detto, sono possibili varie soluzioni specifiche, più o meno naturali.

Per esempio, dopo la lettura della stringa $a\,a$ la pila contiene i simboli $A\,A$ (eccesso di due a), dopo la lettura di $b\,a$ è vuota (numero uguale di a e b) e dopo la lettura di $b\,b\,a$ contiene B (eccesso di una b su a). L'automa parte con la pila vuota, ossia con il simbolo iniziale Z_0, e riconosce la stringa a pila vuota. Ecco questa versione dell'automa a pila A:



L'automa A è deterministico, ha un alfabeto di memoria di due simboli (A e B), riconosce a pila vuota e ha un solo stato. La distinzione tra eccesso di a su b o viceversa è espressa dai simboli di memoria. Tuttavia ciò non è essenziale. Eccone infatti una versione multistato di A, ma con alfabeto di memoria unario:



Questa versione di A (pure deterministica) ha tre stati ma un solo simbolo di memoria (X). È facile vedere che qui la distinzione tra eccesso di a su b o viceversa è espressa dagli stati, mentre la pila è un contatore puro. L'automa riconosce indifferentemente a pila vuota o a stato finale: infatti la pila è vuota se e solo se si è nello stato finale 0. Naturalmente ci possono essere altre soluzioni.

4 Traduzione e analisi semantica 20%

1. L'ingresso della traduzione è una sequenza di dati appartenenti ai tipi intero (int) i e carattere (char) c, che hanno rispettivamente le dimensioni 2 e 1 (in byte). La funzione di traduzione trasforma la sequenza di ingresso in una sequenza di record, dove il singolo record ha dimensione di 4 byte. Ogni record deve contenere il numero massimo possibile di dati e, se necessario, terminare con un riempitivo (padding), denotato con un solo simbolo pad (esempio sotto). L'inizio e la fine di ogni record sono segnati da beg ed end. Ecco un esempio (con terminatore $\dashv$ nella sorgente):

   Stringa sorgente:

   $$c\ i\ i\ i\ c\ c\ i\ c\ \dashv$$

   Stringa immagine:

   $$\mathsf{beg}\ c\ i\ \mathsf{pad}\ \mathsf{end}\ \mathsf{beg}\ i\ i\ \mathsf{end}\ \mathsf{beg}\ c\ c\ i\ \mathsf{end}\ \mathsf{beg}\ c\ \mathsf{pad}\ \mathsf{end}$$

   Se l'ingresso è la stringa vuota (con terminatore), l'uscita emessa è beg pad end.

   Si risponda alle domande seguenti:

   (a) Si specifichi il traduttore scegliendo una tecnica opportuna tra le seguenti: IO-automa, schema di traduzione puramente sintattico o espressione regolare di traduzione.

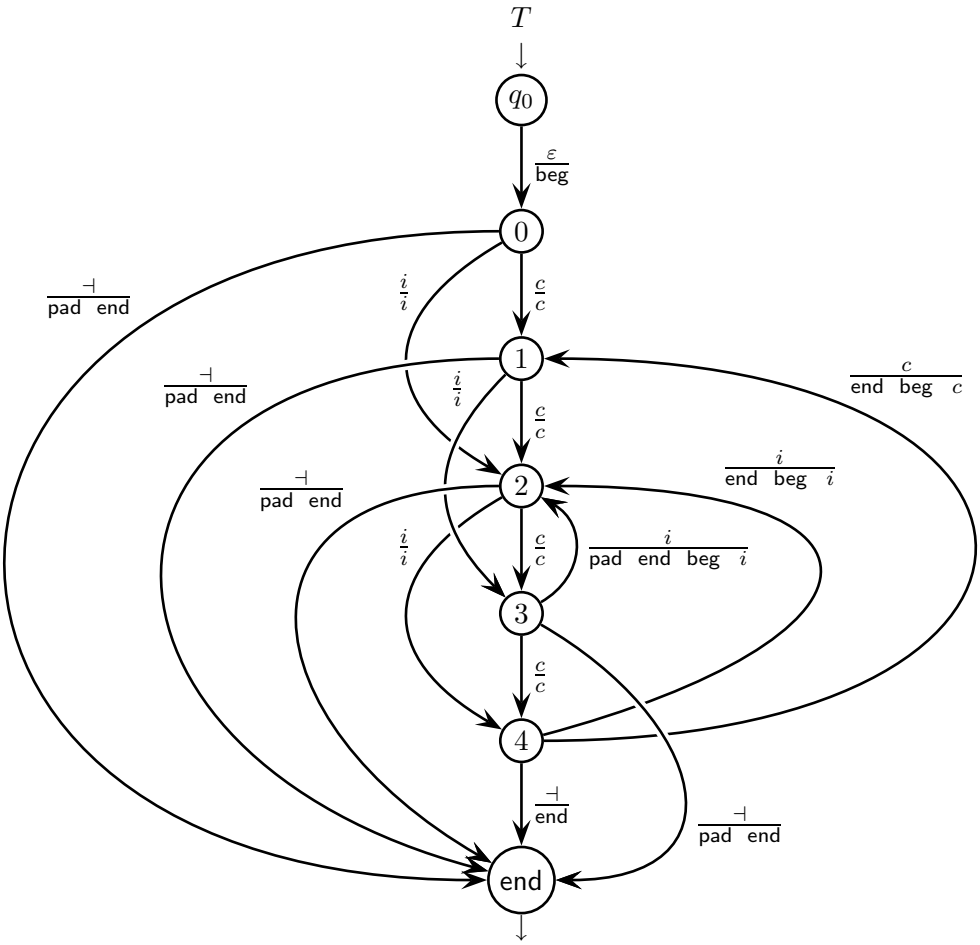
   (b) (facoltativa) Se già non lo è, si spieghi o si mostri come ottenere dalla specifica precedente un algoritmo deterministico che calcoli la traduzione.

Soluzione

(a) La traduzione descritta informalmente è piuttosto semplice e fondalmentalmente deve calcolare modulo 4 la dimensione dei record da emettere. Trattandosi dunque di un contatore modulare, uno IO-automa a stati finiti, capace di codificare nei suoi stati i quattro valori di conteggio 0, 1, 2 e 3, sembra un modello di traduttore con capacità espressiva adeguata alla natura del problema (ma anche gli altri modelli proposti sono ragionevoli per lo scopo).

Lo IO-automa deve registrare nel proprio stato la dimensione in byte del record in via di composizione, compresa tra 0 e 4. All'inizio la dimensione vale 0 e viene incrementata di 1 o 2 leggendo c o i, rispettivamente. Se però l'incremento porta a superare la dimensione 4, l'automa emette subito il simbolo pad e chiude il record corrente (con end), poi ne inizia uno nuovo (con beg) dove inserisce il simbolo di ingresso che ha superato la dimensione del record precedente; e continua in questo modo fino a esaurimento della stringa sorgente. Se al termine l'ultimo record emesso non risulta del tutto pieno, va completato con il riempitivo pad. L'esempio dato sopra è sufficientemente illuminante. Se la sorgente è vuota vale la specifica data nel testo, di emettere un record contenente solo il riempitivo.

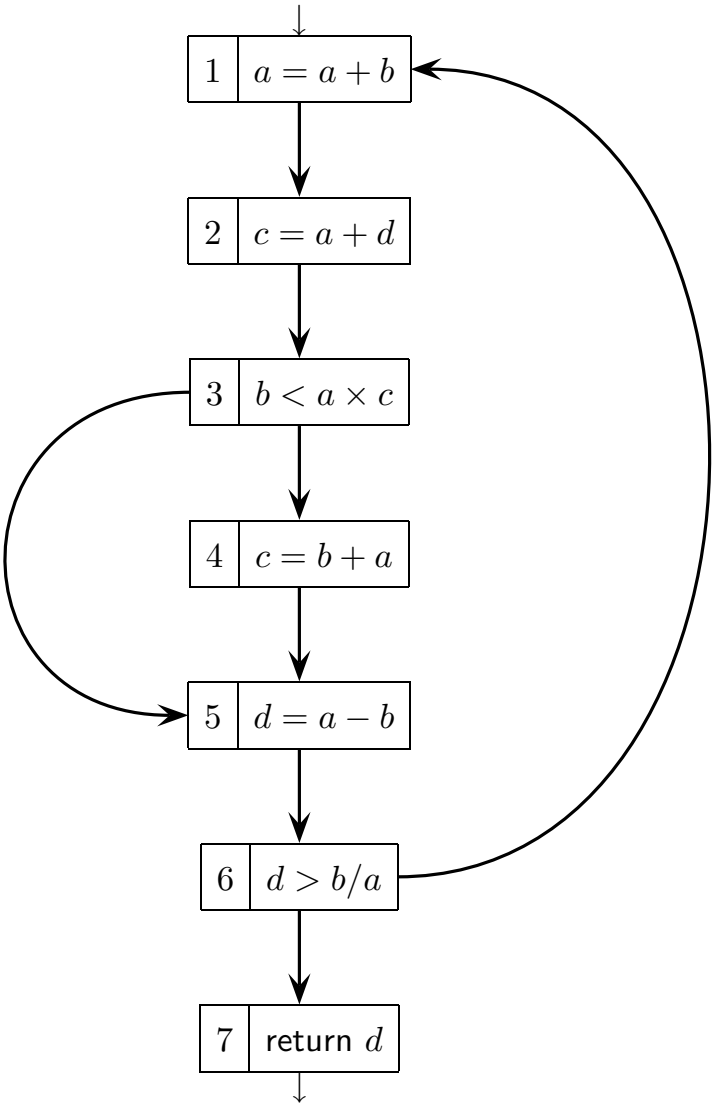
Ecco il grafo stato-transizione (con sette stati) dello IO-automa T che realizza la traduzione così descritta:



È facile riscrivere lo IO-automa T come schema sintattico di traduzione, di tipo lineare a destra, oppure come espressione regolare di traduzione.

(b) Il traduttore T così disegnato non è deterministico solo perché ha un automa riconoscitore soggiacente con una mossa spontanea dallo stato q_0 allo stato 0. Ma è semplice ottenerne uno deterministico spostando il simbolo beg sugli archi uscenti dallo stato 0, eliminando lo stato q_0 e rendendo iniziale lo stato 0 stesso (si lascia l'esercizio al lettore).

2. Si consideri il seguente grafo di controllo di una routine (con sette istruzioni):



Si risponda alle domande seguenti:

(a) Si scrivano le equazioni di flusso per il calcolo degli intervalli di vita delle variabili (scrivendo nella tabella e nella griglia predisposte all'uopo).

(b) Usando tali equazioni si calcolino gli insiemi di variabili vive in ogni punto del programma (scrivendo nella griglia predisposta all'uopo).

(c) Ancora usando gli insiemi di variabili vive, si indichi se esistano nel programma istruzioni rimovibili in quanto corrispondenti a definizioni inutili.

(d) (facoltativa) Si indichi se e come gli insiemi di variabili vive siano utilizzabili per minimizzare la memoria occupata dalle variabili.

#	**def**	**use**
1		
2		
3		
4		
5		
6		
7		

$in(1)$	=	$use(1) \cup (out(1) - def(1)) =$	– definizione di vitalità in ingresso
$out(1)$	=		– il nodo 1 ha ...
$in(2)$	=		– definizione di vitalità in ingresso
$out(2)$	=		– il nodo 2 ha ...
$in(3)$	=		– definizione di vitalità in ingresso
$out(3)$	=		– il nodo 3 ha ...
$in(4)$	=		– definizione di vitalità in ingresso
$out(4)$	=		– il nodo 4 ha ...
$in(5)$	=		– definizione di vitalità in ingresso
$out(5)$	=		– il nodo 5 ha ...
$in(6)$	=		– definizione di vitalità in ingresso
$out(6)$	=		– il nodo 6 ha ...
$in(7)$	=		– definizione di vitalità in ingresso
$out(7)$	=		

	stato 0		**stato** 1		**stato** 2		**stato** 3	
#	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*
1								
2								
3								
4								
5								
6								
7								

	stato 3		**stato** 4		**stato** 5		**stato** 6	
#	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*
1								
2								
3								
4								
5								
6								
7								

Soluzione

L'esercizio non presenta difficoltà di sorta e la tecnica di soluzione è del tutto standard, suddivisa in fasi.

(a) Calcolo dei termini costanti (definizione e uso di variabile):

#	**def**	**use**
1	a	a, b
2	c	a, d
3	-	a, b, c
4	c	a, b
5	d	a, b
6	-	a, b, d
7	-	d

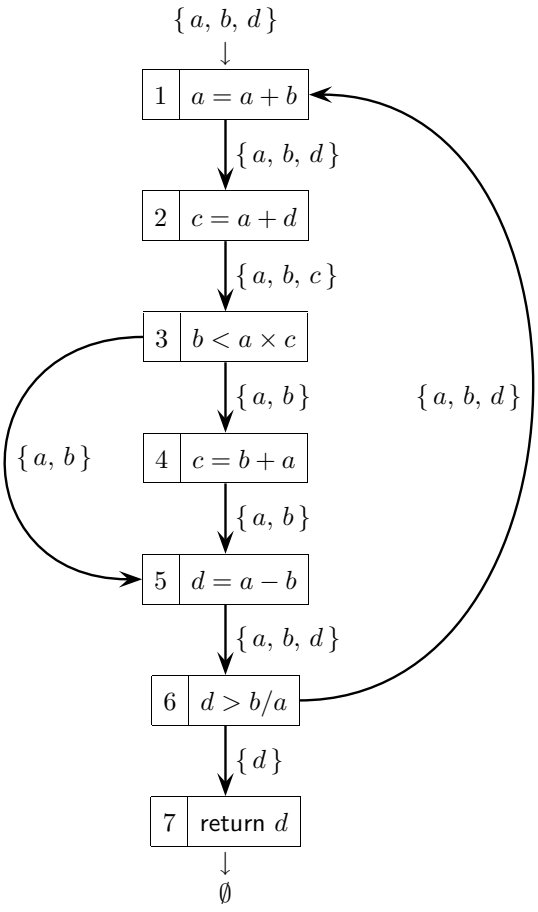
Scrittura delle equazioni di flusso per le variabili vive ai nodi:

$$
\begin{array}{rcll}
in(1) & = & use(1) \cup (out(1) - def(1)) = & \text{– definizione di vitalità in ingresso} \\
 & = & \{a, b\} \cup (out(1) - \{a\}) = & \\
out(1) & = & in(2) & \text{– il nodo 1 ha una sola via d'uscita} \\
\hline
in(2) & = & use(2) \cup (out(2) - def(2)) = & \text{– definizione di vitalità in ingresso} \\
 & = & \{a, d\} \cup (out(2) - \{c\}) = & \\
out(2) & = & in(3) & \text{– il nodo 2 ha una sola via d'uscita} \\
\hline
in(3) & = & use(3) \cup (out(3) - def(3)) = & \text{– definizione di vitalità in ingresso} \\
 & = & \{a, b, c\} \cup (out(3) - \emptyset) = & \\
 & = & \{a, b, c\} \cup out(3) & \\
out(3) & = & in(4) \cup in(5) & \text{– il nodo 3 ha due vie d'uscita} \\
\hline
in(4) & = & use(4) \cup (out(4) - def(4)) = & \text{– definizione di vitalità in ingresso} \\
 & = & \{a, b\} \cup (out(4) - \{c\}) = & \\
out(4) & = & in(5) & \text{– il nodo 4 ha una sola via d'uscita} \\
\hline
in(5) & = & use(5) \cup (out(5) - def(5)) = & \text{– definizione di vitalità in ingresso} \\
 & = & \{a, b\} \cup (out(5) - \{d\}) = & \\
out(5) & = & in(6) & \text{– il nodo 5 ha una sola via d'uscita} \\
\hline
in(6) & = & use(6) \cup (out(6) - def(6)) = & \text{– definizione di vitalità in ingresso} \\
 & = & \{a, b, d\} \cup (out(6) - \emptyset) = & \\
 & = & \{a, b, d\} \cup out(6) & \\
out(6) & = & in(1) \cup in(7) & \text{– il nodo 6 ha due vie d'uscita} \\
\hline
in(7) & = & use(7) \cup (out(7) - def(7)) = & \text{– definizione di vitalità in ingresso} \\
 & = & \{d\} \cup (out(7) - \emptyset) = & \\
 & = & \{d\} \cup out(7) & \\
out(7) & = & \emptyset &
\end{array}
$$

(b) Calcolo iterativo della soluzione alle equazioni di flusso:

	passo 1		**passo** 2		$2 = 3$			
	stato 0		**stato** 1		**stato** 2		**stato** 3	
#	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*
1	$\emptyset$	$\emptyset$	a, b	a, d	a, b, d	a, b, d	a, b, d	a, b, d
2	$\emptyset$	$\emptyset$	a, d	a, b, c	a, b, d	a, b, c	a, b, d	a, b, c
3	$\emptyset$	$\emptyset$	a, b, c	a, b	a, b, c	a, b	a, b, c	a, b
4	$\emptyset$	$\emptyset$	a, b	a, b	a, b	a, b	a, b	a, b
5	$\emptyset$	$\emptyset$	a, b	a, b, d	a, b	a, b, d	a, b	a, b, d
6	$\emptyset$	$\emptyset$	a, b, d	a, b, d	a, b, d	a, b, d	a, b, d	a, b, d
7	$\emptyset$	$\emptyset$	d	$\emptyset$	d	$\emptyset$	d	$\emptyset$

Al passo numero 2 si raggiunge la convergenza, giacché lo stato numero 3 non differisce dallo stato numero 2. In totale dunque occorrono due passi per calcolare la soluzione. Ecco il grafo di controllo decorato con gli insiemi di variabili vive:



(c) L'istruzione numero 4 dà una definizione inutile della variabile c (che non appartiene all'insieme della variabili vive alla sua uscita) e pertanto è rimovibile.

(d) Le variabili c e d non interferiscono (non figurano mai vive insieme) e pertanto sono memorizzabili nella stessa cella, così riducendo il consumo di memoria.