Linguaggi Formali e Compilatori
(Formal Languages and Compilers)

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

Prova scritta - 7 marzo 2007 - 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 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 un anno.
- Per superare la parte I (teoria) occorre dimostrare di possedere conoscenza sufficiente 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. Le due tabelle seguenti A e B constano di due righe ciascuna e ogni riga contiene due espressioni regolari: la prima espressione è nella colonna L_1, la seconda nella L_2.

   Si svolgano i punti seguenti:

   (a) A ogni riga si elenchino per lunghezza crescente le prime tre stringhe appartenenti al linguaggio definito dalla differenza insiemistica $L_1 \setminus L_2$.

   Tabella A

L_1	L_2	le tre stringhe più corte di $L_1 \setminus L_2$
$a((b \mid bb)a)^+$	$(ab)^*ba$	
$(a(ab)^*b)^*(\varepsilon \mid a(ab)^* \mid ab)$	$(a(ab)^*b)^*$	

   (b) A ogni riga si scriva un'espressione regolare che generi il linguaggio $L_1 \setminus L_2$, usando soltanto gli operatori di unione, concatenamento, stella e croce.

   Tabella B

L_1	L_2	un'espressione regolare di $L_1 \setminus L_2$
$a((b \mid bb)a)^+$	$(ab)^*ba$	
$(a(ab)^*b)^*(\varepsilon \mid a(ab)^* \mid ab)$	$(a(ab)^*b)^*$	

   Si scriva la risposta alle domande nella terza colonna di ogni riga nelle tabelle A e B.

Soluzione

Per comodità si danno insieme le risposte a entrambi i punti (a) e (b). Nella prima riga i due linguaggi L_1 e L_2 si caratterizzano nel modo seguente:

- l'insieme L_1 contiene stringhe di lunghezza ≥ 3, ognuna delle quali è una lista di due o più elementi a separati da una sola b oppure da bb, indifferentemente
- l'insieme L_2 contiene:
  - la stringa corta ba, di lunghezza 2

– stringhe di lunghezza ≥ 4, ognuna delle quali è una lista di due o più elementi a separati sempre da una sola b, tranne tra penultimo e ultimo elemento dove il separatore è sempre bb.

Ne viene che una lista di elementi a appartiene a L_1 ma non a L_2 nei casi seguenti:

- l'ultimo separatore è b (e gli altri indifferentemente b o bb)
- l'ultimo separatore è bb e la stringa contiene, da qualche altra parte, almeno un secondo separatore bb (o anche più di uno, al limite tutti); ciò implica peraltro che vi siano almeno tre elementi a

L_1	L_2	le tre stringhe più corte e l'espr. reg. di $L_1 \setminus L_2$
$a\ ((b \mid bb)\ a)^+$	$(a\ b)^*\ b\ a$	$a\ b\ a \qquad a\ b\ a\ b\ a \qquad a\ bb\ a\ b\ a$ $(a\ (b \mid bb))^*\ a\ b\ a \cup$ $(a\ (b \mid bb))^*\ a\ bb\ (a\ (b \mid bb))^*\ a\ bb\ a$

Le tre stringhe più corte (tutte relative al primo caso) e l'espressione regolare sono ricavate di conseguenza. Le spaziature mettono in evidenza i gruppi di lettere.

Addendum Per completezza, ci sono due stringhe più corte in quarta posizione, "$a\,b\,a\,b\,a\,b\,a$" e "$a\,bb\,a\,bb\,a$", e la seconda appartiene al secondo caso.

Nella seconda riga si interpretino a e b come parentesi aperta e chiusa, rispettivamente. Allora si ha quanto segue:

- l'insieme L_1 contiene tutte e sole le stringhe di parentesi che hanno profondità di annidamento ≤ 2 e soddisfano a uno dei due vincoli seguenti:
  - sono stringhe di Dyck ben costruite (sono i casi (1) e (3))
  - rispetto alle stringhe di Dyck ben costruite, hanno esattamente una parentesi aperta di troppo (è il caso (2))
- l'insieme L_2 contiene tutte e sole le stringhe di Dyck che hanno profondità di annidamento ≤ 2

Pertanto le stringhe della differenza $L_1 \setminus L_2$ sono quelle (di Dyck con profondità ≤ 2) aventi esattamente una parentesi aperta di troppo, vale a dire il caso (2) di L_1.

L_1	L_2	le tre stringhe e l'espr. reg. di $L_1 \setminus L_2$
$(a\ (ab)^*\ b)^*\ (\underbrace{\varepsilon}_{1} \mid \underbrace{a\ (ab)^*}_{2} \mid \underbrace{ab}_{3})$	$(a\ (ab)^*\ b)^*$	$a \qquad a\ ab \qquad ab\ a$ $(a\ (a\,b)^*\ b)^*\ a\ (a\,b)^*$

Le tre stringhe più corte e l'espressione regolare sono ricavate di conseguenza. Le spaziature mettono in evidenza i gruppi di lettere.

Addendum In alternativa, si può osservare che in L_1 il caso (3) è contenuto nel caso (1) e il caso (2) è disgiunto dal caso (1) (e a maggior ragione dal caso (3)), e che L_2 coincide con il caso (1) di L_1. Ne segue che la differenza $L_1 \setminus L_2$ coincide con il caso (2) di L_2, la stessa conclusione raggiunta con il ragionamento precedente.

Addendum In alternativa, si possono elencare le tre stringhe più corte prima ricavando le stringhe abbastanza brevi di L_1 e L_2, e poi facendone la differenza insiemistica.

Addendum La seconda espressione regolare è semplificabile ulteriormente:

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

Ciò si può giustificare facilmente esaminando un po' di stringhe di esempio, abbastanza corte da non allungare troppo l'esame.

Tuttavia si arriva alla semplificazione anche riflettendo che una stringa di Dyck di profondità ≤ 2 e con esattamente una parentesi aperta a di troppo, deve essere strutturata come segue:

- inizia certamente con a, giacché non può iniziare con b se deve essere di Dyck e per giunta contenere da qualche parte una a di troppo
- prosegue con a e b in uguale numero, cosicché al termine complessivamente contenga esattamente una a di troppo (ricordando la a iniziale, che sbliancia il conto; nota bene però che non è detto la a eccedente sia proprio quella iniziale)
- ma non contiene da nessuna parte come fattore né aaa né bbb (tre o più a o b consecutive), cosicché non abbia mai profondità > 2 (infatti, entrambi i fattori vietati implicherebbero profondità di annidamento pari ad almeno 3 o superiore)

Tale è precisamente il comportamento dell'espressione regolare semplificata a destra, che inizia con a e prosegue con a e b in numero uguale e ordine del tutto libero, ma senza mai formare i fattori vietati perché non accosta mai più di due lettere identiche.

Per inciso, ciò mostra come il linguaggio sia di tipo locale ed esprimibile anche nel modo seguente (dove $\Sigma = \{a, b\}$ e la soprallineatura indica il complemento):

$$a\ (\ b\,a\ \mid\ a\,b\ )^* = a\ \left(\Sigma^2\right)^* \cap \overline{\Sigma^*\ \{\,a\,a\,a,\ b\,b\,b\,\}\ \Sigma^*}$$

La semplificazione (forse un po' inattesa) mostra come ci possano essere linee di ragionamento notevolmente diverse per raggiungere il medesimo scopo.

2. Dato un automa a stati finiti deterministico M che riconosce il linguaggio $L(M)$, si considerino le costruzioni seguenti (nell'ordine dato):

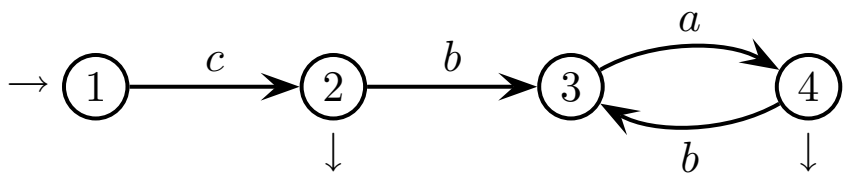
   i. trasforma M nell'automa N_R che riconosce il linguaggio riflesso $L(M)^R$

   ii. trasforma N_R nell'automa deterministico M_R a esso equivalente

   iii. trasforma M_R nell'automa M_{RR} che riconosce il linguaggio riflesso $L(M_R)^R$

   Si svolgano i punti seguenti:

   (a) Si eseguano le costruzioni (**i**), (**ii**) e (**iii**) (nell'ordine dato), partendo dall'automa M seguente e disegnando gli automi N_R, M_R e M_{RR}.

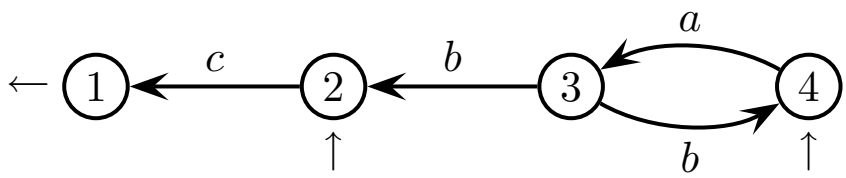


   (b) Si confrontino gli automi M e M_{RR}, spiegando in che cosa differiscano e perché.

Soluzione

(a) L'automa M accetta liste (non vuote) di tipo $c\ (\,b\,a\,)^*$: esse iniziano sempre con elemento c e poi (eventualmente) continuano solo con elementi a; gli elementi (c o a) sono sempre separati da esattamente una b. Gli automi costruiti dopo ogni passo sono elencati progressivamente di seguito.

i. Ecco l'automa (nondeterministico) N_R, ottenuto come riflessione di M (ha due stati iniziali):

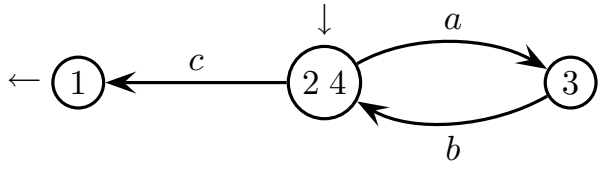


L'automa è nondeterministico a causa dello stato 3, da dove partono due archi entrambi etichettati con b, e a causa dei due stati iniziali 2 e 4.

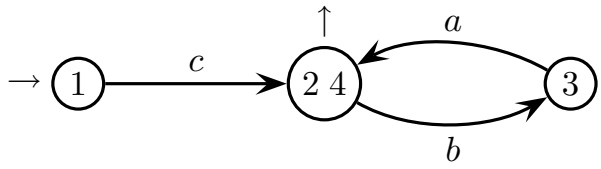
ii. Ecco l'automa deterministico M_R (equivalente a N_R), ottenuto applicando la costruzione dei sottinsiemi, dove (essendoci più stati iniziali) la tabella dei successori deve partire dal gruppo di stati iniziali [2 4], il quale andrà a costituire l'unico stato iniziale di M_R:

#	a	b	c	**finale ?**
2 4	3	$\emptyset$	1	no
3	$\emptyset$	2 4	$\emptyset$	no
1	$\emptyset$	$\emptyset$	$\emptyset$	sì
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	no

Ecco il grafo degli stati di M_R, dove come d'uso si omette lo stato di errore:



iii. Ed ecco l'automa M_{RR}, ottenuto come riflessione di M_R:



(b) Chiaramente il linguaggio riconosciuto da M_{RR} non è altri che L stesso, poiché si ha $(L^R)^R = L$. Pertanto l'automa M_{RR} equivale a quello iniziale M, ma ne differisce nel numero minore di stati, essendo M_{RR} minimo come ora si vedrà.
In altre parole, i passi (i), (ii) e (iii) costituiscono una procedura di minimizzazione, come si può comprendere dal ragionamento seguente. Due stati p e q di M sono indistinguibili se gli insiemi di etichette y dei cammini che da p e q portano a stati finali sono eguali. Si pensi a una stringa y tale che:

$$p \xrightarrow{y} f \qquad\qquad q \xrightarrow{y} g$$

dove f e g siano stati finali (distinti o coincidenti). Nell'automa riflesso determinizzato M_R, la stringa riflessa y^R etichetta un cammino dallo stato iniziale (unico) a un gruppo $[\ldots, p, \ldots, q, \ldots]$ (costruito dall'algoritmo di determinizzazione) contenente gli stati p e q.
Costruendo l'automa riflesso M_{RR}, gli stati non mutano, dunque gli stati indistinguibili p e q si trovano a essere fusi nel gruppo $[\ldots, p, \ldots, q, \ldots]$, e il risultato è lo stesso che si sarebbe ottenuto con l'algoritmo classico di minimizzazione.

2 Grammatiche libere e automi a pila 20%

1. Si consideri un sottinsieme del linguaggio di Dyck di alfabeto { ‘(’, ‘)’, ‘[’, ‘]’, ‘{’, ‘}’ }, il quale rispetti entrambi i vincoli seguenti:

   - le coppie di parentesi quadre possono solo contenere un numero pari di coppie di parentesi tonde (a qualunque livello di profondità)
   - le coppie di parentesi graffe possono solo contenere un numero dispari di coppie di parentesi tonde (a qualunque livello di profondità)

   Esempi di stringhe corrette:

   ((( ) ( ))) ((( ))) (([( ) ( )])) **{** ((( ))) **}**

   [( ) ( )] [(( ))] **{** [(**{** (( ) ( )) **}**)] ([(**{** ( ) **}**)]) **}**

   Esempi di stringhe errate:

   [( ) ( ) ( )] **{** (( )) **}** **{** [((( ) ( )))] **}** [(**{** ( ) ( ) **}**) ( )]

   Si svolgano i punti seguenti:

   (a) Si progetti una grammatica BNF (non estesa) non ambigua che generi il linguaggio descritto sopra.

   (b) Si spieghi a parole, in forma concisa ma efficace, come funziona la grammatica progettata al punto precedente.

Soluzione

(a) Ecco una soluzione possibile, BNF e non ambigua, modellata per restrizione sulla grammatica di Dyck non ambigua canonica (assioma S):

$$
\begin{array}{lcl}
S & \to & \text{‘[’ } P \text{ ‘]’ } S \;\mid\; \text{‘\{’ } D \text{ ‘\}’ } S \;\mid\; \text{‘(’ } S \text{ ‘)’ } S \;\mid\; \varepsilon \\
P & \to & \text{‘[’ } P \text{ ‘]’ } P \;\mid\; \text{‘\{’ } D \text{ ‘\}’ } D \;\mid\; \text{‘(’ } P \text{ ‘)’ } D \;\mid\; \text{‘(’ } D \text{ ‘)’ } P \;\mid\; \varepsilon \\
D & \to & \text{‘[’ } P \text{ ‘]’ } D \;\mid\; \text{‘\{’ } D \text{ ‘\}’ } P \;\mid\; \text{‘(’ } P \text{ ‘)’ } P \;\mid\; \text{‘(’ } D \text{ ‘)’ } D
\end{array}
$$

La grammatica è presentata in forma ordinata, in modo da mettere in evidenza affinità e differenze tra regole. Si osservi in particolare:

- l’uso di P o D nelle coppie di parentesi quadre o graffe, rispettivamente
- l’uso di S nella prima regola, per generare coppie di parentesi tonde non annidate in quadre o graffe
- l’alternanza tra P e D passando dalla seconda alla terza regola

Si veda il punto (b) per un commento più articolato ed esauriente.

(b) L'idea di fondo è che le classi sintattiche (nonterminali) P e D codifichino la generazione di un numero pari e dispari di coppie di parentesi tonde, rispettivamente. Le regole di produzione sono tutte forme specializzate della regola fondamentale di Dyck, cioè "$S \to$ '(' S ')' $S \mid \varepsilon$", dove la classe sintattica (nonterminale) S è sostituita delle combinazioni di nonterminali P e D che conservano la parità (o la disparità) del numero di parentesi tonde generate, rispettivamente, come per esempio succede in "$D \to$ '(' P ')' P" perché "dispari = pari + 1 + pari", ecc. (ovviamente ε, cioè nessuna parentesi, vale 0 e si considera pari), mentre le parentesi quadre e graffe, le quali di per sé non contribuiscono al conteggio (che considera solo le tonde), figurano come contenenti solamente il nonterminale accettabile (P o D) secondo i vincoli di parità dati nel testo del problema.

All'inzio le parentesi tonde esterne non presentano vincoli di sorta e pertanto sono generate dalla classe sintattica S) (assioma). La non ambiguità della regola di Dyck assicura che la grammatica così ottenuta per restrizione (cioè escludendo certe derivazioni) sia pure non ambigua.

2. Si consideri un sottoinsieme semplificato del linguaggio C, con i concetti seguenti:

   - il programma è una lista (non vuota) di istruzioni di assegnamento in stile C, puramente sequenziale cioè senza condizionali o cicli
   - l'istruzione di assegnamento ha la forma sintattica seguente:

     ```
     <variabile> = <espressione> ;
     ```

   - a sinistra dell'operatore di assegnamento "`=`" può figurare un identificatore:
     - di variabile nominale
     - di variabile puntatore, preceduto dall'operatore di deriferimento "`*`"

     e non occorre preoccuparsi della dichiarazione di tali identificatori
   - nell'espressione possono figurare i tipi seguenti di termine:
     - variabile nominale
     - variabile puntatore deriferito
     - chiamata a funzione
   - la chiamata a funzione ha la forma sintattica seguente:

     ```
     <nome-funzione> ( <lista-parametri> )
     ```

     - il nome di funzione è un identificatore (qui non va dichiarato)
     - il parametro è una variabile nominale o un puntatore, deriferito o no
     - la funzione restituisce un valore intero o un puntatore, che va deriferito
     - i parametri della lista sono separati da "`,`" (virgola)
     - la lista di parametri può essere vuota
   - gli operatori ammessi nell'espressione sono "+" (addizione) e "/" (divisione)
   - l'espressione può contenere parentesi tonde per modificare la precedenza consueta tra gli operatori "+" e "/' (la divisione precede l'addizione)
   - l'operatore di deriferimento "`*`" ha precedenza massima

   Esempi:

   ```
   z = x / (y + f (z, w));

   w = x / (y + *p (l, *m));

   *n = y + x / g (z, *m, *i) + z;
   ```

   Si svolgano i punti seguenti:

   (a) Si progetti una grammatica EBNF (estesa), non ambigua, che modella il sottoinsieme del linguaggio C semplificato descritto sopra.

   (b) Si dica quali aspetti del problema proposto non si possono modellare tramite uno strumento sintattico come la grammatica EBNF.

Soluzione

(a) Ecco una soluzione ragionevole, in forma EBNF (estesa), dove le parentesi quadre indicano opzionalità (assioma ⟨PROGRAM⟩):

⟨PROGRAM⟩	→	(⟨ASSIGNMENT⟩ ';')$^+$
⟨ASSIGNMENT⟩	→	⟨LEFT_MEMBER⟩ '=' ⟨EXPRESSION⟩
⟨LEFT_MEMBER⟩	→	⟨NOMINAL_VAR⟩ | '*' ⟨POINTER_VAR⟩
⟨EXPRESSION⟩	→	⟨TERM⟩ ('+' ⟨TERM⟩)*
⟨TERM⟩	→	⟨FACTOR⟩ ('/' ⟨FACTOR⟩)*
⟨FACTOR⟩	→	⟨NOMINAL_VAR⟩ | '*' ⟨POINTER_VAR⟩ |
		| ['*'] ⟨FUNCT_CALL⟩
⟨FUNCT_CALL⟩	→	id '(' ⟨PARAM_LIST⟩ ')'
⟨PARAM_LIST⟩	→	⟨PARAMETER⟩ (',' ⟨PARAMETER⟩)*
⟨PARAMETER⟩	→	⟨NOMINAL_VAR⟩ | ['*'] ⟨POINTER_VAR⟩
⟨NOMINAL_VAR⟩	→	id - identificatore generico
⟨POINTER_VAR⟩	→	id - identificatore generico

L'operatore di deriferimento di funzione è opzionale, perché la funzione può restituire direttamente un valore intero o un puntatore a esso, e nel secondo caso occorre deriferirlo. Il parametro di funzione può anche essere puntatore, eventualmente deriferito; conformemente al testo del problema non si prevede di passare un'intera espressione come parametro attuale di funzione.

La grammatica ha struttura modulare e le classi sintattiche hanno nomi espressivi. Le classi NOMINAL_VAR e POINTER_VAR sono unificabili, volendo, giacché entrambe sono identificatori generici. Beninteso ci possono essere soluzioni differenti, più o meno equivalenti a quella data sopra.

(b) Non sono modellabili sintatticamente gli aspetti semantici seguenti:

- Dichiarare la variabile (di tipo nominale o puntatore) prima di usarla.
- Dichiarare la funzione prima di chiamarla passandole parametri.
- Rispettare il numero di parametri da passare alla funzione e la compatibilità di tipo tra parametro formale (qui nemmeno dichiarato) e attuale.
- Distinguere tra variabile di tipo nominale piuttosto che puntatore; infatti la distinzione fatta sopra tra le classi NOMINAL_VAR e POINTER_VAR è chiara sintatticamente e rende più leggibile la grammatica, ma non è usabile per la verifica di tipo nell'espressione (e neppure per verificare la corrispondenza tra parametro formale e attuale) perché entrambi gli oggetti in ultimo sono risolti tramite la medesima classe sintattica, quella dell'identificatore generico "id", e pertanto sintatticamente sono indistinguibili.

Ci possono essere altri aspetti semantici non modellabili sintatticamente, più o meno di dettaglio. Il lettore li cerchi da sé.

3 Analisi sintattica e parsificatori 20%

1. È data la grammatica G seguente (in forma estesa), con alfabeto terminale $\{a, b, c, d\}$ e assioma S:

$$
\begin{aligned}
S &\rightarrow A \dashv \\
A &\rightarrow (\; a \; A^* \; A \; \mid \; B \;) \; b \\
B &\rightarrow a \, C \; \mid \; c \\
C &\rightarrow d \, C \; \mid \; d
\end{aligned}
$$

Si svolgano i punti seguenti:

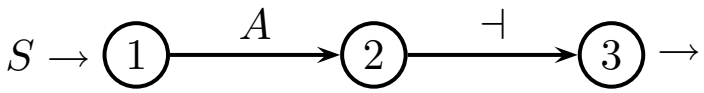
(a) Si rappresenti la grammatica G come rete di automi ricorsivi a stati finiti.

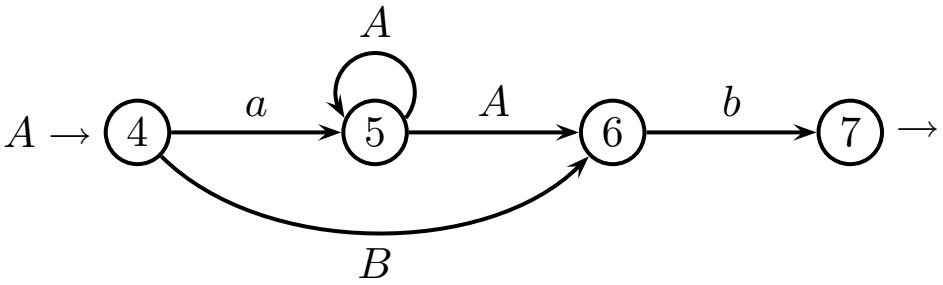
(b) Per ciascuno dei quattro nonterminali S, A, B e C, si stabilisca quale sia il minimo $k \geq 1$ per il quale esso risulta $LL(k)$.

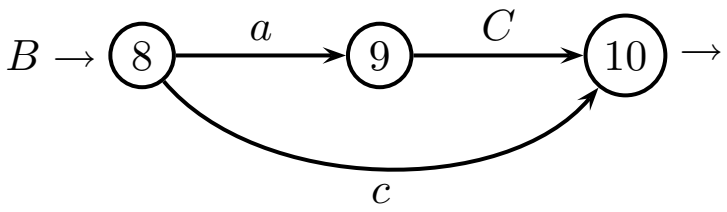
(c) (facoltativo) Se necessario, si progetti una nuova grammatica equivalente alla data, ma che sia $LL(k)$ per un qualche k minore (di quello trovato al punto (b)).

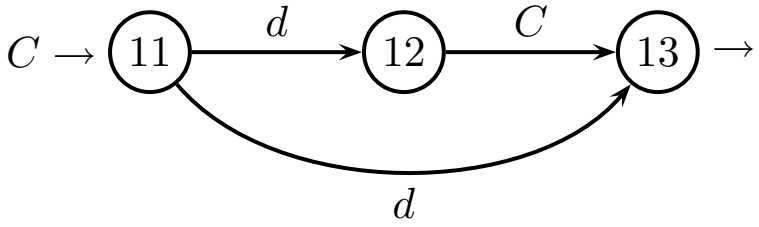
Soluzione

(a) Ecco la rete di automi ricorsivi a stati finiti, non deterministici, direttamente corrispondenti alle regole (in forma estesa) nel modo più ovvio, vale a dire secondo la costruzione modulare di Thompson (applicata però in modo da evitare da subito le transizioni spontanee):

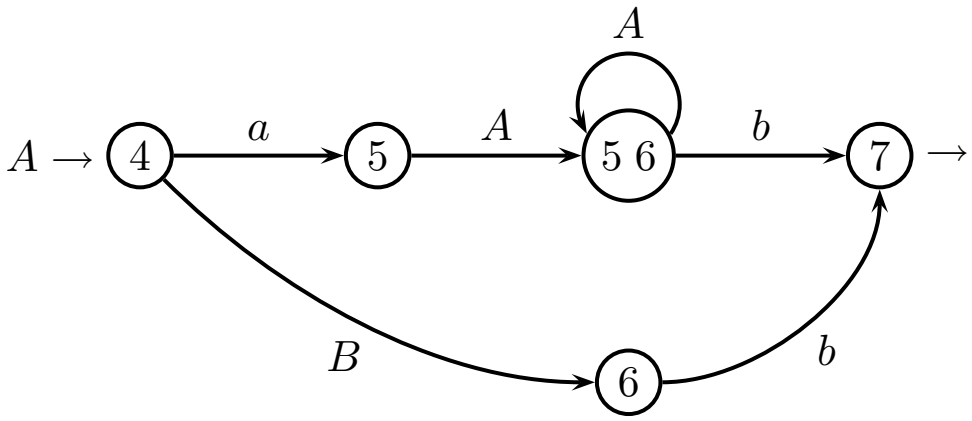


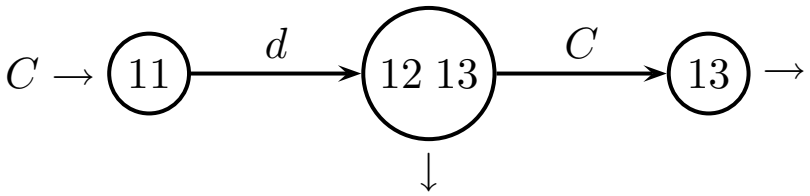






Non sono deterministici gli automi di A e C, che ai fini dell'analisi LL vanno dunque determinizzati (tramite la costruzione dei sottinsiemi, qui applicabile in modo pressoché intuitivo) rispetto all'alfabeto totale della grammatica G, cioè $\Sigma \cup V_N$ (simboli terminali e nonterminali). Eccoli:





Essi corrispondono a riformulare le regole estese nel modo seguente:

$$A \;\rightarrow\; a\ A\ A^*\ b \;\mid\; B\ b$$

$$C \;\rightarrow\; d\ (C \;\mid\; \varepsilon\ )$$

dove il nondeterminismo (rispetto all'alfabeto totale) è risolto.

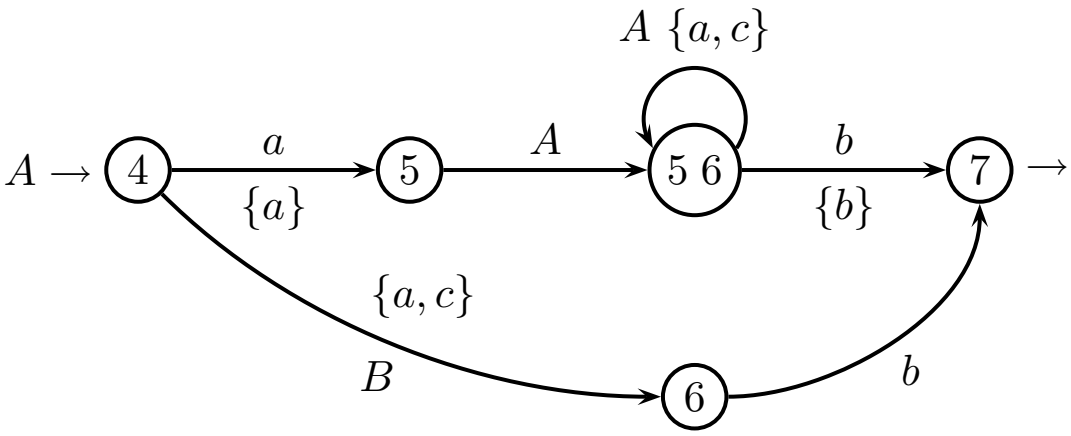
Oppure, se si preferisce, nel modo seguente (usando l'operatore croce):

$$A \;\rightarrow\; a\; A^{+}\; b \;\mid\; B\; b$$

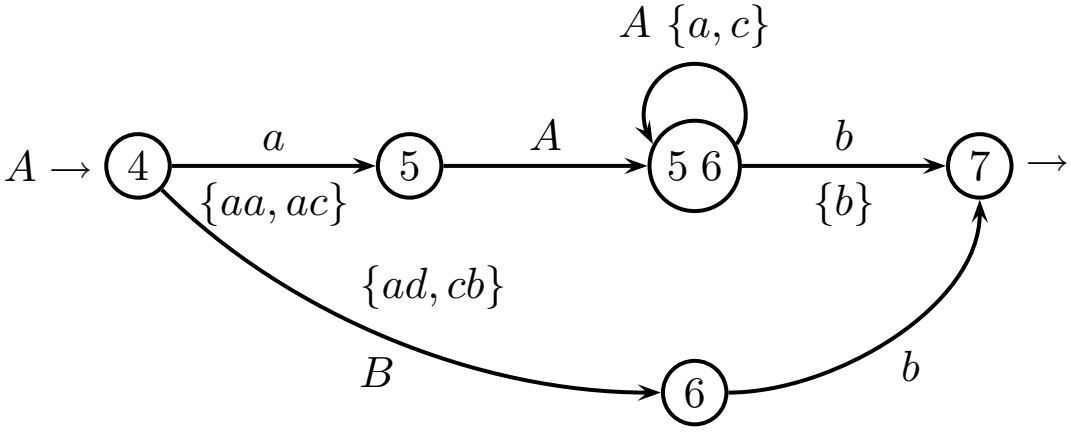
$$C \;\rightarrow\; d\; (C \;\mid\; \varepsilon\;)$$

Beninteso si possono tracciare direttamente le versioni deterministiche degli automi in questione, qui si è intenzionalmente proceduto in modo molto graduale.

(b) Si procede all'esame $LL(k)$, per ciascun nonterminale ordinatamente.
Il nonterminale S è chiaramente $LL(1)$ (non ci sono biforcazioni su nessun nodo).
Ecco gli insiemi guida di A (alle biforcazioni dei nodi 4 e 5 6), con $k = 1$:

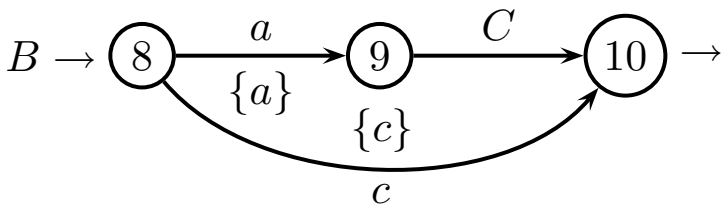


Poiché sul nodo 4 gli insiemi guida ancora non sono disgiunti, A non è $LL(1)$.
Per $k = 2$ essi sono (solo dove serve, cioè alla biforcazione del nodo 4):

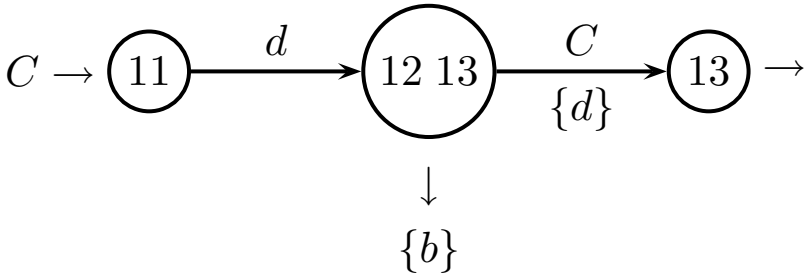


e poiché ora sono disgiunti, A è $LL(2)$.

Ecco gli insiemi guida di B (alla biforcazione del nodo 8) per $k = 1$ (sono banali):



Poiché essi sono disgiunti, B è $LL(1)$.
Ecco gli insiemi guida di C (alla biforcazione del nodo 12 13), con $k = 1$:



e poiché sono disgiunti, C è $LL(1)$.
Complessivamente, la grammatica G risulta essere $LL(2)$.

(c) Dato che la grammatica G è già $LL(k)$ con $k = 2$, al minimo si può cercare una nuova grammatica G', equivalente a G, che sia $LL(k)$ con $k = 1$. Per cominciare si osservi che il nonterminale B genera un linguaggio regolare, cioè:

$$L(B) = a\ d^+ \ \mid\ c$$

perché le regole che espandono B sono o terminali o lineari a destra. Poi si osservi che l'alternativa $A \to B\ b$ genera il linguaggio seguente (ancora regolare):

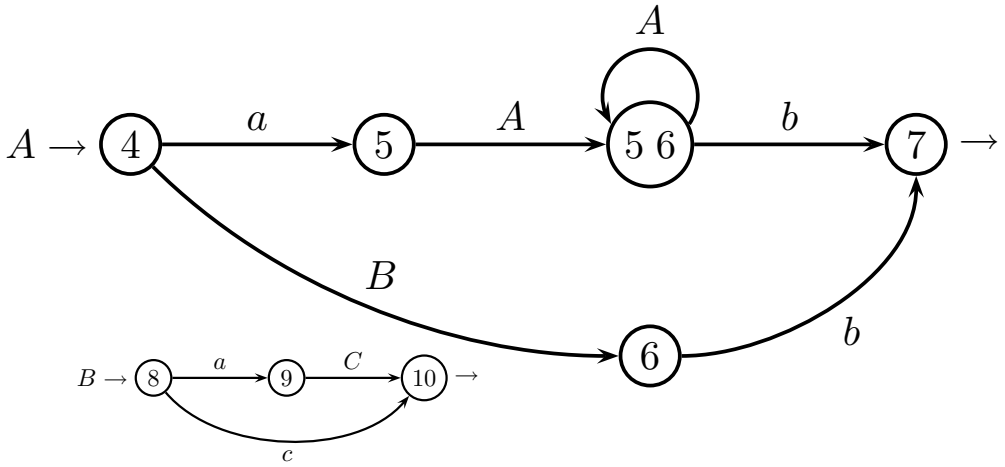
$$L(B\ b) = \left(\ a\ d^+ \ \mid\ c\ \right)\ b = a\ d^+\ b \ \mid\ c\ b$$

Riesaminando le regole che espandono A, si vede come il valore $k = 2$ derivi proprio dalla necessità di scegliere tra i primi due membri di $A \to (\ a\ A^*\ A \ \mid\ B\ )\ b = a\ A^+\ b \ \mid\ B\ b = \underbrace{a\ A^+\ b}_{1} \ \mid\ \underbrace{a\ d^+\ b}_{2} \ \mid\ \underbrace{c\ b}_{3}$, giacché entrambi iniziano con a. Viene pertanto spontaneo pensare di *fonderli*, riducendoli a uno solo. Fattorizzando a e b a sinistra e a destra, rispettivamente, nel primo e nel secondo membro, si arriva subito alla grammatica seguente (assioma S):

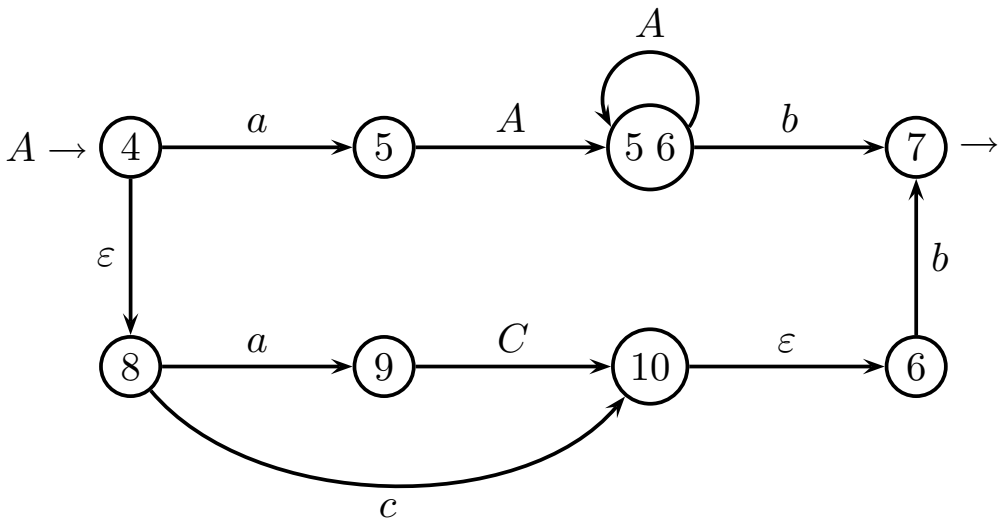
$$G' = \begin{cases} S & \to & A\ \dashv \\ A & \to & a\ (\ A^+ \ \mid\ d^+\ )\ b \ \mid\ c\ b \end{cases}$$

La nuova grammatica G' è in forma estesa ed è equivalente a quella originale (si provi con qualche stringa), e chiaramente è $LL(1)$ giacché il nonterminale A non è nullabile e inizia con a, cosicché le alternative nella regola sono tutte separabili con prospezione di ampiezza $k = 1$.

Addendum Si arriva più o meno allo stesso risultato lavorando per sostituzione direttamente sugli automi, se si preferisce. Si prenda l'automa che espande il nonterminale A e alla transizione con etichetta B, che fa parte della biforcazione responsabile di essere $LL(2)$, si sostituisca l'intero automa che espande il nonterminale B, nel modo seguente:

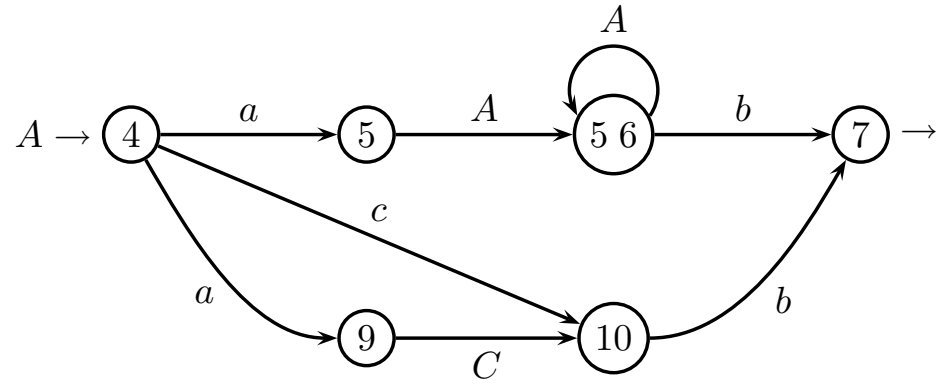


Ecco che cosa si ottiene (inserendo l'automa B tramite ε-transizioni):

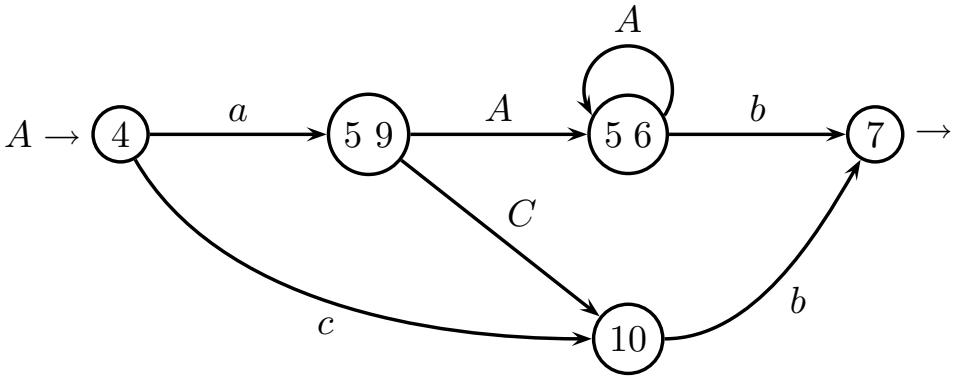


Dopo la sostituzione, l'automa è indeterministico (a causa delle ε-transizioni).

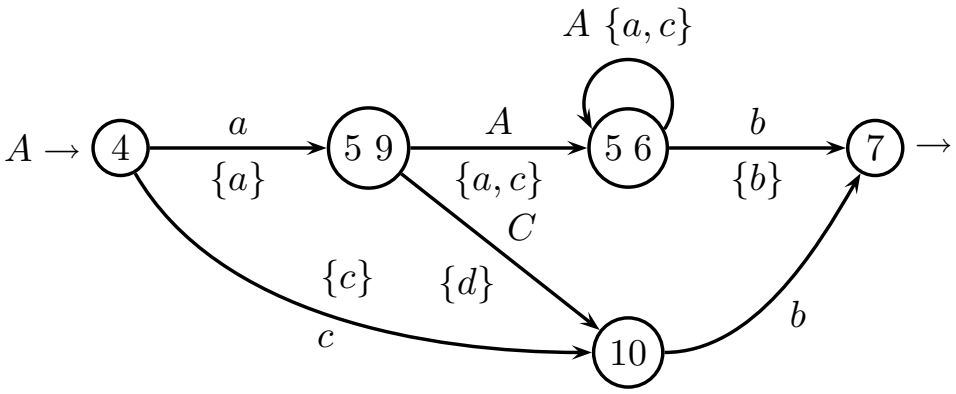
Ora si determinizzi l'automa rispetto all'alfabeto totale (terminali e nonterminali). Prima si taglino le ε-transizioni:



Tagliando le ε-transizioni e retraendo gli archi uscenti dai nodi 6 e 8, questi ultimi restano irraggiungibili e pertanto sono stati eliminati. Poi si applichi la costruzione dei sottinsiemi (qui del tutto banale):



Così l'automa è tornato deterministico. Ora basta ricalcolare gli insiemi guida alle biforcazioni (nodi 4, 5 9 e 5 6), per $k = 1$, ottenendo quanto segue:



Si constata subito come l'automa che espande il nonterminale A sia diventato $LL(1)$. Riscrivendo tale automa in forma di regola, si ottiene circa lo stesso risultato di prima (basta sostituire anche C, benché non sia necessario).

2. È data la grammatica G seguente, con alfabeto terminale $\{a, b\}$ e assioma S:

$$\begin{aligned} S &\rightarrow A \dashv \\ A &\rightarrow b\, B\, a \\ B &\rightarrow b\, B \mid a\, A \mid a \end{aligned}$$

Si svolgano i punti seguenti:

(a) Si argomenti in breve se il linguaggio $L(G)$ sia regolare o no.
(b) Si dimostri rigorosamente che la grammatica G è di tipo $LR(1)$.
(c) Si stabilisca rigorosamente se la grammatica G sia di tipo $LR(0)$.
(d) (facoltativo) Si discuta in breve se la grammatica G sia di tipo $LALR(1)$.

Soluzione

(a) Osservando la derivazione seguente di G (di tipo autoinclusivo in A e B):

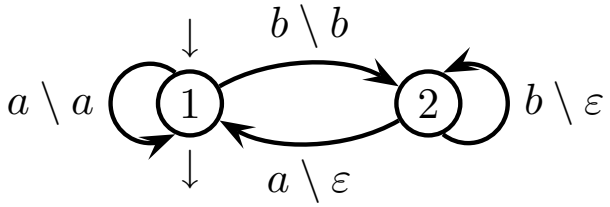
$$S \stackrel{*}{\Rightarrow} b^+Ba \Rightarrow b^+aAa \stackrel{*}{\Rightarrow} b^+ab^+Baa \Rightarrow b^+ab^+aAaa \Rightarrow b^+ab^+a \ldots a \ldots aa$$

si conclude subito che il linguaggio $L(G)$ ha l'aspetto generale seguente:

$$L(G) = \left\{ \underbrace{b^+a}_{1}\ \underbrace{b^+a}_{2}\ \ldots\ \underbrace{b^+a}_{n}\ a\ \underbrace{a}_{n}\ \ldots\ \underbrace{a}_{2}\ \underbrace{a}_{1} \;\middle|\; n \geq 1 \right\}$$

Chiaramente con un'espressione regolare non si riesce a garantire che il numero di fattori di tipo b^+a sia uguale al numero di a finali (escludendo quella centrale); dunque $L(G)$ non è regolare. Tale argomentazione è assai convincente.

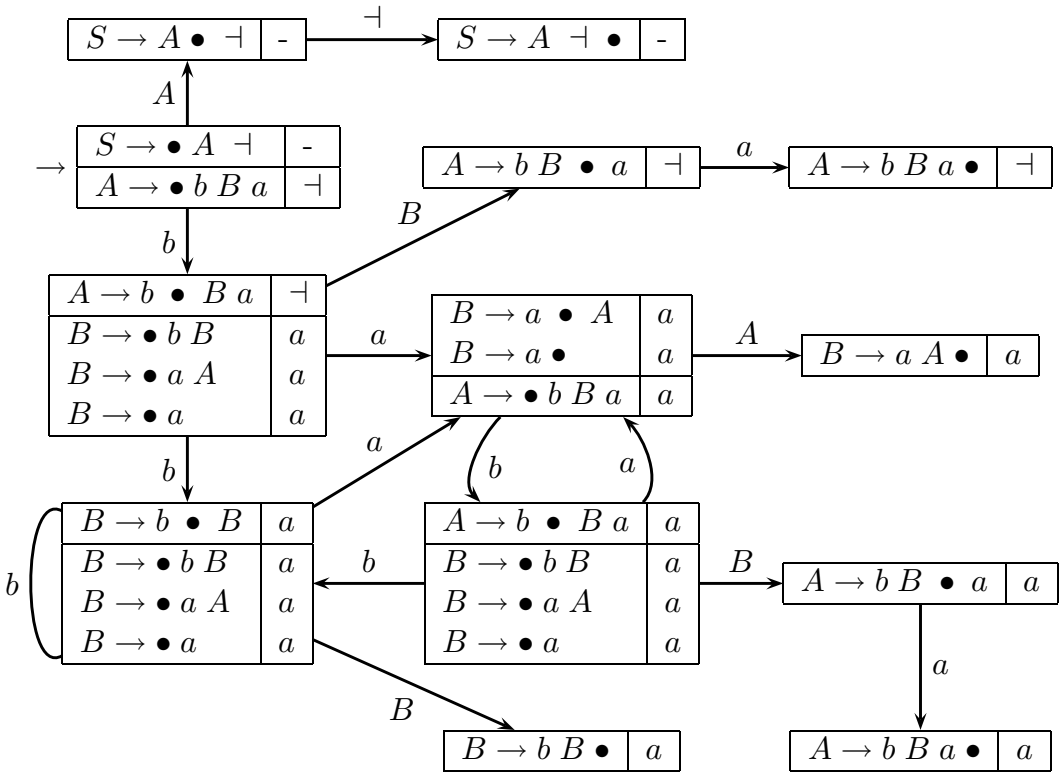
Addendum Un'argomentazione del tutto rigorosa è la seguente: dando una generica stringa di $L(G)$ in ingresso a un automa trasduttore a stati finiti τ opportunamente definito, in uscita ciascun fattore b^+a viene tradotto in una sola lettera b. Un automa trasduttore τ (deterministico) adatto allo scopo (ma non necessariamente l'unico) è il seguente:



Esso traduce la stringa b^+ emettendo una sola b e cancella la a immediatamente successiva, mentre le a rimanenti vengono emesse inalterate.

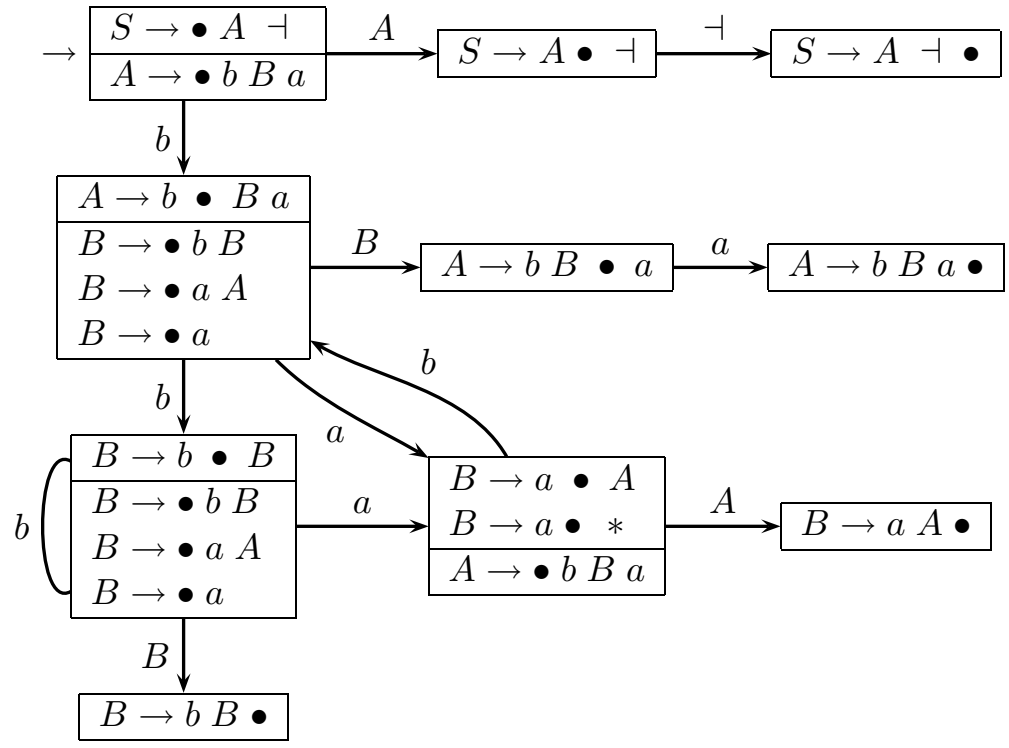
Ne segue che $\tau\,(L(G)) = \{b^n\, a\, a^n \mid \quad n \geq 1\}$, linguaggio notoriamente non regolare (ciò si dimostra, per esempio, mediante il "pumping lemma"). Ma dato che la famiglia dei linguaggi regolari è chiusa rispetto alla trasduzione a stati finiti, il linguaggio $L(G)$ non può essere regolare, altrimenti si avrebbe contraddizione.

(b) La grammatica G è effettivamente $LR(1)$, come asserisce il testo. Per dimostrarlo rigorosamente bisogna tracciarne il grafo pilota $LR(1)$ con prospezione. Eccolo:



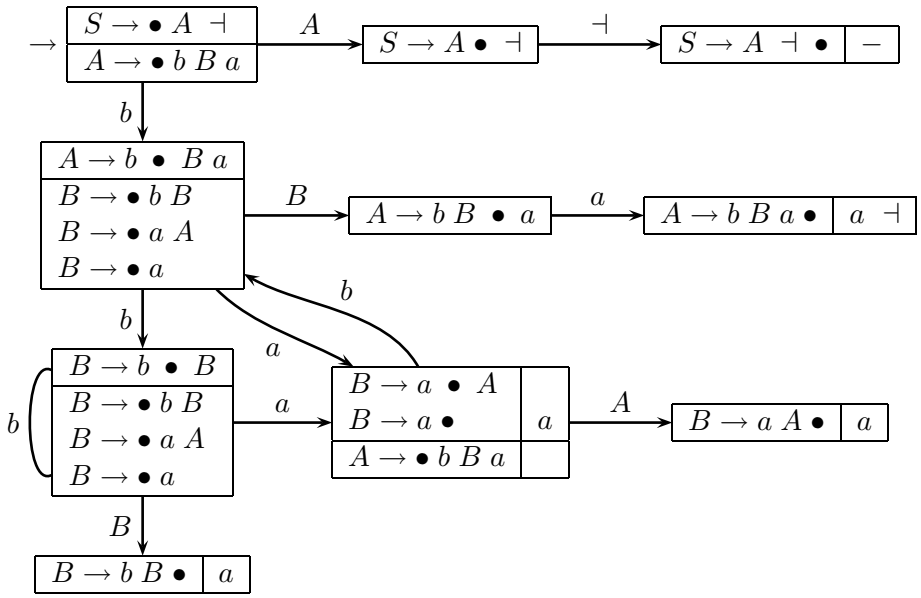
Tutti gli stati sono adeguati, dunque la grammatica G è effettivamente $LR(1)$.

(c) Avendo già il grafo $LR(1)$ e osservando che ci sono macrostati misti di riduzione e spostamento, si può concludere subito che la grammatica G non è $LR(0)$.
Tracciando il grafo pilota $LR(0)$ della grammatica G, si vede più estesamente che essa non è $LR(0)$. Ecco il grafo:



C'è uno stato inadeguato, con conflitto di tipo riduzione-spostamento (la riduzione è marcata con $*$), dunque la grammatica G non è $LR(0)$.
La regola di riduzione conflittuale è $B \to a\ \bullet$, la quale infatti contrasta con la regola di spostamento $B \to a\ \bullet\ A$. Entrambe infatti espandono lo stesso nonterminale, B, ma la prima è prefisso della seconda: esse finiscono dunque nello stesso macrostato (perché hanno lo stesso membro sinistro e vengono analizzate in parallelo), ma senza prospezione non è possibile decidere se la riduzione sia effettivamente da fare o se invece si debba procedere con lo spostamento.

(d) Risulta che la grammatica G è $LALR(1)$. Basta ripiegare il grafo pilota $LR(1)$ di G e constatare che il grafo pilota $LALR(1)$ così ottenuto (topologicamente identico al grafo pilota $LR(0)$) non contiene stati inadeguati, perché la prospezione riesce comunque a risolvere il conflitto di tipo riduzione-spostamento di prima. Ecco il grafo:



Tutti i macrostati sono adeguati, pertanto la grammatica G è $LALR(1)$.

4 Traduzione e analisi semantica 20%

1. Le frasi sorgente sono delle stringhe non vuote di alfabeto $\{a, c\}$, dove a e c stanno per parentesi aperta e chiusa, rispettivamente. Le frasi possono essere parentesizzate bene o male. Nel secondo caso (male) ci possono essere delle c in eccesso (come per esempio in $ac\overset{\downarrow}{c}ac$, dove la freccetta indica la c eccedente), mentre si vieta che manchino delle c (una stringa come aac è esclusa). Nota bene: la freccetta non fa parte del linguaggio, qui serve solo come ausilio visivo a vantaggio del lettore.

   Si vorrebbe avere un traduttore puramente sintattico (cioè senza attributi) capace di correggere una frase sorgente traducendola in una stringa ben parentesizzata di alfabeto $\{begin,\ end\}$, a essa corrispondente, inserendo le parentesi aperte mancanti in corrispondenza delle chiuse in eccesso. Ecco tre esempi (b ed e sono abbreviazioni di *begin* ed *end*, rispettivamente), che mostrano come funziona la correzione:

stringa sorg.	stringa eventualmente corretta e poi tradotta (con spiegazione)	
$a\,ac\,ac\,c$	$b\,be\,be\,e$	la stringa sorgente è giusta e pertanto viene solamente traslitterata sull'alfabeto destinazione
$\overset{\downarrow}{c}\,ac$	$b\overset{\downarrow}{e}\,be$	la stringa sorgente è errata e pertanto la lettera a mancante viene inserita dal traduttore come b
$a\,ac\,c\overset{\downarrow}{c}\overset{\downarrow}{c}\,ac$	$b\,be\,e\,b\overset{\downarrow}{e}\,b\overset{\downarrow}{e}\,be$	la stringa sorgente è errata e pertanto le due lettere a mancanti vengono inserite dal traduttore come b

   Si svolgano i punti seguenti:

   (a) Si progetti il grafo di un IO-automa a pila, preferibilmente deterministico, per calcolare la traduzione descritta sopra; si può supporre che la stringa sorgente abbia terminatore '$\dashv$'. Si spieghi a parole il funzionamento dell'automa.

   (b) (facoltativo) Si scriva una grammatica o uno schema per calcolare la traduzione.

Soluzione

(a) Lo IO-automa a pila richiesto è una variante dell'automa riconoscitore del linguaggio di Dyck, modificato in modo da comportarsi come segue:

- legge dal nastro di ingresso la parentesi aperta a (indipendentemente dalla pila), la impila ricodificandola come A ed emette b sul nastro di uscita
- se legge dal nastro di ingresso la parentesi chiusa c e la pila non è vuota, emette subito e sul nastro di uscita e spila una A
- se tuttavia legge c dal nastro di ingresso quando la pila è vuota, invece di andare in errore (come accadrebbe nel riconoscimento di Dyck) emette sul nastro di uscita la parentesi aperta mancante b, subito seguita dalla chiusa e corrispondente, e prosegue normalmente (senza toccare la pila)
- se arriva in fondo al nastro di ingresso (leggendo il terminatore '$\dashv$') e la pila è vuota, accetta la stringa letta, convalida la traduzione corrispondente emessa (cioè la versione corretta della stringa letta) e termina
- se arriva in fondo al nastro di ingresso (leggendo il terminatore '$\dashv$') e la pila non è vuota, va in errore (si ha eccesso di a, caso vietato per ipotesi)

Il solo simbolo di memoria è A, e se si adotta riconoscimento a pila vuota lo IO-automa è deterministico e monostatico. Sotto tale ipotesi, ecco l'elenco delle transizioni (siano q l'unico stato dell'automa e Z_0 il simbolo inziale di pila):

$$\begin{aligned}
\delta\,(q, a, Z_0) &= (q, b, Z_0 A) \\
\delta\,(q, a, A) &= (q, b, AA) \\
\delta\,(q, c, A) &= (q, e, \varepsilon) \\
\delta\,(q, c, Z_0) &= (q, b\,e, Z_0) \\
\delta\,(q, \dashv, Z_0) &= \text{non emettere niente, } \underline{\text{accetta}} \text{ e termina} \\
\delta\,(q, \dashv, A) &= \text{non emettere niente, } \underline{\text{rifiuta}} \text{ e termina } (a \text{ in eccesso})
\end{aligned}$$

Lo IO-automa a pila così progettato è palesemente deterministico e, in quanto tale, la trasduzione è a un valore, cioè la correzione fatta è univoca.

(b) Dato che c'è uno IO-automa a pila monostatico, essenzialmente esso è già lo schema sintattico di traduzione richiesto e va solo riscritto come grammatica di traduzione. Eccola (in forma combinata, con assioma S che corrisponde a Z_0):

$$\begin{aligned}
S &\to a\ \{b\}\ A\ S && \text{– traslittera } a \text{ (non annidata) in } b \\
A &\to a\ \{b\}\ A\ A && \text{– traslittera } a \text{ (annidata) in } b \\
A &\to c\ \{e\} && \text{– traslittera } c \text{ (annidata o no) in } e \\
S &\to c\ \{b\,e\}\ S && \text{– corregge inserendo } b \text{ e traslittera } c \text{ in } e \\
S &\to \dashv && \text{– termina la derivazione}
\end{aligned}$$

Le regole della grammatica sono ordinatamente in corrispondenza con le mosse dell'automa. Ovviamente qui l'ultima regola, quella di rifiuto, non figura più, perché semplicemente la parte sorgente non genera stringhe con eccesso di a. La grammatica in questione è deterministica $LL(1)$; ciò si verifica facilmente e del resto è ovvio per come è stata ottenuta; la trasduzione è dunque a un valore.

Addendum Ecco un'altra soluzione (in forma separata) derivata della grammatica canonica (non ambigua) di Dyck (cioè da $S \to a\,S\,c\,S \mid \varepsilon$):

G_s	G_p
$S \to a\ S\ c\ S$	$S \to b\ S\ e\ S$
$S \to \varepsilon\ S\ c\ S$	$S \to b\ S\ e\ S$
$S \to \varepsilon$	$S \to \varepsilon$

La soluzione data è semplicissima e risponde in pieno al punto (b) dell'esercizio. Giusto come osservazione, si noti che la soluzione data è fortemente indeterministica e non è $LL(k)$ per nessun k (il determinismo peraltro non è richiesto). La trasduzione infatti non è a un valore; per esempio, la stringa sorgente errata $a\,c\,c$ viene corretta e tradotta sia come $b\,e\,b\,e$ sia come $b\,b\,e\,e$, secondo si immagini che la c in eccesso dovesse essere $a\,c\ \overset{\downarrow}{c}$ oppure $a\ \overset{\downarrow}{c}\ c$, o equivalentemente che la stringa sorgente corretta dovesse essere $a\,c\,a\,c$ oppure $a\,a\,c\,c$, rispettivamente. In effetti basta poco per convincersi che la soluzione data, per ogni stringa sorgente errata emette tutte le possibili forme corrette e tradotte.

Incidentalmente, l'osservazione torna utile per aggiungere che trasformando la semplicissima soluzione data in automa a pila monostatico (secondo la trasformazione standard da grammatica ad automa a pila monostatico, in generale non deterministico, come è esposta nel libro di testo), si ottiene istantaneamente anche la risposta al punto (a), benché con un automa indeterministico (il determinismo peraltro è indicato solo come preferenziale, non obbligatorio).

Addendum Solo come ulteriore esercizio, ecco una soluzione deterministica (in forma separata), anch'essa derivata per restrizione della grammatica canonica (non ambigua) di Dyck (cioè da $S \to a\,S\,c\,S \mid \varepsilon$):

G_s	G_p
$S \to a\,T\,c\,S \mid \varepsilon \mid c\,C$	$S \to b\,T\,e\,S \mid \varepsilon \mid b\,e\,C$
$T \to a\,T\,c\,T \mid \varepsilon$	$T \to b\,T\,e\,T \mid \varepsilon$
$C \to c\,C \mid \varepsilon$	$C \to b\,e\,C \mid \varepsilon$

La restrizione consiste nell'idea di modificare la grammatica facendole emettere in aggiunta la parentesi aperta b mancante quando la stringa sorgente presentasse una parentesi chiusa c che non fosse né annidata né preceduta immediatamente da una parentesi aperta a, perché solo allora si potrebbe deterministicamente considerare la c come in eccesso. Ciò si ottiene differenziando la generazione delle parentesi chiuse annidate dalla generazione di quelle concatenate senza annidamento; le seconde sono generate dal nonterminale C, le cui regole in G_p effettuano la correzione inserendo b. Le alternative blu sono equivalenti alle solite regole di Dyck, quelle rosse realizzano la correzione. La soluzione che così si ricava è più prolissa della precedente, ma è deterministica $LL(1)$, come si verifica facilmente. Lo schema di traduzione è certamente a un valore, cioè la correzione fatta è univoca (tra le tante possibili, come si è visto prima).

2. Si consideri un programma costituito da una lista sequenziale di *istruzioni* (senza condizionali o cicli), ognuna delle quali è modellata per mezzo di due insiemi: le variabili che l'istruzione *definisce* e quelle che *usa*. Per esempio, l'istruzione di assegnamento"$c = c + a$" è modellata da "def c" e "usa a, c", mentre l'istruzione di I/O "$read(a, c)$" è modellata da "def a, c" e "usa $\emptyset$".

   Il supporto sintattico è il seguente (assioma S):

$$
\begin{aligned}
S &\rightarrow L \\
L &\rightarrow \langle istr \rangle\ L \\
L &\rightarrow \langle istr \rangle \\
\langle istr \rangle &\rightarrow \text{`def'}\ D\ \text{`usa'}\ U\ \text{`;'} \\
D &\rightarrow \text{`}\emptyset\text{'} \\
D &\rightarrow V \\
U &\rightarrow \text{`}\emptyset\text{'} \\
U &\rightarrow V \\
V &\rightarrow \text{`id'}\ \text{`,'}\ V \\
V &\rightarrow \text{`id'}
\end{aligned}
$$

   Il simbolo "id" è un identificatore di variabile, e qui è modellato sintatticamente come terminale. Il nome dell'identificatore è un attributo lessicale disponibile, di nome n.

   Si svolgano i punti seguenti:

   (a) Si definiscano gli attributi necessari e si scrivano le regole semantiche di una grammatica con attributi, per calcolare come attributo l'insieme delle *variabili vive* all'ingresso della prima istruzione del programma (si usino gli schemi all'uopo già predisposti alle pagine successive).

   (b) Si decori l'*albero sintattico* che figura più avanti, scrivendo nei nodi i valori degli attributi calcolati dalla grammatica progettata al punto precedente.

   (c) Si disegnino i *grafi delle dipendenze* tra gli attributi e si scriva, almeno in parte, il codice del valutatore semantico adatto ai tipi di dipendenza riscontrati.

---

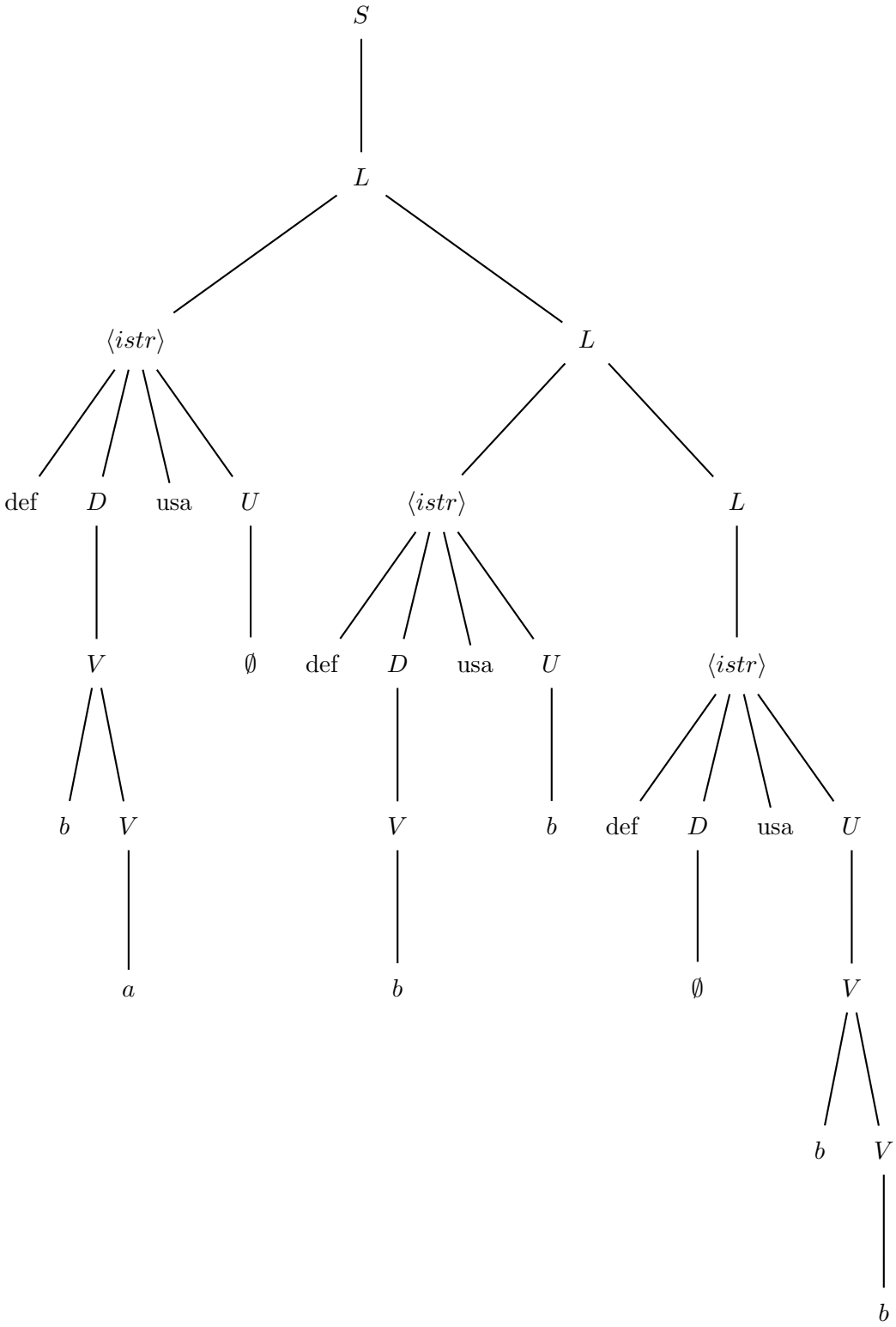
attributi da usare per la grammatica

tipo	nome	(non)terminali	dominio	significato
già dati nel testo dell'esercizio				
dx	n	id	stringa	nome dell'identificatore
da aggiungere o estendere				

sintassi	funzioni semantiche
$S_0 \rightarrow L_1$	
$L_0 \rightarrow \langle istr \rangle_1 \; L_2$	
$L_0 \rightarrow \langle istr \rangle_1$	
$\langle istr \rangle_0 \rightarrow$ 'def' D_1 'usa' U_2 ';'	
$D_0 \rightarrow$ '$\emptyset$'	

sintassi	funzioni semantiche
$D_0 \rightarrow V_1$	
$U_0 \rightarrow$ '$\emptyset$'	
$U_0 \rightarrow V_1$	
$V_0 \rightarrow$ 'id' ',' V_1	
$V_0 \rightarrow$ 'id'	

albero sintattico da decorare



Soluzione

Soluzione estesa Si dà prima una soluzione molto estesa, che modella il calcolo delle varibili vive come è presentato comunemente. Poi se ne darà un'altra, altrettanto valida, ma più semplice e rapida da ideare e scrivere.

(a) Ecco l'elenco degli attributi:

attributi da usare per la grammatica

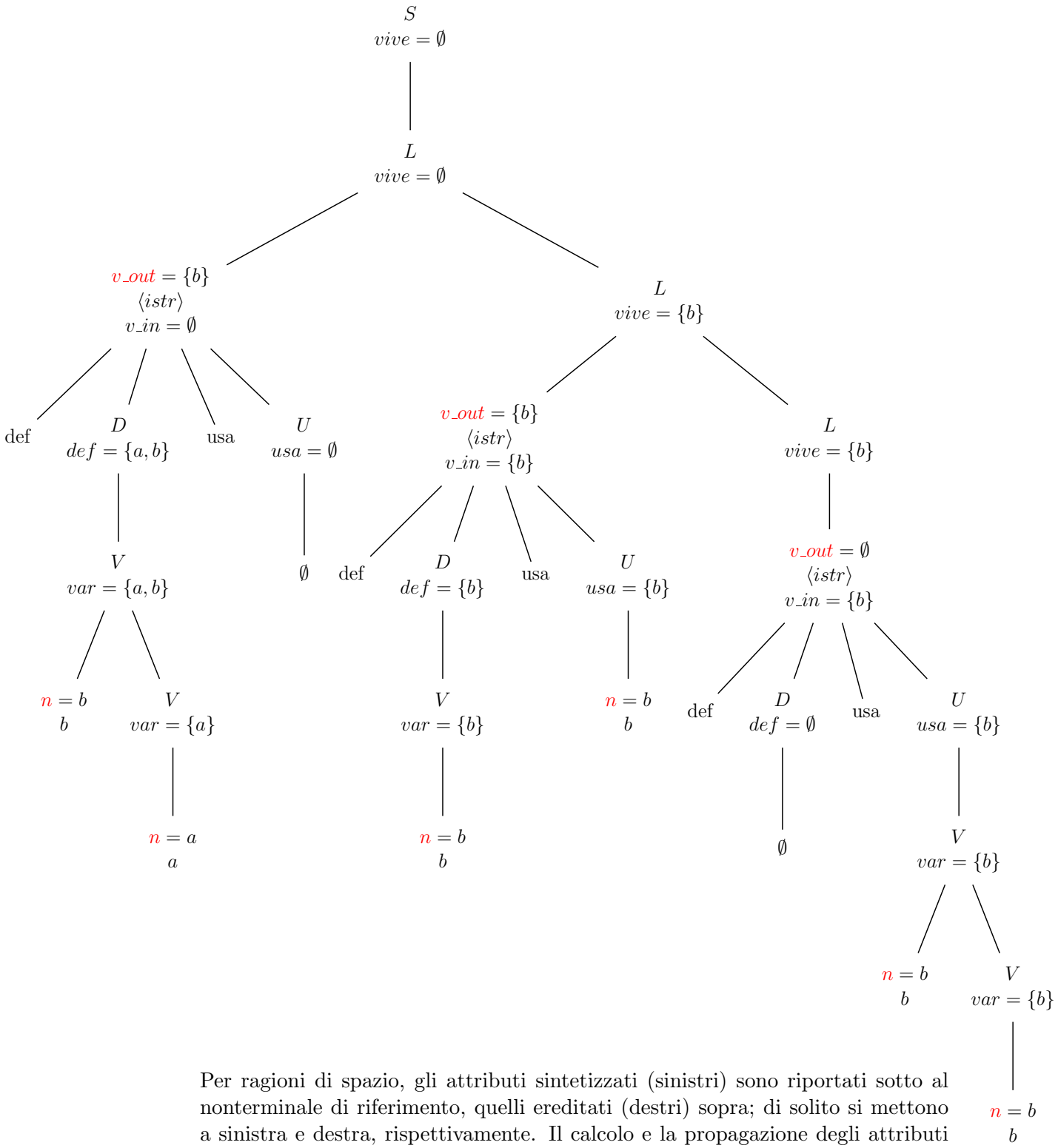
tipo	nome	(non)terminali	dominio	significato
già dati nel testo dell'esercizio				
dx	n	id	stringa	nome dell'identificatore
da aggiungere o estendere				
sx	*var*	V	insieme di stringhe	lista di nomi di variabile
sx	*def*	D	idem	variabili definite da istr. corr.
sx	*usa*	U	idem	variabili usate da istr. corr.
sx	*v_in*	$\langle istr \rangle$	idem	variabili vive in ingresso a istr.
dx	*v_out*	$\langle istr \rangle$	idem	variabili vive in uscita da istr.
sx	*vive*	L, S	idem	variabili vive del programma

Il solo attributo di tipo destro significativo è *v_out* e rappresenta l'insieme delle variabili vive in uscita all'istruzione corrente, perché viene ereditato dalle variabili vive in ingresso alla classe sintattica L che genera le istruzioni a valle di quella corrente. L'attributo n è associato a un terminale, "id", e pertanto d'ufficio è supposto di tipo destro. Comunque qui si tratta di un attributo precalcolato (e proprio per tale ragione è supposto di tipo destro, appunto), di importanza limitata. Tutti gli altri attributi sono di tipo sinistro (sintetizzato).

Gli attributi sono definiti in modo da simulare il calcolo standard delle variabili vive con il metodo delle equazioni di flusso, su un programma puramente sequenziale. In particolare le variabili vive in ingresso e uscita dell'istruzione corrente, cioè gli attributi *v_in* (sta per vive-in-ingresso) e *v_out* (sta per vive-in-uscita) rispettivamente, sono associate al nodo $\langle istr \rangle$ che rappresenta l'istruzione, esattamente come si fa nel calcolo standard. Un attributo addizionale, *vive*, raccoglie le variabili vive all'ingresso di ciascuna istruzione e le propaga verso l'alto, fino all'inizio del programma, seguendo la catena di nonterminali L; esso dunque simula il processo di soluzione iterativa delle equazioni di flusso. L'ereditarietà dell'attributo *v_out* serve per portare le variabili vive all'uscita dell'istruzione corrente, giacché lì sono necessarie per ricalcolare le variabili vive all'ingresso dell'istruzione stessa. Seguono le funzioni semantiche:

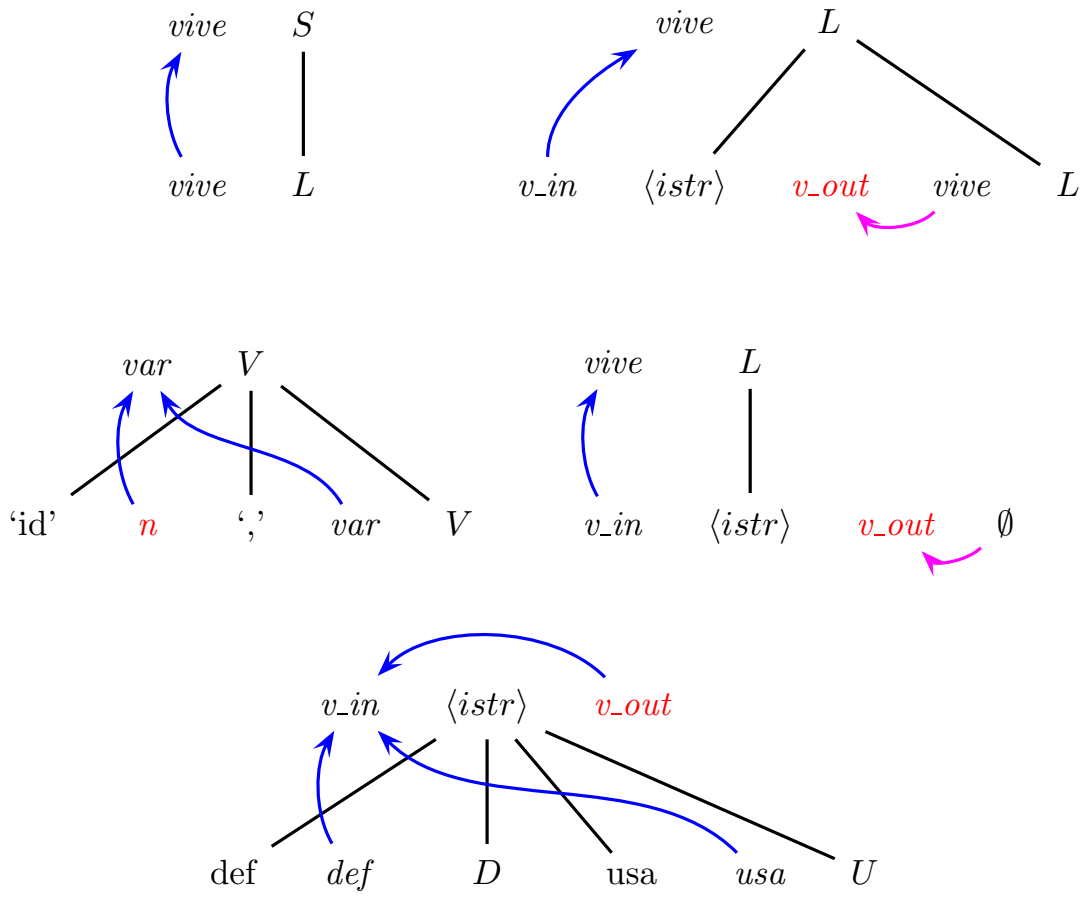
sintassi	funzioni semantiche
$S_0 \rightarrow L_1$	$vive_0 = vive_1$
$L_0 \rightarrow \langle istr \rangle_1\ L_2$	$v\_out_1 = vive_2$ $vive_0 = v\_in_1$
$L_0 \rightarrow \langle istr \rangle_1$	$v\_out_1 = \emptyset$ $vive_0 = v\_in_1$
$\langle istr \rangle_0 \rightarrow$ 'def' D_1 'usa' U_2 ';'	$v\_in_0 = usa_2 \cup (v\_out_0 - def_1)$
$D_0 \rightarrow$ '$\emptyset$'	$def_0 = \emptyset$
$D_0 \rightarrow V_1$	$def_0 = var_1$
$U_0 \rightarrow$ '$\emptyset$'	$usa_0 = \emptyset$
$U_0 \rightarrow V_1$	$usa_0 = var_1$
$V_0 \rightarrow$ 'id' ',' V_1	$var_0 = \{n\} \cup var_1$
$V_0 \rightarrow$ 'id'	$var_0 = \{n\}$

(b) Ecco l'albero sintattico decorato:



Per ragioni di spazio, gli attributi sintetizzati (sinistri) sono riportati sotto al nonterminale di riferimento, quelli ereditati (destri) sopra; di solito si mettono a sinistra e destra, rispettivamente. Il calcolo e la propagazione degli attributi seguono le funzioni semantiche date prima. Gli attributi ereditati *v_out* e *n* sono colorati in rosso, per distinguerli a prima vista.

(c) Tutti gli attributi sono sintetizzati, tranne *v_out* e *n* che sono ereditati. Ecco le dipendenze funzionali tra attributi (solo quelle delle regole più importanti):



Le regole rimanenti sono banali e qui non sono mostrate. Esaminando le dipendenze (in blu per i sintetizzati, in rosa per gli ereditati) si vede facilmente che la grammatica con attributi è ben definita, perché le dipendenze sono palesemente acicliche, e che inoltre rispetta la condizione one-sweep, in quanto:

- gli attributi sintetizzati dipendono solo da attributi dei nodi figli oppure da altri attributi (di qualunque tipo) dello stesso nodo
- l'attributo ereditato *v_out*, associato al sottoalbero $\langle istr \rangle$ nella regola "$L \to \langle istr \rangle\ L$", dipende da un attributo sintetizzato, *vive*, associato al sottoalbero L fratello (destro) di $\langle istr \rangle$ (nella stessa regola, ovviamente)

Giacché l'albero L si trova a destra di quello di $\langle istr \rangle$, per soddisfare la dipendenza di *v_out* basta valutare prima l'albero di L e poi quello di $\langle istr \rangle$.

Tuttavia la grammatica con attributi non è di tipo L (left), perché come detto prima occorre permutare i due sottoalberi $\langle istr \rangle$ e L, rispetto all'ordine di valutazione L, che di per sé procederebbe da sinistra verso destra.

Ecco le procedure semantiche (in pseudocodice) di L, $\langle istr \rangle$ e V, vale a dire le tre più strutturate (per rispondere alla domanda del testo, ne basterebbe una):

procedure L (*vive*: **out**; *albero*: **in**)
– variabili locali per passaggio attributi
    var *vive_tmp*
– alternativa "$L \rightarrow \langle istr \rangle\ L$"
    call L (*vive_tmp*, *albero* $\rightarrow L$)
    call ISTR (*vive*, *albero* $\rightarrow \langle istr \rangle$, *vive_tmp*)
– alternativa "$L \rightarrow \langle istr \rangle$"
    call ISTR (*vive*, *albero* $\rightarrow \langle istr \rangle$, $\emptyset$)
end L

Si noti come l'ordine di calcolo di $\langle istr \rangle$ e L sia permutato rispetto alla regola.

procedure ISTR (*v_in*: **out**; *albero*, *v_out*: **in**)
– variabili locali per passaggio attributi
    var *def_tmp*, *usa_tmp*
– regola "$\langle istr \rangle \rightarrow$ 'def' D 'usa' U"
    call D (*def_tmp*, *albero* $\rightarrow D$)
    call U (*usa_tmp*, *albero* $\rightarrow U$)
    $v\_in = usa\_tmp \cup (v\_out - def\_tmp)$
end ISTR

Qui l'ordine di chiamata delle procedure D e U è il medesimo della regola, perché sono coinvolti solo attributi sintetizzati (tranne n, che comunque è disponibile).

procedure V (*var*: **out**; *albero*: **in**)
– variabili locali per passaggio attributi
    var *var_tmp*
– alternativa "$V \rightarrow$ 'id' ',' V"
    – attributo n di 'id' precalcolato ...
    call V (*var_tmp*, *albero* $\rightarrow V$)
    $var = \{n\} \cup var\_tmp$
– alternativa "$V \rightarrow$ 'id'"
    – attributo n di 'id' precalcolato ...
    $var = \{n\}$
end V

Le parole chiave **in** e **out** indicano i parametri in ingresso (attributi ereditati) e in uscita (sintetizzati) alle procedure semantiche, rispettivamente; non vanno confuse con gli attributi *v_in* (sintetizzato, tipo **out**) e *v_out* (ereditato, tipo **in**) associati al nonterminale $\langle istr \rangle$. Le variabili locali (se ci sono) hanno il nome dell'attributo che memorizzano temporaneamente, con il suffisso *tmp* mnemonico; esse servono per trasportare attributi tra le procedure e calcolare le funzioni semantiche. Il parametro in ingresso *albero* è il puntatore al (sotto)albero sintattico che la procedura è chiamata ad analizzare. Si noti come la lista dei

parametri formali nella testata della procedura segua lo stesso ordine che si usa convenzionalmente nel decorare l'albero sintattico; il puntatore al (sotto)albero (che idealmente ne rappresenta il nonterminale radice) divide la lista in parte sinistra (attributi sintetizzati) e destra (ereditati); così si interpreta facilmente.

Le dipendenze dell'alternativa "$V \rightarrow$ 'id'" non sono mostrate al punto precedenze, ma sono ovvie. Come prima, gli attributi passati a una procedura semantica come parametri in ingresso (e pertanto ereditati) sono messi in evidenza in rosso (a parte che, come detto prima, figurano a destra del puntatore al (sotto)albero).

Le procedure semantiche rimanenti sono piuttosto banali e qui sono omesse. Si suppone l'albero sintattico sia già interamente costruito e passato tramite il parametro *albero*.

Addendum Beninteso ci possono essere altre soluzioni e quella data qui si può forse semplificare un po', pur rendendola meno leggibile e aderente alla struttura del calcolo standard delle variabili vive. Per esempio, si potrebbe cercare di eliminare l'attributo ereditato *v_out* (che, come si sarà forse già notato, ha una funzione piuttosto modesta, giacché serve solamente per trasferire le variabili vive dal nodo L al nodo fratello $\langle istr \rangle$) e ridurre il tutto a una soluzione puramente sintetizzata, magari rinunciando a simulare il calcolo delle variabili vive in modo assolutamente fedele all'algoritmo standard (scelta che del resto non è affatto imposta dal testo del problema, e che qui è stata preferita principalmente per dare un esempio articolato e valido in generale). Al riguardo si veda la prossima soluzione, data in bozza.

Soluzione semplificata Si usano solo attributi sintetizzati (a parte l'attributo n, banale e comunque già dato nel testo del problema). Rispetto alla soluzione estesa, l'idea è di eliminare gli attributi v_in e v_out e di calcolare e trasportare le variabili vive tramite il solo attributo *vive* lungo la catena di nonterminali L (che di fatto è una lista). Il programma essendo qui puramente sequenziale (giacché non ci sono né salti né cicli), basta un solo insieme di variabili vive per condurre il calcolo, non i due o più abituali (in ingresso e uscita). Infatti, avere gli insiemi separati di variabili in ingresso e uscita per ogni nodo del grafo del programma, serve essenzialmente quando il nodo ha due o più successori differenti, perché le variabili vive in uscita potrebbero differire secondo la diramazione da seguire (e allora andrebbero memorizzate in altrettanti insiemi di variabili vive in uscita), mentre quelle in ingresso costituiscono un solo insieme. Ma nel caso presente i nodi (le istruzioni) hanno sempre un solo successore. Bisogna però estendere gli attributi def e usa anche al nonterminale $\langle istr \rangle$.

(a) Ecco l'elenco degli attributi:

attributi da usare per la grammatica

tipo	nome	(non)terminali	dominio	significato
già dati nel testo dell'esercizio				
dx	n	id	stringa	nome dell'identificatore
da aggiungere o estendere				
sx	*var*	V	insieme di stringhe	lista di nomi di variabile
sx	*def*	D, $\langle istr \rangle$	idem	variabili definite da istr. corr.
sx	*usa*	U, $\langle istr \rangle$	idem	variabili usate da istr. corr.
sx	*vive*	L, S	idem	variabili vive del programma

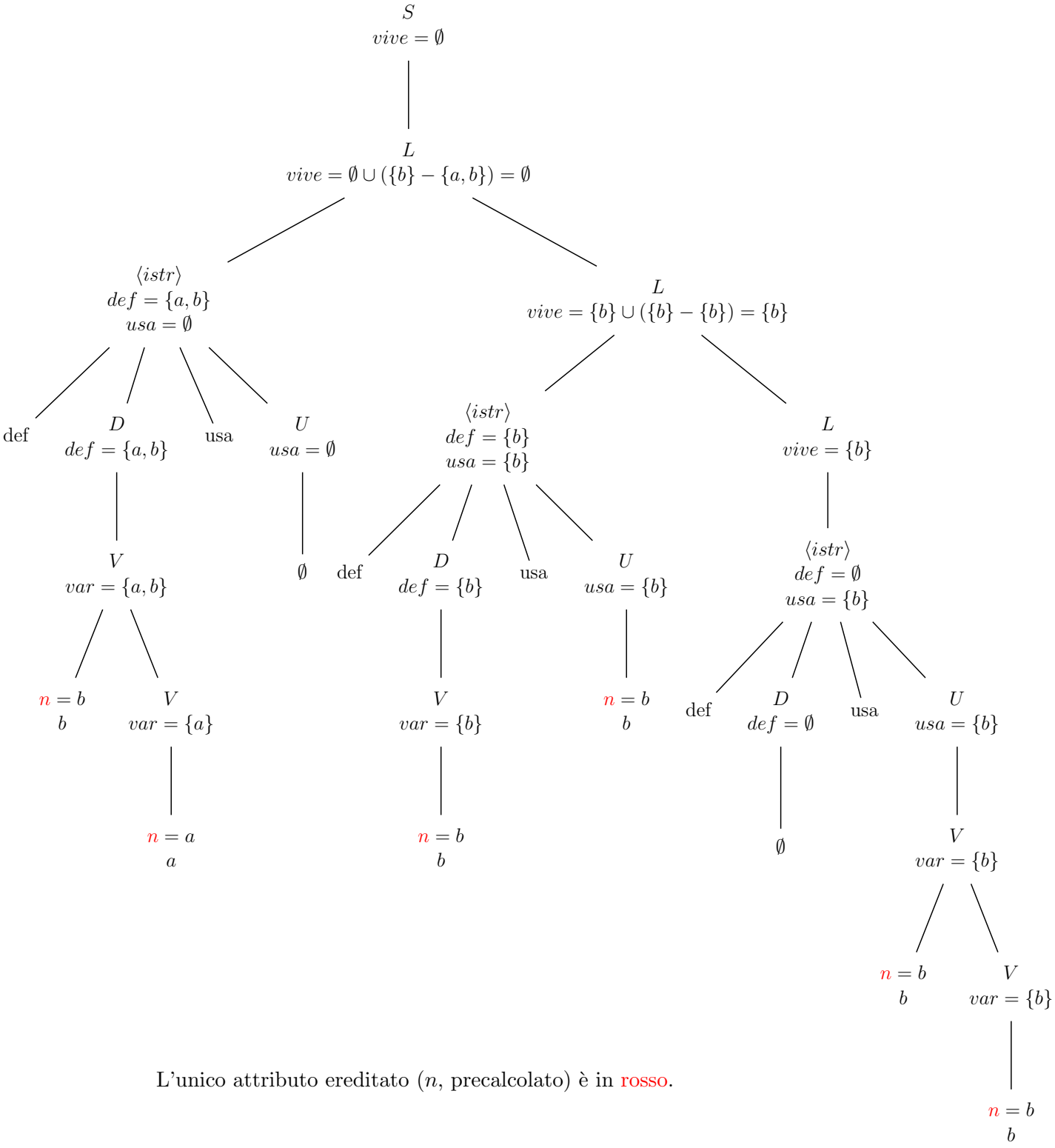
Come si vede tutti gli attributi significativi sono sintetizzati, tranne n che comunque è precalcolato e non reca alcun disturbo.

Ecco le funzioni semantiche:

sintassi	funzioni semantiche
$S_0 \rightarrow L_1$	$vive_0 = vive_1$
$L_0 \rightarrow \langle istr \rangle_1\ L_2$	$vive_0 = usa_1 \cup (vive_2 - def_1)$
$L_0 \rightarrow \langle istr \rangle_1$	$vive_0 = usa_1$
$\langle istr \rangle_0 \rightarrow$ 'def' D_1 'usa' U_2 ';'	$def_0 = def_1$ $usa_0 = usa_2$
$D_0 \rightarrow$ '$\emptyset$'	$def_0 = \emptyset$
$D_0 \rightarrow V_1$	$def_0 = var_1$
$U_0 \rightarrow$ '$\emptyset$'	$usa_0 = \emptyset$
$U_0 \rightarrow V_1$	$usa_0 = var_1$
$V_0 \rightarrow$ 'id' ',' V_1	$var_0 = \{n\} \cup var_1$
$V_0 \rightarrow$ 'id'	$var_0 = \{n\}$

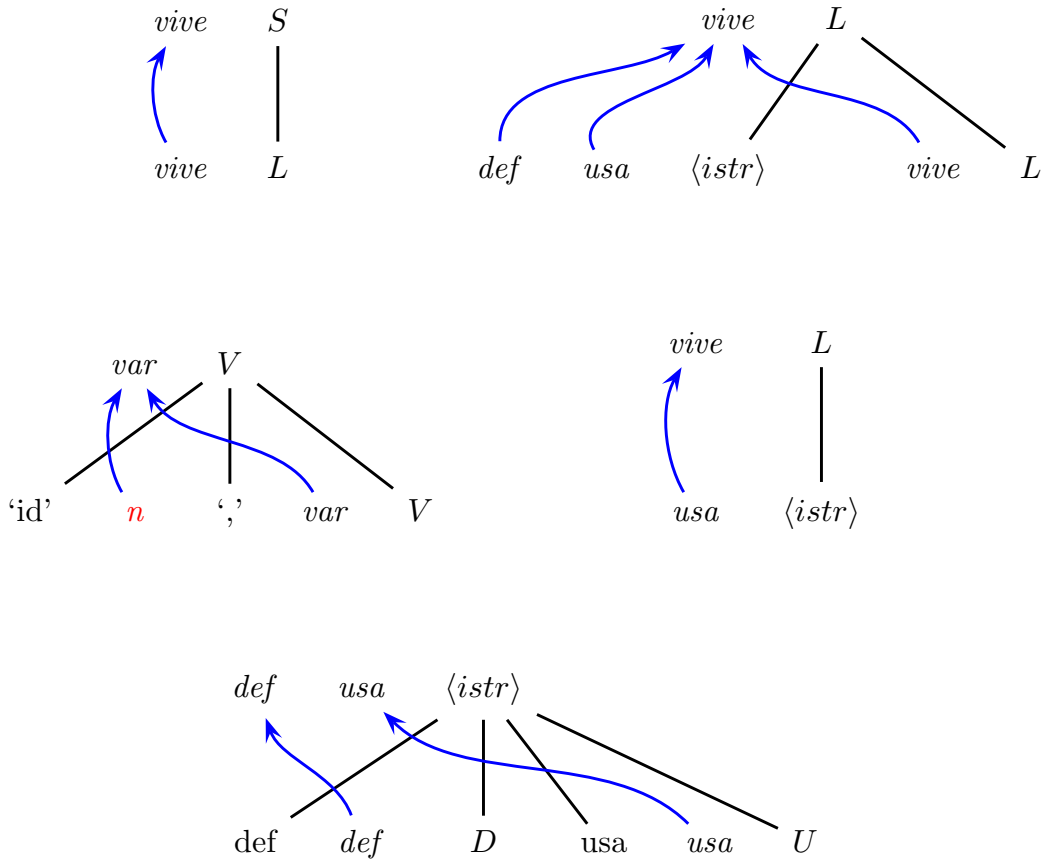
Il punto essenziale è che il calcolo delle variabili vive è concentrato a livello del nonterminale L che genera la lista di istruzioni, non a livello della singola istruzione ovvero del nonterminale $\langle istr \rangle$.

(b) Ecco l'albero sintattico decorato (e con il calcolo esplicitato), dove per ragioni di spazio gli attributi (tutti sintetizzati) sono elencati sotto i nodi, invece che a sinistra come si fa d'abitudine:



L'unico attributo ereditato (n, precalcolato) è in rosso.

(c) Ecco i grafi delle dipendenze elementari delle regole della grammatica:



Le regole rimanenti sono banali e qui non sono riportate. Naturalmente ci sono solo frecce verso l'alto perché tutti gli attributi sono sintetizzati (tranne n).

Giacché si serve solo di attributi sintetizzati, la grammatica è certamente di tipo L (ciò implica anche sia di tipo one-sweep, beninteso).

Ecco una procedura semantica significativa (quella del nonterminale L, che contiene il calcolo effettivo delle variabili vive):

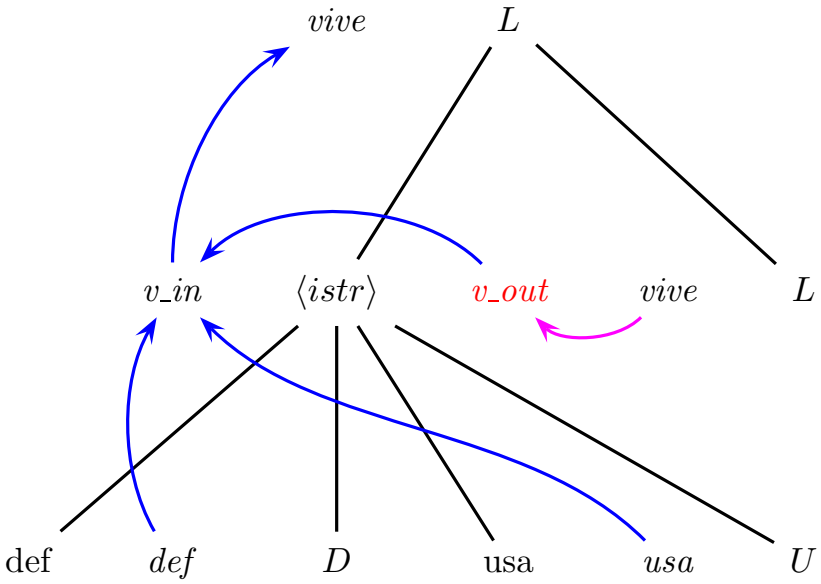
procedure L (*vive*: **out**; *albero*: **in**)
– variabili locali per passaggio attributi
    var *def_tmp*, *usa_tmp*, *vive_tmp*
– alternativa "$L \rightarrow \langle istr \rangle\ L$"
    call ISTR (*def_tmp*, *usa_tmp*, $albero \rightarrow \langle istr \rangle$)
    call L (*vive_tmp*, $albero \rightarrow L$)
    $vive = usa\_tmp \cup (vive\_tmp - def\_tmp)$
– alternativa "$L \rightarrow \langle istr \rangle$"
    call ISTR (*def_tmp*, *usa_tmp*, $albero \rightarrow \langle istr \rangle$)
    $vive = usa\_tmp$
end L

Naturalmente l'ordine di chiamata delle procedure dei nodi figli del nonterminale L è quello da sinistra verso destra. È utile confrontare questa versione della procedura, con la stessa nella soluzione estesa al problema, data prima.

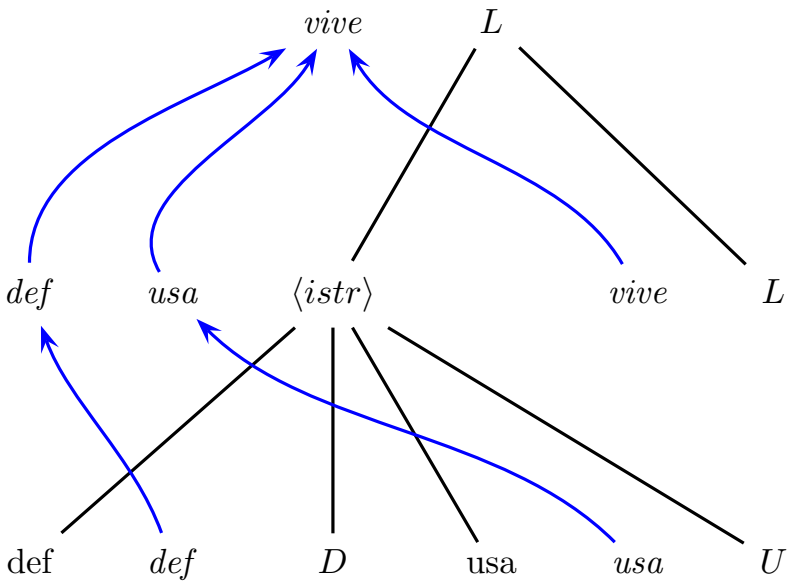
Le procedure rimanenti sono lasciate al lettore (ma questa basta). Ovviamente le procedure hanno solo parametri in uscita, corrispondenti agli attributi sintetizzati (tranne l'argomento *albero*, puntatore al sottoalbero da analizzare, che c'è sempre e ovviamente è in ingresso).

Addendum Come si vede, la soluzione è semplicissima ma del tutto valida. Per trovarla bisogna naturalmente avere assimilato bene l'essenza del calcolo delle variabili vive, per poi modellarlo tramite un calcolo di attributi quanto più lineare possibile. Al riguardo, si confronti la propagazione degli attributi che ha luogo combinando le regole che espandono L e $\langle istr \rangle$ nella soluzione estesa piuttosto che in quella semplificata, come è riportata di seguito:

nella soluzione estesa



nella soluzione semplificata



La rappresentazione grafica rende molto evidente come la semplificazione consista nell'evitare il passaggio intermedio delle variabili vive mediato dagli argomenti *v_out* e *v_in*, rispettivamente ereditato e sintetizzato, i quali vengono entrambi eliminati (mentre *def* e *usa* vanno estesi anche a ⟨*istr*⟩).

Peraltro, la soluzione estesa si presta meglio a generalizzazioni dove l'istruzione abbia due o più vie d'uscita, nei quali casi occorrerebbe avere due o più insiemi separati di variabili vive in uscita (uno per ciascuna via), come si è detto prima.