Linguaggi Formali e Compilatori
(Formal Languages and Compilers)

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

Prova scritta - 7 febbraio 2007 - Parte I: Teoria

CON SOLUZIONI (sono molto estese a scopo didattico, e non si chiede che il candidato risponda con altrettanta ampiezza, bensì a suo giudizio in modo ragionevolmente sufficiente)

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. Sono date le due espressioni regolari seguenti, di alfabeto $\Sigma = \{a, b\}$:

$$R_1 = a^*b^*a^* \qquad R_2 = (ab \mid bb \mid a)^*$$

Esse generano i linguaggi regolari L_1 e L_2, rispettivamente. Si svolgano i punti seguenti:

(a) Si elenchino tutte le stringhe di lunghezza ≤ 4 appartenenti ai linguaggi definiti dalle differenze insiemistiche $L_1 \setminus L_2$ e $L_2 \setminus L_1$ (si scriva nello spazio sotto).

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

(b) Si scriva l'espressione regolare R_3 del linguaggio $L_3 = \neg L_1$ (il complemento di L_1), usando soltanto gli operatori di unione, concatenamento, stella e croce.

Soluzione

(a) Per calcolare rapidamente le differenze insiemistiche si può procedere in modo intuitivo, osservando quanto segue:

- in R_1 le (eventuali) lettere b sono generate solo consecutivamente e possono essere in numero qualunque (anche nessuna o una sola)
- in R_2 si possono (eventualmente) avere lettere b separate da almeno una lettera a oppure un solo blocco di lettere b consecutive con lunghezza qualunque, ma mai dispari se esso si trova all'inizio della frase

Ne viene che le differenze insiemistiche si caratterizzano nel modo seguente:

- le frasi di lunghezza ≤ 4 contenute in L_1 ma non in L_2 sono tutte e sole quelle (di L_1) che iniziano con una sola lettera b o con tre lettere b
- le frasi di lunghezza ≤ 4 contenute in L_2 ma non in L_1 sono tutte e sole quelle (di L_2) che contengono lettere b non consecutive

Di conseguenza, ecco le stringhe appartenenti a L_1 ma non a L_2 (a sinistra), e a L_2 ma non a L_1 (a destra):

$L_1 \setminus L_2$	$L_2 \setminus L_1$
b	$a\,b\,a\,b$
$b\,a$	$b\,b\,a\,b$
$b\,a\,a$	
$b\,b\,b$	
$b\,a\,a\,a$	
$b\,b\,b\,a$	

Le frasi sono ordinate lessicograficamente, con $a < b$. Si vede subito che le frasi soddisfano precisamente alle due caratterizzazioni intuitive enunciate sopra.

Addendum In alternativa, si possono ottenere le differenze insiemistiche in modo puramente algoritmico, elencando prima tutte le frasi di lunghezza ≤ 4 dei linguaggi L_1 e L_2, e poi calcolando i due insiemi differenza. Si lascia tale compito (un po' noioso forse, ma non lungo e del tutto banale) al lettore.

(b) Come già detto prima (punto (a)), le frasi del linguaggio L_1 sono caratterizzate dall'assenza di due b non adiacenti, cioè più precisamente le frasi di L_1:

- o non contengono nessuna lettera b
- o ne contengono una sola
- o ne contengono due o più ma tutte consecutive, non separate da a

Ne viene che le frasi del linguaggio $L_3 = \neg L_1$, il complemento di L_1, sono caratterizzate dal contenere almeno due b separate da almeno una a, come segue:

$$L_3 = \{\ x \in \Sigma^* \ \mid \quad x \text{ contiene almeno due } b \text{ non adiacenti } \}$$

Pertanto per il linguaggio L_3 si ha l'espressione regolare R_3 seguente:

$$R_3 = (a \mid b)^*\ b\ a^+\ b\ (a \mid b)^*$$

o, se si preferisce, $R_3 = \Sigma^*\ b\ a^+\ b\ \Sigma^*$, dove si mette in evidenza il ruolo del linguaggio universale Σ^* come componente.

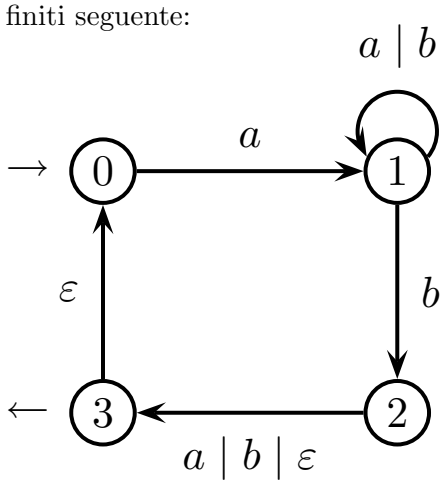
Addendum Si noti che l'espressione regolare R_3 data è fortemente indeterministica (anzi addirittura ambigua), e come tale molto sintetica ed efficace per descrivere L_3 senza inutili dettagli. Volendone una non ambigua, eccola: $R_3 = a^*\ b^+\ a^+\ b\ \Sigma^*$; dove si precisa che la coppia di b separate generata dal componente b^+a^+b è la *più a sinistra*, le altre sono generate da Σ^*.

Naturalmente si può ricavare per via algoritmica, senza ricorrere all'intuizione, un'espressione regolare che generi il linguaggio $L_3 = \neg L_1$, come segue:

- costruisci l'automa deterministico equivalente all'espressione R_1 (con la costruzione di McNaughton-Yamada, oppure con quella di Thompson seguita da determinizzazione con quella dei sottinsiemi)
- costruisci l'automa complemento (che risulta anch'esso deterministico)
- ricava l'espressione R_3 equivalente all'automa complemento (con la costruzione di Brozozowsky o con l'algoritmo delle equazioni insiemistiche lineari)

In tale modo non occorre nessuna intuizione, ma il percorso è presumibilmente più lungo. Si lascia l'esercizio al lettore.

2. È dato l'automa a stati finiti seguente:



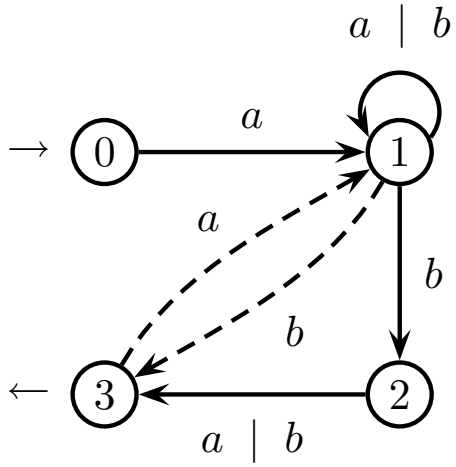
Si svolgano i punti seguenti:

(a) Si costruisca l'automa deterministico equivalente, usando un metodo a scelta.

(b) Si minimizzi il numero di stati dell'automa deterministico ricavato al punto precedente.

Soluzione

(a) Si osservi intanto che l'automa è pulito (ogni stato è sia raggiungibile sia definito). Per determinizzarlo bisogna in primo luogo togliere le due transizioni spontanee. Conviene prima eliminare l'arco $3 \xrightarrow{\varepsilon} 0$ arretrando da 0 a 3 l'arco a uscente da 0, e poi eliminare l'arco $2 \xrightarrow{\varepsilon} 3$ avanzando da 2 a 3 l'arco b entrante in 2 (senza togliere gli archi a e b originali, beninteso); qui tali scelte minimizzano il numero di archi da replicare ed evitano di creare nuovi stati iniziali e finali. Così procedendo si ottiene l'automa seguente, ancora indeterministico:

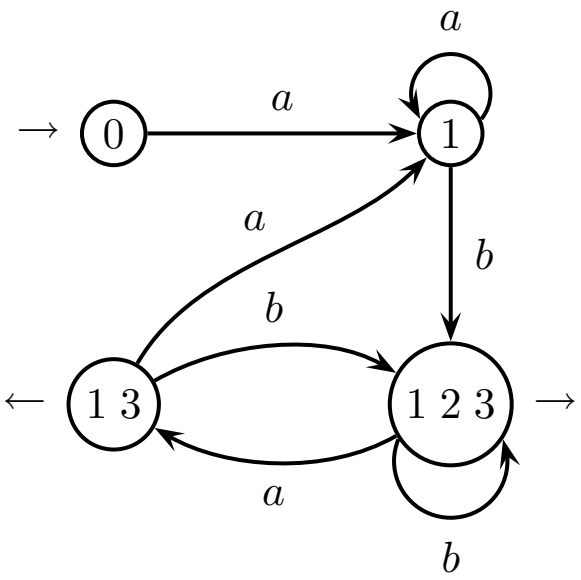


I due archi a e b tratteggiati sono quelli aggiuntivi (repliche) e ricostruiscono i percorsi etichettati andati persi togliendo le due transizioni spontanee.

Poi si può ricavare l'automa deterministico equivalente tramite la costruzione dei sottinsiemi (realizzata tramite l'algoritmo della tabella dei successori):

gruppo	a	b	**finale ?**
0	1	$\emptyset$	no
1	1	123	no
123	13	123	sì
13	1	123	sì
$\emptyset$	$\emptyset$	$\emptyset$	no

La costruzione porta all'automa deterministico seguente, ancora a 4 stati (dove lo stato di errore $\emptyset$ è omesso, come d'uso):



L'automa deterministico così costruito è pulito (tutti gli stati sono sia definiti sia raggiungibili), giacché l'algoritmo della tabella dei successori produce un automa pulito se l'automa (nondeterministico) di partenza lo era, come è qui il caso.

(b) Dato che uno stato finale è sempre distinguibile da uno non finale, per verificare se l'automa precedente sia minimo basta esaminarne le coppie di stati entrambi finali oppure entrambi non finali. Si ha quanto segue:

- gli stati 0 e 1 sono distinguibili, perché da 1, ma non da 0, si esce con b
- gli stati [1 2 3] e [1 3] sono distinguibili, perché da [1 2 3] con a si va in uno stato finale, mentre da [1 3] con a si va in uno stato non finale

Pertanto l'automa deterministico è già minimo e, da prima, è pulito. Non è dunque possibile alcuna riduzione del numero di stati a meno di 4.

Addendum Se si rinuncia al determinismo, ci potrebbe essere un automa equivalente nondeterministico con meno di 4 stati ... il lettore ne può cercare uno da sé intuitivamente (ma non è detto esista).

2 Grammatiche libere e automi a pila 20%

1. Si consideri il linguaggio di Dyck di alfabeto $\Sigma = \{a, b, c, d\}$, dove a, c sono parentesi aperte e b, d sono le parentesi chiuse corrispondenti, rispettivamente.

   Si svolgano i punti seguenti:

   (a) Si scrivano le grammatiche G_1 e G_2 dei linguaggi L_1 e L_2, rispettivamente, entrambe in forma BNF (non estesa). I linguaggi L_1 e L_2 sono definiti nel modo seguente:

   - L_1 = linguaggio di Dyck con profondità massima di annidamento pari a 2

     Esempi: $\varepsilon \quad a\,b \quad c\,d \quad a\,b\,c\,d \quad a\,c\,d\,b \quad a\,a\,b\,c\,d\,b$

     Controesempi: $a\,a\,c\,d\,b\,b \quad a\,a\,b\,c\,a\,b\,d\,b$

   - L_2 = linguaggio di Dyck che ammette solo strutture annidate di parentesi (con profondità arbitraria di annidamento), ma non strutture concatenate

     Esempi: $\varepsilon \quad a\,b \quad c\,d \quad a\,c\,d\,b \quad a\,a\,c\,d\,b\,b$

     Controesempi: $a\,a\,b\,c\,d\,b \quad c\,d\,a\,b$

   (b) Si scriva la grammatica G del linguaggio L, non ambigua e in forma BNF (non estesa), con L definito nel modo seguente:

   $$L = L_1 \cup L_2$$

   ovvero l'unione di L_1 e L_2.

Soluzione

(a) Ecco le grammatiche G_1 e G_2 che generano L_1 e L_2, rispettivamente:

$$G_1 = \begin{cases} S \;\rightarrow\; a\,X\,b\,S \;\mid\; c\,X\,d\,S \;\mid\; \varepsilon \\ X \;\rightarrow\; a\,b\,X \;\mid\; c\,d\,X \;\mid\; \varepsilon \end{cases}$$

$$G_2 = \begin{cases} S \;\rightarrow\; a\,S\,b \;\mid\; c\,S\,d \;\mid\; \varepsilon \end{cases}$$

Sono due grammatiche ricavate da quella di Dyck standard (ben nota):

$$S \rightarrow a\,S\,b\,S \;\mid\; c\,S\,d\,S \;\mid\; \varepsilon$$

applicandole restrizioni per escludere certi tipi di derivazione, come segue:

- la grammatica G_1 non può generare strutture parentetiche con profondità di annidamento maggiore di 2, dato che il nonterminale X non genera strutture annidate, ma solo concatenate
- la grammatica G_2 non può concatenare strutture parentetiche, ma solo annidarle, dato che manca il nonterminale S concatenato in fondo alle regole

In quanto ottenute per restrizione dalla grammatica di Dyck, G_1 e G_2 sono entrambe non ambigue. La rispettiva verifica di correttezza è immediata ed è lasciata al lettore (si provi con gli esempi e controesempi dati prima).

(b) L'unione di L_1 e L_2 è ambigua, perché genera in modo ambiguo i nidi semplici di parentesi (cioè quelli senza strutture parentetiche concatenate) con profondità massima 2. Le frasi ambigue sono le seguenti:

$$\varepsilon \qquad a\,b \qquad c\,d \qquad a\,a\,b\,b \qquad a\,c\,d\,b \qquad c\,a\,b\,d \qquad c\,c\,d\,d$$

Per unire L_1 e L_2 in modo non ambiguo, bisogna eliminare la doppia generazione delle frasi ambigue elencate sopra (sono in numero finito). Ecco la grammatica G non ambigua del linguaggio unione L (assioma S):

$$G = (V_N = \{S, S_1, X, S_2, Y_1, Y_2, Y_{\geq 3}\}, V_T = \{a, b, c, d\}, S, P)$$

$$P = \begin{cases} S \;\rightarrow\; S_1 \mid S_2 \\ \hline G_1 \;=\; \begin{cases} S_1 \;\rightarrow\; a\,X\,b\,S_1 \mid c\,X\,d\,S_1 \mid \varepsilon \\ X \;\rightarrow\; a\,b\,X \mid c\,d\,X \mid \varepsilon \end{cases} \\ \hline G_2 \underset{\text{modificata}}{\;=\;} \begin{cases} S_2 \;\rightarrow\; a\,Y_1\,b \mid c\,Y_1\,d \\ Y_1 \;\rightarrow\; a\,Y_2\,b \mid c\,Y_2\,d \\ Y_2 \;\rightarrow\; a\,Y_{\geq 3}\,b \mid c\,Y_{\geq 3}\,d \\ Y_{\geq 3} \;\rightarrow\; a\,Y_{\geq 3}\,b \mid c\,Y_{\geq 3}\,d \mid \varepsilon \end{cases} \end{cases}$$

Si tratta in sostanza dell'unione di G_1 e G_2, fatta tramite la costruzione modulare standard, che sarebbe la seguente:

- disgiungi gli insiemi di nonterminali (dove già non siano disgiunti)
- unisci gli assiomi precedenti S_1 e S_2 sotto un nuovo assioma S

Tuttavia qui la componente G_2 è modificata leggermente sì da imporle di generare strutture parentetiche del tutto simili a quelle prodotte dalla G_2 originale, ma con profondità di annidamento pari ad almeno 3. In tale modo la componente G_2 modificata evita di riprodurre le frasi già generate dalla componente G_1 e l'ambiguità di unione non si manifesta.

Per giustificare tale comportamento, basta osservare che nella componente G_2 modificata la derivazione non può terminare prima di avere raggiunto il nonterminale $Y_{\geq 3}$, dal nome volutamente espressivo, così inevitabilmente annidando almeno tre strutture parentetiche (di tipo non concatenato). La verifica è semplice ed è lasciata al lettore (si provi con le frasi ambigue elencate sopra).

Addendum Beninteso si sarebbe potuto scegliere di modificare la componente G_1, invece della G_2, forse però con un po' meno facilità. Il lettore si può esercitare da sé nel progetto di questa seconda possibile soluzione.

2. Si consideri il linguaggio di interrogazione SQL, semplificato, con i concetti seguenti:

   - Ci sono nomi di attributo e nomi di relazione (nessuno dei due va definito qui). L'attributo si può denotare tramite nome semplice, o articolato mediante "`.`":

     ```
     nome-di-relazione.nome-di-attributo
     ```

   - Ci sono espressioni logiche con parentesi "`(`" e "`)`", costituite da operatori logici `and` e `not` e da predicati relazionali; `not` ha precedenza su `and`.
   - Ci sono predicati relazionali, costituiti da operatore di confronto uguale o minore - cioè "`==`" o "`<`" - e da due operandi, nel modo seguente:

     ```
     termine1  op-rel  termine2
     ```

     L'operando `termine1` è solo attributo, mentre l'operando `termine2` può essere attributo o valore numerico costante (qui non va definito). Gli operatori relazionali hanno precedenza su quelli logici.
   - Il blocco di interrogazione SQL contiene tutte le clausole seguenti (in ordine):
     - `select`, seguita da:
       - o lista dei nomi di attributo con separatore "`,`"
       - o terminale "`*`" per indicare tutti gli attributi
     - `from`, seguita da lista di nomi di relazione con separatore "`,`"
     - `where`, seguita da espressione logica

     Il blocco può concludersi con clausola `order-by`, seguita da uno degli attributi elencati in `select` da ordinare in modo ascendente o discendente.
   - L'espressione di interrogazione SQL si compone di un solo blocco di interrogazione, oppure di due legati mediante operatore insiemistico `union` o `intersect`.

   Esempi:

   ```
   SELECT   CCB.NOME, MOVIMENTI.IMPORTO, MOVIMENTI.CAUSALE
   FROM     CCB, MOVIMENTI
   WHERE    CCB.NUMCCB == MOVIMENTI.NUMCCB AND NOT (MOVIMENTI.IMPORTO < 1000)
       UNION
   SELECT   CCB.NOME, MOVIMENTI.IMPORTO, MOVIMENTO.CAUSALE
   FROM     CCB, MOVIMENTI
   WHERE    CCB.NUMCCB == MOVIMENTI.NUMCCB AND CCB.NUMCCB < 1356789
   ```

   e anche

   ```
   SELECT   *
   FROM     CCB
   WHERE    SALDO < 200000
   ORDER-BY CCB.NUMCCB ASC
   ```

   Si svolgano i punti seguenti:

   (a) Si progetti la grammatica EBNF (estesa) non ambigua che modella il linguaggio delle espressioni di interrogazione SQL descritto sopra.

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

Soluzione

(a) Ecco una grammatica EBNF dell'espressione di interrogazione SQL, dove l'assioma è ⟨QUERY_EXP⟩ e le parentesi quadre "`[`" e "`]`" indicano opzionalità:

⟨QUERY_EXP⟩	→	⟨QUERY_BLOCK⟩ [⟨SET_OPERATOR⟩ ⟨QUERY_BLOCK⟩]
⟨SET_OPERATOR⟩	→	'`UNION`' | '`INTERSECT`'
⟨QUERY_BLOCK⟩	→	⟨SELECT_CLAUSE⟩ ⟨FROM_CLAUSE⟩ ⟨WHERE_CLAUSE⟩ [⟨ORDER-BY_CLAUSE⟩]
⟨SELECT_CLAUSE⟩	→	'`SELECT`' (⟨ATTRIBUTE_LIST⟩ | '`*`')
⟨FROM_CLAUSE⟩	→	'`FROM`' ⟨RELATION_LIST⟩
⟨WHERE_CLAUSE⟩	→	'`WHERE`' ⟨LOGICAL_EXP⟩
⟨ORDER-BY_CLAUSE⟩	→	'`ORDER-BY`' ⟨ATTR_PATHNAME⟩ ['`ASC`' | '`DISC`']
⟨ATTRIBUTE_LIST⟩	→	⟨ATTR_PATHNAME⟩ ('`,`' ⟨ATTR_PATHNAME⟩)*
⟨ATTR_PATHNAME⟩	→	[⟨RELATION_NAME⟩ '`.`'] ⟨ATTRIBUTE_NAME⟩
⟨ATTRIBUTE_NAME⟩	→	... (identificatore generico)
⟨RELATION_LIST⟩	→	⟨RELATION_NAME⟩ ('`,`' ⟨RELATION_NAME⟩)*
⟨RELATION_NAME⟩	→	... (identificatore generico)
⟨LOGICAL_EXP⟩	→	⟨LOGICAL_EXP_1⟩ ('`AND`' ⟨LOGICAL_EXP_1⟩)*
⟨LOGICAL_EXP_1⟩	→	['`NOT`'] ⟨LOGICAL_EXP_2⟩
⟨LOGICAL_EXP_2⟩	→	'`T`' | '`F`' | ⟨REL_PREDICATE⟩ | '`(`' ⟨LOGICAL_EXP⟩ '`)`'
⟨REL_PREDICATE⟩	→	⟨TERM_1⟩ ⟨REL_OPERATOR⟩ ⟨TERM_2⟩
⟨TERM_1⟩	→	⟨ATTR_PATHNAME⟩
⟨TERM_2⟩	→	⟨ATTR_PATHNAME⟩ | ⟨CONST_VALUE⟩
⟨REL_OPERATOR⟩	→	'`==`' | '`<`'
⟨CONST_VALUE⟩	→	... (numero generico)

L'espressione logica è modellata a tre livelli: AND (precedenza minima), NOT (precedenza media) e predicato relazionale (precedenza massima). L'intera grammatica ha struttura fortemente modulare, come la suddivisione in gruppi di regole mette in evidenza. Tale suddivisione ordinata, la scelta meditata di nomi espressivi per i nonterminali e la corrispondenza ben visibile tra regole e specifiche, giustificano sufficientemente la correttezza della soluzione.

(b) Chiaramente non si possono modellare gli aspetti semantici seguenti:

- la concordanza tra dichiarazione e uso di nome di attributo o di relazione
- la compatibilità dei tipi degli attributi con gli operatori di confronto uguale e minore (dovrebbero essere solo attributi a valore numerico)
- la compatibilità dei tipi delle relazioni tra blocchi di interrogazione uniti o intersecati in senso insiemistico (i blocchi dovrebbero avere relazioni comuni)
- che l'attributo di `order-by` figuri tra quelli di `select`

ed eventualmente altri aspetti, più o meno specifici, che il lettore può ricercare e precisare da sé (se ce ne sono).

3 Analisi sintattica e parsificatori 20%

1. È data la grammatica G seguente (con assioma S):

$$S \;\rightarrow\; (\; E \text{ ‘=’ } E \;\mid\; \text{‘}i\text{’ ‘=’ } E \;)^{+}$$
$$E \;\rightarrow\; \text{‘}i\text{’ } (\; \text{‘+’ ‘}i\text{’ })^{*}$$

Si svolgano i punti seguenti:

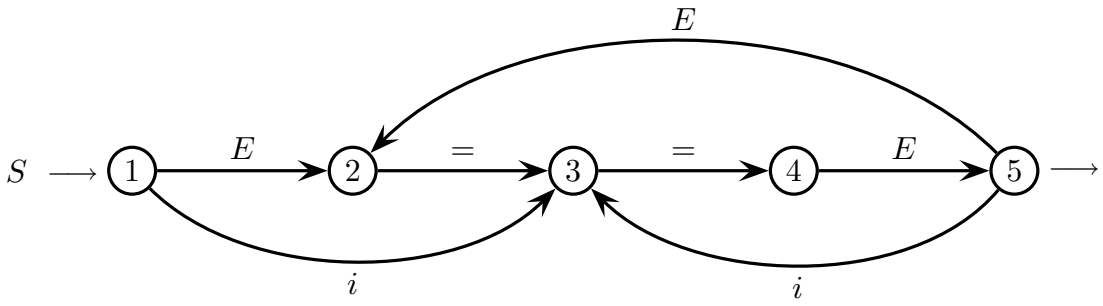
(a) Si rappresenti la grammatica G come rete di macchine ricorsive a stati finiti.

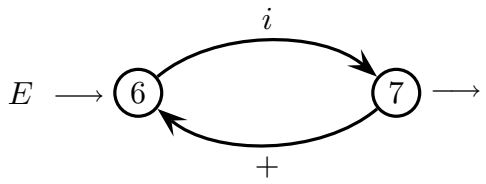
(b) Si verifichi se la grammatica, in forma di rete, sia di tipo $LL(k)$, per un $k \geq 1$.

(c) Se è necessario, si modifichi la grammatica in modo da renderla $LL(k)$, per $k = 1$.

Soluzione

(a) Ecco la rappresentazione della grammatica G come rete di automi ricorsivi a stati finiti, del tutto equivalente a G:

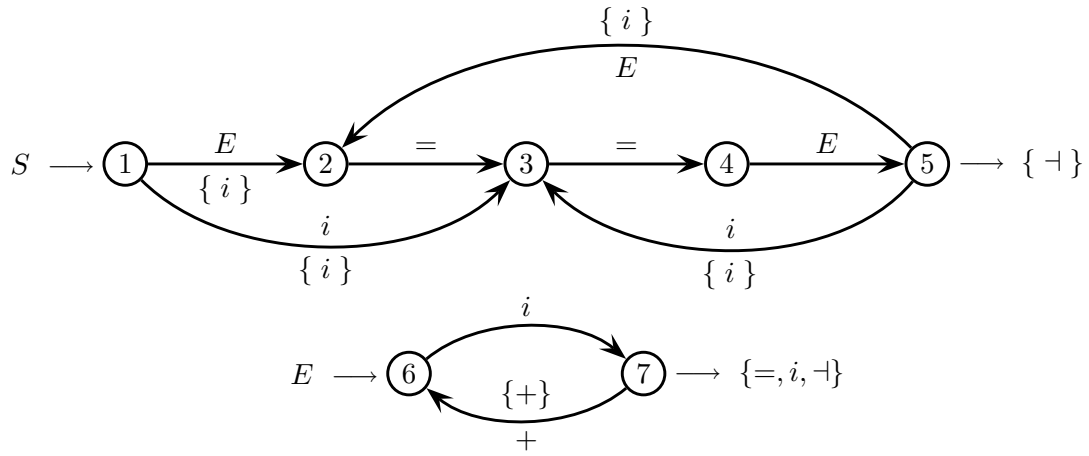




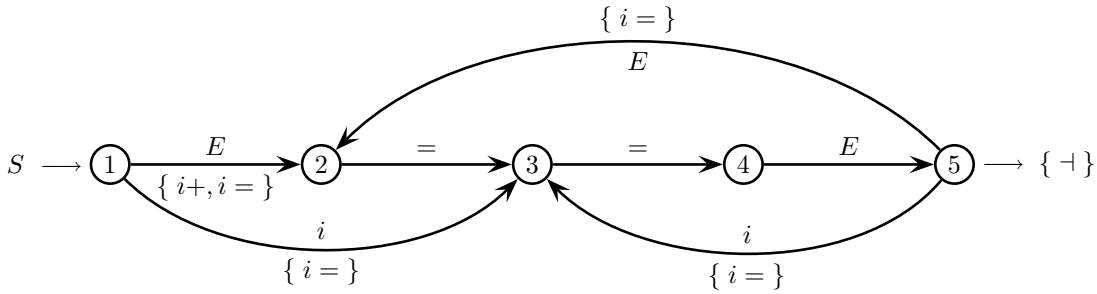
Entrambi gli automi sono puliti e in forma deterministica, come da standard, e inoltre sono in forma minima (quest'ultima proprietà non è necessaria, ma naturalmente fa comodo). Il secondo automa non è neppure ricorsivo, perché non ha archi etichettati con nonterminali.

Addendum Volendo, si potrebbe dare anche una rappresentazione con automi (ricorsivi) nondeterministici, ma non sembra che qui aiuterebbe a risparmiare stati, mentre di certo ostacolerebbe l'analisi LL.

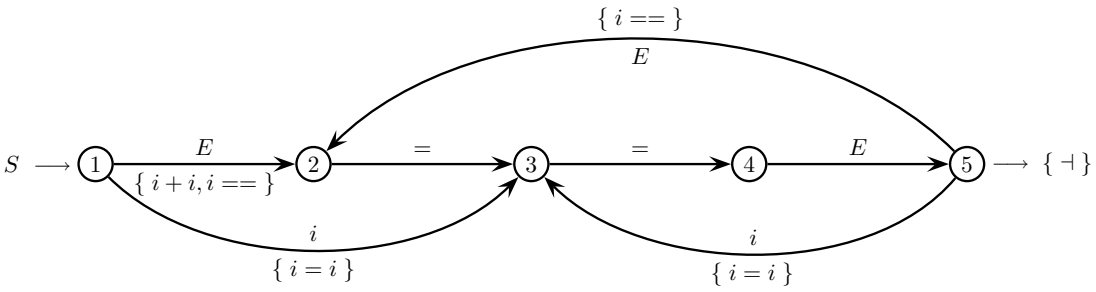
(b) Verifica della proprietà $LL(1)$ (per entrambi gli automi):



Gli stati 1 e 5 violano la condizione LL: G non è $LL(1)$.
Verifica della proprietà $LL(2)$ (solo per l'automa che espande S):

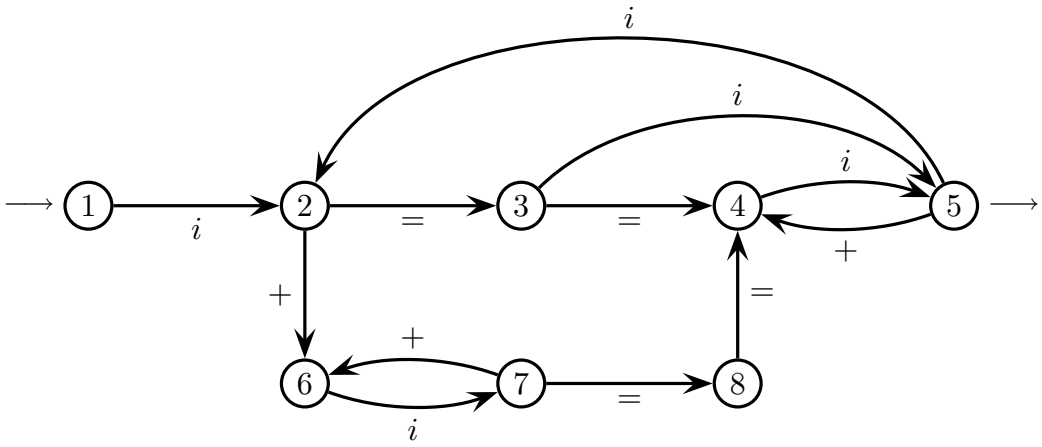


Gli stati 1 e 5 violano ancora la condizione LL: G non è $LL(2)$.
Verifica della proprietà $LL(3)$ (solo per l'automa che espande S):



Nessuno stato viola più la condizione LL: G è $LL(3)$.

(c) Una frase di $L(G)$ è una lista non vuota (e senza separatore) di relazioni, ciascuna con uno o due caratteri "=" ed espressioni di addizione (di variabili i) a destra e sinistra, ma con il vincolo che a sinistra di una relazione con un solo "=" ci possa essere solo i, non un'espressione con addizione. Il linguaggio $L(G)$ è dunque regolare e pertanto se ne può trovare una grammatica $LL(1)$.
Per esempio, basta tracciare l'automa deterministico a stati finiti del linguaggio $L(G)$ e la grammatica lineare a destra corrispondente sarà senz'altro $LL(1)$, come ben noto. Ecco l'automa che riconosce $L(G)$, qui ricavato in modo intuitivo (è pulito e in forma minima, come è facile verificare):



Ed ecco la grammatica $LL(1)$ lineare a destra che genera $L(G)$ (assioma $\langle 1 \rangle$):

$$
\begin{cases}
\langle 1 \rangle & \rightarrow & \text{`}i\text{'}\ \langle 2 \rangle \\
\langle 2 \rangle & \rightarrow & \text{`}=\text{'}\ \langle 3 \rangle \ \mid\ \text{`}+\text{'}\ \langle 6 \rangle \\
\langle 3 \rangle & \rightarrow & \text{`}=\text{'}\ \langle 4 \rangle \ \mid\ \text{`}i\text{'}\ \langle 5 \rangle \\
\langle 4 \rangle & \rightarrow & \text{`}i\text{'}\ \langle 5 \rangle \\
\langle 5 \rangle & \rightarrow & \varepsilon \ \mid\ \text{`}i\text{'}\ \langle 2 \rangle \ \mid\ \text{`}+\text{'}\ \langle 4 \rangle \\
\langle 6 \rangle & \rightarrow & \text{`}i\text{'}\ \langle 7 \rangle \\
\langle 7 \rangle & \rightarrow & \text{`}+\text{'}\ \langle 6 \rangle \ \mid\ \text{`}=\text{'}\ \langle 8 \rangle \\
\langle 8 \rangle & \rightarrow & \text{`}=\text{'}\ \langle 4 \rangle
\end{cases}
$$

Evidentemente, tutti gli insiemi guida delle regole alternative sono disgiunti (quello della regola nulla contiene il terminatore, naturalmente).

Addendum Dato che nella rappresentazione di G come rete di due automi, uno dei due non è ricorsivo, si sarebbe potuto ricavare per via algoritmica l'automa a stati finiti che riconosce $L(G)$, nel modo seguente:

- prendi l'automa ricorsivo di S, espandine gli archi con etichetta E mediante l'automa non ricorsivo di E e ottieni così un automa ordinario, non ricorsivo e (in generale) nondeterministico
- determinizza l'automa tramite la costruzione dei sottinsiemi ed eventualmente minimizzane il numero di stati

Il risultato è l'automa deterministico (ed eventualmente minimo) di $L(G)$. Non è detto risulti identico a quello dato sopra (a meno che non sia anch'esso minimo), benché naturalmente debba essergli equivalente.

In alternativa si può dare direttamente (in modo intuitivo) una grammatica $LL(1)$ di tipo EBNF. Per esempio:

$$\begin{cases} S \;\rightarrow\; (\; \text{`}i\text{'}\; (\; \text{`}=\text{'}\; [\; \text{`}=\text{'}\; ]\;\mid\; \text{`}+\text{'}\; E\; \text{`}=\text{'}\; \text{`}=\text{'})\; E\;)^{+} \\ E \;\rightarrow\; \text{`}i\text{'}\; (\; \text{`}+\text{'}\; \text{`}i\text{'}\;)^{*} \end{cases}$$

Si constata facilmente che tale grammatica è $LL(1)$: basta rimodellarla come rete di automi e verificare. Del resto uno sguardo rapido mostra come essa corrisponda direttamente all'automa di $L(G)$ dato prima. Oppure si provi a codificare l'analizzatore sintattico $LL(1)$ della grammatica, scrivendo le procedure sintattiche di S ed E. L'esercizio di verifica di correttezza della grammatica (nell'uno o l'altro modo) è lasciato al lettore.

2. È data la grammatica G seguente (con assioma S):

$$
\begin{aligned}
S &\rightarrow E \text{ `}\dashv\text{'} \\
E &\rightarrow T \text{ `}+\text{'} E \mid T \\
T &\rightarrow \text{`}i\text{'} \mid \text{`}i\text{'} \text{ `}+\text{'} T \mid \text{`(' } E \text{ `)'}
\end{aligned}
$$

Si svolgano i punti seguenti:

(a) Si stabilisca, nel modo opportuno, se la grammatica G sia di tipo $LR(1)$.

(b) Se è necessario, si modifichi la grammatica G in modo da ottenere una nuova grammatica G' equivalente a G, ma che sia $LR(1)$.

(c) (facoltativo) Si discuta in breve se per il linguaggio $L(G)$ si possa dare una grammatica di tipo $LR(0)$.

Soluzione

(a) La risposta si trova in modo pressoché immediato: la grammatica G non è $LR(1)$, perché è ambigua. Infatti le frasi del linguaggio di G sono del tipo seguente, con parentesi annidate:

$$i \ + \ \ldots \ + \ (\ i \ + \ \ldots \ + \ (\ i \ + \ \ldots \ ) \ + \ \ldots \ ) \ + \ \ldots \ \dashv$$

e si possono avere più coppie di parentesi concatenate (a qualunque livello di profondità), tipo:

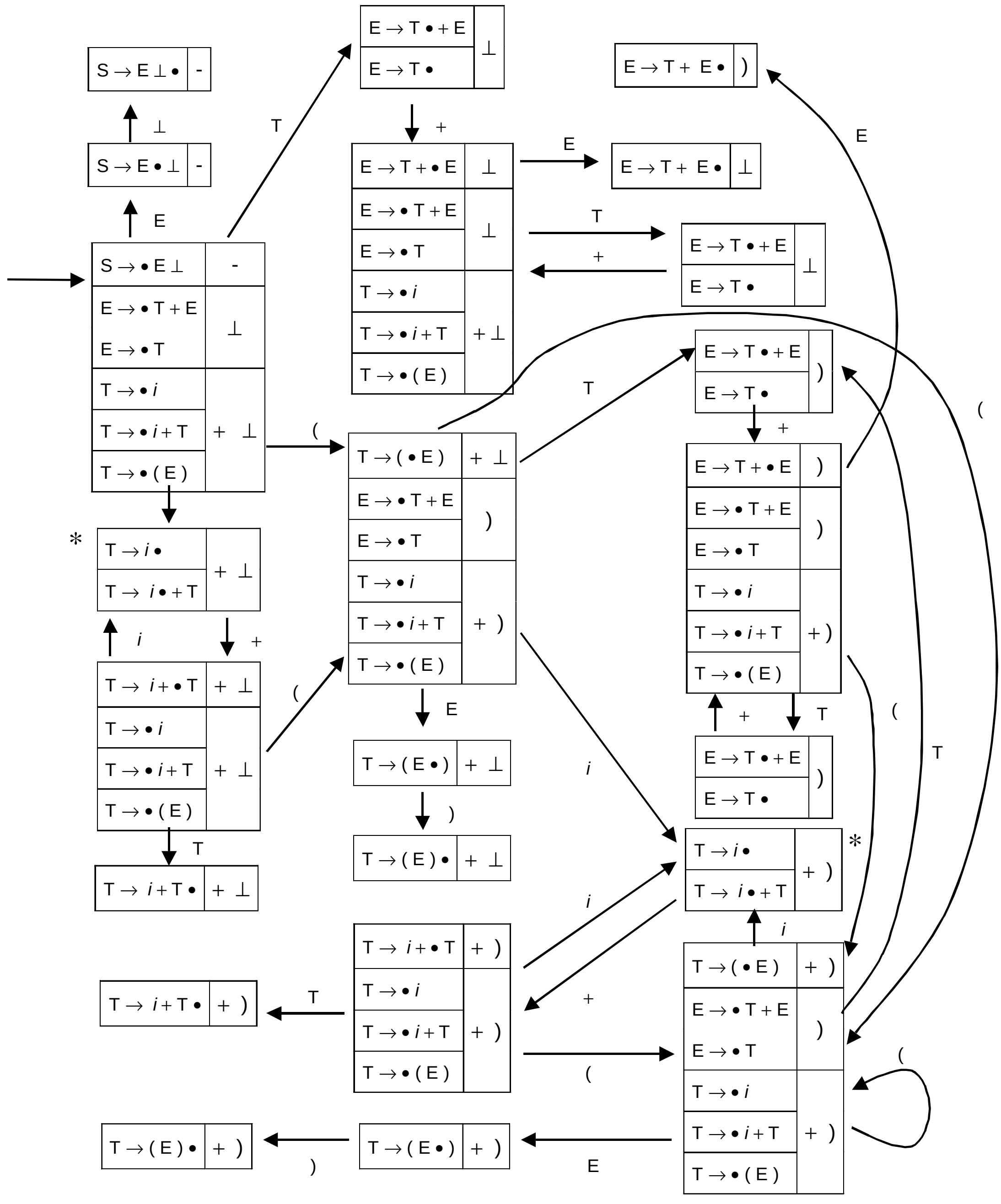
$$\cdots + (\ldots) + \cdots + (\ldots) + \ldots$$

oltre che parentesi annidate. Ma le catene di addizione sono generate in modo ambiguo da entrambi i nonterminali E e T, come si vede subito osservando la grammatica G.

Addendum Si può anche ragionare notando che il linguaggio $L(G)$ è definito da G come una lista di liste, ma con lo stesso separatore "+" al primo e secondo livello, e che per tale motivo è definito ambiguamente, non essendo chiaro a quale livello si debbano attribuire gli elementi della lista (cioè le variabili "i").

Inoltre, il linguaggio $L(G)$ (senza il terminatore "$\dashv$") coincide con la sua immagine speculare, cioè vale l'uguaglianza $L(G) = L(G)^R$; questa osservazione tornerà utile più avanti (per il punto (c)).

Non è necessario tracciare il grafo pilota $LR(1)$ della grammatica G, perché l'osservazione sull'ambiguità è sufficiente. Comunque per completezza eccolo:

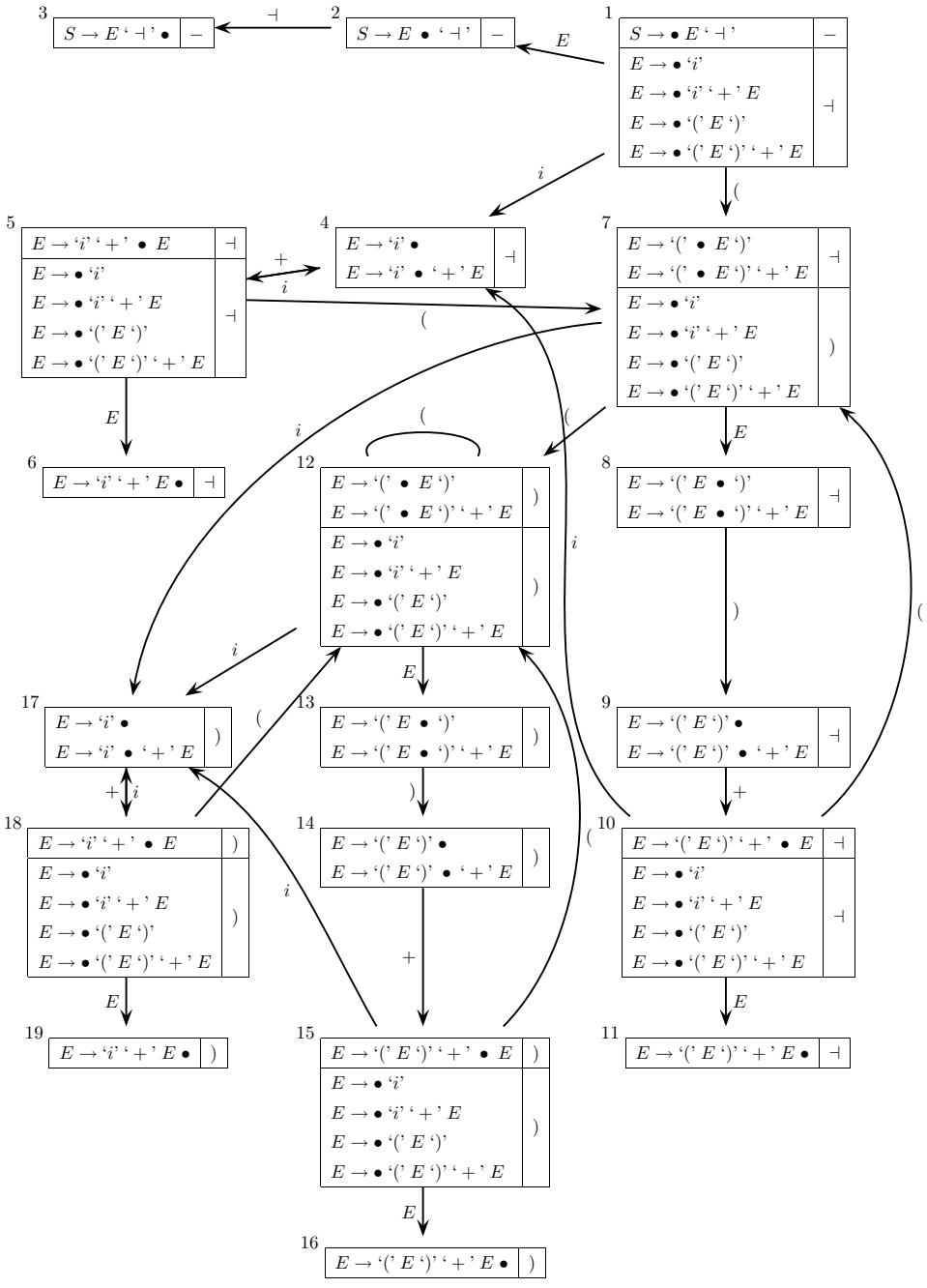


Nel grafo $LR(1)$ ci sono due macrostati inadeguati (marcati con $*$): sono quelli corrispondenti alla riduzione della regola $T \to$ 'i' $\bullet$ con prospezione comprendente "+", la quale va a confliggere con la regola di spostamento $T \to$ 'i' $\bullet$ ' + ' T quando in ingresso si ha proprio "+". Del resto è ovvio: non si sa se la catena di addizioni in esame sia prodotta dall'espansione di un unico termine T (ciò corrisponde a fare lo spostamento) oppure se il termine T corrente sia concluso (ciò corrisponde a fare la riduzione) e si debba andare avanti espandendo un nuovo termine T generato (ambiguamente) dalla regola di livello superiore $E \to T$ ' + ' E. Pertanto la grammatica G non è $LR(1)$.

(b) Se si vuole il determinismo, per prima cosa si deve togliere l'ambiguità, per esempio così (eliminando T perché $E \Longrightarrow T$, ma evitando produzioni nulle):

$$G' = \begin{cases} S & \rightarrow & E \text{ `}\dashv\text{'} \\ E & \rightarrow & \text{`}i\text{'} \mid \text{`}i\text{' `}+\text{'} E \mid \text{`(' } E \text{ `)'} \mid \text{`(' } E \text{ `)' `}+\text{'} E \end{cases}$$

Palesemente la grammatica G' è equivalente a G. Ma G' è già $LR(1)$, perché prospezionando le candidate di riduzione si distinguono tra sé e da quelle di spostamento. Basta tracciare il grafo pilota $LR(1)$ di G'. Eccolo:



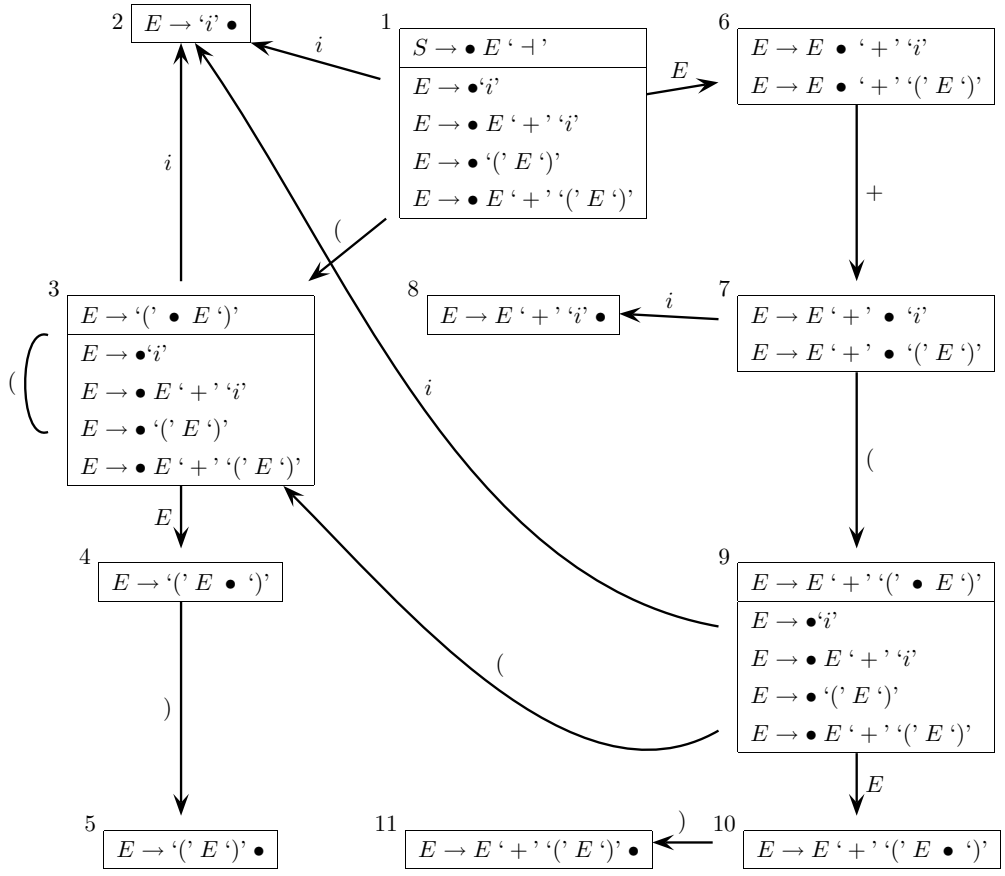
Tutti i macrostati del grafo $LR(1)$ sono adeguati, dunque G' è $LR(1)$.

Addendum In alternativa, si può cercare una grammatica equivalente a G, ma che sia $LL(1)$ e di conseguenza $LR(1)$. Ci sono alcune soluzioni possibili, strutturalmente poco eleganti ma che funzionano. Il lettore si eserciti da sé a progettarne una.

(c) Sì, si può trovare una grammatica G'' equivalente a G, ma che sia $LR(0)$. Intanto si osservi che G' non è $LR(0)$, perché contiene regole che sono prefisso di altre regole e ciò dà luogo a macrostati di tipo $LR(0)$ inadeguati. Mentre ricorrere a una grammatica che fosse $LL(1)$ (vedi punto precedente) non basterebbe, perché non è affatto garantito che risulterebbe anche $LL(0)$. Tuttavia, qui basta prendere G' e voltare la ricorsione destra in sinistra. Ecco come si può procedere:

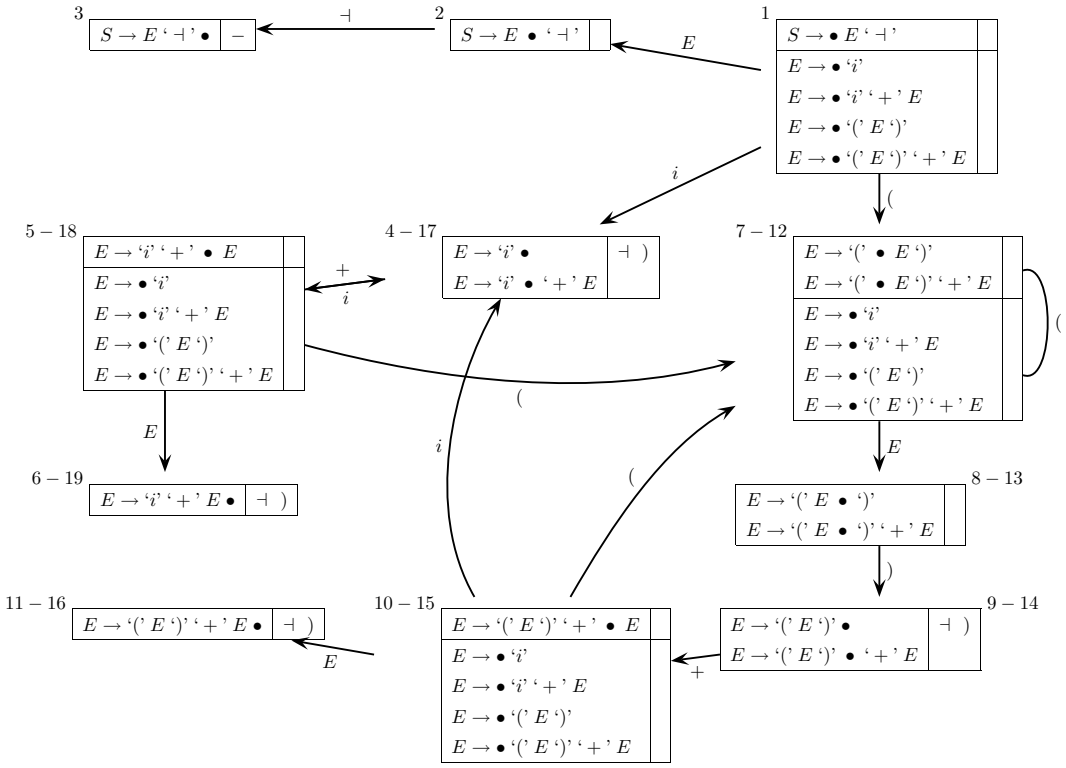
$$G'' = \begin{cases} S & \to & E \text{ ‘}\dashv\text{’} \\ E & \to & \text{‘}i\text{’} \mid E \text{ ‘}+\text{’ ‘}i\text{’} \mid \text{‘(’ } E \text{ ‘)’} \mid E \text{ ‘}+\text{’ ‘(’ } E \text{ ‘)’} \end{cases}$$

La ricorsione destra per generare le addizioni viene voltata in sinistra. Così nessuna regola è più prefisso di nessun'altra regola. Per la specularità delle stringhe del linguaggio, la grammatica G'' è equivalente a G' e dunque a G. Ora è evidente che nel grafo pilota $LR(0)$ di G'' le candidate di riduzione possono comparire solo isolatamente (dunque senza conflitto), perché le sequenze di scorrimento che conducono a esse sono tutte diverse. Ecco il grafo pilota $LR(0)$ di G'':



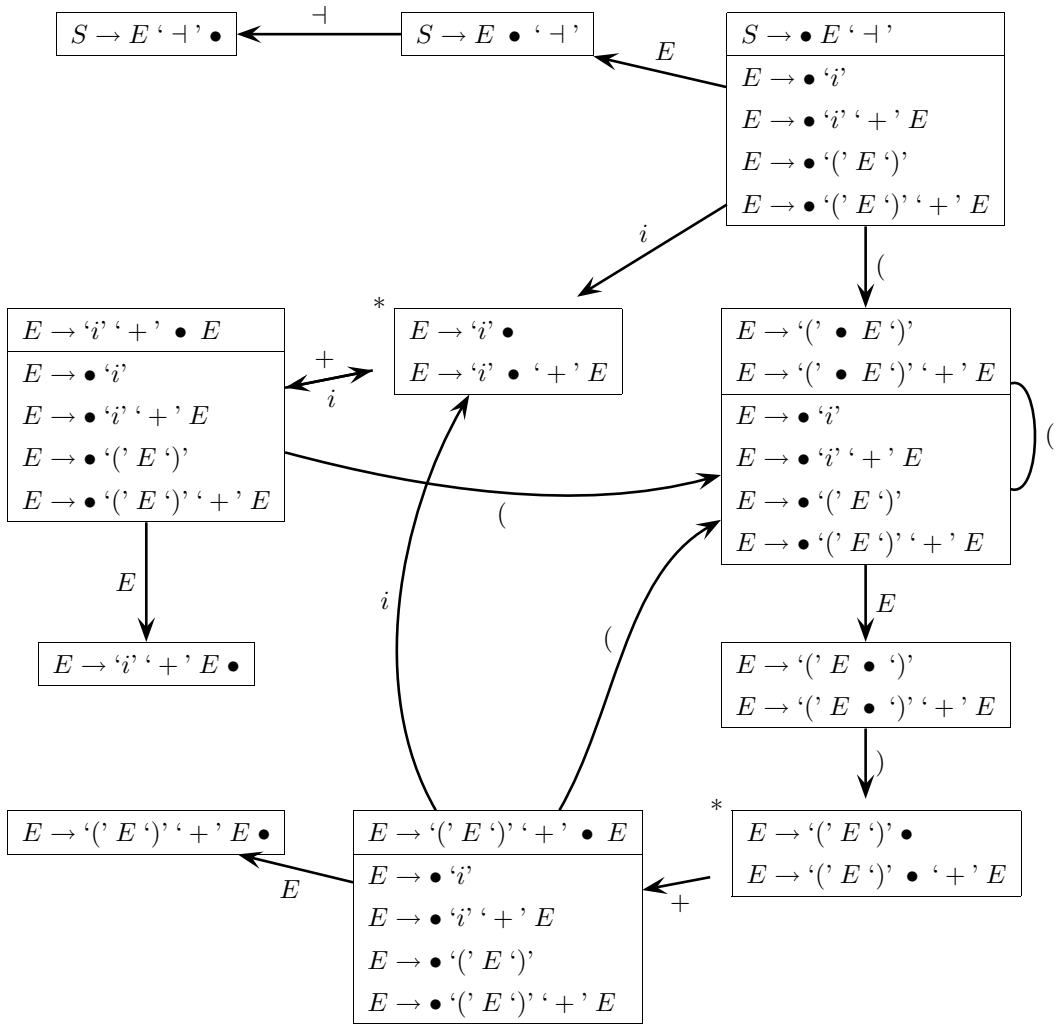
Tutti i macrostati del grafo $LR(0)$ sono adeguati, dunque G'' è $LR(0)$.

Addendum Incidentalmente, si può osservare che la grammatica G' è $LALR(1)$ (benché non $LR(0)$, come detto prima). Si tracci infatti il grafo pilota $LALR(1)$ di G', sovrapponendo i macrostati del grafo pilota $LR(1)$ di G' aventi la medesima base (cioè contenenti le medesime candidate, marcate in modo identico) e unendo gli insiemi di prospezione di candidate di riduzione uguali. Si ottiene quanto segue (le etichette numeriche dei macrostati indicano le sovrapposizioni):



Per comodità sono lasciati in vista solo gli insiemi di prospezione delle candidate di riduzione, non di quelle di spostamento (i quali servono solo per tracciare il grafo, ma non per usarlo come pilota dell'analizzatore sintattico). Si constata subito che tutti i macrostati del grafo $LALR(1)$ di G' sono adeguati. Pertanto G' è $LALR(1)$.

Comunque, la grammatica G' non è $LR(0)$, come già osservato sufficientemente prima. Tuttavia torna facile dimostrarlo qui rigorosamente tracciando il grafo pilota $LR(0)$ di G'. Come ben noto, esso ha la stessa topologia di quello $LALR(1)$, togliendo però del tutto gli insiemi di prospezione, ed è pertanto immediato tracciarlo avendo già il grafo $LALR(1)$. Eccolo:



Per comodità la numerazione dei macrostati è rimossa. Si vede subito che ci sono macrostati inadeguati (marcati con $*$), vale a dire quelli che contengono una candidata di riduzione e una di spostamento. Essi sono proprio i macrostati contenenti una regola che è prefisso di un'altra regola (giusta l'osservazione già fatta al punto (b)). Ovviamente la regola prefisso, essendo corta, è già candidata alla riduzione, mentre quella più lunga è ancora candidata allo spostamento, e senza l'aiuto fornito dalla prospezione non si sa come risolvere il conflitto.

Si ricordi però (punto (c)) che la grammatica G'', non ricorsiva a destra, è $LR(0)$.

4 Traduzione e analisi semantica 20%

1. Una serie di moltiplicazioni e addizioni tra interi, senza parentesi, va tradotta nella notazione polacca prefissa. La moltiplicazione ha precedenza sull'addizione e gli operandi sono numeri di una cifra (da 0 a 9). Il linguaggio sorgente L_s è definito dalla grammatica (sorgente) G_s seguente (assioma E):

G_s	G_p
$E \rightarrow T$ ' + ' E	
$E \rightarrow T$	
$T \rightarrow$ '0' ' × ' T	
$T \rightarrow$ '1' ' × ' T	
...	
$T \rightarrow$ '9' ' × ' T	
$T \rightarrow$ '0'	
$T \rightarrow$ '1'	
...	
$T \rightarrow$ '9'	

   Ecco un esempio di traduzione:

   $$3 + 1 \times 5 \times 4 + 2 \quad \xrightarrow{\tau} \quad \text{add 3 add mul 1 mul 5 4 2}$$

   Si svolgano i punti seguenti:

   (a) Si completi la grammatica di traduzione abbozzata sopra, scrivendo la grammatica pozzo G_p che genera la traduzione τ (si usi lo spazio previsto).

   (b) Senza cambiare il linguaggio sorgente L_s, si vorrebbe migliorare quanto più possibile la traduzione τ evitando di generare operazioni inutili, vale a dire:

   - moltiplicazione per 0
   - moltiplicazione per 1

   Ecco due esempi di traduzione ottimizzata τ':

   $$3 + 1 \times 5 + 2 \quad \xrightarrow{\tau'} \quad \text{add 3 add 5 2}$$

   $$3 \times 5 \times 0 + 4 + 0 \quad \xrightarrow{\tau'} \quad \text{add 0 add 4 0}$$

   Si scriva (a pagina successiva) una nuova grammatica di traduzione per calcolare la traduzione ottimizzata τ' (dandone le nuove componenti sorgente e pozzo) e si disegnino gli alberi sorgente e pozzo dell'ultimo esempio.

   (c) (facoltativo) Si discuta in breve come migliorare ulteriormente la traduzione τ', evitando di tradurre le operazioni di addizione quando un addendo è nullo.

G'_s	G'_p

Soluzione

(a) Ecco la grammatica completa della trasduzione τ, si tratta di una trasduzione infisso-prefisso piuttosto standard e senza alcuna difficoltà (assioma E):

G_s	G_p
$E \to\ T\ \text{`}+\text{'}\ E$	$E \to\ \text{`}\texttt{add}\text{'}\ T\ E$
$E \to\ T$	$E \to\ T$
$T \to\ \text{`}0\text{'}\ \text{`}\times\text{'}\ T$	$T \to\ \text{`}\texttt{mul}\text{'}\ \text{`}0\text{'}\ T$
$T \to\ \text{`}1\text{'}\ \text{`}\times\text{'}\ T$	$T \to\ \text{`}\texttt{mul}\text{'}\ \text{`}1\text{'}\ T$
$\ldots$	$\ldots$
$T \to\ \text{`}9\text{'}\ \text{`}\times\text{'}\ T$	$T \to\ \text{`}\texttt{mul}\text{'}\ \text{`}9\text{'}\ T$
$T \to\ \text{`}0\text{'}$	$T \to\ \text{`}0\text{'}$
$T \to\ \text{`}1\text{'}$	$T \to\ \text{`}1\text{'}$
$\ldots$	$\ldots$
$T \to\ \text{`}9\text{'}$	$T \to\ \text{`}9\text{'}$

(b) Per raffinare la trasduzione τ come richiesto, occorre modificare la grammatica sorgente G_s (ma senza cambiare il linguaggio L_s) onde distinguere tra i vari casi indicati. Infatti la classe sintattica (cioè il nonterminale) T, che genera il prodotto come catena di fattori, va così suddivisa:

- T_0 genera un prodotto che contiene almeno un fattore 0 (in posizione qualunque, anche all'inizio o alla fine del prodotto):
  $T_0 \stackrel{+}{\Rightarrow} \ldots \times 0 \times \ldots$
- T_N genera un prodotto che non contiene nessun fattore 0 ma non è costituito esclusivamente da fattori 1
  $T_N \stackrel{+}{\not\Rightarrow} \ldots \times 0 \times \ldots$ e $T_N \stackrel{+}{\not\Rightarrow} 1 \times 1 \times \ldots \times 1$
- T_1 genera un prodotto che contiene solo fattori 1
  $T_1 \stackrel{+}{\Rightarrow} 1 \times 1 \times \ldots \times 1$

Ciò detto, ecco la grammatica di traduzione G' (a un valore) che realizza τ'. Essa distingue la generazione e traduzione dei vari tipi di prodotto:

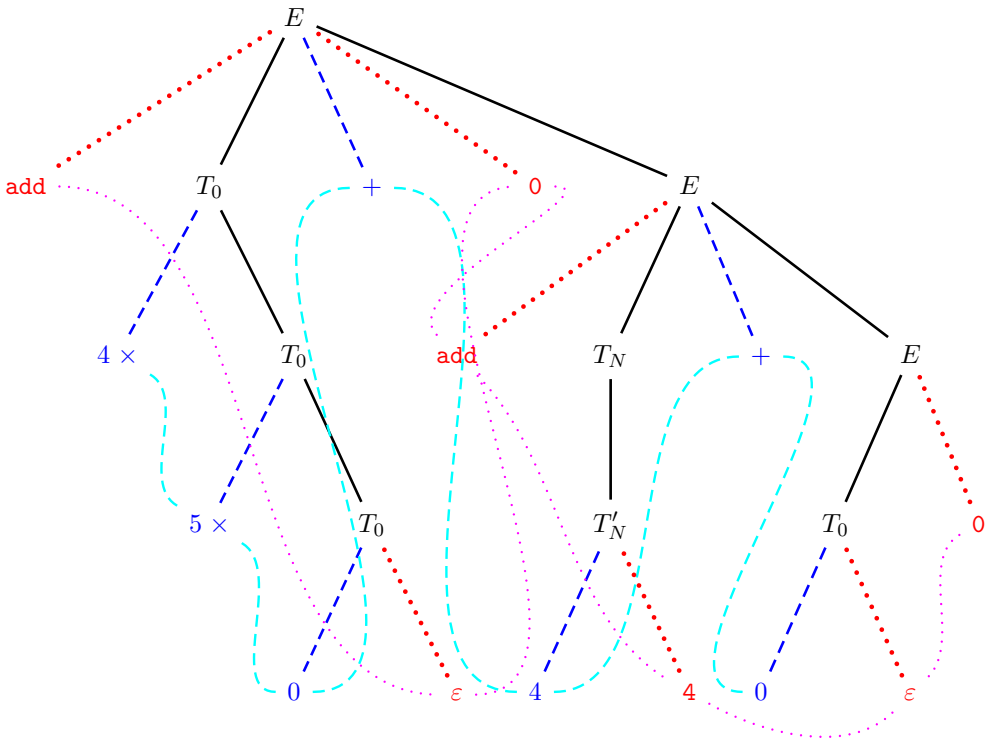
- nei casi di T_0 e T_1 ne emette in anticipo il risultato, 0 e 1, rispettivamente
- nel caso di T_N i fattori del prodotto vengono emessi inalterati, tranne sopprimere i fattori di valore 1 e le moltiplicazioni relative (si ricordi che T_N deve contenere almeno un fattore diverso sia da 0 sia da 1)

La grammatica G' è presentata in una forma volutamente piuttosto modulare, dove sono separati i sottinsiemi di regole che generano l'espressione completa e i tre tipi di prodotto elencati sopra. È forse possibile fondere parzialmente i moduli, riducendone il numero di regole ma rendendola meno leggibile.

G'_s	G'_p	
$E \to T_0$ ‘ + ’ E	$E \to$ ‘`add`’ T_0 ‘0’ E	
$E \to T_N$ ‘ + ’ E	$E \to$ ‘`add`’ T_N E	
$E \to T_1$ ‘ + ’ E	$E \to$ ‘`add`’ T_1 ‘1’ E	
$E \to T_0$	$E \to T_0$ ‘0’	
$E \to T_N$	$E \to T_N$	
$E \to T_1$	$E \to T_1$ ‘1’	
$T_0 \to$ ‘x’ ‘ × ’ T_0	$T_0 \to T_0$	$x = 1, \ldots, 9$
$T_0 \to$ ‘0’ ‘ × ’ T'_0	$T_0 \to T'_0$	
$T_0 \to$ ‘0’	$T_0 \to \varepsilon$	
$T'_0 \to$ ‘y’ ‘ × ’ T'_0	$T'_0 \to T'_0$	$y = 0, \ldots, 9$
$T'_0 \to$ ‘y’	$T'_0 \to \varepsilon$	$y = 0, \ldots, 9$
$T_N \to T_1$ ‘ × ’ T'_N	$T_N \to T_1\, T'_N$	
$T_N \to T'_N$	$T_N \to T'_N$	
$T'_N \to$ ‘z’ ‘ × ’ T_N	$T'_N \to$ ‘`mul`’ ‘z’ T_N	$z = 2, \ldots, 9$
$T'_N \to$ ‘z’ ‘ × ’ T_1	$T'_N \to$ ‘z’ T_1	$z = 2, \ldots, 9$
$T'_N \to$ ‘z’	$T'_N \to$ ‘z’	$z = 2, \ldots, 9$
$T_1 \to$ ‘1’ ‘ × ’ T_1	$T_1 \to T_1$	
$T_1 \to$ ‘1’	$T_1 \to \varepsilon$	

Ci possono essere altre soluzioni, naturalmente, che realizzano la trasduzione τ' con un meccanismo sintattico diverso, e anche soluzioni parziali che non realizzano tutta quanta l'ottimizzazione possibile per τ', ma solo alcuni aspetti di essa, per esempio non rimuovendo tutti i fattori 1 e altre semplificazioni similari.
Infine, ecco l'albero sintattico del secondo esempio di trasduzione τ':

$$3 \times 5 \times 0 + 4 + 0 \quad \xrightarrow{\tau'} \quad \texttt{add 0 add 4 0}$$



In nero è disegnato lo scheletro nonterminale dell'albero sintattico, che per definizione è comune a sorgente e destinazione, in blu la parte sorgente e in rosso la parte destinazione. Similmente la frontiera sorgente è in azzurro e quella destinazione in rosa. L'albero giustifica la correttezza della grammatica di traduzione G' progettata.
Il lettore può esercitarsi da sé a disegnare l'albero sintattico del primo esempio di trasduzione τ', che coinvolge il nonterminale T_1.

(c) Per come è impostata la soluzione G', è facile sopprimere del tutto l'emissione di un prodotto il cui valore sarà certamente 0. In effetti basta modificare la parte destinazione della prima coppia di regole di G':

$$E \to T_0 \text{ ‘+’ } E \qquad E \to \text{ ‘}\texttt{add}\text{’ } T_0 \text{ ‘0’ } E$$

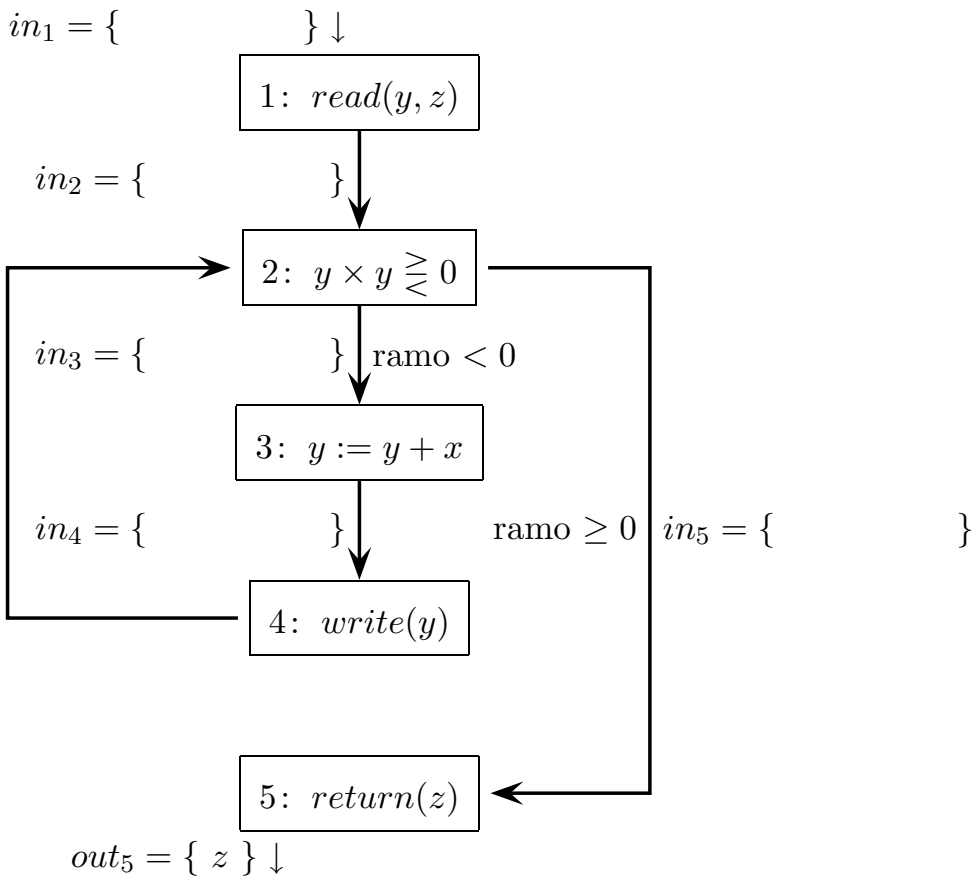
nel modo seguente:

$$E \to T_0 \text{ ‘+’ } E \qquad E \to T_0\ E$$

sopprimendo l'emissione dell'addendo 0 e dell'operatore "`add`", diventato inutile.

Addendum Volendo, si potrebbe gestire anche il caso dove tutti gli addendi si annullano, in quanto va comunque emesso 0 come valore complessivo dell'espressione E; infatti una traduzione diversa dalla stringa vuota deve esistere sempre, perché non avrebbe senso dire che l'espressione sorgente, quand'anche valesse 0, venisse tradotta in ε. Si può trattare tale caso sulla falsariga di quanto fatto per la moltiplicazione, suddividendo la classe sintattica (il nonterminale) E e prevedendo una sottoclasse (un nonterminale) E_0 che generi una catena di addizioni i cui termini (cioè i prodotti di fattori) siano tutti destinati ad annullarsi (per tali termini c'è già la classe sintattica pronta, T_0), e traducendo E_0 una volta per tutte con l'emettere un solo 0. Ovviamente resta una sottoclasse E_1 dove almeno un termine non si potrà annullare, e anch'essa va trattata nel modo opportuno. L'esercizio è relativamente semplice ed è lasciato al lettore.

2. Un sottoprogramma (routine), avente un parametro in ingresso x e uno in uscita z, è modellato tramite il grafo di controllo seguente:



Il nodo 2 è un'istruzione condizionale a due vie. Sul grafo è già specificato che il parametro z è vivo all'uscita del nodo 5.

Si svolgano i punti seguenti:

(a) Si scrivano le equazioni di flusso per calcolare in ogni punto del programma le *variabili vive*.

(b) Si calcoli la soluzione delle equazioni di flusso e si scrivano le variabili vive nella figura precedente (usando gli spazi previsti nei vari punti).

(c) (facoltativo) Sapendo che il nodo condizionale 2 effettua il test seguente:

$$\textsf{if} \quad (y \times y \geq 0) \quad \textsf{go to } 5 \textsf{ else go to } 3$$

si indichino quali insiemi di variabili vive cambino, e come.

Soluzione

(a) L'esercizio non presenta difficoltà di sorta e la tecnica di soluzione è del tutto standard, suddivisa in fasi (ma non si faccia confusione tra *variabili vive* e *definizioni raggiungenti*, che qui non c'entrano !). Eccole (punti (a) e (b)):
Calcolo dei termini costanti (definizione e uso di variabile):

#	def	use
1	y, z	-
2	-	y
3	y	x, y
4	-	y
5	-	z

- y e z figurano in *read* in 1, dunque lì sono definite
- y figura nell'espressione in 2, dunque lì è usata
- y viene assegnata in 3, dunque lì è definita
- x e y figurano nell'espressione in 3, dunque lì sono usate
- y figura in *write* in 4, dunque lì è usata
- z è usata del programma che chiama la routine

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

$$
\begin{array}{rcll}
in(1) & = & use(1) \cup (out(1) - def(1)) = & \text{– definizione di vitalità in ingresso} \\
 & = & \emptyset \cup (out(1) - \{y, z\}) = & \\
 & = & out(1) - \{y, z\} & \\
out(1) & = & in(2) & \text{– il nodo 1 ha una sola via d'uscita} \\
\hline
in(2) & = & use(2) \cup (out(2) - def(2)) = & \text{– definizione di vitalità in ingresso} \\
 & = & \{y\} \cup (out(2) - \emptyset) = & \\
 & = & \{y\} \cup out(2) & \\
out(2) & = & in(3) \cup in(5) & \text{– il nodo 2 ha due vie d'uscita} \\
\hline
in(3) & = & use(3) \cup (out(3) - def(3)) = & \text{– definizione di vitalità in ingresso} \\
 & = & \{x, y\} \cup (out(3) - \{y\}) = & \text{– vedi nota sotto} \\
 & = & \{x, y\} \cup out(3) & \\
out(3) & = & in(4) & \text{– il nodo 3 ha una sola via d'uscita} \\
\hline
in(4) & = & use(4) \cup (out(4) - def(4)) = & \text{– definizione di vitalità in ingresso} \\
 & = & \{y\} \cup (out(4) - \emptyset) = & \\
 & = & \{y\} \cup out(4) & \\
out(4) & = & in(2) & \text{– il nodo 4 ha una sola via d'uscita} \\
\hline
in(5) & = & use(5) \cup (out(5) - def(5)) = & \text{– definizione di vitalità in ingresso} \\
 & = & \{z\} \cup (out(5) - \emptyset) = & \\
 & = & \{z\} \cup out(5) & \\
out(5) & = & \{z\} & \text{– l'uso è già precisato nel testo}
\end{array}
$$

Addendum Attenzione a non commettere l'*errore gravissimo* di effettuare la semplificazione seguente (del tutto errata):

$$\{x, y\} \cup (out(3) - \{y\}) = \{x\} \cup out(3) \qquad \text{È FALSO !!!}$$

Si ripensi bene a come funziona l'operatore di differenza insiemistica.

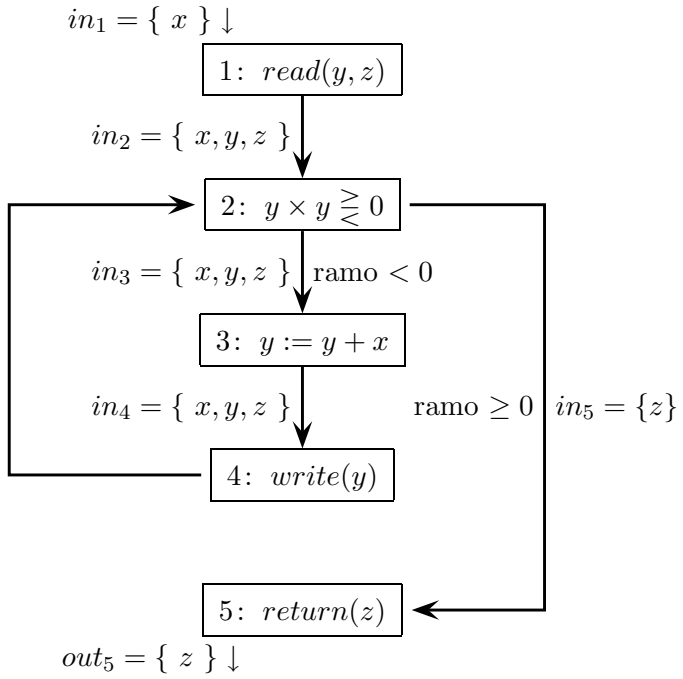
(b) Calcolo iterativo della soluzione alle equazioni di flusso:

	passo 0		**passo** 1		**passo** 2		**passo** 3	
#	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*
1	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	y	$\emptyset$	x, y, z
2	$\emptyset$	$\emptyset$	y	$\emptyset$	y	x, y, z	x, y, z	x, y, z
3	$\emptyset$	$\emptyset$	x, y	$\emptyset$	x, y	y	x, y	y
4	$\emptyset$	$\emptyset$	y	$\emptyset$	y	y	y	y
5	$\emptyset$	$\emptyset$	z	z	z	z	z	z

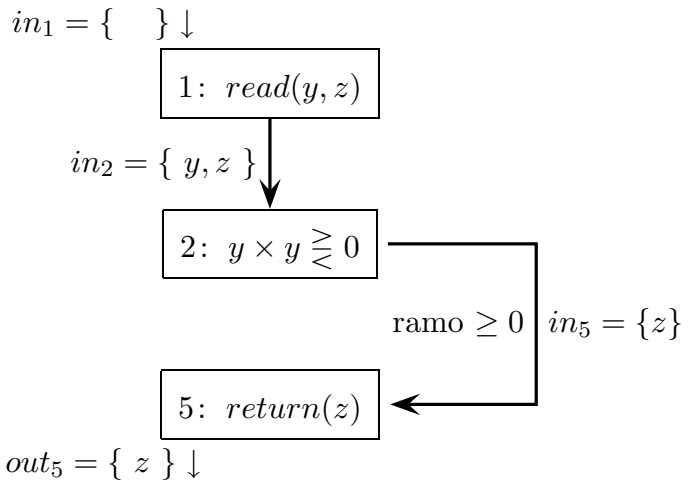
	passo 4		**passo** 5		**passo** 6		**passo** 7	
#	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*
1	x	x, y, z	x	x, y, z	x	x, y, z	x	x, y, z
2	x, y, z	x, y, z	x, y, z	x, y, z	x, y, z	x, y, z	x, y, z	x, y, z
3	x, y	y	x, y	x, y, z	x, y, z	x, y, z	x, y, z	x, y, z
4	y	x, y, z	x, y, z	x, y, z	x, y, z	x, y, z	x, y, z	x, y, z
5	z	z	z	z	z	z	z	z

Al passo 6 si raggiunge la convergenza, giacché il passo 7 non differisce dal passo 6. In totale dunque occorrono 6 passi per calcolare la soluzione. Può darsi si riesca ad accelerare il calcolo, se dalle equazioni si deducono delle identità tra insiemi sfruttabili per fondere due o più passi in uno. Il lettore si può esercitare da sé a sveltire il procedimento, come possibile.

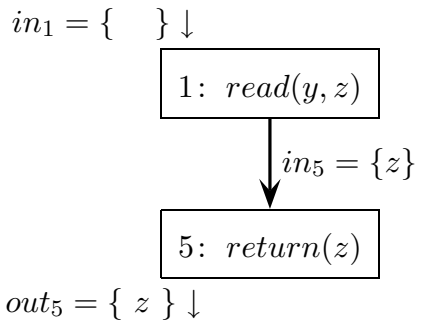
Riporto della soluzione (variabili vive in ingresso) sul grafo di controllo:



(c) Esaminando la semantica della routine, si vede che il corpo 3 4 del ciclo non è mai eseguito (perché il quadrato $y \times y = y^2$ non può essere < 0) e pertanto i nodi 3 e 4 si possono eliminare. Dunque all'ingresso del nodo 2 il parametro x non è più vivo. Pertanto x non è vivo neppure all'ingresso del nodo 1 (in definitiva il parametro x è inutile, così come i nodi 3 e 4 sono irraggiungibili). Ecco il grafo semplificato con le variabili vive ridotte:



Si può eliminare anche il nodo di test 2, in quanto il predicato ivi contenuto è tautologico, e di conseguenza la variabile y non è più viva all'ingresso del nodo. In conclusione la routine legge in ingresso la variabile z e la restituisce in uscita così com'è. Ecco il grafo semplificato con le variabili vive ulteriormente ridotte:



Naturalmente, volendo si potrebbe anche eliminare y da *read* (ciò ha però effetto collaterale sul comportamento di I/O della routine).