Linguaggi Formali e Compilatori
(Formal Languages and Compilers)

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

Prova scritta - 4 marzo 2011 - Parte I: Teoria

CON SOLUZIONI - A SCOPO DIDATTICO LE SOLUZIONI SONO MOLTO ESTESE E COMMENTATE VARIAMENTE - NON SI RICHIEDE CHE IL CANDIDATO SVOLGA IL COMPITO IN MODO ALTRETTANTO AMPIO, BENSÌ CHE RISPONDA IN MODO APPROPRIATO E A SUO GIUDIZIO RAGIONEVOLE

NOME:

MATRICOLA: FIRMA:

ISTRUZIONI - LEGGERE CON ATTENZIONE:

- L'esame si compone di due parti:
  - I (80%) Teoria:
    1. espressioni regolari e automi finiti
    2. grammatiche libere e automi a pila
    3. analisi sintattica e parsificatori
    4. traduzione sintattica e analisi semantica
  - II (20%) Esercitazioni Flex e Bison
- Per superare l'esame l'allievo deve sostenere con successo entrambe le parti (I e II), in un solo appello oppure in appelli diversi, ma entro un anno.
- Per superare la parte I (teoria) occorre dimostrare di possedere conoscenza sufficiente di tutte le quattro sezioni (1-4), rispondendo alle domande obbligatorie.
- È permesso consultare libri e appunti personali.
- Per scrivere si utilizzi lo spazio libero e se occorre anche il tergo del foglio; è vietato allegare nuovi fogli o sostituirne di esistenti.
- Tempo: Parte I (teoria): 2h.30m - Parte II (esercitazioni): 45m

1 Espressioni regolari e automi finiti 20%

1. Dato l'alfabeto $\Sigma = \{\ a,\ b,\ c\ \}$, di tre lettere, si consideri il linguaggio regolare L di alfabeto Σ, definito dai vincoli seguenti:

   - le stringhe iniziano con la lettera a e finiscono con la lettera b o c
   - se una stringa contiene la tripla di lettere consecutive $a\,c\,b$, non appartiene a L

   Si risponda alle domande seguenti:

   (a) Si scriva l'espressione regolare R che genera il linguaggio L; è permesso usare anche intersezione e complemento.

   (b) Si tracci un automa a stati finiti A, preferibilmente deterministico, che riconosce il linguaggio L.

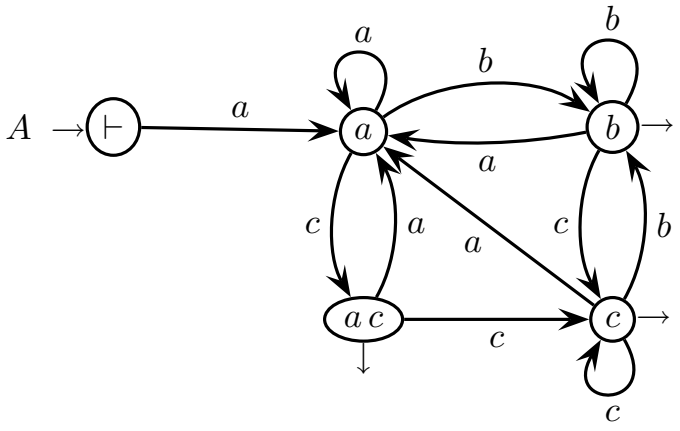
Soluzione

(a) Ecco l'espressione estesa R, con complemento e intersezione:

$$R = \Big(\ a\ \Sigma^*\ \ (\ b\ \mid\ c\ )\ \Big)\ \bigcap\ \Big(\ \neg\,(\ \Sigma^*\ a\ c\ b\ \Sigma^*\ )\ \Big)$$

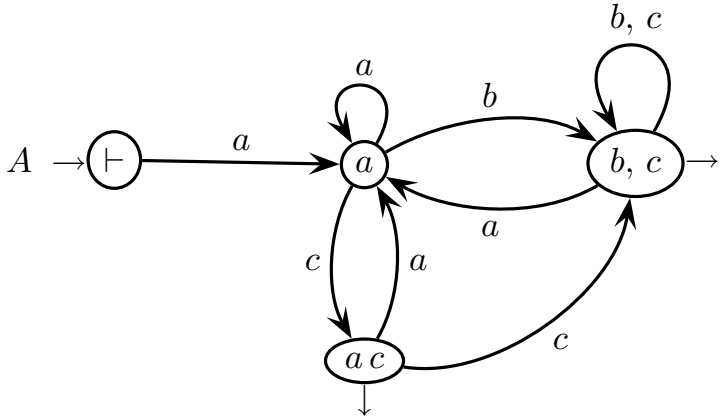
Tramite l'automa A di cui al punto (b), si potrebbe ricavare un'espressione equivalente con solo gli operatori standard (si veda l'osservazione in calce).

(b) È facile costruire direttamente un automa deterministico. Infatti il linguaggio L è locale con triple. Pertanto c'è un automa A deterministico con stati etichettati dalle tre lettere, uno stato iniziale e uno stato etichettato dalla coppia $a\,c$. Eccolo:



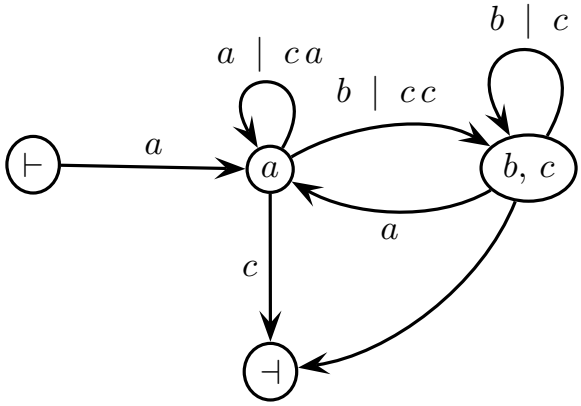
L'automa A è deterministico e ovviamente l'arco $a\,c \xrightarrow{b} b$ manca. Si entra negli stati a, b o c solo con la lettera a, b o c, rispettivamente, e nello stato $a\,c$ solo con la sequenza di lettere $a\,c$.

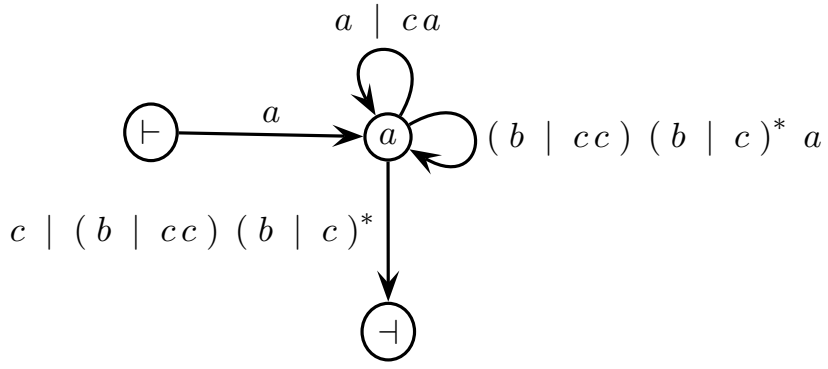
Si può osservare che l'automa A è ridotto (non ha stati inutili), ma non è minimo: gli stati finali b e c sono indistinguibili. Si riduce l'automa unificandoli così:

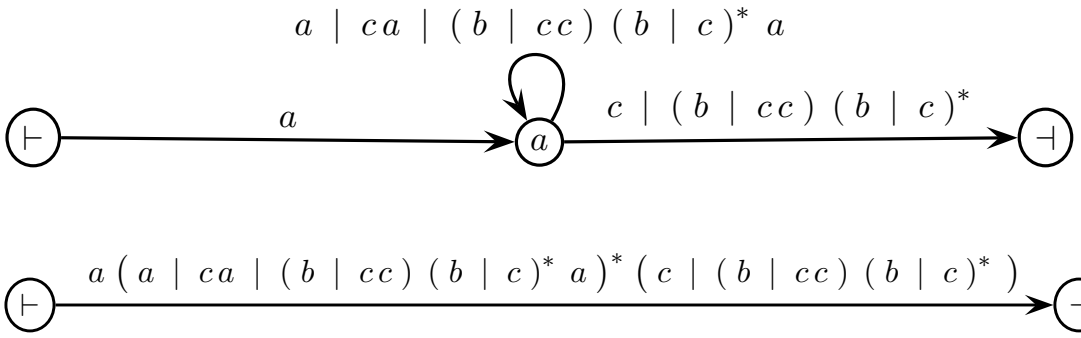


Ora l'automa è in forma minima, poiché gli stati finali ($a\,c$) e (b, c) sono chiaramente distinguibili ($a\,c$ non ha arco b uscente), e quelli non finali $\vdash$ e a sono pure ovviamente distinguibili ($\vdash$ non ha archi b e c uscenti). In alternativa, per ottenere l'automa A si può usare la costruzione di complemento e prodotto cartesiano partendo dai componenti dell'espressione estesa R (lasciata al lettore).

Osservazione: volendo trovare un'espressione regolare R non estesa, basta eliminare i nodi dall'automa A, così (i vari passaggi sono un po' compattati):







$$\begin{aligned} R &= a\,(\,a \mid ca \mid (\,b \mid cc\,)\,(\,b \mid c\,)^*\,a\,)^*\,(\,c \mid (\,b \mid cc\,)\,(\,b \mid c\,)^*\,) \\ &= a\,\Big(\,(\,\varepsilon \mid c \mid (\,b \mid cc\,)\,(\,b \mid c\,)^*\,)\,a\,\Big)^*\,(\,c \mid (\,b \mid cc\,)\,(\,b \mid c\,)^*\,) \end{aligned}$$

Ora basta osservare che $x\,(\,(\,\varepsilon \mid y\,)\,x\,)^* = x^+\,(\,y\,x^+\,)^*$, porre $x = a$ e $y = c \mid (\,b \mid cc\,)\,(\,b \mid c\,)^*$ e concludere così:

$$R = a^+\,\Big(\,\big(c \mid (\,b \mid cc\,)\,(\,b \mid c\,)^*\,\big)\,a^+\,\Big)^*\,\big(c \mid (\,b \mid cc\,)\,(\,b \mid c\,)^*\,\big)$$

In forma standard, l'espressione R è un po' complicata ma comprensibile. Basta notare che il termine $c \mid (\,b \mid cc\,)\,(\,b \mid c\,)^*$ è il linguaggio universale di alfabeto $\{\,b,\,c\,\}$ escluse le stringhe ε e cb, ossia è $\{\,b,\,c\,\}^+ - \{\,cb\,\}$. Pertanto l'espressione R potrebbe anche essere riscritta nella forma seguente, leggermente estesa:

$$R = a^+\,\Big(\,\big(\{\,b,\,c\,\}^+ - \{\,cb\,\}\,\big)\,a^+\,\Big)^*\,\big(\{\,b,\,c\,\}^+ - \{\,cb\,\}\,\big)$$

che riesce del tutto chiara e anzi ne dà un'altra formulazione possibile, più acuta di quella data prima.

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

$$R = \Big(\big(a\ b \big)^+\ a\ c^+ \Big)^+$$

Si risponda alle domande seguenti:

(a) Si costruisca l'automa a stati finiti deterministico A che riconosce il linguaggio regolare generato da R, tramite il metodo Berri-Sethi (BS).

(b) (facoltativa) Partendo dall'automa A ottenuto al punto precedente, si ricavi, usando il metodo di eliminazione dei nodi (BMC), un'espressione regolare R' per il linguaggio riconosciuto da A. Si verifichi che l'espressione regolare R' risultante è equivalente a R.

Soluzione

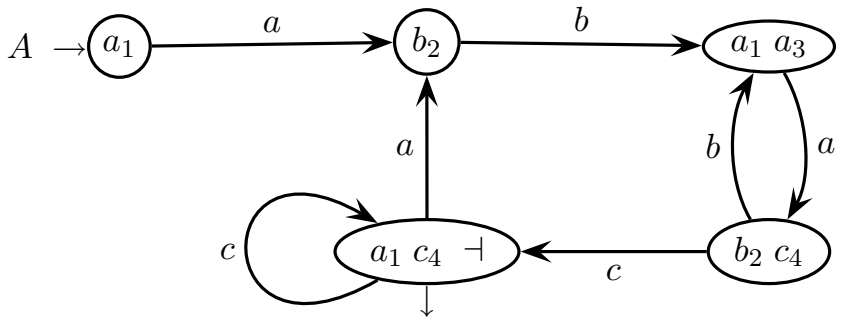
(a) Ecco l'espressione numerata e terminata:

$$R_{\#} = \Big(\big(a_1\ b_2 \big)^+\ a_3\ c_4^+ \Big)^+\ \dashv$$

Insiemi degli inizi e dei seguiti:

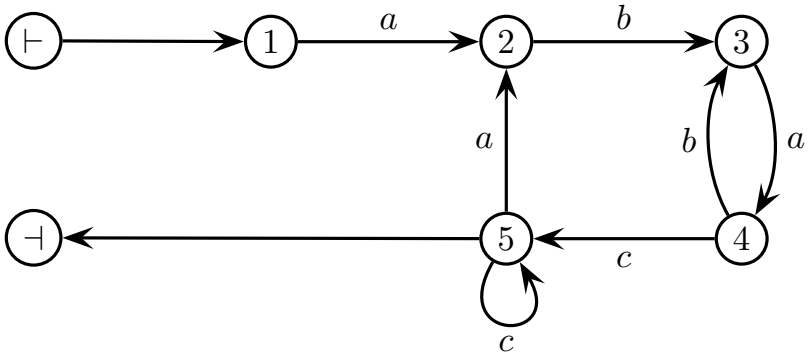
inizi	a_1
generatore	seguiti
a_1	b_2
b_2	$a_1\ a_3$
a_3	c_4
c_4	$a_1\ c_4\ \dashv$

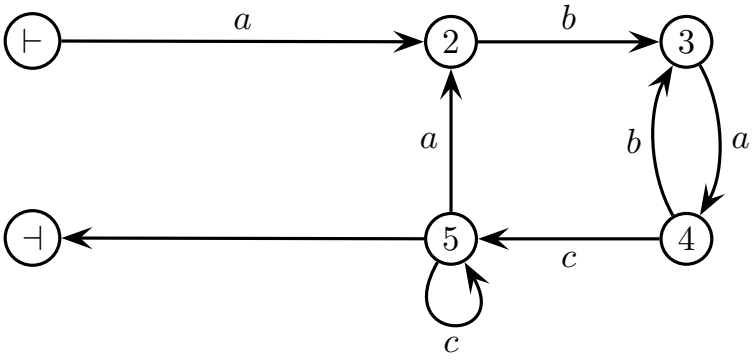
Automa deterministico A di Berri-Sethi derivato da $R_{\#}$:

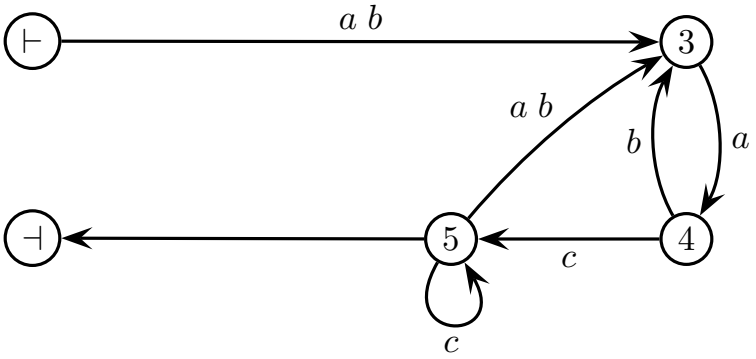


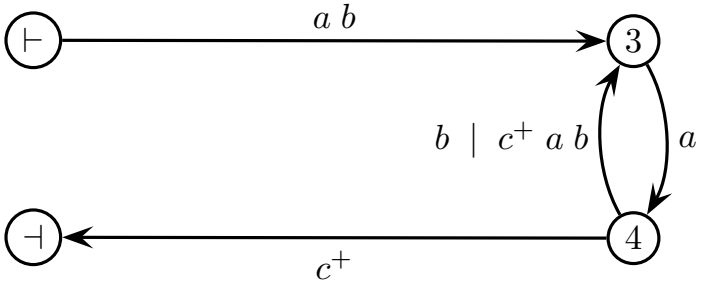
Un rapido esame mostra che l'automa A è in forma minima.

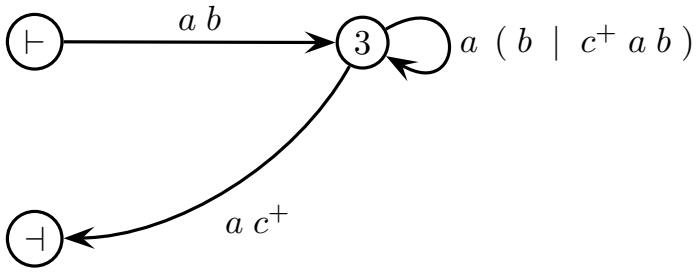
(b) Ecco l'applicazione ad A del metodo di eliminazione dei nodi (BS):

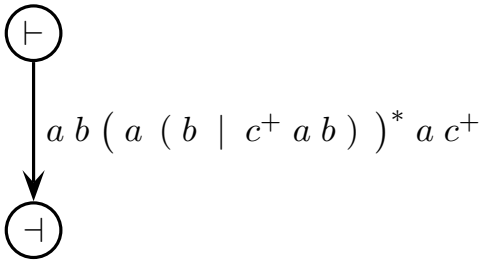












da cui si ha:

$$R' = a\ b\ \big(\ a\ \ (\ b\ \mid\ c^{+}\ a\ b\ )\ \big)^{*}\ a\ c^{+}$$

Per verificare l'equivalenza di R e R', semiformalmente si ha:

$$\begin{aligned}
R' &= a\ b\ \big(\ a\ \ (\ b\ \mid\ c^{+}\ a\ b\ )\ \big)^{*}\ a\ c^{+} \\
&= a\ b\ \big(\ a\ b\ \mid\ a\ c^{+}\ a\ b\ \big)^{*}\ a\ c^{+} \\
&= a\ b\ \big(\ (\ \varepsilon\ \mid\ a\ c^{+}\ )\ a\ b\ \big)^{*}\ a\ c^{+} \\
&= (\ a\ b\ )^{+}\ \big(\ a\ c^{+}\ \ (\ a\ b\ )^{+}\ \big)^{*}\ a\ c^{+} \\
&= \big(\ (\ a\ b\ )^{+}\ \ a\ c^{+}\ \big)^{+} \\
&= R
\end{aligned}$$

con una serie di passaggi che usa la semplificazione $x\ \big(\ (\ \varepsilon\ \mid\ y\ )\ \ x\ \big)^{*} = x^{+}\ (\ y\ x^{+}\ )^{*}$ ponendo $x = a\,b$ e $y = a\,c^{+}$, analogamente a quanto fatto nell'osservazione in calce all'esercizio precedente, e la semplificazione $x^{+}\ (\ y\ x^{+}\ )^{*}\ y = (\ x^{+}\ y\ )^{+}$, con le stesse posizioni di x e y.

2 Grammatiche libere e automi a pila 20%

1. Si consideri il linguaggio L seguente (le cui stringhe sono palindromi con c centrale), di alfabeto $\Sigma = \{ a, b, c \}$, che come ben noto è libero:

$$L = \left\{ u\ c\ u^R \ \mid \ u \in \{ a,\ b \}^* \right\}$$

Si risponda alle domande seguenti:

(a) Si consideri il linguaggio complementare $\overline{L}$ di L e si dica, giustificando la risposta, se il linguaggio $\overline{L}$ è libero deterministico.

(b) Si scriva la grammatica G di $\overline{L}$, a discrezione usando la forma BNF estesa.

(c) (facoltativa) Si dica se la grammatica G precedente è ambigua.

Soluzione

(a) The answer is yes: language L is clearly deterministic context-free and it is well known (see Chapter 4 p. 163 of the textbook Crespi Reghizzi, *Formal languages and compilers*) that the DET family is closed under complement.

(b) The next grammar $\overline{G}$ of $\overline{L}$ can be obtained observing that every string in $\neg L = \neg \left\{ u\ c\ v \ \mid \ u \in \{ a,\ b \}^* \ \wedge \ v = u^R \right\}$ necessarily falls into one of the cases or sub-cases below:

i. strings not containing any letter c

ii. strings containing two or more letters c

iii. strings containing exactly one letter c, parted into two sub-cases:

A. strings $u\ c\ v$ with left and right factors u and v of different length, i.e. with $|u| \neq |v|$; more precisely, strings of the forms:

$$\underbrace{(\ a\ \mid\ b\ )^+\ x}_{u}\ c\ \underbrace{x^R}_{v} \qquad \text{or} \qquad \underbrace{x}_{u}\ c\ \underbrace{x^R\ (\ a\ \mid\ b\ )^+}_{v}$$

B. strings $u\ c\ v$ with left and right factors u and v possibly of the same length, i.e. with possibly $|u| = |v|$, but such that u and v are not mirror images of each other; more precisely, strings of the form:

$$\underbrace{y\ x}_{u}\ c\ \underbrace{x^R\ z}_{v} \qquad \text{such that} \quad z \neq y^R \quad \text{and} \quad y,\ z \neq \varepsilon$$

For each case and sub-case a set of simple rules can be written (axiom $\overline{S}$):

i. $\overline{S} \rightarrow (\ a\ \mid\ b\ )^*$

ii. $\overline{S} \rightarrow \Sigma^*\ c\ \Sigma^*\ c\ \Sigma^*$

iii. sub-cases:

A. $\overline{S} \rightarrow (\ a\ \mid\ b\ )^+\ X\ \mid\ X\ (\ a\ \mid\ b\ )^+$ and $X \rightarrow a\ X\ a\ \mid\ b\ X\ b\ \mid\ c$

B. $\overline{S} \rightarrow (\ a\ \mid\ b\ )^*\ a\ X\ b\ (\ a\ \mid\ b\ )^*\ \mid\ (\ a\ \mid\ b\ )^*\ b\ X\ a\ (\ a\ \mid\ b\ )^*$

(c) The previous grammar (axiom $\overline{S}$) is somewhat simple and is not ambiguous, because the cases (i), (ii) and the sub-cases (iii.A) and (iii.B) are individually not ambiguous and mutually exclusive.

2. Si consideri il frammento di linguaggio di programmazione con le caratteristiche seguenti, in stile C:

   - il programma consiste di una lista di dichiarazioni, eventualmente vuota, e di una lista d'istruzioni (statement), non vuota
   - l'identificatore alfanumerico è schematizzato dal terminale id e la costante intera è schematizzata dal terminale const, che non vanno espansi
   - ci sono i tipi scalari intero e puntatore, e il tipo strutturato struct; per esempio:

     ```
     int num;                struct {            struct T1 {
                               int a2;             int b3;
     int * pnum;               ...                 ...
                             } s1;               };

     struct T1 s2;           struct T1 * ps;
     ```

     si noti che s1 e s2 sono oggetti (variabili) di tipo struct e che T1 è un tipo struct; le struct possono essere annidate
   - le istruzioni sono assegnamenti a variabile scalare o a campo scalare di struct (eventualmente annidate); per esempio:

     ```
     num = 1;              *pnum = num;        pnum = &num;

     s1.a2 = 2;            *ps.b3 = 5;
     ```

   - si può usare l'operatore freccia "->" per aprire struct puntate; per esempio:

     ```
     ps->b3 = 5;
     ```

     e così via per struct concatenate
   - la parte destra di un assegnamento può essere un'espressione contenente costanti intere, variabili intere, l'operazione di addizione, e gli operatori "∗" e "&" (ossia referenziamento e dereferenziamento); non ci sono parentesi

   Si scriva una grammatica, non ambigua e in forma estesa (EBNF), che modella il linguaggio così descritto.

Soluzione

Ecco la grammatica richiesta (assioma PROG), piuttosto ovvia:

$$
\left\{
\begin{array}{rcl}
\langle \mathsf{PROG} \rangle & \rightarrow & \left(\langle \mathsf{DECL} \rangle \text{ ';' } \right)^* \left(\langle \mathsf{STAT} \rangle \text{ ';' } \right)^+ \\
\hline
\langle \mathsf{DECL} \rangle & \rightarrow & \langle \mathsf{INT} \rangle \mid \langle \mathsf{S\_TYP} \rangle \mid \langle \mathsf{S\_VAR} \rangle \\
\langle \mathsf{STAT} \rangle & \rightarrow & \left(\text{'}*\text{'} \right)^* \langle \mathsf{S\_CHAIN} \rangle \text{ '=' } \langle \mathsf{EXPR} \rangle \\
\hline
\langle \mathsf{INT} \rangle & \rightarrow & \text{'int' } \left(\text{'}*\text{'} \right)^* \langle \mathsf{ID\_LIST} \rangle \\
\langle \mathsf{S\_TYP} \rangle & \rightarrow & \text{'struct' 'id' } \langle \mathsf{S\_BODY} \rangle \\
\langle \mathsf{S\_VAR} \rangle & \rightarrow & \text{'struct' } \left(\text{'id' } \mid \langle \mathsf{S\_BODY} \rangle \right) \left(\text{'}*\text{'} \right)^* \langle \mathsf{ID\_LIST} \rangle \\
\langle \mathsf{EXPR} \rangle & \rightarrow & \langle \mathsf{TERM} \rangle \left(\text{'}+\text{'} \langle \mathsf{TERM} \rangle \right)^* \\
\hline
\langle \mathsf{ID\_LIST} \rangle & \rightarrow & \text{'id' } \left(\text{',' 'id'} \right)^* \\
\langle \mathsf{S\_BODY} \rangle & \rightarrow & \text{'\{' } \left(\langle \mathsf{DECL} \rangle \text{ ';' } \right)^+ \text{'\}'} \\
\langle \mathsf{TERM} \rangle & \rightarrow & \text{'const' } \mid \left[\left(\text{'}*\text{'} \right)^+ \mid \text{'\&'} \right] \langle \mathsf{S\_CHAIN} \rangle \\
\hline
\langle \mathsf{S\_CHAIN} \rangle & \rightarrow & \text{'id' } \left(\left(\text{'.'} \mid \text{'}->\text{'} \right) \text{ 'id'} \right)^*
\end{array}
\right.
$$

La grammatica non è ambigua ed è in forma estesa. Le parentesi quadre indicano opzionalità. La stratificazione delle regole mette in evidenza la struttura.

Il termine elementare è una costante numerica 'const', un identificatore naturale 'id', un identificatore referenziato o dereferenziato tramite '$*$' (eventualmente ripetuto) o '&', oppure una lista eventualmente (de)referenziata d'identificatori naturali concatenati tramite gli operatori '.' e '$->$'. Chiaramente l'uso corretto di tali operatori dipende dal tipo degl'identificatori, ma questo è un aspetto semantico non modellabile sintatticamente.

Il testo non indica esplicitamente le precedenze tra i vari operatori: le regole danno a '.' e '$->$' precedenza su '$*$' e '&', come in linguaggio C; e ammettono l'eventuale uso di '$*$' e '&' solo in testa alla lista d'identificatori, non essendo altrimenti chiaro in quale ordine si dovebbero applicare. È una scelta semplice che contempla sintatticamente tutti gli esempi dati. Per modificare tale precedenza occorrerebbero le parentesi, peraltro qui non previste. Si noti che l'esempio `*ps.b3 = 5` è semanticamente scorretto poiché in C verrebbe interpretato come `*(ps.b3) = 5`, e per renderlo corretto andrebbe scritto come `(*ps).b3 = 5`, se qui ci fossero parentesi.

Circa l'uso di struct: la grammatica data sopra permette di dichiarare tipi `T` come `struct T { ... }` e variabili `s` come `struct { ... } s` o struct T `s`, ma non tipi e variabili insieme come `struct T { ... } s`; in linguaggio C quest'ultima possibilità esiste, ma qui è omessa per semplicità peraltro essendo di ben scarsa utilità o perfino una bizzarria (gli esempi non la contemplano neppure); tuttavia volendo si potrebbero facilmente ritoccare le regole in modo da ammetterla; invece le regole date sopra non permettono di dichiarare una struct anonima come `struct { ... }`, giacché essa risulterebbe del tutto inutilizzabile; del resto neppure il linguaggio C la ammette.

3 Analisi sintattica e parsificatori 20%

1. Si consideri la grammatica G seguente (assioma A), di alfabeto $\{ a, b, c \}$:

$$G \begin{cases} A & \rightarrow & B\ a \\ B & \rightarrow & B\ c \mid A\ b \mid a \end{cases}$$

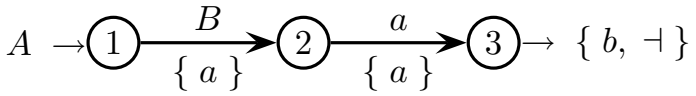
Si risponda alle domande seguenti:

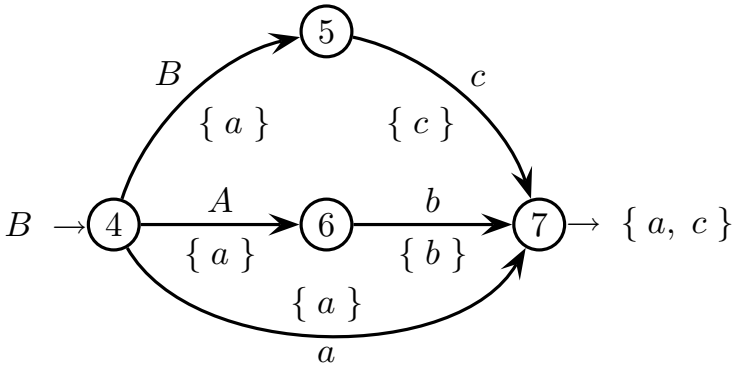
(a) Si mostri che la grammatica G non è di tipo $LL(k)$ per alcun $k \geq 1$.

(b) (facoltativa) Si trasformi la grammatica G in un'equivalente grammatica G' che sia di tipo LL, e si trovi qual è il più piccolo k per cui G' è di tipo $LL(k)$.

Soluzione

(a) Basta osservare che G è ricorsiva a sinistra, dunque non LL per nessun $k \geq 1$. Per completezza, ecco gli automi (deterministici e minimi) delle regole con gli insiemi guida per $k = 1$:





Da qui si vede che G non è $LL(1)$ nello stato 4 e che non lo è neppure per $k \geq 2$.

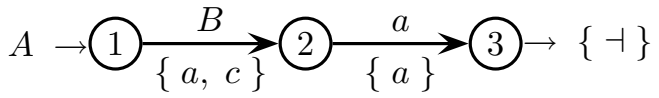
(b) Per rendere LL la grammatica bisogna eliminare la ricorsione sinistra diretta su B e quella indiretta su A tramite B. Si ottiene, sostituendo A nella produzione di B, quanto segue:

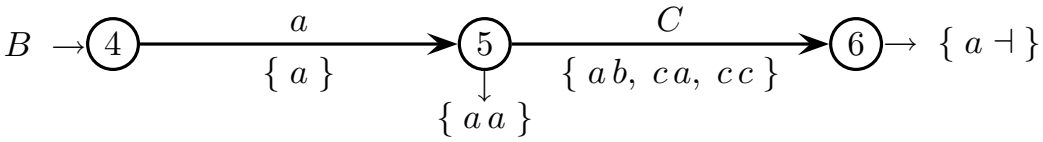
$$\begin{cases} A & \rightarrow & B\ a \\ B & \rightarrow & B\ c \mid B\ a\ b \mid a \end{cases}$$

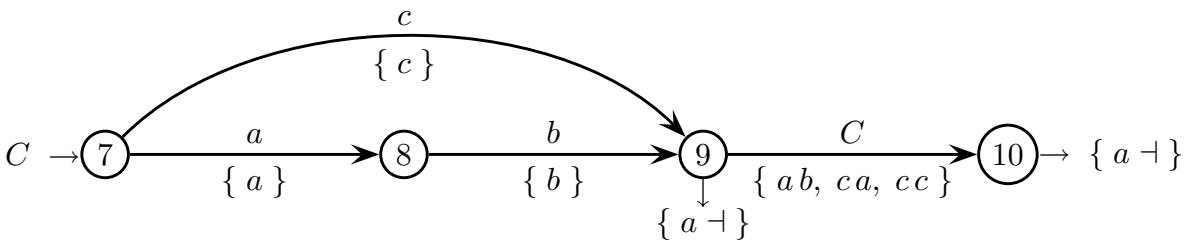
e poi, eliminando la ricorsione sinistra di B, si ottiene G', così:

$$G' \begin{cases} A & \rightarrow & B\ a \\ B & \rightarrow & a \mid a\ C \\ C & \rightarrow & c \mid a\ b \mid c\ C \mid a\ b\ C \end{cases}$$

Si vede poi che G' non è $LL(1)$, però è $LL(2)$. Eccone gli automi per $k = 1$ e $k = 2$ dove serve:







Da questi si vede che G' è $LL(2)$, ma non $L(1)$.
Peraltro il linguaggio $L(G)$ è regolare, definito anche dall'espressione regolare R seguente:

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

Se ne dà facilmente una grammatica LL lineare a dx. Per esempio la seguente (assioma A):

$$\begin{cases} A & \rightarrow & a\ B \\ B & \rightarrow & a\ b\ B \mid c\ B \mid a \end{cases}$$

che, tracciandone gli automi, risulta subito essere $LL(1)$.

2. Si consideri la grammatica G seguente, in forma estesa (EBNF):

$$S \;\rightarrow\; \Big(\; a\; S\; (\; c \;\mid\; d\;)\;\Big)^*$$

Si vuole costruire un analizzatore sintattico di tipo $LR(1)$ per il linguaggio $L = L(G)$, osservando che il linguaggio L è simile a un linguaggio di Dyck.

Si risponda alle domande seguenti:

(a) Si costruisca una grammatica G' equivalente a G, adatta al metodo $LR(1)$.

(b) Si verifichi che la grammatica G' sia di tipo $LR(1)$, costruendo l'automa pilota.

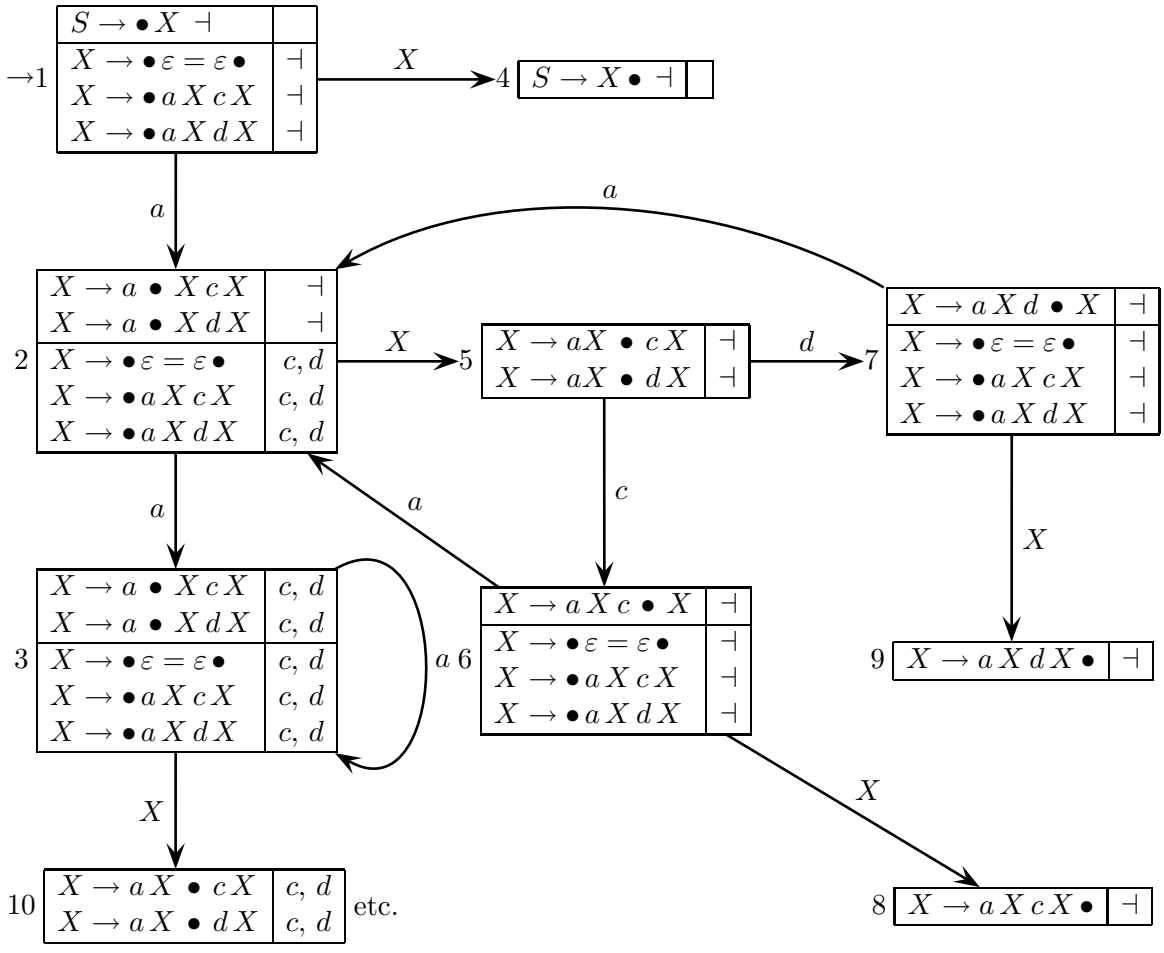
Soluzione

(a) Since G is an extended BNF grammar and the direct construction of an $LR(1)$ parser for such grammars (see the textbook) is not included in the course program, it is necessary to transform the grammar into a non-extended one:

$$S \to X \qquad X \to \varepsilon \mid a\ X\ c\ X \mid a\ X\ d\ X$$

This is nothing but a standard Dyck grammar, not ambiguous, with two equivalent closed brackets c and d.

(b) Ecco l'automa pilota, solo una parte il resto essendo piuttosto ovvio:



L'automa pilota è senza conflitti, dunque la grammatica è $LR(1)$.

4 Traduzione e analisi semantica 20%

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

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

La grammatica G non è ambigua e genera il linguaggio delle stringhe $w = a^h\ c^k\ d^k\ b^h$, con esponenti $h, k \geq 0$.

Si risponda alle domande seguenti:

(a) Modificando opportunamente la grammatica sorgente G, si scriva lo schema (o la grammatica) di traduzione sintattica G', non ambiguo, della trasduzione τ che traduce la stringa w nella stringa $a^h\ d^k\ c^k\ b^h$ se l'esponente h è dispari, altrimenti emette la stringa w inalterata. Ecco due esempi:

$$\tau\left(\ a^3\ c^4\ d^4\ b^3\right) = a^3\ d^4\ c^4\ b^3$$

$$\tau\left(\ a^2\ c^4\ d^4\ b^2\right) = a^2\ c^4\ d^4\ b^2$$

(b) Si argomenti se la traduzione inversa τ^{-1} è una funzione o no, e in caso affermativo se ne dia uno schema (o una grammatica) di traduzione.

Soluzione

(a) Si fa sdoppiando il nonterminale S così da distinguere tra esponente h pari e dispari, e a seguire sdoppiando il nonterminale T così da generare separatamente strutture annidate $c\,d$ e $d\,c$. Ecco lo schema di traduzione G' (assioma S_0):

$$G' \left\{ \begin{array}{lcl|lcl} S_0 & \to & a\ S_1\ b & S_0 & \to & a\ S_1\ b \\ S_1 & \to & a\ S_0\ b & S_1 & \to & a\ S_0\ b \\ S_0 & \to & T_0 & S_0 & \to & T_0 \\ S_1 & \to & T_1 & S_1 & \to & T_1 \\ T_0 & \to & c\ T_0\ d & T_0 & \to & c\ T_0\ d \\ T_1 & \to & c\ T_1\ d & T_1 & \to & d\ T_1\ c \\ T_0 & \to & \varepsilon & T_0 & \to & \varepsilon \\ T_1 & \to & \varepsilon & T_1 & \to & \varepsilon \end{array} \right.$$

oppure la grammatica di traduzione:

$$G' \left\{ \begin{array}{lcl} S_0 & \to & a\ \{\ a\ \}\ S_1\ b\ \{\ b\ \} \\ S_1 & \to & a\ \{\ a\ \}\ S_0\ b\ \{\ b\ \} \\ S_0 & \to & T_0 \\ S_1 & \to & T_1 \\ T_0 & \to & c\ \{\ c\ \}\ T_0\ d\ \{\ d\ \} \\ T_1 & \to & c\ \{\ d\ \}\ T_1\ d\ \{\ c\ \} \\ T_0 & \to & \varepsilon\ \{\ \varepsilon\ \} \\ T_1 & \to & \varepsilon\ \{\ \varepsilon\ \} \end{array} \right.$$

ottenuta sovrapponendo grammatica sorgente e destinazione.

C'è anche la variante seguente, piuttosto simile, qui data ancora come grammatica di traduzione (assioma S_0):

$$G' \left\{ \begin{array}{lcl} S_0 & \to & a\ a\ \{\ a\ a\ \}\ S_0\ b\ b\ \{\ b\ b\ \}\ \mid\ S_1\ \mid\ T_0 \\ S_1 & \to & a\ \{\ a\ \}\ T_1\ b\ \{\ b\ \} \\ T_0 & \to & c\ \{\ c\ \}\ T_0\ d\ \{\ d\ \}\ \mid\ \varepsilon\ \{\ \varepsilon\ \} \\ T_1 & \to & c\ \{\ d\ \}\ T_1\ d\ \{\ c\ \}\ \mid\ \varepsilon\ \{\ \varepsilon\ \} \end{array} \right.$$

che naturalmente si potrebbe riscrivere come schema di traduzione.

(b) La grammatica sorgente non è ambigua: essa è ottenuta dalla grammatica G data nel testo, non ambigua, tramite operazioni che di certo non introducono ambiguità, poiché si va semplicemente a separare derivazioni già esistenti. Dunque la traduzione τ è una funzione, poiché la stringa sorgente, avendo una sola derivazione, ha ovviamente una sola stringa destinazione che le corrisponde.

Resta però da stabilire se una stringa destinazione possa provenire da due diverse stringhe sorgente. Ora si nota che la grammatica destinazione è strutturalmente identica a quella sorgente, non solo nella disposizione dei nonterminali (il che vale per definizione giacché è uno schema sintattico) ma anche nella disposizione dei terminali, tranne lo scambio puramente nominale tra c e d in una regola, ciò che certamente non rende ambigua la grammatica destinazione. Pertanto la traduzione inversa τ^{-1}, che si ottiene scambiando grammatica sorgente e destinazione, è pure una funzione. Eccone lo schema sintattico (assioma S_0):

$$G' \left\{ \begin{array}{lcl|lcl} S_0 & \to & a\ S_1\ b & S_0 & \to & a\ S_1\ b \\ S_1 & \to & a\ S_0\ b & S_1 & \to & a\ S_0\ b \\ S_0 & \to & T_0 & S_0 & \to & T_0 \\ S_1 & \to & T_1 & S_1 & \to & T_1 \\ T_0 & \to & c\ T_0\ d & T_0 & \to & c\ T_0\ d \\ T_1 & \to & d\ T_1\ c & T_1 & \to & c\ T_1\ d \\ T_0 & \to & \varepsilon & T_0 & \to & \varepsilon \\ T_1 & \to & \varepsilon & T_1 & \to & \varepsilon \end{array} \right.$$

E di conseguenza la traduzione τ è una funzione invertibile, dunque biunivoca.

Osservazione: la grammatica G data nel testo, per svista è leggermente ambigua: si può derivare $S \Longrightarrow T \Longrightarrow \varepsilon$ oppure $S \Longrightarrow \varepsilon$. Ciò non ha conseguenze sulla traduzione τ, che è comunque una funzione. Del resto basta togliere da G la regola $S \to \varepsilon$ (refuso della preparazione del testo), del tutto superflua, per avere una grammatica equivalente non ambigua. Gli schemi sintattici di traduzione dati nella soluzione, come anche le grammatiche di traduzione, sono tutti non ambigui.

2. Si deve calcolare il valore logico di una proposizione contenente le costanti True (T) e False (F), le due variabili x e y, i connettivi "$\wedge$" e "$\neg$" (ossia and e not), e le parentesi per imporre un ordine al calcolo. La sintassi delle proposizioni è la seguente:

$$\begin{cases} P & \rightarrow & (\, P \wedge P \,) \mid (\, \neg P \,) \mid \langle \mathsf{const} \rangle \mid \langle \mathsf{var} \rangle \\ \langle \mathsf{const} \rangle & \rightarrow & \mathsf{T} \mid \mathsf{F} \\ \langle \mathsf{var} \rangle & \rightarrow & x \mid y \end{cases}$$

Esempi:

$$\Big(\big(\, (\, \mathsf{T} \wedge y \,) \wedge (\, \neg x \,) \, \big) \wedge y \Big) \qquad\qquad \big(\, (\, \mathsf{T} \wedge \mathsf{F} \,) \wedge \mathsf{F} \, \big)$$

Per inizializzare le variabili, prima della proposizione P ci può essere una lista I di assegnamenti A, secondo la sintassi seguente:

$$\begin{cases} I & \rightarrow & A \; I \mid \varepsilon \\ A & \rightarrow & x = \langle \mathsf{const} \rangle \mid y = \langle \mathsf{const} \rangle \end{cases}$$

Il linguaggio completo è allora definito dalla regola sintattica seguente (assioma S):

$$S \rightarrow I \; P$$

Esempi completi di frasi del linguaggio e del loro valore *val* da calcolare:

$x = \mathsf{T}$	$\Big(\big(\, (\, \mathsf{T} \wedge y \,) \, \big) \wedge y \Big)$	$val = \perp$ (undef.)
$x = \mathsf{F}$	$\big(\, (\, \mathsf{T} \wedge \mathsf{F} \,) \wedge \mathsf{T} \, \big)$	$val = \mathsf{F}$
$x = \mathsf{T} \;\; y = \mathsf{F} \;\; x = \mathsf{F}$	$(\, \neg x \,)$	$val = \mathsf{T}$
$x = \mathsf{T} \;\; y = \mathsf{T} \;\; x = \mathsf{F}$	$\big(\, (\, y \wedge \mathsf{T} \,) \wedge (\, \neg x \,) \, \big)$	$val = \mathsf{T}$

Se una qualsiasi variabile "var" che compare nella proposizione non è inizializzata, la proposizione vale "$\perp$" (per rappresentare lo stato d'indefinizione).

Si risponda alle domande seguenti:

(a) Si progetti una grammatica con attributi per calcolare il valore della proposizione, cioè l'attributo sinistro *val*. Si dica degli altri eventuali attributi quali sono sinistri e quali destri. Allo scopo si usino le tabelle alle pagine seguenti.

(b) (facoltativa) Si esamini se la grammatica con attributi è del tipo a una scansione (one sweep).

attributi da usare per la grammatica

tipo	nome	(non)terminali	dominio	significato
sin	*val*	$S, \ldots$	booleano	valore logico della proposizione

sintassi	calcolo attributi - 1
1: $S_0 \rightarrow I_1\ P_2$	
2: $I_0 \rightarrow A_1\ I_2$	
3: $I_0 \rightarrow \varepsilon$	
4: $A_0 \rightarrow x = \langle \mathsf{const} \rangle_1$	
5: $A_0 \rightarrow y = \langle \mathsf{const} \rangle_1$	

sintassi	calcolo attributi - 2
6: $P_0 \rightarrow (P_1 \wedge P_2)$	
7: $P_0 \rightarrow (\neg P_1)$	
8: $P_0 \rightarrow \langle \mathsf{const} \rangle_1$	
9: $P_0 \rightarrow \langle \mathsf{var} \rangle_1$	

sintassi	calcolo attributi - 3
10: $\langle \mathsf{const} \rangle_0 \rightarrow \mathsf{T}$	
11: $\langle \mathsf{const} \rangle_0 \rightarrow \mathsf{F}$	
12: $\langle \mathsf{var} \rangle_0 \rightarrow x$	
13: $\langle \mathsf{var} \rangle_0 \rightarrow y$	

Soluzione

(a) Idea: raccogliere in due attributi sinistri val_x e val_y i valori di inizializzazione delle due variabili x e y, rispettivamente; memorizzare in due attributi destri asg_x e asg_y i valori di inizializzazione delle due variabili x e y, rispettivamente, e propagarli alla proposizione; e infine calcolare in un attributo sinistro val (parzialmente dato nel testo) il valore logico della proposizione.

attributi da usare per la grammatica

tipo	nome	(non)terminali	dominio	significato
sin	val	S, P, $\langle$const$\rangle$, $\langle$var$\rangle$	booleano	valore logico della proposizione, della costante o della variabile
sin	val_x	I, A	booleano	ultimo valore logico (oppure indefinito), ossia quello più a destra, assegnato alla variabile x
sin	val_y	I, A	booleano	ultimo valore logico (oppure indefinito), ossia quello più a destra, assegnato alla variabile y
des	asg_x	P, $\langle$var$\rangle$	booleano	valore logico assegnato alla variabile x, da propagare alla proposizione per la valutazione
des	asg_y	P, $\langle$var$\rangle$	booleano	valore logico assegnato alla variabile y, da propagare alla proposizione per la valutazione

Nota bene: gli attributi dx asg_x e sx val_x sono associati a nonterminali disgiunti, rispettivamente I, A e $P, \langle$var$\rangle$; pertanto in teoria questi due attributi sono riunibili in un unico attributo di tipo misto che si comporta come dx in I, A e come sx in $P, \langle$var$\rangle$, ossia viene sintetizzato ed ereditato in regole diverse ovvero in parti disgiunte dell'albero sintattico, dunque senza causare confusione semantica. Osservazione simile vale per gli attributi dx asg_y e sx val_y.

Tuttavia la convenzione delle grammatiche con attributi prescrive, come buona regola di programmazione pulita, di non definire mai attributi misti, neppure quando sarebbe possibile, onde rendere ben leggibile la grammatica ed evitare a priori situazioni che, seppure in certi casi speciali sarebbero comunque corrette, troppo spesso facilitano gli errori semantici invece di tendere a prevenirli.

(b) Ecco la semantica:

	sintassi	calcolo attributi - 0 - radice S
1:	$S_0 \to I_1\ P_2$	$-$eredità assegnamenti $asg\_x_2 = val\_x_1$ $asg\_y_2 = val\_y_1$ $-$sintesi valore $val_0 = val_2$

	sintassi	calcolo attributi - 1 - parte sx albero radice I	

2:	$I_0 \to A_1\ I_2$	$-$sintesi assegnamenti
		$-$prevale asg. a dx
		if $(val\_x_2 \neq \perp)$ **then**
		$val\_x_0 = val\_x_2$
		else
		$val\_x_0 = val\_x_1$
		endif
			$-$sintesi assegnamenti
			$-$prevale asg. a dx
			if $(val\_y_2 \neq \perp)$ **then**
			$val\_y_0 = val\_y_2$
			else
			$val\_y_0 = val\_y_1$
			endif

3:	$I_0 \to \varepsilon$	$-$sintesi assegnamenti $val\_x_0 = \perp$ $val\_y_0 = \perp$	
4:	$A_0 \to x = \langle \mathsf{const} \rangle_1$	$-$sintesi assegnamenti $val\_x_0 = val_1$ $val\_y_0 = \perp$	
5:	$A_0 \to y = \langle \mathsf{const} \rangle_1$	$-$sintesi assegnamenti $val\_x_0 = \perp$ $val\_y_0 = val_1$	

sintassi	calcolo attributi - 2 - parte dx albero radice P

6: $P_0 \rightarrow (\, P_1 \wedge P_2 \,)$	$-$eredità assegnamenti
	$asg\_x_1 = asg\_x_0$ $\quad$ $asg\_x_2 = asg\_x_0$
	$asg\_y_1 = asg\_y_0$ $\quad$ $asg\_y_2 = asg\_y_0$
	$-$sintesi valore
	if $(val_1 \neq \perp$ **and** $val_2 \neq \perp)$ **then**
	$\quad val_0 = val_1 \wedge val_2$
	else
	$\quad val_0 = \perp$
	endif

7: $P_0 \rightarrow (\, \neg P_1 \,)$	$-$eredità assegnamenti
	$asg\_x_1 = asg\_x_0$
	$asg\_y_1 = asg\_y_0$
	$-$sintesi valore
	if $(val_1 \neq \perp)$ **then**
	$\quad val_0 = \neg val_1$
	else
	$\quad val_0 = \perp$
	endif

8: $P_0 \rightarrow \langle \mathsf{const} \rangle_1$	$val_0 = val_1$ $\quad -$sintesi valore
9: $P_0 \rightarrow \langle \mathsf{var} \rangle_1$	$-$eredità assegnamenti $asg\_x_1 = asg\_x_0$ $asg\_y_1 = asg\_y_0$ $-$sintesi valore $val_0 = val_1$

	sintassi	calcolo attributi - 3 - foglie albero
10:	$\langle \mathsf{const} \rangle_0 \to \mathsf{T}$	$val_0 = \mathsf{T}$ −sintesi valore
11:	$\langle \mathsf{const} \rangle_0 \to \mathsf{F}$	$val_0 = \mathsf{F}$ −sintesi valore
12:	$\langle \mathsf{var} \rangle_0 \to x$	$val_0 = asg\_x_0$ −sintesi valore
13:	$\langle \mathsf{var} \rangle_0 \to y$	$val_0 = asg\_y_0$ −sintesi valore

Se si generalizza la definizione degli operatori "$\wedge$" e "$\neg$", ossia and e not, così che se almeno uno degli operandi è indefinito ($\perp$) allora pure il risultato è indefinito, si semplifica la codifica delle funzioni semantiche riducendo le clausole `if-then-else` nelle regole 6 e 7, seppure le dipendenze funzionali tra gli attributi restano le stesse. Qui si è preferito scrivere in modo esplicito per non lasciare dubbi.

Si badi bene che il condizionale semantico `if-then-else` nella regola 2 è invece essenziale poiché esprime la prevalenza dell'assegnamento più a destra ossia l'ultimo, qualora una variabile venga assegnata due o più volte.

Come osservato prima, in teoria gli attributi asg_x e val_x sono unificabili; infatti si vede che p. es. nelle regole dove compare $asg\_x_1 = \ldots$ o $asg\_x_2 = \ldots$, non si trova mai $val\_x_0 = \ldots$; e similmente per asg_y e val_y; pertanto un solo attributo misto (uno per x e uno per y) non causerebbe conflitto semantico tra eredità e sintesi; tuttavia le funzioni semantiche rimarrebbero le stesse; e in ogni caso si prescrive di non ricorrere ad attributi misti, onde evitare dubbi e confusioni.

(c) Si noti che la parte di albero, a sx, per inizializzare le variabili è disgiunta dalla parte di albero, a dx, per la proposizione.

La grammatica con attributi è a una passata (one sweep): nella parte sx dell'albero con radice I, si raccolgono bottom-up le inizializzazioni delle variabili tramite gli attributi sx val_x e val_y; nella parte dx dell'albero con radice P, le si propagano top-down alla proposizione tramite gli attributi dx asg_x e asg_y; e nella parte dx dell'albero con radice P, si valuta bottom-up la proposizione tramite l'attributo sx val.

Pertanto il calcolo è solo bottom-up negli assegnamenti, e prima top-down e poi bottom-up nella proposizione, dunque complessivamente è fattibile in una passata. Ciò è evidente tracciando le dipendenze funzionali oppure applicando la condizione L.

Osservazione: si può pensare a una variante con tabella globale delle variabili. La tabella globale è creata vuota nella radice dell'albero sintattico e quivi mantenuta. La gestione della tabella è realizzata mediante puntatori ereditati e apposite procedure o funzioni d'inserimento (per l'assegnamento della variabile) e consultazione (per l'uso della variabile). Il calcolo del valore procede come nella soluzione precedente. Come dipendenze, tale variante è sostanzialmente analoga alla soluzione precedente, ma evita la risalita dei valori assegnati alle variabili nella parte di albero che modella l'assegnamento e la discesa degli stessi nella parte di albero che modella la proposizione. Eccola:

attributi da usare per la grammatica

tipo	nome	(non)terminali	dominio	significato
sin	*val*	S, P, ⟨const⟩, ⟨var⟩	booleano	valore logico della proposizione, della costante o della variabile, da calcolare in salita nel sottoalbero della proposizione
des	*var_tab*	T	tabella	tabella globale delle variabili - formalmente è un attributo destro - è centralizzata nella radice dell'albero e gestita ai livelli inferiori tramite puntatore - è associata a un nonterminale di servizio T, qui non espanso - è creata all'inizio, in quanto attributo destro
des	p	I, A, P, ⟨var⟩	puntatore	puntatore alla tabella delle variabili, da propagare in discesa al sottoalbero di assegnamento e al sottoalbero della proposizione

Le procedure e funzioni per gestire la tabella e di supporto, sono le seguenti:

- **create** ( ) $\mapsto$ *var_tab*, funzione senza argomenti che crea la tabella delle variabili *var_tab*, inizialmente vuota
- **put** (p, *name*, *value*), procedura che inserise la variabile *name* inizializzata con valore logico *value* nella tabella puntata da p, eventualmente sovrascrivendo l'assegnamento precedente alla stessa variabile
- **get** (p, *name*) $\mapsto$ *value*, funzione che consulta la tabella puntata da p e restituisce il valore logico *value* assegnato alla variabile *name* o restituisce valore indefinito $\perp$ se la variabile non è in tabella
- inoltre le funzioni di supporto **gen_and** e **gen_not** sono versioni generalizzate degli operatori and e not, rispettivamente, che lavorano come and e not ordinari se gli argomenti sono tutti definiti e restituiscono valore indefinito $\perp$ se gli argomenti non sono tutti definiti

	sintassi	calcolo attributi	
1:	$S_0 \rightarrow I_1\ P_2\ T_3$	$var\_tab_3 =$ **create** ( ) $p_1,\ p_2 = \&\, var\_tab_3$ $val_0 = val_2$	−eredità −eredità −sintesi
2:	$I_0 \rightarrow A_1\ I_2$	$p_1,\ p_2 = p_0$	−eredità
3:	$I_0 \rightarrow \varepsilon$		
4:	$A_0 \rightarrow x = \langle \mathsf{const} \rangle_1$	**put** (p_0, 'x', val_1)	
5:	$A_0 \rightarrow y = \langle \mathsf{const} \rangle_1$	**put** (p_0, 'y', val_1)	
6:	$P_0 \rightarrow (\ P_1\ \wedge\ P_2\ )$	$p_1,\ p_2 = p_0$ $val_0 =$ **gen_and** (val_1, val_2)	−eredità −sintesi
7:	$P_0 \rightarrow (\ \neg\, P_1\ )$	$p_1 = p_0$ $val_0 =$ **gen_not** (val_1)	−eredità −sintesi
8:	$P_0 \rightarrow \langle \mathsf{const} \rangle_1$	$val_0 = val_1$	−sintesi
9:	$P_0 \rightarrow \langle \mathsf{var} \rangle_1$	$p_1 = p_0$ $val_0 = val_1$	−eredità −sintesi
10:	$\langle \mathsf{const} \rangle_0 \rightarrow \mathsf{T}$	$val_0 = \mathsf{T}$	−sintesi
11:	$\langle \mathsf{const} \rangle_0 \rightarrow \mathsf{F}$	$val_0 = \mathsf{F}$	−sintesi
12:	$\langle \mathsf{var} \rangle_0 \rightarrow x$	$val_0 =$ **get** (p_0, 'x')	−sintesi
13:	$\langle \mathsf{var} \rangle_0 \rightarrow y$	$val_0 =$ **get** (p_0, 'y')	−sintesi

La grammatica è una passata: top-down la propagazione dei puntatori e bottom-up il calcolo del valore; il flusso degli assegnamenti è centralizzato nella tabella delle variabili. Questa soluzione è più programmativa della precedente ed è compatta, poiché ricorre a nozioni di progetto dei compilatori (nonché di programmazione a oggetti od object-oriented) che, se messe bene a fuoco sul problema specifico, forniscono una soluzione precostituita e sintetica a situazioni frequenti.