Esercizi, blocco 3

Nel seguito, dati

- lo stato P di un automa deterministico A
- la stringa $Y_1 \dots Y_n$

si indica con $P[\![Y_1 \dots Y_n]\!]$ lo stato di A che si raggiunge da P tramite il cammino $Y_1 \dots Y_n$.

$\mathcal{S}_1$: Sia $\mathcal{S}_1$ il seguente SDD:

$$\begin{array}{lcll} S & \to & E & \{S.v = E.v;\} \\ E & \to & n & \{E.v = n.lexval;\} \\ E & \to & E_1\, a\, E_2 & \{E.v = E_1.v * E_2.v;\} \\ E & \to & E_1\, b\, E_2 & \{E.v = E_1.v + E_2.v;\} \end{array}$$

Esercizio 1

Sia P lo stato iniziale del parser LALR(1) per la grammatica dello SDD $\mathcal{S}_1$. Il parser ha 4 conflitti shift/reduce: uno in $[P[\![EaE]\!], a]$, uno in $[P[\![EaE]\!], b]$, uno in $[P[\![EbE]\!], a]$ e uno in $[P[\![EbE]\!], b]$. Supponiamo che tutti e 4 i conflitti siano risolti a favore dello shift. Supponiamo inoltre che l'attributo $n.lexval$ del terminale n sia il numero intero rappresentato da n. Se l'input $2a3b4$ non è riconosciuto, rispondere "ERROR". Altrimenti dire quale valore viene valutato per $S.v$ su input $2a3b4$.

Esercizio 2

Sia P lo stato iniziale del parser LALR(1) per la grammatica dello SDD $\mathcal{S}_1$. Il parser ha 4 conflitti shift/reduce: uno in $[P[\![EaE]\!], a]$, uno in $[P[\![EaE]\!], b]$, uno in $[P[\![EbE]\!], a]$ e uno in $[P[\![EbE]\!], b]$. Alcuni di questi conflitti dipendono dal fatto che la grammatica non modella la precedenza dell'operatore di moltiplicazione (operatore a) sull'operatore di somma (operatore b). Si dica quali conflitti sono dovuti alla suddetta carenza della grammatica e si dica come risolvere ciascuno di essi per fare in modo che a abbia precedenza su b.

Esercizio 3

Sia $\mathcal{S}_{1b}$ il seguente SDD:

$$\begin{array}{lcll} S & \to & A & \{while(T\,not\,empty)\{num = pop(T); print(num,);\}\} \\ A & \to & CB & \{push(T,1);\} \\ B & \to & aCB & \{push(T,2);\} \\ B & \to & \epsilon & \{push(T,3);\} \\ C & \to & ED & \{push(T,4);\} \\ D & \to & bED & \{push(T,5);\} \\ D & \to & \epsilon & \{push(T,6);\} \\ E & \to & g & \{push(T,7);\} \end{array}$$

dove T è una pila inizialmente vuota. Sulla pila sono definite le comuni funzioni di $push(pila, elemento)$ e $pop(pila)$. Si immagini di analizzare $\mathcal{S}_{1b}$ con un parser LALR(1). Se l'input $gagbg$ non è riconosciuto dal parser scrivere "ERROR". Altrimenti scrivere il risultato della valutazione dell'input $gagbg$.

Esercizio 4

Sia $\mathcal{G}$ la seguente grammatica ambigua per un linguaggio con identificatori id e operatori binari a e b

$$S \to S\ a\ S \mid S\ b\ S \mid (S) \mid id$$

Fornire una grammatica LL(1) per la generazione di $\mathcal{L}(\mathcal{G})$ in cui l'ambiguità è risolta rispettando le seguenti convenzioni: l'operatore a ha precedenza sull'operatore b; entrambi gli operatori associano a sinistra.

Esercizio 5

Sia $\mathcal{D}$ la seguente porzione di *syntax directed translation*:

$$\begin{array}{lcll} P & \to & S & \{ \quad S.next = newlabel() \\ & & & \quad\ P.code = S.code \triangleright label(S.next) \quad \} \\ S & \to & \textsf{for}\ (S_1\ ;\ B\ ;\ S_2)\ S_3 & \end{array}$$

Assumendo che:

- B è gestita con gli usuali attributi $B.code$, $B.true$ e $B.false$
- la semantica del comando "for $(S_1\ ;\ B\ ;\ S_2)\ S_3$" è la stessa di "S_1 ; while (B) $\{S_3\ ;\ S_2;\}$;"

dire quali regole semantiche vanno associate all'ultima produzione per ottenere la traduzione del for-statement.

Esercizio 6

Sia data la seguente *syntax-directed translation* per array

$S \to id = E$	$gen(table.get(id)\ '='\ E.addr)$
$S \to L = E$	$gen(L.array\_base\ '['\ L.addr\ ']'\ '='\ E.addr)$
$E \to E_1 + E_2$	$E.addr = newtemp()$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$E \to id$	$E.addr = table.get(id)$ $E.code = ''$
$E \to L$	$E.addr = newtemp()$ $gen(E.addr\ '='\ L.array.base\ '['L.addr']')$
$L \to id[E]$	$L.array = table.get(id)$ $L.type = arg2(table.getType(id))$ $L.width = width(L.type)$ $L.addr = newtemp()$ $gen(L.addr\ '='\ E.addr\ '*'\ L.width)$
$L \to L_1[E]$	$L.array = L_1.array$ $L.type = arg2(L_1.type)$ $L.width = width(L.type)$ $t = newtemp()$ $gen(t\ '='\ E.addr\ '*'\ L.width)$ $L.addr = newtemp()$ $gen(L.addr\ '='\ L_1.addr\ '+'\ t)$

Si assumano le seguenti condizioni: il tipo di a è $array(2, array(3, integer))$; la *base* di a è 0; c, i, j sono interi; la dimensione di un intero è 4. Si assuma inoltre di risolvere l'ambiguità della grammatica assegnando all'operatore di somma la usuale associtività a sinistra. Dire quale codice viene generato nell'analisi bottom-up della stringa

$$b = c + a[i][j]$$