Linguaggi Formali e Traduttori

- traduzione diretta dalla sintassi -

Alcuni esercizi risolti

1. Data la grammatica con il seguente insieme di produzioni:

   $S \to$ **(**L**)** | a

   $L \to$ L**,**S | S

   definire le opportune azioni semantiche per calcolare, per ciascuna stringa del linguaggio, il **numero di coppie di parentesi** presenti nella stringa.

Regole sintattiche	*Regole semantiche*
$S \to$ **(**L**)**	S.*npar* = L.*npar* + 1
$S \to$ a	S.*npar* = 0
$L \to L_1$**,**S	L.*npar* = L_1.*npar* + S.*npar*
$L \to$ S	L.*npar* = S.*npar*

3. Data la grammatica con il seguente insieme di produzioni:

   $S \to RA \mid A[S]$ $\quad R \to E = E$

   $E \to (E+E) \mid a$ $\quad A \to bA \mid \varepsilon$

   a) Calcolare gli insiemi guida delle produzioni, a partire dalla loro definizione, indicando i passaggi del calcolo;

$S \to RA$	F(RA) = F(R) = F(E = E) = F(E) = {(, a}
$S \to A[S]$	F(A[S]) = F(A) – {ε} ∪ F([S]) = {b, [}
$R \to E = E$	F(E = E) = F(E) = {(, a}
$E \to (E+E)$	F((E + E)) = {(}
$E \to a$	{a}
$A \to bA$	{b}
$A \to \varepsilon$	F(ε) – {ε} ∪ FW(A) = {[} ∪ FW(S) = {[,], $}

Gui($S \to RA$) ∩ Gui($S \to A[S]$) = Φ

Gui($E \to (E+E)$) ∩ Gui($E \to a$) = Φ

Gui($A \to bA$) ∩ Gui($A \to \varepsilon$) = Φ

La grammatica è LL(1)

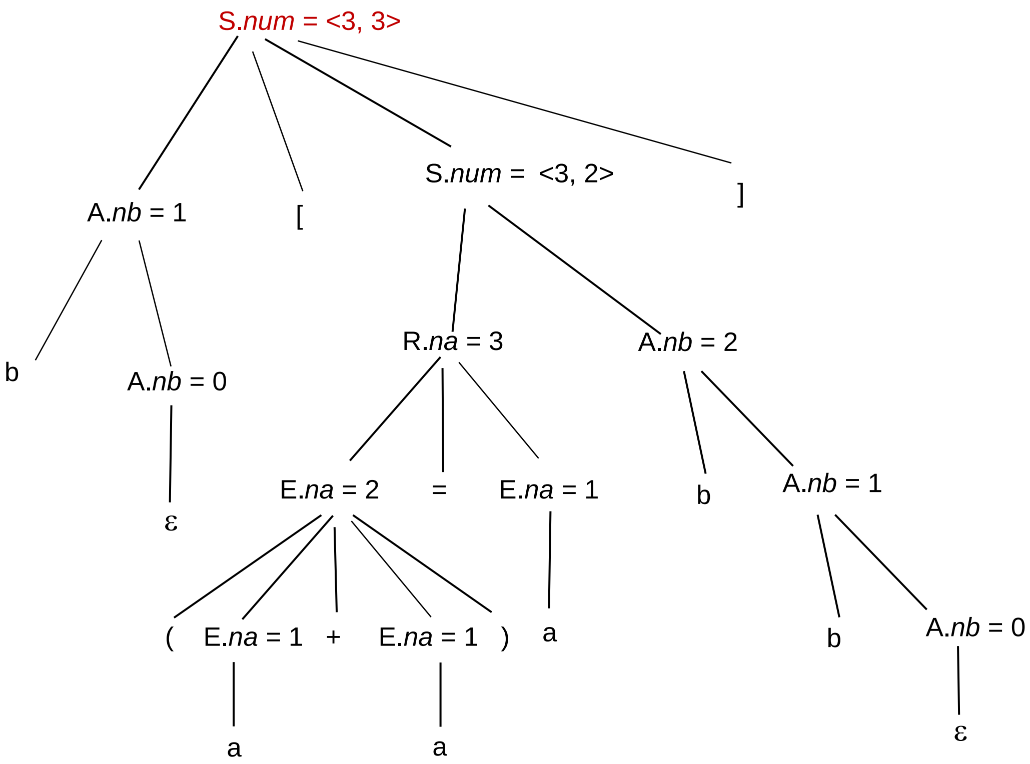
b) Se la grammatica risulta LL(1), scrivere la procedura di analisi a discesa ricorsiva per lo start symbol;

```
function S( )
 if (cc ∈ {‘a’, ‘(’})
        R( )
        A( )
 elseif (cc ∈ {‘b’, ‘[’})
        A( )
        if (cc = ‘[’) cc ← PROSS
        else ERRORE (…)
        S( )
        if (cc = ‘]’) cc ← PROSS
        else ERRORE (…)
 else ERRORE (…)
```

c) Attribuire la grammatica in modo da calcolare il numero di a e il numero di b complessivamente presenti in ciascuna stringa del linguaggio e mostrare un esempio di albero annotato.

- La variabile E, e quindi anche la variabile R, generano stringhe formate solo da a, mentre la variabile A genera stringhe di soli b. Allora si può definire un attributo *.na* per E ed R e un attributo *.nb* per A. Solo per S, che genera stringhe di a e di b, si deve definire l'attributo *.num* come coppia del numero di a e di b.
- p_1 e p_2 sono le funzioni che estraggono rispettivamente la prima e la seconda componente di una coppia (proiezioni).

$S \to RA$	$S.num = \langle R.na, A.nb \rangle$
$S \to A[S_1]$	$S.num = \langle p_1(S_1.num), p_2(S_1.num) + A.nb \rangle$
$R \to E_1 = E_2$	$R.na = E_1.na + E_2.na$
$E \to (E_1+E_2)$	$E.na = E_1.na + E_2.na$
$E \to a$	$E.na = 1$
$A \to bA_1$	$A.nb = A_1.nb + 1$
$A \to \varepsilon$	$A.nb = 0$



5. Dato il seguente schema di traduzione:

$L \rightarrow \{\ S.pc = 1\ \}\ S\ \{\ T.pc = 2\ \}\ T\ \{\ L.out = S.out \parallel T.out\ \}$

$T \rightarrow \{\ S.pc = T.pc\ \} S\ \{\ T_1.pc = T.pc + 1\ \}\ T_1\ \{\ T.out = S.out \parallel T_1.out\ \}$

$T \rightarrow \varepsilon\ \ \{\ T.out = \text{‘ ’}\ \}$

$S \rightarrow$ **id** := E; { S.*out* = S.*pc* || **id**.*name* || ‘:=’ || E.*val* }

$E \rightarrow$ **num** G { E.*val* = **num**.*val* + G.*val* }

$G \rightarrow$ **+ num** G_1 { G.*val* = **num**.*val* + G_1.*val* }

$G \rightarrow \varepsilon\ \ \{\ G.val = 0\ \}$

in cui || è l’operatore di concatenazione di stringhe:

– dire quali attributi sono ereditati e quali sono sintetizzati;

.pc è un attributo ereditato
.val e *.out* sono sintetizzati

– scrivere il traduttore a discesa ricorsiva;

```
Program traduzione( )
  var TR
    cc ← PROSS( )
    TR ← L( )
    if (cc = '$') scrivi ("stringa corretta",
                  "la sua traduzione è" TR)
    else ERRORE (…)
```

```
function L( )
  var Spc, Sout, Tpc, Tout, Lout
   if (cc = 'id')
        Spc ← 1
        Sout ← S(Spc)
        Tpc ← 2
        Tout ← T(Tpc)
        Lout ← Sout || Tout
   else ERRORE (…)
   return (Lout)
```

```
function T(Tpc )
  var Spc, Sout, T1pc, T1out, Tout
   if (cc = 'id')
        Spc ← Tpc
        Sout ← S(Spc)
        T1pc ← Tpc + 1
        T1out ← T(T1pc)
        Tout ← Sout || T1out
   elseif (cc = '$') Tout ← ε
   else ERRORE (…)
   return (Tout)
```

```
function S(Spc)
  var Eval, Sout, Nome
   if (cc = 'id') Nome ← id.name
              cc ← PROSS( )
              if (cc = ':=') cc ← PROSS( )
              else ERRORE (…)
              Eval ← E( )
              if (cc = ';') cc ← PROSS( )
              else ERRORE (…)
              Sout ← Spc || Nome || ':=' || Eval
   else ERRORE (…)
   return (Sout)
```

```
function G( )
  var Gval, G1val, V
   if (cc = '+') cc ← PROSS( )
              if (cc = 'num') V ← num.val
                        cc ← PROSS( )
              else ERRORE (…)
              G1val ← G( )
              Gval ← V + G1val
   elseif (cc = ';') Gval ← 0
   else ERRORE (…)
   return (Gval)
```

```
function E( )
  var Gval, Eval, V
   if (cc = 'num') V ← num.val
              cc ← PROSS( )
              Gval ← G( )
              Eval ← V + Gval
   else ERRORE (…)
```

6. Scrivere il traduttore deterministico top-down per il seguente schema di traduzione:

$A \rightarrow bA_1A_2 \; \{A.num = A_1.num + A_2.num + <0,1,0>\}$

$A \rightarrow cA_1A_2 \; \{A.num = A_1.num + A_2.num + <0,0,1>\}$

$A \rightarrow a \; \{A.num = <1,0,0>\}$

che associa ad ogni stringa generata dalla grammatica la terna formata dal numero di *a*, di *b* e di *c* di cui è formata la stringa.

N.B. Definiamo la somma tra k n-ple $A_1, A_2, \ldots, A_k$ come la n-pla ottenuta sommando $A_1, A_2, \ldots, A_k$ componente per componente. Ad esempio: $<5,3,4> + <1,0,7> = <6,3,11>$.

```
function A( )
 var A_n, A1_n, A2_n
   if (cc = 'a')  cc ← PROSS( )
                  A_n ← <1,0,0>
   elseif (cc = 'b') cc ← PROSS( )
                  A1_n  ← A( )
                  A2_n ← A( )
                  A_n ← A1_n + A2_n + <0,1,0>
   elseif (cc = 'c') cc ← PROSS( )
                  A1_n  ← A( )
                  A2_n ← A( )
                  A_n ← A1_n + A2_n + <0,0,1>
   else ERRORE (…)
   return (A_n)
```

7. Data la grammatica con l'insieme di produzione {$S \rightarrow E$, $E \rightarrow$ **id** E', E' $\rightarrow$ + **id** E' | ε}
   a) attribuirla in modo che, dato in input un insieme L di identificatori, la traduzione associ ad ogni parola generata da S, "true" se e solo se la parola contiene identificatori tutti appartenenti all'insieme L, "false" altrimenti. Si usi un attributo S.*set* per memorizzare l'insieme L e si supponga di disporre di una funzione *in*(x,I) che verifica se l'elemento x appartiene all'insieme I. Usare l'attributo S.*ok* per il risultato della traduzione.
   Per esempio, dato L = {A, B}, e l'espressione A+B+A, il risultato sarà true, mentre sarà false per l'espressione A+B+C.

S → {E.*set* = S.*set*) E {S.*ok* = E.*ok*}

E → **id** {E'.*set* = E.*set*} E' { E.*ok* = *in*(**id**.*lexval*, E.*set*) and (E'.*ok* }

E' → + **id** {E_1'.*set* = E'.*set*} E_1' { E'.*ok* = *in*(**id**.*lexval*, E'.*set*) and (E_1'.*ok* }

E' → ε {E'.*ok* = true}

   b) Scrivere le funzioni del traduttore a discesa ricorsiva.

```
Program  main(L)
  var S_set, S_ok
     S_set ← L
     S_ok ←  S(S_set)
     if (cc = $)  return (S_ok)
     else ERROR()


function S(S_set)
  var E_set, S_ok
     E_set = S_set
     S_ok ← E(E_set)
     return (S_ok)


function E(E_set)
  var E'_set, E'_ok , E_ok, N
     if (cc =  'id')
          N ← id.lexval
          cc ← PROSS( )
          E'_set = E_set
          E'_ok ← E (E'_set)
          E_ok ←  E'_ok  and in(N, E_set)
     else ERROR( )
     return (E_ok)
```

```
function E'(E'_set)
       var E1'_set, E'_ok, Nome
       if (cc =  '+')
              cc← PROSS( )
              if (cc = 'id')  Nome ← id.lexval
                                cc ← PROSS( )
              else ERRORE( )
              E1'_set = E'_set
              E1'_ok ← E'(E1'_set)
              E'_ok ←  E1'_ok  and in(Nome, E'_set)
       elseif  (cc = '$')  E'_ok ← true
       else ERRORE( )
       return (E'_ok)
```

8. Fornire la traduzione nel Java bytecode degli statement:

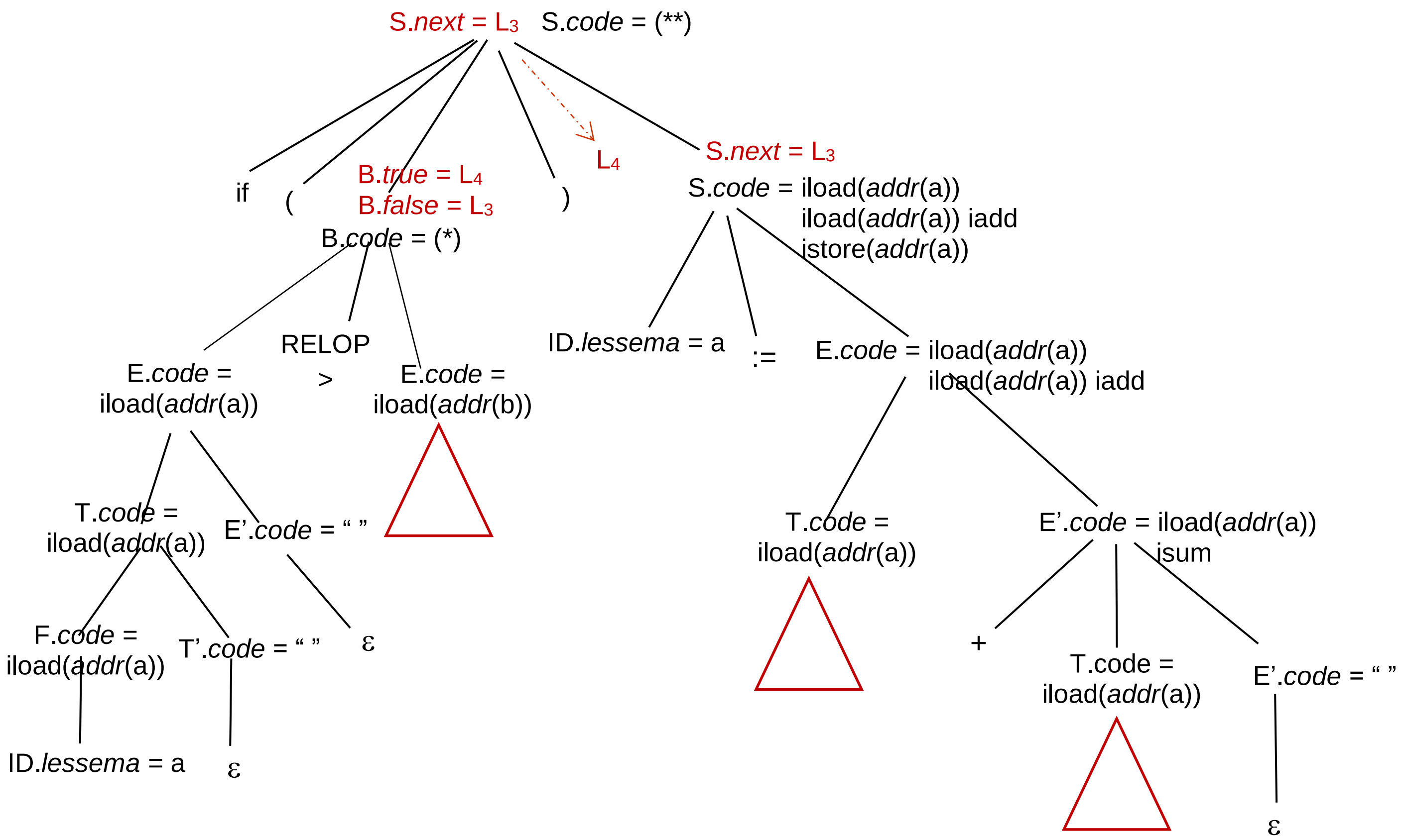
a) S: a := 8 ; b := a

b) S: **if** (a > b) a := a + a

c) S**:** **if** (a < b) x := y **else** x := 2 ∗ y

Per tutti e tre gli statement si supponga S.*next* = L_3.

b) S: if (a > b) a := a + a



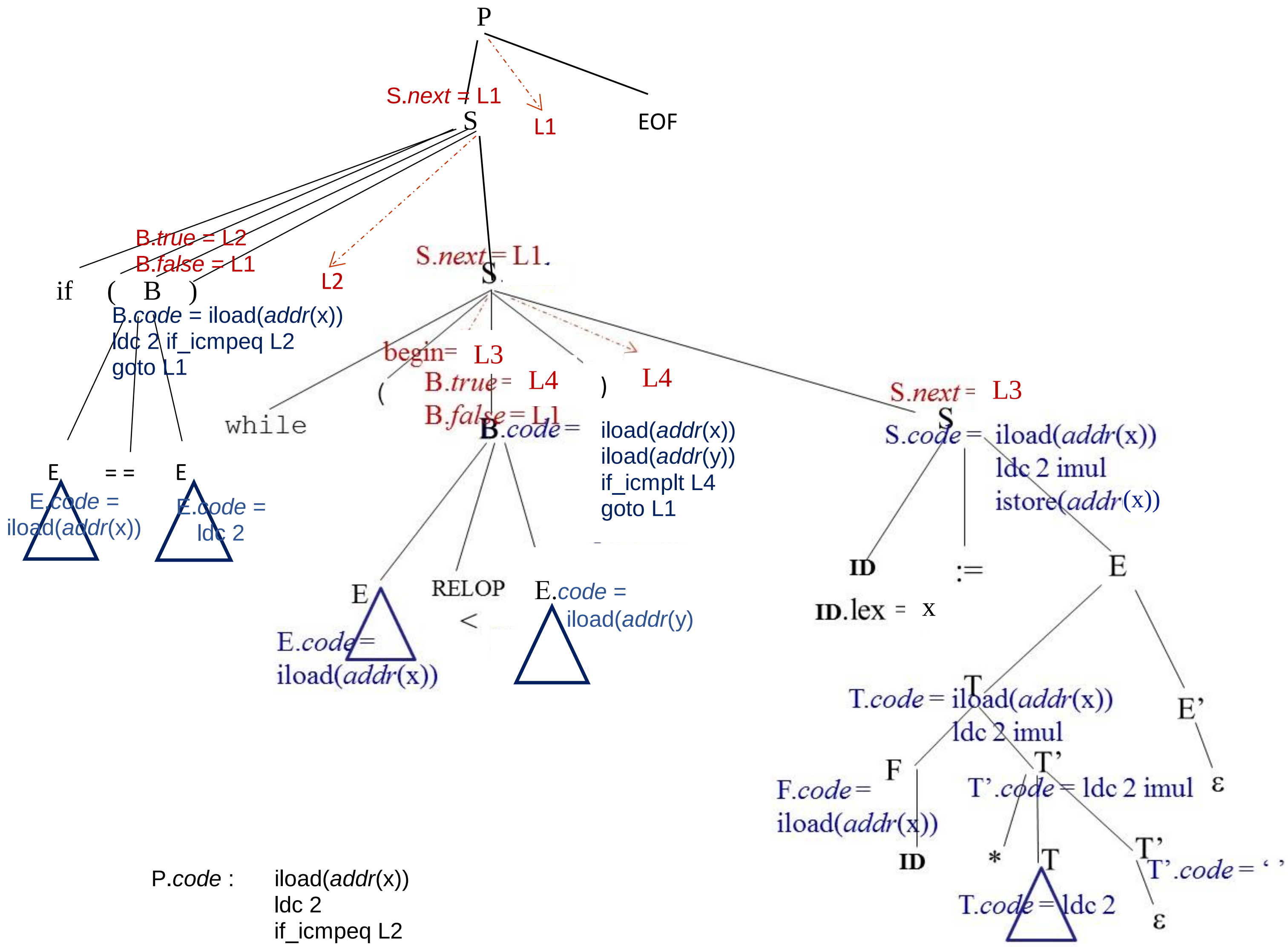
(*) B.*code* = iload(*addr*(a))
iload(*addr*(b))
if_icmpgt L_4
goto L_3

(**) S.*code* = iload(*addr*(a))
iload(*addr*(b))
if_icmpgt L_4
goto L_3
L_4 iload(*addr*(a))
iload(*addr*(a))
iadd
istore(*addr*(a))

9. Costruire l'albero di parsificazione per il seguente programma:

if (x = = 2) **while** (x < y) x:= x * 2 EOF

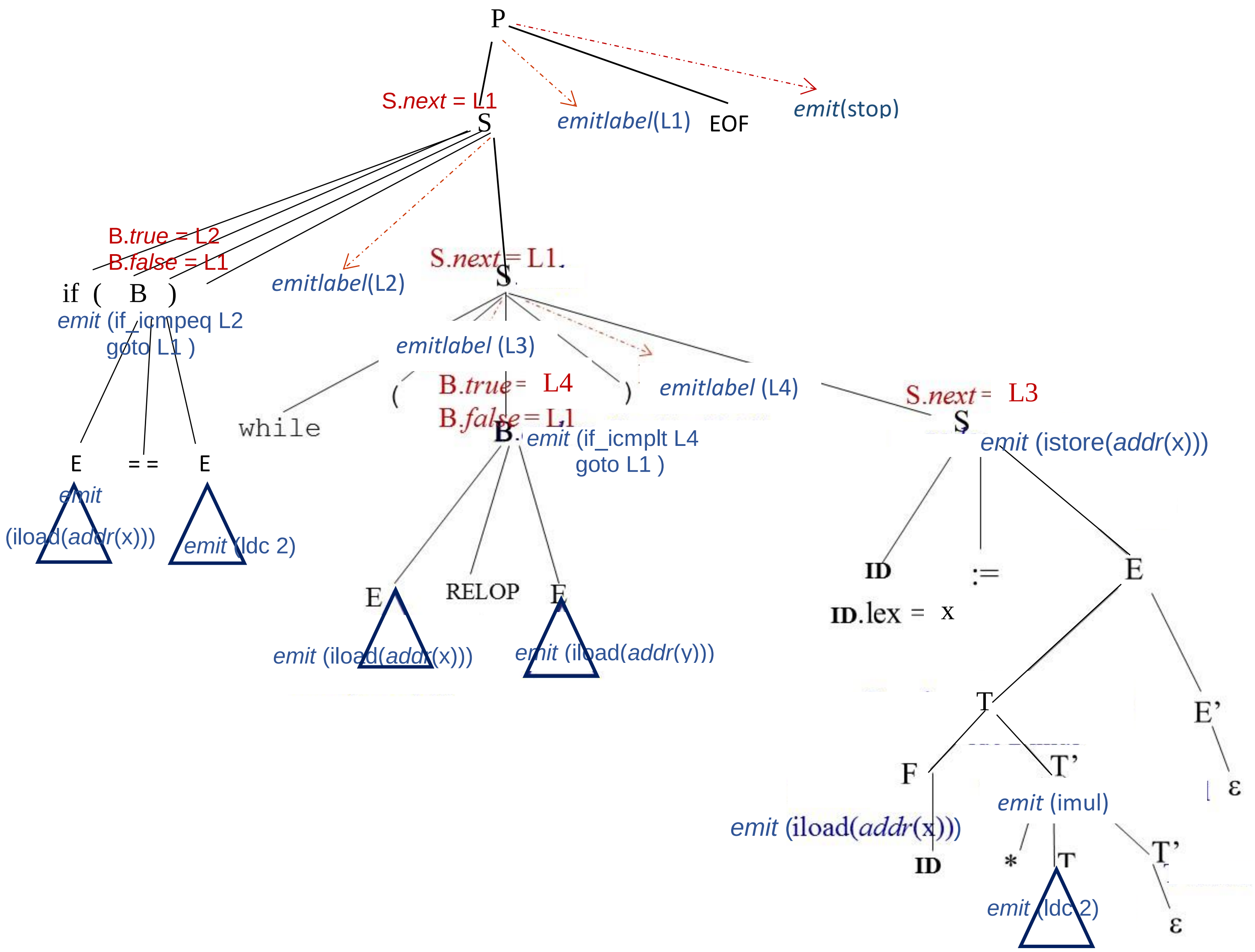
- annotarlo con gli attributi necessari a calcolare la sua traduzione nel bytecode;



P.*code* :

```
        iload(addr(x))
        ldc 2
        if_icmpeq L2
        goto L1
L2
L3      iload(addr(x))
        iload(addr(y))
        if_icmplt L4
        goto L1
L4      iload(addr(x))
        ldc 2
        imul
        istore(addr(x))
        goto L3
L1      stop
```

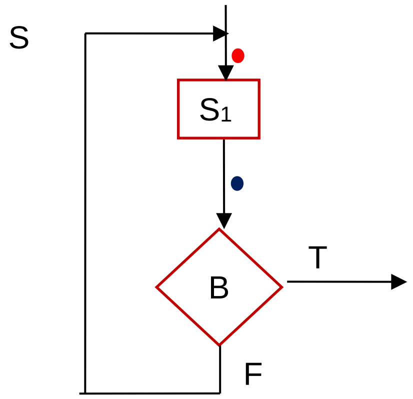
- annotarlo per la traduzione on-the fly



10. Individuare le regole semantiche per la traduzione nel bytecode dello statement

repeat S **until** B

con la seguente interpretazione “esegui S; se B è vero esegui l’istruzione successiva, altrimenti ripeti il ciclo”.



Dopo l’esecuzione di S_1 si deve valutare B per cui al codice per B viene premessa un’etichetta che è il valore di S_1.*next* (pallino blu).

Quando B risulta vero si deve eseguire l’istruzione etichettata S.*next*, pertanto B.*true* sarà uguale a S.*next*; se invece B risulta falso, si deve effettuare il salto incondizionato all’esecuzione del codice per S_1 (pallino rosso). È necessario pertanto inserire una nuova etichetta, che possiamo chiamare “begin”, prima di S_1.

begin = *newlabel* ( )

S_1.*next* = *newlabel* ( )

B.*true* = S.*next*

B.*false* = begin

S.*code* = *label*(begin) || S_1.*code* || *label*(S_1.*next*) || B.*code*

1) <u>Schema di traduzione</u>

S → **repeat** {begin = *newlabel* ( ), S_1.*next* = *newlabel* ( )} S_1 **until**

{B.*true* = S.*next*, B.*false* = begin} B {S.*code* = *label*(begin) || S_1.*code* ||
|| *label*(S_1.*next*) || B.*code*}

2) <u>Schema di traduzione “on-the-fly”</u>

S → **repeat** {begin = *newlabel* ( ), S_1.*next* = *newlabel* ( ), *emitlabel*(begin)}

S_1 {*emitlabel*(S_1.*next*)} **until** {B.*true* = S.*next*, B.*false* = begin} B