UNIVERSITÀ
DEGLI STUDI
DI TORINO

Generazione di codice intermedio

a.a. 2018-2019

Linguaggio sorgente e linguaggio target

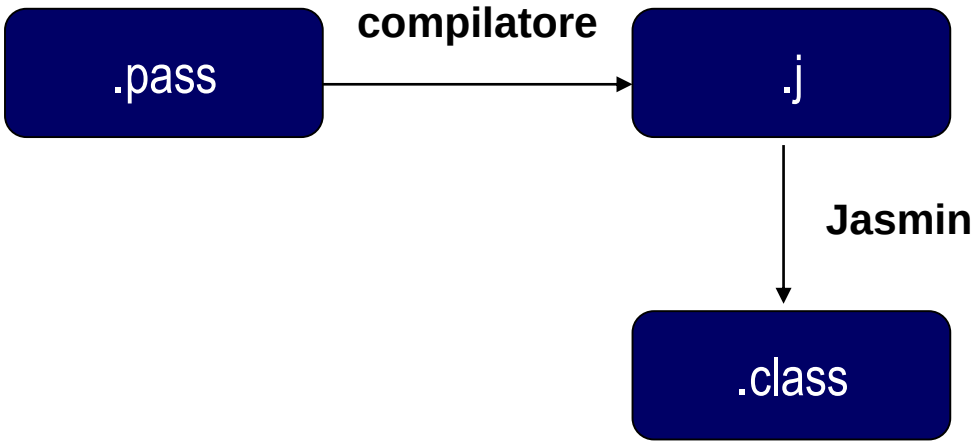
Il linguaggio considerato, che chiameremo $\mathbb{P}$, è un frammento di un linguaggio di programmazione imperativo ed include espressioni aritmetiche e booleane e istruzioni di assegnazione, condizionali e iterazione.

Non vedremo invece:
- Chiamate e rientri da procedure
- Istruzioni di copiatura indexata: x = y[i] e x[i] = y
- Assegnazione di indirizzi e puntatori

Il linguaggio target è un codice postfisso rappresentato da un sottoinsieme del JAVA bytecode, eseguibile dalla JVM

Linguaggio targed

Generare bytecode direttamente non è semplice per la complessità del formato dei file .class (binario). Useremo perciò un linguaggio mnemonico (un linguaggio assembly) che viene tradotto successivamente nel formato .class dal programma assembler Jasmine, che effettua una traduzione 1-1 delle istruzioni mnemoniche nella corrispondente istruzione binaria della JVM.



Il linguaggio $\mathbb{P}$ da tradurre

Programma

P ::= SL EOF

Istruzioni

- S → **ID** := E
- S → `print` (E)
- S → `read` (ID)
- S → `while` (B) S
- S → `if` (B) S
- S → `if` (B) S `else` S
- S → {SL}
- SL → SL ; S
- SL → S

Espressioni aritmetiche

- E → T E'
- E' → + T E'
- E' → - T E'
- E' → ε

- T → F T'
- T' → * F T'
- T' → / F T'
- T' → ε

- F → (E)
- F → **NUM**
- F → **ID**

Espressioni booleane

- B → E RELOP E

Le istruzioni del linguaggio target

Un programma in bytecode è costituito da una sequenza di istruzioni. È possibile introdurre dei *label* nella sequenza per consentire i salti nell'esecuzione.

Istruzioni:

- ldc, iload, istore
- iadd, imul, isub, idiv
- if_icmpeq, if_icmpne, if_icmpgt, if_icmpge, if_icmplt, if_icmple } (salti condizionati)
- goto label (salto incondizionato)

Dal linguaggio ℙ al bytecode

Si assume di disporre di una tabella dei simboli (symbol table) in cui vengono registrate le proprietà delle variabili usate, come il tipo e l'indirizzo di allocazione in memoria.

Nelle regole semantiche per la traduzione vengono usate le funzioni:

- *addr*(ID.lessema) trova nella symbol table "corrente" l'indirizzo associato al lessema (la posizione di memoria che contiene il valore della variabile).
- *newlabel*( ) genera una nuova label simbolica
- *label*(x.yyy) "attacca" il valore dell'attributo x.yyy allo statement seguente

Dal linguaggio ℙ al bytecode: esempi

Tabella dei simboli

lexeme	type	address	etc..
a	int	101…	…….
b	int	110…	…….
…	…	…	
…	…	…	
c	int	101…	…….
…	…	…	

a := b + c * b

```
iload (addr(b))
iload (addr(c))
iload (addr(b))
imul
iadd
istore (addr(a))
```

if a == 1 then b := 4 else b := 5

```
      iload (addr(a))
      ldc 1
      if_icmpeq L2
      goto L3
L2    ldc 4
      istore(addr(b))
      goto L1
L1    ldc 5
      istore (addr(b))
L1
```

Dal linguaggio $\mathbb{P}$ al bytecode: espressioni aritmetiche

Consideriamo le espressioni generate dalla variabile E.

- $E \rightarrow T\ E'$
- $E' \rightarrow +\ T\ E'$
- $E' \rightarrow \varepsilon$

Usiamo per le espressioni aritmetiche un solo attributo (*.code*), sintetizzato, per associare ad ogni variabile il pezzo di codice generato. Consideriamo inoltre '+' associativo a sinistra, cioè vogliamo, ad esempio per la forma sentenziale:

$$T_1 + T_2 + T_3$$

una traduzione come:

$$T_1.code \;||\; T_2.code \;||\; \text{iadd} \;||\; T_3.code \;||\; \text{iadd}$$

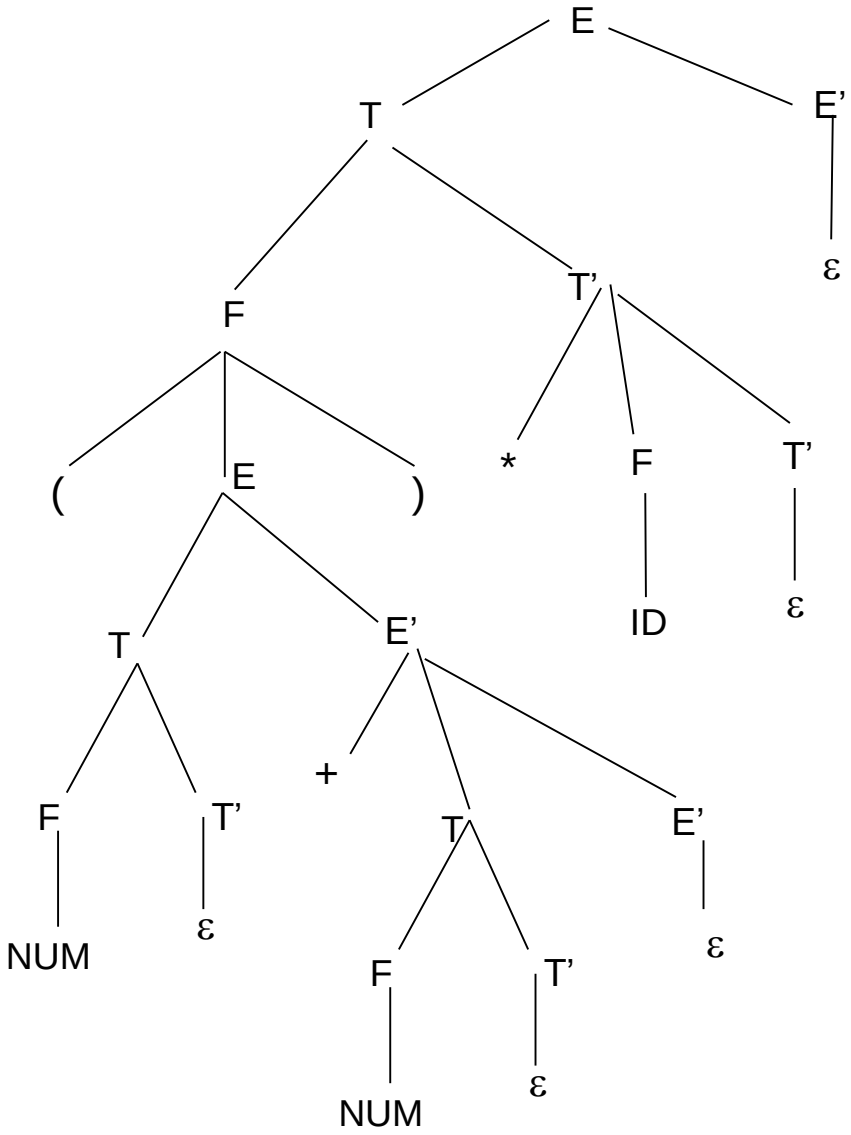
Nota: La traduzione è una sequenza di istruzioni, cioè di stringhe. Come di consueto, usiamo || come operatore di concatenazione.

Dal linguaggio ℙ al bytecode: espressioni aritmetiche

Esempio: (3 + 5) * a

(3 + 5) * a

ldc 3
ldc 5
iadd
iload(*addr*(a))
imul

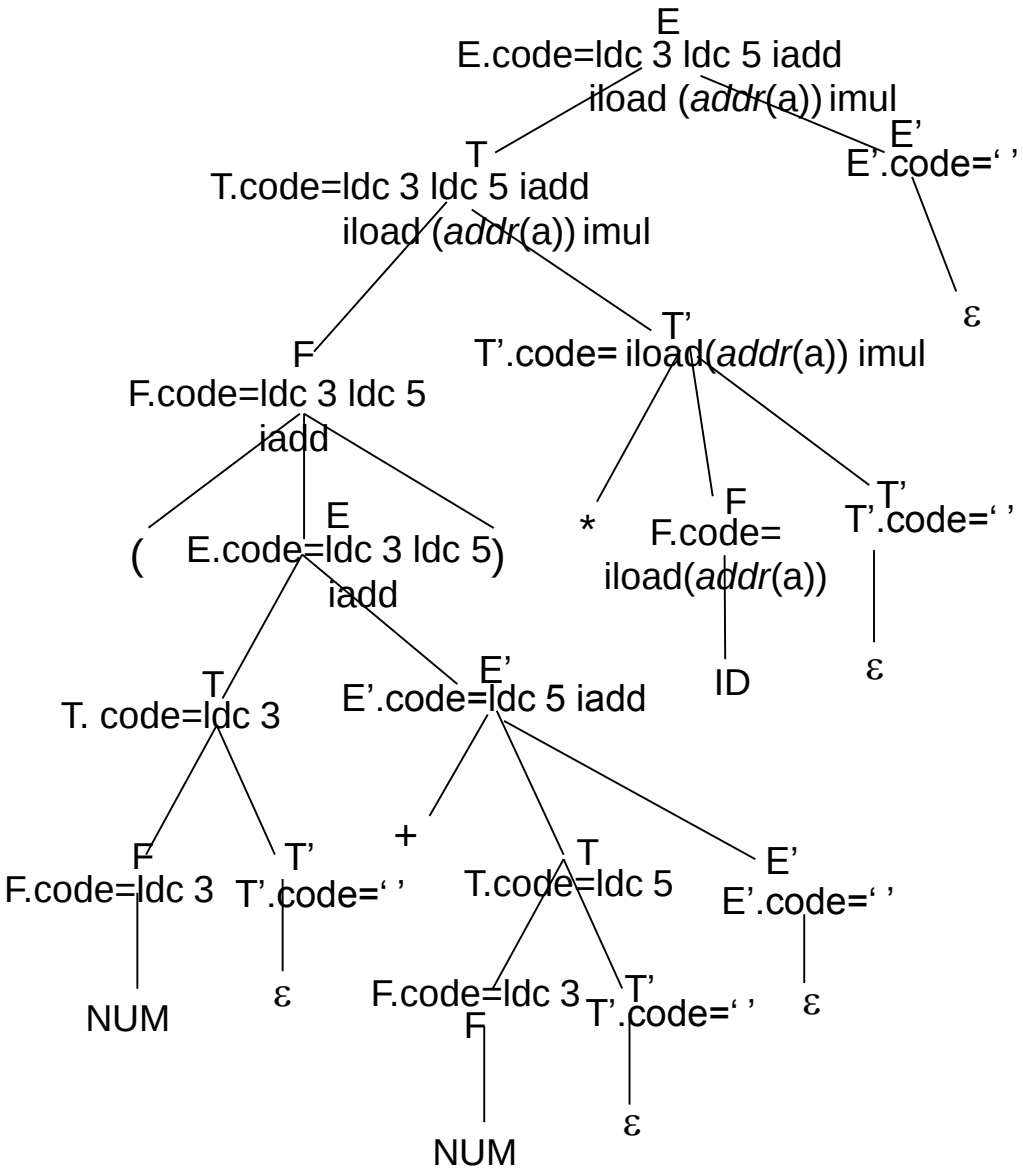


Dal linguaggio ℙ al bytecode: espressioni aritmetiche

Esempio: (3 + 5) * a

(3 + 5) * a

ldc 3
ldc 5
iadd
iload(*addr*(a))
imul



Dal linguaggio $\mathbb{P}$ al bytecode: espressioni aritmetiche

L'esempio suggerisce la definizione del seguente schema di traduzione:

$E \rightarrow T\ E'$ {E.*code* = T.*code* || E'.*code*}

$E' \rightarrow +\ T\ E'_1$ {E'.*code* = T.*code* || "iadd" || E'_1.*code*}

$E' \rightarrow -\ T\ E'_1$ {E'.*code* = T.*code* || "isub" || E'_1.*code*}

$E' \rightarrow \varepsilon$ {E'.*code* = " "}

Considerazioni analoghe si possono fare per le variabili T e T', per le cui produzioni si ottengono schemi di traduzione simili a quelli per le produzioni di E ed E'.

Per la variabile F:

$F \rightarrow (E)$ {F.*code* = E.*code*}

$F \rightarrow$ **NUM** {F.*code* = ldc(**NUM**.val)}

$F \rightarrow$ **ID** {F.*code* = iload(*addr*(**ID**.lessema)}

Dal linguaggio ℙ al bytecode: espressioni aritmetiche

Anche per una espressione aritmetica molto semplice, come (3 + 5) * a, i valori degli attributi .*code* sono stringhe piuttosto lunghe.

E.*code* = ldc 3 ldc 5 iadd iload(*addr*(a)) imul

1. L'attributo principale è sintetizzato
2. Le regole semantiche sono tali che:
   a) L'attributo è ottenuto dal concatenamento di stringhe
   b) Gli attributi dei non terminali si presentano nella regola semantica nello stesso ordine in cui i non terminali si presentano nel corpo della produzione

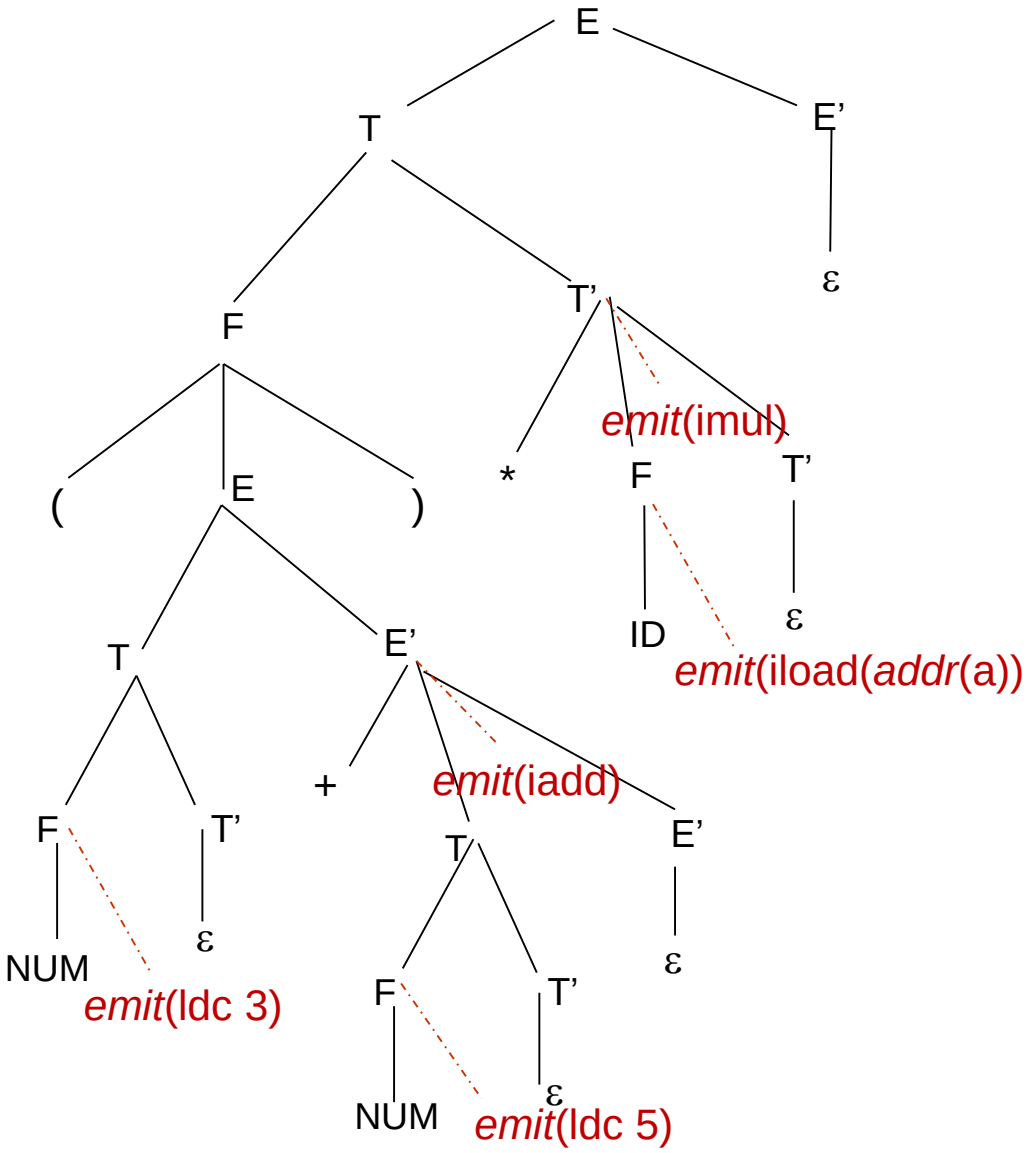
Traduzione on-the-fly

Dal linguaggio ℙ al bytecode: espressioni aritmetiche

Esempio: (3 + 5) * a

(3 + 5) * a

ldc 3
ldc 5
iadd
iload(*addr*(a))
imul



Dal linguaggio $\mathbb{P}$ al bytecode: espressioni aritmetiche

Riscriviamo lo schema :

$E \rightarrow T\ E'$

$E' \rightarrow + T\ \{emit(\text{iadd})\}\ E'_1$

$E' \rightarrow - T\ \{emit(\text{isub})\}\ E'_1$

$E' \rightarrow \varepsilon$

Dal linguaggio ℙ al bytecode: espressioni booleane

In un linguaggio di programmazione le espressioni booleane sono usate:

- in statement condizionali per modificare il flusso di controllo
- per calcolare valori logici

Consideriamo solo la traduzione per le espressioni da usare negli statement condizionali

Grammatica per le espressioni booleane

B ::= E RELOP E

Consideriamo solo l'operatore relazionale ==.
L'estensione ad altri operatori relazionali è ovvia.

Dal linguaggio $\mathbb{P}$ al bytecode: espressioni booleane

Per le espressioni booleane usiamo tre attributi:

a) .*code* sintetizzato per la traduzione in bytecode
b) .*true* e .*false* , attributi ereditati che servono per definire le label delle prime istruzioni che traducono gli statement da eseguire nei casi B vero e B falso rispettivamente.

$B \rightarrow E_1 == E_2$ {B.*code* = E_1.*code* || E_2.*code* ||
|| 'if_icmpeq' B.*true* || 'goto' B.*false*}

Dal linguaggio $\mathbb{P}$ al bytecode: istruzioni

Per gli statement del linguaggio usiamo due attributi.
Oltre all'attributo sintetizzato .*code*, un attributo ereditato .*next*, che memorizza la label della prima istruzione che traduce lo statement che segue S nel programma.

P $\rightarrow$ SL EOF	SL.*next* = *newlabel*( ) P.*code* = SL.*code* || *label*(SL.*next*) || 'stop'

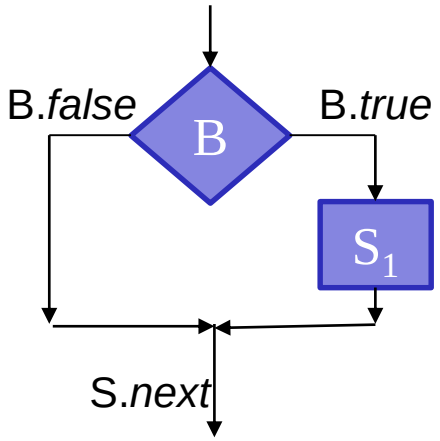
La traduzione del programma P è ottenuta concatenando la traduzione per SL a una label nuova, che è il valore dell'attributo S.*next*

S $\rightarrow$ **ID** := E	S.*code* = E.*code* || istore(*addr*(**ID**.*lessema*))

La traduzione di S è ottenuta concatenando la traduzione di E a una nuova istruzione, che realizza l'assegnazione

Dal linguaggio ℙ al bytecode: istruzioni

$S \rightarrow$ `if` (B) S_1

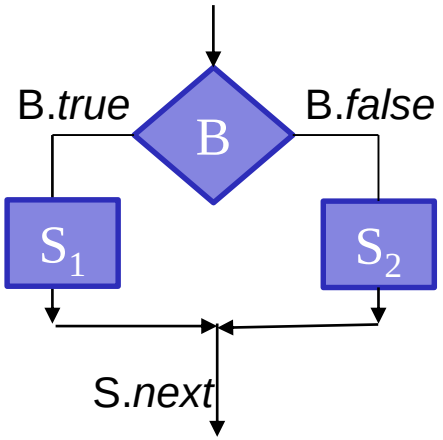


Viene creata una nuova label come valore per B.*true* per etichettare la prima istruzione della traduzione di S_1, da eseguire se B risulta vero, mentre nel caso in cui B sia falso l'istruzione da eseguire è quella successiva a S, quindi l'etichetta B.*false* è uguale a S.*next*. Anche dopo l'esecuzione del codice per S_1 deve essere eseguita l'istruzione etichettata con il valore di S.*next*.

B.*true* = *newlabel*( )
B.*false* = S_1.*next* = S.*next*
S.*code* = B.*code* || *label*(B.*true*) || S_1.*code*

Dal linguaggio ℙ al bytecode: istruzioni

$S \rightarrow$ `if` (B) S_1 `else` S_2

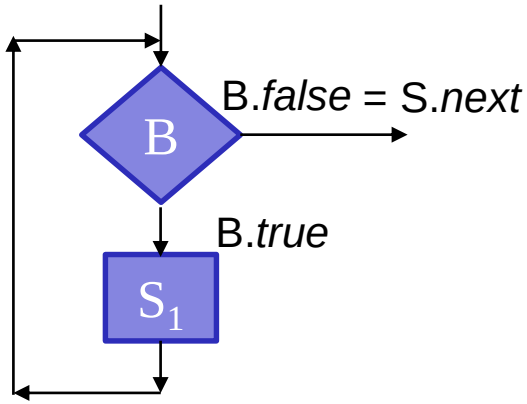


I valori per B.*true* e B.*false* sono due nuove label che servono per etichettare rispettivamente la prima istruzione della traduzione di S_1 e di quella di S_2. Sia dopo S_1 sia dopo S_2 va eseguita l'istruzione con label S.*next,* quindi S_1.*next* e S_2.*next* hanno il valore di S.*next.*

B.*true* = *newlabel*( )
B.*false* = *newlabel*( )
S_1.*next* = S_2.*next* = S.*next*
S.*code* = B.*code* || *label*(B.*true*) || S_1.*code* ||
|| 'goto' S_1.*next* || *label*(B.*false*) || S_2.*code*

Dal linguaggio ℙ al bytecode: istruzioni

$S \rightarrow$ `while` (B) S_1



Dopo l'esecuzione di S_1 si deve valutare nuovamente B per cui al codice per B viene premessa un'etichetta che permette di effettuare il salto incondizionato dopo l'esecuzione del codice per S_1. Tale etichetta è pertanto anche il valore di S_1.*next*. Quando B risulta falso si deve eseguire l'istruzione etichettata S.*next*.

begin = *newlabel*( )
B.*true* = *newlabel*( )
B.*false* = S.*next*
S_1.*next* = begin
S.*code* = *label*(begin) || B.*code* || *label*(B.*true*) ||
|| S_1.*code* || 'goto' S_1.*next*

Dal linguaggio $\mathbb{P}$ al bytecode: istruzioni

$S \rightarrow \{SL\}$	$SL.next = S.next$ $S.code = SL.code$
$SL \rightarrow SL_1 ; S$	$SL_1.next = newlabel( )$ $S.next = SL.next$; $SL.code = SL_1.code \mid\mid label(SL_1.next) \mid\mid S.code$
$SL \rightarrow S$	$S.next = SL.next$ $SL.code = S.code$

Il codice per una concatenazione SL;S è ottenuto dalla concatenazione dei due codici per SL e per S. L'istruzione che segue S è la stessa che segue SL (S.*next* = SL.*next*), mentre l'istruzione che segue SL nel corpo della produzione, cioè S, viene etichettata con una label nuova.

Esercizio: per la traduzione deterministica, eliminare la ricorsione sinistra e riscrivere le regole semantiche per le produzioni così ottenute.

Dal linguaggio ℙ al bytecode: SDT istruzioni

P → {SL.*next* = *newlabel*( )} SL EOF {P.*code* = SL.*code* || *label*(SL.*next*) || 'stop'}

S → **ID** := E {S.*code* = E.*code* || istore(*addr*(**id**.lessema))}

S → `if` ({B.*true* = *newlabel*( ) ; B.*false* = S.*next*}
B) {S_1.*next* = S.*next*} S_1 {S.*code* = B.*code* || label(B.*true*) || S_1.*code*}

S → `if` ({B.*true* = *newlabel*( ) ; B.*false* = *newlabel*( )}
B) {S_1.*next* = S.*next*} S_1 `else`
{S_2.*next* = S.*next*} S_2 {S.*code* = B.*code* || *label*(B.*true*) || S_1.*code* ||
|| '*goto*' S_1.*next*) || *label*(B.*false*) || S_2.*code*}

S → `while(` {begin = *newlabel*( ) ; B.*true* = *newlabel*( ) ; B.*false* = S.*next*}
B) {S_1.*next* = begin}
S_1 {S.*code* = *label*(begin) || B.*code* || *label*(B.*true*) || S_1.*code* || 'goto S_1.*next*}

S → { {SL.*next* = S.*next*} SL }{S.*code* = SL.*code*}

SL → {SL_1.*next* = *newlabel*( )} SL_1 ; {S.*next* = SL.*next*}
S {SL.*code* = SL_1.*code* || *label*(SL_1.*next*) || S.*code*}

SL → {S.*next* = SL.*next*} S {SL.*code* = S.*code*}

Dal linguaggio ℙ al bytecode: statement 'while'

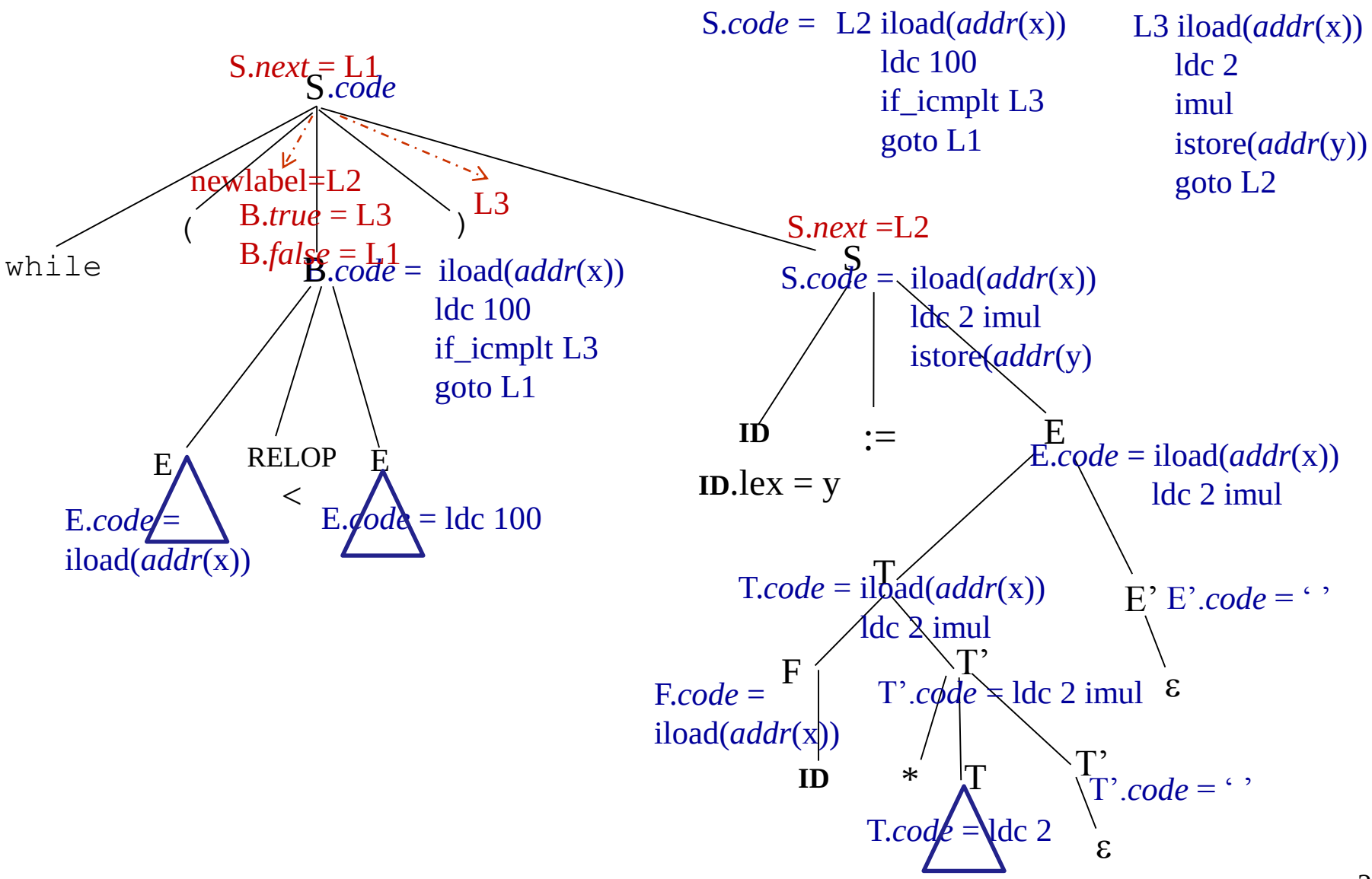
S → `while` ({begin = *newlabel*( ) ; B.*true* = *newlabel*( ) ; B.*false* = S.*next*}
B) {S_1.*next* = begin} S_1
{S.*code* = *label*(begin)||B.*code*||label(B.*true*)||S_1.*code*||'goto' S_1.*next*}

```
function S(Snext)
  var …
        …
    elseif (cc = 'while')
        cc ← PROSS
        if (cc = '(' ) cc ← PROSS
        else ERRORE(…)
        begin ← newlabel( )
        Btrue ← newlabel( )
        Bfalse ← Snext
        Bcode ← B(Btrue, Bfalse)
        if (cc = ')' ) cc ← PROSS
        else ERRORE(…)
        S1next ← begin
        S1code ← S(S1next)
        return(label(begin) || Bcode || label(Btrue)|| S1code || 'goto' S1next)
    elseif ….
```

Dal linguaggio ℙ al bytecode: esempio

S: `while` (x < 100) y := x * 2

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



Dal linguaggio ℙ al bytecode: on-the-fly

Le condizioni per scrivere la traduzione “on the fly” del linguaggio ℙ nel bytecode sono verificate:

1. c’è un attributo principale: .*code*
2. “.*code*” assume come valori stringhe di caratteri, che si possono concatenare
3. nelle regole semantiche gli attributi “T.*code*” e “E.*code*” si presentano nell’ordine in cui si presentano le variabili T e E nei corpi delle regole sintattiche.

Nella traduzione on the fly, oltre alla funzione *emit*( ) per scrivere il codice, usiamo una funzione, *emitlabel*( ) per scrivere le etichette.

Schemi di traduzione ‘on-the-fly’: espressioni

Espressioni aritmetiche

$E \rightarrow T\ E'$

$E' \rightarrow + T$ {*emit*(iadd)} E'_1

$E' \rightarrow - T$ {*emit*(isub)} E'_1

$E' \rightarrow \varepsilon$

$T \rightarrow F\ T'$

$T' \rightarrow * F$ {*emit*(imul)} T'_1

$T' \rightarrow / T$ {*emit*(idiv)} T'_1

$T' \rightarrow \varepsilon$

$F \rightarrow (E)$

$F \rightarrow$ **NUM** {*emit*(ldc(**NUM**.val))}

$F \rightarrow$ **ID** {*emit*(iload(*addr*(**ID**.lessema))}

Espressioni booleane

$B \rightarrow E_1 == E_2$ {*emit*(‘if_icmpeq’ B.*true* ;
emit(‘goto’ B.*false*}

Schemi di traduzione ‘on-the-fly’: statement

Esempio: traduzione on-the-fly dello statement S → while (B) S

S → `while` ({begin = *newlabel*( ) ; B.*true* = *newlabe*l( ) ; B.*false* = S.*next*} B)
{S_1.*next* = begin} S_1
{S.*code* = *label*(begin) || B.*code* || *label*(B.*true*) || S_1.*code* || ‘goto’ S_1.*next*}

Consideriamo la traduzione al volo

1. La funzione B( ) quando chiamata emetterà il suo codice
2. Altrettanto farà la funzione S per la traduzione dello statement nel corpo del while.

Che cosa resta da fare alla funzione S nella chiamata esterna?

1. Preparare la label del while
2. Dare un valore agli attributi ereditati .*true* e .*false* prima della chiamata alla funzione B( )
3. Dare un valore all’attributo .*next* prima della chiamata ricorsiva alla funzione S( ) e al rientro inserire il goto al test del while
4. Emettere le label create nei posti giusti

S → `while(` {begin=*newlabel*(); *emitlabel*(begin); B.*true*=*newlabel*( ); B.*false*=S.*next*}
B) {*emitlabel*(B.*true*)); S_1.*next* = begin} S_1 {*emit*(‘goto’ S_1.*next*)}

Schemi di traduzione ‘on-the-fly’: statement

P → {SL.*next* = *newlabel*( )} SL {*emitlabel*(SL.*next*)} EOF {*emit*(‘stop’)}

S → **ID** := E {*emit*(istore (*addr*(**ID.**lessema)))}

S → `if` ({B.*true* = *newlabel*( ) ; B.*false* = S.*next*}
B) {*emitlabel*(B.*true*) ; S_1.*next* = S.*next*} S_1

S → `if` ({B.*true* = *newlabel*( ) ; B.*false* = newlabel( )}
B) {*emitlabel*(B.*true*) ; S_1.*next* = S.*next* }
S_1 {*emit*(‘goto’ S_1.*next*))} `else` {*emitlabel*(B.*false*), S_2.*next* = S.*next* } S_2

S → `while(` {begin=*newlabel*(); *emitlabel*(begin); B.*true*=*newlabel*( ); B.*false*=S.*next*}
B) {*emitlabel*(B.*true*)); S_1.*next* = begin}
S_1 {*emit*(‘goto’ S_1.*next*)}

S → { {SL.*next* = S.*next*} SL }

SL → {SL_1.*next* = *newlabel*( )} SL_1 ; {*emitlabel*(SL_1.*next*) ; S.*next* = SL.*next*} S

SL → {S.*next* = SL.*next*} S

Traduzione ‘on-the-fly’: statement ‘while’

S → `while` ({begin = *newlabel*(), *emitlabel*(begin),
B.*true* = *newlabel*( ), B.*false* = S.*next*}
B) {*emitlabel*(B.*true*)), S_1.*next* = begin}
S_1 {*emit*(‘goto’ S_1.*next*)}

```
function S(Snext)
  var …
        …
    elseif (cc = ‘while’)
        cc ← PROSS
        if (cc = ‘(’ ) cc ← PROSS
        else ERRORE(…)
        begin ← newlabel( )
        Btrue ← newlabel( )
        Bfalse ← Snext
        Bcode ← B(Btrue, Bfalse)      { emitlabel(begin)
                                        B(Btrue, Bfalse)
        if (cc = ‘)’ ) cc ← PROSS
        else ERRORE(…)
        S1next ← begin                { emitlabel(Btrue)
        S1code ← S(S1next)              S(S1next)
                                        emit(‘goto’ S1next)
        return(label(begin) || Bcode || label(Btrue) || S1code || ‘goto’ S1next)
    elseif  ….
```

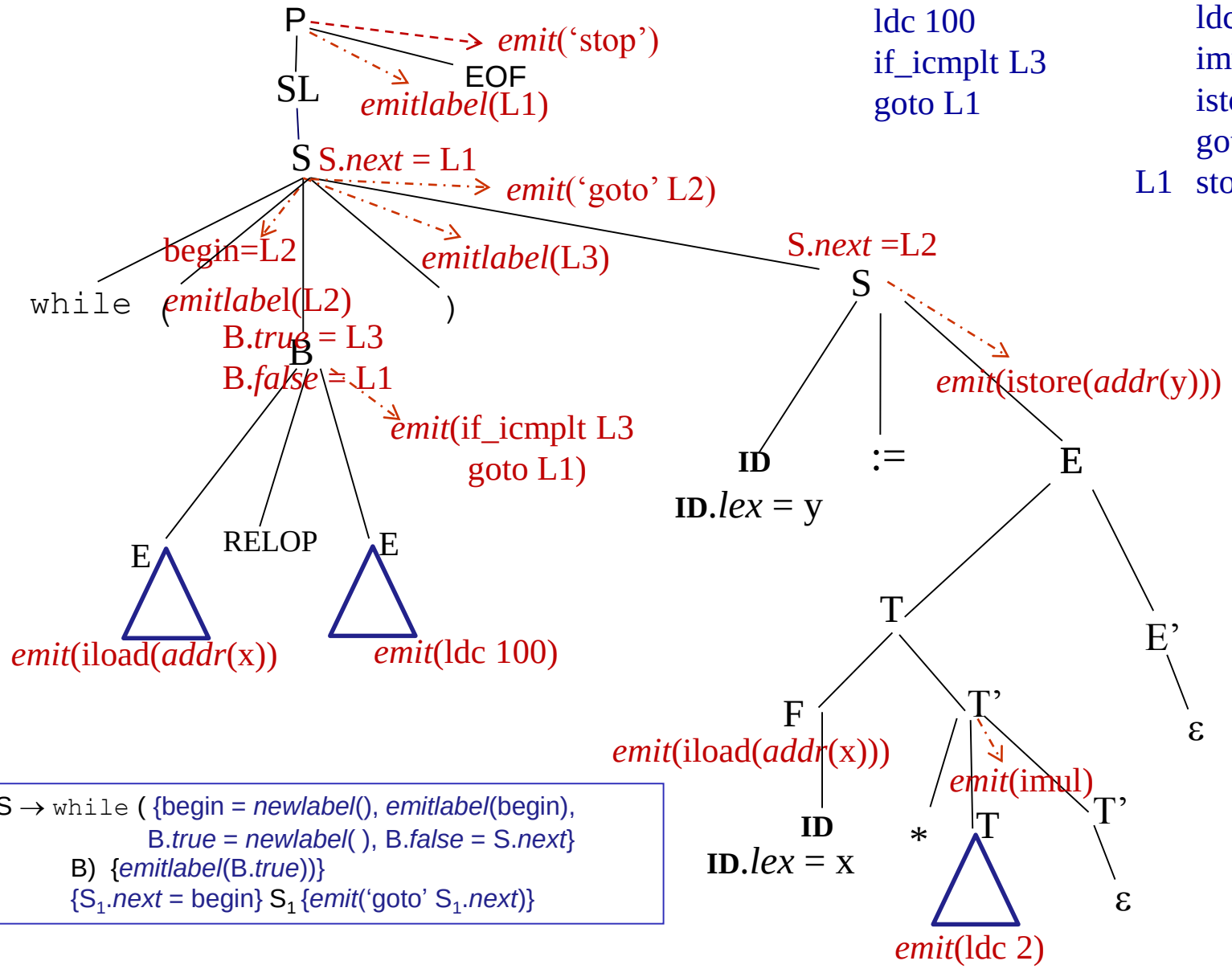
Traduzione ‘on-the-fly’: statement ‘while’

S → `while` ({begin = *newlabel*(), *emitlabel*(begin),
B.*true* = *newlabel*( ), B.*false* = S.*next*}
B) {*emitlabel*(B.*true*))}
{S_1.*next* = begin} S_1 {*emit*(‘goto’ S_1.*next*)}

```
function S(Snext)
  var …
          …
   elseif (cc = ‘while’)
        cc ← PROSS
        if (cc = ‘(’ ) cc ← PROSS
        else ERRORE(…)
        begin ← newlabel( )
        emitlabel(begin)
        Btrue ← newlabel( )
        Bfalse ← Snext
        B(Btrue, Bfalse)
        if (cc = ‘)’ ) cc ← PROSS
        else ERRORE(…)
        emitlabel(Btrue)
        S1next ← begin
        S(S1next)
        emit(‘goto’ S1next)
   elseif  ….
```

Traduzione ‘on-the-fly’: esempio

P: `while` (x < 100) y := x * 2 EOF



L2 iload(*addr*(x))
ldc 100
if_icmplt L3
goto L1

L3 iload(*addr*(x))
ldc 2
imul
istore(*addr*(y))
goto L2

L1 stop

S → `while` ({begin = *newlabel*(), *emitlabel*(begin),
B.*true* = *newlabel*( ), B.*false* = S.*next*}
B) {*emitlabel*(B.*true*))}
{S_1.*next* = begin} S_1 {*emit*(‘goto’ S_1.*next*)}

Dal linguaggio ℙ al bytecode: espressioni booleane

Estendiamo il linguaggio delle espressioni booleane con l'introduzione dei connettivi logici *and*, *or* e *not*.

Grammatica

B $\rightarrow$ E RELOP E | B||B | B&&B | !B

- grammatica ambigua
- nessuna precedenza tra gli operatori

Codice del "corto circuito"

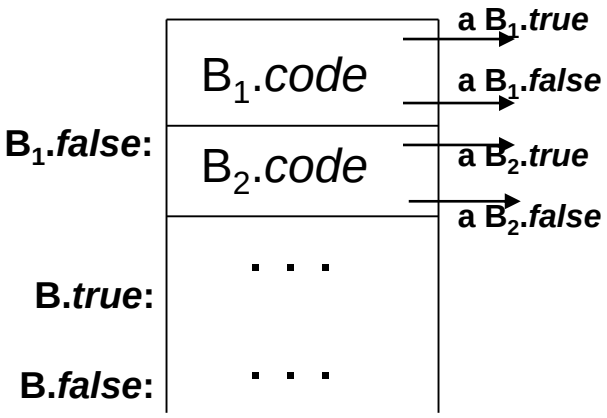
Osservazione: se B_1 è vero, anche $B_1||B_2$ è vero
se B_1 è falso, anche $B_1\&\&B_2$ è falso

Gli operatori || e && non compaiono nel codice, il loro valore è rappresentato da una posizione nella sequenza di istruzioni

Dal linguaggio ℙ al bytecode: espressioni booleane

Commentiamo in dettaglio le regole semantiche da associare ad una produzione, le regole associate alle altre produzioni possono essere commentate in modo analogo.

$B \rightarrow B_1 \,||\, B_2$



Usando la logica del corto circuito, se B_1 è vero si può eseguire lo statement per B vero ($B_1.true$ = B.*true*), mentre se B_1 è falso bisogna valutare B_2. Viene allora introdotta una nuova etichetta che permette di specificare quali istruzioni eseguire per valutare B_2. $B_2.true$ e $B_2.false$ hanno gli stessi valori degli analoghi attributi di B.

$B_1.true$ = B.*true* ; $B_1.false$ = *newlabel*( )
$B_2.true$ = B.*true* ; $B_2.false$ = B.*false*
B.*code* = $B_1.code$ || *label*($B_1.false$) || $B_2.code$

Dal linguaggio $\mathbb{P}$ al bytecode: espressioni booleane

$B \rightarrow E_1$ RELOP E_2 — $B.code = E_1.code \,||\, E_2.code \,||$ ‘if cmp…’ $B.true) \,||$ ‘*goto*’ $B.false$

$B \rightarrow B_1 || B_2$ — $B_1.true = B.true$; $B_1.false = newlabel( )$
$B_2.true = B.true$; $B_2.false = B.false$
$B.code = B_1.code \,||\, label(B_1.false) \,||\, B_2.code$

$B \rightarrow B_1 \&\& B_2$ — $B_1.true = newlabel( )$; $B_1.false = B.false$
$B_2.true = B.true$; $B_2.false = B.false$
$B.code = B_1.code \,||\, label(B_1.true) \,||\, B_2.code$

$B \rightarrow !B_1$ — $B_1.true = B.false$; $B_1.false = B.true;$ $B.code = B_1.code$

Dal linguaggio ℙ al bytecode: esempio

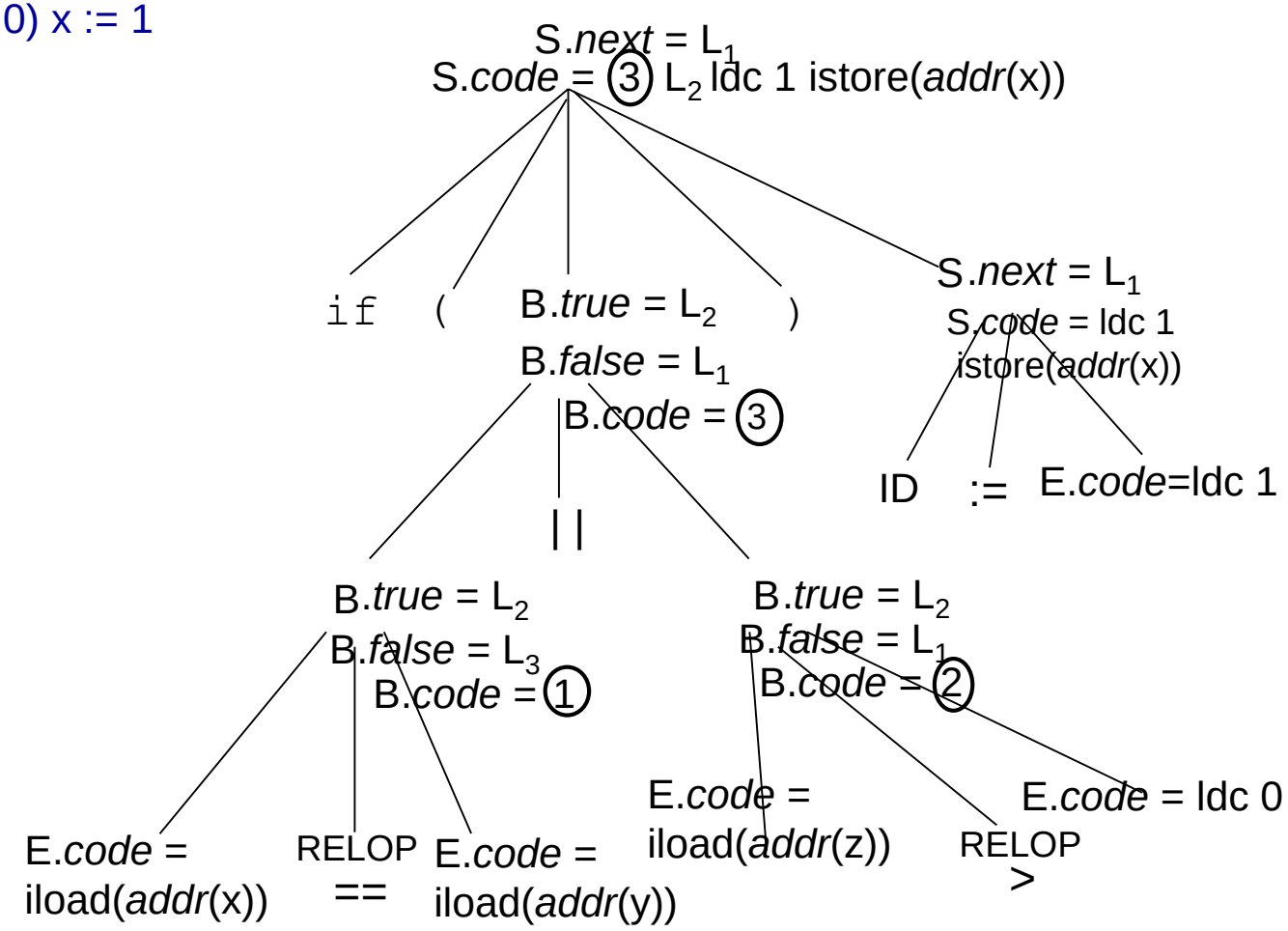
S: `if` (x == y||z > 0) x := 1

① iload(*addr*(x))
iload(*addr*(y))
if_icmpeq L_2
goto L_3

② iload(*addr*(z))
ldc 0
if_icmpgt L_2
goto L_1

③ iload(*addr*(x))
iload(*addr*(y))
if_icmpeq L_2
goto L_3

L_3 iload(*addr*(z))
ldc 0
if_icmpgt L_2
goto L_1



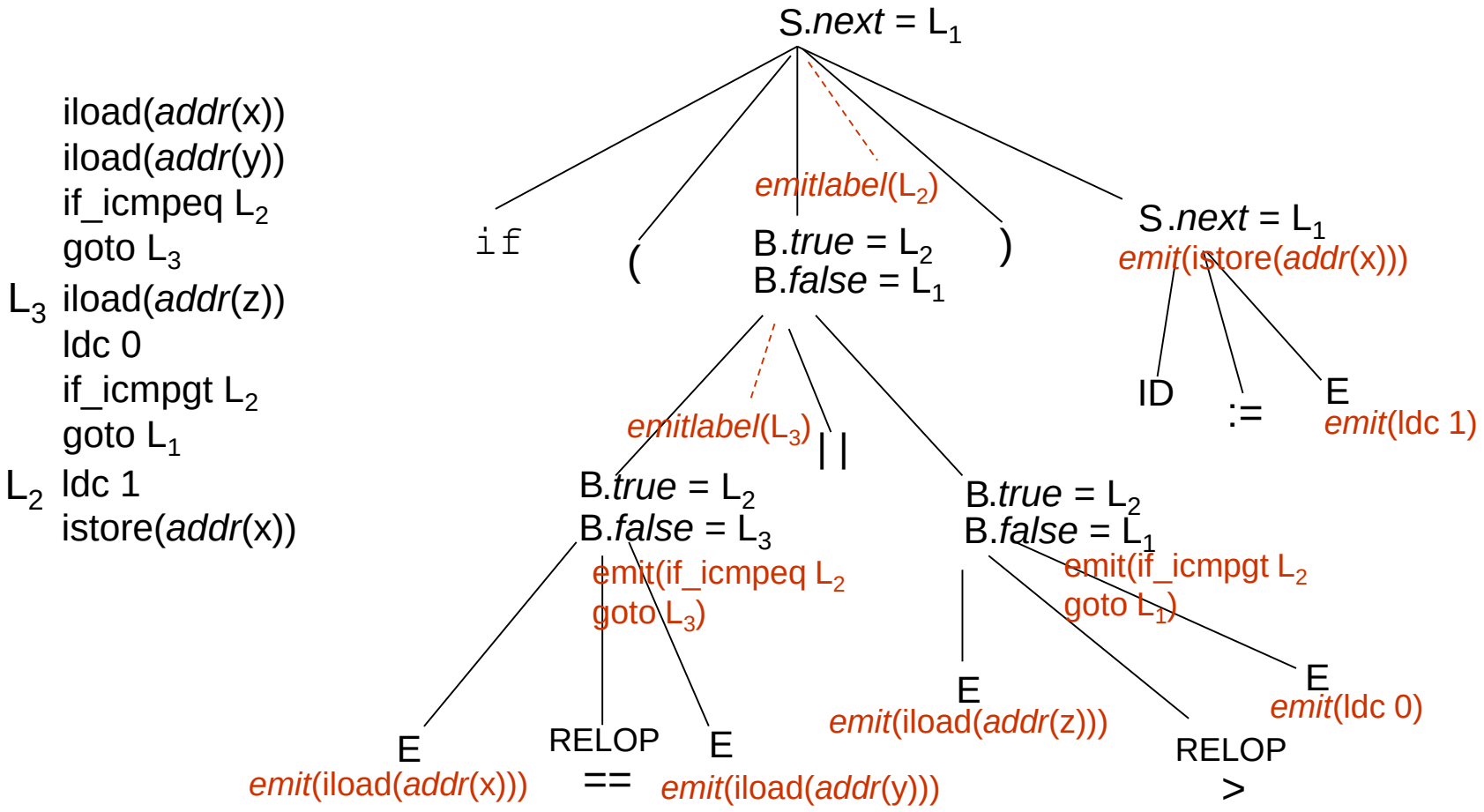
S.*code:* iload(*addr*(x))
iload(*addr*(y))
if_icmpeq L_2
goto L_3

L_3 iload(*addr*(z))
ldc 0
if_icmpgt L_2
goto L_1

L_2 ldc 1
istore(*addr*(x))

Dal linguaggio ℙ al bytecode: esempio on-the-fly

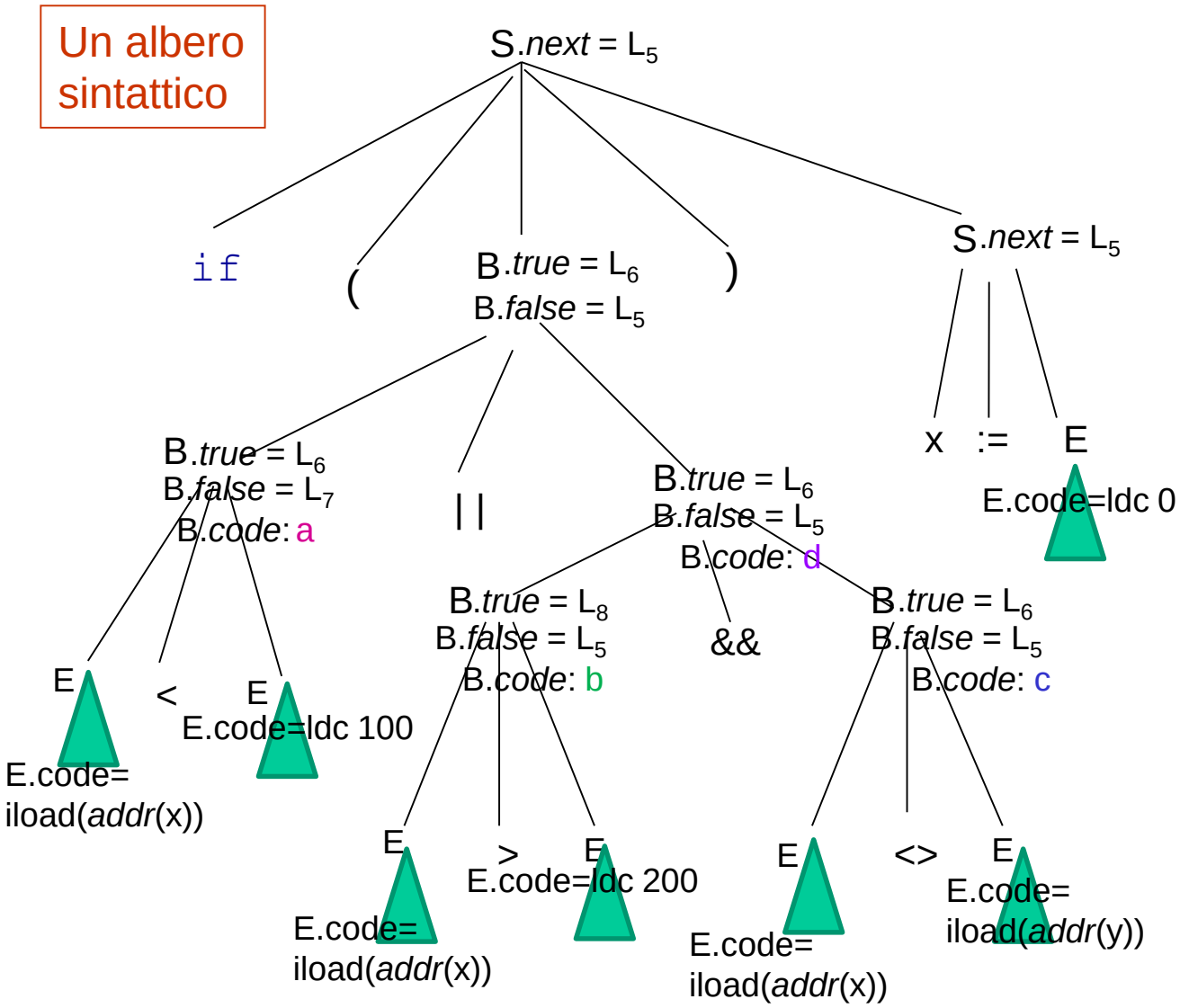
S: `if` (x == y||z > 0) x := 1



Dal linguaggio ℙ al bytecode: esempio

S: `if (x < 100 || x > 200 && x <> y) x := 0`

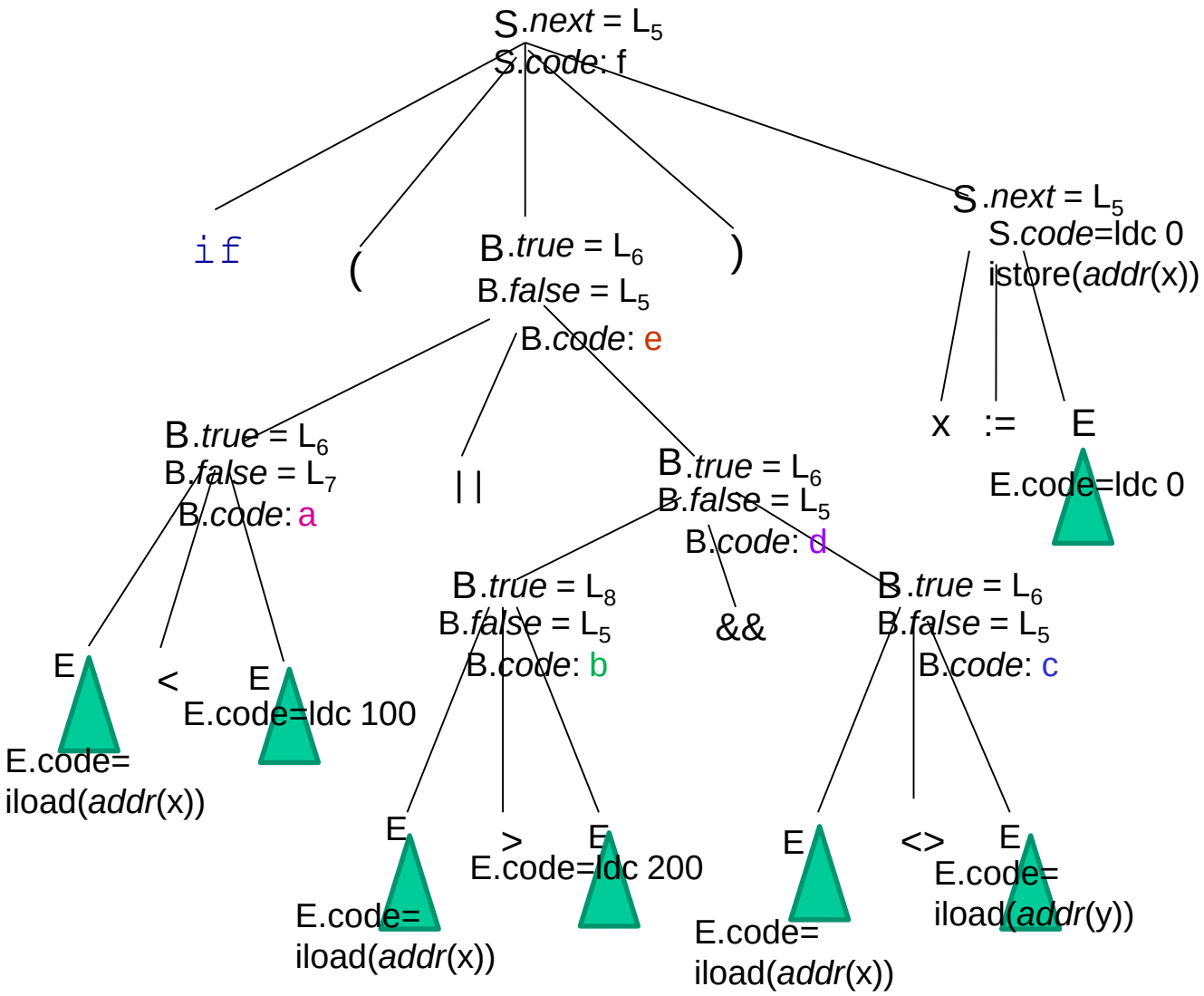
Un albero sintattico




```
    iload(addr(x))
    ldc 100
    if_icmplt L6
    goto L7
-------------------------
    iload(addr(x))
    ldc 200
    if_icmpgt L8
    goto L5
-------------------------
    iload(addr(x))
    iload(addr(y))
    if_icmpne L6
    goto L5
-------------------------
    iload(addr(x))
    ldc 200
    if_icmpgt L8
    goto L5
L8  iload(addr(x))
    iload(addr(y))
    if_icmpne L6
    goto L5
```

Dal linguaggio ℙ al bytecode: esempio

S: `if (x < 100 || x > 200 && x <> y) x := 0`




```
        iload(addr(x))
        ldc 100
        if_icmplt L6
        goto L7
L7      iload(addr(x))
        ldc 200
        if_icmpgt L8
        goto L5
L8      iload(addr(x))
        iload(addr(y))
        if_icmpne L6
        goto L5
```

```
        iload(addr(x))
        ldc 100
        if_icmplt L6
        goto L7
L7      iload(addr(x))
        ldc 200
        if_icmpgt L8
        goto L5
L8      iload(addr(x))
        iload(addr(y))
        if_icmpne L6
        goto L5
L6      ldc 0
        istore(addr(x)
```

Esercizi

1. Costruire l'albero sintattico annotato per generare il bytecode per le espressioni:

   (c – d) e a + b + (c – d)

2. Fornire la traduzione nel bytecode degli statement:
   S: if (a > b) a := a + a e S: if (a < b) x := y else x := 2 * y
   In entrambi i casi si supponga S.*next* = L3.

3. Fornire l'albero di parsificazione per il seguente statement e annotarlo con gli attributi necessari a calcolare la sua traduzione nel bytecode:

   S: if (a > b) a := a - b else if (a < b) b := b - a

   Si assuma S.*next* = L5**.**

4. 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".