UNIVERSITÀ
DEGLI STUDI
DI TORINO

Generazione di codice intermedio

a.a. 2017-2018

Linguaggio sorgente e bytecode

Consideriamo la traduzione in un codice intermedio di alcune espressioni e comandi tipici dei linguaggi di programmazione, assumendo una sintassi di tipo C per il programma sorgente

Ci occuperemo principalmente di istruzioni di controllo.

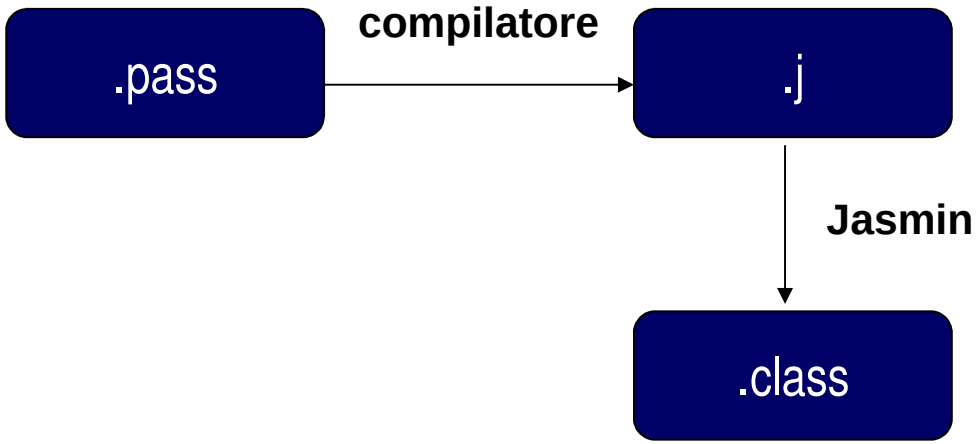
Non vedremo, in queste slides:
- Chiamate e rientri da procedure
- Istruzioni di copiature indexata: x = y[i] e x[i] = y
- Assegnazione di indirizzi e puntatori

Come linguaggio target consideriamo un codice postfisso rappresentato da un sottoinsieme del JAVA bytecode, eseguibile dalla JVM

Per rendere efficiente la traduzione scriveremo solo schmi di traduzione usando la tecnica ‘’on-the-fly’’.

Linguaggio targed

Generare bytecode direttamente non e' semplice per la complessita' del formato dei file .class (binario). Useremo percio` un linguaggio mnemonico (un linguaggio assembler) 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.



Le istruzioni considerate del linguaggio target

Istruzioni:

- ldc, iload, istore (azioni sulla memoria)
- iadd, imul, isub, idiv (operazioni su interi)
- iand, ior (operazioni logiche)
- if_icmpeq, if_icmpne, if_icmpgt, if_icmpge, if_icmplt, if_icmple } (salti condizionati)
- goto *label* (salto incondizionato)

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

Da ℙ al bytecode: esempio

a = b * c + b

```
iload (address(id.b))
iload (address(id.c))
imul
iload (address(id.b))
iadd
istore (address(id.a))
```

Tabella dei simboli

lexeme	type	address	etc..
a	int	0.	
b	int	1	……
…	…	…	
…	…	…	
c	int	35…	…….
…	…	…	

If a==1 then b=4 else b=5

```
    iload (address(id.a)
    ldc 1
    if_icmeq l1
    goto l2
l1:ldc 4
    istore (address(id.b))
    goto l3
l2: ldc 5
    istore (address(ID.b))
l3:
```

Funzioni di base

Negli schemi di traduzione verranno usate le funzioni:

- *address*(ID.lexeme), (spesso abbreviata in *address*(ID.lex)) che trova nella symbol table “corrente” l’indirizzo associato al lessema (la posizione di memoria che contiene il valore della variabile).
- *newlabel*( ), che genera una nuova label simbolica
- *label*(---) inserisce label nel codice
- *emit*(instr), *emitlabel*(label) aggiungono, rispettivamente, un’istruzione o un label al file di output in cui viene costruito il codice.

Si assume anche di poter disporre di una tabella dei simboli (symbol table) in cui vengono registrati i nomi delle variabili usate e le loro proprietà, come il tipo o l’indirizzo di allocazione in memoria.

Espressioni aritmetiche

Ricordiamo la traduzione delle espressioni artmetiche in notazione postfissa, aggiungendo gli operatori – e /

$E \rightarrow T\ E'$

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

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

$E' \rightarrow \varepsilon$

$T \rightarrow F\ T'$

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

$T' \rightarrow /\ F\ \{emit(\text{'/'})\ T'$

$T' \rightarrow \varepsilon$

$F \rightarrow \mathbf{id}\ \{emit(\mathbf{id}.\text{lexeme})\}$

$F \rightarrow (E)$

Espressioni aritmetiche

Schema di traduzione per un sottoinsieme di espressioni aritmetiche in bytecode

E → T E'

E' → + T {*emit*('iadd')} E'

E' → - T {*emit*('isub')} E'

E' → ε

T → F T'

T' → * F {*emit*('imult') T'

T' → / F {*emit*('idiv') T'

T' → ε

F → **id** {*emit*("iload" (*address*(**id**.lexeme)))}

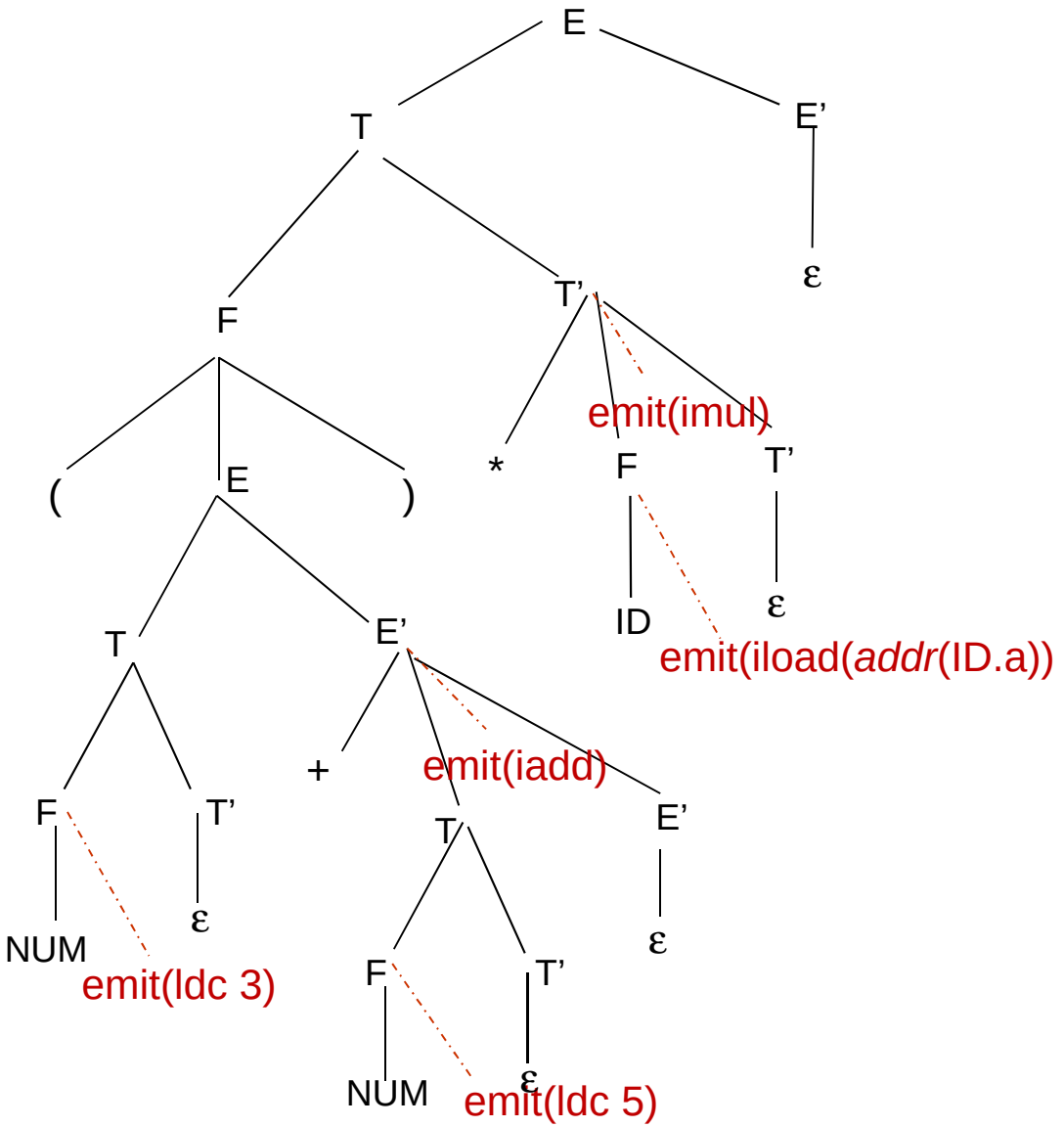
F → (E)

Dal linguaggio ℙ al bytecode: esempio

Esempio: (3 + 5) * a

(3 + 5) * a

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



Esempi di funzione

```
function E()
    begin
        if cc = '(' or cc = id)   T()
                                  E'()
        return
    end
```

```
function E'()
    begin  if (cc = '+')
               cc ← PROSS
               T()
               emit('iadd')
               E'()
           elseif (cc = '-')
               cc ← PROSS
               T()
               emit('isub')
               return
           else if (cc = ')' or cc=$)
                return
            else ERRORE (…)
     end
```

Esempi di funzione

```
function F()
    begin
        if (cc = '('  then  E()
                    if cc = ')' ) cc← PROSS
                                    return
                     else ERROR()
        else if (cc = id)
                    emit( "iload"  (address(id.lexeme)))
                     cc ← PROSS
         else ERROR()
    end
```

Esercizio: scrivere le altre funzioni

Istruzioni di assegnazione (semplificata)

$S \rightarrow$ **id** = E ……..

$E \rightarrow$ T E'

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

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

$E' \rightarrow \varepsilon$

$T \rightarrow$ F T'

$T' \rightarrow *$ F {*emit*('*') T'

$T' \rightarrow /$ F {*emit*('/') T'

$T' \rightarrow \varepsilon$

$F \rightarrow$ **id** {*emit*(**id**.lexeme)}

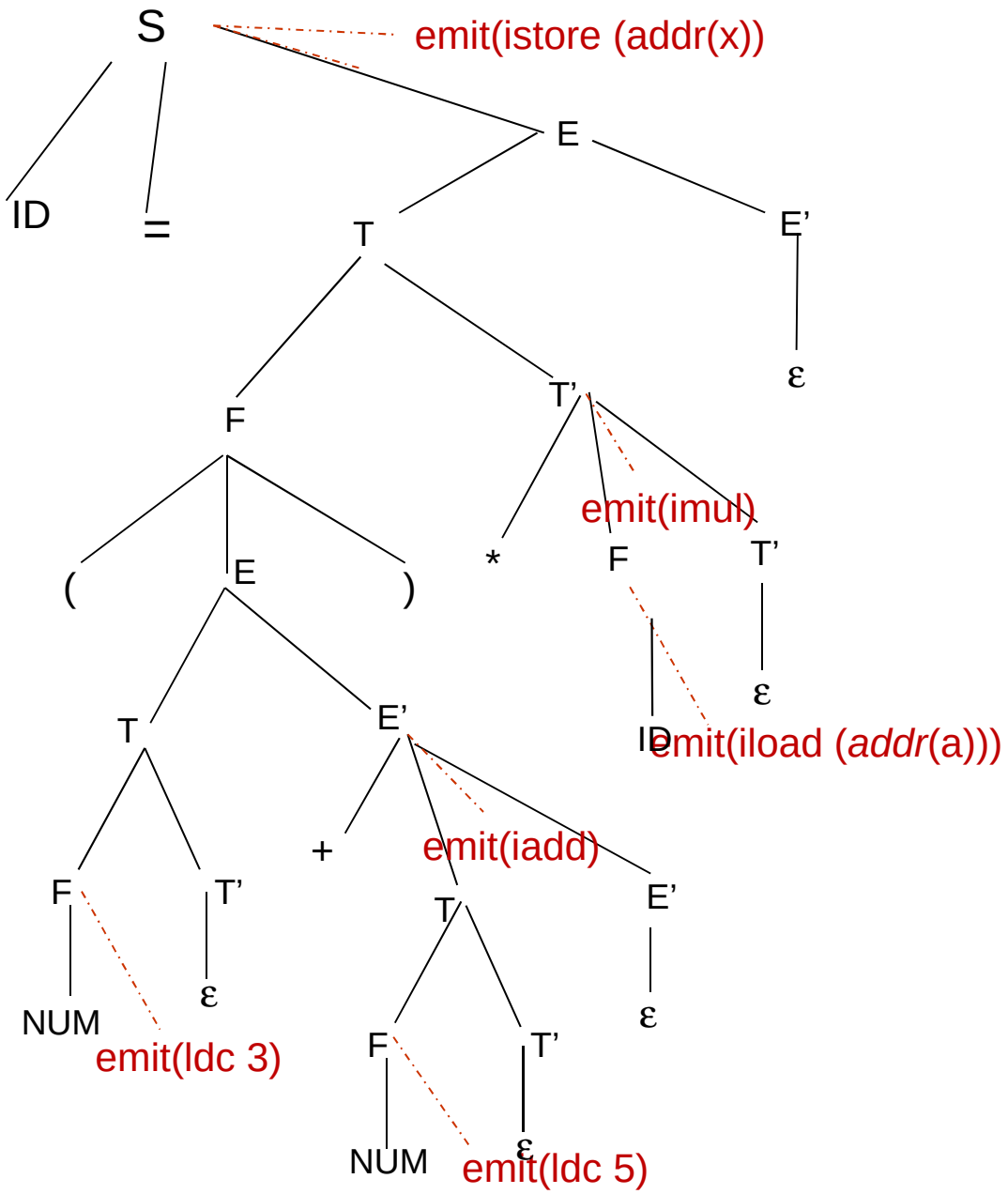
$F \rightarrow$ (E)

Assegnazione: esempio

Esempio: x = (3 + 5) * a

x = (3 + 5) * a

ldc 3
ldc 5
iadd
lload (*addr*(a))
Imul
Istore (*addr*(x))



$P \rightarrow SL$

$SL \rightarrow SL\ ;\ S$

$SL \rightarrow S$

$S \rightarrow \mathbf{id} = E$

$S \rightarrow \mathbf{if}\ \ B\ \ \mathbf{then}\ \ S$

$S \rightarrow \mathbf{if}\ \ B\ \mathbf{then}\ S\ \mathbf{else}\ S$

$S \rightarrow \mathbf{while}\ \ B\ \ \mathbf{do}\ S$

$S \rightarrow \mathbf{begin}\ SL\ \mathbf{end}$

$B \rightarrow\ E\ \mathbf{relop}\ E$

Traduzione di espressioni booleane e istruzioni

La traduzione delle espressioni booleane e delle istruzioni è orientata al loro uso nel controllo di flusso.

La traduzione è basata sui seguenti principi:

- Bisogna distinguere la sintassi delle espressioni booleane da quella delle espressioni aritmetiche;
- ogni espressione booleana è associata a due attributi ***ereditati*** *.true* e *.false* che rappresentano gli indirizzi cui viene passato il controllo nei cui l'espressione sia vera o falsa;
- ogni istruzione è associata a un attributo ***ereditato*** *.next* che rappresentano il label dell'istruzione successiva.
- l'attributo .*code* che contiene il codice è **sintetizzato**

Assegnazione e sequenza

- $P \rightarrow SL$ SL.*next* = *newlabel* ( )
  P.*code* = SL.*code* || label(SL.*next*) || ‘stop’

SDT (schema di traduzione)
P $\rightarrow$ {SL.*next* = *newlabel* ( )} **SL** {P.*code* = SL.*code* || label(SL.*next*) || ‘stop’}

- $SL \rightarrow SL_1$; S
  SL_1.*next* = *newlabel*( )
  S.*next* = SL.*next*
  SL.*code* = SL_1.*code* || *label*(SL_1.*next*) || S.*code*

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

- $SL \rightarrow S$ S.*next* = SL.*next* SL.*code* = S.*code*

SDT
SL $\rightarrow$ {S.next = SL_1.*next* } **S** {SL.*code* = S.*code*}

Assegnazione e sequenza

- S → **begin** SL **end** SL.*next* = S.*next* S.*code* = SL.*code*

  SDT

  SL → **begin** {S.next = SL_1.*next* } **S** **end** {SL.*code* = S.*code*}

- S → **id** = E ; S.*code* = E.*code* || *istore*(addr(**id**.*lexeme*))

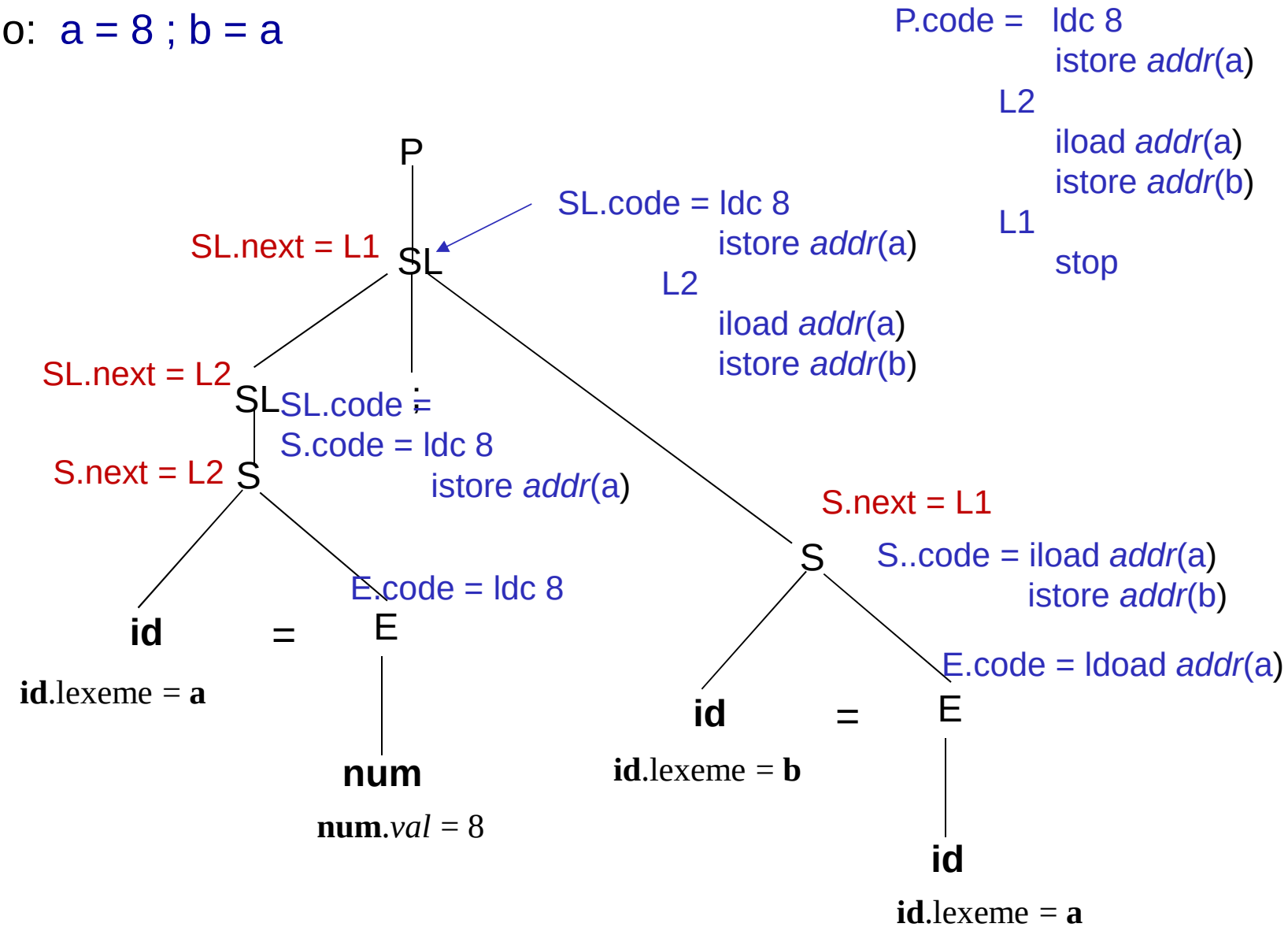
  SDT

  S → **id** = **E** {S.*code* = E.*code* || *istore*(addr(**id**.*lexeme*))}

Nota: per la traduzione deterministica top-down si deve anche eliminare la *ricorsione sinistra* e riscrivere le regole semantiche per le produzioni così ottenute.

esempio: assegnazione e concatenazione

Esempio: a = 8 ; b = a



Consideriamo solo il caso in cui le espressioni booleane siano usate in statement condizionali per modificare il flusso di controllo.

$B \rightarrow E == E$

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

SDD:

$B \rightarrow E_1 == E_2$ $\quad$ B.*code* = E_1.*code* || E_2.*code* || 'if_cmpeq B.*true*' || 'goto' B.*false*

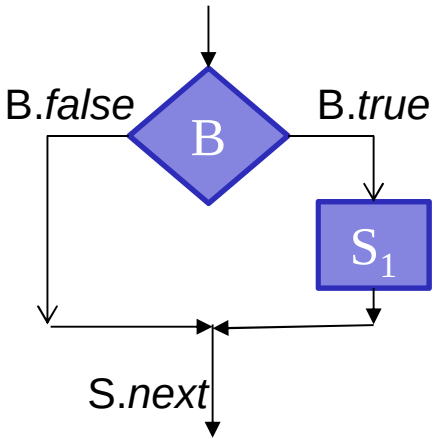
SDT

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

Nota: la traduzione è S-attribuita

Dal linguaggio ℙ al bytecode: istruzioni

$S \rightarrow$ **if** B **then** S_1



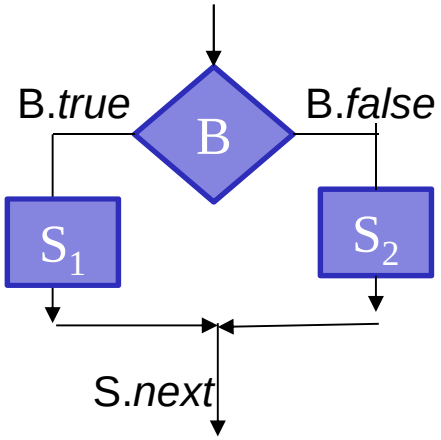
Viene creata una nuova label come valore per B.*true* per etichettare la prima istruzione della traduzione di S_1, da eseguire se B risulta vero, mentre nel caso in cui B sia falso l'istruzione da eseguire è quella successiva a S, quindi l'etichetta B.*false* è uguale a S.*next*. Anche dopo l'esecuzione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

SDT (schema di traduzione)
$S \rightarrow$ **if** {B.*true* = *newlabel*( ), B.*false* = S.*next* } **B then**
{S_1.*next* = S.*next*} **S_1** {S.*code* = B.*code* || *label*(B.*true*) || S_1.*code*}

Dal linguaggio ℙ al bytecode: istruzioni

$S \rightarrow$ **if** B **then** 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.*

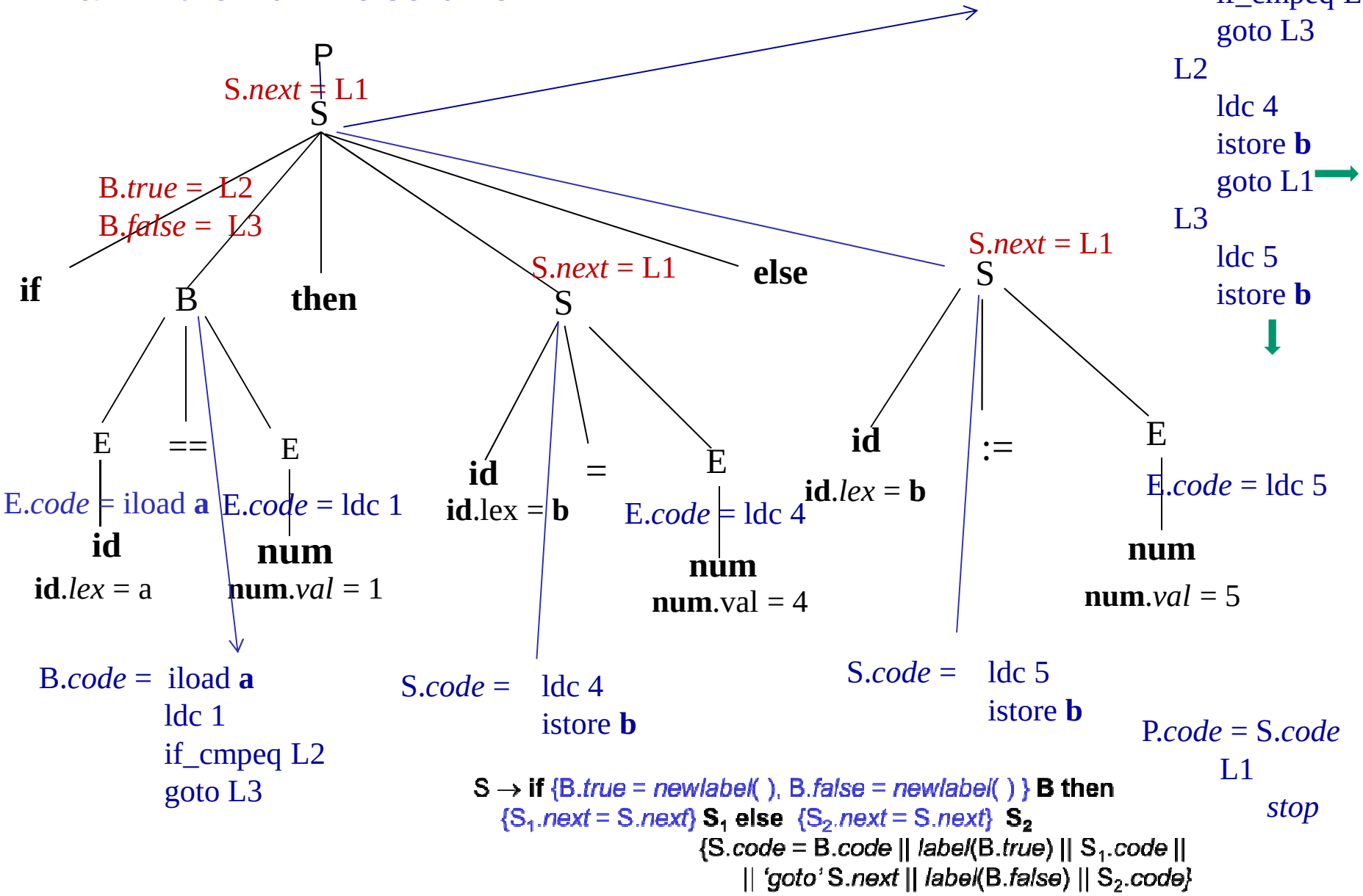
B.*true* = *newlabel*( )
B.*false* = *newlabel*( )
S_1.*next* = S_2.*next* = S.*next*
S.*code* = B.*code* || *label*(B.*true*) || S_1.*code* ||
|| *'goto'* S.*next* || *label*(B.*false*) || S_2.*code*

SDT

$S \rightarrow$ **if** {B.*true* = *newlabel*( ), B.*false* = *newlabel*( ) } **B then**
{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.*next* || *label*(B.*false*) || S_2.*code*}

Traduzione di statement: esempio

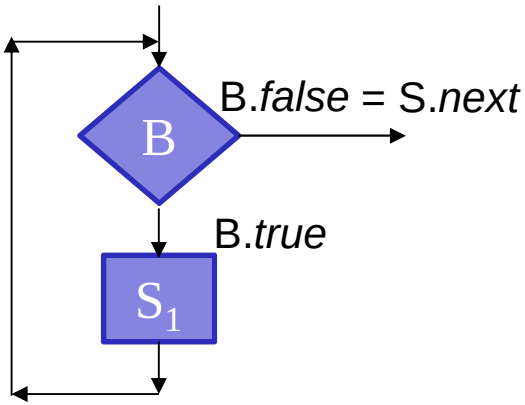
If a==1 then b:=4 else b:=5



$S \rightarrow$ **if** {B.*true* = *newlabel*( ), B.*false* = *newlabel*( ) } **B then**
{S_1.*next* = S.*next*} $\mathbf{S_1}$ **else** {S_2.*next* = S.*next*} $\mathbf{S_2}$
{S.*code* = B.*code* || *label*(B.*true*) || S_1.*code* ||
|| *'goto'* S.*next* || *label*(B.*false*) || S_2.*code*}

Dal linguaggio ℙ al bytecode: istruzioni

$S \rightarrow$ **while** B **do** 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*.

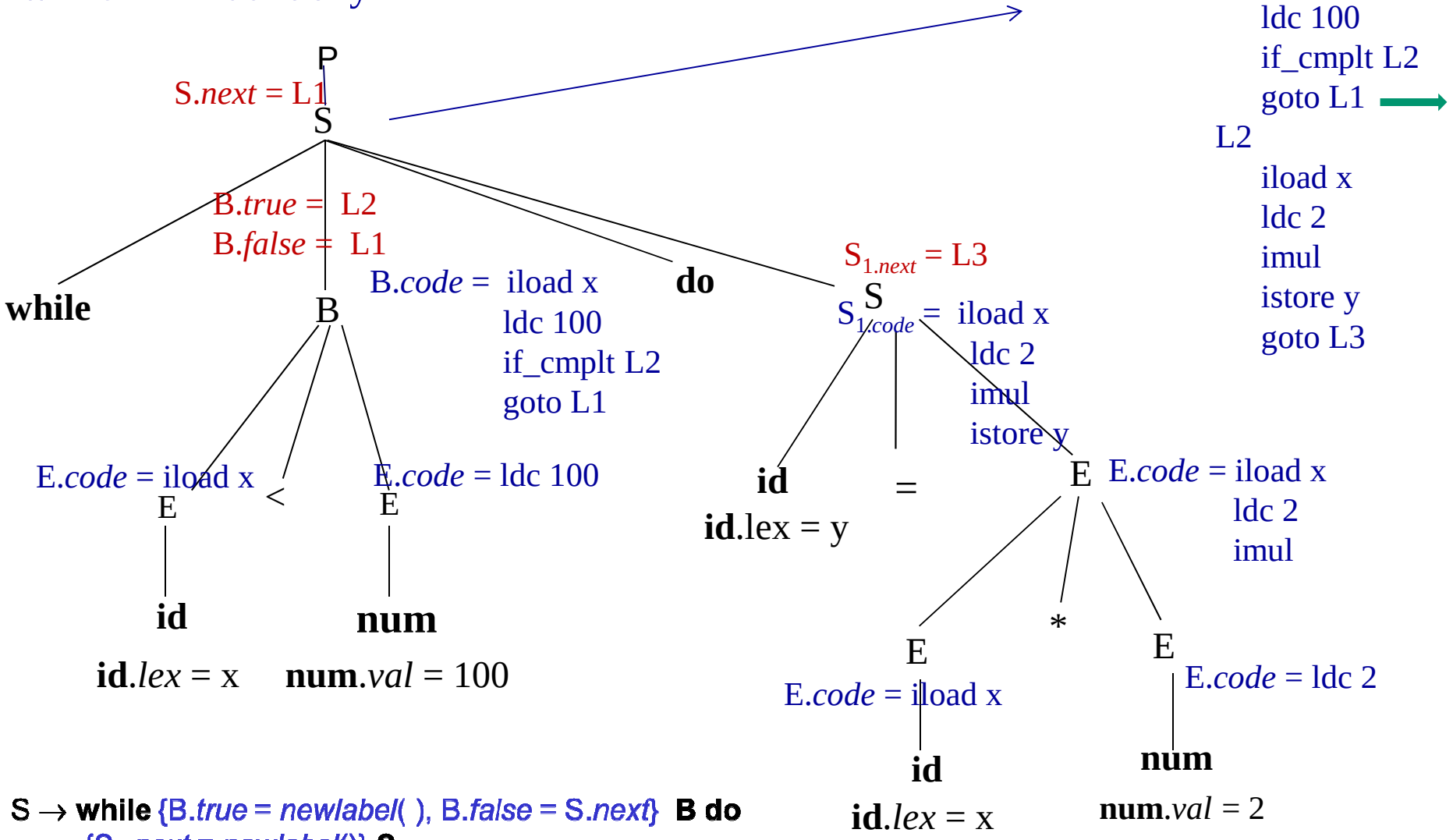
B.*true* = *newlabel*( )
B.*false* = S.*next*
S_1.*next* = *newlabel*()
S.*code* = *label*(S_1.*next*) || B.*code* || *label*(B.*true*) ||
|| S_1.*code* || 'goto' S_1.*next*

SDT

$S \rightarrow$ **while** {B.*true* = *newlabel*( ), B.*false* = S.*next*} **B do**
{S_1.*next* = *newlabel*()} **S_1**
{ S.*code* = *label*(S_1.*next*) || B.*code* || *label*(B.*true*) ||
S_1.*code* || 'goto' S_1.*next* }

Traduzione di statement: esempio

while x < 100 **do** y = x * 2



S → **while** {B.*true* = *newlabel*( ), B.*false* = S.*next*} **B do**
{S_1.*next* = *newlabel*()} $\mathbf{S_1}$
{ S.*code* = *label*(S_1.*next*) || B.*code* || *label*(B.*true*) ||
S_1.*code* || 'goto' S_1.*next* }

Traduzione di statement: tabella riassuntiva SDD

$P \rightarrow S$	S.*next* = *newlabel* ( ) P.*code* = S.*code* || label(S.*next*) || *emit*(Stop)
$S \rightarrow$ **id** = E ;	S.*code* = E.*code* || *istore*(addr(**id**.*lex*))
$S \rightarrow$ **if** B **then** S_1	B.*true* = newlabel ( ) B.*false* = S_1.*next* = S.*next* S.*code* = B.*code* || label(B.*true*) || S_1.*code*
$S \rightarrow$ **if** B **then** S_1 **else** S_2	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.*next*) || label(B.*false*) || S_2.*code*
$S \rightarrow$ **while** B **do** S_1	B.*true* = *newlabe*l ( ) B.*false* = S.*next* S_1.*next* = *newlabel*() S.*code* = *label*(S_1.*next*) || B.*code* || label(B.*true*) || || S_1.*code* || 'goto S_1.*next*'
$S \rightarrow$ **begin** SL **end**	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* || *label*(SL_1.*next*) || S.*code*
$SL \rightarrow S$	S.*next* = SL.*next* SL.*code* = S.*code*

Schemi di traduzione con codice ‘on-the fly’

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

S → **id** = E {*emit*(‘istore’ (**id**.*addr*))}

S → **if** {B.*true* = *newlabel*( ), B.*false* = S.*next*} B **then**
{*emitlabel*(B.*true*), S_1.*next* = S.*next* } S_1 {*gen*(‘goto’ B.*false*)}

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

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

B → E_1 == E_2 {*emit*(‘if_cmpeq’ B.*true*); *emit*(‘goto’ B.*false*)}

Schemi di traduzione con codice ‘on-the fly’

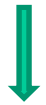
S → **begin** {SL.*next* = S.*next* } SL **end**

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

SL → {S.*next* = SL.*next* } S

Da schema a on-the-fly

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



P → {SL.*next* = *newlabel*( )} SL {*emitlabel*(S.*next*), *emit*('stop')}

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

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

$S \rightarrow$ **if** {B.*true* = *newlabel*( ), B.*false* = S.*next* } **B then**
{S_1.*next* = S.*next*} **S_1** {S.*code* = B.*code* || *label*(B.*true*) || S_1.*code*}

$S \rightarrow$ **if** {B.*true* = *newlabel*( ), B.*false* = S.*next*} **B then**
{*emitlabel*(B.*true*), S_1.*next* = S.*next* } **S_1** {*gen*('goto' B.*false*)}

$S \rightarrow$ **if** {B.*true* = *newlabel*( ), B.*false* = *newlabel*( ) } **B then**
{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.*next* || *label*(B.*false*) || S_2.*code*}

$S \rightarrow$ **if** {B.*true* = *newlabel*( ), B.*false* = newlabel( )} **B then**
{*emitlabel*(B.*true*), S_1.*next* = S.*next* } **S_1 else**
{*emit*(goto S.*next*), *emitlabel*(B.*false*)), S_2.*next* = S.*next* } **S_2**

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

S → **while** {B.*true* = *newlabel*( ), B.*false* = S.*next*, S_1.*next* = *newlabel*(),
emitlabel(S_1.*next*) } **B do** {*emitlabel*(B.*true*) } **S_1** {*emit*('goto' S_1.*next*)}

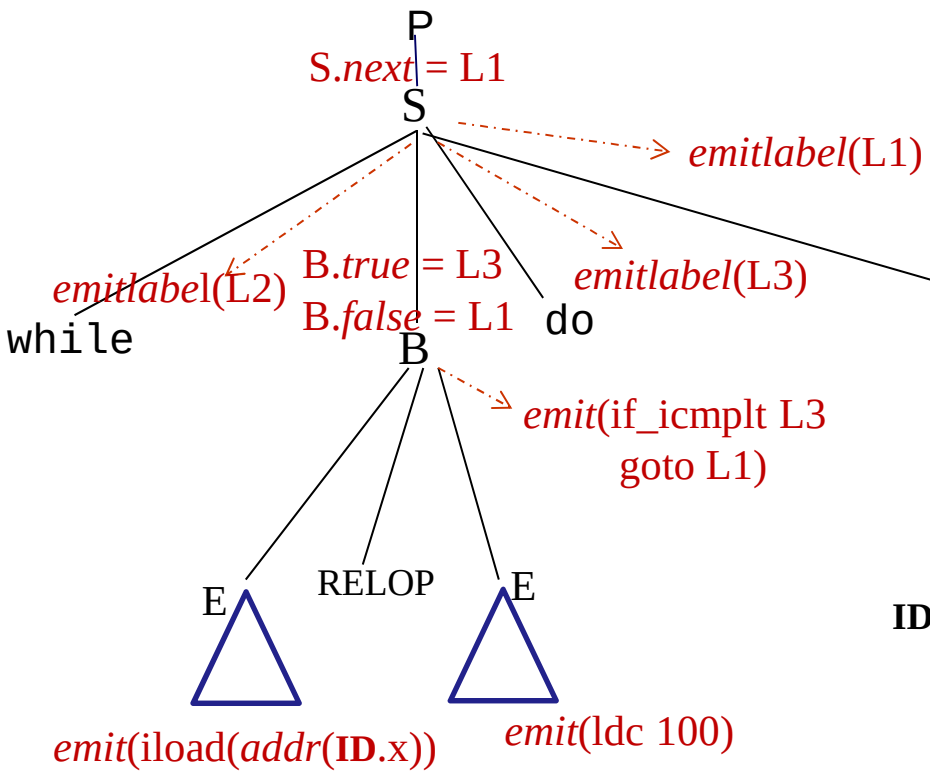
Traduzione ‘on-the-fly’: : statement ‘while’

$S \rightarrow$ `while` {S_1.*next* = *newlabel*(), B.*true* = *newlabel*( ),
B.*false* = S.*next*, *emitlabel*(S_1.*next*), } do
B {*emitlabel*(B.*true*)}
S_1 {*emit*(‘goto’ S_1.*next*)}

```
function S(S_next)
  var …
      ……
      if (cc = ‘while’)
            cc ← PROSS
            S1_next ← newlabel( )
            B_true ← newlabel( )
            B_false ← S_next
            emitlabel(S1_next)
            B(B_true, B_false)
            if (cc = ‘do’ )
                        cc ← PROSS
                        emitlabel(B_true)
                        S(S1_next)
                        emit(‘goto’ S1_next)
                  else Errore( )
            else Errore( )
            return
      else  …….
```

Traduzione ‘on-the-fly’

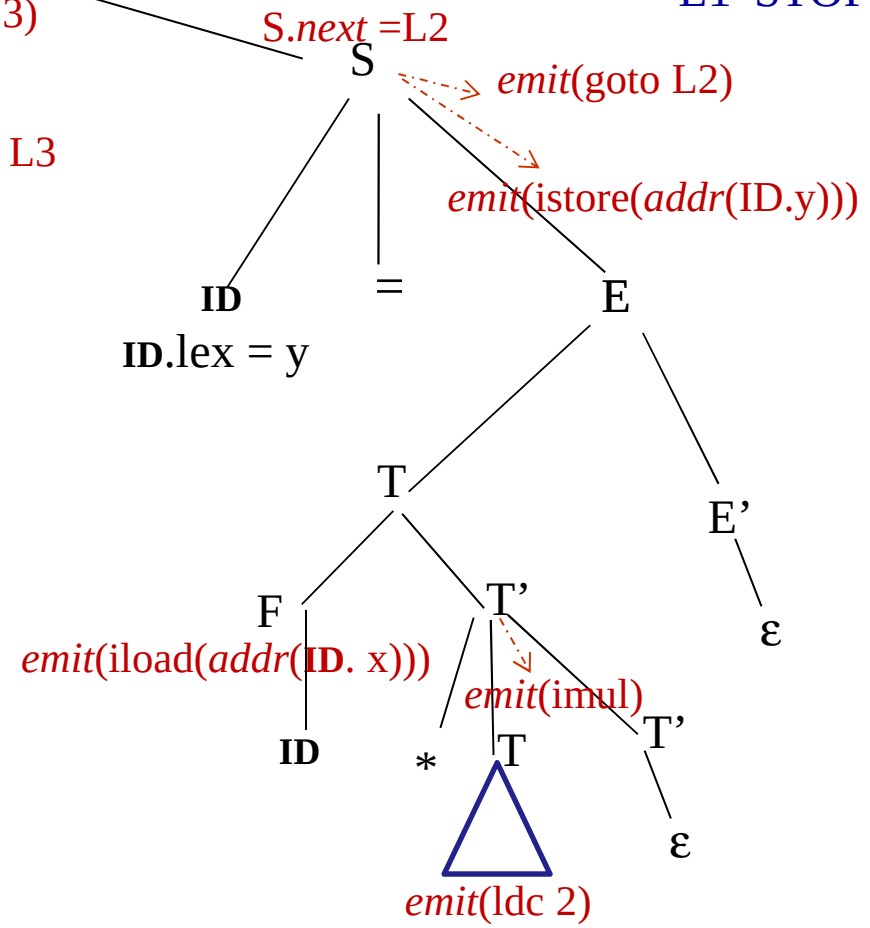
Esempio: `while` x < 100 `do` y = x ∗ 2 $



S

CODICE

```
L2  iload(addr(ID.x))
    ldc 100
    if_icmplt L3
    goto L1
L3  iload(addr(ID.x))
    ldc 2
    imul
    istore(addr(ID.y))
    goto L2
L1  STOP
```



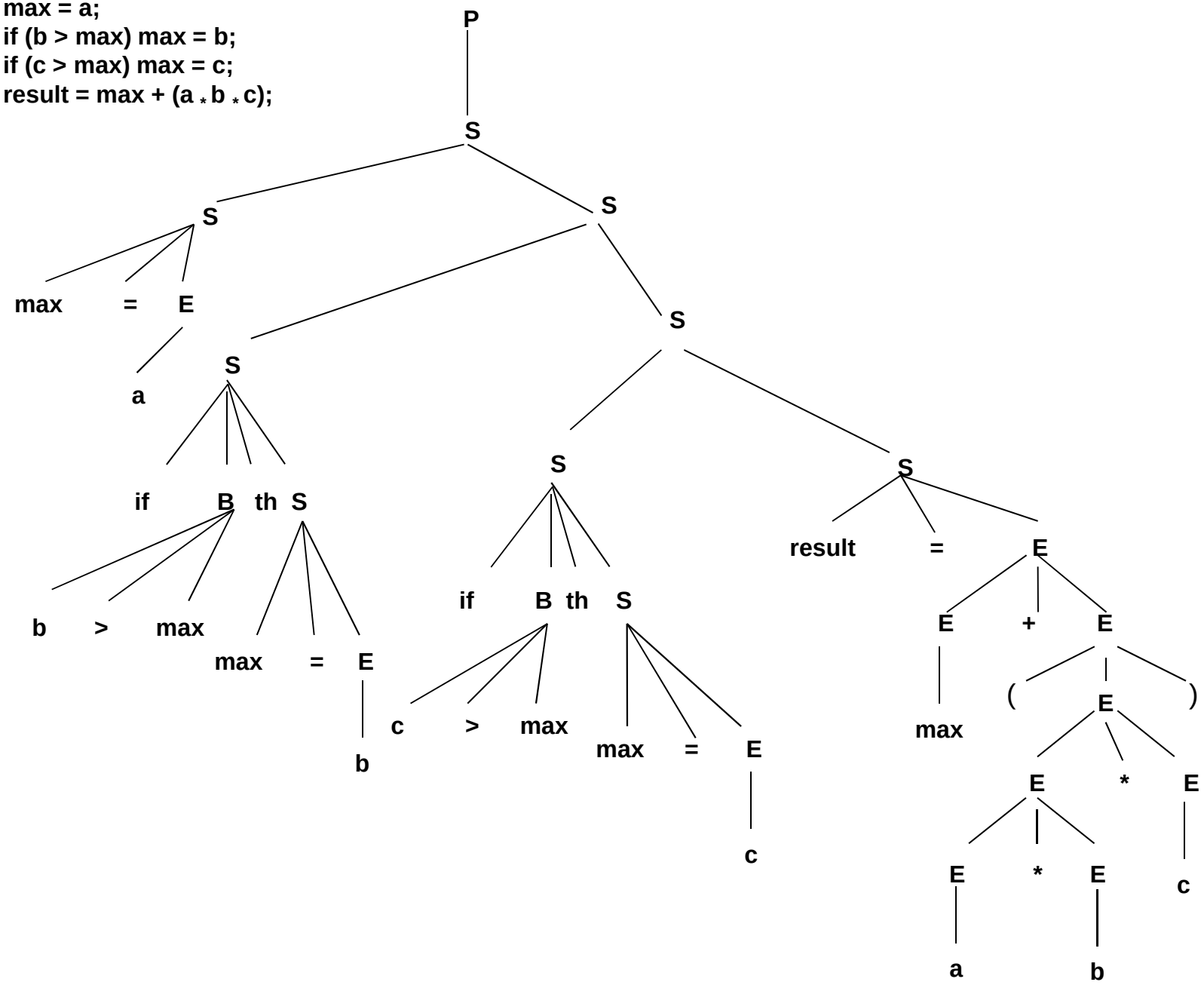
Dal linguaggio al bytecode: esempio esteso

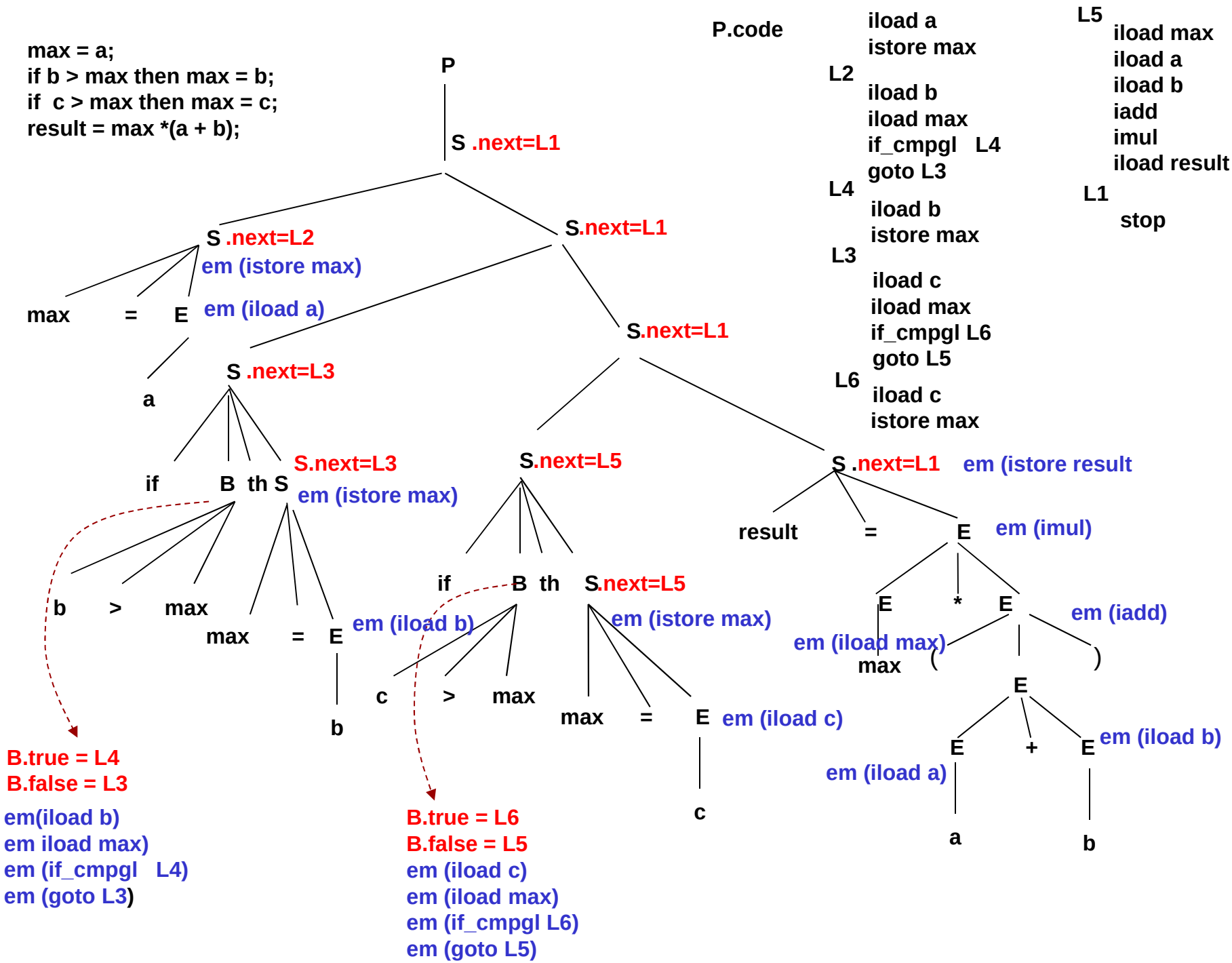
Il seguente programma nel linguaggio sorgente trova il massimo tra tre elementi e calcola il risultato di un'espressione aritmetica.

```
max = a;
if (b > max) max = b;
if (c > max) max = c;
result = max;
```

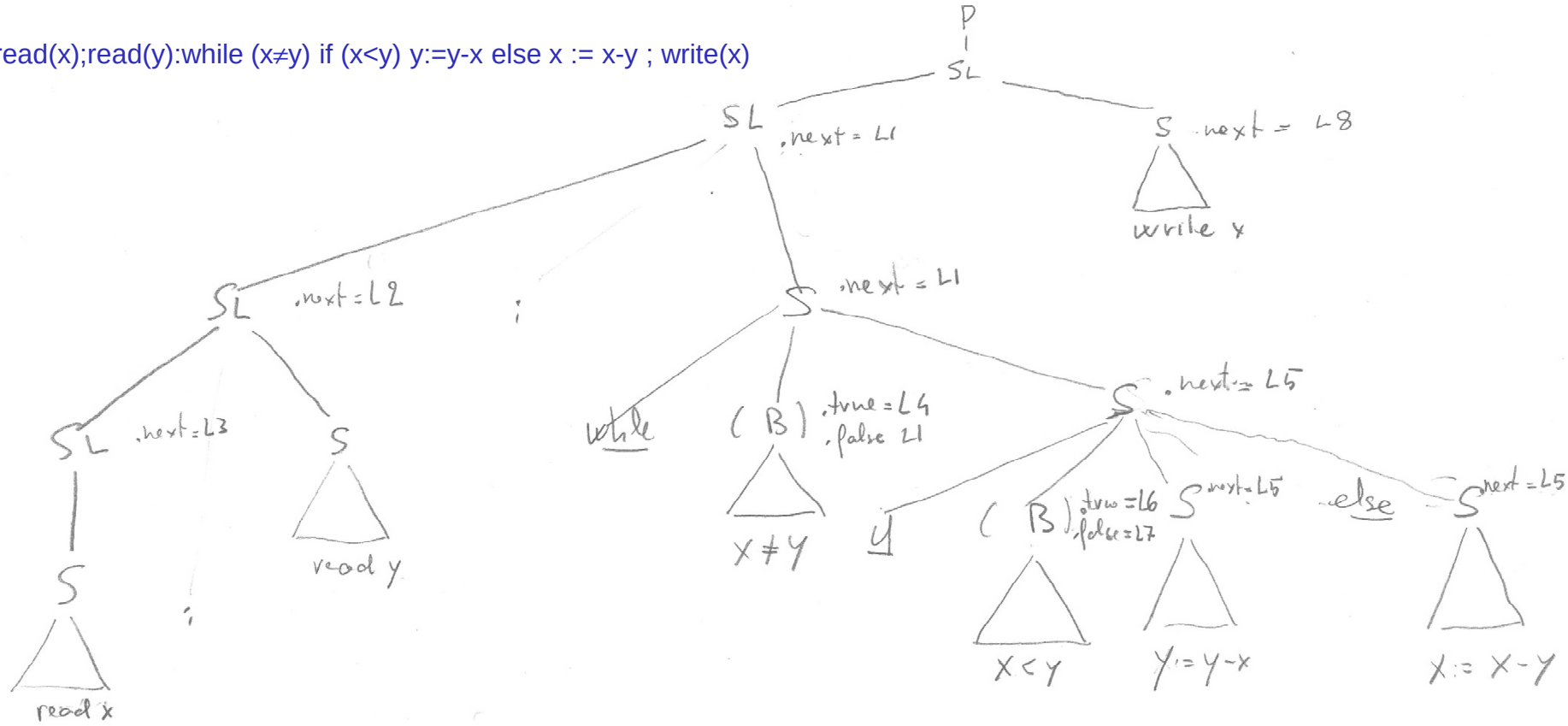
Costruiamo l'albero di parsificazione e valutiamo gli attributi in modo da fornire la traduzione nel codice a tre indirizzi.

max = a;
if (b > max) max = b;
if (c > max) max = c;
result = max + (a * b * c);




max = a;
if b > max then max = b;
if c > max then max = c;
result = max *(a + b);
P
S .next=L1
S .next=L2
em (istore max)
S.next=L1
max = E
em (iload a)
a
S .next=L3
S.next=L1
if B th S
S.next=L3
em (istore max)
S.next=L5
S .next=L1
em (istore result
b > max
max = E
em (iload b)
b
if B th S.next=L5
em (istore max)
c > max
max = E
em (iload c)
c
result = E
em (imul)
E * E
em (iadd)
em (iload max)
max
( )
E
E + E
em (iload b)
em (iload a)
a
b
B.true = L4
B.false = L3
em(iload b)
em iload max)
em (if_cmpgl L4)
em (goto L3)
B.true = L6
B.false = L5
em (iload c)
em (iload max)
em (if_cmpgl L6)
em (goto L5)
P.code
iload a
istore max
L2
iload b
iload max
if_cmpgl L4
goto L3
L4
iload b
istore max
L3
iload c
iload max
if_cmpgl L6
goto L5
L6
iload c
istore max
L5
iload max
iload a
iload b
iadd
imul
iload result
L1
stop

read(x);read(y):while (x≠y) if (x<y) y:=y-x else x := x-y ; write(x)




```
read x
L3
read y
L2
```

```
L5
iload x
iload y
if_cmpne L4
goto L1
```

```
L4
iload x
iload y
if_cmplt L6
goto L7
```

```
L6
iload y
iload x
isub
istore y
goto L5
L7
```

```
L7
iload x
iload y
isub
istore x
goto L5
L1 write x
L8 stop
```

Esempio di prima (MCD) sviluuppato meglio

Programma (parte da valori di x e y fisati)

```
x = 49 ;
y = 21 ;
while x != y do
    if x < y  then y = y – x
              else x = x – y
EOF
```

NOTA: al posto delle istruzioni di read qui si assegna un valore iniziale ad x e y e si omette l'istruzione di write finale

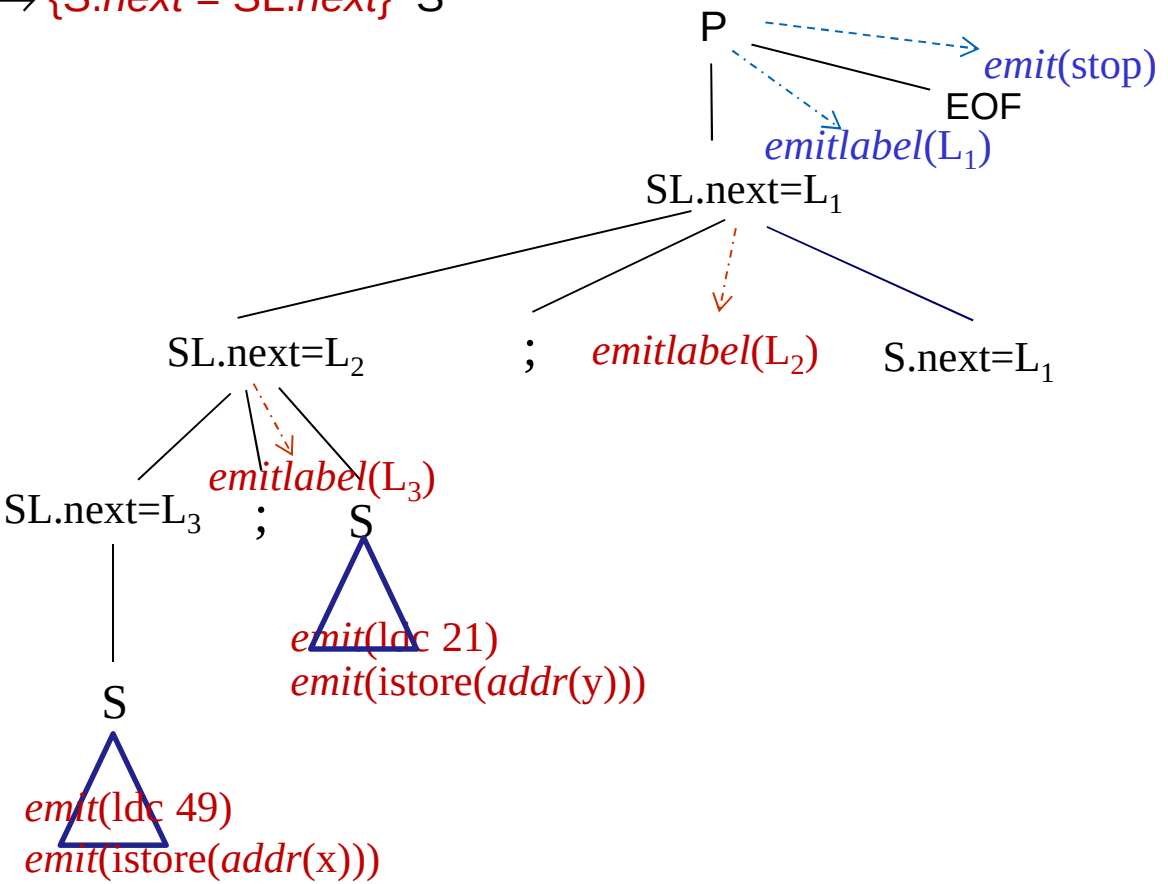
Traduzione ‘on-the-fly’

`x = 49 ; y = 21 ; while x != y do if x < y then y = y – x else x = x – y` EOF

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

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

SL → {S.*next* = SL.*next*} S

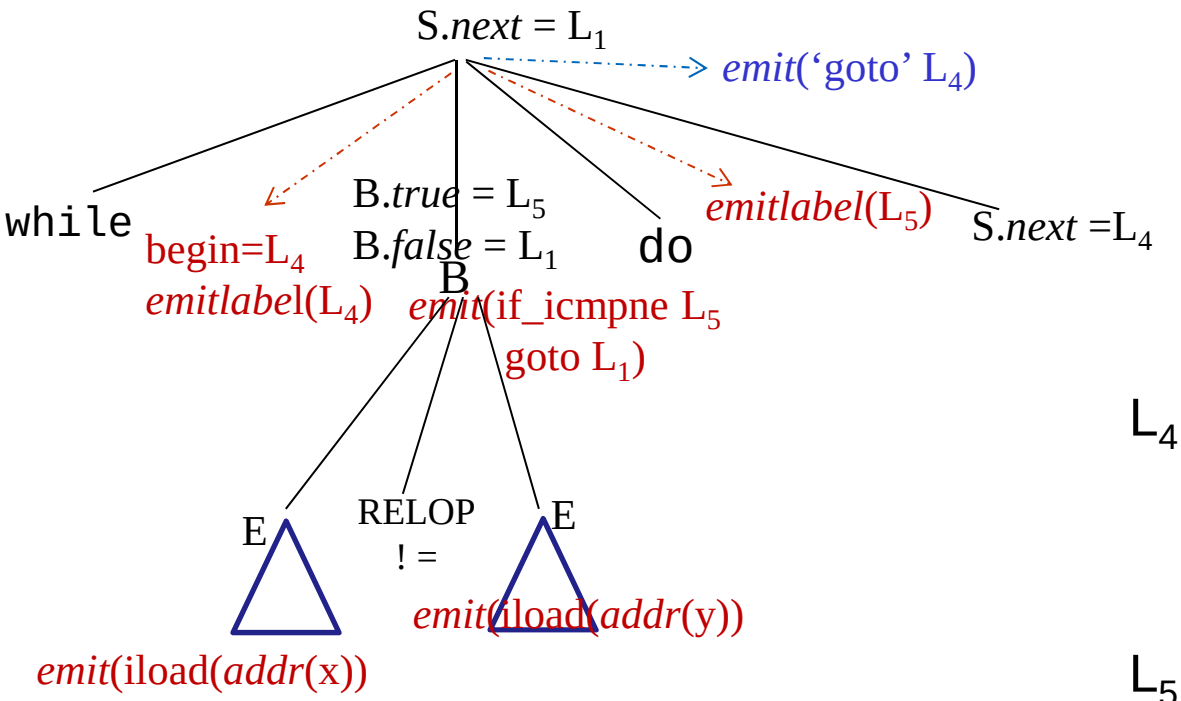


ldc 49
istore(*addr*(x)
L_3 ldc 21
istore(*addr*(y))
L_2

Traduzione ‘on-the-fly’

`x = 49 ; y = 21 ; while x != y do if x < y then y = y – x else x = x – y EOF`

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



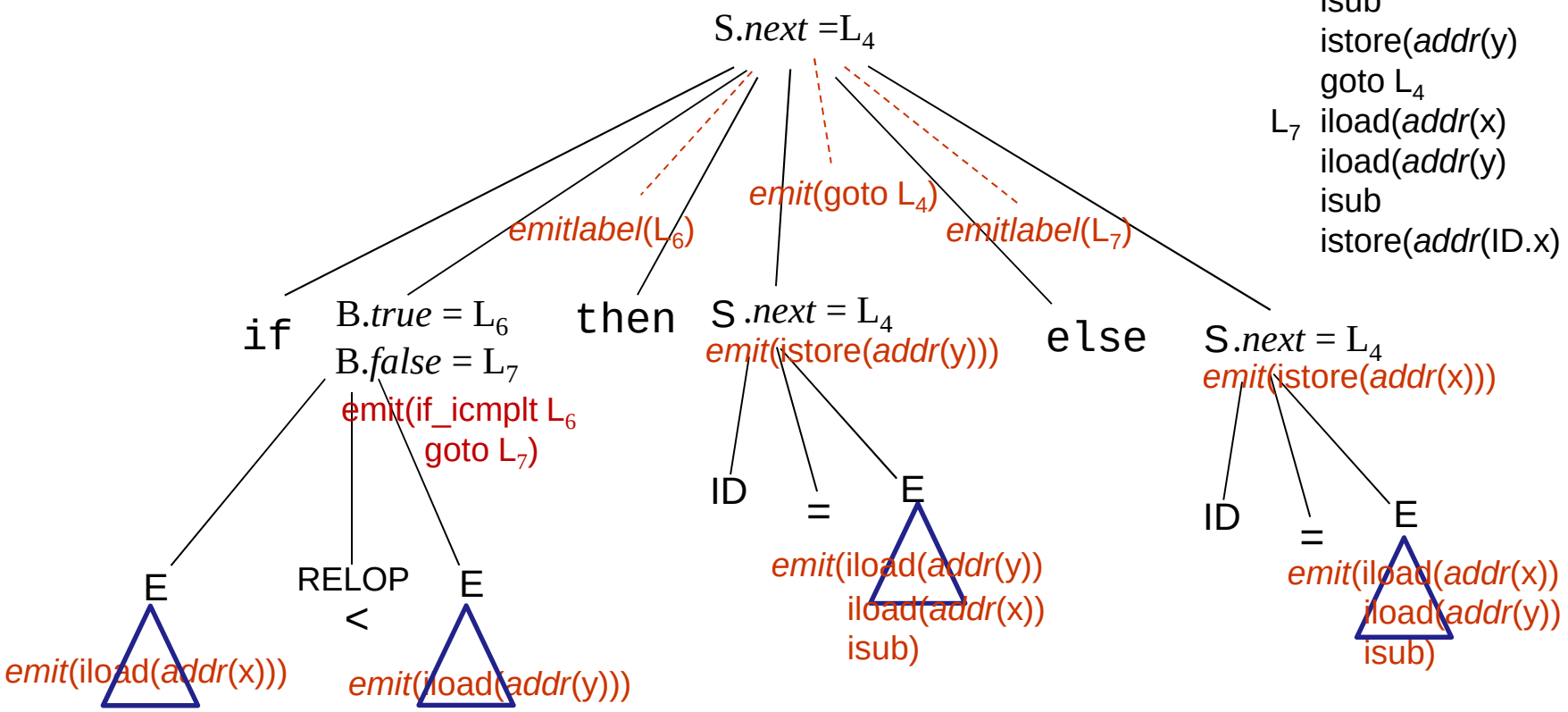
L4 iload(*addr*(x)
iload(*addr*(y)
if_cmpne L5
goto L1
L5

Traduzione ‘on-the-fly’

`x = 49 ; y = 21 ; while x != y do if x < y then y = y – x else x = x – y EOF`

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

```
    iload(addr(x)
    iload(addr(y)
    if_cmplt L6
    goto L7
L6  iload(addr(y)
    iload(addr(x)
    isub
    istore(addr(y)
    goto L4
L7  iload(addr(x)
    iload(addr(y)
    isub
    istore(addr(ID.x)
```



Traduzione ‘on-the-fly’

`x = 49 ; y = 21 ; while x != y do if x < y then y = y – x else x = x – y` EOF

```
   ldc 49
   istore(addr(x)
L3 ldc 21
   istore(addr(y)
L2
L4 iload(addr(x)
   iload(addr(y)
   if_cmpne L5
   goto L1
L5 iload(addr(x)
   iload(addr(y)
   if_cmplt L6
   goto L7
L6 iload(addr(y)
   iload(addr(x)
   isub
   istore(addr(y)
   goto L4
L7 iload(addr(x)
   iload(addr(y)
   isub
   istore(addr(x)
   goto L4
L1 stop
```

Esercizi

1. Costruire l'albero sintattico annotato per generare il bytecode per la seguente espressione:

   a + (**b** - **c**) + **d**.

2. Fornire la traduzione nel bytecode degli statement:

   S = **if** a>b **then** a = a + a

   S **= if** a < b **then** x = y **else** x =1 - 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 **then** a = a - b **else if** a < b **then** b = b + a

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

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

   repeat S **until** B