Recap: Code Generation for while

\[
[\text{while } (\text{cond}) \text{ stmt}] = \\
\text{nStart: } [\text{cond}] \\
\text{ifeq(nExit)} \\
[\text{stmt}] \\
\text{goto(nStart)} \\
\text{nExit: }
\]

\[
\text{nStart} \rightarrow (\text{cond})? \\
\rightarrow (\neg\text{cond})? \\
\rightarrow \text{nStart} \\
\text{nExit} \rightarrow \text{stmt} \\
\rightarrow \text{nExit}
\]
\[ [e_1 < e_2] = \]
\[
[e_1] \\
[e_2]
\]

if_icmplt(nTrue)

iconst_0
goto(nExit)

nTrue:  iconst_1

nExit:
Compare Two Translations

```java
while (counter < to) {
    counter = counter + step;
}
```

Translation 1:
```
nBegin: iload #counter
    iload #to
    if_icmplt nTrue
    iconst_0
    goto nAfter
nTrue:  icnst_1
nAfter: ifeq nExit
    iload #counter
    iload #step
    iadd
    istore #counter
    goto nBegin
nExit:
```

Translation 2:
```
nBegin: iload #counter
    iload #to
    if_icmplt nBody
    goto nExit
nBody:  iload #counter
    iload #step
    iadd
    istore #counter
    goto nBegin
nExit:
```
Compare Two Translations

```java
while (counter < to) {
    counter = counter + step;
}
```

Translation 1:

```java
nBegin: iload #counter
    iload #to
    if_icmplt nTrue
    iconst_0
    goto nAfter
nTrue:  iconst_1
nAfter: ifeq nExit
    iload #counter
    iload #step
    iadd
    istore #counter
    goto nBegin
nExit:
```

Translation 2:

```java
nBegin: iload #counter
    iload #to
    if_icmplt nBody
    goto nExit
nBody:  iload #counter
    iload #step
    iadd
    istore #counter
    goto nBegin
nExit:
```

Translation 2 immediately jumps to body, no intermediate result for while condition.
Macro branch Instruction

- Introduce an imaginary big instruction
  \[ \text{branch}(c, n\text{True}, n\text{False}) \]
  - \( c \) is a potentially complex Java boolean expression
    - Main reason why branch is not a real instruction
  - \( n\text{True} \) is label to jump to when \( c \) evaluates to true
  - \( n\text{False} \) is label to jump to when \( c \) evaluates to false
  - No “fall through” - always jumps (symmetrical)

We show how to:

- Use branch to compile \textit{if}, \textit{while}, etc.
- Expand branch recursively into concrete bytecodes
Using `branch` in Compilation

\[
\begin{align*}
[\text{if } (c) \ t \ \text{else } e] &= \text{branch}(c, n\text{True, nFalse}) \\
n\text{True}: &\quad [t] \\
goto(n\text{After}) \\
n\text{False}: &\quad [e] \\
n\text{After}: &
\end{align*}
\]

\[
\begin{align*}
[\text{while } (c) \ s] &= \text{branch}(c, n\text{Body, nExit}) \\
n\text{Test}: &\quad \text{branch}(c, n\text{Body, nExit}) \\
n\text{Body}: &\quad [s] \\
goto(n\text{Test}) \\
n\text{Exit}: &
\end{align*}
\]
Decomposing \textit{branch}

\[
\text{branch}(\neg c, n\text{Then}, n\text{Else}) = \text{branch}(c, n\text{Else}, n\text{Then})
\]

\[
\text{branch}(c_1 \land c_2, n\text{Then}, n\text{Else}) = \begin{align*}
\text{branch}(c_1, n\text{Next}, n\text{Else}) \\
n\text{Next}: \quad \text{branch}(c_2, n\text{Then}, n\text{Else})
\end{align*}
\]

\[
\text{branch}(c_1 \lor c_2, n\text{Then}, n\text{Else}) = \begin{align*}
\text{branch}(c_1, n\text{Then}, n\text{Next}) \\
n\text{Next}: \quad \text{branch}(c_2, n\text{Then}, n\text{Else})
\end{align*}
\]

\[
\text{branch}(\text{true}, n\text{Then}, n\text{Else}) = \text{goto } n\text{Then}
\]

\[
\text{branch}(\text{false}, n\text{Then}, n\text{Else}) = \text{goto } n\text{Else}
\]

\[
\text{boolean variable } b \text{ with slot } N
\]

\[
\text{branch}(b, n\text{Then}, n\text{Else}) = \begin{align*}
iload_N \\
\text{ifeq } n\text{Else} \\
goto n\text{Then}
\end{align*}
\]
branch\( (e_1 \text{ R } e_2, n\text{Then}, n\text{Else}) = \)

\[
\begin{align*}
[e_1] \\
[e_2] \\
\text{if\_icmp}(\text{R}(n\text{Then})) \\
\text{goto}(n\text{Else})
\end{align*}
\]

\textbf{R can be} <, >, ==, !=, <=, >=,...
• Consider storing $x = c$ where $x, c$ are boolean and $c$ has &&, ||

• How to put result of branch on stack to allow istore?

\[
\begin{align*}
[x = c] &= \\
&= \text{branch}(c, n\text{Then}, n\text{Else}) \\
n\text{Then}: &\quad \text{iconst}_1 \\
&\quad \text{goto}(n\text{After}) \\
n\text{Else}: &\quad \text{iconst}_0 \\
n\text{After}: &\quad \text{istore} \#x
\end{align*}
\]
Fewer push/pop of boolean constants compared to previous translation

```java
if ((x < y) && !((y < z) && cond))
    return
else
    y = y + 1
```

```java
branch(x<y,n1,else)
n1:  branch(y<z,n2,then)
then: return
goto after
n2:  branch(cond,else,then)
else: iload #y
    iconst_1
    iadd
    astore #y
after:
```
Implementing `branch`

- Option 1: emit code using `branch`, then rewrite
- Option 2: `branch` is a just a function in the compiler that expands into instructions

```
branch(c,nTrue,nFalse)
```

```
public List<Bytecode> compileBranch(Expression c, Label nTrue, Label nFalse) {
    ...
}
```

- The function takes two destination labels
break Statement

• A common way to exit from a loop is to use a `break` statement

```java
while (true) {
    code1
    if (cond) break;
    cond2
}
```

• Consider a language that has expressions, assignments, blocks {...}, `if`, `while`, and a `break` statement

• `break` statement exits the innermost loop and can appear inside arbitrarily complex blocks and `if` conditions

• How would translation scheme for such construct look like?

• We need a generalization of compilation functions […]
• Pass a **label** to compilation functions \([\cdots]\) indicating to which instructions to jump after they finish
  - No fall-through

```
[x = e] after =       // new parameter 'after'
[e]
istore #x
goto(after)       // at the end jump to it
```

```
[s_1; s_2] after =
  [s_1] freshL
freshL:  [s_2] after
```

we could have any junk in here
because \([s_1] freshL\) ends in a jump
Translation of if, while, return

\[
[\text{if } (c) \ t \ \text{else } e] \ \text{after} = \\
\quad \text{branch}(c, n\text{True}, n\text{False}) \\
\quad n\text{True}: \ [t] \ \text{after} \\
\quad n\text{False}: \ [e] \ \text{after}
\]

\[
[\text{while } (c) \ s] \ \text{after} = \\
\quad n\text{Test}: \ \text{branch}(c, n\text{Body}, \text{after}) \\
\quad n\text{Body}: \ [s] \ n\text{Test}
\]

\[
[\text{return } e] \ \text{after} = \\
\quad [e] \\
\quad \text{ireturn}
\]
[if (x < y) return; else y = 2;] after =
   iload #x
   iload #y
   if_icmplt nTrue
   goto nFalse

nTrue:  return
nFalse:  iconst_2
         istore #y
         goto after

Note: no goto after return because

• translation of if does not generate goto as it did before, since it passes it to the translation of the body
• translation of return knows it can ignore the after parameter
Two Destination Parameters

\[
[s_1; s_2] \text{ after brk } = \\
[s_1] \text{ freshL brk} \\
\text{freshL: } [s_2] \text{ after brk} \\
\text{[break] after brk } = \\
goto \text{ brk}
\]

\[
[x = e] \text{ after brk } = \\
[e] \\
\text{istore } #x \\
\text{goto after}
\]

\[
[\text{return } e] \text{ after brk } = \\
[e] \\
\text{ireturn}
\]

\[
[\text{while } (c) s] \text{ after brk } = \\
\text{test: } \text{branch}(c, \text{body}, \text{after}) \\
\text{body: } [s] \text{ test after} \\
\]

this is where the second parameter gets bound to the exit of the loop
if with two parameters

\[ \text{[if (c) t else e] after brk} = \]

\[ \text{branch (c, nTrue, nFalse)} \]

\[ \text{nTrue: [t] after brk} \]

\[ \text{nFalse: [e] after brk} \]
break and continue

[break] after brk cont =
    goto brk

[continue] after brk cont =
    goto cont

[while (c) s] after brk cont =
    nTest: branch (c, nBody, after)
    nBody: [s] nTest after nTest