# Code Generation: Notation

We use brackets, $[s]$ to denote "result of compiling s".
For compilation of expressions, we can thus write as follows.

$[\, e_1 + e_2 \,]$ =
  $[e_1]$
  $[e_2]$
  **i32.add**

$[\, e_1 * e_2 \,]$ =
  $[e_1]$
  $[e_2]$
  **i32.mul**

# Sequential Composition

How to compile statement sequence?

$$s_1; s_2; \ldots; s_N$$

# Sequential Composition

How to compile statement sequence?

$$s_1; s_2; \ldots; s_N$$

Solution: concatenate bytecodes for each statement:

$$[\![ s_1; s_2; \ldots; s_N ]\!] =$$
$$[\![ s_1 ]\!]$$
$$[\![ s_2 ]\!]$$
$$\ldots$$
$$[\![ s_N ]\!]$$

# Same Thing in Scala-Like Notation

```
def compileStmt(e: Stmt): List[Bytecode] = e match {
  ...
  case Sequence(sts) =>
    for { st <- sts;
          bcode <- compileStmt(st)
    } yield bcode
  ...
}
```

In other words, the case of sequence returns flatMap with recursive call:

```
  ...
  case Sequence(sts) => sts.flatMap(compileStmt)
  ...
```

In practice, concatenating lots of lists is inefficient.
We can use e.g. imperative append.

# Compiling Control: Example

```
int count(int counter,
          int to,
          int step) {
  int sum = 0;
  do {
    counter = counter + step;
    sum = sum + counter;
  } while (counter < to);
  return sum; }
```

We need to see how to:

- translate boolean expressions
- generate jumps for control

```
(func $func0
 (param $var0 i32) (param $var1 i32)
 (param $var2 i32) (result i32)
 (local $var3 i32)
  i32.const 0
  set_local $var3
  loop $label0
    get_local $var3
    get_local $var0
    get_local $var2
    i32.add
    tee_local $var0
    i32.add
    set_local $var3
    get_local $var0
    get_local $var1
    i32.lt_s
    br_if $label0
  end $label0
  get_local $var3  )
```

# Representing Booleans

"All comparison operators yield 32-bit integer results with 1 representing true and 0 representing false." – WebAssembly spec

Our generated code uses 32 bit int to represent boolean values in: **local variables**, **parameters**, and intermediate **stack values**.

**1**, representing true

**0**, representing false

i32.eq: sign-agnostic compare equal

i32.ne: sign-agnostic compare unequal

i32.lt_s: signed less than

i32.le_s: signed less than or equal

i32.gt_s: signed greater than

i32.ge_s: signed greater than or equal

i32.eqz: compare equal to zero (return 1 if operand is zero, 0 otherwise) // not

# Truth Values for Relations: Example

```
int test(int x, int y){
  return (x < y);
}
```

```
(func $func0
  (param $var0 i32)
  (param $var1 i32)
  (result i32)

  get_local $var0
  get_local $var1
  i32.lt_s
)
```

# Comparisons, Conditionals, Scoped Labels

```c
int fun(int x, int y){
 int res = 0;
 if (x < y) {
  res = (y / x);
 } else res = (x / y);
 return res+x+y;
}
```

```
(local $var2 i32)
block $label1 block $label0
 get_local $var0
 get_local $var1
 i32.ge_s
 br_if $label0        // to else branch
 get_local $var1
 get_local $var0
 i32.div_s
 set_local $var2
 br $label1           // done with if
end $label0           // else branch
 get_local $var0
 get_local $var1
 i32.div_s
 set_local $var2
end $label1           // end of if
get_local $var1
get_local $var0
i32.add
get_local $var2
i32.add
```

# Main Instructions for Labels

- **block**: the beginning of a block construct, a sequence of instructions with a **label at the end**
- **loop**: a block with a label at the **beginning** which may be used to form loops
- **br**: branch to a given label in an enclosing construct
- **br_if**: conditionally branch to a given label in an enclosing construct
- **return**: return zero or more values from this function
- **end**: an instruction that marks the end of a block, loop, if, or function

# Compiling If Statement

Notation for compilation:

**[ if (**cond**)** tStmt **else** eStmt **]** =

**block** $nAfter **block** $nElse

**[** !cond **]**

**bf_if $nElse**

**[** tStmt **]**

**br $nAfter**

**end $nElse**:

**[** eStmt **]**

**end $nAfter**:

```
block $label1 block $label0
  (negated condition code)
  br_if $label0        // to else branch
    (true case code)
  br $label1           // done with if
end $label0            // else branch
    (false case code)
end $label1            // end of if
```

Is there alternative without negating condition?

# How to introduce labels

- For forward jumps to $label: use
  **block** $label
  …
  **end** $label

- For backward jumps to $label: use
  **loop** $label
  …
  **end** $label

# WebAssembly's *if*

WebAssembly has dedicated bytecodes for if expressions, i.e., *if*, *else*, *end*:

$[e_{cond}]$
**if**
  $[e_{then}]$
**else**
  $[e_{else}]$
**end**
$[e_{rest}]$

▷ Given the *block* and *br[_if]* instructions you saw this construct isn't necessary. How can we desugar snippets like the above?

# WebAssembly's *if*

▷ Given the *block* and *br[_if]* instructions you saw this construct isn't necessary. How can we desugar snippets like the above?

```
block nAfter
  block nElse
    [! e_cond]
    br_if nElse
    [e_then]
    br nAfter
  end //nElse:
  [e_else]
end //nAfter:
[e_rest]
```

# WebAssembly's *if*

▷ Given the *block* and *br[_if]* instructions you saw this construct isn't necessary. How can we desugar snippets like the above?

```
block nAfter
  block nElse
    [! e_cond]
    br_if nElse
    [e_then]
    br nAfter
  end //nElse:
  [e_else]
end //nAfter:
[e_rest]
```

▷ Can we avoid the negation on the branching condition $e_{cond}$?

# Avoiding negation

- Can we avoid the negation on the branching condition $e_{cond}$?

```
block nAfter
  block nThen
    [e_cond]
    br_if nThen
    [e_else]
    br nAfter
  end //nThen:
  [e_then]
end //nAfter:
[e_rest]
```

# Translating control flow structures more efficiently

Introduce an imaginary large instruction **`branch`**`(c,nThen,nElse)`.

Here `c` is a potentially complex boolean expression (the main reason why **`branch`** is not a built-in bytecode instruction),
whereas `nTrue` and `nFalse` are the labels we jump to depending on the boolean value of `c`.

We will show how to

- use **`branch`** to compile **`if`** and short-circuiting operators,
- by expanding **`branch`** recursively into concrete bytecode instructions.

# Translating control flow structures more efficiently

[*if* ($e_{cond}$) $e_{then}$ *else* $e_{else}$] :=

```
block nAfter
  block nElse
    block nThen
      branch(e_cond, nThen, nElse)
    end //nThen:
    [e_then]
    br nAfter
  end //nElse:
  [e_else]
end //nAfter:
[e_rest]
```

# Decomposing conditions in branch

```
branch(!e,nThen,nElse) :=
  branch(e,nElse,nThen)

branch(e1 && e2,nThen,nElse) :=
  block nLong
    branch(e1,nLong,nElse)
  end //nLong:
  branch(e2,nThen,nElse)

branch(e1 || e2,nThen,nElse) :=
  block nLong
    branch(e1,nThen,nLong)
  end //nLong:
  branch(e2,nThen,nElse)
```

# Decomposing conditions in `branch`

```
branch(true,nThen,nElse) :=
  br nThen

branch(false,nThen,nElse) :=
  br nElse

branch(b,nThen,nElse) :=   (where b is a local var)
  get_local #b
  br_if nThen
  br nElse
```

# Decomposing conditions in branch

**branch**($e_1 == e_2$,nThen,nElse) $:=$ ($\textit{where } e_1, e_2 \textit{ are of type } int$)

```
  [e1]
  [e2]
  i32.eq
  br_if nThen
  br nElse
```

... *analogously for other relations*

# Returning the result from branch

Consider storing $x = c$
where $x, c$ are boolean and $c$ contains $\&\&$ or $\|$.

How do we put the result of $c$ on the stack so it can be stored in $x$?

$[x = c] :=$

```
block nAfter
  block nElse
    block nThen
      branch(c,nThen,nElse)
    end //nThen:
    i32.const 1
    br nAfter
  end //nElse:
  i32.const 0
end //nAfter:
set_local #x
```

# Destination label parameters

Recall that in **`branch`**`(c,nThen,nElse)` we had two arguments `nThen` and `nElse`, which told us where to jump to execute code of the corresponding branches.

Similarly, up until now we explicitly enclosed our translated program fragments in an `nAfter` block, so we could jump to the “rest” of the program.

# Destination label parameters

Recall that in **`branch`**`(c,nThen,nElse)` we had two arguments `nThen` and `nElse`, which told us where to jump to execute code of the corresponding branches.

Similarly, up until now we explicitly enclosed our translated program fragments in an `nAfter` block, so we could jump to the "rest" of the program.

$\Rightarrow$ We can generalize our translation function $[\,\cdot\,]$ to take a destination label designating the "rest" in the surrounding code.

# Destination label parameters

Recall that in **`branch`**`(c,nThen,nElse)` we had two arguments `nThen` and `nElse`, which told us where to jump to execute code of the corresponding branches.

Similarly, up until now we explicitly enclosed our translated program fragments in an `nAfter` block, so we could jump to the “rest” of the program.

⇒ We can generalize our translation function $[\![\cdot]\!]$ to take a destination label designating the “rest” in the surrounding code.

$$[\![\cdot]\!] \Rightarrow [\![\cdot]\!]\,\texttt{nAfter}$$

⇒ The caller of the translation function determines where to continue!

# Translations with an `nAfter` label parameter (1)

```
[x = e] nAfter :=
  block nSet
    [e] nSet
    // note that the rest of this block is never reached!
  end //nSet:
  set_local #x
  br nAfter

[s1; s2] nAfter :=
  block nSecond
    [s1] nSecond
  end //nSecond:
  [s2] nAfter
```

# Translations with an `nAfter` label parameter (2)

```
[if (e_cond) e_then else e_else] nAfter :=
  block nElse
    block nThen
      branch(e_cond,nThen,nElse)
    end //nThen:
    [e_then] nAfter
  end //nElse:
  [e_else] nAfter

[return e] nAfter :=
  block nRet
    [e] nRet
  end //nRet:
  return
```

# Switch statements

Let us assume our language had a switch statement (like C and Java do, for instance):

**switch** $(e_{scrutinee})$ {
  **case** $c_1$**:** $e_1$
  **...**
  **case** $c_n$**:** $e_n$
  **default:** $e_{default}$
}

▷ How can we compile such switch statements?

# Compiling switch statements

```
[s_switch] nAfter :=
 block nDefault
  block nCase_n
   ...
    block nCase_1
     block nTest
      [e_scrutinee] nTest
     end //nTest:
     tee_local #s   (where s is some fresh local of type i32)
     i32.const c_1; i32.eq; br_if nCase_1
     get_local #s
     i32.const c_2; i32.eq; br_if nCase_2
     ...
     br nDefault
    end //nCase_1:
    [e_1] nCase_2
   ...
  end //nCase_n:
  [e_n] nDefault
 end //nDefault:
 [e_default] nAfter
```

# Compiling switch statements

```
[s_switch] nAfter :=
 block nDefault
  block nCase_n
   ...
    block nCase_1
     block nTest
      [e_scrutinee] nTest
     end //nTest:
     tee_local #s   (where s is some fresh local of type i32)
     i32.const c_1; i32.eq; br_if nCase_1
     get_local #s
     i32.const c_2; i32.eq; br_if nCase_2
     ...
     br nDefault
    end //nCase_1:
    [e_1] nCase_2
   ...
  end //nCase_n:
  [e_n] nDefault
 end //nDefault:
 [e_default] nAfter
```

▷ How do we translate `break`?

# Compiling switch statements

At any point during the translation of **`switch`** we want to keep track not only where to jump *after*, but also where to jump on a `break`!

# Compiling switch statements

At any point during the translation of **`switch`** we want to keep track not only where to jump *after*, but also where to jump on a `break`!

⇒ Let us extend the translation function by another label parameter.

# Compiling switch statements

At any point during the translation of **`switch`** we want to keep track not only where to jump *after*, but also where to jump on a `break`!

⇒ Let us extend the translation function by another label parameter.

$$[\,\cdot\,]\ \texttt{nAfter} \ \Rightarrow\ [\,\cdot\,]\ \texttt{nAfter}\ \texttt{nBreak}$$

⇒ The caller of the translation function determines where to continue in the "normal" case, but also when `break` is called!

# Compiling switch statements

Translating `break` then is straightforward: One simply ignores `nAfter` and follows `nBreak` instead.

```
[break] nAfter nBreak :=
  br nBreak
```

▷ What do we have change in our translation of `switch` statements?

# Compiling switch statements with breaks

```
[s_switch] nAfter nBreak :=
  block nDefault
    block nCase_n
      ...
        block nCase_1
          block nTest
            [e_scrutinee] nTest nBreak
          end //nTest:
          tee_local #s   (where s is some fresh local of type i32)
          i32.const c_1; i32.eq; br_if nCase_1
          get_local #s
          i32.const c_2; i32.eq; br_if nCase_2
          ...
          br nDefault
        end //nCase_1:
        [e_1] nCase_2 nAfter
      ...
    end //nCase_n:
    [e_n] nDefault nAfter
  end //nDefault:
  [e_default] nAfter nAfter
```

# Translating While Statement

Consider translation of the **while** statement, which gets 'nextLabel' destination, specifying where to jump when exiting the loop.
We assume that the instructions emitted are inside the block that introduced nextLabel.

What is the translation schema?

[ **`while`** (cond) stmt ] `nextLabel =`

# Translating While Statement

Consider translation of the **while** statement, which gets 'nextLabel' destination, specifying where to jump when exiting the loop.
We assume that the instructions emitted are inside the block that introduced nextLabel.

What is the translation schema?

```
[ while (cond) stmt ] nextLabel =
  loop startLabel
    block bodyLabel
      branch(cond, bodyLabel, nextLabel)
    end // bodyLabel
    [ stmt ] startLabel
  end
```

# break Statement

In many languages, a break statement can be used to exit from the loop. For example, it is possible to write code such as this:

```
while (cond1) {
  code1
  if (cond2) break;
  code2
}
```

Loop executes `code1` and checks the condition `cond2`. If condition holds, it exists. Otherwise, it continues and executes `code2` and then goes to the beginning of the loop, repeating the process.

Give translation scheme for this loop construct and explain how the translation of other constructs needs to change.

# break Statement - Propagating Exit Label

For a **break** statement to know where to jump, it needs to be given a label indicating the exit of the loop. When we translate a statement (such as **if**) potentially containing **break**, the translation of this statement needs both the parameter to pass on to **break** as well as the parameter to jump to during normal execution. Therefore, each statement needs two destination parameters: the 'nextLabel' and the 'loopExit' label. For example,

[ **if** (cond) thenC **else** elseC ] `nextL loopExitL =`

# break Statement - Propagating Exit Label

For a **break** statement to know where to jump, it needs to be given a label indicating the exit of the loop. When we translate a statement (such as **if**) potentially containing **break**, the translation of this statement needs both the parameter to pass on to **break** as well as the parameter to jump to during normal execution. Therefore, each statement needs two destination parameters: the 'nextLabel' and the 'loopExit' label. For example,

```
[ if (cond) thenC else elseC ] nextL loopExitL =
  block elseL
    block thenL
      branch(cond, thenL, elseL)
    end // thenL
    [thenC] nextL loopExitL
  end // elseL
  [elseC] nextL loopExitL
```

# break Statement - Using and Setting Labels

Translating **break**:

```
[ break ] nextLabel loopExitLabel =
```

# break Statement - Using and Setting Labels

Translating **break**:

```
[ break ] nextLabel loopExitLabel =
  br loopExitLabel
```

# break Statement - Using and Setting Labels

Translating **break**:

```
[ break ] nextLabel loopExitLabel =
  br loopExitLabel
```

Translating while:

```
[ while (cond) stmt ] nextLabel loopExitLabel =
```

# break Statement - Using and Setting Labels

Translating **break**:

```
[ break ] nextLabel loopExitLabel =
  br loopExitLabel
```

Translating while:

```
[ while (cond) stmt ] nextLabel loopExitLabel =
  loop startLabel
    block bodyLabel
      branch(cond, bodyLabel, nextLabel)
    end // bodyLabel
    [ stmt ]
```

# break Statement - Using and Setting Labels

Translating **break**:

```
[ break ] nextLabel loopExitLabel =
  br loopExitLabel
```

Translating while:

```
[ while (cond) stmt ] nextLabel loopExitLabel =
  loop startLabel
    block bodyLabel
      branch(cond, bodyLabel, nextLabel)
    end // bodyLabel
    [ stmt ] startLabel
```

# break Statement - Using and Setting Labels

Translating **break**:

```
[ break ] nextLabel loopExitLabel =
  br loopExitLabel
```

Translating while:

```
[ while (cond) stmt ] nextLabel loopExitLabel =
  loop startLabel
    block bodyLabel
      branch(cond, bodyLabel, nextLabel)
    end // bodyLabel
    [ stmt ] startLabel nextLabel
  end
```

# break Statement - Using and Setting Labels

Translating **break**:

```
[ break ] nextLabel loopExitLabel =
  br loopExitLabel
```

Translating while:

```
[ while (cond) stmt ] nextLabel loopExitLabel =
  loop startLabel
    block bodyLabel
      branch(cond, bodyLabel, nextLabel)
    end // bodyLabel
    [ stmt ] startLabel nextLabel
  end
```

What if we want to have **continue** that goes to beginning of the loop?

# Loops with break and continue

Translating **break**:

```
[ break ] nextL loopExitL loopStartL =
  br loopExitL
```

Translating **continue**:

```
[ continue ] nextL loopExitL loopStartL =
  br loopStartL
```

Translating while:

```
[ while (cond) stmt ] nextL loopExitL loopStartL =
  loop startLabel
    block bodyLabel
      branch(cond, bodyLabel, nextL)
    end // bodyLabel
    [ stmt ] startLabel nextL startLabel
  end
```

Explain difference between labels `loopStartL` and `startLabel`