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

```
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]
```