Code Generation for Control Structures

Sequential Composition

How to compile statement sequence?

s1; s2; … ; sN

- Concatenate byte codes for each statement!

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

i.e. sts **flatMap** compileStmt

that is: (sts **map** compileStmt) **flatten**

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