Code Generation

A reference to WebAssembly instructions can be found at:
https://webassembly.github.io/spec/core/syntax/instructions.html
A list of useful instructions is found at the end of this document.

Exercise 1

Part 1

Show the WebAssembly instructions corresponding to the following 32-bit integer expressions:

```
(x + 1) + y

x * (y + 2)
```

You may assume that the local variable x is stored in the locals at index `xidx` and the variable y at index `yidx`.

Solution:

```
[[ (x + 1) + y ]] =
      local.get xidx
      i32.const 1
      i32.add
      local.get yidx
      i32.add

[[ x * (y + 2) ]] =
      local.get xidx
      local.get yidx
      i32.const 2
      i32.add
      i32.mul
```

Part 2

Compute the stack usage of the WebAssembly code in both cases.

Solution: The stack grows up to size 2 in the first case, 3 in the second.

Part 3

Come up with a recursive function that computes the stack space given an AST for the expression. In the scope of this exercise, you may assume the AST only has constructors for binary addition and multiplication, as well as constants and variables.

Solution:

```
stackSize(c) = 1  // Constants
stackSize(v) = 1  // Variables
stackSize(e1 + e2) = max(stackSize(e1), 1 + stackSize(e2))  // Add.
stackSize(e1 * e2) = max(stackSize(e1), 1 + stackSize(e2))  // Mul.
```

Part 4

Show how to emit code for binary addition with small stack usage by using commutativity.

Solution:

(`minStackSize` is used to compute the minimal stack size assuming commutativity)

```
minStackSize(c) = 1  // Constants
minStackSize(v) = 1  // Variables
minStackSize(e1 + e2) = min(
      max(minStackSize (e1), 1 + minStackSize (e2))
      max(minStackSize (e2), 1 + minStackSize (e1)))  // Add.
minStackSize(e1 * e2) = min(
      max(minStackSize (e1), 1 + minStackSize (e2))
      max(minStackSize (e2), 1 + minStackSize (e1)))  // Mul.

      [[ e1 + e2 ]] =
            if (minStackSize(e1) >= minStackSize(e2)) then
                  [[ e1 ]]
                  [[ e2 ]]
                  i32.add
            else
                  [[ e2 ]]
                  [[ e1 ]]
                  i32.add
```

Exercise 2

Consider the following new control flow structure:

```
fix (function) { start }
```

The metavariable *`start`* represents an expression of type 32-bit integer, and *`function`* is a static function from 32-bit integer to 32-bit integer. The meaning of the construct can be summarized by the following pseudo-code:

```
var current = start
var next = function(current)
while (current != next) {
  current = next
  next = function(current)
}
current
```

Emit code for the new construct. You may assume that you know the index `funidx` of the static function to be called. You also have access to a single entry in the locals to store a 32-bit integer value, at index `varidx`.

Solution:

```
[[ fix (function) { start } ]]  =
      [[ start ]]
      local.set varidx
      loop i32
      local.get varidx
      local.get varidx
      call funidx
      local.tee varidx
      i32.neq
      br_if 0
      local.get varidx
      end
```

Exercise 3

Part 1

Emit WebAssembly code for the following expression, using the `block`, `br_if` and `br` instructions.

Solution:

```
[[ if (condition) { thenBranch } else { elseBranch } ]] =
      block
      block
      [[ condition ]]
      br_if 0
      [[ elseBranch ]]
      br 1
      end
      [[ thenBranch ]]
      end
```

Part 2

Emit code for the `match` construct, assuming *`scrutinee`* is always an expression of type 32-bit integer, and assuming patterns can only be constant patterns, except for the last, which is always a wildcard pattern.

Solution:

```
[[ scrutinee match {
     case constant_1 => expression_1
     ...
     case constant_n => expression_n
     case _ => default } ]] =

      [[ scrutinee ]]
      local.set sidx
      block

      block
      local.get sidx
      i32.const constant_1
      i32.neq
      br_if 0
      [[ expression_1 ]]
      br 1
      end

      ...

      block
      local.get sidx
      i32.const constant_n
      i32.neq
      br_if 0
      [[ expression_n ]]
      br 1
      end

      [[ default ]]
      end
```

```
// Push the 32bit integer constant n on the stack.
i32.const n

// Pop the top two entries of the stack and push their equality.
i32.eq

// Pop the top two entries of the stack and push their inequality.
i32.neq

// Pop the top of the stack and set the local variable x to it.
local.set x

// Set the local variable x to the top value of the stack.
local.tee x

// Get the local variable x and push it on the stack.
local.get x

// Declare a block of instructions. Branching out leads to the end.
block type instruction* end

// Declare a block of instructions. Branching out leads to the start.
loop type instruction* end

// Branches out of a block.
br label

// Pop the top of the stack and branches out of a block if true.
br_if label

// Call a function.
// Arguments are popped from the stack, and the result is pushed.
call funindex
```