Expression Types

(const C) A constant value. It acts somewhat like quote. To get the number one, you would use (const 1). To get the list (1 2) you would use (const (1 2)).

(reg R) Retrieve the value of a register R. To get the value of the register arg0, you would use (reg arg0).

(label L) Retrieve the offset of the given label L. To get the value of the label loop-top, you would use (label loop-top).

(op O) Perform operation O on some values. Following the (op O), you should list the input arguments to the operation, which may be consts, regs, or labels. An expression may only contain 1 op. In order to compute the result of adding 1 to the register arg0, you would use (op +) (reg arg0) (const 1).

Instruction Types

(assign reg expr) Sets register reg to be the result of expression expr. The assigned register doesn’t need a tag because it is always a register being assigned. For example, to increment the result register:
(assign result (op +) (reg result) (const 1))

(perform expr) Sometimes an expr is done only for it’s side-effect and its value is unwanted. perform works just like assign, but without assigning a register. For example, to output register tmp:
(perform (op write-line) (reg tmp))

(test expr) This is equivalent to assigning the cr. The cr register is used to determine whether to take a branch. For example, to set the cr based on whether the register x is less than 10:
(test (op <) (reg x) (const 10))

(goto expr) Sets the pc to be the result of expr, which is usually a label or a register. Effectively continues the execution at another point in the code. To jump to the label loop-top:
(goto (label loop-top))

(branch expr) If the value in the cr is true, acts like a goto. Otherwise it does nothing. To conditionally jump to the label loop-done:
(branch (label loop-done))
Writing Code

Write double: code to compute $2x$, given $x$ in arg0, and leave the output in result.

double
  (assign result (op *) (reg arg0) (const 2))
  (goto (reg continue))

1. Write func: code to compute $x^2 + y$, given $x$ in arg0, $y$ in arg1, and leave the output in result.

2. Write abs: code to compute $|x|$, give $x$ in arg0, leave the output in result. abs is not an available primitive.

3. Write infinite-loop: code that never halts.

4. Determine what the following code does, then write the scheme code that does the same thing.

  foo
  (test (op <) (reg arg0) (reg arg1))
  (branch (label foo-done))
  (assign arg0 (op -) (reg arg0) (reg arg1))
  (goto (label foo))
  foo-done
  (assign result (op =) (reg arg0) (const 0))
  (goto (reg continue))
5. Write `sum-digits`: code that computes the sum of the digits in `arg0`, leaving the output in `result`. You may assume `quotient` and `remainder` are available as primitive operations.

6. Write `reduce-to-digit`: code that sums the digits of a number repeatedly until a result with only a single digit remains. For example, 185 ⇒ 14 ⇒ 5. As usual, input in `arg0`, but output in `result` a cons cell whose car is the original number and cdr is the digit it reduces to (result of previous example is (185 . 5)).