1 The Virtual Machine

This problem set involves translating Scheme to instructions for a *virtual machine*. A virtual machine is a program that resembles an idealized CPU. It reads and executes a sequence of instructions, which are idealized versions of CPU instructions. Whereas a real CPU reads chunks of binary data into fixed-sized registers, our virtual machine's instructions operate on Scheme datatypes.

This strategy, of a simple compiler targetting a language-specific virtual machine, is by far the most common way to implement programming languages today. The only popular languages which *don't* follow this strategy are C and its descendants, as well as Go.

Our virtual machine has the following instructions. You can guess what most of them do from their names:

- `add`, `sub`, `mul`, `div`
- `and`, `or`, `not`
- equal
- cons car cdr
- call, tail-call, ret – used to implement procedure calls and returns
- cjump – used to implement `if`, as well as the boolean operators for reasons you will soon see.
- push-constant, popn
- env-lookup, env-mutate
- make-procedure
- halt

1.1 Virtual Machine State and Execution

The virtual machine is a a loop, very similar to the one you wrote in the lexer assignment. At every iteration, it reads an instruction and executes it. Executing an instruction changes the state of the machine. Then, the machine reads and executes the next instruction, and so on.

The machine state consists of five variables:

`code-memory`: a vector of instructions for the machine to execute. This won't change during the execution of the machine.

`instruction-pointer`: the index of the instruction to be executed in the current cycle. Usually this is incremented after each instruction, so that the immediately following instruction will be executed next. The exceptions to this are `call`, `ret`, `tail-call`, and `cjump`.

`data-stack`: This is a stack of scheme values. All instructions read their operands by popping them off the stack, and push the result to the top of the stack.

`environment`: This is the environment that the "environment model" refers to. The environment is a list of frames, and each frame is a vector of values. We don't store the variable names; the environment uses lexical addressing.

`call-stack`: When we call a procedure, we need to save the current temporary values (the `data-stack`), the current `environment`, and the current `instruction-pointer`. When the callee returns, these all need to be restored. The structure containing all this information is a `stack-frame` and the `call-stack` is a list of `stack-frames`.

1.2 Instructions

`add sub mul div`: Each of these instructions behaves the same way. The top two elements of the stack are poppe and used as operands of the corresponding arithmetic instruction. Then the result is pushed. Execution resumes at the next instruction.

`and or equal` These work the same as the instructions in the previous paragraph, but produce a boolean result.

`not` Like the preceding instructions, but only takes one operand, therefore only pops one element from the stack.

`cons car cdr` It should be obvious, at this point, what these instructions do and how many operands each takes.

`call`: This is probably the most complex instruction. Each call instruction looks like `(call n)`, where `n` is a constant embedded in the instruction itself, not an operand from the data stack.

A `(call n)` instruction pops $n+1$ operands. The 1th through nth operands are the arguments of the procedure call, and the 0th operand, i.e. the top of the stack, is the procedure itself.

A procedure is represented by a `machine-procedure` struct. In the environment model, a procedure is represented by some code and an environment to evaluate the code in, and this is what a `machine-procedure` struct contains.

So a `(call n)` first pops a `machine-procedure` off the stack, and examines the `environment` stored in the procedure. It then pops n operands off the data stack and adds them to the procedure's `environment`. It then stashes the current virtual machine environment, data stack, and instruction pointer in a `call-stack-frame` and pushes it to the call stack. Then it sets the instruction pointer to the `address` stored in the procedure, and the `environment` to the new environment it fashioned from the arguments and procedure environment.

`ret`: Pops the top stack frame off the `call-stack`. Restores the `environment` and `data-stack` registers to those saved in the stack frame, but sets the `instruction-pointer` to *one past* the value saved in the stack frame. Since the saved instruction pointer is the address of the `call` instruction, this means that procedures return to the next instruciton after the `call`. So the caller can treat a procedure call as if it were a single instruction computing the desired result.

`tail-call`: Functions like a `call`, but does not push a new stack frame. When we do a `tail-call`, the callee returns directly to *our* caller.

(env-lookup m n) reads the nth entry of the mth frame of the dynamic environment, and pushes it onto the `data-stack`. It is used for translating variable references.

(env-mutate m n) pops an operand from the `data-stack`, and writes it to the nth entry of the mth frame of the dynamic environment. Used for implementing `set!`

`(push-constant c)` takes the constant `c`, embedded in the instruction, and pushes it to the `data-stack`.

`(popn n)` pops n values from the `data-stack`. n is usually 1, and this instruction is used for discarding values of expressions (for example, every expression in a procedure body except the last).

`(cjump addr)` pops one operand off the `data-stack`. If it is anything but `#f`, the instruction pointer is set to `addr` and execution continues from there. Otherwise, the instruction does nothing and execution continues from the next instruction.

`(make-procedure arity addr)` creates a `machine-procedure` with the address and arity given in the instruction, and with the current `environment`, and pushes that procedure to the stack. Used for compiling `lambda`s. Note it reads no operands from the `data-stack`.

`halt` stops execution of the virtual machine.

2

Translate the following programs to stack machine code:

```
(+ 1 2)

(* 1 2 3 4 5)

(+ 1 2 (* 3 4))

(lambda (x y) (+ x y))

(lambda (n) (if (even? n) (/ n 2) (+ n 3)))

(lambda (x) (lambda (y) (+ x y)))
```

3

Implement `translate-variable` and `translate-constant`, the parts of a compiler that can translate constants and variable references.

For obvious reasons, you can only test this with variables that are defined in the global environment.

4

Implement `translate-call`, to support procedure calls. Now you should be able to evaluate nested expressions using the built-in procedures.

5

Now implement `translate-if`, `translate-and`, and `translate-or`. Note that `and` and `or` are similar to `if`, in that they have special evaluation rules which cannot be implemented as procedures.

If you have the expression `(and e1 e2)`, then `e1` must be evaluated first. If its result is false, then you already know that the result of the whole `and` expression will be false, so there's no point in evaluating `e2`. The rule is: `and` evalutes its second argument only if its first is true, and `or` evaluates its second argument only if its first is false. This behavior is known as *short-circuiting*.

6

Now implement `translate-lambda`. This is the hard part. Note that in scheme, evaluating a lambda expression produces a value, the procedure. So certainly, when you compile a lambda expression, you need to emit a `make-procedure` instruction to put the procedure value on top of the stack.

But you also need to compile the body of the procedure! And that will emit code! Where do you put this code? If you just spit out the body of the procedure after the `make-procedure` instruction, evaluating a lambda will cause the body of the procedure to immediately start executin (executing in the wrong environment, moreover, and corrupting the call stack when it returns).

Here is a quick and dirty way to do it: emit the `make-procedure` instruction, then emit an unconditional jump (made up of `push-constant` and `cjump`), and then emit the body of the procedure. Arrange for your unconditional jump to jump over the body.

7

Now implement `translate-let`, by desugaring a `let` expression into a `lambda`, and compiling the result of the desugaring.

```
(let ((a 1)
      (b 2)
      (c 3))
  (+ (* a b) c))
```

desugars to

```
((lambda (a b c) (+ (* a b) c)) 1 2 3)
```

8

Now implement `translate-letrec`, which is required to define recursive functions. Scheme's `define` forms are defined (no pun intended) in terms of `letrec`.

For example:

```
(define (fac (n) (if (zero? n) 1 (fac (- n 1)))))
```

is equivalent to

```
(letrec ((fac (lambda (n) (if (zero? n) 1 (fac (- n 1)))))))
```

except, of course, the definition of `fac` will only be visible inside the lectrec.

You can implement letrec by desugaring to `let`. This:

```
(letrec ((a a-expr)
         (b b-expr)
         (c c-expr))
 body)
```

desugars to this:

```
(let ((a '()) (b '()) (c '()))
 (set! a a-expr)
 (set! b b-expr)
 (set! c c-expr)

 body)
```

Here we are using `'()` as a placeholder value, which should never be observed if `a-expr`, `b-expr`, `c-expr` are lambda expressions.

9