17 Constructing a Hybrid Interpreter Level 2

Some More Bytecode Instructions

```
UNARY_NEGATIVE     = 11       # hex 0B
BINARY_MULTIPLY    = 20       # hex 14
BINARY_DIVIDE      = 21       # hex 15
BINARY_ADD         = 23       # hex 17
BINARY_SUBTRACT    = 24       # hex 18
PRINT_ITEM         = 71       # hex 47
PRINT_NEWLINE      = 72       # hex 48
STORE_NAME         = 90       # hex 5A
LOAD_CONST         = 100      # hex 64
LOAD_NAME          = 101      # hex 65
COMPARE_OP         = 106      # hex 6A
JUMP_FORWARD       = 110      # hex 6E
POP_JUMP_IF_FALSE  = 111      # hex 6F
JUMP_ABSOLUTE      = 113      # hex 71
```

Figure 17.1

```
LT = 0    # <  code
LE = 1    # <= code
EQ = 2    # == code
NE = 3    # != code
GT = 4    # >  code
GE = 5    # >= code
```

Figure 17.2

```
 1 def relexpr():
 2     expr()
 3     savecat = token.category
 4     if savecat in [LESSTHAN, LESSEQUAL, EQUAL, NOTEQUAL,
 5                    GREATERTHAN, GREATEREQUAL]:
 6         advance()
 7         expr()
 8         co_code.append(COMPARE_OP)
 9         if savecat == LESSTHAN:
10             co_code.append(LT)
11         elif savecat == LESSEQUAL:
12             co_code.append(LE)
13         ... <============================== missing instructions
```

index	co_code	mnemonic
50	110	JUMP_FORWARD
51	5	relative address
52	100	LOAD_CONSTANT
53	2	constant
54	100	LOAD_CONSTANT
55	3	constant
56	23	BINARY_ADD
57	71	PRINT_ITEM

Value in `pc` (52) + Relative address in `JUMP_FORWARD` instruction (5) = 57

Compiling Backward Branches to Bytecode

```
backaddress = len(co_code) # save address of relexpr bytecode
relexpr()                  # parse exit-test expression

co_code.append(JUMP_ABSOLUTE)
co_code.append(backaddress)
```

to generate the `JUMP_ABSOLUTE` instruction that, when executed, branches back to the bytecode that evaluates the exit-test relational expression.

Compiling Forward Branches to Bytecode

```
    address1 = len(co_code)

    co_code.append(None)

    co_code[address1] = len(co_code)
```

```
 1 def ifstmt():
 2    advance()
 3    relexpr()
 4    consume(COLON)
 5    co_code.append(POP_JUMP_IF_FALSE)
 6    address1 = len(co_code)
 7    co_code.append(None)
 8    codeblock()                        # if code block
 9    if token.category == ELSE:
10       advance()
11       consume(COLON)
12       co_code.append(JUMP_FORWARD)    # jump over else part
13       address2 = len(co_code)
14       co_code.append(None)
15       startaddress = len(co_code)
16       co_code[address1] = len(co_code)
17       codeblock()                     # else code block
18       co_code[address2] = len(co_code)-startaddress
19    else:
20       co_code[address1] = len(co_code)
```

```
def interpreter():
   co_values = [None] * len(co_names)
   stack = []
   pc = 0
   while pc < len(co_code):
      opcode = co_code[pc]
      pc += 1
      if opcode == UNARY_NEGATIVE:
         stack[-1] = -stack[-1]
      elif if opcode == BINARY_MULTIPLY:
         right = stack.pop()
         left = stack.pop()
         stack.append(left * right)
      ... <============================ missing instructions
      elif opcode == COMPARE_OP:
         op = co_code[pc]          # get type of comparison
         pc += 1                   # increment pc to next instruction
         right = stack.pop()       # pop two values off the stack
         left = stack.pop()
         if op == LT:              # check if LT comparison
            stack.append(left < right) # push True or False
         elif op == LE:
            stack.append(left <= right)
         ... <========================== missing instructions
      elif opcode == JUMP_FORWARD:
         reladdr = co_code[pc]   # get rel addr from JUMP_FORWARD inst
         pc += 1                 # increment pc to next instruction
         pc = pc + reladdr       # load pc with branch-to address
      elif opcode == POP_JUMP_IF_FALSE:
         if not stack.pop():     # is top of stack False?
            pc = co_code[pc]     # absolute branch
         else:
            pc += 1              # go to next instruction
      elif opcode == JUMP_ABSOLUTE:
         pc = co_code[pc]
      else:
         break
```