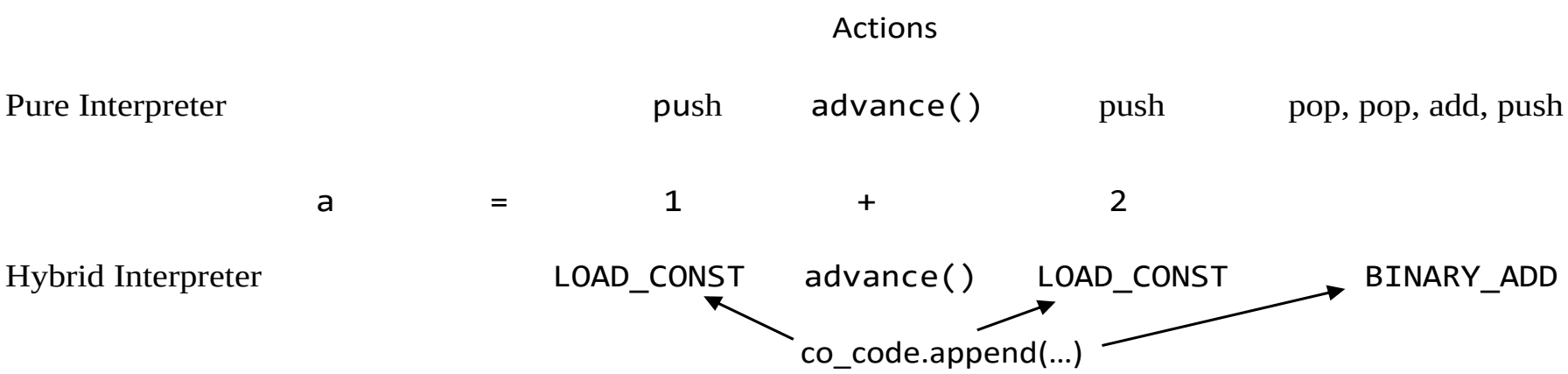
10 Constructing a Hybrid Interpreter

Pure Interpreter Versus Hybrid Interpreter

Pure interpreter executes as it parses
Hybrid interpreter generates bytecode that when ultimately executed does what the pure interpreter does



Actions in pure interpreter:

push: `operandstack.append(sign*int(token.lexeme))`

pop: `rightoperand = operandstack.pop()`
pop: `leftoperand = operandstack.pop()`
add, push: `operandstack.append(leftoperand + rightoperand)`

Global Constants and Variables in Our Hybrid Interpreter

```
# bytecode opcodes
UNARY_NEGATIVE    = 11      # hex 0B
BINARY_MULTIPLY   = 20      # hex 14
BINARY_ADD        = 23      # hex 17
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
```

Figure 10.1

Data structures

```
co_code = []
co_names = []
co_consts = []
```

Embedding the Code Generator in the Parser

```
1 def expr():
2     term()     # generates bytecode that pushes term's value
3     while token.category == PLUS:
4         advance()
5         term()  # generates bytecode that pushes term's value
6         co_code.append(BINARY_ADD) # generate the add instruction
```

```
1 def term():
2     global sign
3     sign = 1        # initialize sign
4     factor()
5     while token.category == TIMES:
6         advance()
7         sign = 1    # initialize sign
8         factor()
7         co_code.append(BINARY_MULTIPLY)
```

```
def factor():
   global sign
   if token.category == PLUS:
      advance()
      factor()
   elif token.category == MINUS
      sign = -sign      # change sign for each unary minus
      advance()
      factor()
   elif token.category == UNSIGNEDINT:
      v = sign*int(token.lexeme)
      if v in co_consts:
         index = co_consts.index(v)
      else:
         index = len(co_consts)
         co_consts.append(v)
      co_code.append(LOAD_CONST)
      co_code.append(index)
      advance()
   elif token.category == NAME:
      if token.lexeme in co_names:
         index = co_names.index(token.lexeme)
      else:
         raise RuntimeError('Name ' + token.lexeme + ' is not defined')
      co_code.append(LOAD_NAME)
      co_code.append(index)
      if sign == -1:
         co_code.append(UNARY_NEGATIVE)
      advance()
   elif token.category == LEFTPAREN:
      advance()
      # expr() calls term() which sets sign to 1 so must save sign
      savesign = sign
      expr()
      if savesign == -1:                  # use saved value of sign
         co_code.append(UNARY_NEGATIVE)  # negate expr
      consume(RIGHTPAREN)
   else:
      raise RuntimeError('Expecting factor')
```

Modifying the printstmt() Function

```
1 def printstmt():
2    advance()
3    consume(LEFTPAREN)
4    expr()
5    consume(RIGHTPAREN)
```

```
1 def printstmt():
2     advance()
3     consume(LEFTPAREN)
4     expr()
5     co_code.append(PRINT_ITEM)          # pop stack and display
6     co_code.append(PRINT_NEWLINE)       # output a newline char
7     consume(RIGHTPAREN)
```

Modifying the assignmentstmt() Function

```
 1 def assignmentstmt():
 2     if token.lexeme in co_names:
 3         index = co_names.index(token.lexeme)
 4     else:
 5         index = len(co_names)
 6         co_names.append(token.lexeme)
 7     advance()
 8     consume(ASSIGNOP)
 9     expr()
10     co_code.append(STORE_NAME)
11     co_code.append(index)
```

Implementing a Bytecode Interpreter

```
1 def main():
2     ...
3     try:
4         tokenizer()
5         parser()
6     except RuntimeError as emsg
7         ...
8     interpreter()
```

co_values, stack, pc

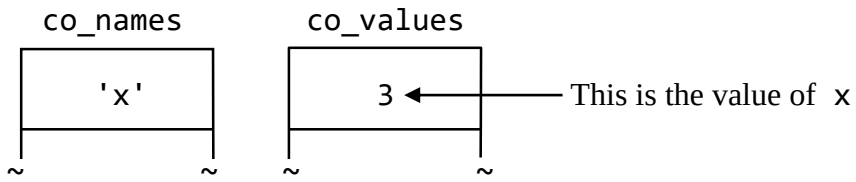


Figure 10.2

```
co_values = [None] * len(co_names)

stack = []

pc = 0
```

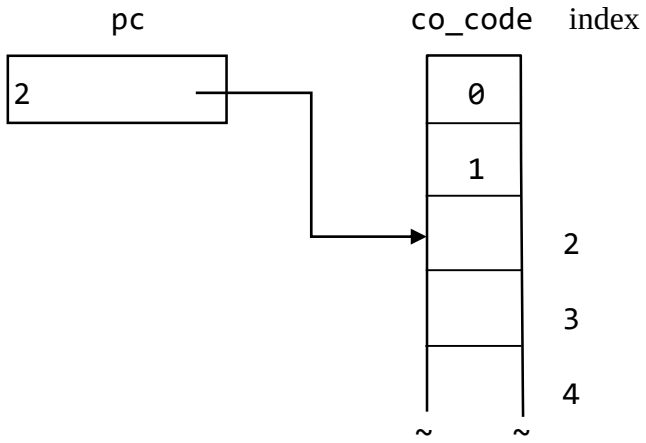


Figure 10.3

```
# bytecode interpreter
def interpreter():
   co_values = [None] * len(co_names)
   stack = []
   pc = 0

   while pc < len(co_code):
      opcode = co_code[pc]        # get opcode from co_code
      pc += 1                     # increment pc past the opcode

      if opcode == UNARY_NEGATIVE:
         stack[-1] = -stack[-1]
      elif opcode == BINARY_MULTIPLY:
          right = stack.pop()
          left = stack.pop()
          stack.append(left * right)
      elif opcode == BINARY_ADD:
         ...
      elif opcode == PRINT_ITEM:
         ...
      elif opcode == PRINT_NEWLINE:
         ...
      elif opcode == STORE_NAME:
         ...
      elif opcode == LOAD_CONST:
         index = co_code[pc]      # get index from inst
         pc += 1                  # increment pc to next inst
         value = co_consts[index] # get value from co_consts
         stack.append(value)      # push value onto stack
      elif opcode == LOAD_NAME:
         index = co_code[pc]      # get index of variable
         pc += 1                  # increment pc to next inst
         value = co_values[index] # get value of variable
         if value == None:
            print('No value for ' + co_names[index])
            sys.exit(1)
         stack.append(value)      # push value onto stack
      else:
         break
```

Figure 10.4

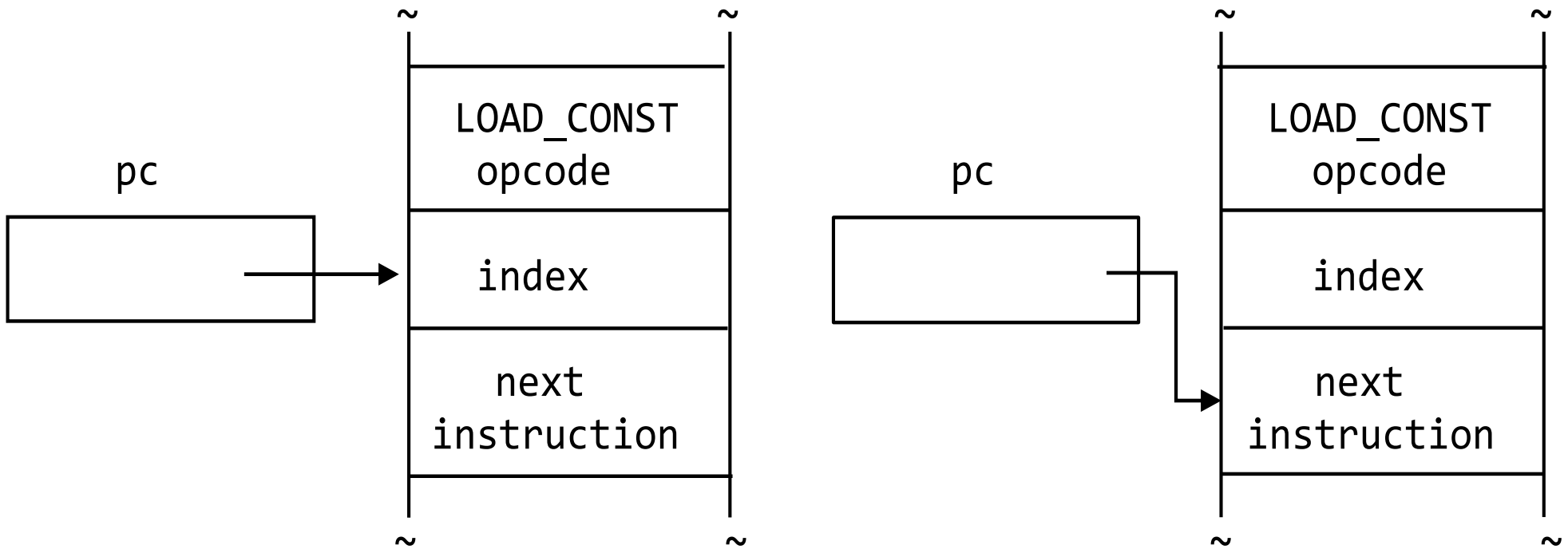

After incrementation of pc on line 9
pc
LOAD_CONST
opcode
index
next
instruction
After incrementation of pc on line 27
pc
LOAD_CONST
opcode
index
next
instruction

Figure 10.5