20 Constructing a Hybrid Interpreter Level 3

Introduction

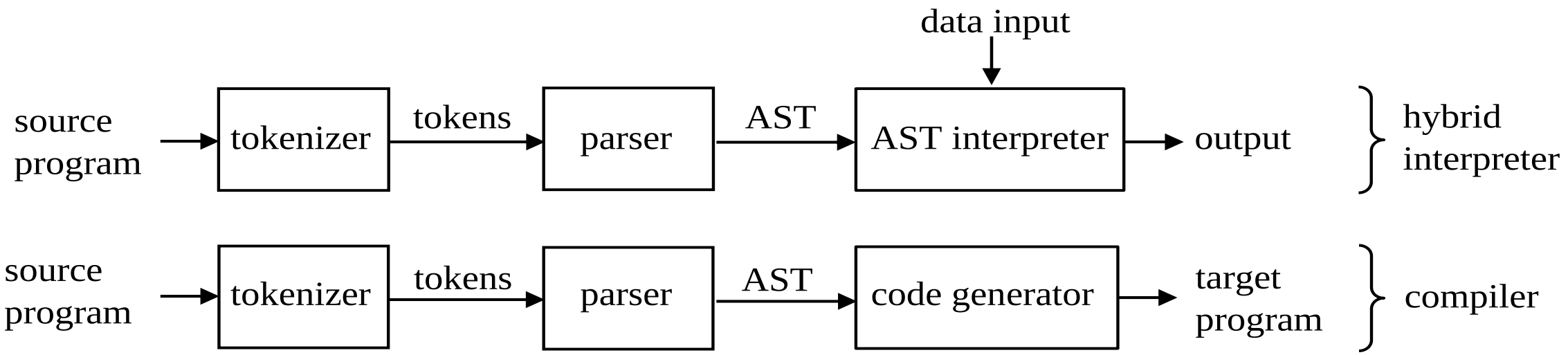
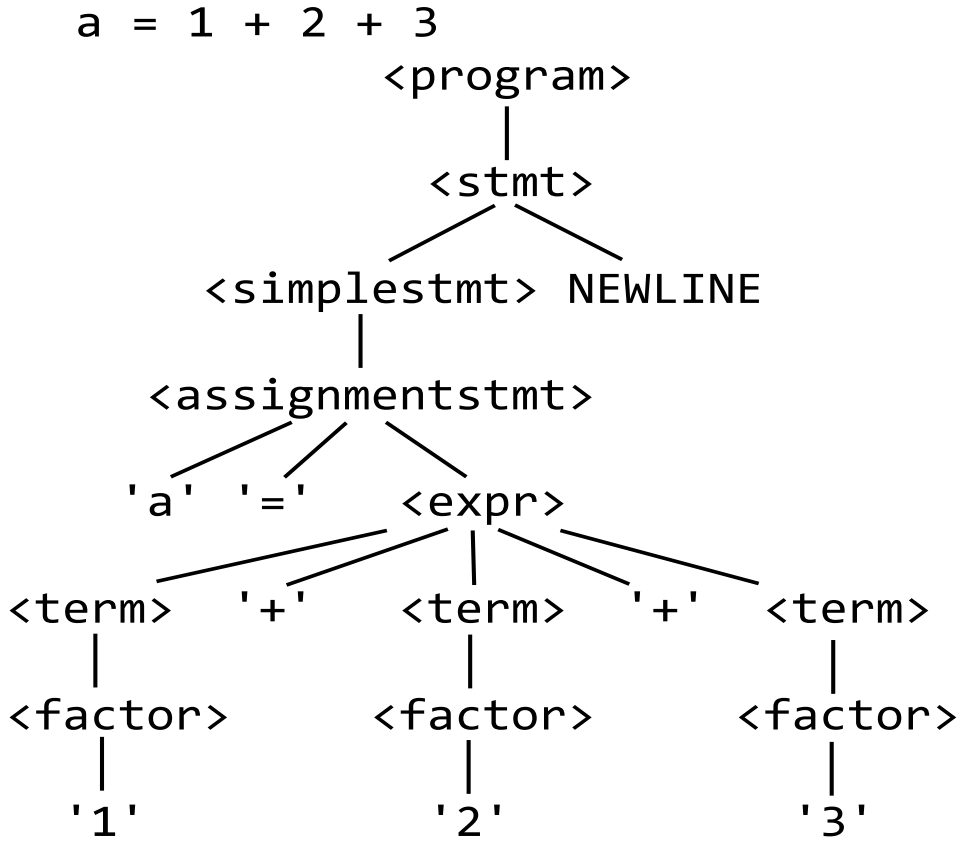
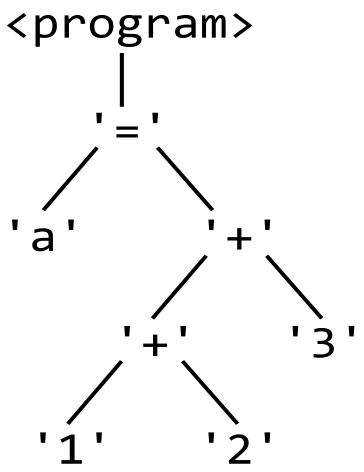


Figure 20.1

Parse Trees Versus ASTs



Parse tree



Abstract syntax tree

Figure 20.2

Representing an AST

```
1 class Node:
2    def __init__(self, type, left, right):
3       self.type = type      # type of the node
4       self.left = left      # pointer to left child
5       self.right = right    # pointer to right child (if any)
```

Figure 20.3

Constructing an AST

```
23    if token.category == UNSIGNEDINT:
24       node =  Node(INTEGER, sign*int(token.lexeme), None)
25       advance()
26       return node


27    elif token.category == NAME:
28       node = Node(NAME, token.lexeme, None)
29       advance()
30       if sign == -1:
31          node = Node(NEGATE, node, None)
32       return node
```

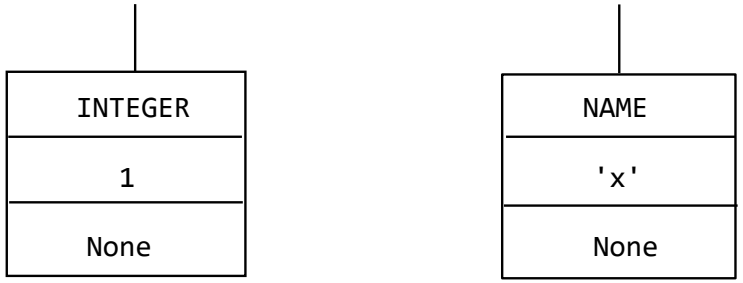


Figure 20.4

```
 1 def term():
 2    global sign
 3    sign = 1
 4    left = factor()      # get reference to left factor
 5    while token.category == TIMES:
 6       advance()
 7       sign = 1
 8       right = factor()  # get reference to right factor
 9       # create Node with TIMES, left, and right
10       node = Node(TIMES, left, right)
11       left = node       # node becomes left factor
12    return left
13
```

```
14 def factor():
15    global sign
16    if token.category == PLUS:
17       advance()
18       return factor()
19    elif token.category == MINUS:
20       sign = -sign
21       advance()
22       return factor()
23    if token.category == UNSIGNEDINT:
24       node =  Node(INTEGER, sign*int(token.lexeme), None)
25       advance()
26       return node
27    elif token.category == NAME:
28       node = Node(NAME, token.lexeme, None)
29       advance()
30       if sign == -1:
31          node = Node(NEGATE, node, None)
32       return node
33    elif token.category == LEFTPAREN:
34       advance()
35       savesign = sign     # must save sign because expr()
36       node = expr()       # calls term() which resets sign to 1
37       if savesign == -1:  # so use the saved value of sign
38          node = Node(NEGATE, node, None)
39       consume(RIGHTPAREN)
40       return node
41    else:
42       raise RuntimeError('Expecting factor')
```

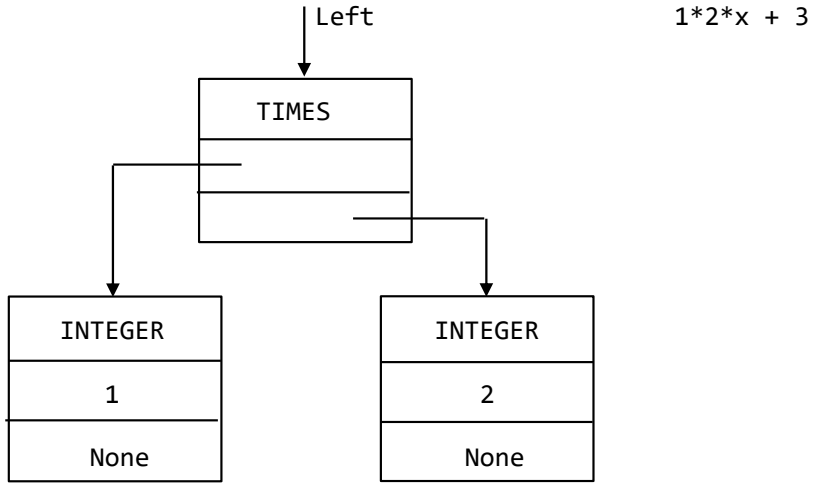
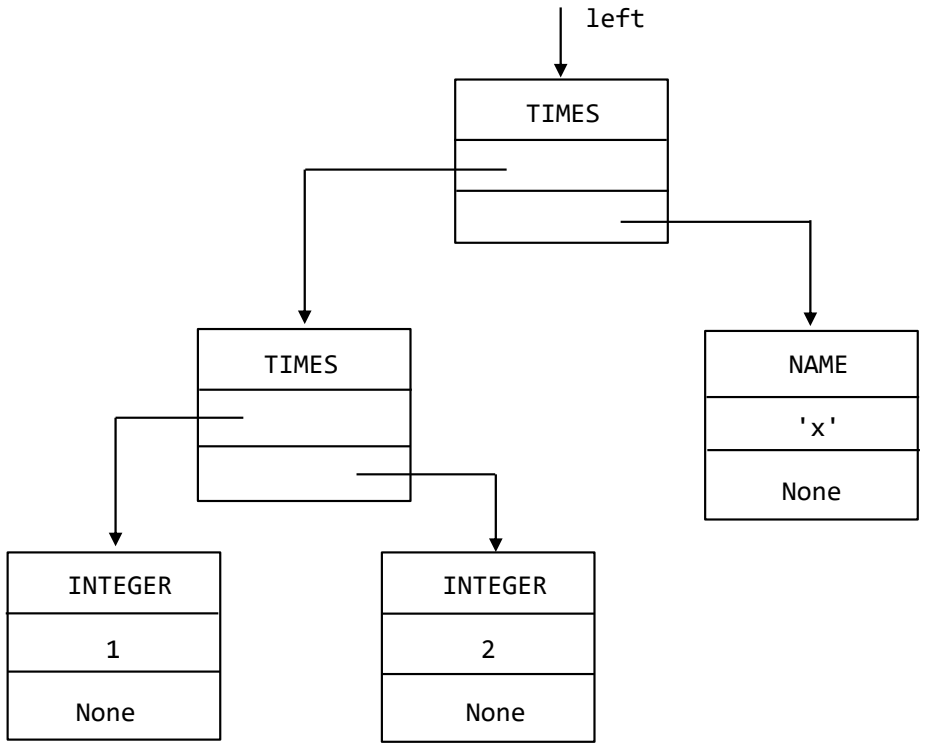

Left
1*2*x + 3
TIMES
INTEGER
1
None
INTEGER
2
None

Figure 20.6


left
TIMES
TIMES
NAME
'x'
None
INTEGER
1
None
INTEGER
2
None

```
def program():
   stmtlist = []
   while token.category in [NAME, PRINT]:
      stmtlist.append(stmt())
   if token.category != EOF:
      raise RuntimeError('Expecting EOF')
   return Node(PROGRAM, stmtlist, None)

def stmt():
   ast = simplestmt()
   consume(NEWLINE)
   return ast

def simplestmt():
   if token.category == NAME:
      return assignmentstmt()
   elif token.category == PRINT:
      return printstmt()
   else:
      raise RuntimeError('Expecting statement')

def assignmentstmt():
   lexeme = token.lexeme     # save token.lexeme
   advance()
   consume(ASSIGNOP)
   node = Node(ASSIGNOP, lexeme, expr())
   return node

def printstmt():
   advance()
   consume(LEFTPAREN)
   node = Node(PRINT, expr(), None)
   consume(RIGHTPAREN)
   return node
```

Figure 20.8

```
ast = parser()      # get AST corresponding to entire program
...
interpreter(ast)    # call interpreter passing it the AST
```

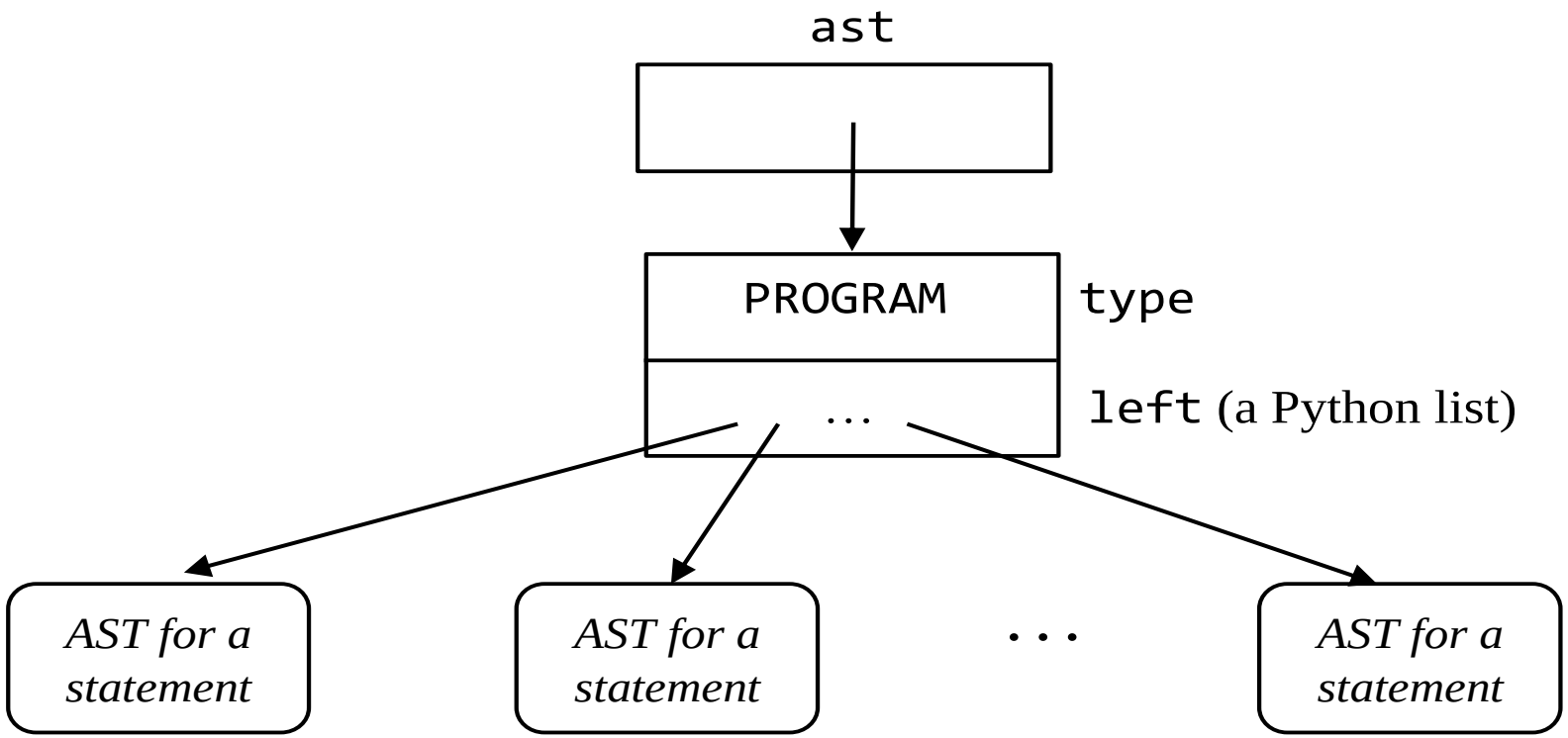


Figure 20.9

Interpreting an AST

```
1 def interpreter(ast):
2    for stmt in ast.left:
3       if stmt.type == ASSIGNOP:
4          symtab[stmt.left] = value(stmt.right)
5       elif stmt.type == PRINT:
6          print(value(stmt.left))
7       else:
8          raise RuntimeError('Expecting stmt')
```

Figure 20.10

1. Evaluate the left subtree.

2. Evaluate the right subtree

3. Using the values of the left and right subtrees, compute the value of the root node of the tree according to the operation specified by the root node of the tree.

```
 1 def value(node):
 2    if node.type == INTEGER:
 3       return node.left
 4    elif node.type == NAME:
 5       return symtab[node.left]
 6    elif node.type == PLUS:
 7       return value(node.left) + value(node.right)
 8    elif node.type == TIMES:
 9       return value(node.left) * value(node.right)
10    elif node.type == NEGATE:
11      return -value(node.left)
12    else:
13      raise RuntimeError('Invalid structure')
```

Figure 20.11

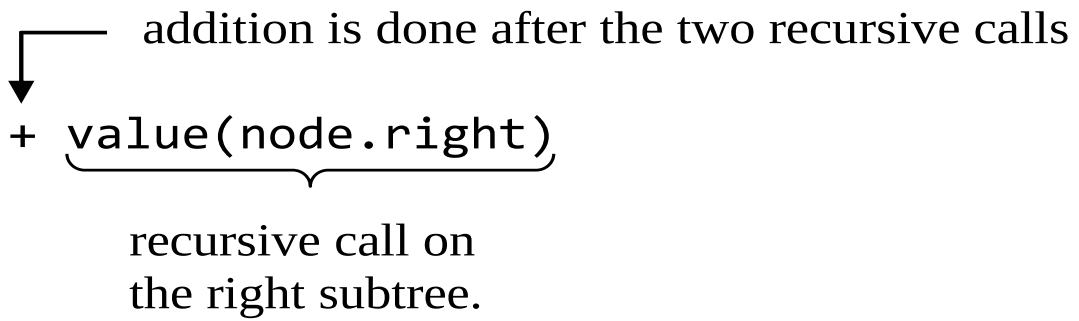


Figure 20.12