19 Constructing a Pure Interpreter Level 3

New Symbol Tables

```
localsymbol = {}

globalsymbol = {}

globalsdeclared = []


global f
```

```
<program>       → <stmt>* EOF

<stmt>          → <simplestmt> NEWLINE
<stmt>          → <compoundstmt>

<simplestmt>    → <assignmentstmt>
<simplestmt>    → <printstmt>
<simplestmt>    → <passstmt>
<simplestmt>    → <globalstatment>
<simplestmt>    → <returnstmt>
<simplestmt>    → <functioncall>

<compoundstmt>  → <whilestmt>
<compoundstmt>  → <ifstmt>
<compoundstmt>  → <defstmt>

<assignmentstmt>→ NAME '=' <relexpr>
<printstmt>     →'print' '(' [<relexpr> (',' <relexpr> )* [',']] ')'
<passstmt>      → 'pass'
<globalstmt>    → 'global' NAME (',' NAME)*
<returnstmt>    → 'return' [<relexpr>]
<whilestmt>     → 'while' <relexpr> ':' <codeblock>
<ifstmt>        → 'if' <relexpr> ':' <codeblock> ['else' ':' <codeblock>]
<defstmt>      → 'def' NAME '(' [NAME (',' NAME)*] ')' ':' <codeblock>

<codeblock>     →  NEWLINE INDENT <stmt>+ DEDENT
<relexpr>       → <expr> [ ('==' | '!=' | '<' | '<=' | '>' | '>=') <expr> ]
<expr>          → <term> (('+' | '-' ) <term>)*
<term>          → <factor> (('*' | '/') <factor>)*

<factor>        → '+' <factor>
<factor>        → '-' <factor>
<factor>        → UNSIGNEDINT
<factor>        → UNSIGNEDFLOAT
<factor>        → NAME
<factor>        → '(' <relexpr> ')'
<factor>        → STRING
<factor>        → 'True'
<factor>        → 'False'
<factor>        → 'None'
<factor>        → 'input' '(' STRING ')
<factor>        → 'int' '(' <relexpr> ')'
<factor>        → <functioncall>
<functioncall>  → NAME '(' [<relexpr> [',' <relexpr>]*] ')
```

Figure 19.1

```
def g():
   global y      # infunction = 2 here
   y = 1
def f():
   z = 2         # infunction = 1 here
   g()
   z = 3         # infunction = 1 here

x = 1            # infunction = 0 here
f()
x = 2            # infunction = 0 here
```

Figure 19.2

```
if v in globalsdeclared or infunction == 0:
    Enter v and its value into the global symbol table, or update its value if it is already there
else:
    Enter v and its value into the local symbol table, or update its value if it is already there
```

```
def getvalue(s):
   if s in localsymbol:
      return localsymbol[s]
   if s in globalsymbol:
      return globalsymbol[s]
   else:
      raise RuntimeError('No value for ' + s)
```

How a Function Definition is Handled

```
x = 2
def f(z)
    print(x + z)
f(10)
```

```
def defstmt():
   advance()                 # advance past DEF
   if token.lexeme + '()' not in globalsymbol:
      globalsymbol[token.lexeme + '()'] = tokenindex
   else:
      raise RuntimeError('Duplicate function definition')
   while token.category != INDENT:
      advance()              # adv up to INDENT at end of function header
   indentcol = token.column  # save column of INDENT token
   while True:
      if token.category == DEDENT and token.column < indentcol:
         advance()           # advance past DEDENT
         break
      advance()
```

Saving Return Addresses

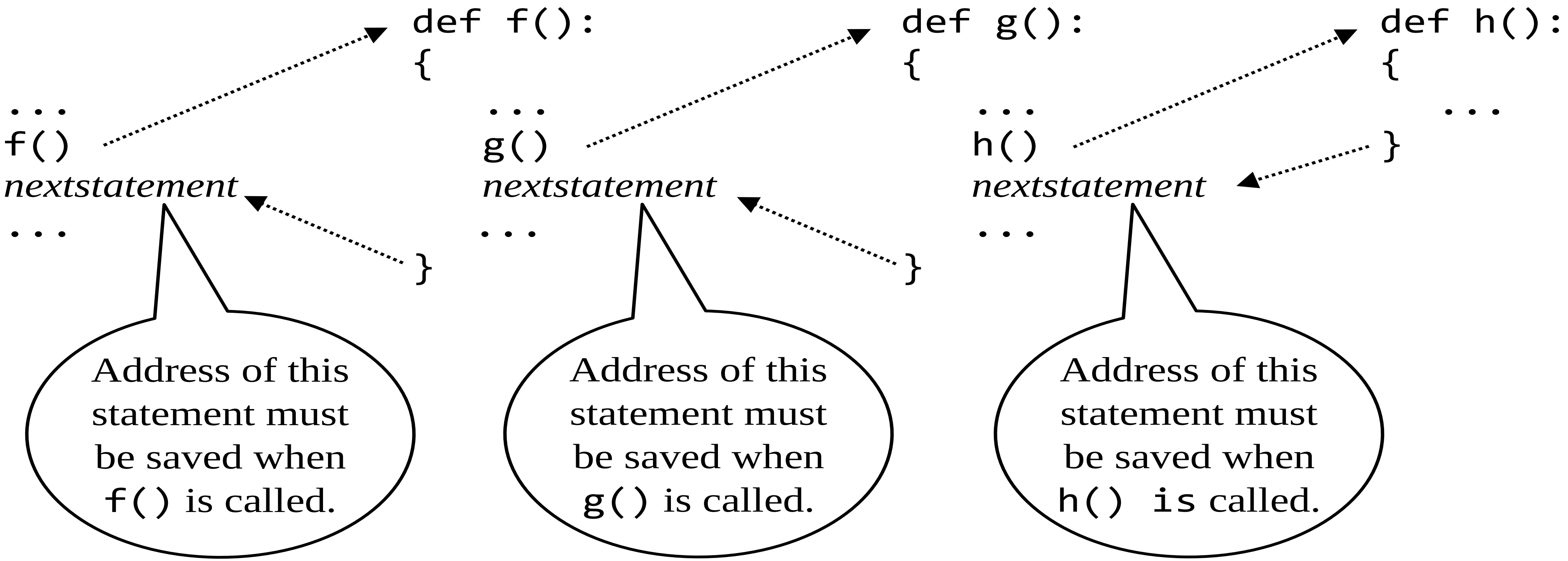

def f():
{
}
...
f()
nextstatement
...
Address of this statement must be saved when f() is called.
def g():
{
}
...
g()
nextstatement
...
Address of this statement must be saved when g() is called.
def h():
{
...
}
...
h()
nextstatement
...
Address of this statement must be saved when h() is called.

Figure 19.3

```
returnstack = []
```

Saving the Local Symbol Table

```
localsymtabstack = []
```

Saving Global Declarations

```
x = 1             # x is global here
y = 2             # y is global here
z = 3             # z is global here
def f():
   global x, y
   x = 10         # x is global here
   y = z          # y and z are global here
   g()
   print(x)       # x is global here
def g():
   global y
   x = 40         # x is local here
   y = 50         # y is global here
   z = 60         # z is local here
f()
print(x)          # displays 10
print(y)          # displays 50
print(z)          # displays 3
```

Figure 19.4

```
globalsdeclaredstack = []
```

Structure of the functioncall() Function

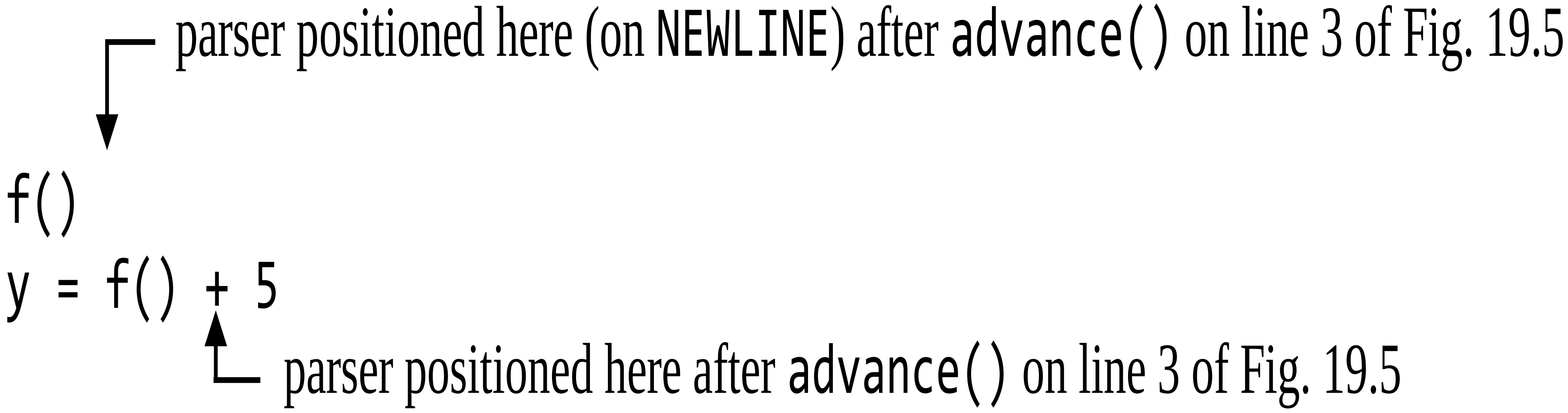
```
<simplestmt>     → <functioncall>
<factor>         → <functioncall>
<functioncall>   → NAME '(' [<relexpr> (',' <relexpr>)*] ')'

addressoffunc = getvalue(token.lexeme + '()')

arglist = []
```

```
def functioncall()
   ... <================================ missing instructions
   advance()                            # adv past right parenthesis
   returnstack.append(tokenindex)       # save return address
   infunction += 1                      # increment function call depth
   localsymtabstack.append(localsymbol) # save local symbol table
   globalsdeclaredstack.append(globalsdeclared)# save globalsdeclared
   tokenindex = addressoffunc           # reset tokenindex
   token = tokenlist[tokenindex]             # get token at this address
   try:
      functiondef(arglist)              # execute called function
   except Returnsignal:
      pass
   localsymbol = localsymtabstack.pop() # restore local symbol table
   globalsdeclared = globalsdeclaredstack.pop()# restore globalsdeclared
   infunction -= 1                      # decrement function call depth
   tokenindex = returnstack.pop()       # reposition parser at ret addr
   token = tokenlist[tokenindex]        # get current token
```

Figure 19.5



parser positioned here (on `NEWLINE`) after `advance()` on line 3 of Fig. 19.5

```
f()
y = f() + 5
```

parser positioned here after `advance()` on line 3 of Fig. 19.5

Structure of the functiondef() Function

```
localsymbol = {}
globalsdeclared = []
```

```
operandstack.append(None)
```

`functiondef()` then returns to `functioncall()` which immediately returns to its caller, which is `factor()` or `simplestmt()`.

Structure of the returnstmt() Function

1. `return <relexpr>`
2. `return`

```
y = 2*f()

f()

functioncall()
operandstack().pop()        # pop and discard value returned
```