

```

reg_names : array [0..k-1] of register_name := ["r1", "r2", ..., "rk"]
           -- ordered set of temporaries

program → stmt
  ▷ stmt.next_free_reg := 0
  ▷ program.code := ["main:" ] + stmt.code + ["goto exit"]

while : stmt1 → expr stmt2 stmt3
  ▷ expr.next_free_reg := stmt2.next_free_reg := stmt3.next_free_reg := stmt1.next_free_reg
  ▷ L1 := new_label(); L2 := new_label()
  stmt1.code := ["goto" L1] + [L2 ":" ] + stmt2.code + [L1 ":" ] + expr.code
             + ["if" expr.reg "goto" L2] + stmt3.code

if : stmt1 → expr stmt2 stmt3 stmt4
  ▷ expr.next_free_reg := stmt2.next_free_reg := stmt3.next_free_reg := stmt4.next_free_reg :=
    stmt1.next_free_reg
  ▷ L1 := new_label(); L2 := new_label()
  stmt1.code := expr.code + ["if" expr.reg "goto" L1] + stmt3.code + ["goto" L2]
             + [L1 ":" ] + stmt2.code + [L2 ":" ] + stmt4.code

assign : stmt1 → id expr stmt2
  ▷ expr.next_free_reg := stmt2.next_free_reg := stmt1.next_free_reg
  ▷ stmt1.code := expr.code + [id.stp→symbol ":" ] + expr.reg + stmt2.code

call : stmt1 → id expr stmt2
  ▷ expr.next_free_reg := stmt2.next_free_reg := stmt1.next_free_reg
  ▷ stmt1.code := expr.code + ["a1 :=" expr.reg] + ["call" id.stp→symbol] + stmt2.code

call : expr1 → id expr2
  ▷ expr2.next_free_reg := expr1.next_free_reg
  ▷ expr1.reg := reg_names[expr1.next_free_reg mod k]
  expr1.code := expr2.code + ["a1 :=" expr2.reg] + ["call" id.stp→symbol] + [expr1.reg ":" ] + rv"]

null : expr → ε
  ▷ expr.reg := "r1"           -- harmless
  expr.code := null

null : stmt → ε
  ▷ stmt.code := null

'<>' : expr1 → expr2 expr3
      handle_op(expr1, expr2, expr3, "≠")

'>' : expr1 → expr2 expr3
      handle_op(expr1, expr2, expr3, ">")

'-' : expr1 → expr2 expr3
      handle_op(expr1, expr2, expr3, "-")

id : expr → ε
  ▷ expr.reg := reg_names[expr.next_free_reg mod k]
  ▷ expr.code := [expr.reg ":" ] + expr.stp→symbol]

```

Figure 15.6 Attribute grammar to generate code from a syntax tree. Square brackets delimit individual target instructions. Juxtaposition indicates concatenation within instructions; the '+' operator indicates concatenation of instruction lists. The handle_op macro is used in three of the attribute rules. (continued)

```

macro handle_op(ref result, L_operand, R_operand : syntax_tree_node; op : string)
  ▷ L_operand.next_free_reg := result.next_free_reg
  R_operand.next_free_reg := result.next_free_reg + 1
  ▷ result.reg := L_operand.reg
  if R_operand.next_free_reg < k
    spill_code := restore_code := null
  else
    spill_code := ["*sp := " reg_names[R_operand.next_free_reg mod k]
      + ["sp := sp - 4"]
    restore_code := ["sp := sp + 4"]
      + [reg_names[R_operand.next_free_reg mod k] " := *sp"]
  result.code := L_operand.code + spill_code + R_operand.code
  + [result.reg " := " L_operand.reg op R_operand.reg] + restore_code

```

Figure 15.6 (continued)

Chapter 4, notation like *while* : *stmt* on the left-hand side of a production indicates that a *while* node in the syntax tree is one of several kinds of *stmt* node; it may serve as the *stmt* in the right-hand side of its parent production. In our attribute grammar fragment, *program*, *expr*, and *stmt* all have a synthesized attribute code that contains a sequence of instructions. *Program* has an inherited attribute name of type string, obtained from the compiler command line. *Id* has a synthesized attribute stp that points to the symbol table entry for the identifier. *Expr* has a synthesized attribute reg that indicates the register that will hold the value of the computed expression at run time. *Expr* and *stmt* have an inherited attribute next_free_reg that indicates the next register (in an ordered set of temporaries) that is available for use (i.e., that will hold no useful value at run time) immediately before evaluation of a given expression or statement. (For simplicity, we will be managing registers as if they were a stack; more on this in Section 15.3.2.) ■

Because we use a symbol table in our example, and because symbol tables lie outside the formal attribute grammar framework, we must augment our attribute grammar with some extra code for storage management. Specifically, prior to evaluating the attribute rules of Figure 15.6, we must traverse the symbol table in order to calculate stack-frame offsets for local variables and parameters (two of which—*i* and *j*—occur in the GCD program) and in order to generate assembler directives to allocate space for global variables (of which our program has none). Storage allocation and other assembler directives will be discussed in more detail in Section 15.5.

15.3.2 Register Allocation

Evaluation of the rules of the attribute grammar itself consists of two main tasks. In each subtree we first determine the registers that will be used to hold various quantities at run time; then we generate code. Our naive register allocation strat-

EXAMPLE 15.7

Stack-based register allocation

```

-- first few lines generated during symbol table traversal
.data      -- begin static data
i: .word 0  -- reserve one word to hold i
j: .word 0  -- reserve one word to hold j
.text      -- begin text (code)
-- remaining lines accumulated into program.code
main:
  a1 := r1  -- harmless
  call getint -- "getint" and "putint" are library subroutines,
              -- to be found by the linker

  r1 := rv
  i := r1
  a1 := r1  -- harmless
  call getint
  r1 := rv
  j := r1
  goto L1
L2: r1 := i  -- body of while loop
    r2 := j
    r1 := r1 > r2
    if r1 goto L3
    r1 := j  -- "else" part
    r2 := i
    r1 := r1 - r2
    j := r1
    goto L4
L3: r1 := i  -- "then" part
    r2 := j
    r1 := r1 - r2
    i := r1
L4:
L1: r1 := i  -- test terminating condition
    r2 := j
    r1 := r1 ≠ r2
    if r1 goto L2
    r1 := i
    a1 := r1
    call putint
    goto exit  -- return to operating system

```

Figure 15.7 Target code for the GCD program, generated from the syntax tree of Figure 15.2, using the attribute grammar of Figure 15.6.