null

v7 := i
v8 := j
v9 := v7 − v8
i := v9

v10 := j
v11 := i
v12 := v10 − v11
j := v12

v13 := i
a1 := v13
call putint

Start

call getint
i := rv
call getint
j := rv

v1 := i
v2 := j
v3 := v1 ≠ v2
test v3

v4 := i
v5 := j
v6 := v4 > v5
test v6

T

F

null

End

push a  
r2 := a  
push b  
r3 := b  
push c  
r4 := c  
add  
r1 := r2 + r3  
add  
r1 := r1 + r4  
push 2  
r1 := r1 / 2  
divide  
pop s  
push s  
push s  
r2 := r1 - r2  
push a  
subtract  
push s  
r3 := r1 - r3  
push b  
subtract  
push s  
r3 := r1 - r3  
push b  
subtract  
push s  
r4 := r1 - r4  
push c  
subtract  
multiply  
r3 := r3 × r4  
multiply  
r2 := r2 × r3  
multiply  
r1 := r1 × r2  
push sqrt  
call sqrt  
call
Abstract syntax tree with annotations

Front end

Scanner (lexical analysis)

Token stream

Token stream

Parser (syntax analysis)

Parse tree

Semantic analysis

Back end

Naive register allocation

Abstract syntax tree with annotations

Target code generation

Syntax tree with additional annotations

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

while : stmt1 → expr stmt2 stmt3
  ▷ 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→name ":" expr.reg] + stmt2.code

read : stmt1 → id1 id2 stmt2
  ▷ stmt1.code := ["a1 := &" id1.stp→name] -- file
    + ["call" if id2.stp→type = int then "readint" else ...]
    + [id2.stp→name ":= rv"] + stmt2.code

write : stmt1 → id expr stmt2
  ▷ expr.next_free_reg := stmt2.next_free_reg := stmt1.next_free_reg
  ▷ stmt1.code := ["a1 := &" id.stp→name] -- file
    + ["a2 :=" expr.reg] -- value
    + ["call" if id.stp→type = int then "writeint" else ...] + stmt2.code

writeln : stmt1 → id stmt2
  ▷ stmt1.code := ["a1 := &" id.stp→name] + ["call writeln"] + stmt2.code

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→name]
macro handle_op(ref result, L_operand, R_operand, op : syntax_tree_node)
    result.reg := L_operand.reg
    L_operand.next_free_reg := result.next_free_reg
    R_operand.next_free_reg := result.next_free_reg + 1
    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
-- 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 := &input    -- "input" and "output" are file control blocks
        -- located in a library, to be found by the linker
call readint    -- "readint", "writeint", and "writeln" are library subroutines
    i := rv
    a1 := &input
call readint
    j := rv
    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:    r1 := i    -- test terminating condition
    r2 := j
    r1 := r1 \neq r2
    if r1 goto L2
    a1 := &output
    r1 := i
    a2 := r1
call writeint
    a1 := &output
call writeln
goto exit    -- return to operating system
Kernel address space (inaccessible to user programs)

Stack

Shared libraries and memory-mapped files

Heap

Uninitialized data

Initialized data

Read-only code ("text") and constants

Shared libraries and memory-mapped files
Relocatable object files

A

Imports
M
M

Exports
X

Relocation

Code
... 
 r1 := &M (2300)
call M

Data
X:

B

Exports
X

Imports
M

Relocation
---

Code
... 
 r1 := &L (1000)
 r2 := Y (400)
 r3 := X

Data
X:

Y:

Executable object file

Code
... 
 r1 := &M (2300)
call M (2300)
...
 r1 := &L (1800)
 r2 := Y (3900)
 r3 := X (3300)
L:
M:

Data
X:

Y:
program :=

(7) call :=

(4) null

(8) 0.0

for

(10) 1

(7) :=

(9) call

(5) null

(8) +

(8) float

(6) ÷

null

<table>
<thead>
<tr>
<th>Index</th>
<th>Symbol</th>
<th>Type</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>void</td>
<td>type</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>type</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>real</td>
<td>type</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>getint</td>
<td>func: (1) → (2)</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>getreal</td>
<td>func: (1) → (3)</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>putreal</td>
<td>func: (3) → (1)</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>n</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>sum</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>x</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>i</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>