Team 04
Haohan Jiang, g3jiangh
Maria Yancheva, c2yanche
Timo Vink, c4vinkti
Chandeep Singh, g2singh

1 Program 2

As described above, we start with the `HALT` instruction at address 0, which will be used as our return address for the 'main procedure'.

```
0    HALT
```

Next we need to put the activation record on the stack and set the display register to point to it. The activation record contains the return address `0`, space to save a display register, and space for local variables.

```
     # Set display register
1     PUSHMT
2     SETD        0

     # Create activation record
2     PUSH        0
3     PUSH        UNDEFINED
4     PUSH        UNDEFINED
5     PUSH        UNDEFINED
6     PUSH        10     # 10 words needed for local storage in this scope
7     DUPN
```

The first line that requires generated code is line `2-5`. We need to evaluate the expression and then branch based on the output.The addresses of `a, b, c, p, q, r, w, x, t, u` are $2, 3, 4, 5, 6, 7, 8, 9, 10, 11$ from the activation record base respectively.

```
     # Get addr of p and LOAD the value
8     ADDR    0   5
9     LOAD

     # Get addr of q and LOAD the value
10    ADDR    0   6
11    LOAD

     # OR operation
12    OR

     # BF instruction
13    PUSH    17      # beginning of instructions for 2-6
14    BF
```

```
    # Then statement, get addr of a and assign it the value of 3
15    ADDR    0   2
16    PUSH    3
17    STORE
```

For the if statement in line 6, we first need to evaluate the expression:

```
    # Use De Morgan's laws to do the "and" with OR ops and negations
    # since we don't have "not" op, do 1 - bool result to get negation

      PUSH    1         # for later negation

    # Get addr of q, load value, negate, then negate again for deMorgan's law
      PUSH    1         # for later negation
      PUSH    1         # for later negation
18    ADDR    0   6
19    LOAD
20    SUB
21    SUB

    # Get addr of q, load value, negate, then negate again for deMorgan's law
      PUSH    1         # for later negation
      PUSH    1         # for later negation
22    ADDR    0   6
23    LOAD
24    SUB
25    SUB

    # Do an OR instead of AND (since we don't have AND, use deMorgan's law)
    # then negate result (also deMorgan's law)
26    OR
27    SUB
```

Then we need to emit the address for the branch false

```
28    PUSH 33   # beginning of false part
29    BF
```

True part

```
    # get addr of b and assign it the value of 2
30    ADDR    0   3
31   PUSH    2
32    STORE
33    BR      37      # branch to statement after end of if
```

Else part

```
    # get addr of b and assign it the value of 0
34    ADDR    0   3
35    PUSH    0
36    STORE
```

For the while loop on 2-7

```
    # Get the addr of c and load the value
37    ADDR    0     4
38    LOAD

    # PUSH 7 on the stack and compare
39    PUSH    7
40    LT

    # Branch to the end if false
41    PUSH    48
42    BF

    # Do block
43    ADDR    0    4
44    PUSH    8
45    STORE
46    PUSH    37
47    BR
```

For the loop on 2-8

```
    # Get the addr of a and assign it the value of 3
48    PUSH    2
49    PUSH    3
50    STORE

    # Exit statement
51    PUSH    57
52    BR

    # Get the addr of b and assign it the value of 7
53    PUSH    3
54    PUSH    7
55    PUSH    48
56    BR
```

For the while loop on 2-9

```
    # Not p
57    PUSH    1
58    ADDR    0   6
59    LOAD
```

```
60    SUB

    # Load r and do the & operation
61    PUSH    1
62    SUB
63    PUSH    1
64    ADDR    0   7
65    LOAD
66    SUB
67    OR
68    PUSH    1
69    SUB

    # Load q and do the | operation
70    ADDR    0   5
71    LOAD
72    OR

    # Branch to the end if false
73    PUSH    85
74    BF

    # Do block
75    PUSH    85
76    ADDR    0   3
77    LOAD
78    PUSH    10
79    EQ
80    PUSH    1
81    SUB
82    BF

    # Branch to beginning of while loop
83    PUSH    57
84    BR
```

For the put statement on 2-10

```
    # put "Value is "
    PUSH    86  # V
    PRINTC
    PUSH    97  # a
    PRINTC
    PUSH    108 # l
    PRINTC
    PUSH    117 # u
    PRINTC
    PUSH    101 # e
```

```
PRINTC
PUSH    32  # <space>
PRINTC
PUSH    105 # i
PRINTC
PUSH    115 # s
PRINTC
PUSH    32  # <space>
PRINTC

# evaluate a / b and print
ADDR    0   2   # load a
LOAD
ADDR    0   3   # load b
LOAD
DIV
PRINTI

# put " or "
PUSH    32  # <space>
PRINTC
PUSH    111 # o
PRINTC
PUSH    114 # r
PRINTC
PUSH    32 # <space>
PRINTC

# evaluate b * -c and print
ADDR    0   3 # load b
LOAD
ADDR    0   4 # load c
LOAD
NEG           # negate c
MUL
PRINTI

# put skip
PUSH    10  # <newline>
PRINTC
```

For the get statement on line 2-11

```
# get a, c, b
ADDR    0   2   # get a
READI
STORE
ADDR    0   4   # get c
```

```
READI
STORE
ADDR    0   3   # get b
READI
STORE
```

For the nested begin statement, create a new activation record for outer begin/end

```
# save display[1] into main
ADDR    0   2   # main's display_m
ADDR    1   0   # save prev display[1] (there is none, but follow template)
STORE

# start activation record
PUSH    <end of outer begin/ende>   # return addr

ADDR    0   0   # dynamic link
PUSH    undefined # display_m

# update display[1]
PUSHMT
PUSH    3
SUB
SETD    1

# prologue, allocate space for local storage (m, n, c)
PUSH    undefined
PUSH    3
DUPN
```

Then for line 2-14

```
# m is assigned the value of 7 - b + c
ADDR    1   3   # addr of m
PUSH    7
ADDR    0   4   # load b
LOAD
SUB
ADDR    1   5   # load c
LOAD
ADD
STORE
```

For the inner nested begin statement on line 2-15, we must put another activation record on the stack.

```
# save display[2] into outer begin/end activation record
ADDR    1   2   # outer begin/end's display_m
```

```
ADDR    2   0   # save prev display[2] (there is none, but follow template)
STORE

# start activation record
PUSH    <end of inner begin/ende>   # return addr

ADDR    1   0       # dynamic link
PUSH    undefined   # display_m

# update display[2]
PUSHMT
PUSH    3
SUB
SETD    2

# prologue, allocate space for local storage (p, q, r)
PUSH    undefined
PUSH    3
DUPN
```

For the assignment statement on line 2-17, we need to load the address of p and then create another activation record for the anon function.

```
# Load the address of p onto the stack
ADDR    2   3

# save display[3] into inner begin/end activation record
ADDR    2   2   # inner begin/end's display_m
ADDR    3   0   # save prev display[3] (there is none, but follow template)
STORE

# start activation record
PUSH    undefined           # return value
PUSH    <end of anon func>   # return addr

ADDR    2   0       # dynamic link
PUSH    undefined   # display_m

# update display[3]
PUSHMT
PUSH    3
SUB
SETD    3
```

Create another activation record for the begin/end scope inside the anon func

```
# save display[4] into anon func's activation record
ADDR    3   3   # anon func's display_m
```

```
ADDR    4   0   # save prev display[4] (there is none, but follow template)
STORE

# start activation record
PUSH    <end of this scope>   # return addr

ADDR    3   0       # dynamic link
PUSH    undefined   # display_m

# update display[3]
PUSHMT
PUSH    3
SUB
SETD    4
```

Then for `p <= a`

```
# Assign p the value of a
ADDR    0   2   # addr of p
ADDR    0   5   # load a
LOAD
STORE
```

Epilogue/cleanup for begin/end inside anon func.

```
# no local storage to pop
POP     # pop display_m

# dynamic link is now at top, revert display[4]
PUSH    3      # load caller's display_m
LOAD
SETD    4

# return to return addr
BR
```

Yield statement

```
# yields r - b
ADDR    3   3
LOAD
ADDR    0   4
LOAD
SUB
```

Epilogue/cleanup for anon func.

```
# no local storage to pop
POP     # pop display_m
```

```
# dynamic link is now at top, revert display[3]
PUSH    3      # load caller's display_m
LOAD
SETD    3

# return to return addr
BR
```

Epilogue/cleanup for inner begin/end (starting on line 2-15)

```
# pop local storage
PUSH    3
POPN

POP     # pop display_m

# dynamic link is now at top, revert display[2]
PUSH    3      # load caller's display_m
LOAD
SETD    2

# return to return addr
BR
```

Epilogue/cleanup for our begin/end (starting on line 2-12)

```
# pop local storage
PUSH    3
POPN

POP     # pop display_m

# dynamic link is now at top, revert display[1]
PUSH    3      # load caller's display_m
LOAD
SETD    1

# return to return addr
BR
```

While loop on line 2-20

```
# evaluate expression ! ( p | q )
PUSH    1       # used later to negate with SUB
ADDR    0   5   # load p
LOAD
ADDR    0   6   # load q
LOAD
```

```
    OR
    SUB             # negate

    # branch to end of loop
    PUSH <addr>     # addr of end of while loop
    BF

    # exit when p & r
    # evaluate p & r
    PUSH    1
    ADDR    0   5   # load p
    LOAD
    ADDR    0   7   # load r
    LOAD
    SUB

    # branch when !(p & r) is false
    PUSH <while-loop end addr>
    BF

    <normal-loop-beginning>

    # if w <= a then exit end
    # evaluate w <= a
    PUSH    1       # for future negation
    ADDR    0   8   # load w
    LOAD
    ADDR    0   2   # load a
    LOAD

    # perform > op and negate the result
    GT
    SUB             # negate
    PUSH    <end of normal-loop addr>
    BF

    PUSH    <while-loop end addr>
    BR

    <end of if>

    # t <= { anon function }
    ADDR    0   10      # push addr of t

    # save display[1] into main's activation record
    ADDR    0   3   # main's display_m
    ADDR    1   0   # save prev display[1] (there is none, but follow template)
    STORE
```

```
# start activation record
PUSH    <end of anon func>   # return addr

ADDR    0   0       # dynamic link
PUSH    undefined   # display_m

# update display[1]
PUSHMT
PUSH    3
SUB
SETD    1

# prologue, allocate local storage (boolean m)
PUSH    undefined
PUSH    1
POPN

# m <= w < t
ADDR    1   3   # addr of m
ADDR    0   8   # load w
LOAD
ADDR    0   10  # load t
LOAD
LT
STORE

# if m then t <= t + c end
ADDR    1   3   # load m
PUSH    <end of if>
BF
ADDR    0   10  # addr of t
ADDR    0   10  # load t
ADDR    0   4   # load c
ADD
STORE           # t <= t + c

<end of if>

# yields t
# set return value to value of t
ADDR    1   0   # return val addr
ADDR    0   10  # load t
LOAD
STORE


# epilogue for anon func
```

```
# pop local storage
PUSH    1
POPN

POP     # pop display_m

# dynamic link is now at top, revert display[1]
PUSH    3      # load caller's display_m
LOAD
SETD    1

# return to return addr
BR

# save return value to t
# top of stack should be return val, followed by addr of t
STORE

# go back to beginning of loop
PUSH    <normal-loop-beginning>
BR

<end of normal-loop>
<while-loop end>
```

We're now at the end of the 'main procedure'. So we need to clean up the activation record and branch to the return address, which is where the `HALT` instruction is.

```
# Clean up activation record (10 vars + display[m] + dynamic link + return addr)
PUSH        13
POPN

# Branch to return address (HALT)
ADDR        0       0
LOAD
BR
```