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

# 1 Example Programs

## 1.1 Program 1

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

```
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` (1 word), space to save a display register (1 word), and space for local variables (2683 words).

```
# Set display register
PUSHMT
SETD        0

# Create activation record
PUSH        0
PUSH        UNDEFINED
PUSH        2684
DUPN
```

The first line in the program that requires computation is line `1-4`. We need to evaluate the expression and store the result in the address of `k`. The addresses of `i, j, k, l` are 2, 3, 4, 5 from the activation record base respectively.

```
# Get address of k
ADDR        0       4

# Calculate (i + 3)
ADDR        0       2
LOAD
PUSH        3
ADD

# Calculate (j * k), subtract from the above
ADDR        0       3
LOAD
ADDR        0       4
MUL
SUB

# Calculate (k / l), add to the above
```

```
ADDR        0       4
LOAD
ADDR        0       5
DIV
ADD

# Store result in k
STORE
```

Next up, we have lines `1-6` and `1-7`. We need to store constants in `p` and `q`, which are at offsets 7 and 8 from the activation record base respectively.

```
# Store TRUE in p
ADDR        0       7
PUSH        1
STORE

# Store FALSE in q
ADDR        0       8
PUSH        0
STORE
```

Next up we have line `1-8`. We need to evaluate the expression and store the result in the address of `r`. The addresses of `p, q, r, s` are 7,8,9,10 from the activation record base respectively. Recall that $s \land \neg p \equiv \neg(\neg s \lor p)$.

```
# Get address of r
ADDR        0       9

# Calculate (!q)
PUSH        MACHINE_TRUE
ADDR        0       8
LOAD
SUB

# Calculate (p | q), OR with result above
ADDR        0       7
LOAD
ADDR        0       8
LOAD
OR
OR

# Calculate (s & !p), OR with result above
PUSH        MACHINE_TRUE
PUSH        MACHINE_TRUE
ADDR        0       10
LOAD
SUB
```

```
PUSH        MACHINE_TRUE
PUSH        MACHINE_TRUE
ADDR        0       7
LOAD
SUB
SUB
OR
SUB
OR

# Store result in r
STORE
```

We're now at line 1-9. We need to evaluate the expression and store the result at the address of p. The addresses of `i, j, k, l, p` are 2, 3, 4, 5, 7 from the activation record base respectively. Recall that $a \leq b \equiv \neg(a > b)$.

```
# Get address of p
ADDR        0       9

# Calculate (i < j)
ADDR        0       2
LOAD
ADDR        0       3
LOAD
LT

# Calculate (k <= l), OR with result above
PUSH        MACHINE_TRUE
ADDR        0       5
LOAD
ADDR        0       4
LOAD
LT
SUB
OR

# Calculate (j = l), OR with result above
ADDR        0       3
LOAD
ADDR        0       5
LOAD
EQ
OR

# Store result in p
STORE
```

Similar to before, for line 1-19 we need to evaluate the expression and store the result at the address of s using the fact that $a \neq b \equiv \neg(a = b)$ as well as the two equivalences outlined for the previous two lines. The addresses of `j, k, m, r, s` are 3, 4, 6, 9, 10 from the activation record base respectively.

```
# Get address of s
ADDR        0       9

# Calculate !(k != m)
PUSH        MACHINE_TRUE
PUSH        MACHINE_TRUE
PUSH        MACHINE_TRUE
ADDR        0       4
LOAD
ADDR        0       6
LOAD
EQ
SUB
SUB

# Calculate !(j >= k), OR with result above and negate
PUSH        MACHINE_TRUE
ADDR        0       3
LOAD
ADDR        0       4
LOAD
LT
SUB
OR
SUB

# Calculate !(r = s), OR with result above
PUSH        MACHINE_TRUE
ADDR        0       9
LOAD
ADDR        0       10
LOAD
EQ
SUB
OR

# Store result in s
STORE
```

Next up is line 1-11. No new concepts here. The addresses of `q, r, s` are 8, 9, 10 from the activation record base respectively.

```
# Get address of q
```

```
ADDR        0       8

# Calculate (r = s)
ADDR        0       9
LOAD
ADDR        0       10
LOAD
EQ

# Calculate (!s != r), OR with result above
PUSH        MACHINE_TRUE
PUSH        MACHINE_TRUE
ADDR        0       10
LOAD
SUB
ADDR        0       9
LOAD
EQ
SUB
OR

# Store result in q
STORE
```

Next line requiring any computation is line `1-14`. We know the stride of the first dimension of `B` is 151. The base addresses of `A, B` are 12, 19 from the activation record base respectively, and the offsets of `i, j` are 2, 3 respectively.

```
# Get base address of B
ADDR        0       19

# Calculate offset due to first dimension
ADDR        0       2
LOAD
PUSH        1
ADD
PUSH        -100
SUB
PUSH        151
MUL

# Calculate offset due to second dimension
ADDR        0       3
LOAD
PUSH        100
SUB
PUSH        -40
SUB
```

```
# Combine results to find address of B[i + 1, j - 100]
ADD
ADD

# Get value at A[j - 2]
ADDR        0       12
ADDR        0       3
LOAD
PUSH        2
SUB
PUSH        1
SUB
ADD
LOAD

# Store result in B[i + 1, j - 100]
STORE
```

And similarly for line 1-15. We know the stride of the first dimension of D is 50. The base addresses of C, D are 1680, 1685 from the activation record base respectively, and the offsets of i, k are 2, 4 respectively.

```
# Get address of C[-4]
ADDR        0       1680
PUSH        -4
PUSH        -7
SUB
ADD

# Get base address of D
ADDR        0       1685


# Calculate offset due to first dimension
ADDR        0       2
LOAD
PUSH        20
ADD
PUSH        -100
SUB
PUSH        50
MUL

# Calculate offset due to second dimension
ADDR        0       4
LOAD
PUSH        7
```

```
    SUB
    PUSH        1
    SUB

    # Combine results to find address of D[i + 20, k - 7]
    ADD
    ADD

    # Store result in C[-4]
    STORE
```

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
    PUSH        2684
    POPN

    # Branch to return address
    ADDR        0       0
    LOAD
    BR
```

### 1.2 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
```

### 1.3 Program 3

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: return address, dynamic link, display[M], 8 params+vars
3    PUSH       0
4    PUSH       UNDEFINED
5    PUSH       UNDEFINED
6    PUSH       UNDEFINED
7    PUSH       8
8    DUPN
```

The first line requiring code generation is line `3-29`. Before calling procedure Q, we save the display data for lexical level 1. Since the main program does not have a return value, it is equivalent to a procedure (i.e., its display[M] entry is the third one in its activation record stack).

```
      # Get address of display[M] entry in the activation record of main program
9     ADDR       0      2

      # Get display data for lexical level 1, and store it in the main program
10    ADDR       1      0
11    STORE
```

Allocate space for control items in activation record of Q: return address, dynamic link and display:

```
12    PUSH      ?? return_addr_for_Q
13    ADDR      0       0
14    PUSH      UNDEFINED
```

Next, update the display for lexical level 1:

```
15    PUSHMT
16    PUSH          2
17    SUB
18    SETD          1
```

Evaluate argument expressions, write them to activation record, and branch to the procedure body code.

Q: argument 1

```
    # Not p
19  1
20  ADDR    0    7
21  LOAD
22  SUB

    # Or q
23  ADDR    0    8
24  LOAD
25  OR
```

Q: argument 2. Execute function call to F.
F (call 1): store display data for lexical level 1 within caller.

```
26  ADDR    1    2
27  ADDR    1    0
28  STORE
```

F (call 1): allocate space for return value, return address, dynamic link and display.

```
29  PUSH    UNDEFINED
30  PUSH    ?? return_addr_for_F1
31  ADDR    1    0
32  PUSH    UNDEFINED
```

F (call 1): update the display for lexical level 1 to point to current activation record.

```
33  PUSHMT
34  PUSH    3
35  SUB
36  SETD    1
```

F (call 1): evaluate parameter expressions. Argument 1: execute function call to F.
F (call 2): store display data for lexical level 1 within caller:

```
37  ADDR    1    3
38  ADDR    1    0
39  STORE
```

F (call 2): allocate space for return value, return address, dynamic link and display.

```
40  PUSH    UNDEFINED
41  PUSH    ?? return_addr_for_F2
42  ADDR    1    0
43  PUSH    UNDEFINED
```

F (call 2): update the display for lexical level 1 to point to current activation record.

```
44  PUSHMT
45  PUSH    3
46  SUB
47  SETD    1
```

F (call 2): evaluate parameter expressions. Argument 1: b, argument 2: p. Both exist in lexical level 0.

```
48    ADDR      0       4
49    LOAD
50    ADDR      0       7
51    LOAD
```

F (call 2): branch to function entrance code.

```
52    PUSH      addr_F_entrance_code
53    BR
```

F entrance code: allocate space for parameters and identifiers.

```
54    PUSH      UNDEFINED
55    PUSH      2
56    DUPN

      # F body code
57    ADDR      1       5
58    LOAD
59    PUSH      branch_false_addr
60    BF
      # True condition code: return m+b
61    ADDR      1       4
62    LOAD
63    ADDR      0       4
64    LOAD
65    ADD
66    PUSH      addr_F_epiloguecode
67    BR

      # False condition code: return c-m
68    ADDR      0       5
69    LOAD
70    ADDR      1       4
71    LOAD
72    SUB
73    PUSH      addr_F_epiloguecode
74    BR
```

F epilogue code: pop all params + identifiers, and restore the display data from parent's activation record. Finally, the return address is on the top of the stack, so simply branch to it.

```
75    PUSH      2
76    POPN
77    POP
78    PUSH      3
79    LOAD
80    SETD      1
81    BR
```

F (call 1): argument 2 (not q).

```
82    PUSH      1
83    ADDR      0       8
84    LOAD
85    SUB
```

F (call 1): branch to function entrance code.

```
86    PUSH      addr_F_entrance_code
87    BR
```

Q: argument 3. Execute anonymous function call.
Anonymous function: store current display[M] into the caller (Q).

```
88    ADDR      1       2
89    ADDR      2       0
90    STORE
```

Anonymous function: allocate space for return value, return address, dynamic link and display.

```
91    PUSH      UNDEFINED
92    PUSH      return_addr_anon
93    ADDR      1       0
94    PUSH      UNDEFINED
```

Anonymous function: update display.

```
95    PUSHMT
96    PUSH      3
97    SUB
98    SETD      2
```

Anonymous function: no parameter expressions to evaluate. Execute body code. First statement invokes a call to procedure P.
P: store current display[M] into the caller (anon).

```
99    ADDR      2       3
100   ADDR      1       0
101   STORE
```

P: allocate space for return address, dynamic link and display.

```
102   PUSH      return_addr_P
103   ADDR      2       0
104   PUSH      UNDEFINED
```

P: update display.

```
105   PUSHMT
106   PUSH      2
107   SUB
108   SETD      1
```

P: no parameter expressions to evaluate. Branch to procedure entrance code and body.

```
109   PUSH      addr_P_entrancecode
110   BR
```

P: entrance code. Allocate space for identifiers. Then execute body statements.

```
111   PUSH      UNDEFINED
112   PUSH      2
113   DUPN

      # P body code
114   ADDR      0       7
115   LOAD
116   PUSH      addr_fwd
117   BF
      # True condition code.
118   PUSH      addr_epilogue_P
119   BR

      # Assignment e <= a
120   ADDR      1       3
121   ADDR      0       3
122   LOAD
123   STORE

      # Return
124   PUSH      addr_epilogue_P
125   BR
```

P: epilogue. Pop all identifiers off the stack. Pop display. Restore display from caller. Then branch to return address.

```
126   PUSH      2
127   POPN
128   POP
129   PUSH      3
130   LOAD
131   SETD      1
132   BR
```

Anonymous function: return statement.

```
133   PUSH      1
134   ADDR      0       7
135   LOAD
136   ADDR      0       8
137   EQ
138   SUB
139   PUSH      addr_epilogue_anon
140   BR
```

Anonymous function: epilogue. Pop display, restore display, branch to return address.

```
141   POP
142   PUSH     2
143   LOAD
144   SETD     2
145   BR
```

Q: branch to function entrance code.

```
146   PUSH     addr_entrancecode_Q
147   BR
```

Q: entrance code. Allocate space for params and identifiers. Then execute body statements.

```
148   PUSH     UNDEFINED
149   PUSH     6
150   DUPN

      # Now call function F.
```

F (call 3): save current display in caller (Q).

```
151   ADDR     1       2
152   ADDR     1       0
153   STORE
```

F (call 3): allocate space for return value, return address, dynamic link and display.

```
154   PUSH     UNDEFINED
155   PUSH     ?? return_addr_for_F3
156   ADDR     1       0
157   PUSH     UNDEFINED
```

F (call 3): update display.

```
158   PUSHMT
159   PUSH     3
160   SUB
161   SETD     1
```

F (call 3): evaluate parameter expressions. Argument 1: t - n + a.

```
162   ADDR     1       6
163   LOAD
164   ADDR     1       4
165   LOAD
166   SUB
167   ADDR     0       3
168   LOAD
169   ADD
```

F (call 3): argument 2.

```
170   PUSH     1
```

At this point, need to execute function G. Save current display in caller (F).

```
151   ADDR     1       3
152   ADDR     2       0
153   STORE
```

G: allocate space for return value, return address, dynamic link and display.

```
154   PUSH     UNDEFINED
155   PUSH     return_addr_G
156   ADDR     1       0
157   PUSH     UNDEFINED
```

G: update display.

```
158   PUSHMT
159   PUSH     3
160   SUB
161   SETD     2
```

G: no parameters to evaluate. Branch to function entrance code.

```
162   PUSH     addr_entrancecode_G
163   BR
```

G: entrance code. Allocate space for identifiers. Then execute body code.

```
164   PUSH     UNDEFINED
165   PUSH     2
166   DUPN
```

```
      # Body of G: execute anonymous function.
```

Anonymous function (call 2): save current display into caller.

```
167   ADDR     2       3
168   ADDR     3       0
169   STORE
```

Anonymous function (call 2): allocate space for return value, return address, dynamic link and display.

```
170   PUSH     UNDEFINED
171   PUSH     return_addr_anon2
172   ADDR     2       0
173   PUSH     UNDEFINED
```

Anonymous function (call 2): update display.

```
174   PUSHMT
175   PUSH     3
176   SUB
177   SETD     3
```

Anonymous function (call 2): no parameters to evaluate. Execute function entrance code and body statements.

```
PUSH      UNDEFINED
PUSH      2
DUPN
ADDR      3       5
ADDR      0       5
STORE

    # Call procedure P.
```

P (call 2): store current display[M] into the caller (anon 2).

```
 ADDR      3       3
ADDR      1       0
STORE
```

P (call 2): allocate space for return address, dynamic link and display.

```
PUSH      return_addr_P
ADDR      3       0
PUSH      UNDEFINED
```

P: update display.

```
PUSHMT
PUSH      2
SUB
SETD      1
```

P: no parameter expressions to evaluate. Branch to procedure entrance code and body.

```
PUSH      addr_P_entrancecode
BR
```

Anonymous function (call 2): execute return statement.

```
ADDR      3       5
LOAD
ADDR      2       4
LOAD
ADD
ADDR      1       8
LOAD
SUB
PUSH      12
LT
PUSH      addr_epilogue_anon2
BR
```

Anonymous function (call 2): epilogue. Clean up allocated space. Pop display. Restore display. Then branch to return address.

```
208   PUSH     2
209   POPN
210   POP
211   PUSH     3
212   LOAD
213   SETD     3
214   BR
```

G: at this point, the return expression (returned by the anonymous function) is at the top of the stack. Now execute the return statement.

```
215   PUSH     addr_epilogue_G
216   BR
```

G: epilogue. Clean up allocated space. Pop display. Restore display. Then branch to return address.

```
217   PUSH     2
218   POPN
219   POP
220   PUSH     3
221   LOAD
222   SETD     2
223   BR
```

F (call 3): argument 2 processing. Right now at top of stack we have the return value of G.

```
      # 1 - return value of G => !G
224   SUB
225   ADDR     0       9
226   LOAD
227   OR
```

F (call 3): branch to function entrance code.

```
228   PUSH     addr_entrancecode_F
229   BR
```

Q: at this point we have the return value of F at the top of the stack. Print it out, then print out a newline (skip), which is ASCII character code 10.

```
      # Print out return value of F
230   PRINTI
      # Print out newline (skip)
231   PUSH     10
232   PRINTC
```

Q: the body has been executed. Now go to epilogue code.

```
233   PUSH     addr_epilogue_Q
234   BR
```

Q: epilogue. Clean up allocated space. Pop display. Restore display. Branch to return address.

```
235   PUSH      6
236   POPN
237   POP
238   PUSH      2
239   LOAD
240   SETD      1
241   BR
```

Main program: we have finished executing all body statements. Now branch to epilogue code.

```
242   PUSH      addr_epilogue_main
243   BR
```

Main program: epilogue. Pop identifiers, pop display. Branch to return address.

```
244   PUSH      8
245   POPN

      # pop display and dynamic link words
246   POP
247   POP

      # Branch to return address
248   ADDR        0       0
249   LOAD
250   BR
```