```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Simple library functions

write := 0x8B55, 0x8BEC, 0x085E, 0x4E8B, 0x8B04, 0x0656,
         0x00B8, 0xCD40, 0x7321, 0x3102, 0x8BC0, 0x5DE5,
         0x90C3

; 0100 55             PUSH    BP
; 0101 8BEC           MOV     BP,SP
; 0103 8B5E08         MOV     BX,[BP+08]
; 0106 8B4E04         MOV     CX,[BP+04]
; 0109 8B5606         MOV     DX,[BP+06]
; 010C B80040         MOV     AX,4000
; 010F CD21           INT     21
; 0111 7302           JNB     0115
; 0113 31C0           XOR     AX,AX
; 0115 8BE5           MOV     SP,BP
; 0117 5D             POP     BP
; 0118 C3             RET

read := 0x8B55, 0x8BEC, 0x085E, 0x4E8B, 0x8B04, 0x0656,
        0x00B8, 0xCD3F, 0x7321, 0x3102, 0x8BC0, 0x5DE5,
        0x90C3

; 0100 55             PUSH    BP
; 0101 8BEC           MOV     BP,SP
; 0103 8B5E08         MOV     BX,[BP+08]
; 0106 8B4E04         MOV     CX,[BP+04]
; 0109 8B5606         MOV     DX,[BP+06]
; 010C B8003F         MOV     AX,3F00
; 010F CD21           INT     21
; 0111 7302           JNB     0115
; 0113 31C0           XOR     AX,AX
; 0115 8BE5           MOV     SP,BP
; 0117 5D             POP     BP
; 0118 C3             RET

open := 0x8955, 0x8BE5, 0x0656, 0x468B, 0x3C04, 0x7501,
        0x3107, 0xB8C9, 0x3C00, 0x02EB, 0x3DB4, 0x21CD,
        0x0273, 0xC031, 0xEC89, 0xC35D

; 0100 55             PUSH    BP
; 0101 89E5           MOV     BP,SP
; 0103 8B5606         MOV     DX,[BP+06]
; 0106 8B4604         MOV     AX,[BP+04]
; 0109 3C01           CMP     AL,01
; 010B 7507           JNZ     0114
; 010D 31C9           XOR     CX,CX
; 010F B8003C         MOV     AX,3C00
; 0112 EB02           JMP     0116
; 0114 B43D           MOV     AH,3D
; 0116 CD21           INT     21
; 0118 7302           JNB     011C
; 011A 31C0           XOR     AX,AX
; 011C 89EC           MOV     SP,BP
; 011E 5D             POP     BP
; 011F C3             RET

close := 0x8955, 0x8BE5, 0x045E, 0x00B8, 0xCD3E, 0x7221,
         0x3102, 0x89C0, 0x5DEC, 0x90C3

; 0100 55             PUSH    BP
; 0101 89E5           MOV     BP,SP
; 0103 8B5E04         MOV     BX,[BP+04]
; 0106 B8003E         MOV     AX,3E00
; 0109 CD21           INT     21
; 010B 7202           JB      010F
; 010D 31C0           XOR     AX,AX
; 010F 89EC           MOV     SP,BP
; 0111 5D             POP     BP
; 0112 C3             RET
; 0113 90             NOP

seek := 0x8955, 0x8BE5, 0x065E, 0x178B, 0x4F8B, 0x8B02,
        0x085E, 0x468B, 0xB404, 0xCD42, 0x7321, 0xB806,
        0xFFFF, 0xFFBA, 0x8BFF, 0x065E, 0x0789, 0x5789,
        0x8902, 0x5DEC, 0x90C3

; 0100 55             PUSH    BP
; 0101 89E5           MOV     BP,SP
; 0103 8B5E06         MOV     BX,[BP+06]
; 0106 8B17           MOV     DX,[BX]
; 0108 8B4F02         MOV     CX,[BX+02]
; 010B 8B5E08         MOV     BX,[BP+08]
; 010E 8B4604         MOV     AX,[BP+04]
; 0111 B442           MOV     AH,42
; 0113 CD21           INT     21
; 0115 7306           JNB     011D
; 0117 B8FFFF         MOV     AX,FFFF
; 011A BAFFFF         MOV     DX,FFFF
; 011D 8B5E06         MOV     BX,[BP+06]
; 0120 8907           MOV     [BX],AX
; 0122 895702         MOV     [BX+02],DX
; 0125 89EC           MOV     SP,BP
; 0127 5D             POP     BP
; 0128 C3             RET
; 0129 90             NOP

unlink := 0x8955, 0x8BE5, 0x0456, 0x00B8, 0xCD41, 0x7221,
          0x3102, 0x89C0, 0x5DEC, 0x90C3

; 0100 55             PUSH    BP
; 0101 89E5           MOV     BP,SP
; 0103 8B5604         MOV     DX,[BP+04]
; 0106 B80041         MOV     AX,4100
; 0109 CD21           INT     21
; 010B 7202           JB      010F
; 010D 31C0           XOR     AX,AX
; 010F 89EC           MOV     SP,BP
; 0111 5D             POP     BP
; 0112 C3             RET
; 0113 90             NOP

stdin  := 0  ; MS-DOS standard input
stdout := 1  ; MS-DOS standard output
stderr := 2  ; and standard error

atoi(s) :=
{
    n := 0
    p := $
    @s&0xFF >= '0' ?
    {
        @s&0xFF <= '9' ?
        {
            n = n*10 + (@s&0xFF - '0')
            s = s+1
            $ = p
        }
    }
    n    ; Return value
}

itoa(n, s) :=
{
    @s = @s & 0xFF00
    c := 0
    p := $
    m := n / 10
    d := n % 10

    m || d ?
    {
        i := c
        q := $
        i > 0 ?
        {
            @(s+i) = (@(s+i)&0xFF00) + (@(s+i-1)&0xFF)
            i = i - 1
            $ = q
        }

        c = c + 1
        @s = (@s & 0xFF00) + d + '0'
        n = m
        $ = p
    }
    c == 0 ?
    {
        @s = '0'
        c = c + 1
    }
    c
}

itox(n, s) :=
{
    i := 0
    x := ?
    p := $
    i < 4 ?
    {
        x = (n & 0xF000) >> 12
        @(s+i) = @(s+i) & 0xFF00 +
                 (x > 9 ? x + 'A' - 10, x + '0')
        n = n << 4
        i = i + 1
```

```
        $ = p
    }
    @s[2] = 0
    i
}

strlen := 0x8955, 0x57E5, 0x7E8B, 0xB904, 0xFFFF, 0x30FC,
          0xF2C0, 0xB8AE, 0xFFFE, 0xC829, 0x5D5F, 0x90C3

; 0100 55             PUSH    BP
; 0101 89E5           MOV     BP,SP
; 0103 57             PUSH    DI
; 0104 8B7E04         MOV     DI,[BP+04]
; 0107 B9FFFF         MOV     CX,FFFF
; 010A FC             CLD
; 010B 30C0           XOR     AL,AL
; 010D F2             REPNZ
; 010E AE             SCASB
; 0110 B8FEFF         MOV     AX,FFFE
; 0113 29C8           SUB     AX,CX
; 0115 5F             POP     DI
; 0116 5D             POP     BP
; 0117 C3             RET
; 0118 90             NOP

strncpy := 0x8955, 0x57E5, 0x8B56, 0x067E, 0x4E8B, 0xFC04,
           0xC030, 0xAEF2, 0x4E2B, 0xF704, 0x8BD9, 0x087E,
           0x768B, 0xF306, 0x5EA4, 0x5D5F, 0x90C3

; 0100 55             PUSH    BP
; 0101 89E5           MOV     BP,SP
; 0103 57             PUSH    DI
; 0104 56             PUSH    SI
; 0105 8B7E06         MOV     DI,[BP+06]
; 0108 8B4E04         MOV     CX,[BP+04]
; 010B FC             CLD
; 010C 30C0           XOR     AL,AL
; 010E F2             REPNZ
; 010F AE             SCASB
; 0110 2B4E04         SUB     CX,[BP+04]
; 0113 F7D9           NEG     CX
; 0115 8B7E08         MOV     DI,[BP+08]
; 0118 8B7606         MOV     SI,[BP+06]
; 011B F3             REPZ
; 011C A4             MOVSB
; 011D 5E             POP     SI
; 011E 5F             POP     DI
; 011F 5D             POP     BP
; 0120 C3             RET
; 0121 90             NOP

strcpy(d, s) := strncpy(d, s, 0xFFFF)

memcmp := 0x8955, 0x57E5, 0x8B56, 0x044E, 0x7E8B, 0x8B08,
          0x0676, 0xA6F3, 0x01B8, 0x7500, 0x3102, 0x5EC0,
          0x5D5F, 0x90C3

; 0100 55             PUSH    BP
; 0101 89E5           MOV     BP,SP
; 0103 57             PUSH    DI
; 0104 56             PUSH    SI
; 0105 8B4E04         MOV     CX,[BP+04]
; 0108 8B7E08         MOV     DI,[BP+08]
; 010B 8B7606         MOV     SI,[BP+06]
; 010E F3             REPZ
; 010F A6             CMPSB
; 0110 B80100         MOV     AX,0001
; 0113 7502           JNZ     0117
; 0115 31C0           XOR     AX,AX
; 0117 5E             POP     SI
; 0118 5F             POP     DI
; 0119 5D             POP     BP
; 011A C3             RET
; 011B 90             NOP

strncmp(d, s, n) :=
{
    k := strlen(d)
    l := strlen(s)

    0 != ((k > n && l > n) || (k == l)) ?
    {
        k < n ? n = k
        memcmp(d, s, n)
    }, 0xFFFF
}

strcmp(d, s) := strncmp(d, s, 0xFFFF)

puts(o, s) := write(o, s, strlen(s))
putn(o, n) :=
{
    s[3] := ?
    n = itoa(n, #s)
    write(o, #s, n)
}

putx(o, n) :=
{
    s[3] := ?
    n = itox(n, #s)
    write(o, #s, n)
}

at(s, n) := (@s[n >> 1] & ((n&1) ? 0xFF00, 0x00FF)) >>
            ((n&1) ? 8, 0)

printf(s, o) :=
{
    n := 0
    m := 0
    c := ?
    p := #s + 2

    r := $
    c = at(s, n)
    c != 0 ?
    {
        c == '%' ?
        {
            write(o, s + m, n - m)
            m = n + 2
            n = n + 1
            c = at(s, n)
            c == 's' ?
            {
                puts(o, @p)
            },
            c == 'd' ?
            {
                putn(o, @p)
            },
            c == 'x' ?
            {
                putx(o, @p)
            }
            p = p + 2
        }
        n = n + 1
        c = at(s, n)
        $ = r
    }
    m < n ? write(o, s + m, n - m)
}

getc(f) :=
{
    c := 0

    read(f, #c, 1) ? c, 0xFFFF
}

ungetc(f) :=
{
    p[2] := 0xFFFF, 0xFFFF
    seek(f, #p, 1)
}

isalphabetic(c) :=
{
    c>='a' && c<='z' || c>='A' && c<='Z' || c=='_'
}

isdigit(c) :=
{
    c >= '0' ? c <= '9'
}

isalphanumeric(c) :=
{
    isalphabetic(c) || isdigit(c)
}

isspace(c) :=
{
    c == ' ' || c == '\t' || c == '\r' || c == '\n'
}
```

```
; End of library definitions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

FILE_END            := 0
NUMBER              := 1
CHARACTER           := 2
STRING              := 3
SYMBOL              := 4
QUESTION_MARK       := 5
UNARY_OPERATOR      := 6
BINARY_OPERATOR     := 7
OPERATOR            := 8
COMMA               := 9
OPEN_PARENTHESIS    := 10
CLOSE_PARENTHESIS   := 11
OPEN_BRACKET        := 12
ASSIGN_OPERATOR     := 13
CLOSE_BRACKET       := 14
OPEN_BRACE          := 15
CLOSE_BRACE         := 16
DOLLAR              := 17

OPR_AND  := 100
OPR_MUL  := 101
OPR_DIV  := 102
OPR_OR   := 103
OPR_ADD  := 104
OPR_SUB  := 105
OPR_LT   := 106
OPR_GT   := 107
OPR_LTE  := 108
OPR_GTE  := 109
OPR_EQU  := 110
OPR_NEQ  := 111
OPR_SET  := 112
OPR_NOT  := 113
OPR_ADR  := 114
OPR_DRF  := 115
OPR_SHL  := 116
OPR_SHR  := 117
OPR_MOD  := 118
OPR_LND  := 119
OPR_LOR  := 120
OPR_LNO  := 121

src := ?
com := ?
tmp := ?

function     := ?
token        := ?
character    := ?
address      := 0x0100
heapadr      := 0
heapadj      := 0
line         := 1
number       := ?
operator     := ?
position[2]  := ?
namespace    := 0

definition          := ?
primary_expression  := ?

symbol[128]  := ?
string[128]  := ?
length       := ?

buffer       := ?

space[16384] := ?
thisspace    := 0
currspace    := 16383
symbspace    := 0
basespace    := 0

;Namespace structure:
;+-----------------------+
;| Previous namespace    |= 0 <-+
;| Next namespace        | -+   |
;| Flags                 |  |   |
;| Symbol space          |--+---+--+
;+-----------------------+  |   |  |
;| Symbol                |--+---+--+
;| Value                 |  |   |  |
;+-----------------------+  |   |  |
;| Symbol                |--+---+--+--+
;| Value                 |  |   |  |  |
;| ...                   |  |   |  |  |
;+-----------------------+  |   |  |  |
;| Previous namespace    |<-+ --+  |  | <--thisspace
;| Next namespace        |= 0      |  |
;| Flags                 |         |  |
;| Symbol space          |--+      |  |
;| ...                   |  |      |  | <--currspace
;|                       |  |      |  |
;| ...                   |  |      |  |
;| Symbol\0              |<-+      |  | <--symbspace
;| Symbol\0              |<--------+--+
;| Symbol\0(*)           |<--------+
;+-----------------------+
; *High bit in first char. implies heap (vs. code) symbol

error(code) :=
{
    pMsg[18] := "no start address\r\n\0",
          "out of symbol space\r\n\0",
          "unbalanced curly braces\r\n\0",
          "name too long\r\n\0",
          "string literal too long\r\n\0",
          "unexpected token (ASCII %d)\r\n\0",
          "unknown symbol %s\r\n\0",
          "cannot take address of non-lvalue\r\n\0",
          "cannot call non-lvalue\r\n\0",
          "array operator on non-lvalue\r\n\0",
          "cannot assign to non-lvalue\r\n\0",
          "invalid operator in constant expression\r\n\0",
          "operator stack exhausted\r\n\0",
          "local functions not supported\r\n\0",
          "')' expected\r\n\0",
          "array size not constant\r\n\0",
          "syntax horror\r\n\0",
          "unknown operator\r\n\0"

    printf(line, code + 1000,
        "Error (%d) in line %d: \0", stderr)
    printf(@(#code + 2), pMsg[code], stderr)
    0
}

writecode(c, n) :=
{
    write(com, c, n)
    address = address + n
}

writeheap(c, n) :=
{
    write(tmp, c, n)
    heapadr = heapadr + n
}

newspace(flag) :=
{
    thisspace ? space[thisspace - 1] = currspace

    space[currspace] = thisspace
    space[currspace-1] = 0
    space[currspace-2] = flag
    space[currspace-3] = symbspace

    thisspace = currspace
    currspace = currspace - 4
    namespace = namespace + 1
    currspace > symbspace ? 1, error(1)
}

oldspace(flag) :=
{
    (flag != space[thisspace - 2]) ? error(2),
    {
        currspace = thisspace
        symbspace = space[thisspace - 3]
        thisspace = space[thisspace]
        namespace = namespace - 1
        space[thisspace - 1] = 0
        1
    }
}

addname(name, value) :=
{
    n := (strlen(name) + 2) / 2
    symbspace + n + 2 >= currspace ? 0,
    {
```

```
        strcpy(#space + symbspace << 1, name)
        space[currspace] = symbspace
        space[currspace-1] = value
        currspace = currspace - 2
        symbspace = symbspace + n
        1
    }
}

findname(name, global) :=
{
    n := ?
    p := ?

    global ?
    {
        n = 16383
        p = space[n - 1]
        p == 0 ? p = currspace
    },
    {
        p = currspace
        n = thisspace
    }

    r := ?
    done := 0

    q := $
    n != 0 ?
    {
        r = $
        p < (n - 4) ?
        {
            strcmp(#space + space[p+2] << 1, name) == 0 ?
            {
                done = 1
            },
            {
                p = p + 2
                $ = r
            }
        }
        done == 0 ?
        {
            space[n - 2] == 0 ?
            {
                n = space[n]
                p = p + 4
                $ = q
            }
        }
    }
    done ? space[p + 1]
}

precedence(operator) :=
{
    operator == OPR_SHL || operator == OPR_SHR ? 6,
    operator == OPR_AND || operator == OPR_MUL ||
    operator == OPR_DIV || operator == OPR_MOD ? 5,
    operator == OPR_OR || operator == OPR_ADD ||
    operator == OPR_SUB ? 4,
    operator == OPR_LT || operator == OPR_GT ||
    operator == OPR_LTE || operator == OPR_GTE ||
    operator == OPR_EQU || operator == OPR_NEQ ? 3,
    operator == OPR_LND ? 2,
    operator == OPR_LOR ? 1,
    0
}

readchar(t) :=
{
    c := getc(src)

    t != 0xFFFF && c == t ? 0xFFFF,
    c == '\\' ?
    {
        c = getc(src)
        c == 't' ? 9,
        c == 'n' ? 10,
        c == 'r' ? 13,
        c == '0' ? 0,
        c == 'x' ?
        {
            n := 0

            p := $
            c = getc(src)

            '0' <= c && c <= '9' ?
            {
                n = 16 * n + (c - '0')
                $ = p
            }
            'a' <= c && c <= 'f' ?
            {
                n = 16 * n + (c - 'a' + 10)
                $ = p
            }
            'A' <= c && c <= 'F' ?
            {
                n = 16 * n + (c - 'A' + 10)
                $ = p
            }
            ungetc(src)
            n
        }, c
    }, c
}

get_token() :=
{
    c := getc(src)
    q := ?
    p := $

    c == ';' ?
    {
        q = $
        c != '\n' && c != '\r' && c != 0xFFFF ?
        {
            c = getc(src)
            $ = q
        }
        $ = p
    }
    isspace(c) ?
    {
        c == '\n' ? line = line + 1
        c == '\r' ?
        {
            c = getc(src)
            c != '\n' ? line = line + 1
        },
        c = getc(src)
        $ = p
    }

    c == 0xFFFF ? FILE_END,
    {
        position[0] = (position[1] = 0)
        seek(src, #position, 1)
        position[0] == 0 ? position[1] = position[1] - 1
        position[0] = position[0] - 1

        isalphabetic(c) ?
        {
            i := 0
            p := $
            i > 254 ?
            {
                error(3)
            },
            {
                @(#symbol+i) = @(#symbol+i) & 0xFF00 + c
                i = i + 1
                c = getc(src)
                isalphanumeric(c) ? $ = p
            }
            ungetc(src)
            @(#symbol+i) = @(#symbol+i) & 0xFF00
            SYMBOL
        },
        c == '\"' ?
        {
            ; Read string
            length = 0
            p := $
            length < 256 ?
            {
                c = readchar('\"')
                c != 0xFFFF ?
                {
                    @(#string+length) =
                        @(#string+length) & 0xFF00 + c
                    length = length + 1
```

```
                    $ = p
                }
            }
            c != 0xFFFF ?
            {
                error(4)
                p = $
                readchar('\"') != 0xFFFF ? $ = p
            }
            STRING
        },
        c == '\'' ?
        {
            number = readchar(0xFFFF)
            p := $
            readchar('\'') != 0xFFFF ? $ = p
            CHARACTER
        },
        isdigit(c) ?
        {
            number = 0
            d := getc(src)
            c == '0' && ((d == 'x') || (d == 'X')) ?
            {
                c = getc(src)
                b := 1
                p := $
                c >= '0' && c <= '9' ? c = c - '0',
                c >= 'A' && c <= 'F' ? c = c - 'A' + 10,
                c >= 'a' && c <= 'f' ? c = c - 'a' + 10,
                b = 0
                b ?
                {
                    number = number * 16 + c
                    c = getc(src)
                    $ = p
                }
            },
            {
                ungetc(src)
                p := $
                number = number * 10 + (c - '0')
                c = getc(src)
                isdigit(c) ? $ = p
            }
            ungetc(src)
            NUMBER
        },
        c == '&' ?
        {
            c = getc(src)
            c == '&' ? operator = OPR_LND,
            {
                ungetc(src)
                operator = OPR_AND
            }
            BINARY_OPERATOR
        },
        c == '*' ?
        {
            operator = OPR_MUL
            BINARY_OPERATOR
        },
        c == '/' ?
        {
            operator = OPR_DIV
            BINARY_OPERATOR
        },
        c == '|' ?
        {
            c = getc(src)
            c == '|' ? operator = OPR_LOR,
            {
                ungetc(src)
                operator = OPR_OR
            }
            BINARY_OPERATOR
        },
        c == '+' ?
        {
            operator = OPR_ADD
            OPERATOR
        },
        c == '-' ?
        {
            operator = OPR_SUB
            OPERATOR
        },
        c == '>' ?
        {
            c = getc(src)
            c == '=' ?
            {
                operator = OPR_GTE
                BINARY_OPERATOR
            },
            c == '>' ?
            {
                operator = OPR_SHR
                BINARY_OPERATOR
            },
            {
                ungetc(src)
                operator = OPR_GT
                BINARY_OPERATOR
            }
        },
        c == '<' ?
        {
            c = getc(src)
            c == '=' ?
            {
                operator = OPR_LTE
                BINARY_OPERATOR
            },
            c == '<' ?
            {
                operator = OPR_SHL
                BINARY_OPERATOR
            },
            {
                ungetc(src)
                operator = OPR_LT
                BINARY_OPERATOR
            }
        },
        c == '=' ?
        {
            c = getc(src)
            c == '=' ?
            {
                operator = OPR_EQU
                BINARY_OPERATOR
            },
            {
                ungetc(src)
                operator = OPR_SET
                BINARY_OPERATOR
            }
        },
        c == ':' ?
        {
            c = getc(src)
            c != '=' ? error(c, 5), ASSIGN_OPERATOR
        },
        c == '~' ?
        {
            operator = OPR_NOT
            UNARY_OPERATOR
        },
        c == '#' ?
        {
            operator = OPR_ADR
            UNARY_OPERATOR
        },
        c == '@' ?
        {
            operator = OPR_DRF
            UNARY_OPERATOR
        },
        c == '(' ? OPEN_PARENTHESIS,
        c == ')' ? CLOSE_PARENTHESIS,
        c == '[' ? OPEN_BRACKET,
        c == ']' ? CLOSE_BRACKET,
        c == '{' ? OPEN_BRACE,
        c == '}' ? CLOSE_BRACE,
        c == '?' ? QUESTION_MARK,
        c == ',' ? COMMA,
        c == '$' ? DOLLAR,
        c == '!' ?
        {
            c = getc(src)
            c == '=' ?
            {
                operator = OPR_NEQ
                BINARY_OPERATOR
            },
```

```
            {
                ungetc(src)
                operator = OPR_LNO
                UNARY_OPERATOR
            }
        },
        c == '%' ?
        {
            operator = OPR_MOD
            BINARY_OPERATOR
        }, error(c, 5)
    }
}

read_token() :=
{
    token = get_token()
}

set_position(pos) :=
{
    seek(src, pos, 0)
    position[0] = @pos[0]
    position[1] = @pos[1]
    line = @pos[2]
    read_token()
}

read_position(pos) :=
{
    @pos[0] = position[0]
    @pos[1] = position[1]
    @pos[2] = line
}

fixstack(depth) :=
{
    depth != 0 ?
    {
        depth < 0x80 || depth >= 0xFF80 ?
        {
            depth = depth & 0xFF
            writecode("\x83\xC4", 2)
            writecode(#depth, 1)
            ; 0100 83C400        ADD     SP,+00
        },
        {
            writecode("\x81\xC4", 2)
            writecode(#depth, 2)
            ; 0103 81C48000      ADD     SP,0080
        }
    }
}

loadvar(styp, sadr) :=
{
    styp == 0xFFFF ?
    {
        writecode("\xE8\x00\x00\x58\x05\x08\x00", 7)
        ; 0100 E80000        CALL    0103
        ; 0103 58            POP     AX
        ; 0104 050800        ADD     AX,0008
    },
    styp == 1 ?
    {
        sadr < 0x80 || sadr >= 0xFF80 ?
        {
            sadr = sadr & 0xFF
            writecode("\x8B\x46", 2)
            writecode(#sadr, 1)
        },
        {
            writecode("\x8B\x86", 2)
            writecode(#sadr, 2)
        }
        ; 0100 8B4600        MOV     AX,[BP+00]
        ; 0103 8B868000      MOV     AX,[BP+0080]
    },
    styp == 2 ?
    {
        writecode("\xA1", 1)
        writecode(#sadr, 2)
        ; 0100 A10000        MOV     AX,[0000]
    },
    styp == 3 ?
    {
        sadr = sadr - 1
        sadr < 0x80 || sadr >= 0xFF80 ?
        {
            sadr = sadr & 0xFF
            writecode("\x8B\x45", 2)
            writecode(#sadr, 1)
        },
        {
            writecode("\x8B\x85", 2)
            writecode(#sadr, 2)
        }
        ; 0100 8B4500        MOV     AX,[DI+00]
        ; 0103 8B858000      MOV     AX,[DI+0080]
    }
}

loadadr(styp, sadr) :=
{
    styp == 1 ?
    {
        sadr >= 0xFF80 || sadr < 0x80 ?
        {
            writecode("\x8D\x46", 2)
            writecode(#sadr, 1)
        },
        {
            writecode("\x8D\x86", 2)
            writecode(#sadr, 2)
        }
        ; 0100 8D4601        LEA     AX,[BP+01]
        ; 0103 8D868000      LEA     AX,[BP+0080]
    },
    styp == 2 ?
    {
        writecode("\xB8", 1)
        writecode(#sadr, 2)
        ; 0100 B80000        MOV     AX,0000
    },
    styp == 3 ?
    {
        sadr = sadr - 1
        sadr >= 0xFF80 || sadr < 0x80 ?
        {
            writecode("\x8D\x45", 2)
            writecode(#sadr, 1)
        },
        {
            writecode("\x8D\x85", 2)
            writecode(#sadr, 2)
        }
        ; 0107 8D4501        LEA     AX,[DI+01]
        ; 010A 8D858000      LEA     AX,[DI+0080]
    }
}

loadargs(protect_BX) :=
{
    d := 0

    p := $
    token != CLOSE_PARENTHESIS ?
    {
        d = d + 2
        protect_BX ? writecode("\x53", 1)
        ; 0100 53            PUSH    BX
        @primary_expression() ?
        {
            protect_BX ? writecode("\x5B\x50", 2),
                         writecode("\x50", 1)
            ; 0100 5B            POP     BX
            ; 0101 50            PUSH    AX
            token == COMMA ?
            {
                read_token()
                $ = p
            }
        },
        d = 0
    }
    d
}

unary_expression() :=
{
    uopr := 0
    sadr := 0
    styp := 0
    done := 0

    token == OPERATOR || token == UNARY_OPERATOR ?
    {
```

```
    uopr = operator
    read_token()
}

token == NUMBER || token == CHARACTER ?
{
    writecode("\xB8", 1)
    writecode(#number, 2)
    ; 0100 B80000          MOV     AX,0000
    read_token()
},
token == STRING ?
{
    heapadr >= 0xFF80 || heapadr < 0x80 ?
    {
        writecode("\x8D\x45", 2)
        writecode(#heapadr, 1)
    },
    {
        writecode("\x8D\x85", 2)
        writecode(#heapadr, 2)
    }
    ; 0107 8D4501          LEA     AX,[DI+01]
    ; 010A 8D858000        LEA     AX,[DI+0080]

    writeheap(#string, length)
    read_token()
},
token == OPEN_PARENTHESIS ?
{
    read_token()
    @primary_expression() == 0 ? done = 1,
    token != CLOSE_PARENTHESIS ? done = 1,
    read_token()
},
token == DOLLAR ?
{
    styp = 0xFFFF
    read_token()
},
token != SYMBOL ? done = 1,
{
    styp = styp + 1
    sadr = findname(#symbol, 0)
    sadr == 0 ?
    {
        styp = styp + 1
        sadr = findname(#symbol, 1)
    }
    sadr == 0 ?
    {
        styp = styp + 1
        symbol[0] = symbol[0] | 0x80
        sadr = findname(#symbol, 1)
    }
    sadr == 0 ?
    {
        symbol[0] = symbol[0] & 0xFF7F
        error(#symbol, 6)
        done = 1
    }
    done == 0 ? read_token()
}

done == 0 ? token == OPEN_PARENTHESIS ?
{
    d := 0

    styp == 0xFFFF ? done = 2,
    uopr == OPR_ADR ?
    {
        error(7)
        done = 1
    },
    (styp ? uopr != OPR_DRF) ?
    {
        read_token()
        d = loadargs(0)
        token != CLOSE_PARENTHESIS ? done = 1,
        {
            read_token()

            styp == 1 ?
            {
                sadr >= 0xFF80 || sadr < 0x80 ?
                {
                    writecode("\x8D\x5E", 2)
                    writecode(#sadr, 1)
                },
                {
                    writecode("\x8D\x9E", 2)
                    writecode(#sadr, 2)
                }
                ; 0100 8D5E01      LEA  BX,[BP+01]
                ; 0103 8D9E8000    LEA  BX,[BP+0080]

                writecode("\xFF\xD3", 2)
                ; 0106 FFD3            CALL    BX
            },
            styp == 2 ?
            {
                sadr = sadr - (address + 3)
                writecode("\x8E", 1)
                writecode(#sadr, 2)
                ; 0100 E80000          CALL    0103
            },
            styp == 3 ? ; heap
            {
                sadr = sadr - 1
                sadr >= 0xFF80 || sadr < 0x80 ?
                {
                    writecode("\x8D\x5D", 2)
                    writecode(#sadr, 1)
                },
                {
                    writecode("\x8D\x9D", 2)
                    writecode(#sadr, 2)
                }
                ; 0100 8D5D01      LEA  BX,[DI+01]
                ; 0103 8D9D8000    LEA  BX,[DI+0080]

                writecode("\xFF\xD3", 2)
                ; 0106 FFD3            CALL    BX
            }
        }
    },
    {
        uopr != OPR_DRF ?
        {
            error(8)
            done = 1
        },
        {
            loadvar(styp, sadr)

            writecode("\x89\xC3", 2)

            read_token()
            loadargs(1)
            token != CLOSE_PARENTHESIS ? done = 1,
            {
                read_token()
                writecode("\xFF\xD3", 2)
                ; 0100 89C3            MOV     BX,AX
                ; 0102 FFD3            CALL    BX
            }
            uopr = 0
        }
    }
    done == 0 ? fixstack(d)
},
token == OPEN_BRACKET ?
{
    read_token()
    styp == 0 ? writecode("\x50", 1)
    ; 0101 50              PUSH    AX

    @primary_expression() == 0 ? done = 1,
    token != CLOSE_BRACKET ? done = 1,
    {
        read_token()
        writecode("\xD1\xE0", 2)
        ; 0100 D1E0          SHL     AX,1
        styp ?
        {
            writecode("\x89\xC3", 2)
            ; 0100 89C3            MOV     BX,AX

            token == BINARY_OPERATOR &&
            operator == OPR_SET && uopr != OPR_DRF ?
            {
                loadadr(styp, sadr)
                writecode("\x01\xD8", 2)
                ; 0102 01D8            ADD     AX,BX
            },
            uopr == OPR_DRF ?
            {
                loadvar(styp, sadr)

                token == BINARY_OPERATOR &&
                operator == OPR_SET ?
                    writecode("\x01\xD8", 2),
                    ; 0102 01D8        ADD     AX,BX
                    writecode("\x01\xC3\x8B\x07", 4)
                    ; 0102 01C3        ADD     BX,AX
                    ; 0104 8B07        MOV     AX,[BX]
                uopr = 0
            },
            {
                loadadr(styp, sadr)
                writecode("\x01\xC3\x8B\x07", 4)
                ; 0102 01C3            ADD     BX,AX
                ; 0104 8B07            MOV     AX,[BX]
            }
        },
        {
            uopr != OPR_DRF ?
            {
                error(9)
                done = 1
            },
            {
                writecode("\x5B\x01\xC3\x8B\x07", 5)
                ; 0100 5B              POP     BX
                ; 0101 01C3            ADD     BX,AX
                ; 0103 8B07            MOV     AX,[BX]
            }
        }
    },
    token == BINARY_OPERATOR && operator == OPR_SET ?
    {
        uopr == OPR_DRF ?
        {
            styp ? loadvar(styp, sadr)
            uopr = 0
        },
        {
            styp == 0 ?
            {
                error(10)
                done = 1
            },
            styp == -1 ?
            {
                ; Special processing
                read_token()
                @primary_expression() ?
                    writecode("\xFF\xE0", 2), done = 1
                ; 0100 FFE0            JMP     AX
            }, loadadr(styp, sadr)
        }
    },
    styp ?
    {
        uopr == OPR_ADR ?
        {
            uopr = 0
            loadadr(styp, sadr)
        }, loadvar(styp, sadr)
    }

    done == 0 ? uopr == OPR_ADR ? done = 1,
    uopr == OPR_DRF ?
    {
        writecode("\x89\xC3\x8B\x07", 4)
        ; 0100 89C3            MOV     BX,AX
        ; 0102 8B07            MOV     AX,[BX]
    },
    uopr == OPR_SUB ?
    {
        writecode("\xF7\xD8", 2)
        ; 0100 F7D8            NEG     AX
    },
    uopr == OPR_NOT ?
    {
        writecode("\xF7\xD0", 2)
        ; 0100 F7D0            NOT     AX
    },
    uopr == OPR_LNO ?
    {
        writecode("\x09\xC0\x74\x03\xB8\x01\x00\x48", 8)
        ; 0100 09C0            OR      AX,AX
        ; 0102 7403            JZ      0107
        ; 0104 B80100          MOV     AX,0001
        ; 0107 48              DEC     AX
    }

    done - 1
}

execute(operator) :=
{
    operator == OPR_AND ?
    {
        writecode("\x5B\x21\xD8", 3)
        ; 0100 5B              POP     BX
        ; 0101 21D8            AND     AX,BX
    },
    operator == OPR_LND ?
    {

    writecode("\x5B\x09\xDB\x75\x02\x31\xC0\x09\xC0", 9)
        ; 0100 5B              POP     BX
        ; 0101 09DB            OR      BX,BX
        ; 0103 7502            JNZ     0107
        ; 0105 31C0            XOR     AX,AX
        ; 0107 09C0            OR      AX,AX
    },
    operator == OPR_MUL ?
    {
        writecode("\x5B\xF7\xE3", 3)
        ; 0100 5B              POP     BX
        ; 0101 F7E3            MUL     BX
    },
    operator == OPR_DIV ?
    {
        writecode("\x89\xC3\xBA\x00\x00\x58\xF7\xF3", 8)
        ; 0100 89C3            MOV     BX,AX
        ; 0102 BA0000          MOV     DX,0000
        ; 0105 58              POP     AX
        ; 0106 F7F3            DIV     BX
    },
    operator == OPR_OR || operator == OPR_LOR ?
    {
        writecode("\x5B\x09\xD8", 3)
        ; 0100 5B              POP     BX
        ; 0101 09D8            OR      AX,BX
    },
    operator == OPR_ADD ?
    {
        writecode("\x5B\x01\xD8", 3)
        ; 0100 5B              POP     BX
        ; 0101 01D8            ADD     AX,BX
    },
    operator == OPR_SUB ?
    {
        writecode("\x89\xC3\x58\x29\xD8", 5)
        ; 0100 89C3            MOV     BX,AX
        ; 0102 58              POP     AX
        ; 0103 29D8            SUB     AX,BX
    },
    operator == OPR_LT ?
    {

    writecode("\x5B\x39\xC3\xB8\x01\x00\x72\x02\x31\xC0",
                10)
        ; 0100 5B              POP     BX
        ; 0101 39C3            CMP     BX,AX
        ; 0103 B80100          MOV     AX,0001
        ; 0106 7202            JB      010A
        ; 0108 31C0            XOR     AX,AX
    },
    operator == OPR_GT ?
    {

    writecode("\x5B\x39\xC3\xB8\x01\x00\x77\x02\x31\xC0",
                10)
        ; 0100 5B              POP     BX
        ; 0101 39C3            CMP     BX,AX
        ; 0103 B80100          MOV     AX,0001
        ; 0106 7702            JA      010A
        ; 0108 31C0            XOR     AX,AX
    },
    operator == OPR_LTE ?
    {

    writecode("\x5B\x39\xC3\xB8\x01\x00\x76\x02\x31\xC0",
                10)
        ; 0100 5B              POP     BX
        ; 0101 39C3            CMP     BX,AX
        ; 0103 B80100          MOV     AX,0001
        ; 0106 7602            JBE     010A
        ; 0108 31C0            XOR     AX,AX
    },
    operator == OPR_GTE ?
    {

    writecode("\x5B\x39\xC3\xB8\x01\x00\x73\x02\x31\xC0",
                10)
        ; 0100 5B              POP     BX
        ; 0101 39C3            CMP     BX,AX
        ; 0103 B80100          MOV     AX,0001
        ; 0106 7302            JNB     010A
        ; 0108 31C0            XOR     AX,AX
    },
    operator == OPR_EQU ?
    {

    writecode("\x5B\x39\xC3\xB8\x01\x00\x74\x02\x31\xC0",
                10)
        ; 0100 5B              POP     BX
        ; 0101 39C3            CMP     BX,AX
        ; 0103 B80100          MOV     AX,0001
        ; 0106 7402            JZ      010A
        ; 0108 31C0            XOR     AX,AX
    },
    operator == OPR_NEQ ?
    {

    writecode("\x5B\x39\xC3\xB8\x01\x00\x75\x02\x31\xC0",
                10)
        ; 0100 5B              POP     BX
        ; 0101 39C3            CMP     BX,AX
        ; 0103 B80100          MOV     AX,0001
        ; 0106 7502            JNZ     010A
        ; 0108 31C0            XOR     AX,AX
    },
    operator == OPR_SET ?
    {
        ; In unary-expression(), we can always tell if
        ; the next unevaluated token is an OPR_SET
        ; operator. If it is, we can check if the
        ; current expression is an lvalue, and store the
        ; address instead of dereferencing.
        ; The NOP ensures fixed-length code that
        ; is required for $ to work.
        writecode("\x5B\x89\x07\x90", 4)
        ; 0100 5B              POP     BX
        ; 0101 8907            MOV     [BX],AX
        ; 0103 90              NOP
    },
    operator == OPR_SHL ?
    {
        writecode("\x89\xC1\x58\xD3\xE0", 5)
        ; 0100 89C1            MOV     CX,AX
        ; 0102 58              POP     AX
        ; 0103 D3E0            SHL     AX,CL
    },
    operator == OPR_SHR ?
    {

    writecode("\x89\xC1\x58\xD3\xE8", 5)
        ; 0100 89C1            MOV     CX,AX
        ; 0102 58              POP     AX
        ; 0103 D3E8            SHR     AX,CL
    },
    operator == OPR_MOD ?
    {

    writecode("\x89\xC3\xBA\x00\x00\x58\xF7\xF3\x89\xD0",
                10)
        ; 0100 89C3            MOV     BX,AX
        ; 0102 BA0000          MOV     DX,0000
        ; 0105 58              POP     AX
        ; 0106 F7F3            DIV     BX
        ; 0108 89D0            MOV     AX,DX
    },
    {
        error(17)
        writecode("\x83\xC4\x02", 3)
        ; 0100 83C402          ADD     SP,+02
    }
}

simple_expression() :=
{
    stkopr[7] := ?        ; Seven precedence levels
    stack := 0
    found := 0
    rval := 1

    token == NUMBER && number == 0 ?
    {
        pos[3] := ?
        read_position(#pos)
        read_token()
        token == BINARY_OPERATOR && operator == OPR_NEQ ?
                read_token(), set_position(#pos)
    }

    unary_expression() ?
    {
        p := $
        token == OPERATOR || token == BINARY_OPERATOR ?
        {
            {
                q := $
                0 != (stack > 0 ?
                    precedence(operator) <=
                        precedence(stkopr[stack - 1])) ?
                {
                    stack = stack - 1
                    execute(stkopr[stack])
                    $ = q
                }
            }
            writecode("\x50", 1)
            ; 0100 50              PUSH    AX
            stkopr[stack] = operator
            stack = stack + 1
            read_token()
            unary_expression() ? $ = p, rval = 0
        }
        p = $
        stack > 0 ?
        {
            stack = stack - 1
            execute(stkopr[stack])
            $ = p
        }
        rval
    }, 0
}

evaluate(value, operator) :=
{
    rval := 1

    operator == OPR_AND ? number = value & number,
    operator == OPR_MUL ? number = value * number,
    operator == OPR_DIV ? number = value / number,
    operator == OPR_OR ? number = value | number,
    operator == OPR_ADD ? number = value + number,
    operator == OPR_SUB ? number = value - number,
    operator == OPR_LT ? number = value < number,
    operator == OPR_GT ? number = value > number,
    operator == OPR_LTE ? number = value <= number,
    operator == OPR_GTE ? number = value >= number,
    operator == OPR_EQU ? number = value == number,
    operator == OPR_NEQ ? number = value != number,
    operator == OPR_LND ? number = value && number,
    operator == OPR_LOR ? number = value || number,
    operator == OPR_NOT ? number = ~number,
    operator == OPR_LNO ? number = !number,
    operator == OPR_SHL ? number = value << number,
    operator == OPR_SHR ? number = value >> number,
    operator == OPR_MOD ? number = value % number,
    {
        error(11)
        rval = 0
    }
    rval
}

constant_expression() :=
{
    stkval[7] := ?        ; Seven precedence levels
    stkopr[7] := ?
    stack := 0
    rval := 1

    token == OPERATOR || token == UNARY_OPERATOR ?
    {
        operator == OPR_ADR || operator == OPR_DRF ?
            rval = 0, stack > 5 ?
        {
            error(12)
            rval = 0
        },
        {
            stkopr[stack] = operator
            stkval[stack] = 0
            stack = stack + 1
            read_token()
        }
    }

    p := $
    rval ?
    {
        token == OPEN_PARENTHESIS ?
        {
            read_token()
            constant_expression() ?
            {
                token == CLOSE_PARENTHESIS ?
                    read_token(), rval = 0
            }, rval = 0
        },
        token == NUMBER || token == CHARACTER ?
                read_token(), rval = 0

        rval && ((token == OPERATOR) ||
                 (token == BINARY_OPERATOR)) ?
        {
            operator == OPR_SET ? rval = 0,
            {
                p := $

                stack > 0 && precedence(operator) <=
                    precedence(stkopr[stack - 1]) ?
                {
                    stack = stack - 1
                    evaluate(stkval[stack],stkopr[stack])
                    $ = p
                }
                stack > 5 ?
                {
                    error(12)
                    rval = 0
                },
                {
                    stkopr[stack] = operator
                    stkval[stack] = number
                    stack = stack + 1
                    read_token()
                }
            }
            $ = p
        }
    }

    rval ?
    {
        p = $
        stack > 0 ?
        {
            stack = stack - 1
            evaluate(stkval[stack], stkopr[stack])
        }
    }
    rval
}

compound_expression() :=
{
    token == OPEN_BRACE ?
    {
        read_token()
        ; Add new namespace
        oldbase := basespace
        newspace(0) ?
        {
            p := $
            token != CLOSE_BRACE ?
            {
                @primary_expression() ? $ = p
            },
            {
                read_token()
                oldbase != basespace ?
                {
                    namespace > 3 ?
                    {
                        d := oldbase - basespace
                        d < 0x80 || d >= 0xFF80 ?
                        {
                            writecode("\x83\xC4", 2)
                            writecode(#d, 1)
                        },
                        {
                            writecode("\x81\xC4", 2)
                            writecode(#d, 2);
                        }
                        ; 0100 83C400      ADD   SP,+00
                        ; 0103 81C48000    ADD   SP,0080
                    }
                    basespace = oldbase
                }
                oldspace(0)
            }
        }, 0
    }
}

_primary_expression() :=
{
    pos[3] := ?
    r := 0

    read_position(#pos)

    constant_expression() ?
    {
        function ?
        {
            ; Assignment
            writecode("\x8B", 1)
            writecode(#number, 2)
            ; 0100 B80000          MOV     AX,0000
        },
        {

            ; Add constant to heap
            writeheap(#number, 2)

        }
        r = 1
    },
    {
        set_position(#pos)
        token == OPEN_BRACE ? r = compound_expression(),
        {
            read_position(#pos)
            ; Here we have an ambiguity. If we read a
            ; symbol, we may have a declarator; we need
            ; to look for an ':', or we need to look
            ; for an '[ constant_expression ] :='
            ; sequence to determine what we have here.

            token == SYMBOL ?
            {
                read_token()
                token == OPEN_BRACKET ?
                {
                    read_token()
                    constant_expression() &&
                        token == CLOSE_BRACKET ?
                            read_token(), token = 0

                }
                token == ASSIGN_OPERATOR ?
                {
                    set_position(#pos)
                    r = @definition()
                },
                {
                    set_position(#pos)
                    r = simple_expression()
                }
            }, r = simple_expression()
        }
    }

    (r ? (token == QUESTION_MARK)) ?
    {
        f[2] := 0, 0
        a := ?
        writecode("\x09\xC0\x75\x03\xE9\x00\x00", 7)
        ; 0100 09C0            OR      AX,AX
        ; 0102 7503            JNZ     0107
        ; 0104 E90000          JMP     0107

        seek(com, #f, 1)
        f[0] < 2 ? f[1] = f[1] - 1
        f[0] = f[0] - 2
        a = address
        read_token()

        @primary_expression() ?
        {
            g[2] := 0, 0
            h[2] := 0, 0

            token == COMMA ?
            {
                writecode("\xE9\x00\x00", 3)
                ; 0100 E90000          JMP     0103
                seek(com, #h, 1)
                h[0] < 2 ? h[1] = h[1] - 1
                h[0] = h[0] - 2
            }

            seek(com, #g, 1)
            seek(com, #f, 0)
            a = address - a
            write(com, #a, 2)
            seek(com, #g, 0)
            a = address

            token == COMMA ?
            {
                read_token()
                r = @primary_expression()
            }

            h[0] || h[1] ?
            {
                g[0] = (g[1] = 0)
                seek(com, #g, 1)
                seek(com, #h, 0)
                a = address - a
                write(com, #a, 2)
                seek(com, #g, 0)
            }
        }, r = 0
    }
    r
}

declarator() :=
{
    token == SYMBOL ?
    {
        read_token()
        t := token
        t == OPEN_PARENTHESIS ?
        {
            function ? error(13),
            {
                addname(#symbol, address)

                ; Function preamble emitted here
                writecode("\x55\x89\xE5", 3)
                ; 0100 55              PUSH    BP
                ; 0101 89E5            MOV     BP,SP

                function = 1
                ; start new local namespace
                newspace(1)
                basespace = 0
                read_token()
                p := $
                token == SYMBOL ?
                {
                    addname(#symbol, basespace)
                    basespace = basespace - 2
                    read_token()
                    token == COMMA ?
                    {
                        read_token()
                        $ = p
                    }
                }
                {
                    ; Fix up newly added symbols
                    i := 0
                    basespace = 2 - basespace
                    p := $
                    i < basespace ?
                    {
                        space[currspace + i + 1] =
                                space[currspace + i + 1] +
                                    basespace
                        i = i + 2
                        $ = p
                    }
                }
                basespace = 0
                token == CLOSE_PARENTHESIS ?
                {
                    read_token()
                }, error(14)
            }
        },
        t == OPEN_BRACKET ?
        {
            read_token()
            constant_expression() ?
            {
                number = number * 2
                function ?
                {
                    basespace = basespace - number

                    number < 0x80 || number >= 0xFF80 ?
                    {
                        number = number & 0xFF
                        writecode("\x83\xEC", 2)
                        writecode(#number, 1)
                    },
                    {
                        writecode("\x81\xEC", 2)
                        writecode(#number, 2)
                    }
                    ; 0100 83EC02          SUB     SP,+02
                    ; 0103 81EC8000        SUB     SP,0080
                    addname(#symbol, basespace)
                },
                {
                    basespace = basespace + number
                    symbol[0] = symbol[0] | 0x80
                    addname(#symbol, heapadr + 1)
                }
                token == CLOSE_BRACKET ?
                {
                    read_token()
                    1
                }, 0
            }, error(15)
        },
        {

            ; add symbol to current namespace
            function ?
            {
                basespace = basespace - 2
                writecode("\x83\xEC\x02", 3)
                addname(#symbol, basespace)
                ; 0100 83EC02          SUB     SP,+02
            },
            {

                ; High bit0 indicates a heap symbol
                symbol[0] = symbol[0] | 0x80
                addname(#symbol, heapadr + 1)
            }
            1
        }
    }, 0
}

initializer() :=
{
    ; Whatever we do, at this point our lvalue is the
    ; last item in our current namespace.

    a := space[currspace + 1]
    rval := 1
    p := $

    token == QUESTION_MARK ?
    {
        ; We're happy. No code emitted.
        function == 0 ? writeheap("\x00\x00", 2)
        read_token();
    },
    @primary_expression() ?
    {
        (namespace == 2 ? function) ?
        {
            ; Function postamble emitted here
            writecode("\x89\xEC\x5D\xC3", 4)
            ; 0100 89EC            MOV     SP,BP
            ; 0102 5D              POP     BP
            ; 0103 C3              RET

            function = 0
            oldspace(1)
        },
        {
            function ?
            {
                ; Emit assignment code for local
                ; variable. Fixed-length code needed to
                ; ensure $ works properly.
                writecode("\x89\x86", 2)
                writecode(#a, 2)
                ; 010D 89868000        MOV   [BP+0080],AX
                a = a + 2
            }
        }
    }, rval = 0

    rval ?
    {
        token == COMMA ?
        {
            read_token()
            $ = p
        },
        {
            p = $
            heapadj > heapadr ?
            {
                writeheap("\x00\x00", 2);
                $ = p
            }
            1
        }
    }
}

_definition() :=
{
    declarator() ?
    {
        token == ASSIGN_OPERATOR ?
        {
            p := $
            read_token()
            initializer()
        }, 0
    }, 0
}

program() :=
{
    newspace(0)

    ; Startup code
    writecode("\xBF\x10\x01\xBB\x81\x00\x02\x1E\x80\x00\
xB8\x82\x00\x88\x27\x80\xFB\x81\x75\x02\x88\xD8\x50\xE8\x
05\x00\x83\xC4\x02\xCD\x20\xC3", 32)

    ; 0100 BF1001    MOV     DI,0110   ; Heap base at
                                      ; offset 1
    ; 0103 BB8100    MOV     BX,0081   ; Command line
    ; 0106 021E8000  ADD     BL,[0080]
    ; 010A B88200    MOV     AX,0082
    ; 010D 8827      MOV     [BX],AH
    ; 010F 80FB81    CMP     BL,81
    ; 0112 7502      JNZ     0116
    ; 0114 88D8      MOV     AL,BL
    ; 0116 50        PUSH    AX        ; as function
                                      ; parameter
    ; 0117 E80500    CALL    011F      ; start address at
                                      ; offset 24
    ; 011A 83C402    ADD     SP,+02    ; must be relative
                                      ; to 0x11A
    ; 011D CD20      INT     20
    ; 011F C3        RET

    read_token()
    p := $
    token != FILE_END ?
    {
        @definition() ? $ = p, error(16)
    }

    c[2] := 1, 0
    seek(com, #c, 0)
    write(com, #address, 2)

    a := findname("_\0", 1)
    a ?
    {
        a = a - 0x011A
        c[0] = 24
        c[1] = 0
        seek(com, #c, 0)
        write(com, #a, 2)
    }, error(0)
    0
}

_(arg) :=
{
    definition = #_definition
    primary_expression = #_primary_expression

    c[128] := ?

    n := strlen(arg)
    strcpy(#c, arg)

    at(#c, n - 2) != '.' || at(#c, n - 1) & 0xDF != 'W' ?
        strcpy(#c + n, ".W\0"), n = n - 2

    src = open(#c, 0);
    strcpy(#c + n, ".COM\0");
    com = open(#c, 1)
    strcpy(#c + n, ".TMP\0");
    tmp = open(#c, 1)

    src == 0 ?
        puts(stderr, "Can't open source file\r\n\0"),
    tmp == 0 ?
        puts(stderr, "Can't open temporary file\r\n\0"),
    com == 0 ?
        puts(stderr, "Can't open output file\r\n\0"),
    {
        program()
        close(src)
        close(tmp)
        tmp = open(#c, 0)
        tmp ?
        {
            b[512] := ?
            n := read(tmp, #b, 512)
            p[2] := 0, 0
            seek(com, #p, 2)
            p = $
            n > 0 ?
            {
                write(com, #b, n)
                n = read(tmp, #b, 512)
                $ = p
            }
            close(tmp)
        }
        unlink(#c)
        close(com)

        puts(stdout, "\r\nAddress map (global
symbols)\r\n==========================\r\n\0");
        n = space[16382]
        n == 0 ? n = currspace
        p := $
        n < 16379 ?
        {
            s := #space + space[n + 2] << 1
            h := @s[0] & 0x80
            @s[0] = @s[0] & 0xFF7F

            printf(s, (h?"heap\0","code\0"), space[n+1] +
                (h?address-1,0), "%x (%s) %s\r\n\0", stdout)

            n = n + 2
            $ = p
        }
    }
}
```