# Nuts and Bolts

[ [Go Back](#) ]

## Miscellaneous

Aljebrist applications distributed in byte code form are difficult to reverse engineer: keywords/identifiers are one/two byte values respectively, code tree has no embedded parentheses. Byte code files have a .ALJX extension. Much of the remainder of this web page consists of obsolete material.

## Public Fields

Public fields are Aljebrist fields (method variables) which are declared in a var block, whereas private (inner) fields are declared in an ivar block. Public fields which include getters and/or setters cannot be modified directly using an assignment statement, except within the class in which they are declared. For a field called myfield, the corresponding getter method is called get-myfield, and the corresponding setter method is called set-myfield. For a boolean field called myfield, the corresponding getter method is called is-myfield. For a boolean field called is-myfield, the corresponding getter method is called get-myfield.

## Aljebrist Parsing

Parser uses following sets of initial chars. (in parentheses or on separate line) to help determine class of tokens encountered.

- Alpha:
  - keyword  ( a-z )
  - built-in function  ( a-z )
  - system function*  ( _ )
- Identifiers:
  - local variable  ( A-Z, _ )
  - field  ( A-Z, _ )
  - method  ( A-Z, _ )
  - class  ( A-Z, _ )
- Numeric:
  - 0-9, -
- Punctuation:
  - (, ), {, }, #, ", $, ;
- Operators:
  - +, -, *, /, %, &, |, ^, ~, =, !, <, >, :, ?
- Invalid:
  - Literal Chars.  ( \, . )
  - Symbols  ( [, ], ', `, @, comma )

* System function names begin and end with 2 consecutive underscores. User-defined identifiers begin with optional single underscore followed by a letter, and may contain letters of both cases. The other 3 types of identifiers (keywords, built-in functions, system functions) contain lower case letters only.

Oddball characters:

- ( \ ) backslash found only in string literals
- ( . ) period found only in numeric literals
- ( - ) hyphen found at beginning of numeric literals and 3 operators: negate, subtract, -=
- ( } ) close brace in string literal must be escaped

## Lexical Scanner (Summary)

Each bottom-level category followed by (n), where n = count, category omitted if zero.

- ALPHA
  - KEYWORD
  - BLTINFUNC
  - SYSFUNC
  - IDENTIFIER
- NUMERIC
  - BINARY
  - OCTAL
  - HEXADECIMAL
  - DECIMAL
  - LONG
  - FLOAT
- PUNCT
  - OPENPAR
  - CLOSEPAR
  - SEMICOLON
  - CMTLINE
  - CMTBLK
  - STRLIT
  - OPERATOR
- INVALID
  - ERRSYM
  - ERRESC
  - ERRDOT
- Error messages:
  - Line no., description

## Lexical Scanner (Detail)

```
LN # TYP VAL CNV
==== === === ===
     XXX xxx xxx
0001 [ line buf one ]
     KWD str op
     FUN str
     SYS str
     ID  str
0002 [ line buf two ]
     BIN str dec
     OCT str dec
     HEX str dec
     DEC dec dec
     LNG dec dec
     FLT str val
0003 [ line buf three ]
     PAR (
     PAR )
     PAR ;
     CMT {
     CMT }
     CMT #
     STR str
     OP  str name
     ERR str desc
     [ omit blank lines ]
```

Each 4-digit line no. followed by contents of line in square brackets, followed by tokens, one per line. Global boolean: summary/detail

## Code Execution

All Aljebrist source code is in Polish notation, in which operators precede their operands. The following algorithm is used, in which operators are stored in one stack and operands in a separate stack. Executable code consists of tree nodes.

```
rightp = root
while true do
  if rightp = 0 then
    op = pop operator
    if op = root then
      return true
    if op = while/for/loopbody then
      pop rightp from operator stack
      continue
    if op = if then
      pop rightp from operator stack
      pop (
      continue
    if op = block then
      pop (
      pop if from operator stack
      pop (
      pop rightp from operator stack
      continue
    count = 0
    while true do
        pop operand
        if open parenthesis then break
        push operand on operator stack
        increment count
    if op = call then
      rightp = handlecall(count)
      continue
    if op = constructor then
      rightp = handlecons(count)
      continue
    if op = callback then
      rightp = handlecallback(count)
      continue
    pop operand from operator stack
    push operand
    repeat count - 1 times
        pop operand from operator stack
        push operand
        rightpop = pop
        leftpop = pop
        push op(leftpop, rightpop)
        // (: obj attridx) => obj...
    if count = 1 then
      if unary op then
        push op(pop)
      else
        rightpop = pop
        leftpop = pop
```

```
      push op(leftpop, rightpop)
   pop rightp from operator stack
   continue
currnode = getnode(rightp)
if open parenthesis then
   push on operand stack
   push rightp on operator stack
   rightp = currnode.downp
else if operand then
   push on operand stack
   rightp = currnode.rightp
else if operator then
   push on operator stack
   rightp = currnode.rightp
else if funcbody then
   handlebody
   rightp = currnode.rightp
else if endfunc then
   pop downto begin from operator stack
   pop rightp from operator stack
else if while/for then
   rightp = currnode.rightp
   push rightp, while/for on operator stack
else if do then
   flag = pop
   if not flag then
      pop while, rightp from operator stack
      pop rightp from operator stack
      pop (
else if continue then
   pop downto while from operator stack
   pop rightp from operator stack
else if break then
   pop downto while from operator stack
   pop rightp, rightp from operator stack
   pop (
else if breakfor then
   pop downto for from operator stack
   pop rightp, rightp from operator stack
   pop (
   pop (
else if contfor then
   pop downto loopbody from operator stack
   pop rightp from operator stack
else if then then
   flag = pop
   if flag then
      rightp = currnode.rightp
   else
      pop if from operator stack
      pop (
      pop rightp from operator stack
else
   return false
```

```
pop downto x from operator stack:    // handle pop-downto
  pop multiple from operator stack
  if: pop (
  while: pop (

do block while flag:    // handle do-while loop
  while true do block if not flag then break

handlecons(count):
  pop classref from operator stack
  gen objref: root 0/1 = instance/class vars
  push objref on operator stack
  return handlecall(count)

handlecall(count):
  pop objref from operator stack
  push objref
  pop codept from operator stack
  return handlecodept(codept, count)

handlecodept(codept, count):
  repeat count - 2 times
    pop val from operator stack
    push val
  push count - 1
  return codept

handlecallback(count):
  pop callback from operator stack
  unpack objref, codept
  push objref
  return handlecodept(codept, count)

handlebody:
  count = pop
  root = new node
  for i = count - 2 downto 0 do
    parm = pop
    add parm to 1st half of tree[i]
  objref = parm
  rightp = currnode.rightp
  loccount = currnode value
  repeat loccount times
    add null node to 2nd half of tree
  rightp = currnode.rightp
```